

Versal ACAP AI Engine Programming Environment

User Guide

UG1076 (v2021.2) December 17, 2021

Xilinx is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this [link](#) for more information.



Table of Contents

Chapter 1: Overview.....	7
Navigating Content by Design Process.....	10
Chapter 2: AI Engine Architecture Overview.....	11
AI Engine Tile Architecture.....	12
Tools.....	15
Documentation.....	16
Chapter 3: Introduction to AI Engine Programming.....	17
Prepare the Kernels.....	17
Creating a Data Flow Graph (Including Kernels).....	18
Recommended Project Directory Structure.....	22
Compiling and Running the Graph from the Command Line.....	23
Chapter 4: Window and Streaming Data API.....	25
Data Access Mechanisms.....	25
Window Operations for Kernels.....	28
Stream Operations for Kernels.....	40
Chapter 5: Run-Time Graph Control API.....	45
Graph Execution Control.....	45
Run-Time Parameter Specification.....	48
Run-Time Parameter Update/Read Mechanisms.....	52
Run-Time Graph Reconfiguration Using Control Parameters.....	56
Sharing Run-Time Parameters Across Multiple Kernels.....	59
Run-Time Parameter Support Summary.....	60
Chapter 6: Specialized Graph Constructs.....	62
Look-up Tables.....	62
FIFO Depth.....	64
Kernel Bypass.....	67
Explicit Packet Switching.....	68

Location Constraints.....	72
Buffer Allocation Control.....	80
C++ Kernel Class Support.....	81
C++ Template Support.....	86
Multicast Support.....	89
Chapter 7: AI Engine/Programmable Logic Integration.....	91
Design Flow Using RTL Programmable Logic.....	91
Design Considerations for Graphs Interacting with Programmable Logic.....	93
AI Engine-PL Interface Performance.....	96
Chapter 8: Using a Virtual Platform.....	98
Virtual Platform.....	98
FileIO Attributes.....	99
PLIO Attributes.....	99
GMIO Attributes.....	101
Performance Comparison Between AI Engine/PL and AI Engine/NoC Interfaces.....	106
Enhanced Programming Model.....	109
Chapter 9: Compiling an AI Engine Graph Application.....	113
Setting Up the Vitis Tool Environment.....	113
Inputs.....	114
Outputs.....	115
AI Engine Compiler Options.....	117
Mapper and Router Options.....	121
Viewing Compilation Results in the Vitis Analyzer.....	122
AI Engine Compiler Guidance.....	130
Chapter 10: Simulating an AI Engine Graph Application.....	131
x86 Functional Simulator.....	134
Software Emulation.....	150
AI Engine SystemC Simulator.....	151
Hardware Emulation.....	155
Chapter 11: Performance Analysis of AI Engine Graph	
Application during Simulation.....	158
AI Engine Simulation-Based Performance Analysis.....	158
Viewing the Run Summary in the Vitis Analyzer.....	164
Trace View Data Visualization.....	168

AI Engine Stall Analysis in Vitis Analyzer.....	179
Using Trace Compass to Visualize AI Engine Traces.....	196
Chapter 12: Performance Analysis of AI Engine Graph	
Application on Hardware.....	201
Profiling the AI Engine.....	201
Event Tracing in Hardware.....	221
Chapter 13: Programming the PS Host Application.....	230
Host Programming on Linux.....	231
Host Programming for Bare-Metal Systems.....	263
Chapter 14: Integrating the Application Using the Vitis Tools	
Flow.....	268
Platforms.....	270
PL Kernels.....	271
Linking the System.....	274
Compile the Embedded Application for the Cortex-A72 Processor.....	277
Packaging.....	278
Building a Bare-metal System.....	282
Running the System in Hardware.....	284
Running Software Emulation.....	284
Running Hardware Emulation.....	286
Generating Traffic for Emulation.....	287
Deploying the System.....	290
Chapter 15: Using the Vitis IDE.....	291
Creating the AI Engine Graph Project and Top-Level System Project.....	291
Single Kernel Development.....	304
Adding a PL Kernel Project to the System	304
Configuring the HW-Link Project.....	308
Adding a PS Application to the System.....	310
Building and Running the System.....	315
Building a Bare-metal AI Engine in the Vitis IDE.....	317
Programming Device and Flash Memory.....	327
Chapter 16: Debugging the AI Engine Application.....	329
Launching Debug from the Vitis IDE.....	329
Launching Debug from the Command Line.....	344

Using the Debug Environment.....	354
Pipeline View for Single Kernel Debug.....	366
Chapter 17: Mapper/Router Methodology.....	368
Design Convergence.....	368
Improving Design Performance.....	375
Appendix A: Adaptive Data Flow Graph Specification Reference....	378
Return Code.....	378
Graph Objects.....	379
Platform Objects.....	384
Event API.....	388
Connections.....	390
Constraints.....	391
Appendix B: Event Trace Reference.....	414
Simulation Event Trace	414
Hardware Event Trace.....	416
Appendix C: Using the Restrict Keyword in AI Engine Kernels.....	418
Pointer Aliasing.....	418
Strict Aliasing Rule.....	420
Restrict Keyword.....	422
Restrict Qualification.....	423
Undefined Behavior.....	424
Scope of Restrict Keyword in Inline Function.....	427
Benefits of Using the Restrict Keyword for Read/Modify/Write Loops.....	428
Derived Pointers.....	430
Summary.....	430
Appendix D: Non-Templated Versions of Window and Stream APIs.....	432
Window Operations for Kernels.....	432
Stream Operations for Kernels.....	444
Appendix E: Additional Resources and Legal Notices.....	447
Xilinx Resources.....	447
Documentation Navigator and Design Hubs.....	447
References.....	448

Revision History.....	448
Please Read: Important Legal Notices.....	451

Overview

Versal[®] adaptive compute acceleration platforms (ACAPs) combine Scalar Engines, Adaptable Engines, and Intelligent Engines with leading-edge memory and interfacing technologies to deliver powerful heterogeneous acceleration for any application. Most importantly, Versal ACAP hardware and software are targeted for programming and optimization by data scientists and software and hardware developers. Versal ACAPs are enabled by a host of tools, software, libraries, IP, middleware, and frameworks to enable all industry-standard design flows.

Built on the TSMC 7 nm FinFET process technology, the Versal portfolio is the first platform to combine software programmability and domain-specific hardware acceleration with the adaptability necessary to meet today's rapid pace of innovation. The portfolio includes six series of devices uniquely architected to deliver scalability and AI inference capabilities for a host of applications across different markets—from cloud—to networking—to wireless communications—to edge computing and endpoints.

The Versal architecture combines different engine types with a wealth of connectivity and communication capability and a network on chip (NoC) to enable seamless memory-mapped access to the full height and width of the device. Intelligent Engines are SIMD VLIW AI Engines for adaptive inference and advanced signal processing compute, and DSP Engines for fixed point, floating point, and complex MAC operations. Adaptable Engines are a combination of programmable logic blocks and memory, architected for high-compute density. Scalar Engines, including Arm[®] Cortex[®]-A72 and Cortex-R5F processors, allow for intensive compute tasks.

AI Engines

The Versal AI Core series delivers breakthrough AI inference acceleration with AI Engines that deliver over 100x greater compute performance than current server-class of CPUs. This series is designed for a breadth of applications, including cloud for dynamic workloads and network for massive bandwidth, all while delivering advanced safety and security features. AI and data scientists, as well as software and hardware developers, can all take advantage of the high compute density to accelerate the performance of any application. Given the AI Engine's advanced signal processing compute capability, it is well-suited for highly optimized wireless applications such as radio, 5G, backhaul, and other high-performance DSP applications.

AI Engines are an array of very-long instruction word (VLIW) processors with single instruction multiple data (SIMD) vector units that are highly optimized for compute-intensive applications, specifically digital signal processing (DSP), 5G wireless applications, and artificial intelligence (AI) technology such as machine learning (ML).

AI Engines are hardened blocks that provide multiple levels of parallelism including instruction-level and data-level parallelism. Instruction-level parallelism includes a scalar operation, up to two moves, two vector reads (loads), one vector write (store), and one vector instruction that can be executed—in total, a 7-way VLIW instruction per clock cycle. Data-level parallelism is achieved via vector-level operations where multiple sets of data can be operated on a per-clock-cycle basis. Each AI Engine contains both a vector and scalar processor, dedicated program memory, local 32 KB data memory, access to local memory in any of three neighboring directions. It also has access to DMA engines and AXI4 interconnect switches to communicate via streams to other AI Engines or to the programmable logic (PL) or the DMA. Refer to the *Versal ACAP AI Engine Architecture Manual* ([AM009](#)) for specific details on the AI Engine array and interfaces.

AI Engine Kernels

An AI Engine kernel is a C/C++ program which is written using specialized intrinsic calls that target the VLIW vector processor. The AI Engine kernel code is compiled using the AI Engine compiler (`aiecompiler`) that is included in the Vitis™ core development kit. The AI Engine compiler compiles the kernels to produce an ELF file that is run on the AI Engine processors. [Chapter 2: AI Engine Architecture Overview](#) presents a high-level overview of kernel programming, tools, and documents that can be referenced for AI Engine kernel programming.

AI Engine Graphs

An AI Engine program consists of a data flow graph specification which is written in C++. This specification can be compiled and executed using the AI Engine compiler. An adaptive data flow (ADF) graph application consists of nodes and edges where nodes represent compute kernel functions, and edges represent data connections. Kernels in the application can be compiled to run on the AI Engines or in the PL region of the device. [Chapter 3: Introduction to AI Engine Programming](#) presents a brief overview of the AI Engine programming model, introduction to ADF graphs, and compiling and simulating an AI Engine graph.

Controlling the AI Engine Graph

[Chapter 5: Run-Time Graph Control API](#) describes the various control APIs available to control and update the AI Engine graphs at run time. The graph control APIs can be used to initialize, run, update, and control the graph execution from an external controller and runs in the context of a platform. This platform can be a simulation-only platform, an extensible target platform which can be connected to the PL kernels, or a fixed platform for bare-metal applications.

The external controller can be the host code running on one of the processors in the embedded processing system (PS). [Chapter 13: Programming the PS Host Application](#) describes the process of creating a host application to control the graph and PL kernels of the system. When your design is deployed in hardware, you can install drivers that facilitate initializing and controlling the graph execution via a host application running on the PS, or load and run the AI Engine graph at device boot time.

Application-specific AI Engine control code is generated by the AI Engine compiler as part of compiling the AI Engine design graph and kernel code. The AI Engine control code can:

- Control the initial loading of the AI Engine kernels.
- Run the graph for several iterations, update the run-time parameters (RTP) associated with the graph, exit and reset the AI Engines.

The Vitis core development kit provides the `xilinx_vck190_base_202120_1` platform for building, simulating, debugging, and deploying your AI Engine designs. The `xilinx_vck190_base_202120_1` is a platform targeting the VCK190 board. It enables development of a design including AI Engine and PL kernels with a host application that targets the Linux OS running on the Arm processor in the PS. Designs developed on this platform can be verified using the hardware emulation flow. These designs can also run on the VCK190 board.

Writing an Example AI Engine Design

[Chapter 3: Introduction to AI Engine Programming](#) walks you through the steps involved in creating, compiling and simulating an AI Engine example using the Vitis tools.

The next few chapters describe the APIs that are available for data communication between kernels, controlling and updating graphs at run time, graph constructs to constrain the graph based on your design requirements, and graph constructs to interact with the rest of the Versal architecture areas, the scalar engine and adaptable engine.

Compiling and Simulating the Program

[Chapter 9: Compiling an AI Engine Graph Application](#) describes in detail the different types of compilation available with the AI Engine compiler, the options and input files that can be passed in, and the expected output. You can compile the graph and kernels independently, or as part of a larger system, and set up the design to capture and profile event trace data at run time.

[Chapter 10: Simulating an AI Engine Graph Application](#) describes the AI Engine simulator in detail, as well as the x86 simulator for functional simulation. The AI Engine simulator simulates the graph application as a standalone entity, or as part of the hardware emulation of a larger system design.

Using the AI Engine Graph as Part of a Versal ACAP System Design

The AI Engine kernels and graph developed in the previous steps can be used as part of a larger Versal ACAP system design that can consist of AI Engine kernels, HLS PL kernels, RTL kernels, and the host application. The Vitis compiler builds this larger system.

As described in [Chapter 14: Integrating the Application Using the Vitis Tools Flow](#), you can use a command-line approach for building the system, or use the a GUI based approach as described in [Chapter 15: Using the Vitis IDE](#). Either approach lets you perform simulation or emulation to verify the design, debug the design in an interactive debug environment, and build the design to deploy on hardware.

Chapter 11: Performance Analysis of AI Engine Graph Application during Simulation describes how to extract performance data by performing event tracing when running the hardware emulation build or the hardware build. Chapter 16: Debugging the AI Engine Application shows how to run and use the debug environment from the command line, or from the Vitis IDE. The evaluation of the system performance and debugging the application are the key steps to achieve the application objectives.

Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal® ACAP design process [Design Hubs](#) and the [Design Flow Assistant](#) materials can be found on the [Xilinx.com](https://www.xilinx.com) website. This document covers the following design processes:

- **System and Solution Planning:** Identifying the components, performance, I/O, and data transfer requirements at a system level. Includes application mapping for the solution to PS, PL, and AI Engine. Topics in this document that apply to this design process include:
 - [Chapter 7: AI Engine/Programmable Logic Integration](#)
 - [Chapter 13: Programming the PS Host Application](#)
 - [Chapter 14: Integrating the Application Using the Vitis Tools Flow](#)
- **Embedded Software Development:** Creating the software platform from the hardware platform and developing the application code using the embedded CPU. Also covers XRT and Graph APIs. Topics in this document that apply to this design process include:
 - [Chapter 13: Programming the PS Host Application](#)
- **AI Engine Development:** Creating the AI Engine graph and kernels, library use, simulation debugging and profiling, and algorithm development. Also includes the integration of the PL and AI Engine kernels.
- **System Integration and Validation:** Integrating and validating the system functional performance, including timing, resource use, and power closure. Topics in this document that apply to this design process include:
 - [Chapter 14: Integrating the Application Using the Vitis Tools Flow](#)
 - [Chapter 15: Using the Vitis IDE](#)
 - [Chapter 16: Debugging the AI Engine Application](#)

AI Engine Architecture Overview

Programming the AI Engine array requires a thorough understanding of the algorithm to be implemented, the capabilities of the AI Engines, and the overall data flow between individual functional units. The AI Engine array supports three levels of parallelism:

- **SIMD:** Through vector registers that allow multiple elements to be computed in parallel.
- **Instruction level:** Through the VLIW architecture that allows multiple instructions to be executed in a single clock cycle.
- **Multicore:** Through the AI Engine array, where up to 400 AI Engines can execute in parallel.

While most standard C code can be compiled for the AI Engine, the code might need substantial restructuring to achieve optimal performance on the AI Engine array. The power of an AI Engine is its ability to execute a vector MAC operation, load two 256-bit vectors for the next operation, store a 256-bit vector from the previous operation, and increment a pointer or execute another scalar operation in each clock cycle. The AI Engine compiler does not perform any auto or pragma-based vectorization. The code must be rewritten to use SIMD intrinsic data types (for example, `v8int32`) and vector intrinsic functions (for example, `mac(...)`), and these must be executed within a pipelined loop to achieve the optimal performance. The 32-bit scalar RISC processor has an ALU, some non-linear functions, and data type conversions. Each AI Engine has access to a limited amount of memory, this means that large data sets need to be partitioned.

AI Engine kernels are functions that run on an AI Engine, and form the fundamental building blocks of a data flow graph specification. The data flow graph is a Kahn process network with deterministic behavior that does not depend on the various computational or communication delays. AI Engine kernels are declared as void C/C++ functions that take window or stream arguments for graph connectivity. Kernels can also have static data and run-time parameter arguments that can be either asynchronous or triggering. Each kernel should be defined in its own source file.

To achieve overall system performance, additional reading and experience is required with respect to the architecture, partitioning, as well as with the AI Engine data flow graph generation and optimizing data flow connectivity. The *Versal ACAP AI Engine Architecture Manual* ([AM009](#)) contains more detailed information.

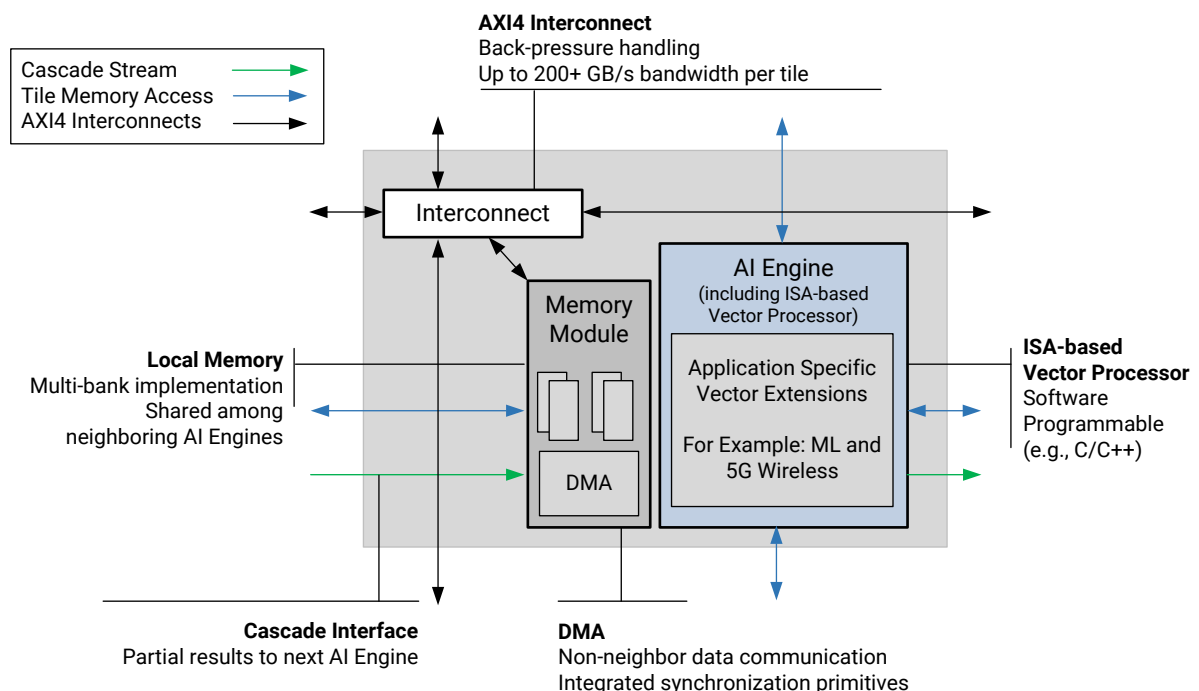
Xilinx provides DSP and communications libraries with optimized code for the AI Engine that should be used whenever possible. The supplied source code is also a great resource for learning about AI Engine kernel coding.

AI Engine Tile Architecture

The AI Engine array consists of a 2D array of AI Engine tiles, where each AI Engine tile contains an AI Engine, memory module, and tile interconnect module. An overview of such an AI Engine tile is shown in the following figure.

- **AI Engine:** Each AI Engine is a very long instruction word (VLIW) processor containing a scalar unit, a vector unit, two load units, and a single store unit.
- **AI Engine Tile:** An AI Engine tile contains an AI Engine, a local memory module together with several communication paths to facilitate data exchange between tiles.
- **AI Engine Array:** AI Engine array refers to the complete 2D array of AI Engine tiles.
- **AI Engine Program:** The AI Engine program consists of a data flow graph specification which is written in C/C++. This program is compiled and executed using the AI Engine tool chain.
- **AI Engine Kernels:** Kernels are written in C/C++ using AI Engine vector data types and intrinsic functions. These are the computation functions running on an AI Engine. The kernels form the fundamental building blocks of a data flow graph specification.

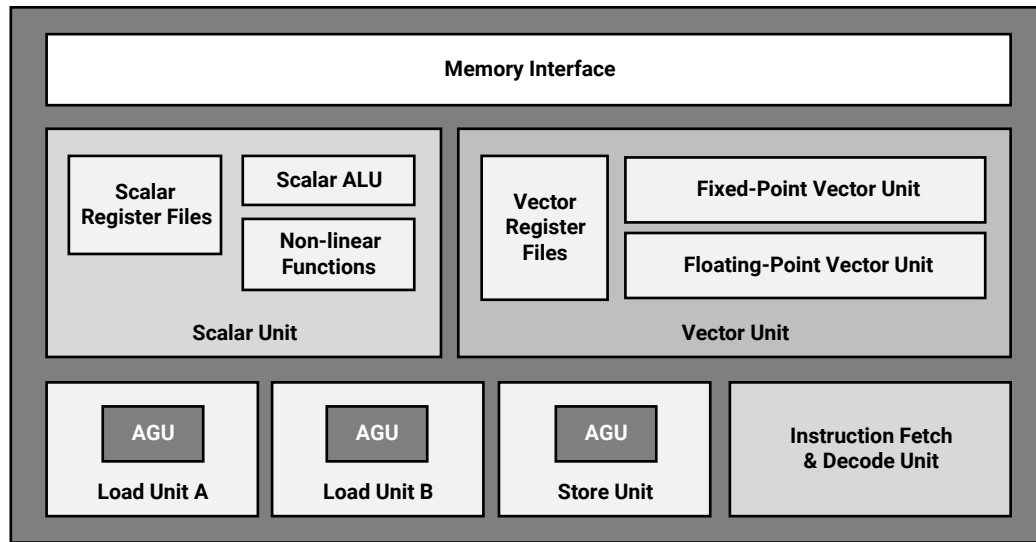
Figure 1: AI Engine Tile Block Diagram



X21602-091321

The following illustration is the architecture of a single AI Engine.

Figure 2: AI Engine



X20821-051618

Each AI Engine is a very long instruction word (VLIW) processor containing a scalar unit, a vector unit, two load units, and one store unit. The main compute power is provided by the vector unit. The vector unit contains a fixed-point unit with 128 8-bit fixed-point multipliers and a floating-point unit with eight single-precision floating-point multipliers. The vector registers and permute network are shared between the floating-point and fixed-point vector units. The peak performance depends on the size of the data types used by the operands. The following table provides the number of MAC operations that can be performed by the vector processor per instruction.

Table 1: Supported Precision Bit Width of the Vector Datapath

X Operand	Z Operand	Output	Number of MACs
8 real	8 real	48 real	128
16 real	8 real	48 real	64
16 real	16 real	48 real	32
16 real	16 complex	48 complex	16
16 complex	16 real	48 complex	16
16 complex	16 complex	48 complex	8
16 real	32 real	48/80 real	16
16 real	32 complex	48/80 complex	8
16 complex	32 real	48/80 complex	8
16 complex	32 complex	48/80 complex	4
32 real	16 real	48/80 real	16
32 real	16 complex	48/80 complex	8
32 complex	16 real	48/80 complex	8

Table 1: Supported Precision Bit Width of the Vector Datapath (cont'd)

X Operand	Z Operand	Output	Number of MACs
32 complex	16 complex	48/80 complex	4
32 real	32 real	80 real	8
32 real	32 complex	80 complex	4
32 complex	32 real	80 complex	4
32 complex	32 complex	80 complex	2
32 SPFP	32 SPFP	32 SPFP	8

To calculate the maximum performance for a given datapath, it is necessary to multiply the number of MACs per instruction with the clock frequency of the AI Engine kernel. For example, with 16-bit input vectors X and Z, the vector processor can achieve 32 MACs per instruction. Using the clock frequency for the slowest speed grade results in:

$$32 \text{ MACs} * 1 \text{ GHz clock frequency} = 32 \text{ Giga MAC operations/second}$$

In most cases, 32 MACs/instruction remains a theoretical upper bound because the algorithm to be implemented cannot continuously use the full capabilities of the AI Engine or might be constrained by I/O bandwidth.

The main I/O interfaces with respect to reading and writing data to and from the AI Engine for compute are the data memory interfaces, the stream interfaces, and the cascade stream interfaces. A complete list of interfaces including the program memory interface and debug interface are available in *Versal ACAP AI Engine Architecture Manual* ([AM009](#)).



RECOMMENDED: Xilinx highly recommends reading *Versal ACAP AI Engine Architecture Manual* ([AM009](#)) prior to starting your AI Engine kernel programming.

- The data memory interface sees one contiguous memory consisting of the data memory modules in all four directions with a total capacity of 128 KB. The AI Engine has two 256-bit wide-load units and one 256-bit wide-store unit.
- The AI Engine has two 32-bit input AXI4-Stream interfaces and two 32-bit output AXI4-Stream interfaces. Each of these streams allow the AI Engine to have a 128-bit access every four clock cycles or a 32-bit wide access per cycle.
- The 384-bit accumulator data from one AI Engine can be forwarded to the neighboring AI Engine by using the cascade stream interfaces to form a chain. The cascade stream interface is uni-directional and its direction depends on the row where the AI Engine is located. There is a small, two deep, 384-bit wide FIFO on both the input and output streams that allow storing up to four values between AI Engines. Each cycle 384-bits can be sent and received by the chained AI Engines.

The program memory size on the AI Engine is 16 KB, which allows storing 1024 instructions of 128-bit each. The AI Engine instructions are 128-bits wide and support multiple instruction formats and variable length instructions to reduce the program memory size. Many instructions outside of the optimized inner loop can use the shorter formats.

Tools

Vitis Integrated Design Environment

The Vitis™ integrated design environment (IDE) can be used to target system programming of Xilinx® devices including, Versal® devices with multiple AI Engine kernels. The following features are available in the tool.

- An optimizing C/C++ compiler that compiles the kernels and graph code making all of the necessary connections, placements, and checks to ensure proper functioning on the device.
- A cycle approximate simulator, accelerated functional simulator, and profiling tools.
- A debugging environment that works in both simulation and hardware environments.

Vitis Command Line Tools

Command line tools are available to build, simulate, and generate output files and reports. Command line outputs which are generated by the IDE are captured to facilitate subsequent integration into customer build environments. The Vitis analyzer IDE is available for report viewing and analysis of the output files and reports generated by the command line tools.

Vitis Model Composer

Vitis™ Model Composer offers a high-level graphical entry environment based on MATLAB® and Simulink® for simulation and code generation of designs that includes AI Engine, HLS, and RTL components. For more information, see the *Vitis Model Composer User Guide* ([UG1483](#)).

- Import AI Engine kernels, graphs, HLS kernels, and RTL based blocks into one Simulink® design for fast co-simulation.
- From the Simulink library browser, drag and drop optimized AI Engine functions such as Finite Impulse Response (FIR) and FFT filters into the design.
- Verify the design using stimulus generated in MATLAB or Simulink, visualize the results, and compare the results with golden reference. Generate graph code and test vectors.
- Assemble imported and block library code to feed into downstream tools.

Documentation

The following links are useful in developing and programming your AI Engine kernels.

- The *Versal ACAP AI Engine Intrinsic Documentation* ([UG1078](#)) is a list of all the intrinsic APIs and data types supported in the current release. It is a good reference guide for all the intrinsic APIs and data types supported by the AI Engine.
- *Versal ACAP AI Engine Architecture Manual* ([AM009](#))
- The *Chess Compiler User Manual* has a list of all the pragmas and functions to help optimize your AI Engine kernel code. It can be found in the AI Engine lounge.
- *Versal ACAP AI Engine Register Reference* ([AM015](#))
- *Vitis Model Composer User Guide* ([UG1483](#))

Introduction to AI Engine Programming

An AI Engine program consists of a *Data Flow Graph Specification* written in C++. As described in [C++ Template Support](#) you can use template classes or functions for writing the AI Engine graph or kernels. The application can be compiled and executed using the AI Engine tool chain. This chapter provides an introduction to writing an AI Engine program.

A complete class reference guide is shown in [Appendix A: Adaptive Data Flow Graph Specification Reference](#). The example that is used in this chapter can be found as a template example in the Vitis™ environment when creating a new AI Engine project.

Prepare the Kernels

Kernels are computation functions that form the fundamental building blocks of the data flow graph specifications. Kernels are declared as ordinary C/C++ functions that return `void` and can use special data types as arguments (discussed in [Chapter 4: Window and Streaming Data API](#)). Each kernel must be defined in its own source file. This organization is recommended for reusability and faster compilation. Furthermore, the kernel source files should include all relevant header files to allow for independent compilation.

Note: For the AI Engine kernel to use AI Engine API, include the following files in the kernel source code:

- `#include "aie_api/aie.hpp"`
- `#include "aie_api/aie_adf.hpp"`

It is recommended that a header file (`kernels.h` in this documentation) should declare the function prototypes for all kernels used in a graph. An example is as follows.

```
#ifndef FUNCTION_KERNELS_H
#define FUNCTION_KERNELS_H

void simple(input_window<cint16> * in, output_window<cint16> * out);

#endif
```

In the example, the `#ifndef` and `#endif` are present to ensure that the include file is only included once, which is good C/C++ practice.

Creating a Data Flow Graph (Including Kernels)

This following process describes how to construct data flow graphs in C++.

1. Define your application graph class in a separate header file (for example `project.h`). First, add the Adaptive Data Flow (ADF) header (`adf.h`) and include the kernel function prototypes. The ADF library includes all the required constructs for defining and executing the graphs on AI Engines.

```
#include <adf.h>
#include "kernels.h"
```

2. Define your graph class by using the objects which are defined in the `adf` name space. All user graphs are derived from the class `graph`.

```
include <adf.h>
#include "kernels.h"

using namespace adf;

class simpleGraph : public graph {
private:
    kernel first;
    kernel second;
};
```

This is the beginning of a graph class definition that declares two kernels (`first` and `second`).

3. Add some top-level ports to the graph.

```
#include <adf.h>
#include "kernels.h"

using namespace adf;

class simpleGraph : public graph {
private:
    kernel first;
    kernel second;
public:
    input_port in;
    output_port out;
};
```

4. Use the `kernel::create` function to instantiate the `first` and `second` C++ kernel objects using the functionality of the C function `simple`.

```
#include <adf.h>
#include "kernels.h"

using namespace adf;
```

```
class simpleGraph : public graph {
private:
    kernel first;
    kernel second;
public:
    input_port in;
    output_port out;
    simpleGraph() {
        first = kernel::create(simple);
        second = kernel::create(simple);
    }
};
```

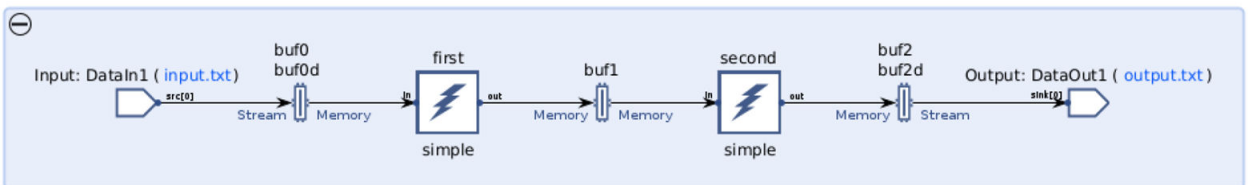
5. Add the connectivity information, which is equivalent to the nets in a data flow graph. In this description, ports are referenced by indices. The first input window or stream argument in the `simple` function is assigned index 0 in an array of input ports (`in`). Subsequent input arguments take ascending consecutive indices. The first output window or stream argument in the `simple` function is assigned index 0 in an array of output ports (`out`). Subsequent output arguments take ascending consecutive indices.

```
#include <adf.h>
#include "kernels.h"

using namespace adf;

class simpleGraph : public graph {
private:
    kernel first;
    kernel second;
public:
    input_port in;
    output_port out;

    simpleGraph() {
        first = kernel::create(simple);
        second = kernel::create(simple);
        connect< window<128> > net0 (in, first.in[0]);
        connect< window<128> > net1 (first.out[0], second.in[0]);
        connect< window<128> > net2 (second.out[0], out);
    }
};
```



This figure represents the graph connectivity specified in the previous graph code. Graph connectivity can be viewed when you open the compilation results in the Vitis analyzer. For more information, see [Viewing Compilation Results in the Vitis Analyzer](#). As shown in the previous figure, the input port from the top level is connected into the input port of the first kernel, the output port of the first kernel is connected to the input port of the second kernel, and the output port of the second kernel is connected to the output exposed to the top level. The first kernel executes when 128 bytes of data (32 complex samples) are collected in a

buffer from an external source. This is specified as a window parameter at the connection `net0`. Likewise, the second kernel executes when its input window has valid data being produced as the output of the first kernel expressed via connection `net1`. Finally, the output of the second kernel is connected to the top level output port as connection `net2`, specifying that upon termination the second kernel will produce 128 bytes of data.

6. Set the source file and tile usage for each of the kernels. The source file `kernel.cc` contains kernel first and kernel second source code. Then the ratio of the function run time compared to the cycle budget, known as the run-time ratio, and must be between 0 and 1. The cycle budget is the number of instruction cycles a function can take to either consume data from its input (when dealing with a rate limited input data stream), or to produce a block of data on its output (when dealing with a rate limited output data stream). This cycle budget can be affected by changing the block sizes.

```
#include <adf.h>
#include "kernels.h"

using namespace adf;

class simpleGraph : public graph {
private:
    kernel first;
    kernel second;
public:
    input_port in;
    output_port out;
    simpleGraph(){

        first = kernel::create(simple);
        second = kernel::create(simple);
        connect< window<128> > net0 (in, first.in[0]);
        connect< window<128> > net1 (first.out[0], second.in[0]);
        connect< window<128> > net2 (second.out[0], out);

        source(first) = "kernels.cc";
        source(second) = "kernels.cc";

        runtime<ratio>(first) = 0.1;
        runtime<ratio>(second) = 0.1;

    }
};
```

Note: See [Run-Time Ratio](#) for more information.

7. Define a top-level application file (for example `project.cpp`) that contains an instance of your graph class and connect the graph to a simulation platform to provide file input and output. In this example, these files are called `input.txt` and `output.txt`.

```
#include "project.h"

simpleGraph mygraph;
simulation<1,1> platform("input.txt","output.txt");
connect<> net0(platform.src[0], mygraph.in);
connect<> net1(mygraph.out, platform.sink[0]);

int main(void) {
    adf::return_code ret;
```



```
mygraph.init();
ret=mygraph.run(<number_of_iterations>);
if(ret!=adf::ok){
    printf("Run failed\n");
    return ret;
}
ret=mygraph.end();
if(ret!=adf::ok){
    printf("End failed\n");
    return ret;
}
return 0;
}
```



IMPORTANT! By default, the `mygraph.run()` option specifies a graph that runs forever. The AI Engine compiler generates code to execute the data flow graph in a perpetual *while* loop. To limit the execution of the graph for debugging and test, specify the `mygraph.run(<number_of_iterations>)` in the graph code. The specified number of iterations can be one or more.

ADF APIs have return enumerate type `return_code` to show the API running status.

The `main` program is the driver for the graph. It is used to load, execute, and terminate the graph. See [Chapter 5: Run-Time Graph Control API](#) for more details.

Note: Kernel code must be written in such a way that no name clashes occur when two kernels get assigned to the same core.

Run-Time Ratio

The run-time ratio is a user-specified constraint that allows the AI Engine tools the flexibility to put multiple AI Engine kernels into a single AI Engine, if their summarized run-time ratio is less to 1. The run-time ratio of a kernel can be computed using the following equation.

$$\text{run-time ratio} = (\text{cycles for one run of the kernel}) / (\text{cycle budget})$$

The cycle budget is the cycles allowed to run one invocation of the kernel which depends on the system throughput requirement.

Cycles for one run of the kernel can be estimated in the initial design stage. For example, if the kernel contains a loop that can be well pipelined, and each cycle is capable of handling that amount of data, then the cycles for one run of the kernel can be estimated by the following.

$$\text{synchronization of synchronous buffers} + \text{function initialization} + \text{loop count} \\ * \text{cycles of each iteration of the loop} + \text{preamble and postamble of the loop}$$

Note: For more information about loop pipelining, see the *AI Engine Kernel Coding Best Practices Guide* ([UG1079](#)).

Cycles for one run of the kernel can also be profiled in the AI Engine simulator when vectorized code is available.

If multiple AI Engine kernels are put into a single AI Engine, they run in a sequential manner, one after the other, and they all run once with each iteration of `graph::run`. This means the following.

- If the AI Engine run-time percentage (specified by the run-time constraint) is allocated for the kernel in each iteration of `graph::run` (or on an average basis, depending on the system requirement), the kernel performance requirement can be met.
- For a single iteration of `graph::run`, the kernel takes no more percentage than that specified by the run-time constraint. Otherwise, it might affect other kernels' performance that are located in the same AI Engine.
- Even if multiple kernels have a summarized run-time ratio less than one, they are not necessarily put into a single AI Engine. The mapping of an AI Engine kernel into an AI Engine is also affected by hardware resources. For example, there must be enough program memory to allow the kernels to be in the same AI Engine, and also, stream interfaces must be available to allow all the kernels to be in the same AI Engine.
- When multiple kernels are put into the same AI Engine, resources might be saved. For example, the buffers between the kernels in the same AI Engine are single buffers instead of ping-pong buffers.
- Increasing the run-time ratio of a kernel does not necessarily mean that the performance of the kernel or the graph is increased, because the performance is also affected by the data availability to the kernel and the data throughput in and out of the graph. An unreasonably high run-time ratio setting might result in inefficient resource utilization.
- Low run-time ratio does not necessarily limit the performance of the kernel to the specified percentage of the AI Engine. For example, the kernel can run immediately when all the data is available if there is only one kernel in the AI Engine, no matter what run-time ratio is set.
- Kernels in different top-level graphs can not be put into the same AI Engine, because the graph API needs to control different graphs independently.
- Set the run-time ratio as accurately as possible, because it affects not only the AI Engine to be used, but also the data communication routes between kernels. It might also affect other design flows, for example, the power estimation.

Recommended Project Directory Structure

The following directory structure and coding practices are recommended for organizing your AI Engine projects to provide clarity and reuse.

- All adaptive data flow (ADF) graph class definitions, that is, all the ADF graphs that are derived from graph class `adf::graph`, must be located in a header file. Multiple ADF graph definitions can be included in the same header file. This class header file should be included in the `main` application file where the actual top-level graph is declared in the file scope (see [Creating a Data Flow Graph \(Including Kernels\)](#)).

- There should be no dependencies on the order that the header files are included. All header files must be self-contained and include all the other header files that it needs.
- There should be no file scoped variable or data-structure definitions in the graph header files. Any definitions (including `static`) must be declared in a separate source file that can be identified in the `header` property of the kernel where they are referenced (see [Look-up Tables](#)).
- There is no need to declare the kernels under `extern "C" { ... }`. However, this declaration can be used in an application meant to run full-program simulation, but it must adhere to the following conditions:
 - If the kernel-function declaration is wrapped with `extern "C"`, then the definition must know about it. This can be done by either including the header file inside the definition file, or wrapping the definition with `extern "C"`.
 - The `extern "C"` must be wrapped with `#ifdef __cplusplus`. This is synonymous to how `extern "C"` is used in `stdio.h`.

Compiling and Running the Graph from the Command Line

1. To compile your graph, execute the following command (see [Chapter 9: Compiling an AI Engine Graph Application](#) for more details).

```
aiecompiler project.cpp
```

The program is called `project.cpp`. The AI Engine compiler reads the input graph specified, compiles it to the AI Engine array, produces various reports, and generates output files in the `Work` directory.

2. After parsing the C++ input into a graphical intermediate form expressed in JavaScript object notation (JSON), the AI Engine compiler does the resource mapping and scheduling analysis and maps kernel nodes in the graph to the processing cores in the AI Engine array and data windows to memory banks. The JSON representation is augmented with mapping information. Each AI Engine also requires a schedule of all the kernels mapped to it.

The input graph is first partitioned into groups of kernels to be mapped to the same core.

The output of the mapper can also be viewed as a tabular report in the file `project_mapping_analysis_report.txt`. This reports the mapping of nodes to processing cores and data windows to memory banks. Inter-processor communication is appropriately double-banked as ping-pong buffers.

3. The AI Engine compiler allocates the necessary locks, memory buffers, and DMA channels and descriptors, and generates routing information for mapping the graph onto the AI Engine array. It synthesizes a main program for each core that schedules all the kernels on the cores, and implements the necessary locking mechanism and data copy among buffers. The C program for each core is compiled using the Synopsys Single Core Compiler to produce loadable ELF files. The AI Engine compiler also generates control APIs to control the graph initialization, execution and termination from the `main` application and a simulator configuration script `scsim_config.json`. These are all stored within the `Work` directory under various sub-folders (see [Chapter 9: Compiling an AI Engine Graph Application](#) for more details).
4. After the compilation of the AI Engine graph, the AI Engine compiler writes a summary of compilation results called `<graph-file-name>.aiecompile_summary` to view in the Vitis analyzer. The summary contains a collection of reports, and diagrams reflecting the state of the AI Engine application implemented in the compiled build. The summary is written to the working directory of the AI Engine compiler as specified by the `--workdir` option, which defaults to `./Work`.

To open the AI Engine compiler summary, use the following command:

```
vitis_analyzer ./Work/graph.aiecompile_summary
```

5. To run the graph, execute the following command (see [Chapter 10: Simulating an AI Engine Graph Application](#) for more details).

```
aiesimulator --pkg-dir=./Work
```

This starts the SystemC-based simulator with the control program being the `main` application. The graph APIs which are used in the control program configure the AI Engine array including setting up static routing, programming the DMAs, loading the ELF files onto the individual cores, and then initiates AI Engine array execution.

At the end of the simulation, the output data is produced in the directory `aiesimulator_output` and it should match the reference data.

The graph can be loaded at device boot time in hardware or through the host application. Details on deploying the graph in hardware and the flow associated with it is described in detail in [Chapter 14: Integrating the Application Using the Vitis Tools Flow](#).

Note: Only AI Engine kernels that have been modified are recompiled in subsequent compilations of the AI Engine graph. Any un-modified kernels will not be recompiled.

Window and Streaming Data API

Data flow graph kernels operate on data streams that are infinitely long sequences of typed values. These data streams can be broken into separate blocks and these blocks are processed by a kernel. Kernels consume input blocks of data and produce output blocks of data. Kernels can also access the data streams in a sample-by-sample fashion. The data access API in these two cases are described in this chapter.

Note: The data movement APIs described in this chapter apply to both vector and scalar, signed and unsigned data. However, note that the AI Engine architecture supports unsigned integer *vector* arithmetic only for the 8-bit data types `aie::vector<uint8,16>`, `aie::vector<uint8,32>`, `aie::vector<uint8,64>`, `aie::vector<uint8,128>`. But for *scalar* arithmetic, all standard C unsigned integer data types `unsigned char(uint8)`, `unsigned short(uint16)`, `unsigned int(uint32)`, `unsigned long long(uint64)` are supported.

Data Access Mechanisms

Window-based Access

The view that a kernel has of incoming blocks of data is called an input window. Input windows are defined by type, to define the type of data contained within that window. This example shows a declaration of an input window carrying complex integers where both the real and the imaginary parts are 16-bits wide.

```
input_window<cint16> myFirstWindow;
```

The view that a kernel has of outgoing blocks of data is called an output window. Again, these are defined by a type. This example shows a declaration of an output window carrying 32-bit integers.

```
output_window<int32> myOtherWindow;
```

These window data structures are automatically inferred by the AI Engine compiler from the data flow graph connections and are automatically declared in the wrapper code implementing the graph control. The kernel functions merely operate on pointers to the window data structures that are passed to them as arguments. There is no need to declare these window data structures in the data flow graph or kernel program.

Synchronous Window Access

A kernel reads from its input windows and writes to its output windows. By default, the synchronization that is required to wait for an input window of data, or to provide an empty output window, is performed before entering the kernel. There is no synchronization needed within the kernel to read or write the individual elements of data after the kernel has started.

The size of the window (in bytes) is declared along with the connection declaration between a producer and a consumer port as shown in the following (see [Connections](#) for details). This establishes a window connection of 128 bytes between port `in` and the first input port of the kernel.

```
connect< window<128> > net0 (in, first.in[0]);
```

An optional second template parameter identifies the overlap (in bytes) from one block of data to the next, which is sometimes also referred to as the margin, as shown in the following. If a margin parameter is specified, the total memory allocated is window size + margin size.

```
connect< window<128, 32> net1 (in, first.in[0]);
```

These windows are designed to be accessed sequentially. The kernel programming reads the window type and starts from the first position. Therefore, a useful model is that of a current position, which can be advanced or rolled back on reads or writes. On starting a kernel, the current position is always in the correct position. For example, the current position for an input window for a filter is on the first sample to restore to the delay line. It could be an older sample in the case of filters requiring overlap of incoming data samples, in which case the connection needs to be declared using the overlap or margin as described above. Similarly, the current position for an output window is on the first sample to send to the next block, irrespective of whether that block requires a duplication of older samples. The kernel is free to manipulate this current position and it is not necessary that this position is at the end of the block when the kernel completes. Window data types are implemented as circular buffers.

Note: The minimum size for window allocation is 16 bytes. Window size allocation is rounded up to a multiple of 16 bytes. The minimum size for margin overlap is 32 bytes and must be a multiple of 32 bytes.

Note: In a multicast communication approach, all receivers are required to be same size. For example,

```
connect< window<128> > net0 (in, first.in[0]);
connect< window<128> > net1 (in, second.in[0]);
```

Asynchronous Window Access

In some situations, if you are not consuming a windows worth of data on every invocation of a kernel, or if you are not producing a windows worth of data on every invocation, then you can control the buffer synchronization by declaring the kernel port to be `async` as shown in the following.

```
connect< window<128, 32> net1 (in, async(first.in[0]));
```

This declaration tells the compiler to omit synchronization of the window buffer upon entry to the kernel. You must use window synchronization APIs shown *inside the kernel code* before accessing the window using read/write APIs, as shown in the following.

```
void super_kernel(input_window<int32> * data, output_window<int32> *
result) {
    ...
    window_acquire(data);          // acquire input window unconditionally inside
the kernel
    if (<somecondition>) {
        window_acquire(result); // acquire output window conditionally
    }
    ...
    window_release(data);          // do some computation with "data" and "result"
// release input window inside the kernel
    if (<somecondition>) {
        window_release(result); // release output window conditionally
    }
    ...
};
```

The `window_acquire` API performs the appropriate synchronization and initialization to ensure that the window object is available for read or write. The API keeps track of the appropriate buffer pointers and locks to be acquired internally, even if the window is shared across AI Engine processors and can be double-buffered. This API can be called unconditionally or conditionally under dynamic control, and is potentially a blocking operation. It is your responsibility to ensure that the corresponding `window_release` API is executed some time later (possibly even in a subsequent kernel call) to release the lock associated with that window object. Incorrect synchronization can lead to a deadlock in your code.

Stream-based Access

With a stream-based access model, the kernels receive an input stream or an output stream of typed data as an argument. Each access to these streams is synchronized, i.e., reads stall if the data is not available in the stream and writes stall if the stream is unable to accept new data.

An AI Engine supports two 32-bit input stream ports with `id=0` or `1` and two 32-bit output stream ports with `id=0` or `1`. This ID is supplied as an argument to the stream object constructors. The AI Engine compiler automatically allocates the input and output stream port IDs from left to right in the argument list of a kernel. Multiple kernels mapped to the same AI Engine are not allowed to share stream ports unless the streams are packet switched (see [Explicit Packet Switching](#)).

There is also a direct stream communication channel between the accumulator register of one AI Engine and the physically adjacent core, called a cascade. The cascade stream is connected within the AI Engine array in a snake-like linear fashion from AI Engine processor to processor.

The stream data structures are automatically inferred by the AI Engine compiler from data flow graph connections, and are automatically declared in the wrapper code implementing the graph control. The kernel functions merely operate on pointers to stream data structures that are passed to them as arguments. There is no need to declare these stream data structures in data flow graph or kernel program.

Window Operations for Kernels

Window Data Types

Table 2: Supported Window Data Types

Input Window Types	Output Window Types
<code>input_window<int8></code>	<code>output_window<int8></code>
<code>input_window<int16></code>	<code>output_window<int16></code>
<code>input_window<int32></code>	<code>output_window<int32></code>
<code>input_window<int64></code>	<code>output_window<int64></code>
<code>input_window<uint8></code>	<code>output_window<uint8></code>
<code>input_window<uint16></code>	<code>output_window<uint16></code>
<code>input_window<uint32></code>	<code>output_window_uint32</code>
<code>input_window<uint64></code>	<code>output_window<uint64></code>
<code>input_window<cint16></code>	<code>output_window<cint16></code>
<code>input_window<cint32></code>	<code>output_window<cint32></code>
<code>input_window<float></code>	<code>output_window<float></code>
<code>input_window<cfloat></code>	<code>output_window<cfloat></code>

Moving the Current Read/Write Position Forward

In the following description, `<input_window_type>` stands for any of the allowed input window data types. Likewise, `<output_window_type>` stands for any of the allowed output window data types.

Purpose	Input Window Type	Output Window Type
To increase the current read/write position by the count times of the underlying window type.	<code>void window_incr(<input_window_type> *w, int count);</code>	<code>void window_incr(<output_window_type> *w, int count);</code>
To increase the current read/write position by four times the count times of the underlying window type.	<code>void window_incr_v4(<input_window_type> *w, int count);</code>	<code>void window_incr_v4(<output_window_type> *w, int count);</code>
To increase the current read/write position by eight times the count times of the underlying window type.	<code>void window_incr_v8(<input_window_type> *w, int count);</code>	<code>void window_incr_v8(<output_window_type> *w, int count);</code>
To increase the current read/write position by 16 times the count times of the underlying window type.	<code>void window_incr_v16(<input_window_type> *w, int count);</code>	<code>void window_incr_v16(<output_window_type> *w, int count);</code>
To increase the current read/write position by 32 times the count times of the underlying window type.	<code>void window_incr_v32(<input_window_type> *w, int count);</code>	<code>void window_incr_v32(<output_window_type> *w, int count);</code>
To increase the current read/write position by 64 times the count times of the underlying window type.	<code>void window_incr_v64(<input_window_type> *w, int count);</code>	<code>void window_incr_v64(<output_window_type> *w, int count);</code>

Moving the Current Read/Write Position Backward

In the following description, `<input_window_type>` stands for any of the allowed input window data types. Likewise, `<output_window_type>` stands for any of the allowed output window data types.

Purpose	Input Window Type	Output Window Type
To decrease the current read/write position by the count times of the underlying window type.	<code>void window_decr(<input_window_type> *w, int count);</code>	<code>void window_decr(<output_window_type> *w, int count);</code>
To decrease the current read/write position by four times the count times of the underlying window type.	<code>void window_decr_v4(<input_window_type> *w, int count);</code>	<code>void window_decr_v4(<output_window_type> *w, int count);</code>
To decrease the current read/write position by eight times the count times of the underlying window type.	<code>void window_decr_v8(<input_window_type> *w, int count);</code>	<code>void window_decr_v8(<output_window_type> *w, int count);</code>
To decrease the current read/write position by 16 times the count times of the underlying window type.	<code>void window_decr_v16(<input_window_type> *w, int count);</code>	<code>void window_decr_v16(<output_window_type> *w, int count);</code>
To decrease the current read/write position by 32 times the count times of the underlying window type.	<code>void window_decr_v32(<input_window_type> *w, int count);</code>	<code>void window_decr_v32(<output_window_type> *w, int count);</code>
To decrease the current read/write position by 64 times the count times of the underlying window type.	<code>void window_decr_v64(<input_window_type> *w, int count);</code>	<code>void window_decr_v64(<output_window_type> *w, int count);</code>

Reading Data from an Input Window

The following code reads a scalar typed value from an input window of the same type. The current position is not modified and both functional form (returns the value) and procedural form (modifies a reference argument) are provided.

```
int8 window_read(input_window<int8> *w);
int16 window_read(input_window<int16> *w);
int32 window_read(input_window<int32> *w);
int64 window_read(input_window<int64> *w);
uint8 window_read(input_window<uint8> *w);
uint16 window_read(input_window<uint16> *w);
uint32 window_read(input_window<uint32> *w);
uint64 window_read(input_window<uint64> *w);
cint16 window_read(input_window<cint16> *w);
cint32 window_read(input_window<cint32> *w);
float window_read(input_window<float> *w);
cfloat window_read(input_window<cfloat> *w);

void window_read(input_window<int8> *w, int8 &v );
void window_read(input_window<int16> *w, int16 &v );
void window_read(input_window<int32> *w, int32 &v );
void window_read(input_window<int64> *w, int64 &v );
void window_read(input_window<uint8> *w, uint8 &v );
void window_read(input_window<uint16> *w, uint16 &v );
void window_read(input_window<uint32> *w, uint32 &v );
void window_read(input_window<uint64> *w, uint64 &v );
void window_read(input_window<cint16> *w, cint16 &v );
void window_read(input_window<cint32> *w, cint32 &v );
void window_read(input_window<float> *w, float &v );
void window_read(input_window<cfloat> *w, cfloat &v );
```

The following code reads a 4-way vector of typed value from an input window of the same type. The current position is not modified and both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
aie::vector<cint16,4> window_read_v<4>(input_window<cint16> *w);
aie::vector<int32,4> window_read_v<4>(input_window<int32> *w);
aie::vector<cint32,4> window_read_v<4>(input_window<cint32> *w);
aie::vector<int64,4> window_read_v<4>(input_window<int64> *w);
aie::vector<float,4> window_read_v<4>(input_window<float> *w);
aie::vector<cfloat,4> window_read_v<4>(input_window<cfloat> *w);

void window_read(input_window<cint16> *w, aie::vector<cint16,4> &v);
void window_read(input_window<int32> *w, aie::vector<int32,4> &v);
void window_read(input_window<cint32> *w, aie::vector<cint32,4> &v);
void window_read(input_window<int64> *w, aie::vector<int64,4> &v);
void window_read(input_window<float> *w, aie::vector<float,4> &v);
void window_read(input_window<cfloat> *w, aie::vector<cfloat,4> &v);
```

The following code reads an 8-way vector of typed value from an input window of the same type. The current position is not modified and both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
aie::vector<int16,8> window_read_v<8>(input_window<int16> *w);
aie::vector<cint16,8> window_read_v<8>(input_window<cint16> *w);
aie::vector<int32,8> window_read_v<8>(input_window<int32> *w);
aie::vector<float,8> window_read_v<8>(input_window<float> *w);

void window_read(input_window<int16> *w, aie::vector<int16,8> &v);
void window_read(input_window<cint16> *w, aie::vector<cint16,8> &v);
void window_read(input_window<int32> *w, aie::vector<int32,8> &v);
void window_read(input_window<float> *w, aie::vector<float,8> &v);
```

The following code reads a 16-way vector of typed value from an input window of the same type. The current position is not modified and both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
aie::vector<int8,16> window_read_v<16>(input_window<int8> *w);
aie::vector<uint8,16> window_read_v<16>(input_window<uint8> *w);
aie::vector<int16,16> window_read_v<16>(input_window<int16> *w);
aie::vector<cint16,16> window_read_v<16>(input_window<cint16> *w);
aie::vector<int32,16> window_read_v<16>(input_window<int32> *w);
aie::vector<cint32,16> window_read_v<16>(input_window<cint32> *w);
aie::vector<float,16> window_read_v<16>(input_window<float> *w);
aie::vector<cfloat,16> window_read_v<16>(input_window<cfloat> *w);

void window_read(input_window<int8> *w, aie::vector<int8,16> &v);
void window_read(input_window<uint8> *w, aie::vector<uint8,16> &v);
void window_read(input_window<int16> *w, aie::vector<int16,16> &v);
void window_read(input_window<cint16> *w, aie::vector<cint16,16> &v);
void window_read(input_window<int32> *w, aie::vector<int32,16> &v);
void window_read(input_window<cint32> *w, aie::vector<cint32,16> &v);
void window_read(input_window<float> *w, aie::vector<float,16> &v);
void window_read(input_window<cfloat> *w, aie::vector<cfloat,16> &v);
```

The following code reads a 32-way vector of typed value from an input window of the same type. The current position is not modified and both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
aie::vector<int8,32> window_read_v<32>(input_window<int8> *w);
aie::vector<uint8,32> window_read_v<32>(input_window<uint8> *w);
aie::vector<int16,32> window_read_v<32>(input_window<int16> *w);
aie::vector<cint16,32> window_read_v<32>(input_window<cint16> *w);
aie::vector<int32,32> window_read_v<32>(input_window<int32> *w);
aie::vector<float,32> window_read_v<32>(input_window<float> *w);

void window_read(input_window<int8> *w, aie::vector<int8,32> &v);
void window_read(input_window<uint8> *w, aie::vector<uint8,32> &v);
void window_read(input_window<int16> *w, aie::vector<int16,32> &v);
void window_read(input_window<cint16> *w, aie::vector<cint16,32> &v);
void window_read(input_window<int32> *w, aie::vector<int32,32> &v);
void window_read(input_window<float> *w, aie::vector<float,32> &v);
```

The following code reads a 64-way vector of typed value from an input window of the same type. The current position is not modified and both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
aie::vector<int8,64> window_read_v<64>(input_window<int8> *w);
aie::vector<uint8,64> window_read_v<64>(input_window<uint8> *w);
aie::vector<int16,64> window_read_v<64>(input_window<int16> *w);

void window_read(input_window<int8> *w, aie::vector<int8,64> &v);
void window_read(input_window<uint8> *w, aie::vector<uint8,64> &v);
void window_read(input_window<int16> *w, aie::vector<int16,64> &v);
```

Reading and Advancing an Input Window

The following code reads a scalar typed value from an input window of the same type and advances the window current position by one times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided.

```
int8 window_readincr(input_window<int8> *w);
int16 window_readincr(input_window<int16> *w);
int32 window_readincr(input_window<int32> *w);
int64 window_readincr(input_window<int64> *w);
uint8 window_readincr(input_window<uint8> *w);
uint16 window_readincr(input_window<uint16> *w);
uint32 window_readincr(input_window<uint32> *w);
uint64 window_readincr(input_window<uint64> *w);
cint16 window_readincr(input_window<cint16> *w);
cint32 window_readincr(input_window<cint32> *w);
float window_readincr(input_window<float> *w);
cfloat window_readincr(input_window<cfloat> *w);

void window_readincr(input_window<int8> *w, int8 &v);
void window_readincr(input_window<int16> *w, int16 &v);
void window_readincr(input_window<int32> *w, int32 &v);
void window_readincr(input_window<int64> *w, int64 &v);
void window_readincr(input_window<uint8> *w, uint8 &v);
void window_readincr(input_window<uint16> *w, uint16 &v);
void window_readincr(input_window<uint32> *w, uint32 &v);
void window_readincr(input_window<uint64> *w, uint64 &v);
void window_readincr(input_window<cint16> *w, cint16 &v);
void window_readincr(input_window<cint32> *w, cint32 &v);
void window_readincr(input_window<float> *w, float &v);
void window_readincr(input_window<cfloat> *w, cfloat &v);
```

The following code reads a 4-way vector of typed value from an input window of the same type and advances the window current position by four times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bit or 256-bit wide for vector operations.

```
aie::vector<cint16,4> window_readincr_v<4>(input_window<cint16> *w);
aie::vector<int32,4> window_readincr_v<4>(input_window<int32> *w);
aie::vector<cint32,4> window_readincr_v<4>(input_window<cint32> *w);
aie::vector<int64,4> window_readincr_v<4>(input_window<int64> *w);
aie::vector<float,4> window_readincr_v<4>(input_window<float> *w);
aie::vector<cfloat,4> window_readincr_v<4>(input_window<cfloat> *w);

void window_readincr(input_window<cint16> *w, aie::vector<cint16,4> &v);
void window_readincr(input_window<int32> *w, aie::vector<int32,4> &v);
void window_readincr(input_window<cint32> *w, aie::vector<cint32,4> &v);
void window_readincr(input_window<int64> *w, aie::vector<int64,4> &v);
void window_readincr(input_window<float> *w, aie::vector<float,4> &v);
void window_readincr(input_window<cfloat> *w, aie::vector<cfloat,4> &v);
```

The following code reads an 8-way vector of typed value from an input window of the same type and advances the window current position by eight times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bit or 256-bit wide for vector operations.

```
aie::vector<int16,8> window_readincr_v<8>(input_window<int16> *w);
aie::vector<cint16,8> window_readincr_v<8>(input_window<cint16> *w);
aie::vector<int32,8> window_readincr_v<8>(input_window<int32> *w);
aie::vector<float,8> window_readincr_v<8>(input_window<float> *w);

void window_readincr(input_window<int16> *w, aie::vector<int16,8> &v);
void window_readincr(input_window<cint16> *w, aie::vector<cint16,8> &v);
void window_readincr(input_window<int32> *w, aie::vector<int32,8> &v);
void window_readincr(input_window<float> *w, aie::vector<float,8> &v);
```

The following code reads a 16-way vector of typed value from an input window of the same type and advances the window current position by sixteen times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bit or 256-bit wide for vector operations.

```
aie::vector<int8,16> window_readincr_v<16>(input_window<int8> *w);
aie::vector<uint8,16> window_readincr_v<16>(input_window<uint8> *w);
aie::vector<int16,16> window_readincr_v<16>(input_window<int16> *w);
aie::vector<cint16,16> window_readincr_v<16>(input_window<cint16> *w);
aie::vector<int32,16> window_readincr_v<16>(input_window<int32> *w);
aie::vector<cint32,16> window_readincr_v<16>(input_window<cint32> *w);
aie::vector<float,16> window_readincr_v<16>(input_window<float> *w);
aie::vector<cfloat,16> window_readincr_v<16>(input_window<cfloat> *w);

void window_readincr(input_window<int8> *w, aie::vector<int8,16> &v);
void window_readincr(input_window<uint8> *w, aie::vector<uint8,16> &v);
void window_readincr(input_window<int16> *w, aie::vector<int16,16> &v);
void window_readincr(input_window<cint16> *w, aie::vector<cint16,16> &v);
void window_readincr(input_window<int32> *w, aie::vector<int32,16> &v);
void window_readincr(input_window<cint32> *w, aie::vector<cint32,16> &v);
void window_readincr(input_window<float> *w, aie::vector<float,16> &v);
void window_readincr(input_window<cfloat> *w, aie::vector<cfloat,16> &v);
```

The following code reads a 32-way vector of typed value from an input window of the same type and advances the window current position by thirty-two times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bit or 256-bit wide for vector operations.

```
aie::vector<int8,32> window_readincr_v<32>(input_window<int8> *w);
aie::vector<uint8,32> window_readincr_v<32>(input_window<uint8> *w);
aie::vector<int16,32> window_readincr_v<32>(input_window<int16> *w);
aie::vector<cint16,32> window_readincr_v<32>(input_window<cint16> *w);
aie::vector<int32,32> window_readincr_v<32>(input_window<int32> *w);
aie::vector<float,32> window_readincr_v<32>(input_window<float> *w);

void window_readincr(input_window<int8> *w, aie::vector<int8,32> &v);
void window_readincr(input_window<uint8> *w, aie::vector<uint8,32> &v);
void window_readincr(input_window<int16> *w, aie::vector<int16,32> &v);
void window_readincr(input_window<cint16> *w, aie::vector<cint16,32> &v);
void window_readincr(input_window<int32> *w, aie::vector<int32,32> &v);
void window_readincr(input_window<float> *w, aie::vector<float,32> &v);
```

The following code reads a 64-way vector of typed value from an input window of the same type and advances the window current position by sixty-four times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bit or 256-bit wide for vector operations.

```
aie::vector<int8,64> window_readincr_v<64>(input_window<int8> *w);
aie::vector<uint8,64> window_readincr_v<64>(input_window<uint8> *w);
aie::vector<int16,64> window_readincr_v<64>(input_window<int16> *w);

void window_readincr(input_window<int8> *w, aie::vector<int8,64> &v);
void window_readincr(input_window<uint8> *w, aie::vector<uint8,64> &v);
void window_readincr(input_window<int16> *w, aie::vector<int16,64> &v);
```

Reading and Decrementing an Input Window

The following code reads a scalar typed value from an input window of the same type and decrements the window current position by one times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided.

```
int8 window_readdecr(input_window<int8> *w);
int16 window_readdecr(input_window<int16> *w);
int32 window_readdecr(input_window<int32> *w);
int64 window_readdecr(input_window<int64> *w);
uint8 window_readdecr(input_window<uint8> *w);
uint16 window_readdecr(input_window<uint16> *w);
uint32 window_readdecr(input_window<uint32> *w);
uint64 window_readdecr(input_window<uint64> *w);
cint16 window_readdecr(input_window<cint16> *w);
cint32 window_readdecr(input_window<cint32> *w);
float window_readdecr(input_window<float> *w);
cfloat window_readdecr(input_window<cfloat> *w);
```



```
void window_readdecr(input_window<int8> *w, int8 &v );
void window_readdecr(input_window<int16> *w, int16 &v );
void window_readdecr(input_window<int32> *w, int32 &v );
void window_readdecr(input_window<int64> *w, int64 &v );
void window_readdecr(input_window<uint8> *w, uint8 &v );
void window_readdecr(input_window<uint16> *w, uint16 &v );
void window_readdecr(input_window<uint32> *w, uint32 &v );
void window_readdecr(input_window<uint64> *w, uint64 &v );
void window_readdecr(input_window<cint16> *w, cint16 &v );
void window_readdecr(input_window<cint32> *w, cint32 &v );
void window_readdecr(input_window<float> *w, float &v );
void window_readdecr(input_window<cfloat> *w, cfloat &v );
```

The following code reads a 4-way vector of typed value from an input window of the same type and decrements the window current position by four times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bit or 256-bit wide for vector operations.

```
aie::vector<cint16,4> window_readdecr_v<4>(input_window<cint16> *w);
aie::vector<int32,4> window_readdecr_v<4>(input_window<int32> *w);
aie::vector<cint32,4> window_readdecr_v<4>(input_window<cint32> *w);
aie::vector<int64,4> window_readdecr_v<4>(input_window<int64> *w);
aie::vector<float,4> window_readdecr_v<4>(input_window<float> *w);
aie::vector<cfloat,4> window_readdecr_v<4>(input_window<cfloat> *w);

void window_readdecr(input_window<cint16> *w, aie::vector<cint16,4> &v);
void window_readdecr(input_window<int32> *w, aie::vector<int32,4> &v);
void window_readdecr(input_window<cint32> *w, aie::vector<cint32,4> &v);
void window_readdecr(input_window<int64> *w, aie::vector<int64,4> &v);
void window_readdecr(input_window<float> *w, aie::vector<float,4> &v);
void window_readdecr(input_window<cfloat> *w, aie::vector<cfloat,4> &v);
```

The following code reads an 8-way vector of typed value from an input window of the same type and decrements the window current position by eight times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bit or 256-bit wide for vector operations.

```
aie::vector<int16,8> window_readdecr_v<8>(input_window<int16> *w);
aie::vector<cint16,8> window_readdecr_v<8>(input_window<cint16> *w);
aie::vector<int32,8> window_readdecr_v<8>(input_window<int32> *w);
aie::vector<float,8> window_readdecr_v<8>(input_window<float> *w);

void window_readdecr(input_window<int16> *w, aie::vector<int16,8> &v);
void window_readdecr(input_window<cint16> *w, aie::vector<cint16,8> &v);
void window_readdecr(input_window<int32> *w, aie::vector<int32,8> &v);
void window_readdecr(input_window<float> *w, aie::vector<float,8> &v);
```

The following code reads a 16-way vector of typed value from an input window of the same type and decrements the window current position by sixteen times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bit or 256-bit wide for vector operations.

```
aie::vector<int8,16> window_readdecr_v<16>(input_window<int8> *w);
aie::vector<uint8,16> window_readdecr_v<16>(input_window<uint8> *w);
aie::vector<int16,16> window_readdecr_v<16>(input_window<int16> *w);
aie::vector<cint16,16> window_readdecr_v<16>(input_window<cint16> *w);
aie::vector<int32,16> window_readdecr_v<16>(input_window<int32> *w);
aie::vector<cint32,16> window_readdecr_v<16>(input_window<cint32> *w);
aie::vector<float,16> window_readdecr_v<16>(input_window<float> *w);
aie::vector<cfloat,16> window_readdecr_v<16>(input_window<cfloat> *w);

void window_readdecr(input_window<int8> *w, aie::vector<int8,16> &v);
void window_readdecr(input_window<uint8> *w, aie::vector<uint8,16> &v);
void window_readdecr(input_window<int16> *w, aie::vector<int16,16> &v);
void window_readdecr(input_window<cint16> *w, aie::vector<cint16,16> &v);
void window_readdecr(input_window<int32> *w, aie::vector<int32,16> &v);
void window_readdecr(input_window<cint32> *w, aie::vector<cint32,16> &v);
void window_readdecr(input_window<float> *w, aie::vector<float,16> &v);
void window_readdecr(input_window<cfloat> *w, aie::vector<cfloat,16> &v);
```

The following code reads a 32-way vector of typed value from an input window of the same type and decrements the window current position by thirty-two times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bit or 256-bit wide for vector operations.

```
aie::vector<int8,32> window_readdecr_v<32>(input_window<int8> *w);
aie::vector<uint8,32> window_readdecr_v<32>(input_window<uint8> *w);
aie::vector<int16,32> window_readdecr_v<32>(input_window<int16> *w);
aie::vector<cint16,32> window_readdecr_v<32>(input_window<cint16> *w);
aie::vector<int32,32> window_readdecr_v<32>(input_window<int32> *w);
aie::vector<cint32,32> window_readdecr_v<32>(input_window<cint32> *w);
aie::vector<float,32> window_readdecr_v<32>(input_window<float> *w);
aie::vector<cfloat,32> window_readdecr_v<32>(input_window<cfloat> *w);

void window_readdecr(input_window<int8> *w, aie::vector<int8,32> &v);
void window_readdecr(input_window<uint8> *w, aie::vector<uint8,32> &v);
void window_readdecr(input_window<int16> *w, aie::vector<int16,32> &v);
void window_readdecr(input_window<cint16> *w, aie::vector<cint16,32> &v);
void window_readdecr(input_window<int32> *w, aie::vector<int32,32> &v);
void window_readdecr(input_window<cint32> *w, aie::vector<cint32,32> &v);
void window_readdecr(input_window<float> *w, aie::vector<float,32> &v);
void window_readdecr(input_window<cfloat> *w, aie::vector<cfloat,32> &v);
```


The following code reads a 64-way vector of typed value from an input window of the same type and decrements the window current position by sixty-four times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bit or 256-bit wide for vector operations.

```
aie::vector<int8,64> window_readdecr_v<64>(input_window<int8> *w);
aie::vector<uint8,64> window_readdecr_v<64>(input_window<uint8> *w);
aie::vector<int16,64> window_readdecr_v<64>(input_window<int16> *w);

void window_readdecr(input_window<int8> *w, aie::vector<int8,64> &v);
void window_readdecr(input_window<uint8> *w, aie::vector<uint8,64> &v);
void window_readdecr(input_window<int16> *w, aie::vector<int16,64> &v);
```

Writing Data to an Output Window

The following code writes a scalar typed value to an output window of the same type. The current position is not modified.

```
void window_write(output_window<int8> *w, int8 v);
void window_write(output_window<uint8> *w, uint8 v);
void window_write(output_window<int16> *w, int16 v);
void window_write(output_window<uint16> *w, uint16 v);
void window_write(output_window<int32> *w, int32 v);
void window_write(output_window<uint32> *w, uint32 v);
void window_write(output_window<int64> *w, int64 v);
void window_write(output_window<uint64> *w, uint64 v);
void window_write(output_window<float> *w, float v);
void window_write(output_window<double> *w, double v);
```

The following code writes a 4-way vector of a typed value to an output window of the same type. The current position is not modified.

```
void window_write(output_window<int16> *w, aie::vector<int16,4> v);
void window_write(output_window<int32> *w, aie::vector<int32,4> v);
void window_write(output_window<int64> *w, aie::vector<int64,4> v);
void window_write(output_window<float> *w, aie::vector<float,4> v);
void window_write(output_window<double> *w, aie::vector<double,4> v);
```

The following code writes an 8-way vector of a typed value to an output window of the same type. The current position is not modified.

```
void window_write(output_window<int16> *w, aie::vector<int16,8> v);
void window_write(output_window<int32> *w, aie::vector<int32,8> v);
void window_write(output_window<int64> *w, aie::vector<int64,8> v);
void window_write(output_window<float> *w, aie::vector<float,8> v);
void window_write(output_window<double> *w, aie::vector<double,8> v);
```

The following code writes a 16-way vector of a typed value to an output window of the same type. The current position is not modified.

```
void window_write(output_window<int8> *w, aie::vector<int8,16> v);
void window_write(output_window<uint8> *w, aie::vector<uint8,16> v);
void window_write(output_window<int16> *w, aie::vector<int16,16> v);
void window_write(output_window<cint16> *w, aie::vector<cint16,16> v);
void window_write(output_window<int32> *w, aie::vector<int32,16> v );
void window_write(output_window<cint32> *w, aie::vector<cint32,16> v);
void window_write(output_window<float> *w, aie::vector<float,16> v );
void window_write(output_window<cfloat> *w, aie::vector<cfloat,16> v);
```

The following code writes a 32-way vector of a typed value to an output window of the same type. The current position is not modified.

```
void window_write(output_window<int8> *w, aie::vector<int8,32> v);
void window_write(output_window<uint8> *w, aie::vector<uint8,32> v);
void window_write(output_window<int16> *w, aie::vector<int16,32> v);
void window_write(output_window<cint16> *w, aie::vector<cint16,32> v);
void window_write(output_window<int32> *w, aie::vector<int32,32> v );
void window_write(output_window<float> *w, aie::vector<float,32> v );
```

The following code writes a 64-way vector of a typed value to an output window of the same type. The current position is not modified.

```
void window_write(output_window<int8> *w, aie::vector<int8,64> v);
void window_write(output_window<uint8> *w, aie::vector<uint8,64> v);
void window_write(output_window<int16> *w, aie::vector<int16,64> v);
```

Writing and Advancing an Output Window

The following code writes a scalar typed value to an output window of the same type and advances the current position based upon that type.

```
void window_writeincr (output_window<int8> *w, int8 v);
void window_writeincr (output_window<uint8> *w, uint8 v);
void window_writeincr (output_window<int16> *w, int16 v);
void window_writeincr (output_window<uint16> *w, uint16 v);
void window_writeincr (output_window<cint16> *w, cint16 v);
void window_writeincr (output_window<int32> *w, int32 v );
void window_writeincr (output_window<uint32> *w, uint32 v );
void window_writeincr (output_window<cint32> *w, cint32 v);
void window_writeincr (output_window<int64> *w, int64 v);
void window_writeincr (output_window<uint64> *w, uint64 v);
void window_writeincr (output_window<float> *w, float v );
void window_writeincr (output_window<cfloat> *w, cfloat v);
```

The following code writes a 4-way vector of a typed value to an output window of the same type and advances the current position by four times the size of the underlying type.

```
void window_writeincr(output_window<cint16> *w, aie::vector<cint16,4> v);
void window_writeincr(output_window<int32> *w, aie::vector<int32,4> v );
void window_writeincr(output_window<cint32> *w, aie::vector<cint32,4> v);
void window_writeincr(output_window<int64> *w, aie::vector<int64,4> v );
void window_writeincr(output_window<float> *w, aie::vector<float,4> v );
void window_writeincr(output_window<cfloat> *w, aie::vector<cfloat,4> v);
```

The following code writes an 8-way vector of a typed value to an output window of the same type and advances the current position by eight times the size of the underlying type.

```
void window_writeincr(output_window<int16> *w, aie::vector<int16,8> v);
void window_writeincr(output_window<cint16> *w, aie::vector<cint16,8> v);
void window_writeincr(output_window<int32> *w, aie::vector<int32,8> v );
void window_writeincr(output_window<float> *w, aie::vector<float,8> v );
```

The following code writes a 16-way vector of a typed value to an output window of the same type and advances the current position by sixteen times the size of the underlying type.

```
void window_writeincr(output_window<int8> *w, aie::vector<int8,16> v);
void window_writeincr(output_window<uint8> *w, aie::vector<uint8,16> v);
void window_writeincr(output_window<int16> *w, aie::vector<int16,16> v);
void window_writeincr(output_window<cint16> *w, aie::vector<cint16,16> v);
void window_writeincr(output_window<int32> *w, aie::vector<int32,16> v );
void window_writeincr(output_window<cint32> *w, aie::vector<cint32,16> v);
void window_writeincr(output_window<float> *w, aie::vector<float,16> v );
void window_writeincr(output_window<cfloat> *w, aie::vector<cfloat,16> v);
```

The following code writes a 32-way vector of a typed value to an output window of the same type and advances the current position by thirty-two times the size of the underlying type.

```
void window_writeincr(output_window<int8> *w, aie::vector<int8,32> v);
void window_writeincr(output_window<uint8> *w, aie::vector<uint8,32> v);
void window_writeincr(output_window<int16> *w, aie::vector<int16,32> v);
void window_writeincr(output_window<cint16> *w, aie::vector<cint16,32> v);
void window_writeincr(output_window<int32> *w, aie::vector<int32,32> v );
void window_writeincr(output_window<float> *w, aie::vector<float,32> v );
```

The following code writes a 64-way vector of a typed value to an output window of the same type and advances the current position by sixty-four times the size of the underlying type.

```
void window_writeincr(output_window<int8> *w, aie::vector<int8,64> v);
void window_writeincr(output_window<uint8> *w, aie::vector<uint8,64> v);
void window_writeincr(output_window<int16> *w, aie::vector<int16,64> v);
```

Stream Operations for Kernels

Stream Data Types

Table 3: Supported Stream Data Types

Input Stream Types	Output Stream Types
<code>input_stream<int8></code>	<code>output_stream<int8></code>
<code>input_stream<int16></code>	<code>output_stream<int16></code>
<code>input_stream<int32></code>	<code>output_stream<int32></code>
<code>input_stream<int64></code>	<code>output_stream<int64></code>
<code>input_stream<uint8></code>	<code>output_stream<uint8></code>
<code>input_stream<uint16></code>	<code>output_stream<uint16></code>
<code>input_stream<uint32></code>	<code>output_stream<uint32></code>
<code>input_stream<uint64></code>	<code>output_stream<uint64></code>
<code>input_stream<cint16></code>	<code>output_stream<cint16></code>
<code>input_stream<cint32></code>	<code>output_stream<cint32></code>
<code>input_stream<acc48></code>	<code>output_stream<acc48></code>
<code>input_stream<cacc48></code>	<code>output_stream<cacc48></code>
<code>input_stream<acc80></code>	<code>output_stream_acc80</code>
<code>input_stream<cacc80></code>	<code>output_stream<cacc80></code>
<code>input_stream<accfloat></code>	<code>output_stream<accfloat></code>
<code>input_stream<caccfloat></code>	<code>output_stream<caccfloat></code>
<code>input_stream<float></code>	<code>output_stream<float></code>
<code>input_stream<cfloat></code>	<code>output_stream<cfloat></code>

Each of the data types in the table can be read or written from the AI Engine as either scalars or in vector groups. However, there are certain restrictions on valid groupings based on the bus data width supported on the AI Engine to programmable logic interface ports or through the stream-switch network. The valid combinations for AI Engine kernels are vector bundles totaling up to 32-bits or 128-bits. The accumulator data types are only used to specify cascade-stream connections between adjacent AI Engines. Its valid groupings are based on the 384-bit wide cascade channel between two processors.

Note: To use these data types, it is necessary to use `#include <adf.h>` in the kernel source file.

Reading and Advancing an Input Stream

AI Engine Operations

The following operations read data from the given input stream and advance the stream on the AI Engine. Because there are two input stream ports on the AI Engine, the physical port assignment is made by the AI Engine compiler automatically and conveyed as part of the stream data structure. Data values from the stream can be read one at a time or as a vector. In the latter case, unless all values are present, the stream operation stalls. The data groupings are based on the underlying single cycle, 32-bit stream operation or 4 cycle, 128-bit wide stream operation. The cascade connection reads all accumulator values in parallel.

```
int32 readincr(input_stream<int32> *w);
uint32 readincr(input_stream<uint32> *w);
cint16 readincr(input_stream<cint16> *w);
float readincr(input_stream<float> *w);
cfloat readincr(input_stream<cfloat> *w);

aie::vector<int8,16> readincr_v16(input_stream<int8> *w);
aie::vector<uint8,16> readincr_v16(input_stream<uint8> *w);
aie::vector<int16,8> readincr_v8(input_stream<int16> *w);
aie::vector<cint16,4> readincr_v4(input_stream<cint16> *w);
aie::vector<int32,4> readincr_v4(input_stream<int32> *w);
aie::vector<cint32,2> readincr_v2(input_stream<cint32> *w);
aie::vector<float,4> readincr_v4(input_stream<float> *w);

aie::accum<acc48,8> readincr_v8(input_stream<acc48> *w);
aie::accum<cacc48,4> readincr_v4(input_stream<cacc48> *w);
aie::accum<acc80,4> readincr_v4(input_stream<acc80> * str);
aie::accum<cacc80,2> readincr_v2(input_stream<cacc80> * str);
aie::accum<accfloat,8> readincr_v8(input_stream<accfloat> * str);
aie::accum<caccfloat,8> readincr_v4(input_stream<caccfloat> * str);
```

Writing and Advancing an Output Stream

AI Engine Operations

The following operations write data to the given output stream and advance the stream on the AI Engine. Because there are two output stream ports on the AI Engine, the physical port assignment is made by the AI Engine compiler automatically and conveyed as part of the stream data structure. Data values can be written to the output stream one at a time or as a vector. In the latter case, until all values are written, the stream operation stalls. The data groupings are based on the underlying single cycle, 32-bit stream operation or 4 cycle, 128-bit wide stream operation. Cascade connection writes all values in parallel.

```
void writeincr(output_stream<int32> *w, int32 v);
void writeincr(output_stream<uint32> *w, uint32 v);
void writeincr(output_stream<cint16> *w, cint16 v);
void writeincr(output_stream<float> *w, float v);
void writeincr(output_stream<cfloat> *w, cfloat v);

void writeincr_v<16>(output_stream<int8> *w, aie::vector<int8,16> &v);
```

```
void writeincr_v<16>(output_stream<uint8> *w, aie::vector<uint8,16> &v);
void writeincr_v<8>(output_stream<int16> *w, aie::vector<int16,8> &v);
void writeincr_v<4>(output_stream<uint16> *w, aie::vector<uint16,4> &v);
void writeincr_v<4>(output_stream<int32> *w, aie::vector<int32,4> &v);
void writeincr_v<2>(output_stream<uint32> *w, aie::vector<uint32,2> &v);
void writeincr_v<4>(output_stream<float> *w, aie::vector<float,4> &v);

void writeincr_v<8>(output_stream<acc48> *w, aie::accum<acc48,8> &v);
void writeincr_v<4>(output_stream<cacc48> *w, aie::accum<cacc48,4> &v);
void writeincr_v<4>(output_stream<acc80> *str, aie::accum<acc80,4> &v);
void writeincr_v<2>(output_stream<cacc80> *str, aie::accum<cacc80,2> &v);
void writeincr_v<8>(output_stream<accfloat> *str, aie::accum<accfloat,8>
&v);
void writeincr_v<4>(output_stream<caccfloat> *str, aie::accum<caccfloat,4>
&v);
```

Using Streams in Parallel

For streaming input and output interfaces, when the performance is limited by the streaming interface, it is possible to use two streaming inputs or two streaming outputs in parallel. To use two parallel streams, it is recommended to use the following pairs of macros, where `idx1` and `idx2` are the two streams. Add the `restrict` keyword to the stream ports to optimize them for parallel processing. The macro is operating on 32 bits every cycle, or 128 bits per four cycles.

```
READINCR(SS_rsrc1, idx1) and READINCR(SS_rsrc2, idx2)
READINCRW(WSS_rsrc1, idx1) and READINCRW(WSS_rsrc2, idx2)
WRITEINCR(MS_rsrc1, idx1, val) and WRITEINCR(MS_rsrc2, idx2, val)
WRITEINCRW(WMS_rsrc1, idx1, val) and WRITEINCRW(WMS_rsrc2, idx2, val)
```

The following example code shows two parallel input streams using pipelining with an interval of one.

```
void simple( input_stream<int32> * restrict data0, input_stream<int32> *
restrict data1,
    output_stream<int32> * restrict out) {
    for(int i=0; i<1024; i++)
        chess_prepare_for_pipelining
        {
            int32_t d = READINCR(SS_rsrc1, data0) ;
            int32_t e = READINCR(SS_rsrc2, data1) ;
            WRITEINCR(MS_rsrc1,out,d+e);
        }
}
```

Packet Stream Operations

Table 4: Supported Packet Stream Data Types

Input Stream Types	Output Stream Types
input_pktstream	output_pktstream

Two additional stream data types are provided to characterize streaming data that consists of packetized interleaving of several different streams. These data types are useful when the number of independent data streams in your program exceeds the number of hardware stream channels or ports available. This mechanism is described in more detail in [Explicit Packet Switching](#).

Packet Stream Reading and Writing

A data packet consists of a one word (32-bit) packet header, followed by some number of data words where the last data word has the TLAST field denotes the end-of-packet. The following operations are used to read and advance input packet streams and write and advance output packet streams.

```
int32 readincr(input_pktstream *w);
int32 readincr(input_pktstream *w, bool &tlast);

void writeincr(output_pktstream *w, int32 value);
void writeincr(output_pktstream *w, int32 value, bool tlast);
```

The API with TLAST argument help to read or write the end-of-packet condition if the packet size is not fixed.

Packet Processing

The first 32-bit word of a packet must always be a packet header, which encodes several bit fields as shown in the following table.

Table 5: Packet Bit Fields

Bits	Field
4-0	Packet ID
11-5	7'b0000000
14-12	Packet Type
15	1'b0
20-16	Source Row
27-21	Source Column
30-28	3'b000
31	Odd parity of bits[30:0]

The packet ID is assigned by the compiler based on routing requirements. The packet type can be any 3-bit pattern that you want to insert to identify the type of packet. The source row and column denote the AI Engine tile coordinates from where the packet originated. By convention, source row and column for packets originating in the programmable logic (PL) is -1,-1.

It is your responsibility to construct and send an appropriate packet header at the beginning of every packet. On the receive side, the packet header needs to be received and decoded before reading the data.

The following operations help to assemble or disassemble the packet header in the AI Engine kernel.

```
void writeHeader(output_pktstream *str, unsigned int pktType, unsigned int ID);
void writeHeader(output_pktstream *str, unsigned int pktType, unsigned int ID, bool tlast);

uint32 getPacketid(input_pktstream *w, int index);
uint32 getPacketid(output_pktstream *w, int index);
```

The `writeHeader` API allows a packet header to be assembled with a given packet ID and packet type. The source row and column are inserted automatically using the coordinates of the AI Engine tile where this API is executed.

The `getPacketid` API allows the compiler assigned packet ID to be queried on the input or output packet stream data structure. The index argument refers to the split or merge branch edge in the graph specification.



IMPORTANT! *The `writeHeader()` and `getPacketid()` APIs are not supported in PL kernels.*



IMPORTANT! *The `generateHeader` API has been deprecated and replaced with the `writeHeader` API.*

See [Explicit Packet Switching](#) for more details.

Run-Time Graph Control API

This chapter describes the control APIs that can be used to initialize, run, update, and control the graph execution from an external controller. This chapter also describes how run-time parameters (RTP) can be specified in the input graph specification that affect the data processing within the kernels and change the control flow of the overall graph synchronously or asynchronously.

Graph Execution Control

In Versal® ACAPs with AI Engines, the processing system (PS) can be used to dynamically load, monitor, and control the graphs that are executing on the AI Engine array. Even if the AI Engine graph is loaded once as a single bitstream image, the PS program can be used to monitor the state of the execution and modify the run-time parameters of the graph.

The `graph` base class provides a number of API methods to control the initialization and execution of the graph that can be used in the `main` program. The user `main` application used in simulating the graph does not support `argc`, `argv` parameters. See [Appendix A: Adaptive Data Flow Graph Specification Reference](#) for more details.

Basic Iterative Graph Execution

The following graph control API shows how to use graph APIs to initialize, run, wait, and terminate graphs for a specific number of iterations. A graph object `mygraph` is declared using a pre-defined graph class called `simpleGraph`. Then, in the `main` application, this graph object is initialized and run. The `init()` method loads the graph to the AI Engine array at prespecified AI Engine tiles. This includes loading the ELF binaries for each AI Engine, configuring the stream switches for routing, and configuring the DMAs for I/O. It leaves the processors in a disabled state. The `run()` method starts the graph execution by enabling the processors. The `run` API is where a specific number of iterations of the graph can be run by supplying a positive integer argument at run time. This form is useful for debugging your graph execution.

```
#include "project.h"
simpleGraph mygraph;

int main(void) {
    mygraph.init();
    mygraph.run(3); // run 3 iterations
    mygraph.wait(); // wait for 3 iterations to finish
    mygraph.run(10); // run 10 iterations
    mygraph.end(); // wait for 10 iterations to finish
    return 0;
}
```

The API `wait()` is used to wait for the first run to finish before starting the second run. `wait` has the same blocking effect as `end` except that it allows re-running the graph again without having to re-initialize it. Calling `run` back-to-back without an intervening `wait` to finish that run can have an unpredictable effect because the `run` API modifies the loop bounds of the active processors of the graph.

Finite Execution of Graph

For finite graph execution, the graph state is maintained across the `graph.run(n)`. The AI Engine is not reinitialized and memory contents are not cleared after `graph.run(n)`. In the following code example, after the first run of three invocations, the AI Engine `main()` wrapper code is left in a state where the kernel will start with the pong buffer in the next run (of ten iterations). The ping-pong buffer selector state is left as-is. `graph.end()` does not clean up the graph state (specifically, does not re-initialize global variables), nor clean up stream switch configurations. It merely exits the core-main. To re-run the graph, you must reload the PDI/XCLBIN.



IMPORTANT! A `graph.wait()` must be followed by either a `graph.run()` or `graph.resume()` prior to a `graph.end()`. Failing to do so means that a graph could wait forever, and `graph.end()` never executes. See [Graph Objects](#) for more details on the use of these APIs.

```
#include "project.h"
simpleGraph mygraph;

int main(void) {
    mygraph.init();
    mygraph.run(3); // run 3 iterations
    mygraph.wait(); // wait for 3 iterations to finish
    mygraph.run(10); // run 10 iterations
    mygraph.end(); // wait for 10 iterations to finish
    return 0;
}
```

Infinite Graph Execution

The following graph control API shows how to run the graph infinitely.

```
#include "project.h"
simpleGraph mygraph;

int main(void) {
    mygraph.init(); // load the graph
    mygraph.run();  // start the graph
    return 0;
}
```

A graph object `mygraph` is declared using a pre-defined graph class called `simpleGraph`. Then, in the `main` application, this graph object is initialized and run. The `init()` method loads the graph to the AI Engine array at prespecified AI Engine tiles. This includes loading the ELF binaries for each AI Engine, configuring the stream switches for routing, and configuring the DMAs for I/O. It leaves the processors in a disabled state. The `run()` method starts the graph execution by enabling the processors. This graph runs forever because the number of iterations to be run is not provided to the `run()` method.

`graph::run()` without an argument runs the AI Engine kernels for a previously specified number of iterations (which is infinity by default if the graph is run without any arguments). If the graph is run with a finite number of iterations, for example, `mygraph.run(3);mygraph.run()` the second run call also runs for three iterations. Use `graph::run(-1)` to run the graph infinitely regardless of the previous run iteration setting.

Parallel Graph Execution

In the previous API methods, only the `wait()` and `end()` methods are blocking operations that can block the `main` application indefinitely. Therefore, if you declare multiple graphs at the top level, you need to interleave the APIs suitably to execute the graphs in parallel, as shown.

```
#include "project.h"
simpleGraph g1, g2, g3;

int main(void) {
    g1.init(); g2.init(); g3.init();
    g1.run(<num-iter>); g2.run(<num-iter>); g3.run(<num-iter>);
    g1.end(); g2.end(); g3.end();
    return 0;
}
```

Note: Each graph should be started (`run`) only after it has been initialized (`init`). Also, to get parallel execution, all the graphs must be started (`run`) before any graph is waited upon for termination (`end`).

Timed Execution

In multi-rate graphs, all kernels need not execute for the same number of iterations. In such situations, a timed execution model is more suitable for testing. There are variants of the `wait` and `end` APIs with a positive integer that specifies a cycle timeout. This is the number of AI Engine cycles that the API call blocks before disabling the processors and returning. The blocking condition does not depend on any graph termination event. The graph can be in an arbitrary state at the expiration of the timeout.

```
#include "project.h"
simpleGraph mygraph;

int main(void) {
    mygraph.init();
    mygraph.run();
    mygraph.wait(10000); // wait for 10000 AI Engine cycles
    mygraph.resume();    // continue executing
    mygraph.end(15000);  // wait for another 15000 cycles and terminate
}
```

Note: The API `resume()` is used to resume execution from the point it was stopped after the first timeout. `resume` only resets the timer and enables the AI Engines. Calling `resume` after the AI Engine execution has already terminated has no effect.

Run-Time Parameter Specification

The data flow graphs shown until now are defined completely statically. However, in real situations you might need to modify the behavior of the graph based on some dynamic condition or event. The required modification could be in the data being processed, for example a modified mode of operation or a new coefficient table, or it could be in the control flow of the graph such as conditional execution or dynamically reconfiguring a graph with another graph. Run-time parameters (RTP) are useful in such situations. Either the kernels or the graphs can be defined to execute with parameters. Additional graph API are also provided to update or read these parameter values while the graph is running.

Two types of run-time parameters are supported. The first is the asynchronous or sticky parameters which can be changed at any time by either a controlling processor such as the Processing System (PS), or by another AI Engine kernel. They are read each time a kernel is invoked without any specific synchronization. These types of parameters can be used as filter coefficients that change infrequently, for example.

Synchronous or triggering parameters are the other type of supported run-time parameters. A kernel that requires a triggering parameter does not execute until these parameters have been written by a controlling processor. Upon a write, the kernel executes once, reading the new updated value. After completion, the kernel is blocked from executing until the parameter is updated again. This allows a different type of execution model from the normal streaming model, which can be useful for certain updating operations where blocking synchronization is important.

Run-time parameters can either be scalar values or array values. In the case where a controlling processor (such as the PS) is responsible for the update, the `graph.update()` API should be used.

Specifying Run-Time Data Parameters

Parameter Inference

If an integer scalar value appears in the formal arguments of a kernel function, then that parameter becomes a run-time parameter. In the following example, the arguments `select` and `result_out` are run-time parameters.

```
#ifndef FUNCTION_KERNELS_H
#define FUNCTION_KERNELS_H
    void simple_param(input_window<int32> *in, output_window<int32> *out,
        int32 select, int32 &result_out);
#endif
```

Run-time parameters are processed as ports alongside those created by streams and windows. Both scalar and array of scalar data types can be passed as run-time parameters. The valid scalar data types supported are `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `cint16`, `cint32`, `float`, `cfloat`.

Note: Structs and pointers cannot be passed as run-time parameters. Every AI Engine has a different memory space and pointers have an inconsistent meaning when passed between AI Engines. Furthermore, the PS and AI Engines have different pointer representations. Because structs can contain pointers as members, the passing of structs as run-time parameters is not supported.

In the following example, an array of 32 integers is passed as a parameter into the `filter_with_array_param` function.

```
#ifndef FUNCTION_KERNELS_H
#define FUNCTION_KERNELS_H
    void filter_with_array_param(input_window_cint16 * in,
        output_window_cint16 * out, const int32 (&coefficients)[32]);
#endif
```

Implicit ports are inferred for each parameter in the function argument, including the array parameters. The following table describes the type of port inferred for each function argument.

Table 6: Port Type per Parameter

Formal Parameter	Port Class
T	Input
const T	Input
T &	Inout
const T &	Input
const T (&)[...]	Input
T(&)[...]	Inout

From the table, you can see that when the AI Engine cannot make externally visible changes to the function parameter, an input port is inferred. When the formal parameter is passed by value, a copy is made, so changes to that copy are not externally visible. When a parameter is passed with a const qualifier, the parameter cannot be written, so these are also treated as input ports.

When the AI Engine kernel is passed a parameter reference and it is able to modify it, an inout port is inferred and can be used to pass parameters between AI Engine kernels or to allow reading back of results from the control processor.

Note: The inout port is a port that a kernel itself can read or write. But, from the graph, the inout port can only behave as an output port. Therefore, the inout port can only be connected to the input RTP port, or be read by `graph::read()`. The inout port *cannot* be updated by `graph::update()`.

Note: If a kernel wants to accept an input run-time parameter, modify its value, and pass back this modified value via an output run-time parameter, then the variable has to appear twice in the `arg` list, once as an input and once as an inout, for example, `kernel_function(int32 foo_in, int32 &foo_out)`.

Parameter Hookup

Both input and inout run-time parameter ports can be connected to corresponding hierarchical ports in their enclosing graph. This is the mechanism that parameters are exposed for run-time modification. In the following graph, an instance is created of the previously defined `simple_param` kernel. This kernel has two input ports, one output port and one inout port. The first argument to appear in the argument list, `in[0]`, is an input window. The second argument is an output window. The third argument is a run-time parameter (it is not a window or stream type) and is inferred as an input parameter, `in[1]`, because it is passed by value. The fourth argument is a run-time parameter and is inferred as an inout parameter, `inout[0]`, because it is passed by reference.

Note: The different port types `in`, `out`, and `inout` for a kernel belong to their own categories, and all start from index 0 in the graph.

In the following graph definition, a `simple_param` kernel is instantiated and windows are connected to `in[0]` and `out[0]` (the input and output windows of the kernel). The input run-time parameter is connected to the graph input port, `select_value`, and the inout run-time parameter is connected to the graph inout port, `result_out`.

```
class parameterGraph : public graph {
private:
    kernel first;

public:
    input_port select_value;
    input_port in;
    output_port out;
    inout_port result_out;
    parameterGraph() {
        first = kernel::create(simple_param);

        connect< window <32> >(in, first.in[0]);
        connect< window <32> >(first.out[0], out);
        connect<parameter>(select_value, first.in[1]); //default sync rtp input
        connect<parameter>(first.inout[0], result_out); //default async rtp
    }
};
```

An array parameter can be hooked up in the same way. The compiler automatically allocates space for the array data so that it is accessible from the processor where this kernel gets mapped.

```
class arrayParameterGraph : public graph {
private:
    kernel first;

public:
    input_port coeffs;
    input_port in;
    output_port out;
    arrayParameterGraph() {
        first = kernel::create(filter_with_array_param);

        connect< window <32> >(in, first.in[0]);
        connect< window <32> >(first.out[0], out);
        connect<parameter>(coeffs, first.in[1]);
    }
};
```

Input Parameter Synchronization

The default behavior for input run-time parameters ports is triggering behavior. This means that the parameter plays a part in the rules that determine when a kernel could execute. In this graph example, the kernel only executes when three conditions are met:

- A valid window of 32 bytes of input data is available
- An empty window of 32 bytes is available for the output data
- A write to the input parameter takes place

In triggering mode, a single write to the input parameter allows the kernel to execute once, setting the input parameter value on every individual kernel call.

There is an alternative mode to allow input kernels parameters to be set asynchronously. To specify that parameters update asynchronously, use the `async` modifier when connecting a port.

```
connect<parameter>(param_port, async(first.in[1]));
```

When a kernel port is designated as asynchronous, it no longer plays a role in the firing rules for the kernel. When the parameter is written once, the value is observed in subsequent firings. At any time, the PS can write a new value for the run-time parameter. That value is observed on the next and any subsequent kernel firing.

Inout Parameter Synchronization

The default behavior for inout run-time parameters ports is asynchronous behavior. This means that the parameter can be read back by the controlling processor or another kernel, but the producer kernel execution is not affected. For synchronous behavior from the inout parameter where the kernel blocks until the parameter value is read out on each invocation of the kernel, you can use a `sync` modifier when connecting the inout port to the enclosing graph as follows.

```
connect<parameter>(sync(first.inout[1]), param_port);
```

Run-Time Parameter Update/Read Mechanisms

This section describes the mechanisms to update or read back the run-time parameters. For these types of applications, it is usually better not to specify an iteration limit at compile time to allow the cores to run freely and monitor the effect of the parameter change.

Parameter Update/Read Using Graph APIs

In default compilation mode, the `main` application is compiled as a separate control thread which needs to be executed on the PS in parallel with the graph executing on the AI Engine array. The `main` application can use update and read APIs to access run-time parameters declared within the graphs at any level. This section describes these APIs using examples.

Synchronous Update/Read

The following code shows the main application of the `simple_param` graph described in [Specifying Run-Time Data Parameters](#).

```
#include "param.h"
parameterGraph mygraph;

int main(void) {
    mygraph.init();
    mygraph.run(2);

    mygraph.update(mygraph.select_value, 23);
    mygraph.update(mygraph.select_value, 45);

    mygraph.end();
    return 0;
}
```

In this example, the graph `mygraph` is initialized first and then run for two iterations. It has a triggered input parameter port `select_value` that must be updated with a new value for each invocation of the receiving kernel. The first argument of the `update` API identifies the port to be updated and the second argument provides the value. Several other forms of update APIs are supported based on the direction of the port, its data type, and whether it is a scalar or array parameter, see [Appendix A: Adaptive Data Flow Graph Specification Reference](#).

If the program is compiled with a fixed number of test iterations, then for triggered parameters the number of update API calls in the `main` program must match the number of test iterations, otherwise the simulation could be waiting for additional updates. For asynchronous parameters, the updates are done asynchronously with the graph execution and the kernel uses the old value if the update was not made.

If additionally, the previous graph was compiled with a synchronous inout parameter, then the update and read calls must be interleaved as shown in the following example.

```
#include "param.h"
parameterGraph mygraph;

int main(void) {
    int result0, result1;
    mygraph.init();
    mygraph.run(2);

    mygraph.update(mygraph.select_value, 23);
    mygraph.read(mygraph.result_out, result0);
    mygraph.update(mygraph.select_value, 45);
    mygraph.read(mygraph.result_out, result1);

    mygraph.end();
    return 0;
}
```

In this example, it is assumed that the graph produces a scalar result every iteration through the inout port `result_out`. The `read` API is used to read out the value of this port synchronously after each iteration. The first argument of the `read` API is the graph inout port to be read back and the second argument is the location where the value will be stored (passed by reference).

The synchronous protocol ensures that the read operation will wait for the value to be produced by the graph before sampling it and the graph will wait for the value to be read before proceeding to the next iteration. This is why it is important to interleave the `update` and `read` operations.

Asynchronous Update/Read

When an input parameter is specified with asynchronous protocol, the kernel execution waits for the first update to happen for parameter initialization. However, an arbitrary number of kernel invocations can take place before the next update. This is usually the intent of the asynchronous update during application deployment. However, for debugging, `wait` API can be used to finish a predetermined set of iterations before the next update as shown in the following example.

```
#include "param.h"
asyncGraph mygraph;

int main(void) {
    int result0, result1;
    mygraph.init();

    mygraph.update(mygraph.select_value, 23);
    mygraph.run(5);
    mygraph.wait();
    mygraph.update(mygraph.select_value, 45);
    mygraph.run(15);
    mygraph.end();
    return 0;
}
```

In the previous example, after the initial update, five iterations are run to completion followed by another update, then followed by another set of 15 iterations. If the graph has asynchronous inout ports, that data can also be read back immediately after the `wait` (or `end`).

Another template for asynchronous updates is to use timeouts in `wait` API as shown in the following example.

```
#include "param.h"
asyncGraph mygraph;

int main(void) {
    int result0, result1;
    mygraph.init();
    mygraph.run();
    mygraph.update(mygraph.select_value, 23);
    mygraph.wait(10000);
    mygraph.update(mygraph.select_value, 45);
    mygraph.resume();
    mygraph.end(15000);
    return 0;
}
```

In this example, the graph is set up to run forever. However, after the `run` API is called, it still blocks for the first update to happen for parameter initialization. Then, it runs for 10,000 cycles (approximately) before allowing the control thread to make another update. The new update takes effect at the next kernel invocation boundary. Then the graph is allowed to run for another 15,000 cycles before terminating.

Chained Updates Between AI Engine Kernels

The previous run-time parameter examples highlight the ability to do run-time parameter updates from the control processor. It is also possible to propagate parameter updates between AI Engines. If an inout port on a kernel is connected to an input port on another kernel, then a chain of updates can be triggered through multiple AI Engines. Consider the two kernels defined in the following code. The producer has an input port that reads a scalar integer and an inout port that can read and write to an array of 32 integers. The consumer has an input port that can read an array of coefficients, and an output port that can write a window of data.

```
#ifndef FUNCTION_KERNELS_H
#define FUNCTION_KERNELS_H

void producer(const int32 &, int32 (&)[32] );
void consumer(const int32 (&)[32], output_window_cint16 *);

#endif
```

As shown in the following graph, the PS updates the scalar input of the producer kernel. When the producer kernel is run, it automatically triggers execution of the consumer kernel (when a buffer is available for the output data).

```
#include <adf.h>
#include "kernels.h"

using namespace adf;

class chainedGraph : public graph {
private:
    kernel first;
    kernel second;

public:
    input_port select_value;
    output_port out;

    chainedGraph() {
        first = kernel::create(producer);
        second = kernel::create(consumer);

        connect< window <32> >(second.out[0], out);
        connect<parameter>(select_value, first.in[0]);
        connect<parameter>(first.inout[0], second.in[0]);
    }
};
```

If the intention is to make a one-time update of values that are used in continuous processing of streams, the consumer parameter input port can use the `async` modifier to ensure it runs continuously (when a parameter is provided).

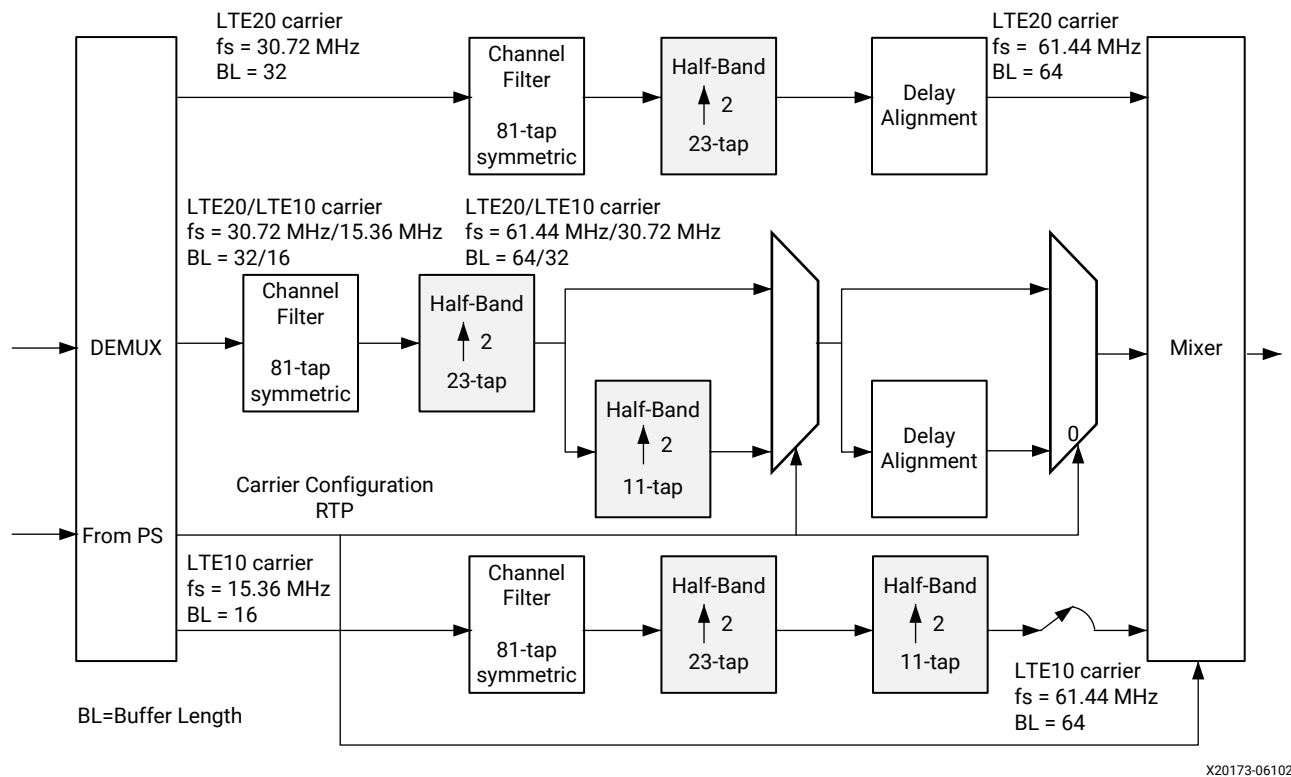
Run-Time Graph Reconfiguration Using Control Parameters

The run-time parameters are also used to switch the flow of data within the graph and provide alternative routes for processing dynamically. The most basic version of this type of processing is a kernel bypass that allows the data to be processed or pass-through based on a run-time parameter (see [Kernel Bypass](#)). This can be useful, for example, in multi-modal applications where switching from one mode to another requires bypassing a kernel.

Bypass Control Using Run-Time Parameters

The following figure shows an application supporting two channels of signal data, where one is split into two channels of lower bandwidth while the other must continue to run undisturbed. This type of dynamic reconfiguration is common in wireless applications.

Figure 3: Dynamic Reconfiguration of 2 LTE20 Channels into 1 LTE20 and 2 LTE10 Channels



In the figure, the first channel processes LTE20 data unchanged, while the middle channel is dynamically split into two LTE10 channels. The control parameters marked as `carrier configuration RTP` are used to split the data processing on a block boundary. When the middle channel is operating as an LTE20 channel, the 11-tap half-band kernel is bypassed. However, when the bandwidth of the middle channel is split between itself and the third channel forming two LTE10 channels, both of them need a 3-stage filter chain before the data can be mixed together. This is achieved by switching the 11-tap half-band filter back into the flow and reconfiguring the mixer to handle three streams of data instead of two.



TIP: The delay alignment kernels are needed to balance the sample delay when mixing LTE20 and LTE10 signals and must also be part of the control flow due to dynamic switching between the two.

The top-level input graph specification for the above application is shown in the following code.

```
class lte_reconfig : public graph {
private:
    kernel demux;
    kernel cf[3];
    kernel interp0[3];
    kernel interp1[2];
    bypass bphb11;
    kernel delay ;
    kernel delay_byp ;
    bypass bpdelay ;
```

```

kernel mixer ;
public:
    input_port in;
    input_port fromPS;
    output_port out ;

lte_reconfig() {
    // demux also handles the control
    demux = kernel::create(demultiplexor);
    connect< window<1536> >(in, demux.in[0]);
    connect< parameter >(fromPS, demux.in[1]);

    runtime<ratio>(demux) = 0.1;
    source(demux) = "kernels/demux.cc";

    // instantiate all channel kernels
    for (int i=0;i<3;i++) {
        cf[i] = kernel::create(fir_89t_sym);
        source(cf[i]) = "kernels/fir_89t_sym.cc";
        runtime<ratio>(cf[i]) = 0.12;
    }
    for (int i=0;i<3;i++) {
        interp0[i] = kernel::create(fir_23t_sym_hb_2i);
        source(interp0[i]) = "kernels/hb23_2i.cc";
        runtime<ratio>(interp0[i]) = 0.1;
    }
    for (int i=0;i<2;i++) {
        interp1[i] = kernel::create(fir_11t_sym_hb_2i);
        source(interp1[i]) = "kernels/hb11_2i.cc";
        runtime<ratio>(interp1[i]) = 0.1;
    }
    bphb11 = bypass::create(interp1[0]);
    mixer = kernel::create(mixer_dynamic);
    source(mixer) = "kernels/mixer_dynamic.cc";
    runtime<ratio>(mixer) = 0.4;
    delay = kernel::create(sample_delay);
    source(delay) = "kernels/delay.cc";
    runtime<ratio>(delay) = 0.1;
    delay_byp = kernel::create(sample_delay);
    source(delay_byp) = "kernels/delay.cc";
    runtime<ratio>(delay_byp) = 0.1;
    bpdelay = bypass::create(delay_byp) ;

    // Graph connections
    for (int i=0; i<3; i++) {
        connect< window<512, 352> >(demux.out[i], cf[i].in[0]);
        connect< parameter >(demux.inout[i], cf[i].in[1]);
    }
    connect< parameter >(demux.inout[3], bphb11.bp);
    connect< parameter >(demux.inout[3], negate(bpdelay.bp)) ;
    for (int i=0;i<3;i++) {
        connect< window<512, 64> >(cf[i].out[0], interp0[i].in[0]);
        connect< parameter >(cf[i].inout[0], interp0[i].in[1]);
    }
    // chan0 is LTE20 and is output right away
    connect< window<1024, 416> >(interp0[0].out[0], delay.in[0]);
    connect< window<1024> >(delay.out[0], mixer.in[0]);
    // chan1 is LTE20/10 and uses bypass
    connect< window<1024, 32> >(interp0[1].out[0], bphb11.in[0]);
    connect< parameter >(interp0[1].inout[0], bphb11.in[1]);
    connect< window<1024, 416> >(bphb11.out[0], bpdelay.in[0]);
    connect< window<1024> >(bpdelay.out[0], mixer.in[1]);
    // chan2 is LTE10 always

```

```
connect< window<512, 32> >(interp0[2].out[0], interp1[1].in[0]);
connect< parameter >(interp0[2].inout[0], interp1[1].in[1]);
connect< window<1024> >(interp1[1].out[0], mixer.in[2]);
//Mixer
connect< parameter >(demux.inout[3], mixer.in[3]);
connect< window<1024> >(mixer.out[0], out);
};
```

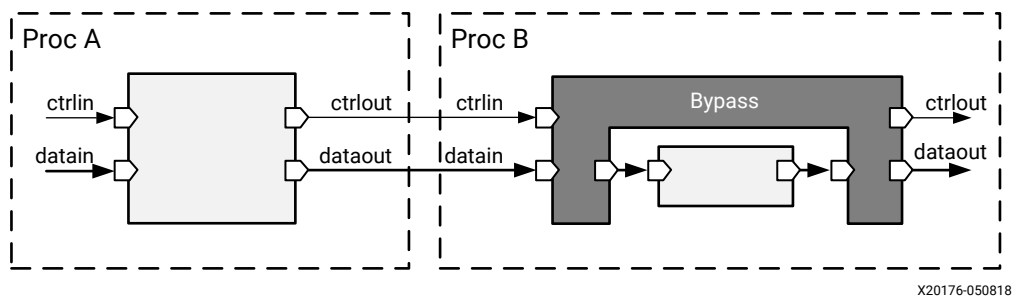
The bypass specification is coded as a special *encapsulator* over the kernel to be bypassed. The port signature of the bypass matches the port signature of the kernel that it encapsulates. It also receives a run-time parameter to control the bypass: 0 for no bypass and 1 for bypass. The control can also be inverted by using the negate function as shown.

The bypass parameter port of this graph is an ordinary scalar run-time parameter and can be driven by another kernel or by the Arm® processor using the interactive or scripted mechanisms described in [Run-Time Parameter Update/Read Mechanisms](#). This can also be connected hierarchically by embedding it into an enclosing graph.

Sharing Run-Time Parameters Across Multiple Kernels

The run-time parameter to switch channels in the previous graph is shared by the bypass encapsulator and the mixer kernel. Both entities need to see the same switched value at the same data boundary. When the nodes sharing the run-time parameter are mapped to the same AI Engine, switching of the parameter value is synchronized because each node, mapped to the same processor, processes the current set of data before any node processes the next set of data. However, when the sharing kernels are mapped to different processors, they can execute in a pipelined fashion on different sets of data, as shown in the following figure. Then, the run-time parameter should be pipelined along with the data.

Figure 4: Pipelined Run-Time Parameters



In the current release, you need to pipeline the control parameter *through* the kernel by making it an inout parameter on the producing kernel connected to an input parameter on the consuming kernel. Pipelining across processors can be intermixed with a one-to-many broadcast connection within a single processor to create arbitrary control topologies.

Run-Time Parameter Support Summary

This section summarizes the AI Engine run-time parameter (RTP) support status.

Table 7: AI Engine RTP Default and Support Status

AI Engine RTP (from/to PS)	Input		Output	
	Synchronous	Asynchronous	Synchronous	Asynchronous
Scalar	Default	Supported	Supported	Default
Array	Default	Supported	Supported	Default

Code snippets for RTP connections from or to the PS:

```
connect<parameter>(fromPS, first.in[0]); //Synchronous RTP, default for input
connect<parameter>(fromPS, sync(first.in[0])); //Synchronous RTP
connect<parameter>(fromPS, async(first.in[0])); //Asynchronous RTP
connect<parameter>(second.inout[0], toPS); //Asynchronous RTP, default for output
connect<parameter>(async(second.inout[0]), toPS); //Asynchronous RTP
connect<parameter>(sync(second.inout[0]), toPS); //Synchronous RTP
```

Table 8: AI Engine RTP to AI Engine RTP Connection Support Status

AI Engine RTP to AI Engine RTP		To		
		Synchronous	Asynchronous	Not Specified
From	Synchronous	Synchronous	Not Supported	Synchronous
	Asynchronous	Not Supported	Asynchronous	Asynchronous
	Not Specified	Synchronous	Asynchronous	Synchronous

Code snippets for RTP connections between AI Engines:

```
connect<parameter>(first.inout[0], second.in[0]); //Not specified for output and input.
Synchronous RTP from first.inout to second.in
connect<parameter>(sync(first.inout[0]), second.in[0]); //Specify "sync" for output.
Synchronous RTP from first.inout to second.in
connect<parameter>(first.inout[0], sync(second.in[0])); //Specify "sync" for input.
Synchronous RTP from first.inout to second.in
connect<parameter>(sync(first.inout[0]), sync(second.in[0])); //Specify "sync" for both.
Synchronous RTP from first.inout to second.in
connect<parameter>(async(first.inout[0]), async(second.in[0])); //Specify "async" for both.
Asynchronous RTP from first.inout to second.in
connect<parameter>(first.inout[0], async(second.in[0])); //Specify "async" for input.
```



```
Asynchronous RTP from first.inout to second.in
connect<parameter>(async(first.inout[0]), second.in[0]); //Specify "async" for output.
Asynchronous RTP from first.inout to second.in
connect<parameter>(async(first.inout[0]), sync(second.in[0])); //Not supported
connect<parameter>(sync(first.inout[0]), async(second.in[0])); //Not supported
```

Note: When using AI Engine RTP to AI Engine RTP connections, the source and destination kernels use shared memory for data communication. DMA is not involved in data communication. So, AI Engine kernels with RTP connections must be placed in the same AI Engine tile or adjacent tiles.

Table 9: AI Engine RTP Broadcast to Multiple AI Engine RTP Destinations Support Status

AI Engine RTP to AI Engine RTP		To				
		D1: Sync D2:Sync	D1:Async D2:Async	D1:Sync D2:Async	D1:Sync D2:Not Specified	D1:Async D2:Not Specified
From	S1:Sync S2:Sync	Synchronous	Not Supported	Not Supported	S1-D1:Sync S2-D2:Sync	Not Supported
	S1:Async S2:Async	Not Supported	Asynchronous	Not Supported	Not Supported	S1-D1:Async S2-D2:Async
	S1:Sync S2:Async	Not Supported	Not Supported	S1-D1:Sync S2-D2:Async	S1-D1:Sync S2-D2:Async	Not Supported
	S1:Sync S2:Not Specified	Not Supported	Not Supported	S1-D1:Sync S2-D2:Async	Synchronous	Not Supported
	S1:Async S2:Not Specified	Not Supported	Asynchronous	Not Supported	Not Supported	S1-D1:Async S2-D2:Sync

Notes:

1. S1 and S2 in this table denote the RTP source for the first and second connections. D1 and D2 denote the RTP ports of multiple destination kernels. Sync and Async denote the synchronous and asynchronous modes specified in the graph. S1-D1: Sync denotes the connection from S1 to D1 is synchronous mode.

Similar to the AI Engine RTP to AI Engine RTP connection, the source and destination kernels for the RTP broadcast must be placed in the same AI Engine tile or adjacent tiles. Code snippets are similar too. Only the source RTP port is used in multiple connections. The RTP buffers are shared for the source and destination kernels.

Note:

- There is a limitation in this version that only synchronous RTP is allowed between AI Engine kernel to AI Engine kernel when they are placed in the same AI Engine tile.
- Array RTP broadcast is not supported in the current version.
- Single buffer is not supported for RTP broadcast.

Specialized Graph Constructs

This chapter describes several graph constructs that help when modeling specific scenarios.

Look-up Tables

Static File-scoped Tables

Kernel functions can use private, read-only data structures that are accessed as file-scoped variables. The compiler allocates a limited amount of static heap space for such data. As an example, consider the following header file (`user_parameter.h`).

```
#ifndef USER_PARAMETER_H
#define USER_PARAMETER_H

#include <adf.h>

static int32 lutarray[8] = {1,2,3,4,5,6,0,0} ;

#endif
```

This header file can be included in the kernel source file and the look-up table can be accessed inside a kernel function directly. The `static` modifier ensures that the array definition is local to this file. The AI Engine compiler then allocates this array in static heap space for the processor where this kernel is used.

```
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include "user_parameter.h"

void simple_lut(input_window<int32> * in, output_window<int32> * out){
    aie::vector<int32,16> sbuff;
    aie::accum<acc80,4> acc;
    aie::vector<int32,8> coeffs=aie::load_v<8>((int32*)lutarray);

    window_readincr_v<16>(in, sbuff);
    acc = aie::sliding_mul<4,8>(coeffs, 0, sbuff, 0);
    window_writeincr(out, acc.template to_vector(0));
}
```

Global Graph-scoped Tables

While the previous example only includes an eight entry look-up table accessed as a global variable, many other algorithms require much larger look-up tables. Because AI Engine local memory is at a premium, it is much more efficient for the AI Engine compiler to manage the look-up table explicitly for specific kernels than to leave a large amount of stack or heap space on every processor. Such tables should *not* be declared static in the kernel header file.

```
#ifndef USER_PARAMETER_H
#define USER_PARAMETER_H

#include <adf.h>

int32 lutarray[8] = {1,2,3,4,5,6,0,0} ;

#endif
```

The kernel source continues to include the header file and use the table as before. But, now you must declare this table as `extern` in the graph class header and use the `parameter::array(...)` function to create a parameter object explicitly in the graph. You also need to attach this parameter object to the kernel as shown in the following code:

```
#include <adf.h>
extern int32 lutarray[8];
class simple_lut_graph : public graph {
public:
    kernel k;
    parameter p;

    simple_lut_graph() {
        k = kernel::create(simple);
        p = parameter::array(lutarray);
        connect<>(p,k);
        ...
    }
}
```

Including this explicit specification of the look-up table in the graph description ensures that the compiler is aware of the requirement to reserve a suitably sized piece of memory for the look-up table when it allocates memory for kernel input and output buffers.

Shared Graph-scoped Tables

Sometimes, the same table definition is used in multiple kernels. Because the AI Engine architecture is a distributed address-space architecture, each processor binary image that executes such a kernel needs to have that table defined in its own local memory. To get the correct graph linkage spread across multiple processors, you must declare the tables as `extern` within the kernel source file as well as the graph class definition file. Then, the actual table definition needs to be specified in a separate header file that is attached as a property to the kernel as shown below.

```
#include <adf.h>
extern int32 lutarray[8];
class simple_lut_graph : public adf::graph {
public:
    kernel k;
    parameter p;

    simple_lut_graph() {
        k = kernel::create(simple);
        p = parameter::array(lutarray);
        connect<>(p,k);

        std::vector<std::string> myheaders;
        myheaders.push_back("./user_parameter.h");
        headers(k) = myheaders;
        ...
    }
}
```

This ensures that the header file that defines the table is included in the final binary link wherever this kernel is used without causing re-definition errors.

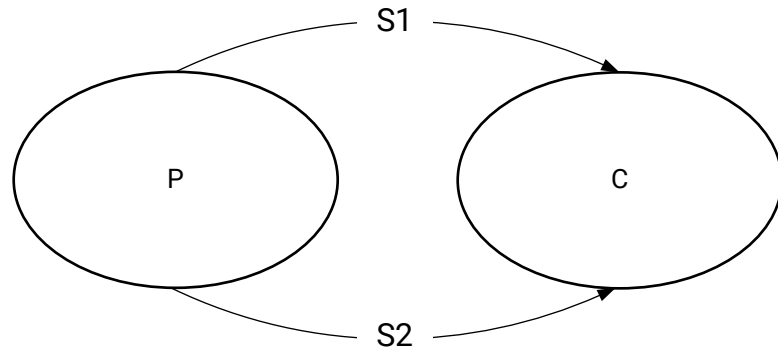
Note: Large look-up tables (>32 KB) are not supported.

Note: Shared data either has to be managed explicitly as run-time parameters or declared at the file scope, which is shared across all kernels mapped to the same AI Engine.

FIFO Depth

The AI Engine architecture uses stream data extensively for DMA-based I/O, for communicating between two AI Engines, and for communicating between the AI Engine and the programmable logic (PL). This raises the potential for a resource deadlock when the data flow graph has reconvergent data paths. If the pipeline depth of one path is longer than the other, the producer kernel can stall and might not be able to push data into the shorter path because of back pressure. At the same time, the consumer kernel is waiting to receive data on the longer path due to the lack of data. If the order of data production and consumption between two data paths is different, a deadlock can happen even between two kernels that are directly connected with two data paths. The following figure illustrates the paths.

Figure 5: Producer and Consumer Kernels with Reconvergent Streams



X20177-050818

If the producer kernel is trying to push data on stream S1 and runs into back pressure while the consumer kernel is still trying to read data from stream S2, a deadlock occurs. A general way to fix this situation is to create more buffering in the paths that have back pressure in the source code by using a `fifo_depth` constraint on a connection.

```
p = kernel::create(producer);
c = kernel::create(consumer);
connect<stream> s1(p.out[0], c.in[0]);
connect<stream> s2(p.out[1], c.in[1]);
fifo_depth(s1) = 20;
fifo_depth(s2) = 10;
```

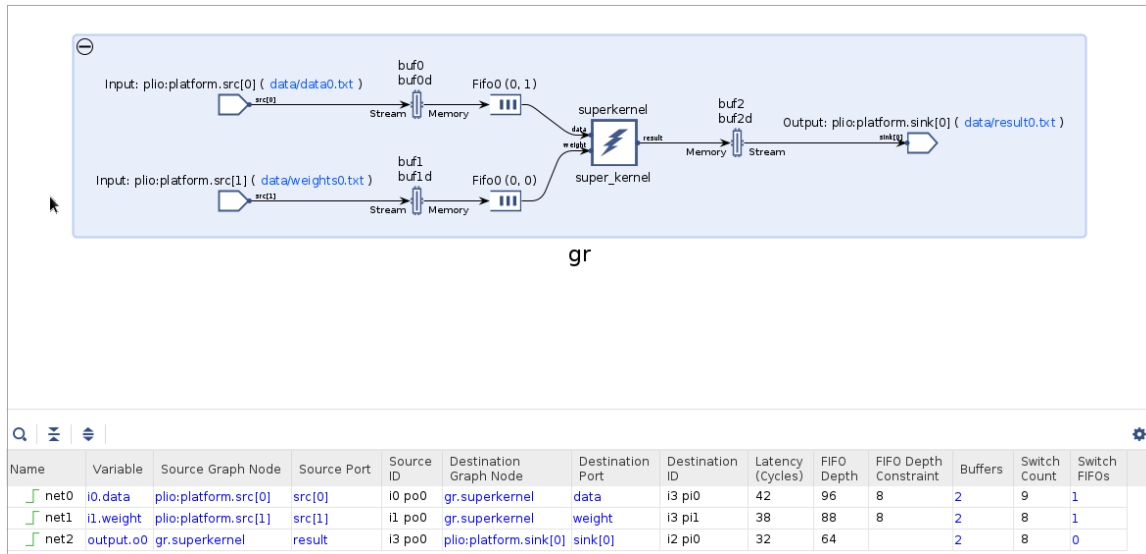
Note: The `fifo_depth()` constraint is only valid on stream and window type kernel connections. It is not available on cascade stream type connections, because there is a two deep, 384-bit wide FIFO on both the input and output cascade streams that allows storing up to four values between AI Engines.

Note: The maximum allowable FIFO depth value is 8188 32-bit words.

Stream Switch FIFO

The AI Engine has two 32-bit input AXI4-Stream interfaces and two 32-bit output AXI4-Stream interfaces. Each stream is connected to a FIFO both on the input and output side, allowing the AI Engine to have a four word (128-bit) access per four cycles, or a one word (32-bit) access per cycle on a stream. A `fifo_depth()` constraint specification below 40 allocates FIFOs from the stream switch. The following is an example of a FIFO allocation on the stream switch requesting a `fifo_depth(8)`.

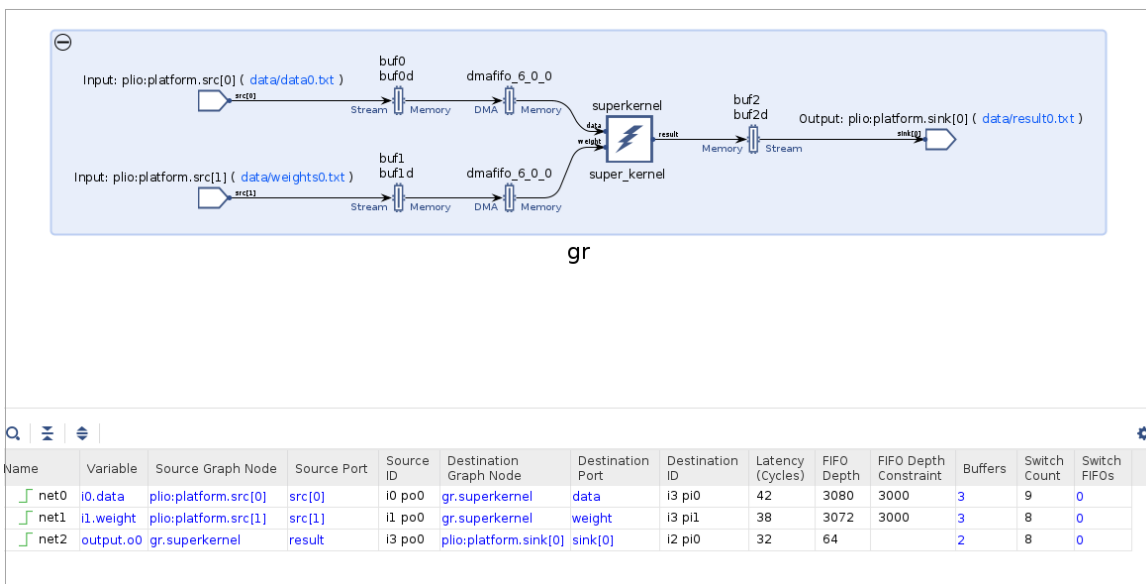
Figure 6: FIFO Allocation on Stream Switch



DMA FIFO

A `fifo_depth()` constraint specification above 40 allocates FIFOs from memory, known as DMA FIFOs. The following is an example of a FIFO allocation for a request of `fifo_depth(3000)` bytes which is allocated in memory.

Figure 7: DMA FIFO Allocation



Note: The maximum allowable FIFO depth value is 8188 32-bit words.

You can also specify the type of FIFO allocated, whether stream switch or DMA, as well as their locations. More information can be found in [FIFO Location Constraints](#).

AI Engine Tile DMA Performance

In high throughput use cases where the AI Engine and PL throughput is close to maximum, when using a DMA FIFO, and the PL communicates with the DMA FIFO in an asynchronous PL to AI Engine clock relationship, the read side must occasionally wait for data due to nature of a single DMA FIFO. This can lead to slightly lower than 100% throughput on the AI Engine. Some of the recommended ways to avoid the small loss in throughput are as follows.

- Choose a `fifo_depth` constraint of less than or up to 40 at the AI Engine-PL boundaries on streaming connections with a slack of 40 or less.
- Add a small asynchronous FIFO in the PL to shift the alignment into the AI Engine clock domain.
- Use a synchronous PL clock to the AI Engine. Use a 128-bit AXI4-Stream interface from the PL and use a PL clock at integer multiples of the AI Engine frequency.

Kernel Bypass

A bypass encapsulator construct discussed in [Run-Time Graph Reconfiguration Using Control Parameters](#) is used to execute a kernel conditionally. The control of the bypass is done through a run-time parameter: 0 for no bypass and 1 for bypass. In addition to the control parameter, the external connections of a bypassed kernel or a graph are directed to the external ports of the bypass construct itself. Internally, the bypass construct is connected to the bypassed kernel or the graph automatically by the compiler. The following example shows the required coding.

```
inout_port control;
bypass b;
kernel f, p, c;
f = kernel::create(filter);
...
b = bypass::create(f);
connect<parameter> (control, b.bp);
connect<window<128>> n1(p.out[0], b.in[0]);
connect<window<128>> n2(b.out[0], c.out[0]);
```

Note: For the bypass to work correctly, a one-to-one correspondence between the input and output data ports is required, both in type and size.

Explicit Packet Switching

Just as multiple AI Engine kernels can share a single processor and execute in a interleaved manner, multiple stream connections can be shared on a single physical channel. This mechanism is known as Packet Switching. The AI Engine architecture and compiler work together to provide a programming model where up to four stream connections can share the same physical channel.

The Explicit Packet Switching feature allows fine-grain control over how packets are generated, distributed, and consumed in a graph computation. Explicit Packet Switching is typically recommended in cases where many low bandwidth streams from common PL source can be distributed to different AI Engine destinations. Similarly many low bandwidth streams from different AI Engine sources to a common PL destination can also take advantage of this feature. Because a single physical channel is shared between multiple streams, you minimize the number of AI Engine - PL interface streams used. This section describes graph constructs to create packet-switched streams explicitly in the graph.

Packet Switching Graph Constructs

Packet-switched streams are essentially multiplexed data streams that carry different types of data at different times. Packet-switched streams do not provide deterministic latency due to the potential for resource contention with other packet-switched streams. The multiplexed data flows in units of packets with a 32-bit packet header and a variable number of payload words. A header word needs to be sent before the actual payload data and the TLAST signal is required on the last word of the packet. Two new data types called `input_pktstream` and `output_pktstream` are introduced to represent the multiplexed data streams as input to or output from a kernel, respectively. More details on the packet headers and data types can be found in [Packet Stream Operations](#).

Note: By convention, packets originating in the programmable logic are initialized with row, column to be -1,-1.

To explicitly control the multiplexing and de-multiplexing of packets, two new templated node classes, `pktsplit<n>` and `pktmerge<n>`, are added to the ADF graph library. A node instance of class `pktmerge<n>` is a n:1 multiplexer of n packet streams producing a single packet stream. A node instance of class `pktsplit<n>` is a 1:n de-multiplexer of a packet stream producing n different packet streams. The maximum number of allowable packet streams is 32 on a single physical channel ($n \leq 32$). See [Appendix A: Adaptive Data Flow Graph Specification Reference](#) for more details.

A kernel can receive packets of data either as windows of data or as `input_pktstream` and `output_pktstream`. To connect a packet stream to a window of data meant for an AI Engine kernel use the following graph construct.

```
connect<pktstream, window<32>>
```


To connect a window of data from an AI Engine kernel to a packet stream use the following graph construct.

```
connect<window<32>, pktstream>
```

To connect a packet split to an AI Engine kernel use the following graph construct.

```
connect<pktstream, pktstream>
```

To connect a packet stream from an AI Engine kernel to a packet merge use the following graph construct.

```
connect<pktstream, pktstream>
```

To connect a stream of data from/to a PLIO connection use the following graph construct.

```
connect<input_port, pktstream>
connect<pktstream, output_port>
```

If the source and destination are both packet streams, it can be specified using any one of the following graph constructs.

```
connect<pktstream>
or
connect<pktstream, pktstream>
```

When a kernel receives packets of data as a window of data, the header and TLAST are dropped prior to the kernel receiving the window of data. If the kernel writes an output window of data, the packet header and TLAST are automatically inserted.

However, if the kernel receives `input_pktstream` of data, the kernel needs to process the packer header and TLAST, in addition to the packet data. Similarly, if the kernel sends an `output_pktstream` of data, the kernel needs to insert the packer header and TLAST, in addition to the packet data into the output stream.

Note: If a kernel is receiving packet data as an input window of data you need to ensure the length of data per packet matches the window size.

These concepts are illustrated in the following example.

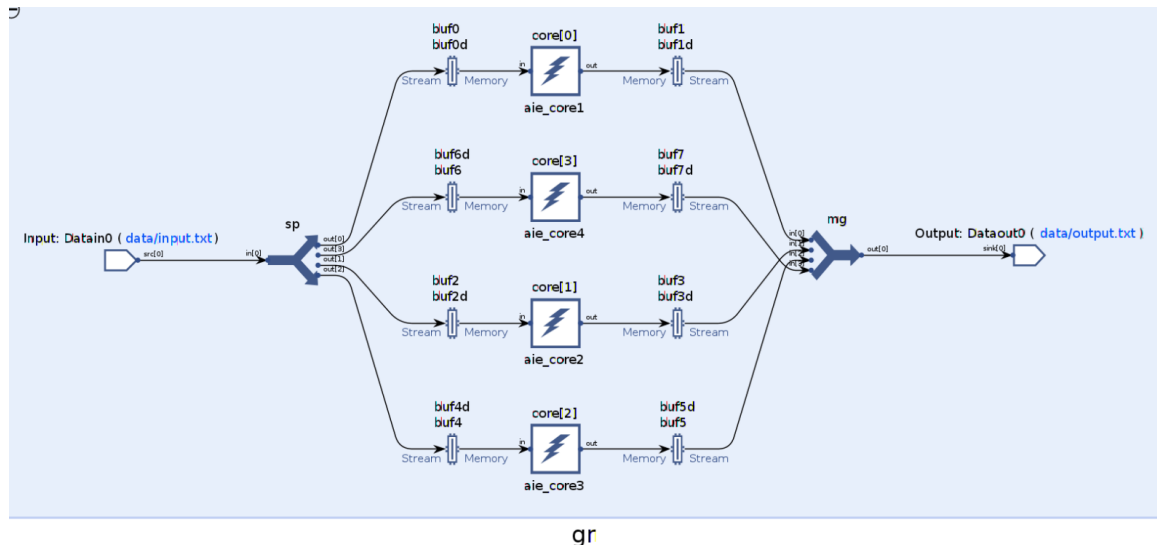
```
class ExplicitPacketSwitching: public adf::graph {
private:
    adf::kernel core[4];
    adf::pktsplit<4> sp;
    adf::pktmerge<4> mg;
public:
    adf::port in;
    adf::port out;
    mygraph() {
        core[0] = adf::kernel::create(aie_core1);
        core[1] = adf::kernel::create(aie_core2);
        core[2] = adf::kernel::create(aie_core3);
        core[3] = adf::kernel::create(aie_core4);

        adf::source(core[0]) = "aie_core1.cpp";
    }
};
```

```
adf::source(core[1]) = "aie_core2.cpp";
adf::source(core[2]) = "aie_core3.cpp";
adf::source(core[3]) = "aie_core4.cpp";
sp = adf::pktsplit<4>::create();
mg = adf::pktmerge<4>::create();
for(int i=0;i<4;i++){
    adf::runtime<ratio>(core[i]) = 0.9;
    adf::connect<adf::pktstream, adf::window<32> > (sp.out[i], core[i].in[0]);
    adf::connect<adf::window<32>, adf::pktstream > (core[i].out[0], mg.in[i]);
}
adf::connect<adf::pktstream> (in, sp.in[0]);
adf::connect<adf::pktstream> (mg.out[0], out);
};
```

The graph has one input PLIO port and one output PLIO port. The input packet stream from the PL is split four ways and input to four different AI Engine kernels. The output streams from the four AI Engine kernels are merged into one packet stream which is output to the PL. The Vitis analyzer Graph view of the code is shown as follows.

Figure 8: Graph View



One kernel code example is as follows.

```
const uint32 pktType=0;
void aie_core1(input_pktstream *in,output_pktstream *out){
    readincr(in);//read header and discard
    uint32 ID=getPacketid(out,0);//for output pktstream
    writeHeader(out,pktType,ID); //Generate header for output

    bool tlast;
    for(int i=0;i<8;i++){
        int32 tmp=readincr(in,tlast);
        tmp+=1;
        writeincr(out,tmp,i==7);//TLAST=1 for last word
    }
}
```

Note: input_pktstream is read as integer input.

Following is an example kernel code that accepts and transfers floating-point data type.

```
const uint32 pktType=0;
void aie_core1_float(input_pktstream *in,output_pktstream *out){
    readincr(in);//read header and discard
    uint32 ID=getPacketid(out,0);//for output pktstream
    writeHeader(out,pktType,ID); //Generate header for output

    bool tlast;
    for(int i=0;i<8;i++){
        int32 tmp=readincr(in,tlast);//read data as integer type
        float tmp_f=reinterpret_cast<float*>(tmp);//Reinterpret memory as float
        writeincr(out,tmp_f,i==7);//TLAST=1 for last word
    }
}
```

Packet Switching and the AI Engine Simulator

Explicit packet switching is supported by the AI Engine simulator. Consider the example of the previous graph that expects packet switched data from the PL; the data is split inside the AI Engine and sent to four AI Engine kernels. On the output side the four kernel outputs are merged into one output stream to the PL.

The input data file from the PL contains all the packet switched data from the PL, for the four AI Engine kernels in the previous example. It contains the data for different kernels, packet by packet. Each packet of data is for one window input for an AI Engine kernel. The data format is as follows.

```
2415853568
0
1
2
3
4
5
6
TLAST
7
```

Here, 2415853568 is 0x8fff0000 in hex format. The five least significant bits are the packet ID, 0 in this case. The last data in the packet has the keyword TLAST, which denotes the last data for the window input for the kernel.

Note: Integer values only are accepted as a packet header ID for the PL packet inputs to the AI Engine simulator.

You can construct the header for each packet manually, or write helper functions to generate the header. The AI Engine compiler generates a packet switching report file `Work/reports/packet_switching_report.json` that lists the packet IDs used in the graph. In addition it also generates `Work/temp/packet_ids.c.h` and `Work/temp/packet_ids.v.h` header files that can be included in your C or Verilog kernel code.

Location Constraints

Kernel Location Constraints

When building large graphs with multiple subgraphs, it is sometimes useful to control the exact mapping of kernels to AI Engines, either relative to other kernels or in an absolute sense. The AI Engine compiler provides a mechanism to specify location constraints for kernels, which when used with the C++ template class specification, provides a powerful mechanism to create a robust, scalable, and predictable mapping of your graph onto the AI Engine array. It also reduces the choices for the mapper to try, which can considerably speed up the mapper. Consider the following graph specification:

```
#include <adf.h>
#include "kernels.h"
#define NUMCORES (COLS*ROWS)
using namespace adf;

template <int COLS, int ROWS, int STARTCOL, int STARTROW>
class indep_nodes_graph1 : public graph {
public:
    kernel kr[NUMCORES];
    port<input> datain[NUMCORES] ;
    port<output> dataout[NUMCORES] ;

    indep_nodes_graph1() {
        for (int i = 0; i < COLS; i++) {
            for (int j = 0; j < ROWS; j++) {
                int k = i*ROWS + j;
                kr[k] = kernel::create(mykernel);
                source(kr[k]) = "kernels/kernel.cc";
                runtime<ratio>(kr[k]) = 0.9;
                location<kernel>(kr[k]) = tile(STARTCOL+i, STARTROW+j);
            }
        }
        for (int i = 0; i < NUMCORES; i++) {
            connect<stream, window<64>>(datain[i], kr[i].in[0]);
            connect<window<64>, stream>(kr[i].out[0], dataout[i]);
        }
    };
};
```

The template parameters identify a COLS x ROWS logical array of kernels (COLS x ROWS = NUMCORES) that are placed within a larger logical device of some dimensionality starting at (STARTCOL, STARTROW) as the origin. Each kernel in that graph is constrained to be placed on a specific AI Engine. This is accomplished using an absolute location constraint for each kernel placing it on a specific processor tile. For example, the following declaration would create a 1 x 2 kernel array starting at offset (3,2). When embedded within a 4 x 4 logical device topology, the kernel array is constrained to the top right corner.

```
indep_nodes_graph1<1,2,3,2> mygraph;
```



IMPORTANT! Earlier releases used `location<absolute>(k)`, function to specify kernel constraints and `proc(x, y)` function to specify a processor tile location. These functions are now deprecated. Instead, use `location<kernel>(k)` to specify the kernel constraints and `tile(x, y)` to identify a specific tile location. See [Appendix A: Adaptive Data Flow Graph Specification Reference](#) for more information.

Buffer Location Constraints

The AI Engine compiler tries to automatically allocate buffers for windows, lookup tables, and run-time parameters in the most efficient manner possible. However, you might want to explicitly control their placement in memory. Similar to the kernels shown previously in this section, buffers inferred on a kernel port can also be constrained to be mapped to specific tiles, banks, or even address offsets using location constraints, as shown in the following example.

```
#include <adf.h>
#include "kernels.h"
#define NUMCORES (COLS*ROWS)
using namespace adf;

template <int COLS, int ROWS, int STARTCOL, int STARTROW>
class indep_nodes_graph2 : public graph {
public:
    kernel kr[NUMCORES];
    port<input> datain[NUMCORES] ;
    port<output> dataout[NUMCORES] ;

    indep_nodes_graph() {
        for (int i = 0; i < COLS; i++) {
            for (int j = 0; j < ROWS; j++) {
                int k = i*ROWS + j;
                kr[k] = kernel::create(mykernel);
                source(kr[k]) = "kernels/kernel.cc";
                runtime<ratio>(kr[k]) = 0.9;
                location<kernel>(kr[k]) = tile(STARTCOL+i, STARTROW+j); // kernel
location
                location<buffer>(kr[k].in[0]) =
                    { address(STARTCOL+i, STARTROW+j, 0x0),
                      address(STARTCOL+i, STARTROW+j, 0x2000) }; // double
buffer location
                location<stack>(kr[k]) = bank(STARTCOL+i, STARTROW+j, 2); // stack
location
                location<buffer>(kr[k].out[0]) = location<kernel>(kr[k]); // relative
buffer location
            }
        }

        for (int i = 0; i < NUMCORES; i++) {
            connect< stream, window<64> >(datain[i], kr[i].in[0]);
            connect< window<64>, stream >(kr[i].out[0], dataout[i]);
        }
    };
};
```

In the previous code, the location of double buffers at port `kr[k].in[0]` is constrained to the specific memory tile address offsets that are created using the `address(col, row, offset)` constructor. Furthermore, the location of the system memory (including the sync buffer, stack and static heap) for the processor that executes kernel instance `kr[k]` is constrained to a particular bank using the `bank(col, row, bankid)` constructor. Finally, the tile location of the buffers connected to the port `kr[k].out[0]` is constrained to be the same tile as that of the kernel instance `kr[k]`. Buffer location constraints are only allowed on window kernel ports.



IMPORTANT! Using location constraint constructors and equality relations between them, you can make fine-grain mapping decisions that the compiler must honor. However, you must be careful because it is easy to create constraints that are impossible for the compiler to satisfy. For example, the compiler cannot allow two buffers to be mapped to the same address offset. See the complete reference in [Appendix A: Adaptive Data Flow Graph Specification Reference](#).

FIFO Location Constraints

The AI Engine compiler tries to automatically allocate FIFOs in the most efficient manner possible. However, you might want to explicitly control their placement in memory, as shown in the following example. This constraint is useful to preserve the placement of FIFO resources between runs of the AI Engine compiler.

Note the following considerations for FIFO constraints.

- If FIFO constraints are used, the entire depth of the FIFO must be constrained. It is not possible to constrain a portion of the FIFO and leave the rest for the compiler to add.
- If FIFO constraints are added to branching nets, the FIFO constraint should be added to each point-to-point net. If you want to share stream switch FIFOs or DMA FIFOs before the branch, this can be achieved by duplicating the FIFO type and location on each point-to-point net.
- The constraint can be used without a location to specify the desired type of FIFO without specifying a location or depth.

The following example shows how a FIFO constraint can be used in a graph file.

```
two_node_graph() {
    loop0 = kernel::create(loopback_stream);
    loop1 = kernel::create(loopback_stream);
    loop2 = kernel::create(loopback_stream);

    source(loop0) = "loopback_stream.cc";
    source(loop1) = "loopback_stream.cc";
    source(loop2) = "loopback_stream.cc";

    connect< stream > net0 (in0, loop0.in[0]);
    connect< stream > net1 (loop0.out[0], loop1.in[0]);
    connect< stream > net2 (loop1.out[0], loop2.in[0]);
    connect< stream > net3 (loop2.out[0], out0);

    runtime(loop0) = 0.9;
    runtime(loop1) = 0.9;
    runtime(loop2) = 0.9;
}
```

```
fifo_depth(net1) = 32;
fifo_depth(net2) = 48;

location< fifo >(net1) = {ss_fifo(shim_tile, 16 , 0, 1), ss_fifo(shim_tile,17,0,0)};
location< fifo >(net2) = dma_fifo(aie_tile, 8, 0, 0x0000, 48);
};
```

The second example shows how a FIFO constraint can be added to a constraints file.

```
{
  "PortConstraints": {
    "fifo_locations_records": {
      "dma_fifos": {
        "r1": {
          "tile_type": "core",
          "row": 0,
          "column": 0,
          "size": 16,
          "offset": 8,
          "bankId": 2
        },
        "r2": {
          "tile_type": "core",
          "row": 0,
          "column": 1,
          "size": 16,
          "offset": 9
        },
        "r4": {
          "tile_type": "mem",
          "row": 2,
          "column": 4,
          "size": 16,
          "offset": 6,
          "bankId": 2
        }
      },
      "stream_fifos": {
        "r3": {
          "tile_type": "shim",
          "row": 1,
          "column": 3,
          "channel": 1
        }
      }
    },
    "mygraph.k2.in[0]": {
      "fifo_locations": ["r1", "r2", "r3"]
    },
    "mygraph.k4.in[0]": {
      "fifo_locations": ["r1", "r2", "r4"]
    }
  }
}
```

Area Location Constraints

Area location constraints direct the compiler to contain nodes to a custom location in the array. Properties to specify on an area group are described in the following table.

Table 10: Area Group Properties

Property	Description
<code>group</code>	Specify the collection of group. Each group can be: <ul style="list-style-type: none"> tile-type: Specify the tile-type for the group. Supported tile-types are <code>aie_tile</code>, <code>shim_tile</code>, or <code>memory_tile</code>. Currently only one rectangle for each type is supported. column_min: Column index for lower left corner of the group. row_min: Row index for lower left corner of the group. column_max: Column index for upper right corner of the group. row_max: Row index for upper right corner of the group.
<code>contain_routing</code>	A boolean value that when specified true ensures all routing, including nets between nodes contained in the nodeGroup, is contained within the area group.
<code>exclusive_routing</code>	A boolean value that when specified true ensures all routing, excluding nets between nodes from the nodeGroup, is excluded from the area group.
<code>exclusive_placement</code>	A boolean value that when specified true prevents all nodes not included in the nodeGroup from being placed within the area group bounding box.

The following examples show how an area location constraint can be applied in a graph file.

```
using namespace adf;

class testGraph1: public adf::graph {

private:
    kernel first;
    kernel second;
public:
    testGraph1() {
        first = kernel::create(simple1);
        second = kernel::create(simple2);
        connect< window<128> > net1 (first.out[0], second.in[0]);
        source(first) = "src/kernels/kernels.cc";
        source(second) = "src/kernels/kernels.cc";
        runtime<ratio>(first) = 0.1;
        runtime<ratio>(second) = 0.1;

        // Create area group with some valid ranges.
        location<graph>(*this) = area_group({{aie_tile, 0, 0, 1, 7},
{shim_tile, 0, 0, 1, 0}});
    }

};
```

```
using namespace adf;

class testGraph2: public adf::graph {

private:
    kernel first;
    kernel second;
public:
    testGraph2() {
        first = kernel::create(simple1);
        second = kernel::create(simple2);
        connect< window<128> > net1 (first.out[0], second.in[0]);
        source(first) = "src/kernels/kernels.cc";
        source(second) = "src/kernels/kernels.cc";
        runtime<ratio>(first) = 0.1;
        runtime<ratio>(second) = 0.1;

        // Explicitly specify contain_routing, exclusive_routing,
exclusive_placement.
        location<graph>(*this) = area_group({{aie_tile, 0, 0, 1, 7},
{shim_tile, 0, 0, 1, 0}}, true, false, true);
    }

};
```

The following table clarifies whether the nodes and nets can access the resources of the area group given various combinations of the flags. The use cases presented in the table as columns are illustrated in the figure that follows. Two important things to consider when applying the rules from the following table.

- Broadcast nets that are driven from one node to several destination nodes, are considered as individual point-to-point nets

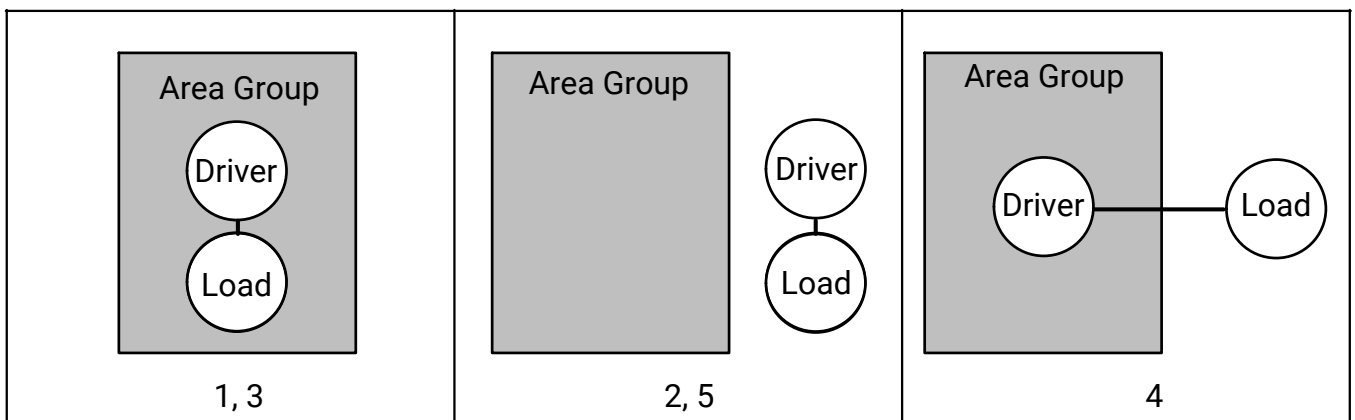
- Any FIFOs on a net (stream switch or DMA) adhere to the same conditions as the net. For example, if a net is fully contained in an area group (both driver and receiver are contained in the area group) and the `contain_routing` flag is used, then the following table indicates that the net routing must be fully contained in the area group. Similarly, any FIFOs on that net, must also be placed within the boundary of the area group.

Table 11: Permission to use Area Group Resources with the Stated Conditions

<code>contain_routing</code>	<code>exclusive_routing</code>	<code>exclusive_placement</code>	Placement of Nodes Contained in the Area Group (1)	Placement of Nodes External to the Area Group (2)	Routes between Nodes Fully Contained in the Area group (3)	Routes between Nodes Spanning the Area Group (4)	Routes between Nodes Entirely External to the Area Group (5)
False	False	False	Must	May	May	May	May
False	False	True	Must	Must Not	May	May	May
False	True	False	Must	May	May	May	Must Not
False	True	True	Must	Must Not	May	May	Must Not
True	False	False	Must	May	Must	May	May
True	False	True	Must	Must Not	Must	May	May
True	True	False	Must	May	Must	May	Must Not
True	True	True	Must	Must Not	Must	May	Must Not

An illustration of the use cases is shown in the following figure.

Figure 9: Use Cases



X25720-090921

Hierarchical Constraints

When creating complex graphs with multiple subgraph classes, or multiple instances of the same subgraph class, the location constraints described previously can also be applied to each kernel instance or kernel port instance individually at the point of subgraph instantiation instead of the definition. In this case, you need to specify the graph qualified name of that kernel instance or kernel port instance in the constraint as shown in the following example. Also, make sure that the kernels or their ports being constrained as described previously are defined to be public members of the subgraph.

```
class ToplevelGraph : public graph {
public:
    indep_nodes_graph1<1,2,3,2> mygraph;
    port<input> datain[2] ;
    port<output> dataout[2] ;

    ToplevelGraph() {
        for (int i = 0; i < 2; i++) {
            connect<stream, window<64> >>(datain[i], mygraph.datain[i]);
            connect<window<64>, stream >>(mygraph.dataout[i], dataout[i]);

            // hierarchical constraints
            location<stack>(mygraph.kr[i]) = bank(3, 2+i, 2);
            location<buffer>(mygraph.kr[i].out[0]) =
location<kernel>(mygraph.kr[i]);
        }
    };
};
```

Note: You can recirculate the previous design placement in your next compilation. This significantly reduces the mapper run time. When the compiler runs, it generates a placement constraints file in the `Work` directory. This constraint file can be specified on the command line for the next iteration.

```
aiecompiler --constraints Work/temp/graph_aie_mapped.aiecst src/graph.cpp
```

To reuse FIFO location constraints, use `aie_routed.aiecst` file.

```
aiecompiler --constraints Work/temp/graph_aie_routed.aiecst src/graph.cpp
```

Buffer Allocation Control

The AI Engine compiler automatically allocates the desired number of buffers for each memory connection. There are several different cases.

- Lookup tables are always allocated as single buffers because they are expected to be read-only and private to a kernel. No locks are needed to synchronize lookup table accesses because they are expected to be accessed in an exclusive manner.
- Window connections are usually assigned double buffers if the producer and consumer kernels are mapped to different processors or if the producer or the consumer is a DMA. This enables the two agents to operate in a pipelined manner using ping-pong synchronization with two locks. The AI Engine compiler automatically generates this synchronization in the respective processor `main` functions.
- If the producer and consumer kernels are mapped to the same processor, then the window connection is given only one buffer and no lock synchronization is needed because the kernels are executed sequentially.
- Run-time parameter connections can be assigned double buffers (default) along with a selector word to choose the next buffer to be accessed.

Run-time parameter connections can also be assigned single buffers. Sometimes, with window connections, it is desirable to use only single buffer synchronization instead of double buffers. This is useful when the local data memory is at a premium and the performance penalty of using a single buffer for data transfer is not critical. This can be achieved using the `single_buffer(port<T>&)` constraint.

```
single_buffer(first.in[0]); //For window input or RTP input
single_buffer(first.inout[0]); //For RTP output
```

C++ Kernel Class Support

The AI Engine compiler supports C++ kernel classes. The following example shows how to set filter coefficients and the number of samples of a FIR filter class through a constructor. The C++ kernel class allows internal states for each kernel instance to be encapsulated within the corresponding class object. In the following code, you can see an example of this where the filter coefficients (`coeffs`) are specified through the constructor. This resolves the problem of using file scope variable, global variable, or static function scope variable to store the internal states of a C function kernel. When multiple instances of such a kernel are mapped to the same core, the internal state variables are shared across multiple instances and cause conflicts.

```
//fir.h
#pragma once
#include "adf.h"
#define NUM_COEFFS 12

class FIR
{
private:
    int32 coeffs[NUM_COEFFS];
    int32 tapDelayLine[NUM_COEFFS];
    uint32 numSamples;

public:
    FIR(const int32(&coefficients)[NUM_COEFFS], uint32 samples);
    void filter(input_window_int32* in, output_window_int32* out);
    static void registerKernelClass()
    {
        REGISTER_FUNCTION(FIR::filter);
    }
};
```

You are required to write the static void `registerKernelClass()` method in the header file. Inside the `registerKernelClass()` method, you need to call the `REGISTER_FUNCTION` macro. This macro is used to register the class `run` method to be executed on the AI Engine core to perform the kernel functionality. In the preceding example `FIR::filter` is registered using this macro. The kernel class constructor and run method should be implemented inside a separate source file. The implementation of a `run` method of a kernel class is the same as writing a kernel function described in previous chapters.

```
//fir.cpp
//implementation in this example is not optimized and is for illustration
//purpose
#include "fir.h"

FIR::FIR(const int32(&coefficients)[NUM_COEFFS], uint32 samples)
{
    for (int i = 0; i < NUM_COEFFS; i++)
        coeffs[i] = coefficients[i];

    for (int i = 0; i < NUM_COEFFS; i++)
        tapDelayLine[i] = 0;
```

```

        numSamples = samples;
    }

void FIR::filter(input_window_int32* in, output_window_int32* out)
{
    for (int i = 0; i < numSamples; i++)
    {
        for (int j = NUM_COEFFS-1; j > 0; j--)
            tapDelayLine[j] = tapDelayLine[j - 1];

        tapDelayLine[0] = window_readincr(in);

        int32 y = 0;
        for (int j = 0; j < NUM_COEFFS; j++)
        {
            y += coeffs[j] * tapDelayLine[j];
        }

        window_writeincr(out, y);
    }
}

```

```

//graph.h
#pragma once
#include "adf.h"
#include "fir.h"
using namespace adf;

class mygraph : public graph
{
public:
    input_port in1, in2;
    output_port out1, out2;
    kernel k1, k2;

    mygraph()
    {
        //see lab8.3 for narrow filter coefficients
        k1 = kernel::create_object<FIR>(std::vector<int>({ 180, 89, -80,
-391, -720, -834, -478, 505, 2063, 3896, 5535, 6504 })), 8);
        runtime<ratio>(k1) = 0.1;
        source(k1) = "src/fir.cpp";

        //see lab8.3 for wide filter coefficients
        k2 = kernel::create_object<FIR>(std::vector<int>({ -21, -249, 319,
-78, -511, 977, -610, -844, 2574, -2754, -1066, 18539 })), 8);
        runtime<ratio>(k2) = 0.1;
        source(k2) = "src/fir.cpp";

        connect<window<32>>(in1, k1.in[0]);
        connect<window<32>>(in2, k2.in[0]);

        connect<window<32>>(k1.out[0], out1);
        connect<window<32>>(k2.out[0], out2);
    }
};

```

For a kernel class with a non-default constructor, you can specify the constructor parameter values in the arguments of `kernel::create_object`, when creating a representation of a kernel instance. In the previous example, two FIR filter kernels (`k1` and `k2`) are created using `kernel::create_object<FIR>`. `k1` has filter coefficients { 180, 89, -80, -391, -720, -834, -478, 505, 2063, 3896, 5535, 6504 } and `k2` has filter coefficients { -21, -249, 319, -78, -511, 977, -610, -844, 2574, -2754, -1066, 18539 }. Both of them consume eight samples for each invocation.

The following code shows the AI Engine compiler generated program. The two FIR kernel objects are instantiated with the proper constructor parameters.

```
//Work/aie/x_y/src/x_y.cc
...
FIR i4({180, 89, -80, -391, -720, -834, -478, 505, 2063, 3896, 5535, 6504},
8);
FIR i5({-21, -249, 319, -78, -511, 977, -610, -844, 2574, -2754, -1066,
18539}, 8);

int main(void) {
    ...
    // Kernel call : i4:filter

    i4.filter(get_input_window_int32(window_buf0_buf0d), get_output_window_int32(
window_buf2_buf2d));
    ...
    // Kernel call : i5:filter

    i5.filter(get_input_window_int32(window_buf1_buf1d), get_output_window_int32(
window_buf3_buf3d));
    ...
}
```

A kernel class can have a member variable occupying a significant amount of memory space that might not fit into program memory. The location of the kernel class member variable can be controlled. The AI Engine compiler supports `array reference` member variables that allow the compiler to allocate or constrain the memory space while passing the reference to the object.

```
//fir.h
#pragma once
#include "adf.h"
#define NUM_COEFFS 12

class FIR
{
private:
    int32 (&coeffs)[NUM_COEFFS];
    int32 tapDelayLine[NUM_COEFFS];
    uint32 numSamples;

public:
    FIR(int32(&coefficients)[NUM_COEFFS], uint32 samples);
    void filter(input_window_int32* in, output_window_int32* out);
    static void registerKernelClass()
```

```

    {
        REGISTER_FUNCTION(FIR::filter);
        REGISTER_PARAMETER(coeffs);
    }
};

//fir.cpp
#include "fir.h"
FIR::FIR(int32(&coefficients)[NUM_COEFFS], uint32 samples)
    : coeffs(coefficients)
{
    for (int i = 0; i < NUM_COEFFS; i++)
        tapDelayLine[i] = 0;

    numSamples = samples;
}

void FIR::filter(input_window_int32* in, output_window_int32* out)
{
    ...
}

```

The previous example shows a slightly modified version of the FIR kernel class. Here, member variable `coeffs` is a `int32 (&)[NUM_COEFFS]` data type. The constructor initializer `coeffs(coefficients)` initializes `coeffs` to the reference to an array allocated externally to the class object. To let the AI Engine compiler know that the `coeffs` member variable is intended to be allocated by the compiler, you must use `REGISTER_PARAMETER` to register an array reference member variable inside the `registerKernelClass`.

The use of `kernel::create_object` to create a representation of a FIR kernel instance and to specify the initial value of the constructor parameters is the same as in the previous example. See the following code.

```

//graph.h
...
class mygraph : public graph
{
    ...
    mygraph()
    {
        k1 = kernel::create_object<FIR>(std::vector<int>({ 180, 89, -80,
-391, -720, -834, -478, 505, 2063, 3896, 5535, 6504 })), 8);
        ...
        k2 = kernel::create_object<FIR>(std::vector<int>({ -21, -249, 319,
-78, -511, 977, -610, -844, 2574, -2754, -1066, 18539 })), 8);
        ...
    }
};

```


The following code shows the corresponding AI Engine compiler generated program. The memory spaces for `int32 i4_coeffs[12]` and `int32 i5_coeffs[15]` are outside the kernel object instances and are passed into the FIR objects by reference.

```
//Work/aie/x_y/src/x_y.cc
int32 i4_coeffs[12] = {180, 89, -80, -391, -720, -834, -478, 505, 2063,
3896, 5535, 6504};
FIR i4(i4_coeffs, 8);
int32 i5_coeffs[15] = {-21, -249, 319, -78, -511, 977, -610, -844, 2574,
-2754, -1066, 18539};
FIR i5(i5_coeffs, 8);

int main(void) {
    ...
    // Kernel call : i4:filter

    i4.filter(get_input_window_int32(window_buf0_buf0d),get_output_window_int32(
window_buf2_buf2d));
    ...
    // Kernel call : i5:filter

    i5.filter(get_input_window_int32(window_buf1_buf1d),get_output_window_int32(
window_buf3_buf3d));
    ...
}
```

Because the memory space for an array reference member variable is allocated by the AI Engine compiler, the location constraint can be applied to constrain the memory location of these arrays, as shown in the following example code. The `REGISTER_PARAMETER` macro allows `kernel::create_object` to create a parameter handle for an array reference member variable, like `k1.param[0]` and `k2.param[0]`, and the `location<parameter>` constraint can be applied.


```
//graph.h
...
class mygraph : public graph
{
    ...
    mygraph()
    {
        k1 = kernel::create_object<FIR>(std::vector<int>({ 180, 89, -80,
-391, -720, -834, -478, 505, 2063, 3896, 5535, 6504 })), 8);
        ...
        k2 = kernel::create_object<FIR>(std::vector<int>({ -21, -249, 319,
-78, -511, 977, -610, -844, 2574, -2754, -1066, 18539 })), 8);
        ...

        location<parameter>(k1.param[0]) = address(...);
        location<parameter>(k2.param[0]) = bank(...);
    }
};
```

The C++ kernel class header files and the C++ kernel function template (see [C++ Template Support](#)) should not contain single-core specific intrinsic APIs and pragmas. This is the same programming guideline as writing regular C function kernels. This is because these header files are included in the graph header file and can be cross-compiled as part of the PS program. The Arm® cross-compiler cannot understand single-core intrinsic APIs or pragmas. Single-core specific programming content must be kept inside the source files.

C++ Template Support

A template is a powerful tool in C++. By passing the data type as a parameter, you eliminate the need to rewrite code to support different data types. Templates are expanded at compile time, like macros. The difference is that the compiler performs type checking before template expansion. The source code contains template functions and class definitions, but the compiled code can contain multiple copies of same function or class. Type parameters, non-type parameters, default arguments, scalar parameters, and template parameters can be passed to a template, where the compiler instantiates the function or class accordingly.

- Support for general C++ template features.
 - Supported data types (T) and connection types between kernels:
 - Data type (T): `int8`, `uint8`, `int16`, `uint16`, `cint16`, `int32`, `uint32`, `cint32`, `int64`, `uint64`, `float`, `cfloat`
-
-  **IMPORTANT!** *`acc48` and `cacc48` data types are not supported in template stream connections.*
-
- Function parameter type: `input_window<T>`, `output_window<T>`, `input_stream<T>`, `output_stream<T>`
 - The compiler does not support pre-compiled headers for template kernels.

Function Templates

Function template source code defines a generic function that can be used for different data types. Example function template:

```
// add.h
template<typename ELEMENT_TYPE, int FACTOR, size_t NUM_SAMPLES> void
add(input_window<ELEMENT_TYPE>* in,
    output_window<ELEMENT_TYPE>* out);
```

```
// add.cpp
template<typename ELEMENT_TYPE, int FACTOR, size_t NUM_SAMPLES> void
add(input_window<ELEMENT_TYPE>* in,
    output_window<ELEMENT_TYPE>* out)
{
    for (int i=0; i<NUM_SAMPLES; i++)
    {
```

```

        ELEMENT_TYPE value = window_readincr(in);
        value += FACTOR;
        window_writeincr(out, value);
    }
}

```

```

// graph.h
mygraph()
{
    k[0] = kernel::create(add<int32, 6, 8>);
    k[1] = kernel::create(add<int16, 3, 8>);
    for (int i=0; i<NUM_KERNELS; i++)
    {
        runtime<ratio>(k[i]) = 0.3;
        source(k[i]) = "src/add.cpp";
    }

    connect<window<32>>(in[0], k[0].in[0]);
    connect<window<32>>(k[0].out[0], out[0]);

    connect<window<16>>(in[1], k[1].in[0]);
    connect<window<16>>(k[1].out[0], out[1]);
}

```

where:

- `add.h` defines a template `add()` function.
- `add.cpp` defines the code for the template `add()` function.
- `graph.h` uses the template `add()` function within `mygraph` class.

Class Templates

Like function templates, class templates are useful when a class defines an object that is independent of a specific data type. Example class template:

```

// fir.h
...
template<size_t NUM_COEFFS, typename ELEMENT_TYPE> class FIR
{
private:
    ELEMENT_TYPE (&coeffs)[NUM_COEFFS];
    ELEMENT_TYPE tapDelayLine[NUM_COEFFS];
    uint32 numSamples;

public:
    FIR(ELEMENT_TYPE(&coefficients)[NUM_COEFFS], uint32 samples);

    void filter(input_window<ELEMENT_TYPE>* in,
output_window<ELEMENT_TYPE>* out);

    //user needs to write this function to register necessary info
    static void registerKernelClass()

```

```
{
    REGISTER_FUNCTION(FIR::filter);
    REGISTER_PARAMETER(coeffs);
}
};
```

```
// fir.cpp
...
template<size_t NUM_COEFFS, typename ELEMENT_TYPE> FIR<NUM_COEFFS,
ELEMENT_TYPE>::FIR(ELEMENT_TYPE(&coefficients)[NUM_COEFFS], uint32
samples):coeffs(coefficients)
{
    ...
}

template<size_t NUM_COEFFS, typename ELEMENT_TYPE> void FIR<NUM_COEFFS,
ELEMENT_TYPE>::filter(input_window<ELEMENT_TYPE>* in,
output_window<ELEMENT_TYPE>* out)
{
    ...
}
```

```
// graph.h
...
mygraph()
{
    k1 = kernel::create_object<FIR<12, int32>>(std::vector<int>({ 180, 89,
-80, -391, -720, -834, -478, 505, 2063, 3896, 5535, 6504 }), 8);
    runtime<ratio>(k1) = 0.1;
    source(k1) = "src/fir.cpp";
    headers(k1) = { "src/fir.h" };

    k2 = kernel::create_object<FIR<15, int32>>(std::vector<int>({ -21,
-249, 319, -78, -511, 977, -610, -844, 2574, -2754, -1066, 18539, 0, 0,
0 }), 8);
    runtime<ratio>(k2) = 0.1;
    source(k2) = "src/fir.cpp";
    headers(k2) = { "src/fir.h" };
    ...
}
```

where:

- `fir.h` defines a class template where class `FIR` is declared.
- `fir.cpp` contains class `FIR` implementation and the class `FIR` member function `filter` implementation.
- `graph.h` demonstrates the template class `FIR` instantiation within the `mygraph` class.

Multicast Support

Various multicast scenarios are supported in the graph, such as from a window to multiple windows, from stream to multiple streams, from PLIO to multiple windows etc. This section lists the supported types of multicast from a single source to multiple destinations. For additional details on PLIO, FileIO, and GMIO, see [Chapter 8: Using a Virtual Platform](#).

Table 12: Multicast Support Scenarios

#	Source	Destination 1	Destination 2	Supported
1	AI Engine Window	AI Engine Window	AI Engine Window	Not Supported
2	AI Engine Window	AI Engine Window	AI Engine Stream	Not Supported
3	AI Engine Window	AI Engine Window	PLIO/FileIO/GMIO	Not Supported
4	AI Engine Window	AI Engine Stream	AI Engine Stream	Supported
5	AI Engine Window	AI Engine Stream	PLIO/FileIO	Supported
6	AI Engine window	PLIO/FileIO	PLIO/FileIO	Supported
7	AI Engine Stream	AI Engine Window	AI Engine window	Supported
8	AI Engine Stream	AI Engine Window	AI Engine Stream	Supported
9	AI Engine Stream	AI Engine Window	PLIO/FileIO/GMIO	Supported
10	AI Engine Stream	AI Engine Stream	AI Engine Stream	Supported
11	AI Engine Stream	AI Engine Stream	PLIO/FileIO/GMIO	Supported
12	AI Engine Stream	PLIO/FileIO/GMIO	PLIO/FileIO/GMIO	Supported
13	PLIO/FileIO/GMIO	AI Engine Window	AI Engine window	Supported
14	PLIO/FileIO/GMIO	AI Engine Window	AI Engine Stream	Not Supported
15	PLIO/FileIO/GMIO	AI Engine Window	PLIO/FileIO/GMIO	Not Supported
16	PLIO/FileIO/GMIO	AI Engine Stream	AI Engine Stream	Supported
17	PLIO/FileIO/GMIO	AI Engine Stream	PLIO/FileIO/GMIO	Not Supported
18	PLIO/FileIO/GMIO	PLIO/FileIO/GMIO	PLIO/FileIO/GMIO	Not Supported
19	AI Engine Window	AI Engine Stream	GMIO	Not Supported
20	AI Engine Window	GMIO	GMIO	Not Supported
21	AI Engine Window	PLIO/FileIO	GMIO	Not Supported

Note the following.

- All source and destination windows in the multicast connections are required to have the same size.
- RTP and packet switching are not covered in this section.
- If the multicast type is supported, the destination number is not limited if it can fit into the hardware.

When multiple streams are connected to the same source, the data is sent to all the destination ports at the same time and is only sent when all destinations are ready to receive data. This might cause stream stall or design hang if the FIFO depth of the stream connections are not deep enough. Refer to the examples in *AI Engine Kernel Coding Best Practices Guide* ([UG1079](#)) for more information about the stream stalls and potential solutions.

AI Engine/Programmable Logic Integration

When you are ready to consider interfacing to the programmable logic (PL), you need to make a decision on the platform you want to interface with. A platform is a fully contained image that defines both the hardware (XSA) as well as the software (bare metal, Linux, or both). The XSA contains the hardware description of the platform, which is defined in the Vivado® Design Suite, and the software is defined with the use of a bare-metal setup, or a Linux image defined through PetaLinux. Depending on the needs of your application you might decide to use an example reference platform provided by Xilinx, or a custom platform created by your organization.

Xilinx recommends interfacing to the PLIO port attributes which represent external stream connections that cross the AI Engine-PL boundary. PLIO represents an ADF graph interface to the PL. This PL could be, for example, a PL kernel, a platform IP representing a signal source or sink, or it could be a data mover to interface the ADF graph to memory.

Alternatively interface connections can also be GMIO port attributes which represent external memory-mapped connections to or from the global memory. Further details on these attributes can be found in [Chapter 8: Using a Virtual Platform](#).

Note: Graphs with PL-only kernels is deprecated.

Design Flow Using RTL Programmable Logic

RTL blocks are not supported inside the ADF graph. Communication between the RTL blocks and the ADF graph requires that you use PLIO interfacing. In the following example, `interpolator` and `classify` are AI Engine kernels. The `interpolator` AI Engine kernel streams data to a PL RTL block, which, in turn, streams data back to the AI Engine `classify` kernel.

```
class clipped : public graph {
private:
    kernel interpolator;
    kernel classify;

public:
    port<input> in;
    port<output> clip_in;
    port<output> out;
    port<input> clip_out;
```

```

clipped() {
    interpolator = kernel::create(fir_27t_sym_hb_2i);
    classify      = kernel::create(classifier);

    connect< window<INTERPOLATOR27_INPUT_BLOCK_SIZE,
INTERPOLATOR27_INPUT_MARGIN> >(in, interpolator.in[0]);
    connect< window<POLAR_CLIP_INPUT_BLOCK_SIZE>, stream
>(interpolator.out[0], clip_in);
    connect< stream >(clip_out, classify.in[0]);
    connect< window<CLASSIFIER_OUTPUT_BLOCK_SIZE> >(classify.out[0], out);

    std::vector<std::string> myheaders;
    myheaders.push_back("include.h");

    adf::headers(interpolator) = myheaders;
    adf::headers(classify) = myheaders;

    source(interpolator) = "kernels/interpolators/hb27_2i.cc";
    source(classify)      = "kernels/classifiers/classify.cc";

    runtime<ratio>(interpolator) = 0.8;
    runtime<ratio>(classify) = 0.8;
};
};

```

`clip_in` and `clip_out` are ports to and from the `polar_clip` PL RTL kernel which is connected to the AI Engine kernels in the graph. For example, the `clip_in` port is the output of the `interpolator` AI Engine kernel that is connected to the input of the `polar_clip` RTL kernel. The `clip_out` port is the input of the `classify` AI Engine kernel and the output of the `polar_clip` RTL kernel.

RTL Blocks and AI Engine Simulator

The top-level application file that contains an instance of your graph class and connects the graph to a simulation platform, also needs to include the PLIO inputs and outputs of the RTL blocks. These files are called `output_interp.txt` and `input_classify.txt` in the following example.

```

#include "graph.h"

PLIO *in0 = new PLIO("DataIn1", adf::plio_32_bits, "data/input.txt");
PLIO *ai_to_pl = new PLIO("clip_in", adf::plio_32_bits, "data/
output_interp.txt", 100);
PLIO *pl_to_ai = new PLIO("clip_out", adf::plio_32_bits, "data/
input_classify.txt", 100);
PLIO *out0 = new PLIO("DataOut1", adf::plio_32_bits, "data/output.txt");

simulation::platform<2,2> platform(in0, pl_to_ai, out0, ai_to_pl);

clipped clipgraph;

connect<> net0(platform.src[0], clipgraph.in);
connect<> net1(clipgraph.clip_in, platform.sink[1]);
connect<> net2(platform.src[1], clipgraph.clip_out);
connect<> net3(clipgraph.out, platform.sink[0]);

#ifdef __AIESIM__

```



```
int main(int argc, char ** argv) {
    clipgraph.init();
    clipgraph.run();
    clipgraph.end();
    return 0;
}
#endif
```

To make the AI Engine simulator work, you must create input test bench files related to the RTL kernel. `data/output_interp.txt` is the test bench input to the RTL kernel. The AI Engine simulator generates the output file from the `interpolator` AI Engine kernel. The `data/input_classify.txt` file contains data from the `polar_clip` kernel which is input to the AI Engine `classify` kernel. Note that PLIO can have an optional attribute, PL clock frequency, which is 100 for the `polar_clip`.

RTL Blocks in Hardware Emulation and Hardware Flows

RTL kernels are fully supported in hardware emulation and hardware flows. You need to add the RTL kernel as an `nk` option and link the interfaces with the `sc` option, as shown in the following code. If necessary, adjust any clock frequency using `freqHz`. The following is an example of a Vitis configuration file.

```
[connectivity]
nk=mm2s:1:mm2s
nk=s2mm:1:s2mm
nk=polar_clip:1:polar_clip
sc=mm2s.s:ai_engine_0.DataIn1
sc=ai_engine_0.clip_in:polar_clip.in_sample
sc=polar_clip.out_sample:ai_engine_0.clip_out
sc=ai_engine_0.DataOut1:s2mm.s
[clock]
freqHz=100000000:polar_clip.ap_clk
```

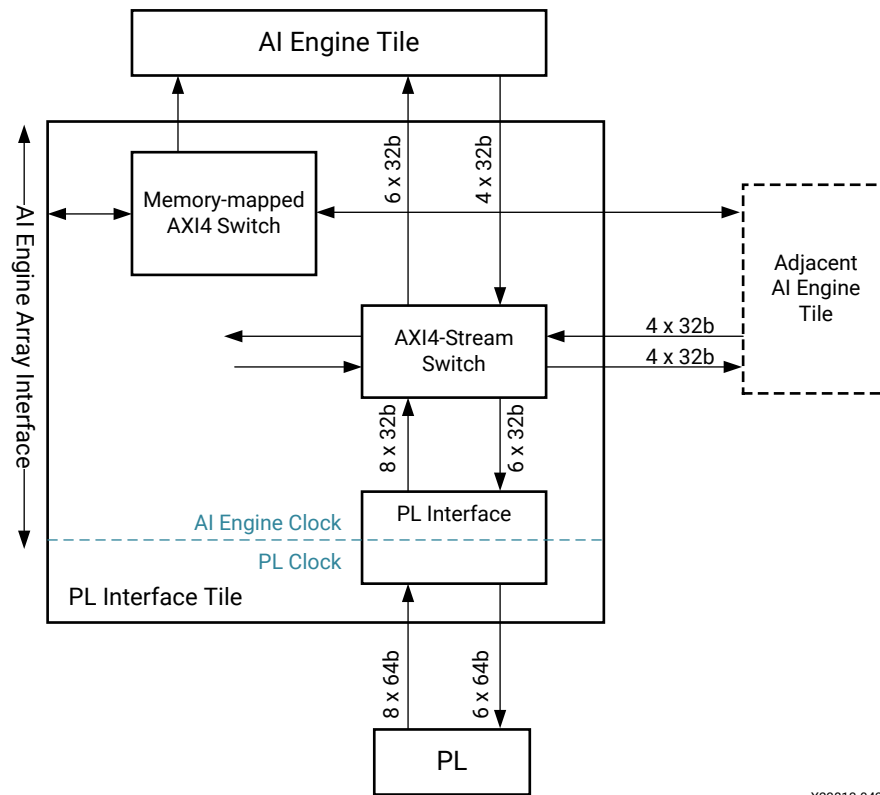
For more information on RTL kernels and the Vitis flow see [Chapter 14: Integrating the Application Using the Vitis Tools Flow](#). SystemC kernels can also be used in an emulation-only form. For this flow see [Working with SystemC Models](#) in the *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393).

Design Considerations for Graphs Interacting with Programmable Logic

The AI Engine array is made up of AI Engine tiles and AI Engine array interface tiles on the last row of the array. The types of interface tiles include AI Engine-PL and AI Engine-NoC.

Knowledge of the PL interface tile, which interfaces and adapts the signals between the AI Engines and the PL region, is essential to take full advantage of the bandwidth between AI Engines and the PL. The following figure shows an expanded view of a single PL interface tile.

Figure 10: AI Engine-PL Interface Tile



X23813-042920

Note: Notice the interface tile supports two different clock domains, AI Engine clock and PL clock, as well as a predefined number of streaming channels available to connect from the AI Engine tile to a specific PL interface tile.

PL Interface Tile Capabilities

The AI Engine clock can run at up to 1 GHz for -1L speed grade devices, or higher, for -2 and -3 speed grade devices. The default width of a stream channel is 32 bits. Because this frequency is higher than the PL clock frequency, it is always necessary to perform a clock domain crossing to the PL region, for example, to either one-half or a quarter of the AI Engine clock frequency.



RECOMMENDED: Though not required, Xilinx recommends running the PL kernel with a frequency where the AI Engine frequency is an integer multiple of the PL kernel frequency.

For C++ HLS PL kernels, choose an appropriate target frequency depending on the complexity of the algorithm implemented. The `--hls.clock` option can be used in the Vitis compiler when compiling HLS C/C++ into Xilinx object (XO) files.

AI Engine-to-PL Rate Matching

The AI Engine runs at 1 GHz (or more, depending on the device) and can write at most two streams with a 32-bit data width per cycle. In contrast, a PL kernel can run at 500 MHz (half the frequency of the AI Engine), while consuming a larger bit width. Rate matching is concerned with balancing the throughput from the producer to the consumer, and is used to ensure that neither of the processes creates a bottleneck with respect to the total performance. The following equation shows the rate matching for each channel:

$$\text{Frequency AI Engine} \times \text{Data AI Engine per cycle} = \text{Frequency PL} \times \text{Data PL per cycle}$$

The following table shows a PL rate matching example for a 32-bit channel written to each cycle by the AI Engine at 1 GHz for -1L speed grade devices. As shown, the PL IP has to consume two times the data at half the frequency or four times the data at one quarter of the frequency.

Table 13: Frequency Response of AI Engine Compared to PL Region

AI Engine		PL	
Frequency	Data per Cycle	Frequency	Data per Cycle
1 GHz	32 bit	500 MHz	64 bit
		250 MHz	128 bit

Because the need to match frequency and adjust data-path width is well understood by the Vitis compiler (v++), the tool automatically extracts the port width from the PL kernel, the frequency from the clock specification, and introduces an upsizer/downsizer to temporarily store the data exchanged between the AI Engine and the PL regions to manage the rate match.

To avoid deadlocks, it is important to ensure that if multiple channels are read or written between the AI Engine and the PL, the data rate per cycle is concurrently achieved on all channels. For example, if one channel requires 32 bits, and the second 64 bits, the AI Engine code must ensure that both channels are written adequately to avoid back pressure or starvation on the channel. Additionally, to avoid deadlock, writing/reading from the AI Engine and reading/writing in the PL code must follow the same chronological order.

The number of interfaces used in the graph function definition for the PL defines the number of AXI4-Stream interfaces. Each argument results in the creation of a separate stream.

AI Engine-PL Interface Performance

Versal AI Core series devices include an AI Engine array with the following column categories.

- **PL column:** provides PL stream access. Each column supports eight 64-bit slave channels for streaming data into the AI Engine and six 64-bit master channels for streaming data to the PL.
- **NoC column:** provides connectivity between the AI Engine array and the NoC. These interfaces can also connect to the PL.

To instruct the AI Engine compiler to select higher frequency interfaces, use the `--pl-freq=<number>` to specify the clock frequency (in MHz) for the PL kernels. The default value is one quarter of the AI Engine frequency and the maximum supported value is a half of the AI Engine frequency, the values depending on the speed grade. Following are examples:

- Option to enable an AI Engine-PL frequency of 300 MHz for all AI Engine-PL interfaces:

```
--pl-freq=300
```

- To set a different frequency for a specific PLIO interface use the following code to set it in the ADF graph.

```
adf::PLIO *(<input>= new adf::PLIO(<logical_name>, <plio_width>, <file>,  
<FreqMHz>);
```

Note: The following information applies to the AI Engine device architecture documented in *Versal ACAP AI Engine Architecture Manual* ([AM009](#)).

The AI Engine-PL AXI4-Stream channels use boundary logic interface (BLI) connections that include optional BLI registers with the exception of the slave channels 3 and 7. The two slave channels channel 3 and channel 7 are slower interfaces. The performance of the data transfer between the AI Engine and PL depends on whether the optional BLI registers are enabled or not.

For less timing-critical designs, all eight channels can be used without using the BLI registers. PL timing can still be met in this case. However, for higher frequency designs, only the six fast channels (0,1,2,4,5,6) can be used and the timing paths from the PL must be registered, using the BLI registers.

To control the use of BLI registers across the AI Engine-PL channels, use the `--pl-register-threshold=<number>` compiler option, specified in MHz. The default value is 1/8 of the AI Engine frequency based on speed grade. Following is an example:

- `--pl-register-threshold=125`

The compiler will map any PLIO interface with an AI Engine-PL frequency higher than this setting (125 MHz in this case) to high-speed channels with the BLI registers enabled. If the PLIO interface frequency is not higher than the `pl-register-threshold` value then any of the AI Engine-PL channels will be used.

In summary, if `pl-freq < pl-register-threshold` all eight channels can be used unregistered. If `pl-freq > pl-register-threshold` only the six fast channels can be used, with registering. `pl-register-threshold` is a way to control the threshold frequency beyond which only fast channels can be used (with registering).

Note: TLAST is required for a 64-bit stream between the AI Engine and PL if single 32-bit words are sent: AI Engine to PL 32-bit stream interfaces are automatically internally up-sized to 64-bit interfaces by the AI Engine compiler. When sending 32-bit stream data (to or from the PL from the AI Engine), single 32-bit words without TLAST are held in the interface until a second 32-bit word arrives to complete a 64-bit up-sizing. The workaround is to send TLAST after the 32-bit stream is sent.

Using a Virtual Platform

[Chapter 3: Introduction to AI Engine Programming](#) briefly introduced the simulation platform class with file I/O support. This chapter continues the discussion and describes other variations in detail.

Virtual Platform

A virtual platform specification helps to connect the data flow graph written with external I/O mechanisms specific to the chosen target for testing or eventual deployment. The platform could be specified for a simulation, emulation, or an actual hardware execution target.

Current release support is only for a simulation platform, which implies that you can execute a data flow graph in a software simulation environment. This is the specification.

```
simulation::platform<#inputs, #outputs> platform-name(port-attribute-list);
```

The `#inputs` and `#outputs` specify how many input and output ports are needed in the platform to connect to the data flow graph. The platform object is pre-populated with `src` and `sink` arrays of output and input ports (respectively) to make the connection. The `simpleGraph` example from [Chapter 3: Introduction to AI Engine Programming](#) is used in this example.

```
simpleGraph mygraph;  
simulation::platform<1,1> platform("input.txt","output.txt");  
connect<> net0(platform.src[0], mygraph.in);  
connect<> net1(mygraph.out, platform.sink[0]);
```

The port-attribute list within the platform declaration is an enumeration of attributes of each platform port starting with all the inputs and followed by the outputs. These are described in the following sections.

FileIO Attributes

By default, a platform port attribute is a string name used to construct an attribute of type `FileIO`. The string specifies the name of an input or output file relative to the current directory that will source or sink the platform data. The explicit form is specified in the following example using a `FileIO` constructor.

```
FileIO *in = new FileIO("input.txt");
FileIO *out = new FileIO("output.txt");
simulation::platform<1,1> platform(in,out);
```

`FileIO` ports are solely for the purpose of application simulation in the absence of an actual hardware platform. They are provided as a matter of convenience to test out a data flow graph in isolation before it is connected to a real platform. An actual hardware platform exports either stream or memory ports.

PLIO Attributes

A PLIO port attribute is used to make external stream connections that cross the AI Engine to programmable logic (PL) boundary. This situation arises when a hardware platform is designed separately and the PL blocks are already instantiated inside the platform. This hardware design is exported from the Vivado tools as a package XSA and it should be specified when creating a new project in the Vitis™ tools using that platform. The XSA contains a logical architecture interface specification that identifies which AI Engine I/O ports can be supported by the platform. The following is an example interface specification containing stream ports (looking from the AI Engine perspective).

Table 14: Example Logical Architecture Port Specification

AI Engine Port	Annotation	Type	Direction	Data Width	Clock Frequency (MHz)
S00_AXIS	Weight0	stream	slave	32	300
S01_AXIS	DataIn0	stream	slave	32	300
M00_AXIS	Dataout0	stream	master	32	300

This interface specification describes how the platform exports two stream input ports (slave port on the AI Engine array interface) and one stream output port (master port on the AI Engine array interface). A PLIO attribute specification is used to represent and connect these interface ports to their respective destination or source kernel ports in data flow graph.

The following example shows how the PLIO attributes shown in the previous table can be used in a program to read input data from a file or write output data to a file. The PLIO width and frequency of the PLIO port are also provided in the PLIO constructor. The constructor syntax is described in more detail in [Appendix A: Adaptive Data Flow Graph Specification Reference](#).

```
adf::PLIO *wts = new adf::PLIO("Weight0", adf::plio_64_bits,
    "inputwts.txt", 300);
adf::PLIO *din = new adf::PLIO("Datain0", adf::plio_64_bits, "din.txt",
    300);
adf::PLIO *dout = new adf::PLIO("Dataout0", adf::plio_64_bits, "dout.txt");
simulation::platform<2,1> platform(wts, din, dout);
```

The example simulation platform can then be connected to a graph that expects two input streams and one output stream in the usual way. During compilation, the logical architecture should be specified using the option `--logical-arch=<filename>`. This option is automatically populated by the Vitis tools if you have specified the XSA while creating the project. When simulated, the input weights and data are read from the two supplied files and the output data is produced in the designated output file in a streaming manner.

When a hardware platform is exported, all the AI Engine to PL stream connections are already routed to specific physical channels from the PL side.

Wide Stream Data Path PLIO

Typically, the AI Engine array runs at a higher clock frequency than the internal programmable logic. The AI Engine compiler can be given a compiler option `--pl-freq` to identify the frequency at which the PL blocks are expected to run. To balance the throughput between AI Engine and internal programmable logic, it is possible to design the PL blocks for a wider stream data path (64-bit, 128-bit), which is then sequentialized automatically into a 32-bit stream on the AI Engine stream network at the AI Engine to PL interface crossing.

The following example shows how wide stream PLIO attributes can be used in a program to read input data from a file or write output data to a file. The constructor syntax is described in more detail in [Appendix A: Adaptive Data Flow Graph Specification Reference](#).

```
PLIO *attr_o = new PLIO("TestLogicalNameOut", plio_128_bits, "data/
output.txt");
PLIO *attr_i = new PLIO("TestLogicalNameIn", plio_128_bits, "data/
input.txt");

simulation::platform<1, 1> platform(attr_i, attr_o); // Platform with PLIO
MEPL128BitClass gMePl;                             // Toplevel graph
connect<> net0(platform.src[0], gMePl.in);
connect<> net1(gMePl.out, platform.sink[0]);
```

In the previous example, a simulation platform with two 128-bit PLIO attributes is declared: one for input and one for output. The platform ports are then hooked up to the graph in the usual way. Data files specified in the PLIO attributes are then automatically opened for reading the input or writing the output respectively.

When simulating PLIO with data files, the data should be organized to accommodate both the width of the PL block as well as the data type of the connecting port on the AI Engine block. For example, a data file representing 32-bit PL interface to an AI Engine kernel expecting `int16` should be organized as two columns per row, where each column represents a 16-bit value. As another example, a data file representing 64-bit PL interface to an AI Engine kernel expecting `cint16` should be organized as four columns per row, where each column represents a 16-bit real or imaginary value. The same 64-bit PL interface feeding an AI Engine kernel with `int32` port would need to organize the data as two columns per row of 32-bit real values. The following examples show the format of the input file for the previously mentioned scenarios.

```
64-bit PL interface feeding AI Engine kernel expecting cint16
input file:
0 0 0 0
1 1 1 1
2 2 2 2

64-bit PL interface feeding AI Engine kernel expecting int32
input file:
0 0
1 1
2 2
```

With these wide PLIO attribute specifications, the AI Engine compiler automatically generates the AI Engine array interface configuration to convert a 64-bit or 128-bits data into a sequence of 32-bit words. The AXI4-Stream protocol followed with all PL IP blocks ensures that partial data can also be sent on a wider data path with the appropriate strobe signals describing which words are valid.

GMIO Attributes

A GMIO port attribute is used to make external memory-mapped connections to or from the global memory. These connections are made between an AI Engine graph and the logical global memory ports of a hardware platform design. The platform can be a base platform from Xilinx or a custom platform that is exported from the Vivado tools as a Xilinx device support archive (XSA) package.

AI Engine tools support mapping the GMIO port to the tile DMA one to one. It does not support mapping multiple GMIO ports to one tile DMA channel. There is a limit on the number of GMIO ports supported for a given device. For example, the XCVC1902 device on the VCK190 board has 16 AI Engine to NoC master units (NMU) in total. For each AI Engine to NMU, it supports two MM2S and two S2MM channels. So, there can be at most 32 AI Engine GMIO inputs and 32 AI Engine GMIO outputs supported, but note that it can be further limited by the existing hardware platform.

Note: GMIO channel constraints should not be used for AI Engine compilation.

While developing data flow graph applications on top of an existing hardware platform, you need to know what global memory ports are exported by the underlying XSA and their functionality. In particular, any input or output ports exposed on the platform are recorded within the XSA and can be viewed as a logical architecture interface.

Programming Model for AI Engine–DDR Memory Connection

The GMIO port attribute can be used to initiate AI Engine–DDR memory read and write transactions in the PS program. This enables data transfer between an AI Engine and the DDR controller through APIs written in the PS program. The following example shows how to use GMIO APIs to send data to an AI Engine for processing and retrieve the processed data back to the DDR through the PS program.

```
GMIO gmioIn("gmioIn", 64, 1000);
GMIO gmioOut("gmioOut", 64, 1000);
simulation::platform<1,1> plat(&gmioIn, &gmioOut);

myGraph gr;
connect<> c0(plat.src[0], gr.in);
connect<> c1(gr.out, plat.sink[0]);

int main(int argc, char ** argv)
{
    const int BLOCK_SIZE=256;
    int32 *inputArray=(int32*)GMIO::malloc(BLOCK_SIZE*sizeof(int32));
    int32 *outputArray=(int32*)GMIO::malloc(BLOCK_SIZE*sizeof(int32));

    // provide input data to AI Engine in inputArray
    for(int i=0;i<BLOCK_SIZE;i++){
        inputArray[i]=i;
    }

    gr.init();

    gmioIn.gm2aie_nb(inputArray, BLOCK_SIZE*sizeof(int32));
    gmioOut.aie2gm_nb(outputArray, BLOCK_SIZE*sizeof(int32));

    gr.run(8);

    gmioOut.wait();

    // can start to access output data from AI Engine in outputArray
    ...

    GMIO::free(inputArray);
    GMIO::free(outputArray);
    gr.end();
}
```

This example declares two GMIO objects. `gmioIn` represents the DDR memory space to be read by the AI Engine and `gmioOut` represents the DDR memory space to be written by the AI Engine. The constructor specifies the logical name of the GMIO, burst length (that can be 64, 128, or 256 bytes) of the memory-mapped AXI4 transaction, and the required bandwidth (in MB/s).

```
GMIO gmioIn("gmioIn", 64, 1000);
GMIO gmioOut("gmioOut", 64, 1000);
```

Assuming the application graph (`myGraph`) has an input port (`myGraph::in`) connecting to the processing kernels and an output port (`myGraph::out`) producing the processed data from the kernels, the following code connects `gm1` (as a platform source) to the input port of the graph and connects `gm2` (as a platform sink) to the output port of the graph.

```
simulation::platform<1,1> plat(&gmioIn, &gmioOut);
connect<> c0(plat.src[0], gr.in);
connect<> c1(gr.out, plat.sink[0]);
```

Inside the main function, two 256-element `int32` arrays are allocated by `GMIO::malloc`. The `inputArray` points to the memory space to be read by the AI Engine and the `outputArray` points to the memory space to be written by the AI Engine. In Linux, the virtual address passed to `GMIO::gm2aie_nb`, `GMIO::aie2gm_nb`, `GMIO::gm2aie` and `GMIO::aie2gm` must be allocated by `GMIO::malloc`. After the input data is allocated, it can be initialized.

```
const int BLOCK_SIZE=256;
int32 *inputArray=(int32*)GMIO::malloc(BLOCK_SIZE*sizeof(int32));
int32 *outputArray=(int32*)GMIO::malloc(BLOCK_SIZE*sizeof(int32));
```

`GMIO::gm2aie_nb` is used to initiate memory-mapped AXI4 transactions for the AI Engine to read from DDR memory spaces. The first argument in `GMIO::gm2aie_nb` is the pointer to the start address of the memory space for the transaction. The second argument is the transaction size in bytes. The memory space for the transaction must be within the memory space allocated by `GMIO::malloc`. Similarly, `GMIO::aie2gm_nb` is used to initiate memory-mapped AXI4 transactions for the AI Engine to write to DDR memory spaces. `GMIO::gm2aie_nb` or `GMIO::aie2gm_nb` is a non-blocking function in a sense that it returns immediately when the transaction is issued - it does not wait for the transaction to complete. By contrast, `GMIO::gm2aie` or `GMIO::aie2gm` behaves in a blocking manner.

In this example, assuming in one iteration, the graph consumes 32 `int32` data from the input port and produces 32 `int32` data to the output port. To run eight iterations, the graph consumes 256 `int32` data and produces 256 `int32` data. The corresponding memory-mapped AXI4 transactions are initiated using the following code, one `gm1.gm2aie_nb` call to issue a read transaction for eight-iteration worth of data, and one `gm2.aie2gm_nb` call to issue a write transaction for eight-iteration worth of data.

```
gmioIn.gm2aie_nb(inputArray, BLOCK_SIZE*sizeof(int32));
gmioOut.aie2gm_nb(outputArray, BLOCK_SIZE*sizeof(int32));
```

`gr.run(8)` is also a non-blocking call to run the graph for eight iterations. To synchronize between the PS and AI Engine for DDR memory read/write access, you can use `GMIO::wait` to block PS execution until the GMIO transaction is complete. In this example, `gmioOut.wait()` is called to wait for the output data to be written to `outputArray` DDR memory space.

Note: The memory is non-cachable for GMIO in Linux.

After that, the PS program can access the data. When PS has completed processing, the memory space allocated by `GMIO::malloc` can be released by `GMIO::free`.

```
GMIO::free(inputArray);
GMIO::free(outputArray);
```

GMIO APIs can be used in various ways to perform different level of control for read/write access and synchronization between the AI Engine, PS, and DDR memory. `GMIO::gm2aie`, `GMIO::aie2gm`, `GMIO::gm2aie_nb` or `GMIO::aie2gm_nb` can be called multiple times to associate different memory spaces for the same GMIO object during different phases of graph execution. Different GMIO objects can be associated with the same memory space for in-place AI Engine–DDR read/write access. Blocking versions of `GMIO::gm2aie` and `GMIO::aie2gm` APIs themselves are synchronization point for data transportation and kernel execution. Calling `GMIO::gm2aie` (or `GMIO::aie2gm`) is equivalent to calling `GMIO::gm2aie_nb` (or `GMIO::aie2gm_nb`) followed immediately by `GMIO::wait`. The following example shows the combination of the aforementioned use cases.

```
GMIO gmioIn("gmioIn", 64, 1000);
GMIO gmioOut("gmioOut", 64, 1000);
simulation::platform<1,1> plat(&gmioIn, &gmioOut);

myGraph gr;
connect<> c0(plat.src[0], gr.in);
connect<> c1(gr.out, plat.sink[0]);

int main(int argc, char ** argv)
{
    const int BLOCK_SIZE=256;
    // dynamically allocate memory spaces for in-place AI Engine read/write
    access
    int32* inoutArray=(int32*)GMIO::malloc(BLOCK_SIZE*sizeof(int32));

    gr.init();

    for (int k=0; k<4; k++)
    {
        // provide input data to AI Engine in inoutArray
        for(int i=0;i<BLOCK_SIZE;i++){
            inoutArray[i]=i;
        }

        gr.run(8);
        for (int i=0; i<8; i++)
        {
            gmioIn.gm2aie(inoutArray+i*32, 32*sizeof(int32)); //blocking
            call to ensure transaction data is read from DDR to AI Engine
            gmioOut.aie2gm_nb(inoutArray+i*32, 32*sizeof(int32));
        }
    }
}
```

```

        gmioOut.wait();

        // can start to access output data from AI Engine in inoutArray
        ...
    }
    GMIO::free(inoutArray);

    gr.end();
}

```

In the previous example, the two GMIO objects `gmioIn` and `gmioOut` are using the same memory space allocated by `inoutArray` for in-place read and write access.

Without knowing data flow dependency among the kernels inside the graph, and to ensure write-after-read for the `inoutArray` memory space, the blocking version `gmioIn.gm2aie` is called to ensure transaction data is copied from DDR memory to AI Engine local memory before issuing a write transaction to the same memory space in `gmioOut.aie2gm_nb`.

```

gmioIn.gm2aie(inoutArray+i*32, 32*sizeof(int32)); //blocking call to ensure
transaction data is read from DDR to AI Engine
gmioOut.aie2gm_nb(inoutArray+i*32, 32*sizeof(int32));

```

`gmioOut.wait()` is to ensure that data has been migrated to DDR memory. After it is done, the PS can access output data for post-processing.

The graph execution is divided into four phases in the for loop, for `(int k=0; k<4; k++)`. `inoutArray` can be re-initialized in the for loop with different data to be processed in different phases.

Hardware Emulation and Hardware Flows

GMIO is not only used as a virtual platform for the AI Engine simulator, but can also work in hardware emulation and hardware flows. To allow it to work in hardware emulation and hardware flows, add the following code to `graph.cpp`.

```

#ifdef __AIESIM__
#include "adf/adf_api/XRTConfig.h"
#include "experimental/xrt_kernel.h"
// Create XRT device handle for ADF API

char* xclbinFilename = argv[1];
auto dhdl = xrtDeviceOpen(0); //device index=0
xrtDeviceLoadXclbinFile(dhdl, xclbinFilename);
xuid_t uuid;
xrtDeviceGetXclbinUUID(dhdl, uuid);

adf::registerXRT(dhdl, uuid);
#endif

```

Using the guard macro `__AIESIM__`, the same version of `graph.cpp` can work for the AI Engine simulator, hardware emulation, and hardware flows. Note that the preceding code should be placed before calling the graph or the GMIO ADF APIs. At the end of the program, close the device using the `xrtDeviceClose()` API.

```
#if !defined(__AIESIM__)
    xrtDeviceClose(dhdl);
#endif
```

To compile the code for hardware flow, see [Chapter 13: Programming the PS Host Application](#).

While it is recommended to use ADF APIs to control the GMIO, it is possible to use XRT. However, only synchronous mode GMIO transactions are supported. The API to perform synchronous transferring data can be found in `experimental/xrt_aie.h`:

```
/**
 * xrtAIESyncBO() - Transfer data between DDR and Shim DMA channel
 *
 * @handle:          Handle to the device
 * @bohdl:           BO handle.
 * @gmioName:        GMIO name
 * @dir:             GM to AIE or AIE to GM
 * @size:            Size of data to synchronize
 * @offset:          Offset within the BO
 *
 * Return:           0 on success, or appropriate error number.
 *
 * Synchronize the buffer contents between GMIO and AIE.
 * Note: Upon return, the synchronization is done or error out
 */
int
xrtAIESyncBO(xrtDeviceHandle handle, xrtBufferHandle bohdl, const char
*gmioName, enum xclBOSyncDirection dir, size_t size, size_t offset);
```

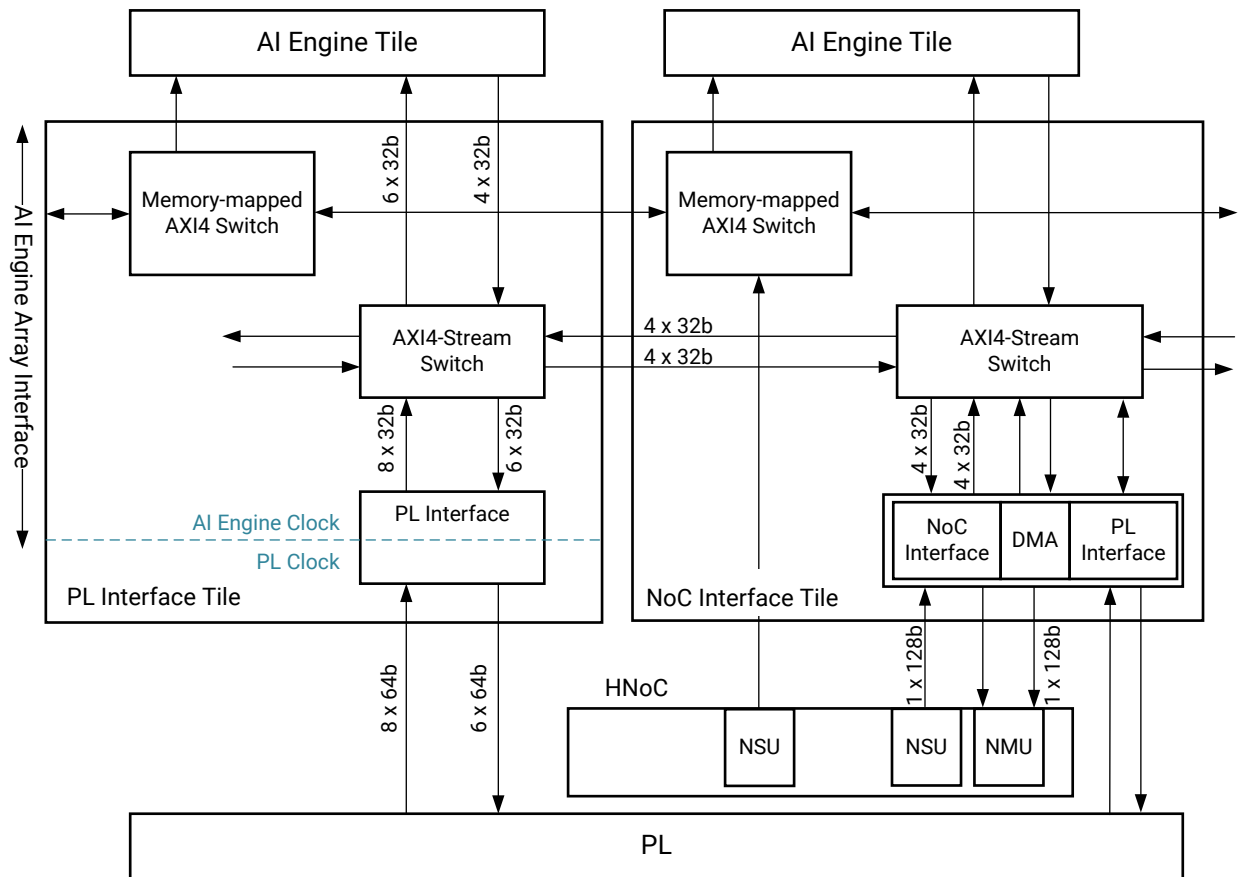
Performance Comparison Between AI Engine/PL and AI Engine/NoC Interfaces

The AI Engine array interface consists of the PL and NoC interface tiles. The AI Engine array interface tiles manage the two following high performance interfaces.

- AI Engine to PL
- AI Engine to NoC

The following image shows the AI Engine array interface structure.

Figure 11: AI Engine Array Interface Topology



X25346-061021

One AI Engine to PL interface tile contains eight streams from the PL to the AI Engine and six streams from the AI Engine to the PL. The following table shows one AI Engine to PL interface tile capacity.

Table 15: AI Engine Array Interface to PL Interface Bandwidth Performance

Connection Type	Number of Connections	Data Width (bits)	Clock Domain	Bandwidth per Connection (GB/s)	Aggregate Bandwidth (GB/s)
PL to AI Engine array interface	8	64	PL (500 MHz)	4	32
AI Engine array interface to PL	6	64	PL (500 MHz)	4	24

Note: All bandwidth calculations in this section assume a nominal 1 GHz AI Engine clock for a -1L speed grade device at VCCINT = 0.70V with the PL interface running at half the frequency of the AI Engine as an example.

The exact number of PL and NoC interface tiles is device-specific. For example, in the VC1902 device, there are 50 columns of AI Engine array interface tiles. However, only 39 array interface tiles are available to the PL interface. Therefore, the aggregate bandwidth for the PL interface is approximately:

- $24 \text{ GB/s} * 39 = 0.936 \text{ TB/s}$ from AI Engine to PL
- $32 \text{ GB/s} * 39 = 1.248 \text{ TB/s}$ from PL to AI Engine

The number of array interface tiles available to the PL interface and total bandwidth of the AI Engine to PL interface for other devices and across different speed grades is specified in *Versal AI Core Series Data Sheet: DC and AC Switching Characteristics* ([DS957](#)).

GMIO uses DMA in the AI Engine to NoC interface tile. The DMA has two 32-bit incoming streams from the AI Engine and two 32-bit streams to the AI Engine. In addition, it has one 128-bit memory mapped AXI master interface to the NoC NMU. The performance of one AI Engine to NoC interface tile is shown in the following table.

Table 16: AI Engine to NoC Interface Tile Bandwidth Performance

Connection Type	Number of connections	Bandwidth per connection (GB/s)	Aggregate Bandwidth (GB/s)
AI Engine to DMA	2	4	8
DMA to NoC	1	16	16
DMA to AI Engine	2	4	8
NoC to DMA	1	16	16

The exact number of AI Engine to NoC interface tiles is device-specific. For example, in the VC1902 device, there are 16 AI Engine to NoC interface tiles. So, the aggregate bandwidth for the NoC interface is approximately:

- $8 \text{ GB/s} * 16 = 128 \text{ GB/s}$ from AI Engine to PL
- $8 \text{ GB/s} * 16 = 128 \text{ GB/s}$ from PL to AI Engine

When accessing DDR memory, the integrated DDR memory controller (DDRMC) number in the platform limits the performance of DDR memory read and write. For example, if all four DDRMCs in a VC1902 device are fully used, the hard limit to access DDR memory is as follows.

- $3200 \text{ Mb/s} * 64 \text{ bit} * 4 \text{ DDRMCs} / 8 = 102.4 \text{ GB/s}$

The performance of GMIO accessing DDR memory through the NoC is further restricted by the NoC lane number in the horizontal and vertical NoC, inter NoC configurations, and QoS. Note that DDR memory read and write efficiency is largely affected by the access pattern and other overheads. For more information about the NoC, memory controller use, and performance numbers, see the *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* ([PG313](#)).

For a single connection from the AI Engine or to the AI Engine, both PLIO and GMIO have a hard bandwidth limit of 4 GB/s. Some advantages and disadvantages for choosing PLIO or GMIO are shown in the following table.

Table 17: Comparison of PLIO vs GMIO

	PLIO	GMIO
Advantages	<ul style="list-style-type: none"> Number of AI Engine to PL interface streams are larger, hence larger aggregate bandwidth No interference between different stream connections Supports packet switching 	<ul style="list-style-type: none"> No PL resource required No timing closure requirement
Disadvantages	<ul style="list-style-type: none"> Congestion risk if there are too many stream connections in a region of the device Timing closure required for achieving best performance 	<ul style="list-style-type: none"> Less GMIO ports available Aggregate bandwidth is lower Multiple GMIO ports competing for bandwidth

Enhanced Programming Model

In the alternative programming model, `simulation::platform` is no longer required and global PLIO/GMIO objects construction are not needed.

An example of `graph.h` in the current programming model is as follows.

```
class simpleGraph : public graph
{
private
    /* declares kernels */
    kernel k1;
    kernel k2;
public
    /* declares input/output ports */
    input_port in0;
    input_port in1;
    output_port out0;
    output_port out1;

    /* declare graph constructor */
    simpleGraph()
    {
        k1 = kernel::create(kernel1);
        k2 = kernel::create(kernel2);
        connect< stream > (in0, k1.in[0]);
        connect< stream > (k1.out[0], out0);
        connect< stream > (in1, k2.in[0]);
        connect< stream > (k2.out[0], out1);
    }
};
```

An example of graph.cpp in the current programming model is as follows.

```
/* Instantiate GMIO/PLIO objects and links to logical names for design. */
GMIO *in0 = new GMIO("GMIO_In0", 64, 1);
GMIO *out0 = new GMIO("GMIO_Out0", 64, 1);
PLIO *in1 = new PLIO("PLIO_In0", plio_32_bits, "data/input.txt", 250.0);
PLIO *out1 = new PLIO("PLIO_Out0", plio_32_bits, "data/output.txt", 250.0);

/* Instantiate mygraph object */
simpleGraph mygraph;

/* Link inputs/outputs to simulation::platform */
simulation::platform<2, 2> platform(in0, in1, out0, out1);
connect<> net0(platform.src[0], mygraph.in0);
connect<> net1(platform.src[1], mygraph.in1);
connect<> net2(platform.sink[0], mygraph.out0);
connect<> net3(platform.sink[1], mygraph.out1);
```

The new alternative AI Engine programming model has the following advantages.

- It eliminates global PLIO/GMIO instances construction. In global scope, using for-loop expression to create an array of PLIO/GMIO objects is not supported. This makes implementing a design with large number of PLIO/GMIO objects error prone. For example:

```
class mygraph : public graph
{
public:
    // Declare N kernels.
    kernel k[N];
    // Declare N input_gmio(s) and output_gmio(s)
    input_gmio gmioIn[N];
    output_gmio gmioOut[N];

    mygraph()
    {
        // Create multiple input_gmio(s)/output_gmio(s) and connections
        for (int i=0; i<N; i++)
        {
            gmioIn[i] = input_gmio::create("gmioIn" + std::io_string(i),
64, 1);
            gmioOut[i] = output_gmio::create("gmioOut" +
std::io_string(i), 64, 1);
            connect<window<128>>(gmio[i].out[0], k[i].in[0]);
            connect<window<128>>(k[i].out[0], gmioOut[i].in[0]);
        }
    }
};
```

Note: It is not required to provide logical names when creating the GMIO/ PLIO objects. If the user chooses not to provide the logical name, AI Engine compiler will automatically come up with a unique logical name.

- It provides ease of use for connections within the design. With PLIO/GMIO objects in local scope, connect<> calls are not required to provide a unique connection name. For example, in the previous graph.cpp example, net0, net1, net2, and net3 are no longer needed.

- It supports the direction property of PLIO/GMIO objects is determined within object construction. With `simulation::platform` in design, the direction property was determined by the position of the objects in `simulation::platform<num_input, num_output>` constructor parameters that is outside of the PLIO/GMIO constructor. For example, in the previous `graph.cpp` example, `simulation::platform<m, n>` indicates there are `m` inputs and `n` outputs.

An example of alternative programming model that addresses these previously listed enhancements is as follows.

```
/* A graph with both PLIO and GMIO */
class mygraph : graph
{
public:

    /* New classes support input_gmio/output_gmio and input_plio/
    output_plio */
    input_gmio gm_in;
    output_gmio gm_out;

    input_plio pl_in;
    output_plio pl_out;

    kernel k1, k2;
    mygraph()
    {
        k1 = kernel::create(...);
        k2 = kernel::create(...);

        /* create() API for PLIO and GMIO objects support same arguments
        specified as in global scope */
        gm_in = input_gmio::create("GMIO_In0", /*const std::string&
        logical_name*/
                                   64,          /*size_t burst_length*/
                                   1             /*size_t bandwidth*/);

        gm_out = output_gmio::create("GMIO_Out0", 64, 1);

        pl_in = input_plio::create("PLIO_In0",          /* std::string
        logical_name */
                                   plio_32_bits,        /* enum plio_type
        plio_width */
                                   "data/input.txt",    /* std::string
        data_file */
                                   250.0                /* double frequency
        */ );

        pl_out = output_plio::create("PLIO_Out0", plio_32_bits, "data/
        output.txt", 250.0);

        /* Each input_gmio/output_gmio and input_plio/output_plio supports
        1 port per direction */
        connect<>(gm_in.out[0], k1.in[0]);
        connect<>(k1.out[0], gm_out.in[0]);

        connect<>(pl_in.out[0], k2.in[0]);
        connect<>(k2.out[0], pl_out.in[0]);

        location<GMIO>(gm_in) = shim(col);
        location<GMIO>(gm_out) = shim(col, ch_num); /* not supported */
    }
};
```

```

        location<PLIO>(pl_in) = shim(col);
        location<PLIO>(pl_out) = shim(col, ch_num);
    }
};

```

The previous code snippet demonstrates:

- PLIO/GMIO objects are no longer in global scope. They are public objects within the graph.
- `input_plio`, `output_plio` classes are inherited from `PLIO` class.
- `input_gmio`, `output_gmio` classes are inherited from `GMIO` class.
- `simulation::platform` is not required.
- PLIO/GMIO object construction contains direction property. It is not determined by the position of `simulation::platform` construction.
- `connect` call does not require unique name.
- Location constraints can be applied to PLIO/GMIO objects directly with the exception of GMIO channel constraint which is not supported in this release.

Note:

1. GMIO shim tile location constraint with DMA channel is not supported.
2. Host application calling profiling APIs need to update from referencing global PLIO/GMIO variables, `in0`, `out0`, `in1`, `out1` to referencing object's public members, `mygraph.gm_in`, `mygraph.gm_out`, `mygraph.pl_in`, `mygraph.pl_out`. See [Chapter 11: Performance Analysis of AI Engine Graph Application during Simulation](#) for additional details about profiling APIs.

An example of a profile call from the original programming model is as follows.

```

event::handle handle0 = event::start_profiling(*in0,
event::io_stream_start_to_bytes_transferred_cycles,
sizeIn*sizeof(cint16));

```

An example of a profile call from the alternative programming model is as follows.

```

event::handle handle0 = event::start_profiling(mygraph.gm_in,
event::io_stream_start_to_bytes_transferred_cycles,
sizeIn*sizeof(cint16));

```



IMPORTANT! The PLIO/GMIO object creation requires to have uniqueness throughout the AI Engine design. The following error message indicates the violation of the PLIO/GMIO name uniqueness.

```

ERROR: [aiecompiler 77-4551] The logical name DataIn1 of node i6
conflicts with the logical/qualified name of node i0.

```

Compiling an AI Engine Graph Application

This chapter describes all the command line options passed to the AI Engine compiler (`aiecompiler`). It takes the code for the data flow graph, the code for individual kernels, and produces an image that can be run on various AI Engine target platforms such as simulators, emulators, and AI Engine devices. The AI Engine compiler statically compiles the graph, map and place the kernels in the AI Engines.



TIP: Unless otherwise specified, all the input file paths are with respect to the current directory, and all the output file paths are with respect to the `Work` directory.

The AI Engine graph and kernels can be compiled individually, or as a standalone application to run in the AI Engine processor array through emulation or hardware. The graph and kernels can also be used as part of a larger system design, that incorporates the AI Engine graph with ELF application running on the embedded processor of the Versal® device and programmable logic (PL) kernels running in the programmable logic of the device. The AI Engine compiler is used to compile the graph and kernels, whether in a standalone configuration or as part of a larger system.

As shown in [Chapter 15: Using the Vitis IDE](#), the Vitis™ IDE can be used to create and manage project build settings, and run the AI Engine compiler. Alternatively, you can build the project from the command line as discussed in [Chapter 14: Integrating the Application Using the Vitis Tools Flow](#), or in a script or Makefile. Either approach lets you perform simulation or emulation to verify the graph application or the integrated system design, debug the design in an interactive debug environment, and build your design to run and deploy on hardware. Whatever method you choose to work with the tools, start by setting up the environment.

Setting Up the Vitis Tool Environment

The AI Engine tools are delivered and installed as part of the Vitis unified software platform. Therefore, when preparing to run the AI Engine tools, for example, the AI Engine compiler and AI Engine simulator, you must set up the Vitis tools. The Vitis unified software platform includes two elements that must be installed and configured, along with a valid Vitis tools license, to work together properly.

- Vitis tools and AI Engine tools
- A target Vitis platform such as the `xilinx_vck190_base_202120_1` platform used for AI Engine applications

For more information, see *Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#)).

When the elements of the Vitis software platform are installed, set up the environment to run in a specific command shell by running the following scripts.

```
#setup XILINX_VITIS and XILINX_HLS variables
source <Vitis_install_path>/Vitis/<version>/settings64.csh
```



TIP: *settings64.sh* and *setup.sh* scripts are also provided in the same directory.

Finally, define the location of the available target platforms to use with the Vitis IDE and AI Engine tools using the following environment variable:

```
setenv PLATFORM_REPO_PATHS <path to platforms>
```



TIP: The `PLATFORM_REPO_PATHS` environment variable points to directories containing platform files (XPFM). This lets you specify platforms using just the folder name for the platform.

You can validate the installation and setup of the tools by using one of the following commands.

```
which vitis
which aiecompiler
```

You can validate the platform installation by using the following command.

```
platforminfo --list
```

Inputs

The AI Engine compiler takes inputs in several forms and produces executable applications for running on an AI Engine device. The command line for running AI Engine compiler is as follows:

```
aiecompiler [options] <Input File>
```

where:

- `<Input File>` specifies the data flow graph code that defines the `main()` application for the AI Engine graph. The input flow graph is specified using a data flow graph language. Refer to [Creating a Data Flow Graph \(Including Kernels\)](#) for a description of the data flow graph.

An example AI Engine compiler command:

```
aiecompile --verbose --pl-freq=100 --workdir=./myWork --
platform=xilinx_vck190_202120_1.xpfm\
--include="." --include="./src" --include="./src/kernels" --include="./
data" --include="${XILINX_HLS}/include" \
./src/graph.cpp
```

Some additional input options for the command line can include the following:

- `--constraints=<jsonfile>` to specify constraints such as location or placement bounding box.

Outputs

By default the AI Engine compiler writes all outputs to a directory called `Work` and a file `libadf.a`, where `Work` is a sub-directory of the current directory where the tool was launched and `libadf.a` is a file used as an input for the Vitis compiler created in the same directory as the AI Engine compiler was launched from. The type of output and contents of the output directory depend on the `--target` specified, as described in [AI Engine Compiler Options](#). For more information about the Vitis compiler, see [Vitis Compiler Command](#) in the *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393).



TIP: You can specify a different output directory using the `--workdir` option.

The structure and contents of the `./Work` directory are described in the following table.

Table 18: Work Directory Structure

Directory/Files	Description
<code>./Work/</code>	
<code><name>.aiecompile_summary</code>	A generated file that can be opened in Vitis analyzer to see a compilation summary.
<code>config/scsim_config.json</code>	A JSON script that specifies options to the SystemC simulator. It includes AI Engine array tile geometry, input/output file specifications, and their connections to the stream switches.
<code>arch/</code>	
<code>logical_arch_aie.larch</code>	This is a JSON file describing the hardware requirements of the AI Engine application.
<code>aieshim_constraints.json</code>	If present, this JSON file represents the user-defined physical interface constraints between AI Engine array and programmable logic provided through the AI Engine application.
<code>aieshim_solution.aiesol</code>	This is a JSON file describing the mapping from logical to physical channels crossing the interface between the AI Engine array and the programmable logic.
<code>cfggraph.xml</code>	This is an XML file describing the hardware requirements of the AI Engine application. This is used by the Vitis tools flow.

Table 18: Work Directory Structure (cont'd)

Directory/Files	Description
aie/ Makefile array and programmable<n>_<m>/ Release/ <n>_<m>.lst <n>_<m>.map scripts/ src/	<p>A Makefile to compile code for all AI Engines.</p> <p>These are individual AI Engine compilation directories.</p> <p>Synopsys release directory for the AI Engine including ELF file.</p> <p>Microcode of the kernel at <n>_<m>.</p> <p>Shows the memory mapping of the kernel at <n>_<m>. It also includes the memory size, width, and offset.</p> <p>Synopsys compiler project and linker scripts.</p> <p>Source files for the processor including kernels and <code>main</code>.</p>
ps/c-rtts/ aie_control.cpp aie_control_xrt.cpp systemC/ Makefile generated-source/ generated-objects/	<p>Directory containing C-based run-time control for modeling PS interaction.</p> <p>This is the AI Engine control code generated implementing the <code>init</code>, <code>run</code>, <code>end</code> graph APIs for the specific graph objects present in the program. This file is linked with the application <code>main</code> to create a PS thread for the simulator and bare metal.</p> <p>This is the AI Engine control code generated implementing the <code>init</code>, <code>run</code>, <code>end</code> graph APIs for the specific graph objects present in the program. This file is linked with the application <code>main</code> to create a PS thread for the Linux application.</p> <p>Directory containing SystemC models for PS <code>main</code>.</p> <p>A Makefile to compile all PS SystemC models.</p> <p>SystemC wrappers for PS <code>main</code>.</p> <p>Compiled shared libraries for PS <code>main</code>.</p>
ps/cdo/ Makefile generateAIEConfig generated-sources/ generated-objects/	<p>Directory containing generator code for graph configuration and initialization in configuration data object (CDO) format. This is used during SystemC-RTL simulation and during actual hardware execution.</p> <p>A Makefile to compile graph CDO</p> <p>A bash script for building graph CDO</p> <p>C++ program to generate CDO.</p> <p>Compiled program to generate CDO.</p>
pthread/ PthreadSim.c sim.out	<p>A source-to-source translation of the input data flow graph into a C program implemented using <code>pthreads</code>.</p> <p>The GCC compiled binary for <code>PthreadSim.c</code>.</p>

Table 18: Work Directory Structure (cont'd)

Directory/Files	Description
reports/ <graph>_mapping_analysis_report.txt <graph>.png <graph>.xpe sync_buffer_addresses.json lock_allocation_report.json dma_lock_report.json	<p>Mapping report describing allocation of kernels to AI Engines and window buffers to AI Engine memory groups.</p> <p>A bitmap file showing the kernel graph connectivity and partitioning over AI Engines.</p> <p>An XML file describing the estimated power profile of the graph based on hardware resources used. This file is used with the Xilinx® Power Estimator (XPE) tool.</p> <p>Shows kernel sync buffer addresses with local and global addresses.</p> <p>Describes the ports and what locks and buffers are associated with the kernels.</p> <p>Shows DMA locks for inputs/outputs to the AI Engine as well as the kernel(s) they connect to with buffer information.</p>
temp/	This directory contains some temporary files generated by the AI Engine compiler that can be useful in debugging. In addition, the CF graph .o file is also created here by default.

AI Engine Compiler Options

Table 19: AI Engine Options

Option Name	Description
--constraints=<string>	Constraints (location, bounding box, etc.) can be specified using a JSON file. This option lets you specify one or more constraint files.
--heapsize=<int>	<p>Heap size (in bytes) used by each AI Engine</p> <p>The stack, heap, and sync buffer (32 bytes, includes the graph run iteration number information) are allocated up to 32768 bytes of data memory. The default heap size is set to 1024 bytes. Before changing the heap size to a different value, ensure that the sum of the stack, heap, and sync buffer sizes does not exceed 32768 bytes.</p> <p>Used for allocating any remaining file-scoped data that is not explicitly connected in the user graph.</p>
--stacksize=<int>	<p>Stack size (in bytes) used by each AI Engine</p> <p>The stack, heap, and sync buffer (32 bytes) are allocated up to 32768 bytes of data memory. The default stack size is set to 1024 bytes. Before changing the stack size to a different value, ensure that the sum of the stack, heap, and sync buffer sizes does not exceed 32768 bytes.</p> <p>Used as a standard compiler calling convention including stack-allocated local variables and register spilling.</p>
--pl-freq=<value>	Specifies the interface frequency (in MHz) for all PLIOs. The default frequency is a quarter of the AI Engine frequency and the maximum supported frequency is half of the AI Engine frequency. The PL frequency specific to each interface is provided in the graph.

Table 19: AI Engine Options (cont'd)

Option Name	Description
<code>--pl-register-threshold=<value></code>	Specifies the frequency (in MHz) threshold for registered AI Engine-PL crossings. The default frequency is one-eighth of the AI Engine frequency dependent on the specific device speed grade. Note: Values above a quarter of the AI Engine array frequency are ignored, and a quarter is used instead.

Table 20: CDO Options

Option Name	Description
<code>--enable-ecc-scrubbing</code>	Enable ECC Scrubbing on all the AI Engines used. This option enables ECC Scrubbing when generating the AI Engine ELF CDO. (One performance counter per core is used.) ECC Scrubbing is turned ON (true) by default.

Table 21: Compiler Debug Options

Option Name	Description
<code>--adf-api-log-level=<value></code>	ADF API log-level (0: errors, 1: level-0 + warnings, 2: level-1 + info messages, 3: level-2 + debug messages) (default: 2)
<code>--kernel-linting</code>	Perform consistency checking between graphs and kernels. The default is false.
<code>--known-tripcount</code>	Converting unknown trip count to known trip count.
<code>--quiet</code>	Suppress the output of the AI Engine compiler.
<code>--verbose</code>	Verbose output of the AI Engine compiler emits compiler messages at various stages of compilation. These debug and tracing logs provide useful messages regarding the compilation process.

Table 22: Execution Target Options

Option Name	Description
<code>--target=<hw x86sim></code>	The AI Engine compiler supports several build targets (default: <code>hw</code>): <ul style="list-style-type: none">The <code>hw</code> target produces a <code>libadf.a</code> for use in the hardware device on a target platform.The <code>x86sim</code> target compiles the code for use in the x86 simulator as described in x86 Functional Simulator.

Table 23: File Options

Option Name	Description
<code>--include=<string></code>	This option can be used to include additional directories in the include path for the compiler front-end processing. Specify one or more include directories.
<code>--output=<string></code>	Specifies an <code>output.json</code> file that is produced by the front end for an input data flow graph file. The output file is passed to the back-end for mapping and code generation of the AI Engine device. This is ignored for other types of input.
<code>--platform=<string></code>	This is a path to a Vitis platform file that defines the hardware and software components available when doing a hardware design and its RTL co-simulation.

Table 23: File Options (cont'd)

Option Name	Description
<code>--workdir=<string></code>	By default, the compiler writes all outputs to a sub-directory of the current directory, called <code>Work</code> . Use this option to specify a different output directory.

Table 24: Generic Options

Option Name	Description
<code>--help</code>	List the available AI Engine compiler options, sorted in the groups listed here.
<code>--help-list</code>	Display an alphabetic list of AI Engine compiler options.
<code>--version</code>	Display the version of the AI Engine compiler.

Table 25: Miscellaneous Options

Option Name	Description
<code>--no-init</code>	<p>This option disables initialization of window buffers in AI Engine data memory. This option enables faster loading of the binary images into the SystemC-RTL co-simulation framework.</p> <p>TIP: This does not affect the statically initialized lookup tables.</p>
<code>--nodot-graph</code>	By default, the AI Engine compiler produces <code>.dot</code> and <code>.png</code> files by default to visualize the user-specified graph and its partitioning onto the AI Engines. This option can be used to eliminate the dot graph output.

Table 26: Module Specific Options

Option Name	Description
<code>--Xchess=<string></code>	<p>Can be used to pass kernel specific options to the CHESS compiler that is used to compile code for each AI Engine.</p> <p>The option string is specified as <code><kernel-function>:<optionid>=<value></code>. This option string is included during compilation of generated source files on the AI Engine where the specified kernel function is mapped.</p>
<code>--Xelfgen=<string></code>	<p>Can be used to pass additional command-line options to the ELF generation phase of the compiler, which is currently run as a <code>make</code> command to build all AI Engine ELF files.</p> <p>For example, to limit the number of parallel compilations to four, you write <code>-Xelfgen="-j4"</code>.</p> <p>Note: If during compilation you see errors with <code>bad_alloc</code> in the log, or if the Vitis IDE crashes, this could be due to insufficient memory on your workstation. A possible workaround (other than increasing the available memory on your machine) is to limit the parallelism used by the compiler during code generation phase. This can be specified in the GUI as the compiler CodeGen option <code>-j1</code> or <code>-j2</code>, or on the command line as <code>-Xelfgen=-j1</code> or <code>-Xelfgen=-j2</code>.</p>

Table 26: Module Specific Options (cont'd)

Option Name	Description
--Xmapper=<string>	<p>Can be used to pass additional command-line options to the mapper phase of the compiler. For example:</p> <pre>--Xmapper=DisableFloorplanning</pre> <p>These are options to try when the design is either failing to converge in the mapping or routing phase, or when you are trying to achieve better performance via reduction in memory bank conflict.</p> <p>See the Mapper and Router Options for a list and description of options.</p>
--Xpreproc=<string>	<p>Pass general option to the PREPROCESSOR phase for all source code compilations (AIE/PS/PL/x86sim). For example:</p> <pre>--Xpreproc=-D<var>=<value></pre>
--Xpslinker=<string>	<p>Pass general option to the PS LINKER phase. For example:</p> <pre>--Xpslinker=-L<libpath> -l<libname></pre>
--Xrouter=<string>	<p>Pass general option to the ROUTER phase. For example:</p> <pre>-Xrouter=enableSplitAsBroadcast</pre>
--fast-floats	Enable fast implementation for linear floating point scalar operations like add, sub, mul, and compare.
--fast-nonlinearfloats	Enable fast implementation for non-linear floating point scalar operations like sine/cosine, sqrt, and inv.

Note: Only AI Engine kernels that have been modified are recompiled in subsequent compilations of the AI Engine graph. Any un-modified kernels will not be recompiled.

Table 27: Event Trace Options

Option Name	Description
--event-trace=<value> where <value> is one of the following: <ul style="list-style-type: none"> functions functions_partial_stalls functions_all_stalls runtime 	<p>Event trace configuration value. Where the specified <value> indicates the following:</p> <ul style="list-style-type: none"> Function transition view without stalls. Function transition view with stream/lock/cascade stalls. Function transition view with all stalls (stream/lock/cascade/memory). Run-time event tracing configuration.
--event-trace-port=<value> <ul style="list-style-type: none"> plio gmio 	<p>Set the AI Engine event tracing port. The default value is <code>plio</code>; however, Xilinx recommends that you use <code>gmio</code> as the event-trace-port configuration. See Event Trace Build Flow for more information.</p> <ul style="list-style-type: none"> Set the AI Engine event tracing port to <code>plio</code> Set the AI Engine event tracing port to <code>gmio</code>
--num-trace-streams=<int>	Number of trace streams (default: 16).
--trace-plio-width=<int>	PLIO width for trace streams (default: 64). Allowed values are 32 and 64.

Table 28: Optimization Options

Option Name	Description
<code>--xlopt=<int></code>	<p>Enable a combination of kernel optimizations based on the <code>opt</code> level (allowed values are 0 to 2, default is 1).</p> <ul style="list-style-type: none"> <code>xlopt=1</code> <ul style="list-style-type: none"> Automatic computation of heap size: Enables ease of use using kernel analysis to automatically compute the heap requirements for each AI Engine. Therefore, you do not need to specify the heap size. Guidance: Guidance is provided to highlight unaligned variables, global arrays that can potentially be mapper allocated, improper usage of restrict, and potential read before write conflicts. <code>xlopt=2</code> <ul style="list-style-type: none"> Automatic inline: Automatically inlining functions if it is practical and possible to do so, even if the functions are not declared as <code>__inline</code> or <code>inline</code>. Loop peeling for unrolled loops: Make loop iteration count a multiple of the unrolling factor via peeling. Split a loop into multiple loops based on its iteration count and profitability heuristics, and add flattening pragma on the split loops. Pragma insertion: Automatically infer and insert pragmas in kernel code. <p>Note: Compiler optimization (<code>xlopt > 0</code>) reduces debug visibility.</p>
<code>--Xxloptstr=<string></code>	<p>Option string to enable/disable optimizations in <code>xlopt</code> level 2.</p> <ul style="list-style-type: none"> <code>-xinline-threshold=T</code>: set the automatic inlining threshold to T (default T = 5000) <code>-annotate-pragma</code>: automatic insertion of loop unrolling, pipelining, and flattening pragma (default = true)

Mapper and Router Options

Table 29: Mapper Options

Options	Description
<code>DisableFloorplanning</code>	This option disables the auto-floor-planning phase in the mapper. This option is useful for heavily constrained designs where you want to guide mapping phase by using location constraints.
<code>BufferOptLevel[1-9]</code>	These options can be used to improve throughput by reducing memory bank conflicts. At higher <code>BufferOptLevel</code> , mapper tries to reduce number of buffers getting mapped into same memory bank, thereby reducing the probability of bank conflicts affecting overall performance. Higher <code>BufferOptLevels</code> can increase the size of the overall mapped region, and in few cases, can fail to find a solution. The default of <code>BufferOptLevels</code> is <code>BufferOptLevel0</code> .
<code>disableSeparateTraceSolve</code>	Default trace behavior forces the AI Engine mapper to keep all PLIOs/GMIOs in the original design location when using the trace debug feature. However, if the original solution did not leave any room for trace GMIOs, no solution will be possible unless the design PLIOs are moved. This option is to be used in this case.

Note: You can recirculate the previous design placement in your next compilation. This significantly reduces the mapper run time. When the compiler runs, it generates a placement constraints file, `graph_aie_mapped.aiecst`, in the `Work/temp` directory. Xilinx recommends that you save `Work/temp/graph_aie_mapped.aiecst` if you want to use it in subsequent compilations because the `Work` folder is regenerated for every compilation. This constraint file can be specified on the command line for the next iteration.

```
aiecompiler --constraints Work/temp/graph_aie_mapped.aiecst src/graph.cpp
```



TIP: The mapper is not aware of the 16K program memory per core limitation. One workaround is to change the run-time usage specification to map kernels to different cores.

Table 30: Router Options

Options	Description
<code>enableSplitAsBroadcast</code>	This option treats all split nets from a split node as a single net, with 100% usage broadcasting to multiple points. This broadcast net does not share resources with any other packet switched net in the design. This option can be used when throughput degradation is observed due to interference on <code>packetstream</code> nets after split node.
<code>dmaFIFOsInFreeBankOnly</code>	This option ensures DMA FIFOs are only inserted into memory banks that have no other buffers mapped. This option can be used when memory stalls are observed due to DMA FIFO buffers being accessed at the same time as some other design buffer placed in the same bank.
<code>disableSSFifoSharing</code>	Disables the ability of the router to share stream switch FIFOs among two or more terminals of a net. This option should only be used when there are not enough stream switch FIFOs in the device to give each terminal its own individual FIFO(s).

Viewing Compilation Results in the Vitis Analyzer

After the compilation of the AI Engine graph, the AI Engine compiler writes a summary of compilation results called `<graph-file-name>.aiecompile_summary` to view in the Vitis analyzer. The summary contains a collection of reports, and diagrams reflecting the state of the AI Engine application implemented in the compiled build. The summary is written to the working directory of the AI Engine compiler as specified by the `--workdir` option, which defaults to `./Work`.

To open the AI Engine compiler summary, use the following command:

```
vitis_analyzer ./Work/graph.aiecompile_summary
```

The Vitis analyzer opens displaying the Summary page of the report. The Report Navigator view lists the different reports that are available in the Summary. For a complete understanding of the Vitis analyzer, see [Using the Vitis Analyzer](#) in the *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393).

The listed reports include:

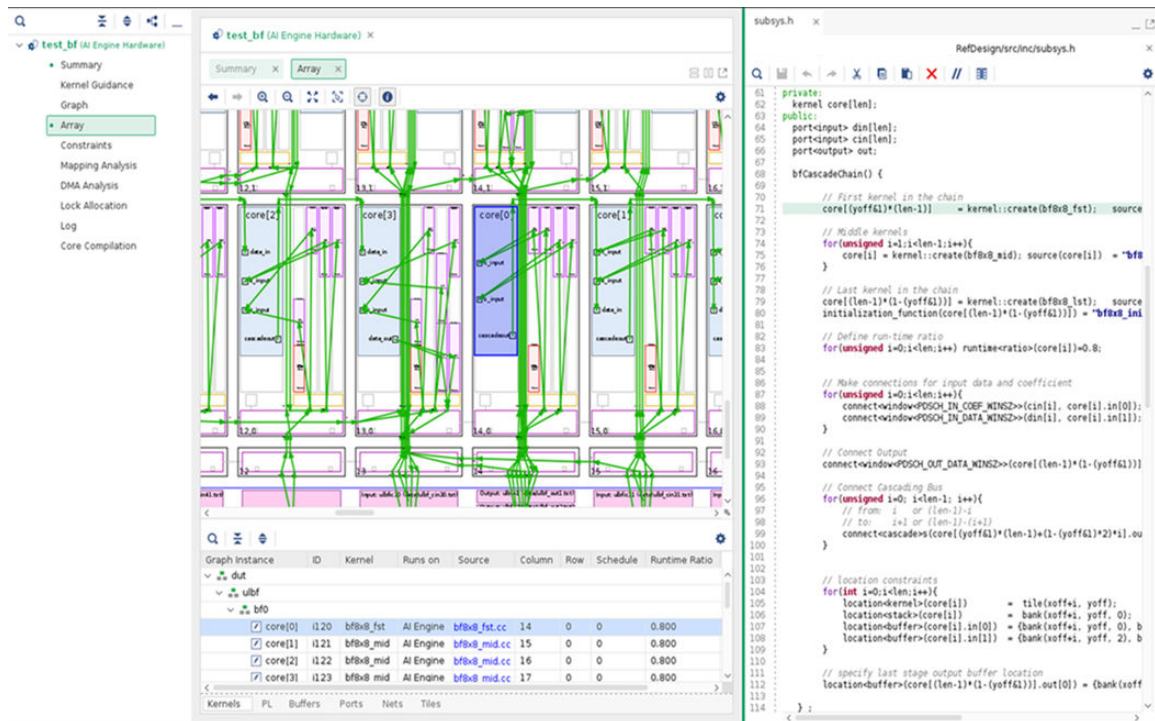
- **Summary:** This is the top-level of the report, and reports the details of the build, such as date, tool version, a link to the graph, and the command-line used to create the build.
- **Kernel Guidance:** Shows a variety of messages to provide guidance on kernel optimization.
- **Graph:** Provides a flow diagram of the AI Engine graph that shows the data flow through the various kernels. You can zoom into and pan the graph display as needed. At the bottom of the Reports view, a table summarizes the graph with information related to kernels, buffers, ports, and nets. Clicking on objects in the graph diagram highlights the selected object in the tables. (See [Graph and Array Details](#)).
- **Array:** Provides a graphical representation of the AI Engine processor array on the Versal device. The graph kernels and connections are placed within the context of the array. You can zoom into and select elements in the array diagram. Choosing objects in the array also highlights the object chosen in the tables at the bottom of the **Reports** view.

Note: The Graph and Array reports both share the same tables. When selecting an item in either of the views, it also selects it in both. For example, selecting a net in the Graph view, also selects it in the Array view.

- **Constraints:** Shows all constraints used within the graph and from .json constraint files.
- **Mapping Analysis:** Displays the text report `graph_mapping_analysis_report.txt`. Reports the block mapping, port mapping, and memory bank mapping of the graph to the device resources.
- **DMA Analysis:** Displays the text report `DMA_report.txt`, providing a summary of DMA accesses from the graph.
- **Lock Allocation:** Displays the text report `Lock_report.txt`, listing DMA locks on port instances.
- **Core Compilation:** Shows the single kernel compilation log file.

The following figure shows the `graph.aiecompile_summary` report open in the Vitis analyzer, with the Array diagram displayed, an AI Engine kernel selected in the diagram and the table views, and the source code for the kernel displayed in the **Source Code** view.

Figure 12: Vitis Analyzer Graph Summary



Graph and Array Details

In the graph and array views in Vitis analyzer are several tables highlighting the details of the graph and kernels. The following sections provide a detailed description of each of the tables and the information available in the columns in each of the different tables.

Kernels

The Kernels table shows detailed information about the kernels used by the ADF graph. For example, the following figure shows two kernels, interpolator and classify. The following example code shows the `fir_27t_sym_hb_2i` and `classifier` kernel functions being instantiated as kernels in the graph.

```
interpolator = kernel::create(fir_27t_sym_hb_2i);
classify = kernel::create(classifier);
```


Figure 13: Kernels Table

Graph Instance	ID	Kernel	Runs on	Source	Column	Row	Schedule	Runtime Ratio	Graph Source
clipgraph	i4	interpolator	AI Engine	hb27_2l.cc	25	4	0	0.800	graph.h:40:22
clipgraph	i5	classify	AI Engine	classify.cc	25	0	0	0.800	graph.h:42:18

Table 31: Column Description

Column	Description
Graph Instance	Shows a hierarchical view of the design graph along with the sub-graphs and kernels.
ID	Unique ID given to the kernel from <code>aiecompiler</code> .
Kernel	The kernel function name. This does not need to match the kernel instantiated name in the graph class. For example, the <code>fir_27t_sym_hb_2i</code> is the function name and instantiated as <code>interpolator</code> as seen in the preceding code.
Runs on	Where the kernel runs. This is always AI Engine.
Source	The kernel source file. Clicking this file name opens up the source file of the kernel.
Column	The column in the AI Engine where the kernel is mapped.
Row	The row in the AI Engine where the kernel is mapped.
Schedule	The order in which kernels, if mapped to the same tile (same Column, Row) executes. A 0 means no scheduling is set.
Runtime Ratio	The run-time ratio set in the graph by using <code>runtime<ratio>(<kernel>) = n</code> constraint.
Graph Source	The source file (<code>graph.h</code>) with line number where the kernel is instantiated. Clicking on the link opens up the source file at the line number.

Programmable Logic (PL)

The PL table, as shown in the following figure, provides detailed information about the PLIO connections to the ADF graph. For example, in this figure, there are four PLIO objects associated with the graph. The name of the PLIO connection, the width of the PLIO data connection, and the simulation test bench file associated with each PLIO connection is provided in the example.

```

PLIO *in0 = new PLIO("DataIn1", adf::plio_32_bits, "data/input.txt");
PLIO *ai_to_pl = new PLIO("clip_in", adf::plio_32_bits, "data/output.txt");
PLIO *pl_to_ai = new PLIO("clip_out", adf::plio_32_bits, "data/input2.txt");
PLIO *out0 = new PLIO("DataOut1", adf::plio_32_bits, "data/output2.txt");

```

Figure 14: PL Table

Name	Data Width	Frequency (Mhz)	Buffers	Connected Ports	Column	Channel	Packet IDs
Input: clip_out (data/input2.txt)	32	100.000	0	1	25	1	
Input: DataIn1 (data/input.txt)	32	100.000	2	1	25	5	
Output: clip_in (data/output.txt)	32	100.000	2	1	25	3	
Output: DataOut1 (data/output2.txt)	32	100.000	2	1	25	1	

Table 32: Column Description

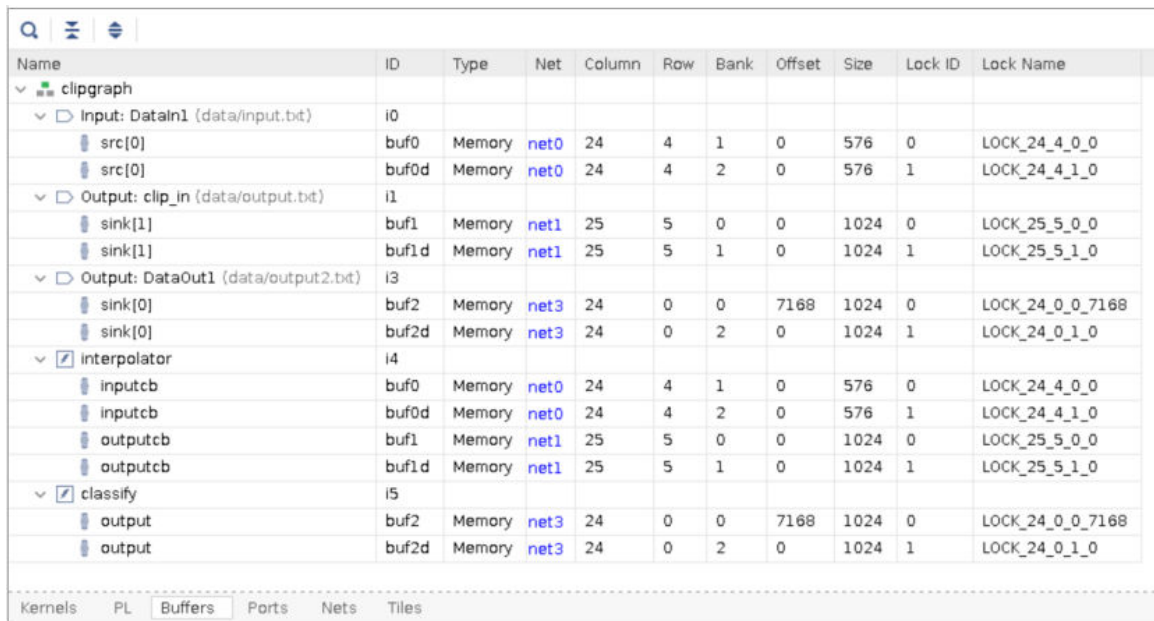
Column	Description
Name	The port name of a PLIO connection and whether it is an input or output.
Data Width	The data width of the PLIO connection defined in the constructor. The width can be either 32 bits, or 64 bits, or 128 bits.
Frequency (MHz)	The frequency (in MHz) defined (optionally) in the PLIO constructor for the PLIO connection.
Buffers	The number of buffers used in a PLIO connection. If a PLIO port is connected to a Window port of an AI Engine kernel two buffers are used, signifying a ping-pong buffer. A connection from a PLIO port to a stream port of the AI Engine kernel does not consume any buffers.
Connected Ports	The number of ports the PLIO is connected to. This PLIO data can be multicasted to multiple destinations in the AI Engine. For more information see Multicast Support .
Column	The interface column used by the PLIO, which is assigned by the <code>aiecompiler</code> . The values could be in the 0-49 range.
Channel	The channel within the interface column used by the PLIO.
Packet ID	The packet switching feature allows you to send packets of data to/from multiple destinations. These packets of data can be sent from/to the PL to/from the AI Engine. This column displays the ID of the packets used when packet switching is used. For more information see Explicit Packet Switching .

Buffers

The Buffers table contains information related to the buffers that are mapped to the ADF graph. Typically, buffers are used in window connections.

Note: The use of `buf#` and `buf#d` means it is a ping-pong buffer.

Figure 15: Buffers Table



Name	ID	Type	Net	Column	Row	Bank	Offset	Size	Lock ID	Lock Name
clipgraph										
Input: DataIn1 (data/input.txt)	i0									
src[0]	buf0	Memory	net0	24	4	1	0	576	0	LOCK_24_4_0_0
src[0]	buf0d	Memory	net0	24	4	2	0	576	1	LOCK_24_4_1_0
Output: clip_in (data/output.txt)	i1									
sink[1]	buf1	Memory	net1	25	5	0	0	1024	0	LOCK_25_5_0_0
sink[1]	buf1d	Memory	net1	25	5	1	0	1024	1	LOCK_25_5_1_0
Output: DataOut1 (data/output2.txt)	i3									
sink[0]	buf2	Memory	net3	24	0	0	7168	1024	0	LOCK_24_0_0_7168
sink[0]	buf2d	Memory	net3	24	0	2	0	1024	1	LOCK_24_0_1_0
interpolator	i4									
inputcb	buf0	Memory	net0	24	4	1	0	576	0	LOCK_24_4_0_0
inputcb	buf0d	Memory	net0	24	4	2	0	576	1	LOCK_24_4_1_0
outputcb	buf1	Memory	net1	25	5	0	0	1024	0	LOCK_25_5_0_0
outputcb	buf1d	Memory	net1	25	5	1	0	1024	1	LOCK_25_5_1_0
classify	i5									
output	buf2	Memory	net3	24	0	0	7168	1024	0	LOCK_24_0_0_7168
output	buf2d	Memory	net3	24	0	2	0	1024	1	LOCK_24_0_1_0

Table 33: Column Description

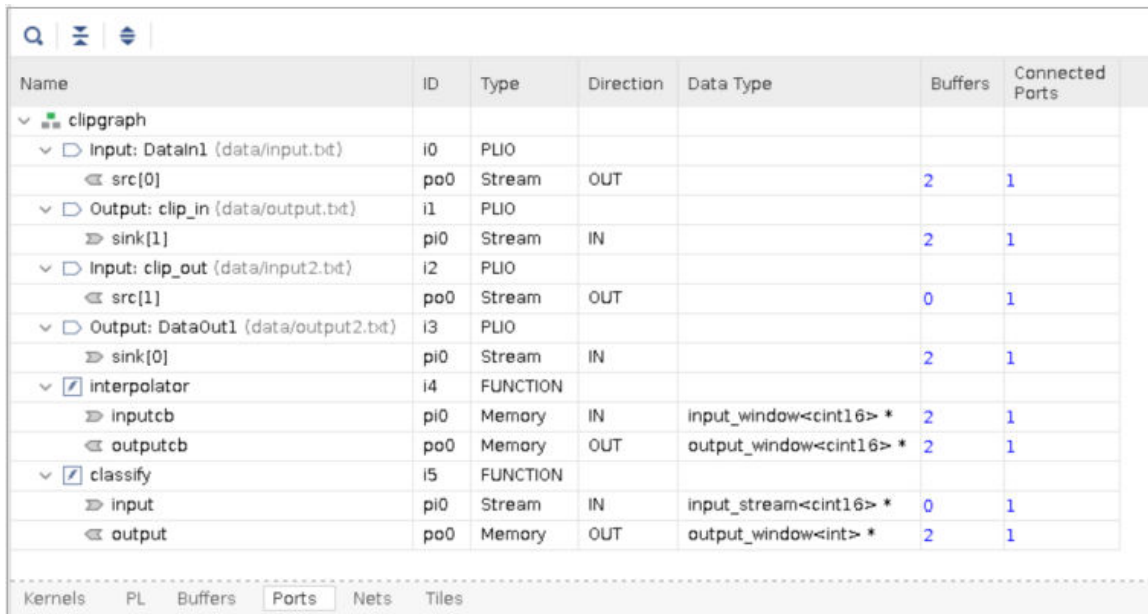
Column	Description
Name	The name of the connection where the buffer is allocated.
ID	The unique ID given to the buffer by the AI Engine compiler.
Type	The type of buffer being used. This can either be Memory or Stream. The connection to a window uses a ping-pong buffer, and connection to a stream might use a DMA buffer.
Net	The net with which the buffer is associated.
Column	The column location of the tile where the buffer is mapped by the compiler.
Row	The row location of the tile where the buffer is mapped by the compiler.
Bank	The bank of the tile where the buffer is mapped. The banks are: 0,1,2, or 3.
Offset	The address offset of the buffer with the bank.
Size	The size of the buffer in bytes.
Lock ID	Unique ID per buffer if placed in the bank.
Lock Name	The unique name of the lock associated with the buffer. This can be used to debug a lock stall on a buffer.

Ports

The Ports table contains all the ports of the design which can be GMIO ports, PLIO ports and input, inout, and output ports on a kernel .

Note: FileIO is displayed as PLIO in the Vitis analyzer.

Figure 16: Ports Table



Name	ID	Type	Direction	Data Type	Buffers	Connected Ports
clipgraph						
Input: DataIn1 (data/input.bit)	i0	PLIO				
src[0]	po0	Stream	OUT		2	1
Output: clip_in (data/output.bit)	i1	PLIO				
sink[1]	pi0	Stream	IN		2	1
Input: clip_out (data/input2.bit)	i2	PLIO				
src[1]	po0	Stream	OUT		0	1
Output: DataOut1 (data/output2.bit)	i3	PLIO				
sink[0]	pi0	Stream	IN		2	1
interpolator	i4	FUNCTION				
inputcb	pi0	Memory	IN	input_window<cint16> *	2	1
outputcb	po0	Memory	OUT	output_window<cint16> *	2	1
classify	i5	FUNCTION				
input	pi0	Stream	IN	input_stream<cint16> *	0	1
output	po0	Memory	OUT	output_window<int> *	2	1

Table 34: Column Description

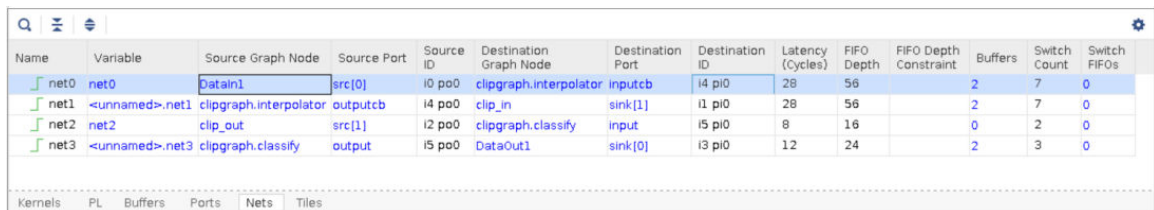
Column	Details
Name	The port name of the input, inout, output ports on a kernel, GMIO, or PLIO ports.
ID	The unique ID the AI Engine compiler designates the port.
Type	Port type. PLIO ports can contain Stream, Packet Switching, GMIO ports contain Global Memory, Function can contain Memory, or Stream.
Direction	Port direction. Can be: IN, OUT, INOUT.
Data Type	The type definition of the port for kernels. For example, <code>input_window<int16>*</code> , <code>input_stream<int16>*</code> .
Buffers	The number of buffers instantiated for the connection. For streaming connection, no buffers are used. For a window connection, it is a ping-pong buffer.
Connected Ports	The number of ports the specific port is connected to. Ports can multicast to more than one port. For more information see Multicast Support .

Nets

The Nets table shows details of the net connections made between the AI Engine kernels, or the AI Engine kernel and the PLIO/GMIO ports. For example, in the snippet of `graph.cpp` file shown in the following you can see examples of the connect constraint used to connect to the stream and window connections between the AI Engine kernels in the graph or to the PLIO/GMIO ports.

```
connect< window >(in, interpolator.in[0]);
connect< window, stream >(interpolator.out[0], clip_in);
connect< stream >(clip_out, classify.in[0]);
connect< window >(classify.out[0], out);
```

Figure 17: Nets Table



Name	Variable	Source Graph Node	Source Port	Source ID	Destination Graph Node	Destination Port	Destination ID	Latency (Cycles)	FIFO Depth	FIFO Depth Constraint	Buffers	Switch Count	Switch FIFOs
net0	net0	DataIn1	src[0]	i0 po0	clipgraph.interpolator	inputcb	i4 pi0	28	56		2	7	0
net1	<unnamed>.net1	clipgraph.interpolator	outputcb	i4 po0	clip_in	sink[1]	i1 pi0	28	56		2	7	0
net2	net2	clip_out	src[1]	i2 po0	clipgraph.classify	input	i5 pi0	8	16		0	2	0
net3	<unnamed>.net3	clipgraph.classify	output	i5 po0	DataOut1	sink[0]	i3 pi0	12	24		2	3	0

Table 35: Column Description

Column	Description
Name	The name of the net that is internally generated.
Variable	The name of the net connection (which can be optionally specified in the connect constraint). <code><unnamed>.net#</code> signifies that the <code>connect<></code> has no unique naming as part of the connect constraint in the graph.
Source Graph Node	The source node of the graph connection which could be a AI Engine kernel, PLIO or GMIO node.
Source Port	The source port of the graph connection which could be a AI Engine kernel, PLIO or GMIO port.






Table 35: Column Description (cont'd)

Column	Description
Source ID	The unique ID the <code>aiecompiler</code> designates the source port.
Destination Graph Node	The destination node of the graph connection which could be a AI Engine kernel, PLIO or GMIO node.
Destination Port	The destination port of the graph connection which could be a AI Engine kernel, PLIO or GMIO port.
Destination ID	The unique ID that the AI Engine compiler designates the destination port.
Latency (Cycles)	The minimum cycle count needed to transfer data from the source node to destination node.
FIFO Depth	The FIFO memory allocated through routing resources in the net. This includes buffers configured as DMA FIFOs, stream switch ports, and stream switch FIFOs. The unit for FIFO depth is 32-bit words.
FIFO Depth Constraint	This reflects the FIFO depth constraint provided in the design.
Buffers	The number of buffers used by the net connection.
Switch Count	The number of switches traversed by the net connection.
Switch FIFOs	The number of stream switch FIFOs used by the net connection.

Tiles

The Tiles table shows all the tiles that have mapped kernels and buffers in the ADF graph. For example, in this design there are five tiles used, where two of them contain kernels (Tile [25,0], and Tile [25,4]), and three of them have buffers mapped (Tile[24,0], Tile[24,4], Tile[25,5]).

Figure 18: Tiles Table

Tile	Column	Row	Kernels	Buffers
 Tile [24,0]	24	0	0	3
 Tile [24,4]	24	4	0	3
 Tile [25,0]	25	0	1	0
 Tile [25,4]	25	4	1	0
 Tile [25,5]	25	5	0	2

Kernels
PL
Buffers
Ports
Nets
Tiles

Table 36: Column Description

Column	Description
Tile	The tile ID.
Column	The column location of the tile.
Row	The row location of the tile.
Kernels	The number of kernels that are mapped to the tile.
Buffers	The number of buffers mapped to the tile. This includes buffers on nets and buffers inside the kernel.

AI Engine Compiler Guidance

After the AI Engine compiler completes the compilation of an AI Engine design it analyzes the design and provides guidance on how to improve the design based on AI Engine rules or best software practices. Some guidance might be corrected by AI Engine compiler automatically. The guidance file lists all the findings with severity, category with tile number, details, correction if done by the AI Engine compiler and suggested resolutions.

The guidance file, `guidance.html`, is located in the `Work/reports` directory. Use a web browser to review this guidance file. It is recommended that you update your design per the design guidance report before running the design in the simulator or on hardware.

The following are examples of the AI Engine compiler-generated guidance available in `Work/reports/guidance.html`.

- Variables are referenced before being initialized.

Id	Name	Severity	Impact	Full Name	Categories	Details	Resolution
1	AIE-MEM-04	advisory		Memory Management	Memory_25_0	Global variable 'state' reads its default/external initialization in kernel 'producer', and is not explicitly written before this read	Ensure that for all global variables, an initialization/write (default = zero) precedes the read. For more information, refer to UG1076.

- Global variables are initialized locally from a kernel implementation.

Id	Name	Severity	Impact	Full Name	Categories	Details	Resolution
1	AIE-MEM-04	advisory		Memory Management	Memory_24_0	Global variable 'state' reads its default/external initialization in kernel 'producer', and is not explicitly written before this read	Ensure that for all global variables, an initialization/write (default = zero) precedes the read. For more information, refer to UG1076.

- Array of data is not 128 bits (16 bytes) aligned.

Id	Name	Severity	Impact	Full Name	Categories	Details	Resolution
1	AIE-MEM-01	advisory		Memory Management	Memory_0_0	Alignment of global array lut1 is not 128 bits. Automatically correcting it.	Align global arrays to 16 byte boundary. Refer to the chess user manual for details on specifying the alignment constraints.

- Use of `__restrict` qualifier causes undefined behavior. This undefined behavior is exhibited running on hardware only.

Id	Name	Severity	Impact	Full Name	Categories	Details	Resolution
...
3	AIE-MEM-05	advisory		Memory Management	Memory_24_0	Store to restrict child pointer *_Z23equalizer_24tap_complexP12input_window6scint16EP13output_windowIS0_E::inputcb_addr must be used in a different block-level scope than the parent pointer	Ensure proper usage of restrict keyword. Refer to the formal definition of restrict in Section 6.7.3.1 of C99 Standard.
4	AIE-MEM-05	advisory		Memory Management	Memory_24_0	Store to restrict child pointer *_Z23equalizer_24tap_complexP12input_window6scint16EP13output_windowIS0_E::outputcb_addr must be used in a different block-level scope than the parent pointer	Ensure proper usage of restrict keyword. Refer to the formal definition of restrict in Section 6.7.3.1 of C99 Standard.

Note: The `--xlopt ≥1` option is required to compile a design that allows the AI Engine compiler to generate the appropriate guidance file.

Simulating an AI Engine Graph Application

This chapter describes the various execution targets available to simulate AI Engine applications at different levels of abstraction, accuracy, and speed. AI Engine graphs can be simulated in four different simulation environments.

The x86 simulator is a fast-functional simulator as described in [x86 Functional Simulator](#). It should be used to functionally simulate your AI Engine graph and is useful for functional development and verification of kernels and graphs. It, however, does not provide timing, resource, or performance information.

The AI Engine simulator (`aiesimulator`) models the timing and resources of the AI Engine array, while using transaction-level SystemC models for the NoC, DDR memory. This allows for faster performance analysis of your AI Engine applications and accurate estimation of the AI Engine resource use, with cycle-approximate timing information.

In both x86 functional simulator and AI Engine simulator, PL interface to AI Engine can be exercised through untimed external traffic. Similarly, both use the user graph's `main()` function as C-test bench to configure and control the AI Engine. This test bench data input and output is not timed, and the user graph's `main()` function acts as a virtual simulation platform.

When you want a fast functional simulation of the entire system including the AI Engine graph, the PL logic along with XRT-based host application to control the AI Engine and PL, you should use the Vitis™ software emulation flow. The software emulation flow enables using HLS-based kernels or C-models of RTL based kernels to be interfaced with AI Engine graph and controlled using host code which can also be run on hardware. This flow includes the x86 functional model of the AI Engine, and Arm® QEMU emulator modeling the PS (untimed).

Finally, when you are ready to simulate the entire system including AI Engine graph and PL logic along with XRT-based host application to control the AI Engine and PL, for a specific board and platform, you should use the Vitis hardware emulation flow. This flow includes the SystemC model of the AI Engine, transaction-level SystemC models for the NoC, DDR memory, PL Kernels (RTL), and PS (running on QEMU). You can also include RTL logic and test bench PL logic for your platform or design.

The following table lists the four simulation flows and whether they support functional or performance level debug, and the level of support for source code debug. It also recommends the usage of these simulation flows in appropriate stages of your AI Engine design development.

Table 37: Simulation Flows

Simulation Tool Flow	Functional Debug	Performance Analysis and Debug	Source Level Debug	Design Development Stage Usage
X86 Simulator	Yes	No	Yes	AI Engine kernel and graph debug
AI Engine Simulator	Yes	Yes (This simulation steps through the AI Engine assembly code and is useful in performance analysis, as well as optimization)	Allows stepping through the AI Engine compiler generated assembly code which aids in code optimization, however, source level visibility could be limited due to compiler optimization	AI Engine graph performance debug
Vitis Software Emulation	Yes	No	Yes	System level emulation and functional debug
Vitis Hardware Emulation	Yes	Yes	Possible—however, provides limited source level visibility due to compiler optimization	System level emulation and performance debug

Simulation Models

The following table lists the simulation flows and recommends the usage of these simulation flows in appropriate stages of your AI Engine design development and the type of simulation model used for the various Versal® architecture domains. The type of simulation model and the simulation tool flow used determine the accuracy of the simulation results.

Table 38: Simulation Models

Simulation Tool Flow	AI Engine Kernels	PL Kernels	PL Platform	NoC/DDR Model	PS Model
X86 Simulator	X86 threads	C/SystemC	N/A	N/A	N/A
AI Engine Simulator	SystemC	SystemC	N/A	SystemC	N/A
Vitis Software Emulation	X86 threads	SystemC/RTL/External traffic generators	N/A	N/A	QEMU
Vitis Hardware Emulation	SystemC	SystemC/RTL/External traffic generators	RTL/SystemC	SystemC	QEMU

Simulation Features

You can obtain profiling data when you run your design in `aiesimulator` or hardware emulation. Analyzing this data helps you gauge the efficiency of the kernels, the stall and active times associated with each AI Engine, and pinpoint the AI Engine kernel whose performance might not be optimal. This also allows you to collect data on design latency, throughput, and bandwidth. Details on running and analyzing profile data can be found in [Chapter 11: Performance Analysis of AI Engine Graph Application during Simulation](#).

The event trace feature allows you to capture and analyze a system-level view of program execution. It can be helpful in identifying problems during program execution including correctness and performance issues. The AI Engine architecture has direct support for generation, collection, and streaming of events as trace data during simulation and hardware emulation. Details on running and analyzing event trace data can be found in [Chapter 11: Performance Analysis of AI Engine Graph Application during Simulation](#).

The `x86simulator` and `aiesimulator` simulate the design via the `main()` function of `graph.cpp()`. Because QEMU emulation support is available for host applications in software emulation and hardware emulation, when you move to software emulation or hardware emulation you can create a host application targeting Baremetal or Linux-XRT and emulate the host application as well. Test bench data can be provided to the x86 simulator and AI Engine simulator via the graph's `main()` function which acts like a virtual test bench platform. There are various levels of test bench support available for the simulation flows. Test bench data can either be file based or via an external traffic generator. Additional details on this feature can be found in [Generating Traffic for Emulation](#).

The data flowing between the AI Engine kernels is available and can be viewed either as data snapshot files or via the Vitis analyzer GUI. Additional details on the x86 simulator snapshot feature can be found in [Data Snapshots](#). Additional information on data visualization of trace data on the results of running the AI Engine simulator can be found in [Trace View Data Visualization](#).

The following table lists the type of simulation features supported with the four simulation flows.

Table 39: Simulation Features

Simulation Tool Flow	Trace	Profile	Host Application	Test Bench Support	AI Engine Data Flow Visibility
X86 Simulator	No	No	Through the <code>main()</code> function in <code>graph.cpp</code>	File-based	Yes (via the snapshot feature)
AI Engine Simulator	Yes	Yes	Through the <code>main()</code> function in <code>graph.cpp</code>	File-based	Yes (via Trace view in Vitis Analyzer)
Vitis Software Emulation	No	No	Through the <code>main()</code> function in host application targeting BareMetal or Linux-XRT	File-based External traffic generators	Yes (via the snapshot feature)
Vitis Hardware Emulation	Yes	Yes	Through the <code>main()</code> function in host application targeting BareMetal or Linux-XRT	File-based External traffic generators	Yes (via Trace view in Vitis Analyzer)

x86 Functional Simulator

When starting development on AI Engine graphs and kernels, it is critical to verify the design behavior. This is called functional simulation and it is useful in identifying errors in the design. For example, window sizes in the graph are related to the number of iterations in a kernel. Troubleshooting to see if every kernel in a large graph is absorbing and generating the right number of samples can be time consuming and iterative in nature. The x86 simulator is an ideal choice for testing, debugging, and verifying this kind of behavior because of the speed of iteration and the high level of data visibility it provides the developer. The x86 simulation does not provide timing, resource, or performance information. The x86 simulator is running exclusively on the tool development machine. This means its performance and memory use are defined by the development machine.

While the AI Engine simulator fully models the memory of the AI Engine array this also means that AI Engine simulator is limited by the memory space of the AI Engine. The x86 simulator is not limited by this and provides a nearly unlimited amount of debug `printf()`s, large arrays, and variables. When combined with the ability to single step through the kernel, very complex design issues can quickly be isolated and addressed.

Several macros are provided to improve the kernel debug. These macros are intended for use with the x86 simulator and can be left in the code for maintainability purposes. With the benefits of the x86 simulator come some trade offs. There are several types of graph constructs that are not supported and cycle accurate behavior cannot be guaranteed to match between the AI Engine simulator and the x86 simulator. The x86 simulator is not a replacement for the AI Engine simulator. The AI Engine simulator must still be run on all designs to verify behavior and obtain performance information. For different stages of project development one tool or the other might better suit your needs.

To run the x86 simulator, change the AI Engine compiler target to `x86sim`.

```
aiecompiler --target=x86sim graph.cpp
```

After the application is compiled for x86 simulation, the x86 simulator can be invoked as follows.

```
x86simulator
```

The complete x86 simulator command help is shown in the following code.

```
$ x86simulator [-h] [--help] [--h] [--pkg-dir=PKGDIR]
optional arguments:
-h,--help --h show this help message and exit
--pkg-dir=PKG_DIR Set the package directory. ex: Work
--timeout=secs Terminate simulation after specified number of seconds
--dump Enable snapshots of data traffic on kernel ports
--gdb Invoke from gdb
--valgrind Run simulator under valgrind to detect access violations
--valgrind-gdb Run simulator under valgrind and debug via gdb server
```

```
--valgrind-args=ARGS Override default options for valgrind. Used in
conjunction with --valgrind.
--stop-on-deadlock Stop simulation if deadlock is detected
--trace Enable trace of kernel stall events
--trace-print Print kernel stall events during simulation
```

The compiled binary for x86 native simulation is produced by the AI Engine compiler under the `Work` directory (see [Chapter 9: Compiling an AI Engine Graph Application](#)) and is started automatically by the x86 simulator.

The input and the output files are specified in the following snippet of graph code.

```
adf::PLIO in1("In", adf::plio_32_bits, "In1.txt");
adf::PLIO out1("Out", adf::plio_32_bits, "Out1.txt");
simulation::platform<1,1> platform(&in1,&out1)
```

When running, the x86 simulator looks in the current working directory for `data/In1.txt` which is one of the inputs used by the ADF graph. To distinguish the output files for the x86 simulator from the output files for the AI Engine simulator the `Out1.txt` are located in `current_working_dir/x86simulator_output/data/`.

Optionally, the output files produced by the simulator can be compared with a golden output ignoring white space differences.

```
diff -w <data>/golden.txt <data>/output.txt
```

Compiling the Design

The GNU debugger allows for C/C++ debugging similar to an IDE based debugger. It allows setting of breakpoints, single stepping, stepping over functions, and multiple hit counts on breakpoints. For AI Engine kernel development the x86 simulator enables single step debugging of kernel code using GDB.

The `target` argument for the AI Engine compiler must be set to `x86sim` to use GDB.

```
aiecompile --target=x86sim graph.cpp
```

Additionally, compiling with the preprocessor directive `-O0` minimizes optimizations which improves debug visibility. If additional debug visibility is required it is possible to reduce the compiler optimization level. Passing the optimization parameter to the preprocessor can be done as follows.

```
aiecompile --target=x86sim --Xpreproc=-O0 graph.cpp
```

Data Snapshots

The `x86simulator --dump` data snapshot feature allows users to dump and inspect data traffic at kernel ports without using the debugger. This feature does not require any instrumentation of kernel code. It also helps diagnose simulation result mismatches between two versions of the same design with `x86simulator`. Data snapshots are supported for kernel ports including streams, windows, packet streams, cascade streams, and RTP ports. This feature is also supported for all platform level ports like PLIO, GMIO, RTP ports, as well as for PS accesses into a specific RTP port.

For input window ports, a data snapshot is taken each time the kernel acquires the window and the snapshot includes the margin. For output window ports, a snapshot is taken each time the kernel releases the window and it excludes the margin. In either case the snapshot consists of the kernel iteration count and sample data. One text file per kernel port is generated. In addition, the iteration count of the kernel is also recorded.

For example, a data snapshot for a graph with a chain of two kernels with window ports where the first window has a non-zero margin is as follows.

```
$x86simulator --pkg-dir=./Work --i=.. --dump
INFO: Reading options file './Work/options/x86sim.options'.
Processing './x86simulator_output/dump/x86sim_dump.data'
File Port direction Port type Data type Kernel or platform port
-----
platform_src_0.txt out window cint16 platform.src[0]
mygraph_first_in_0.txt in window cint16 mygraph.first.in[0]
mygraph_second_out_0.txt out window cint16 mygraph.second.out[0]
platform_sink_0.txt in window cint16 platform.sink[0]
mygraph_first_out_0.txt out window cint16 mygraph.first.out[0]
mygraph_second_in_0.txt in window cint16 mygraph.second.in[0]
Wrote './x86simulator_output/dump/platform_src_0.txt'
Wrote './x86simulator_output/dump/mygraph_first_in_0.txt'
Wrote './x86simulator_output/dump/mygraph_second_out_0.txt'
Wrote './x86simulator_output/dump/platform_sink_0.txt'
Wrote './x86simulator_output/dump/mygraph_first_out_0.txt'
Wrote './x86simulator_output/dump/mygraph_second_in_0.txt'
Simulation completed successfully returning zero
$> more ./x86simulator_output/dump/mygraph_first_in_0.txt
# port: mygraph.first.in[0]
# port_dir: in
# port_type: window
# data_type: cint16
# Iteration 1; snapshot 1
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 1
2 3
4 5
```

```

6 7
8 9
10 11
12 13
14 15
16 17
18 19
20 21
22 23
24 25
26 27
28 29
30 31
32 33
34 35
36 37
38 39
40 41
42 43
44 45
46 47
48 49
50 51
52 53
54 55
56 57
58 59
60 61
62 63
# Iteration 2; snapshot 2
48 49
50 51
52 53
54 55
56 57
58 59
60 61
62 63
1 0
3 2
5 4
7 6
9 8
11 10
13 12
15 14
17 16
19 18
21 20
23 22
25 24
27 26
29 28
31 30
33 32
35 34
37 36
39 38
41 40
43 42
45 44
47 46
49 48

```

```

51 50
53 52
55 54
57 56
59 58
61 60
63 62

```

For stream ports, a data snapshot is taken for each chunk of four bytes of data. Each snapshot includes the value of TLAST and the kernel iteration count. For cascade stream ports, the granularity is eight bytes.

For example, a data snapshot for a graph with three kernels connected via stream ports is as follows.

```

$ x86simulator --pkg-dir=./Work --i=.. --dump
INFO: Reading options file './Work/options/x86sim.options'.
Processing './x86simulator_output/dump/x86sim_dump.data'

File Port direction Port type Data type Kernel or platform port
-----
platform_src_0.txt out stream int32 platform.src[0]
fifo_graph_dist_in_0.txt in stream int32 fifo_graph.dist.in[0]
fifo_graph_aggr_out_0.txt out stream int32 fifo_graph.aggr.out[0]
platform_sink_0.txt in stream int32 platform.sink[0]
fifo_graph_comp0_in_0.txt in stream int32 fifo_graph.comp0.in[0]
fifo_graph_comp1_in_0.txt in stream int32 fifo_graph.comp1.in[0]
fifo_graph_dist_out_0.txt out stream int32 fifo_graph.dist.out[0]
fifo_graph_comp0_out_0.txt out stream int32 fifo_graph.comp0.out[0]
fifo_graph_aggr_in_0.txt in stream int32 fifo_graph.aggr.in[0]
fifo_graph_comp1_out_0.txt out stream int32 fifo_graph.comp1.out[0]
fifo_graph_aggr_in_1.txt in stream int32 fifo_graph.aggr.in[1]

Wrote './x86simulator_output/dump/platform_src_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_dist_in_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_aggr_out_0.txt'
Wrote './x86simulator_output/dump/platform_sink_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_comp0_in_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_comp1_in_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_dist_out_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_comp0_out_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_aggr_in_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_comp1_out_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_aggr_in_1.txt'
Simulation completed successfully returning zero

$ more ./x86simulator_output/dump/fifo_graph_dist_in_0.txt
# port: fifo_graph.dist.in[0]
# port_dir: in
# port_type: stream
# data_type: int32
# Iteration 1
0
1
2
3

```

```

4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
...

```

Deadlock Detection

AI Engine users can run into simulator hangs. A common cause is insufficient input data for the requested number of graph iterations, mismatch between production and consumption of stream data, cyclic dependency with stream, cascade stream or asynchronous windows, or wrong order of blocking protocol calls (acquisition of async window, read/write from streams).

You can use the `--stop-on-deadlock` option on `x86simulator` to detect such deadlocks. This option will enable `x86simulator` to automatically detect a broad category of deadlocks, stop the simulation, and print a message stating the simulation has been terminated prematurely because a deadlock has been detected. When the `x86simulator` is invoked with the `--stop-on-deadlock` option, `x86simulator` detects the deadlock and produces a deadlock diagnosis report.

Additionally, a graph is generated in `x86simulator_output/simulator_state_post_analysis.dot`. This is a `.dot` file that encodes a description of the graph in terms of a block diagram where the agents involved in the deadlock are highlighted in red. To get this file transformed into a `.png` file, you must use the `dot` program as follows.

```
dot -Tpng x86simulator_output/simulator_state_post_analysis.dot >
simulator_state_post_analysis.png
```



IMPORTANT! Absence of deadlock for x86 simulation does not mean absence of deadlock in SystemC simulation. X86 simulation does not model timing and resource constraints and thus there are fewer causes of deadlock. On the other hand, if x86 simulation deadlocks, SystemC simulation deadlocks as well, so it is beneficial to fix the deadlock in x86 simulation before proceeding with SystemC simulation.

Note: The `--stop-on-deadlock` option is not supported for software emulation or use cases with an external test bench.

Trace Report

Trace capability is used for debugging simulation hangs without the need for instrumenting kernel code or using `gdb`. Trace capability is not available in `sw_emu` flow or with external test bench. There are two use models.

The first use model applies to standalone x86 simulation without external test bench and with `--stop-on-deadlock` enabled. At the time that the x86 simulator detects deadlock, it prints a message indicating that the design is deadlocked and the simulation will be terminated and suggests to rerun the simulation with the trace option enabled. In this second simulation, the simulator produces a textual report on the events that occurred during the simulation. Studying this report can help you identify the root cause of the deadlock. The root cause can be as simple as insufficient input data in a data file of the simulation platform, or it can be more involved.

A second use model applies to simulations where `--trace` and `--timeout=secs` options are enabled. After the timeout expires, the simulator terminates and produces the trace report. As in the first use model, analysis of the trace report can help identify the root cause of the deadlock.

The following options are available:

- `--trace` to get a full trace report at the end of the simulation.
- `--trace-print` to get output displayed on the console while the simulation is running.

Trace Report Content

The trace report shows the sequence of events that occurred during the simulation of your design in the x86 simulator. The name of the generated file is `x86simulator_output/trace/x86sim_event_trace.data.txt`. The kind of events that are recorded include:

- The start of a kernel iteration
- The end of a kernel iteration
- The start of a stream stall, i.e., the start of a read from a stream port of a kernel that blocks because of lack of data
- The end of a stream stall, i.e., the point where a read from a stream port that initially blocked, finally returns
- The start of a lock stall, i.e., the start of an attempt to acquire a window port where the lock is initially not available
- The end of a lock stall, i.e., the point in time where an attempt to acquire a window port that initially blocked, finally returns

Output of `--trace-print` versus `--trace`

The output of `--trace-print` is not as polished as the file generated by `--trace`. If you want to quickly see what is going on in the simulation, or if you are planning to terminate the simulation via `CTRL-C` instead of `--timeout=secs`, use `--trace-print`. This option is also a good choice if your simulation does not terminate (deadlock or segmentation fault) and you do not want to specify `--timeout=secs` or `--stop-on-deadlock`.

The columns of the output of `--trace-print` contain the following information.

- Timestamp: It is the same as in `x86sim_event_trace.data.txt`.
- Internal name of the kernel (`x86sim_event_trace.data.txt` uses the user name).
- Event type.
- Numeric value whose meaning depends on the event type: It encodes the port that you are waiting on for a lock or stream stall. It encodes the iteration number of a start of an iteration event.

Memory Access Violations and Valgrind

Memory access violations occur when a kernel is reading or writing out of bounds of an object or reading uninitialized memory. This can manifest itself in multiple ways like a simulator crash or hang. It can also cause simulator results to be non-repeatable. The `x86simulator --valgrind` option will find memory access violations in kernel source code.

Note: Valgrind needs to be installed for this feature to work. Xilinx recommends using Valgrind version 3.16.1.

This option allows detection of memory access violations in kernel source code during x86 simulation with Valgrind. The following kinds of access violations can be detected.

- Out-of-bounds write
- Out-of-bounds read
- Read of uninitialized memory

There are two ways to use this option:

- `x86simulator --valgrind`: This runs the simulation with access violation detection turned on. At the end of the simulation, Valgrind prints a report on access violations. If there are none, the report ends with *ERROR SUMMARY: 0 errors from 0 contexts*. Otherwise, the report lists each access violation found. This includes a stack trace, which highlights the line number in the kernel source code where the access violations occurred.

- `x86simulator --valgrind-gdb`: This runs the simulation with access violation detection turned on and debug with GDB. The simulation comes up in GDB and is halted at `main()`. At this point you can set additional breakpoints. After continuing, the simulation will stop if an access violation is detected. At this point you can inspect local variables and the stack to diagnose the problem.

In either case, some arguments can be added to the Valgrind command using flag `--valgrind-args='list of arguments for valgrind'`. For example:

`--valgrind-args='-v --leak-check=no --track-origins=yes'` will not track memory leakage and would display the overall stack when discovering an access violation.

```
x86simulator --pkg-dir=Work --valgrind-gdb --valgrind-args='-v --leak-check=no --track-origins=yes'
==1021329== Memcheck, a memory error detector
==1021329== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1021329== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==1021329== Command: Work/pthread/sim.out
==1021329==
==1021329== (action at startup) vgdb me ...
==1021329==
==1021329== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==1021329==   /path/to/gdb Work/pthread/sim.out
==1021329== and then give GDB the following command
==1021329==   target remote | /tools/baton/valgrind/3.16.1/lib/valgrind/../../bin/vgdb --pid=1021329
==1021329== --pid is optional if only one valgrind process is running
==1021329==
```

Note: Running the `x86simulator` with the Valgrind option will increase the simulation run time.

Using GDB

After successful compilation with the appropriate target you can launch the simulation and automatically attach a GDB instance to it. To launch an interactive GDB session run the command with the switch `--gdb` as follows.

```
x86simulator --gdb
```

By default, when running the `x86 simulator` with the `--gdb` command line switch it breaks immediately before entering `main()` in `graph.cpp`. This pauses execution before any AI Engine kernels have started because the graph has not been run. To exit GDB type `quit` or `help` for more commands.

Setting a breakpoint can be done in multiple ways. One method is to use the following syntax.

```
break <kernel_function_name>
```

By typing `continue` (shorthand `c`) the debugger runs until the breakpoint in `<kernel function name>` is reached. When the breakpoint is reached it is possible to examine local stack variables and function arguments. The following table shows some commonly used GDB instructions to allow examination of these variables.

Table 40: Common GDB Instructions

Command	Description
<code>info stack</code>	Reports the function call stack at the current breakpoint.
<code>info locals</code>	Shows the current status of local variables within the scope of the function call shown in the call stack.
<code>print <local_variable_name></code>	Prints the current value of a single variable.
<code>finish</code>	Exits the current function call but keeps the simulation paused.
<code>continue</code>	Causes the debugger to run to completion.

GDB is a very powerful debugger with many features. Complete documentation of GDB is beyond the scope of this document. For GDB documentation, see <https://sourceware.org/gdb/current/onlinedocs/gdb.pdf> and <https://sourceware.org/gdb/current/onlinedocs/refcard.pdf>.

Macros

Xilinx provides several predefined compiler macros to help using the x86 simulator. In your top level graph test bench (usually called `graph.cpp`) it can be useful to use the following pre-processor macros with a conditional `#if` to help include or exclude appropriate code.

Table 41: Macros

Macro	Description
<code>__X86SIM__</code>	Use this predefined macro to designate code that is applicable only for the <code>x86sim</code> flow.
<code>__AIESIM__</code>	Use this predefined macro to designate code that is applicable only for the <code>aiesimulator</code> flow.
<code>X86SIM_KERNEL_NAME</code>	Use this macro with <code>printf()</code> to tag instrumentation text with the kernel instance name.

Note: For macros surrounded with underscores `_` there are two underscore characters at the front and behind.

The following is an example of the macro code.

```
myAIEgraph g;
#ifdef __AIESIM__ || defined(__X86SIM__)
int main()
{
    g.init();
}
```

```

    g.run(4);
    g.end();
    return 0;
}
#endif

```



TIP: The `__AIESIM__` macro is used in the AI Engine simulator only and `__X86SIM__` is applicable for the x86 simulator.

The previous example shows the `__X86SIM__` macro surrounding the `main()` which is used by a `graph.cpp` file. This `main()` must be excluded from emulation flows and these macros provide that flexibility. Additionally, consider using the `__X86SIM__` macro to selectively enable debug instrumentation only during x86 simulation.

Printf() Macros

The x86 simulator executes multiple kernels in parallel on separate threads. This means that `printf()` debug messages can often be interleaved and non-deterministic when viewed in standard output. To help identify which kernel is printing which line the `X86SIM_KERNEL_NAME` macro can be useful. The following is an example showing how to combine it with `printf()`.

Note: To use `X86SIM_KERNEL_NAME` you must include `adf/x86sim/x86simDebug.h` as shown in the following code.

```

#include <adf/x86sim/x86simDebug.h>

void simple(input_window_float * in, output_window_float * out) {
    for (unsigned i=0; i<NUM_SAMPLES;i++) {
        float val = window_readincr(in);
        window_writeincr(out, val+0.15);
    }
    static int count = 0;
    printf("%s: %s %d\n", __FILE__, X86SIM_KERNEL_NAME, ++count);
}

```

Using printf() in Kernels

Vector data types are the most commonly used types in AI Engine kernel code. To debug vector operations within a kernel it is helpful to use `printf`.

Using printf() with Vector Data Types

To `printf()` the native vector types you can use a technique as follows.

```

v4cint16 input_vector;
...
int16_t* print_ptr =(int16_t*)&input_vector;
for (int qq=0; qq<4;qq++) //4 here so we print two int16s, real + imag
per loop.
    printf("vector re: %d, im: %d\r\n", print_ptr[2*qq], print_ptr[2*qq+1]);
}

```

With the AI Engine simulator the `--profile` option is required in order to observe `printf()` outputs. With the x86 simulator no additional options are needed to enable `printf` calls. This is one of the benefits of the x86 simulator.



IMPORTANT! Xilinx recommends avoiding `std::cout` in kernel and host code. If `std::cout` is used its outputs can appear interleaved, given the multi-threaded nature of the x86 simulator. Using `printf()` is recommended instead.

Limitations

Memory Model

Kernels that require retaining state from one invocation (iteration) to the next can use global or static variable to store this state. Variables with static storage class, such as global variables and static variables are a cause of discrepancies between x86 simulation and AI Engine simulation. The root causes is that for x86 simulation, the sources files of all kernels are compiled into a single executable, whereas for AI Engine simulation each kernel targeting an AI Engine is compiled independently. Thus, if a variable with static storage class is referred to by two kernels and these kernels are mapped to the same AI Engine, the variable is shared for both x86 simulation and AI Engine simulation. However, if these kernels are mapped to different AI Engines, then the variable is still shared for x86 simulation, but for AI Engine simulation each AI Engine has its own copy and there is no sharing. This leads to mismatches between x86 simulation and AI Engine simulation if the variable is both read and written to by the kernels.

The preferred way of modeling state to be carried across kernel iterations is to use a C++ kernel class (see [C++ Kernel Class Support](#)). This avoids the pitfall of variables with static storage class. Alternatively the storage class of the global or static variable can be changed to `thread_local`, but just for x86 simulation. In this case, each instance of the kernel has its own copy of the variable in x86 simulation. This matches the behavior of AI Engine simulation if using the variable are mapped to different AI Engines. In the following example, the kernel carries the state across kernel iteration via global variable `delayLine` and static variable `pos`. This causes mismatches between x86 simulation and AI Engine simulation if there are multiple kernel instances using this source file. This can be avoided by changing the storage class of these variables to `thread_local`.

Original kernel source code:

```
// fir.cpp
#include <adf.h>
cint16 delayLine[16] = {};
void fir(input_window<cint16> *in1,
        output_window<cint16> *out1)
{
    static int pos = 0;
    ..
}
```

Reworked kernel source code:

```
// fir.cpp
#include <adf.h>
#ifdef __X86SIM__
cint16 delayLine[16] = {};
#else
thread_local cint16 delayLine[16] = {};
#endif
void fir(input_window<cint16> *in1,
        output_window<cint16> *out1)
{
#ifdef __X86SIM__
    static int pos = 0;
#else
    static thread_local int pos = 0;
#endif
    ..
}
```

Graph API Calls

Graph constructs for timed execution (`graph.wait(N)`, `graph.resume()` and `graph.end(N)`) behave differently in the AI Engine simulator and the x86 simulator. For the AI Engine simulator `N` specifies the processor cycle timeout, whereas for the x86 simulator it specifies a wall clock timeout in milliseconds. Thus, if your test bench uses timed execution the AI Engine simulator and x86 simulator might produce different results, in particular, the amount of output data produced might differ.

Preprocessor Considerations

When running the AI Engine compiler with a target of `x86sim` the compiler ignores the `--Xchess` option. This means that the `x86sim` flow does not support kernel-specific compile options.

To understand this better consider the following example. A common method of making compile-time modifications to C code is using preprocessor directives such as `#ifndef`. To control these preprocessor directives it is helpful to pass `#defines` through the command line compiler options. The following example code block takes two different actions based on a preprocessor directive.

```
void example_kernel()
{
#ifdef SIM
    printf("Sumulation Mode\n");
#else
    printf("Default Mode\n");
#endif
}
```

To define the SIM macro at compile time with the AI Engine compiler targeting hardware (hw) you can do the following.

```
aiecompiler -target=hw -Xchess="example_kernel:-DSIM"
```

Because the `-Xchess` argument is ignored when the compilation target is set to `x86sim`, SIM is not defined for the x86 simulator case and the output of the kernel is Default Mode.

If you need to specify preprocessor options with the x86 simulator you can do so using `aiecompiler -target=x86sim --Xpreproc` instead of `-Xchess`. It is important to note that any options passed in this manner applies to all source code and all target flows.

Table 42: AI Engine Compiler Command Line Options

Option	Description
<code>--Xchess=<string></code>	Can be used to pass kernel specific options to the CHES compiler that is used to compile code for each AI Engine. The option string is specified as <code><kernel-function>:<optionid>=<value></code> . This option string is included during compilation of generated source files on the AI Engine where the specified kernel function is mapped.
<code>--Xpreproc=<string></code>	Pass general option to the PREPROCESSOR phase for all source code compilations (AIE/PS/PL/x86sim). For example: <code>--Xpreproc=-D<var>=<value></code>

Simulation Output File Processing Considerations

The files associated with the simulation output(s) may not be written to completely after `graph.end()`. If your `main()` program is reading these files—to verify the results of the simulation, Xilinx recommends a wait of two seconds between `graph.end()` and opening of the files. In the following example, `main()` waits for two seconds.

```
// graph.cpp
#include "simpleGraph.h"
#include
#include
simpleGraph mygraph;
simulation::platform<1,1> platform("input1.txt", "output1.txt");
..
int main(int argc, char ** argv)
{
mygraph.init();
mygraph.run(4);
mygraph.end();

// wait 2 seconds after mygraph.end() before opening output files.
sleep(2);
#ifdef __X86SIM__
#define SIM_OUTPUT "aiesimulator_output"
#else
```

```
#define SIM_OUTPUT "x86simulator_output"
#endif
std::ifstream fOutput1(SIM_OUTPUT "/data/output1.txt");
..
}
```

adf::headers Constraint and aie_api Include Files

If a header file appears in a `adf::headers` constraint and that header file includes `aie_api/aie.hpp` directly or indirectly, `aiecompiler --target=x86sim` will fail. As a workaround, you can move the include of `aie_api/aie.hpp` from the kernel header to the kernel source file.

For example, the kernel `kernel1` has an `adf::headers` constraint (`project.h`), which specifies the header file, `kernels.h`.

project.h

```
#pragma once
#include "kernels.h"
class MyGraph: public adf::graph {
    adf::kernel kernel1;
    ..
    MyGraph()
    {
        kernel1 = adf::kernel::create(F1);
        adf::source(kernel1) = "kernels.cpp";
        adf::headers(kernel1) = {"kernels.h"};
        ..
    }
};
```

Header file, `kernels.h`, includes `aie_api/aie.hpp` (lines 3-5).

kernels.h

```
#pragma once
#include "adf.h"
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include "aie_api/utils.hpp"
void F1(input_stream<int32> *streamin,
        output_stream<acc80> *cascout, output_stream<int32> *streamout);
```

`aiecompiler --target=x86sim` fails.

Compiler Output

```
aiecompiler --target=x86sim project.cpp
...
In file included from ../aietools/include/aie_api/aie.hpp:70:0,
                 from ../../src/kernels.h:3,
                 from PthreadSim.cpp:11:
../aietools/include/aie_api/aie_declaration.hpp:62:12: error: multiple
```



```
definition of 'enum class aie_dm_resource'
enum class aie_dm_resource {
    ^~~~~~
In file included from ...aietools/include/adf/x86sim/x86sim.h:8:0,
                 from PthreadSim.cpp:8:
.../aietools/include/adf/x86sim/x86simResourceTypes.h:4:12: note: previous
definition here
enum class aie_dm_resource {
    ^~~~~~
```

There are several ways to work around this issue:

- The `adf::headers` constraint is needed only if the graph has graph scoped tables (see [Global Graph-scoped Tables](#)). If it does not, the `adf::headers` constraint can be omitted.
- The include of the `aie_api` headers can be moved from `kernels.h` to `kernel.cpp` because the high-level kernel API `aie_api/aie.hpp` is typically needed only in the kernel source file.

Packet Switching and RTP

The x86 simulator and AI Engine simulator have some important differences when it comes to packet switching and run-time parameters (RTP). See [Run-Time Parameter Specification](#) and [Explicit Packet Switching](#) if you are not already familiar with these constructs.

Both packet switching and RTPs exhibit behavior that can manifest itself as a difference between the AI Engine simulator and the x86 simulator. It is important to understand that this dissimilarity is still correct in both cases. As discussed in [Explicit Packet Switching](#), packet switched streams are non-deterministic in all design flows (x86 simulator, AI Engine simulator, and hardware emulation).

Packet Switching

Packet Stream connections have a field known as the packet ID. If the source of the packet ID field comes from within the ADF graph, the x86 simulator uses the canonical, zero-based indexing scheme for packet IDs. The first branch on the output of a split node has a packet ID equal to 0 followed by 1, 2, 3, etc. If the source of the packet ID field comes from outside the ADF graph there will be discrepancies between the AI Engine simulator and the x86 simulator. To resolve these discrepancies see [Packet Switching and the AI Engine Simulator](#) for additional information on providing custom packet IDs when the source is outside the ADF graph.

The nature of packet merging means that the x86 simulator and the AI Engine simulator produce non-deterministic results. If an AI Engine on a packet split branch finishes processing data before any of the other branches its data appears on the output of the packet merge first. Exactly which core finishes first is highly dependent on both the kernel code and the incoming data. So any downstream processing blocks must be prepared to handle this behavior.

Synchronous and Asynchronous RTP

Both synchronous and asynchronous run-time parameters are fully supported in the x86 simulator. However the precise timing and cycle accuracy of when an RTP update occurs differs between the x86 simulator and the AI Engine simulator. Asynchronous RTPs in particular do not affect a kernel on a specific known cycle by their very nature of being asynchronous. This is true of asynchronous RTPs in both the x86 simulator and the AI Engine simulator.

Note: The x86 simulator by its nature is a functional level simulator whereas the AI Engine simulator models cycles (approximately). It is expected there will be differences. Xilinx recommends you consider partitioning your design into pieces that benefit from the x86 simulator.

Best Practices

The x86 simulator works best with the following recommendations.

- Keep variables in kernels function scoped.
- Keep the x86 simulator test bench simple - use `init()`, `run(X)`, and `end()`.
- Keep variables in the ADF graph class scoped (see [Global Graph-scoped Tables](#) and [C++ Kernel Class Support](#)).

Software Emulation

Vitis™ software emulation flow offers a fast functional simulation of the entire system including the AI Engine graph, the PL logic along with XRT-based host application to control the AI Engine, and PL. The software emulation flow enables using HLS-based kernels or C-models of RTL based kernels to be interfaced with AI Engine graph and controlled using host code (which can also be run on hardware). This flow includes the x86 functional model of the AI Engine, and Arm® QEMU emulator modeling the PS (which is untimed). Additional details on the flow can be found in [Running Software Emulation](#).

Reusing x86 Simulator Options

The `x86simulator` generates an options file automatically when using command-line arguments and is stored in the `Work/options` directory. The resulting options file can be manually adjusted with other options as specified in the following table.

Note: Only a subset of the x86 simulator options are supported in the software emulation flow.

This file can be specified on the command-line option for software emulation using `launch_sw_emu.sh` as described in [Running Software Emulation](#). An example of the command line is as follows.

```
./launch_sw_emu.sh -aie-sim-options ${FULL_PATH}/Work/options/x86sim.option
```

Where the `${FULL_PATH}` is the full path to the file.

Table 43: X86 Simulator Options Supported in the Software Emulation Flow

X86 Sim Option	Value	Description
dump	<yes>/<no>	Dumps out the input and output data snapshots for every iteration of individual AI Engine kernel ports.
trace	<yes>/<no>	Provides a timestamped view of events that occurred during AI Engine graph execution. This includes stalls, graph start/stop, and iteration start/stop with respective timestamps.
trace-print	<yes>/<no>	Prints out trace in the terminal and exports trace file result

AI Engine SystemC Simulator

The Versal® ACAP AI Engine SystemC simulator (`aiesimulator`) includes the modeling of the global memory (DDR memory) and the network on chip (NoC) in addition to the AI Engine array. When the application is compiled using the SystemC simulation target, the AI Engine SystemC simulator can be invoked as follows.

```
aiesimulator --pkg-dir=./Work
```



IMPORTANT! Using AI Engine simulator requires the setup described in [Setting Up the Vitis Tool Environment](#).

The various configuration and binary files are generated by the AI Engine compiler under the `Work` directory (see [Chapter 9: Compiling an AI Engine Graph Application](#)) and specified using the `--pkg-dir` option to the simulator. The graph is initialized, run, and terminated by a control thread expressed in the `main` application. The AI Engine compiler compiles that control thread with a PS IP wrapper to be directly loaded into the simulator.

By default, the `graph.run()` option specifies a graph that runs forever. The AI Engine compiler generates code to execute the data flow graph in a perpetual `While` loop, thus simulation also runs perpetually. To create terminating programs for debugging, specify `graph.run(<number_of_iterations>)` in your graph code to limit the execution for the specified number of iterations. The specified number of iterations can be any positive integer value.

Note: `graph::run(-1)` specifies the graph that runs forever.

The AI Engine simulator command first configures the simulator as specified in the compiler generated `Work/config/scsim_config.json` file. This includes loading PL IP blocks and their connections, configuring I/O data file drivers, and configuring the NoC and global memory (DDR memory) connections. It then executes the specified PS application and finally exits the simulator.

The AI Engine simulator has an optional `--profile` option, which enables `printfs` in kernel code to appear on the console, and also generates profile information. Also, the `--dump-vcd <filename>` option generates a value change dump (VCD) for the duration of the simulation. The `--simulation-cycle-timeout <number-of-cycles>` can be used to exit the simulation after a given number of clock cycles.



IMPORTANT! If you do not provide either the clock cycles or the number of runs to `graph.run()` the simulation runs forever. You need to press **Ctrl+c** twice to exit the simulator.



TIP: You might observe cycle count differences between simulation runs on the same design. This is because the simulator waits for a few seconds for all pending transactions (such as DMA) to finish. During this wait time, the simulator process is still ticking but can be context-switched by the OS and total cycles can be different for each run. To ensure that the total cycles are the same for each run, you should use the AI Engine simulator `--simulation-cycle-timeout` option to stop the simulator on the exact cycle. The total cycles that appear on the profiling report are same on each run.

Note: Do not include `<iostream>` in the kernel code to enable `printfs`. The use of `#include <iostream>` in the kernel code results in a compilation error for both the x86 simulator and the SystemC simulators.

Simulator Options

The complete set of the AI Engine simulator (`aiesimulator`) options are described in this section. In most cases, just pointing to `pkg-dir` is sufficient.

Table 44: AI Engine Simulator Options

Options	Description
<code>-h, --help</code>	Show this help message and exit.
<code>--dump-vcd FILE</code>	Dump VCD waveform information into <code>FILE</code> . Because the tool appends <code>.vcd</code> to the specified file name, it is not necessary to include the file suffix.
<code>--gm-init-file <file></code>	Read global memory image from file. This loads the memory initialization file as described in Simulating Global Memory .
<code>--pkg-dir <PKG_DIR></code>	Specify the package directory, for example, <code>./Work</code> .
<code>--profile</code>	Generates profiling data for all used cores. Allows generation of <code>printf</code> trace messages on the <code>stdout</code> and collects profiling statistics during simulation. This can slightly slow down the simulator. Optionally, can specify the profile of specific cores by using <code>--profile=(col,row)(col,row)...</code>

Table 44: AI Engine Simulator Options (cont'd)

Options	Description
<code>--simulation-cycle-timeout CYCLES</code>	Run the application for a given number of cycles after it is loaded. TIP: Specify the <code>--simulation-cycle-timeout</code> option to end the simulation session after the specified number of timeouts. However, when specifying simulation timeout during the debug process, be sure to specify a large number of cycles because the debug will terminate when the timeout cycle is reached.
<code>--online [-ctf] [-wdb]</code>	Call <code>vcddanalyze</code> to parse VCD data on-the-fly, to optionally produce common trace format (CTF), or waveform database (WDB) output. TIP: The <code>--online</code> option and <code>--dump-vcd</code> option cannot be used together. If both options are specified, only <code>--online</code> option takes effect.
<code>--enable-memory-check</code>	Enables run-time program and data memory boundary access - checks. Any access violation will be reported as an [WARNING] message. By default this option is disabled.
<code>--output-time-stamp</code>	When this option is set to <code>no</code> , timestamps in the simulation output text files are no longer generated. Default value for this option is <code>yes</code> .

Simulation Input and Output Data Streams

The default bit width for input/output streams is 32 bits. The bit width specifies the number of samples per line on the simulation input file. The interpretation of the samples on each line of the input file is dependent on the data type expected and the PLIO data width. The following table shows how the samples in the input data file are interpreted, depending on the data type and its corresponding PLIO interface specification.

Table 45: Simulation Input Data Dependency on Data Type and PLIO Width

Data Type	PLIO 32 bit	PLIO 64 bit	PLIO 128 bit
	PLIO *in0 = new PLIO("DataIn1", adf::plio_32_bits)	PLIO *in0 = new PLIO("DataIn1", adf::plio_64_bits)	PLIO *in0 = new PLIO("DataIn1", adf::plio_128_bits)
int8	//4 values per line 6 8 3 2	//8 values per line 6 8 3 2 6 8 3 2	//16 values per line 6 8 3 2 6 8 3 2 6 8 3 2 6 8 3 2
int16	// 2 values per line 24 18	// 4 values per line 24 18 24 18	// 8 values per line 24 18 24 18 24 18 24 18
int32	// single value per line 2386	// 2 values per line 2386 2386	// 4 values per line 2386 2386 2386 2386
int64	N/A	45678	// 2 values per line 45678 95578
cint16	// 1 cint value per line – real, imaginary 1980 485	// 2 cint values per line 1980 45 180 85	// 4 cint values per line 1980 485 180 85 980 48 190 45
cint32	N/A	// 1 cint value per line – real, imaginary 1980 485	// 2 cint values per line 1980 45 180 85

Table 45: Simulation Input Data Dependency on Data Type and PLIO Width (cont'd)

Data Type	PLIO 32 bit	PLIO 64 bit	PLIO 128 bit
	PLIO *in0 = new PLIO("DataIn1", adf::plio_32_bits)	PLIO *in0 = new PLIO("DataIn1", adf::plio_64_bits)	PLIO *in0 = new PLIO("DataIn1", adf::plio_128_bits)
float	//1 floating point value per line 893.5689	//2 floating point values per line 893.5689 3459.3452	//4 floating point values per line 893.5689 39.32 459.352 349.345
cfloat	N/A	//1 floating point cfloat value per line, real, imaginary 893.5689 24156.456	//2 floating point cfloat values per line, real, imaginary 893.5689 24156.456 93.689 256.46

Simulating Global Memory

When an application accesses global memory using the GMIO specification (see [GMIO Attributes](#)), the simulation needs to model the DDR memory and the routing network connecting the DDR memory to the PL and AI Engines. AI Engine to DDR memory connections are mediated by the DMA data mover that is embedded in the AI Engine array interface and controlled through the GMIO APIs in the PS program. Connections to DDR memory from an AXI4-Stream port on a PL block are mediated by a soft GMIO data mover block, which is generated automatically by the AI Engine compiler for simulation purposes. The data mover converts the streaming interface from the PL blocks to memory-mapped AXI4 interface transactions over the NoC with a specific start address, block size, and burst size as shown in [GMIO Attributes](#).

While simulating with global memory, a memory data file can be supplied using an additional option, `--gm-init-file`, which initializes the DDR memory with predefined data. This file is a textual byte-dump of the DDR memory starting at a given address. The format of this file is as follows:

```
<startaddr>:
<byte>
<byte>
...
```

For example, the AI Engine simulator can be invoked with global memory initialization in the following way:

```
aiesimulator --pkg-dir=./Work --gm-init-file=dump.txt
```

The simulator also produces an output byte dump for the DDR memory used, in the simulation output directory (default: `aiesimulator_output`). The name of the output file is based on the internal location of the DDR memory bank (for example, `DDRMCMC_SITE_X1Y0.mem`) starting at the base address `0x0`. You can use this dump to verify the global memory transactions.

Hardware Emulation

To simulate the entire system, including AI Engine graph and PL logic along with XRT-based host application to control the AI Engine and PL, for a specific board and platform, you must use the Vitis hardware emulation flow. This flow includes the SystemC model of the AI Engine, transaction-level SystemC models for the NoC, DDR memory, PL Kernels (RTL), and PS (running on QEMU). You can also include RTL logic and test bench PL logic for your platform or design. Details on the flow can be found in [Running Hardware Emulation](#).

Reusing AI Engine Simulator Options

The AI Engine simulator generates an options file that lists the options used for simulating the AI Engine graph application. The options file is automatically generated when the AI Engine simulator is run. You can reuse the AI Engine simulator options used in the initial graph-level simulation run later in the system-level hardware emulation. You can also manually edit the options file to specify other options as required. The following table lists the options that can be specified in the `aiesim_options.txt` file. This file is located in the `aiesimulator_output` directory and is created if either option, `--dump-vcd` or `--profile` is used with the `aiesimulator` command. This file can be specified as part of the command line option to launch the hardware emulator using the `launch_hw_emu.sh` script as described in [Running the System in Hardware](#). An example command line is as follows.

```
./launch_hw_emu.sh \
-add-env VITIS_LAUNCH_WAVEFORM_BATCH=1 \
-aie-sim-options ${FULL_PATH}/aiesimulator_output/aiesim_options.txt
```

where `${FULL_PATH}` must be the full path to the file or directory.

Table 46: AI Engine Options for Hardware Emulation

Command	Arguments	Description
AIE_DUMP_VCD	<filename>	When AIE_DUMP_VCD is specified, the simulation generates VCD data and writes it to the specified <filename>.vcd.
AIE_PROFILE	All (1,2)(3,4)...	This option profiles either all the used AI Engines or selected AI Engines listed. Hardware Emulation generates profile data files in the <code>sim/behav_waveform/xsim</code> directory and can be viewed in the Vitis analyzer by opening the file <code>default.aierun-summary</code> . This option also logs the ADF kernel <code>printf</code> data to <code>sim/behav_waveform/xsim/simulate.log</code> file.



IMPORTANT! You must set the AI Engine compiler *workdir* environment variable to the *Work* directory generated by the AI Engine compiler using one of the following methods:

- Use `-add-env` in the command line of `launch_hw_emu.sh`. In the Makefile for example, `./launch_hw_emu.sh -aie-sim-options ./sim_options.txt -add-env AIE_COMPILER_WORKDIR=/yourdesigndirectory/Work`.
- Before launching Xilinx® simulator, type `setenv AIE_COMPILER_WORKDIR /yourdesigndirectory/Work` in the shell window.

Note: Any command that has path to a file needs to be an absolute path.

When creating a simulation option file manually it needs to follow the format of `COMMAND=ARGUMENT`, with each command being on a separate line. The following example shows best practice.

```
AIE_PROFILE=All
AIE_VCD=foo
```

The following command brings up the Xilinx simulator waveform GUI during hardware emulation.

```
./launch_hw_emu.sh -g
```

Additionally, you can add more advanced options to log waveform data without having to launch emulation with the Vivado logic simulator GUI. An example command line is as follows.

```
./launch_hw_emu.sh \
-user-pre-sim-script pre-sim.tcl
```

The `pre-sim.tcl` contains Tcl commands to add waveforms or log design waveforms. For an example, see the *Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#)) and for Tcl commands see *Vivado Design Suite User Guide: Logic Simulation* ([UG900](#)).

Enabling Third-Party Simulators

Third-party simulators such as Questa Advanced Simulator (Mentor Graphics), Xcelium (Cadence), and VCS (Synopsys) are supported when executing hardware emulation of your design. You can enable these simulators by updating the Vitis configuration file (`config.ini` or `system.cfg`).

Table 47: Vitis Link Settings

Simulator	v++ --link Configuration
Questa Advanced Simulator	<pre>EXPORT simulator=questa [advanced] param=hw_emu.simulator=QUESTA [vivado] prop=project.__CURRENT___.compplib.questa_compiled_library_dir=/path/to/questa/2020.4/lin64/lib/ prop=project.__CURRENT___.simulator.questa_install_dir=/path/to/questa</pre>
Xcelium	<pre>EXPORT simulator=xcelium [advanced] param=hw_emu.simulator=XCELIUM [vivado] prop=project.__CURRENT___.simulator.xcelium_install_dir=/path/to/xcelium/bin/ prop=project.__CURRENT___.compplib.xcelium_compiled_library_dir=/path/to/xcelium/20.09.006/lin64/lib/ prop=fileset.sim_1.xcelium.elaborate.xmelab.more_options={-timescale 1ns/1ps}</pre>
VCS	<pre>EXPORT simulator=vcs [advanced] param=hw_emu.simulator=VCS [vivado] prop=project.__CURRENT___.simulator.vcs_install_dir=/path/to/vcs/R-2020.12/bin/ prop=project.__CURRENT___.compplib.vcs_compiled_library_dir=/path/to/clibs/vcs/R-2020.12/lin64/lib/ prop=project.__CURRENT___.simulator.vcs_gcc_install_dir=/path/to/synopsys/vg-gnu/2019.06/amd64/gcc-6.2.0_64/bin prop=fileset.sim_1.vcs.simulate.log_all_signals=false</pre>

When the modifications have been made, build the design as normal, run the script `launch_hw_emu.sh` and the new simulator will be used. More information on emulation is provided in [Running the System in Hardware](#).

Performance Analysis of AI Engine Graph Application during Simulation

A system-level view of program execution can be helpful in identifying problems during program execution including correctness and performance issues. Problems such as missing or mismatching locks, buffer overruns, and incorrect programming of DMA buffers are examples that are difficult to debug by using explicit print statements or by using traditional interactive debuggers. A systematic way of collecting system level traces for the program execution is needed. The AI Engine architecture has direct support for generation, collection, and streaming of events as trace data during simulation, hardware emulation, or hardware execution.

AI Engine Simulation-Based Performance Analysis

In simulation, to view time stamped events, different event types, and data associated with each event, value change dump (VCD) files can be used. VCD files provide a detailed dump of the simulated hardware signals. Additionally, a profile summary provides annotated details for the overall application performance.

AI Engine Simulation-Based Profiling

Profiling data generation

In the simulation framework, the AI Engine simulator can generate a profiling report for the complete application. This report is generated using the flag `-profile`.

```
aiesimulator -pkg-dir=Work -profile
```

Text files and xml files are generated in the directory `aiesimulator_output`. Two types of files are generated for the tile located in column C and row R. The `*_funct` reports the number of calls and number of cycles for each function. The `*_instr` is a report that goes down to the assembly code. To visualize the report, use the Vitis analyzer.

```
vitis_analyzer aiesimulator_output/default.aierun_summary
```

The Profile tab opens the Profile report, which shows a menu of sections that show information.

- **Summary:** Reports the total cycle count, total instruction count, and program size in memory.
- **Function Reports:** Shows several key indicators, function by function, in a table and graphs.
 - Number of calls
 - Total function time (cycles and %)
 - Total function + descendant time (cycles and %)
 - Min/Avg/Max function time (cycles)
 - Min/Avg/Max function + descendant time (cycles)
 - Program counter Low/High
- **Profile Details:** Shows the assembly code, function by function, with useful precisions. The columns are as shown in the following table.

Table 48: Profile Details Column Description

Column Name	Content
PC	Program counter
Instruction	Up to 16 bytes for each line
Assembly	Assembly code mnemonic with the full 7-way instruction word
Exe-count	Number of times this line has been executed by the processor
Cycles	Number of cycles required
User Count	
Wait States	For some instructions you may have memory conflicts which end up into a number of wait-states
Relative cycle use within function	Shown as '*' lines where the relative length visually shows the relative cycles use of this instruction within the function
Relative cycle use within simulation	Shown as '*' lines where the relative length visually shows the relative cycles use of this instruction within the simulation (including <code>main()</code> and all functions)
Relative wait-state use within function	Shown as 'W' lines where the relative length visually shows the relative cycles used by wait-states during this instruction within the function
Relative wait state use within simulation	Shown as '*' lines where the relative length visually shows the relative cycles used by wait states during this instruction within the simulation (including <code>main()</code> and all functions)

Performance Debug with Profiling Data

Performance improvement of a design should start by optimizing the function that takes most of the cycles. After it is done, you can optimize functions that take less and less proportion of the cycles. For this purpose, the total function time graph will help you in selecting these functions.

For the optimization itself, you can use pragmas (chess_prepare_for_pipelining, chess_loop_range) for the usual pipelining, unrolling of loops. The Profile Details tab provides insight about wait states. Even if your inner-loop is perfectly optimized, you may lose some cycles due to wait-states. These wait-states occur when there are conflicts in resource access.

- Two reads or one read and one write within the same memory bank, either from the local AI Engine, or two contiguous AI Engines.
- The local AI Engine tries to access a bank (either read or write) while a memory DMA is accessing it for some data transfer.

Here is an example of profile details.

PC	Instruction	Assembly	Exe-count	Cycles	User	Count	Wait	states
1296 30 00 10 ff d4 02 90 04 8a df 60 3f		MOV.s10 m0, #32; MOV.s9 r9, #2; PADDS [p0], #0; MOV.u20 p3, #171872; NOP	4	4	0	0	0	0
1308 08 ac 49 68 67 d5 06 06 22 01 01 3f		PADDA [p1], #24; VLDB w0, [p3]; LSH r12, r6, r9; MOV.u20 c10, #557313; NOP	4	4	0	0	0	0
1320 0a a8 18 c0 04 ff 98 04 02 80 00 3f		LDA p2, [p0], #16; MOV.s9 r8, #0; ADD r12, r12, #1; MOV.u20 c12, #0; NOP	4	4	0	0	0	0
1332 41 a8 16 c0 67 d3 92 04 02 f2 10 3f		LDA cbl, [p0], #12; MOV.s9 r6, #7; LSH r9, r12, r9; MOV.u20 c13, #12816; NOP	4	4	0	0	0	0
1344 49 00 17 c1 65 00 10 00 12 00 01 3f		LDA cs0, [p1]; MOV.s9 r7, #11; ADD.NRW s0, r7, #1; MOV.s12 c10, #1; NOP	4	4	0	0	0	0
1356 40 a8 50 c0 83 9f 3e 04 00 00 00 3f		LDA cbl, [p1], #12; MOV.s9 r5, #4; ADD r1, r9, #1; MOV.u20 cbl, #0; NOP	4	4	0	0	0	0
1368 48 b2 51 c0 68 06 7a 04 22 41 01 3f		LDA cs0, [p1], #28; MOV.s9 r1, #3; MOV.s12 r13, #12; MOV.u20 c11, #33025; NOP	4	4	0	0	0	0
1380 09 00 50 c1 60 00 01 00 0f 06 9a df		LDA p1, [p1]; MOV.s9 r0, #15; NOP; MOV.s12 r12, #8; MOV p3, p1	4	4	0	0	0	0
1392 31 04 12 c2 70 00 00 00 12 40 01 3f		MOV.s10 m1, #16; MOV.s9 r2, #19; NOP; MOV.s12 c11, #1; NOP	4	4	0	0	0	0
1404 80 00 00 00 40 04 00 07 f7		NOP; MOV.u20 cbl, #0; NOP	4	4	0	0	0	0
1412 80 00 00 00 40 60 62 07 f7		NOP; MOV.u20 s, #1588; NOP	4	4	0	0	0	0
1420 80 00 00 00 40 6c 68 07 f7		NOP; MOV.u20 le, #1664; NOP	4	4	0	0	0	0
1428 00 01		NOP	4	4	0	0	0	0
1430 1a 10 80 00 00 00 00 00 37		VLDA w0, [p2], #0, cycl; NOP; NOP	4	4	0	0	0	0
1438 1e 10 85 68 40 00 00 00 17		VLDA w0, [p2], #0, cycl; VLDB vr2, [p2]; NOP	4	4	0	0	0	0
1446 00 01		NOP	4	4	0	0	0	0
1448 aa c5		MOV p4, p2	4	4	0	0	0	0
1450 0e 10 b8 03		NOP; VLDB vr3, [p2], #0, cycl	4	8	0	4	0	0
1454 00 01		NOP	4	4	0	0	0	0
1456 aa d5		MOV p5, p2	4	4	0	0	0	0
1458 00 01		NOP	4	4	0	0	0	0
1460 0a 10 b8 03		NOP; VLDB vr2, [p2], #0, cycl	4	4	0	0	0	0
1464 1a d1 00 00 00 00 00 00 37		VLDA w0, [p4]; NOP; NOP	4	4	0	0	0	0
1472 00 01		NOP	4	4	0	0	0	0
1474 aa c5		MOV p4, p2	4	4	0	0	0	0
1476 00 00 09 15 01 00 3c 80 09 3d a0 8f		NOP; NOP; VMOV xa, xb; NOP; VMUL.48 an10, yd.c16, r9, c3, r7, wc0.s16, #0, c2, c0	4	4	0	0	0	0
1488 1e d1 47 08 50 00 00 00 00 00 94 11 23 3f		VLDA w0, [p5]; VLDB vr3, [p2], #0, cycl; NOP; NOP; NOP; VMAC.48 an10, ya.c16, r8, c3, r6, wc0.s	4	4	0	0	0	0
1504 80 00 01 00 12 33 41 47		NOP; NOP; VMUL.48 an11, yd.c16, r8, c3, r6, wc0.s16, #0, c2, c0	4	4	0	0	0	0
1512 80 00 01 41 12 7b 45 47		NOP; NOP; VMAC.48 an11, yd.c16, r9, c3, r7, wc0.s16, #4, c2, c1	4	4	0	0	0	0
1520 00 00 00 00 10 00 00 46 ab 40 00 92 01 34 b4 10 NOP; NOP; NOP; MOV p5, p2; NOP; VMO.48 an10, yd.c16, r13, c3, r1, wc0.s16, #0, c2, c0			4	8	0	0	0	0
1536 1a d1 05 08 50 00 00 00 00 00 92 01 30 34 14 VLDA w0, [p4]; VLDB vr2, [p2], #0, cycl; NOP; NOP; NOP; VMUL.48 an11, yd.c16, r12, c3, r0, wc0.			4	4	0	0	0	0
1552 00 00 00 00 10 00 00 64 54 00 00 96 11 35 34 54 NOP; NOP; NOP; VMOV xa, xb; NOP; VMAC.48 an11, yd.c16, r13, c3, r2, wc0.s16, #4, c2, c1			4	4	0	0	0	0
1568 00 00 01 aa c1 00 3c b0 69 01 a2 0f		NOP; NOP; MOV p4, p2; NOP; VMAC.48 an10, ya.c16, r12, c3, r0, wc0.s16, #4, c2, c1	56	56	0	0	0	0
1580 00 00 09 15 01 00 3c 80 09 3d a0 8f		NOP; NOP; VMOV xa, xb; NOP; VMUL.48 an10, yd.c16, r9, c3, r7, wc0.s16, #0, c2, c0	56	112	0	0	0	0
1592 1e d1 47 08 50 00 00 00 00 00 94 11 23 3f		VLDA w0, [p5]; VLDB vr3, [p2], #0, cycl; NOP; NOP; NOP; VMAC.48 an10, ya.c16, r8, c3, r6, wc0.s	56	56	0	0	0	0
1608 00 20 48 00 12 33 41 47		NOP; VST.48 SRSS an11, s0, [p1], #1, cycl; NOP; VMUL.48 an11, yd.c16, r9, c3, r6, wc0.s16, #0, c2, c0	56	56	0	0	0	0
1616 00 00 00 1c 00 00 00 00 00 94 11 27 b4 54		NOP; VST.48 SRSS an10, s0, [p1], #1, cycl; NOP; NOP; VMAC.48 an11, yd.c16, r9, c3, r7, wc0.	56	56	0	0	0	0
1632 00 00 00 00 1c 14 09 46 ab 40 00 92 01 34 b4 10 NOP; NOP; VST.48 SRSS an11, s0, [p1], #1, cycl; NOP; MOV p5, p2; NOP; VMUL.48 an10, yd.c16, r13, c3.			56	112	0	0	0	0
1648 1a d1 05 08 5c 10 00 00 00 00 92 01 30 34 14 VLDA w0, [p4]; VLDB vr2, [p2], #0, cycl; VST.48 SRSS an10, s0, [p1], #1, cycl; NOP; NOP; NOP; VMUL.48			56	56	0	0	0	0
1664 00 00 00 10 00 00 64 54 00 00 96 11 35 34 54 NOP; NOP; NOP; VMOV xa, xb; NOP; VMAC.48 an11, yd.c16, r13, c3, r2, wc0.s16, #4, c2, c1			56	56	0	0	0	0
1680 00 00 0f 91 a1 00 3c b0 69 01 a2 0f		NOP; NOP; MOV.s8 cs2, #28; NOP; VMAC.48 an10, ya.c16, r12, c3, r0, wc0.s16, #4, c2, c1	4	4	0	0	0	0

In this screenshot, the inner-loop is first localized with *ls* and *le* which are the loop-start and loop-end PC. This inner loop is shown within the blue rectangle. This seems correct as the Exe-Count is much higher on these lines than on the others.

In the second step, an instruction that is constantly taking two clock cycles instead of one, due to wait-states is localized:

- Exe-count = 56
- Cycles = 112

The VMUL instruction takes data in register `yd` and coefficients in `wc0`. A load taking seven clock cycles before the data is effectively in the register. The third step is to analyze the code seven clock cycles before the first iteration (3) or within the loop on the previous iteration (3'). You can see here that there are two loads in each case which are on the same bank (all the reads are from the same window in the source code).

AI Engine Simulation-Based Value Change Dump

In the simulation framework, the AI Engine simulator can generate a detailed dump of the hardware signals in the form of value change dump (VCD) files. A defined set of abstract events describes the execution of a multi-kernel AI Engine program in terms of these events. The output of a VCD file is enabled using the `aiesimulator --dump-vcd` command.

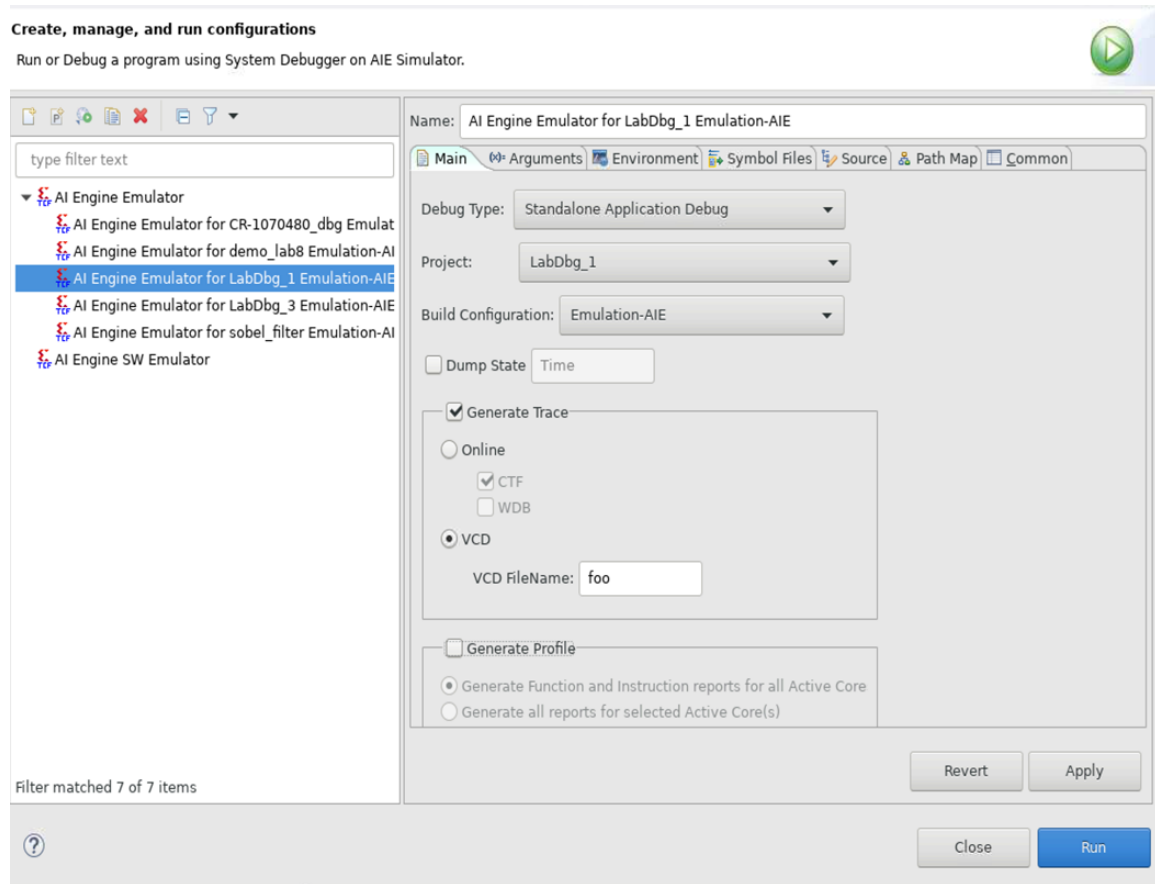
After simulation, or emulation, the VCD file can be processed into events and viewed on a timeline in the Vitis™ analyzer. The events contain information such as time stamps, different event types, and data associated with each event. This information can be correlated to the compiler generated debug information. This includes program counter values mapped to function names and instruction offsets, and source level symbolic data offsets for memory accesses.

The abstract AI Engine events are independent of the VCD format and will be directly extracted from the hardware. The events traces can be produced as plain text, comma-separated values (CSV), common trace format (CTF), or in waveform database (WDB), and the generated event trace data can be viewed in the Vitis analyzer.

VCD File Generation

To generate a VCD file from the Vitis IDE, right-click on your AI Engine graph project from the **Explorer** view and select **Run As → Run Configurations** as described in [Creating the AI Engine Graph Project and Top-Level System Project](#). This opens up the Run Configurations dialog box for the current project.

Figure 19: Vitis IDE to Enable VCD File Generation



Select the **AI Engine Emulator** option and double click to open a new configuration. Select the **Generate Trace** check box to enable trace capture, and select the **VCD Trace** button. By default, this produces a VCD dump in a file called `foo.vcd` in the current directory. You can rename the file if you like.

The VCD file can also be generated by invoking the AI Engine simulator with the `--dump-vcd <filename>` option on the command line. The VCD file is generated in the same directory as the simulation is run. Assuming that the program is compiled using the AI Engine compiler, the simulator can be invoked in a shell with the VCD option.

```
$ aiesimulator --pkg-dir=./Work --dump-vcd=foo
```

This command produces the VCD file (`foo.vcd`) which is written to the current directory.

AI Engine Trace from VCD

The `vcdanalyze` utility is provided to generate an AI Engine event trace from the VCD file. This process is integrated into the Vitis tool flow automatically. From the Vitis IDE, after a simulation run has finished capturing AI Engine events, you can right-click on the project from the **Explorer** view and select **Analyze AIE Events**. The trace data is produced under the current project at `Traces/AIE_AXI_Trace` and various views are automatically loaded into the current project.

The raw event trace under the directory `Traces/AIE_AXI_Trace/ctf/events.txt` should look like the following:

```
time=1741000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0
.data0,col=0,streamid=0,data0=65536,data1=0,tlast=0
time=1742000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0
.data0,col=0,streamid=0,data0=196610,data1=0,tlast=0
time=1743000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0
.data0,col=0,streamid=0,data0=327684,data1=0,tlast=0
time=1743000,event=CORE_RESET,col=1,row=0
time=1744000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0
.data0,col=0,streamid=0,data0=458758,data1=0,tlast=0
time=1745000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0
.data0,col=0,streamid=0,data0=589832,data1=0,tlast=0
time=1746000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0
.data0,col=0,streamid=0,data0=720906,data1=0,tlast=0
time=1747000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0
.data0,col=0,streamid=0,data0=851980,data1=0,tlast=0
time=1748000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0
.data0,col=0,streamid=0,data0=983054,data1=0,tlast=0
time=1749000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0
.data0,col=0,streamid=0,data0=1,data1=0,tlast=0
time=1750000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0
.data0,col=0,streamid=0,data0=131075,data1=0,tlast=0
time=1751000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0
.data0,col=0,streamid=0,data0=262149,data1=0,tlast=0
time=2186000,event=DM_WRITE_REQ,col=0,row=0,port=tl.me.array.tile_0_1.mm.dm.
port_AXI_write_b6
time=2190000,event=DM_WRITE_REQ,col=0,row=0,port=tl.me.array.tile_0_1.mm.dm.
port_AXI_write_b7
time=2194000,event=DM_WRITE_REQ,col=0,row=0,port=tl.me.array.tile_0_1.mm.dm.
port_AXI_write_b6
time=2198000,event=DM_WRITE_REQ,col=0,row=0,port=tl.me.array.tile_0_1.mm.dm.
port_AXI_write_b7
time=2202000,event=DM_WRITE_REQ,col=0,row=0,port=tl.me.array.tile_0_1.mm.dm.
port_AXI_write_b2
time=2206000,event=DM_WRITE_REQ,col=0,row=0,port=tl.me.array.tile_0_1.mm.dm.
port_AXI_write_b3
```

The following command produces the AI Engine trace data for `foo.vcd` in text form in the `./trdata/events.txt` file.

```
vcdanalyze -vcd foo.vcd
```



TIP: Use `vcdanalyze -h` to get help for the command.

The following command produces a CSV file from the AI Engine trace data from the `foo.vcd` file.

```
vcdanalyze -vcd=foo.vcd -csv
```

The following command produces the waveform data files from the AI Engine trace data from the `foo.vcd` file.

```
vcdanalyze -vcd foo.vcd -wdb
```

Viewing the Run Summary in the Vitis Analyzer

After running the system, whether in simulation, hardware emulation, or in hardware, a `run_summary` report is generated when the application has been properly configured.

During simulation of the AI Engine graph, the AI Engine simulator or hardware emulation, captures performance and activity metrics and writes the report to the output directory `./aiesimulator_output` and `./sim/behav_waveform/xsim`. The generated summary is called `default.aierun_summary`.

The `run_summary` can be viewed in the Vitis analyzer. The summary contains a collection of reports, capturing the performance profile of the AI Engine application captured as it runs. For example, to open the AI Engine simulator run summary use the following command:

```
vitis_analyzer ./aiesimulator_output/default.aierun_summary
```

The Vitis analyzer opens displaying the Summary page of the report. The Report Navigator view of the tool lists the different reports that are available in the summary. For a complete understanding of the Vitis analyzer, see [Using the Vitis Analyzer](#) in the *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393).

Note: The `default.aierun_summary` also contains the some of the same reports as `<GRAPH_TB_FILE_NAME>.aiecompile_summary`. These reports are Graph and Array. To see those reports go to the [Viewing Compilation Results in the Vitis Analyzer](#).

Report Summary

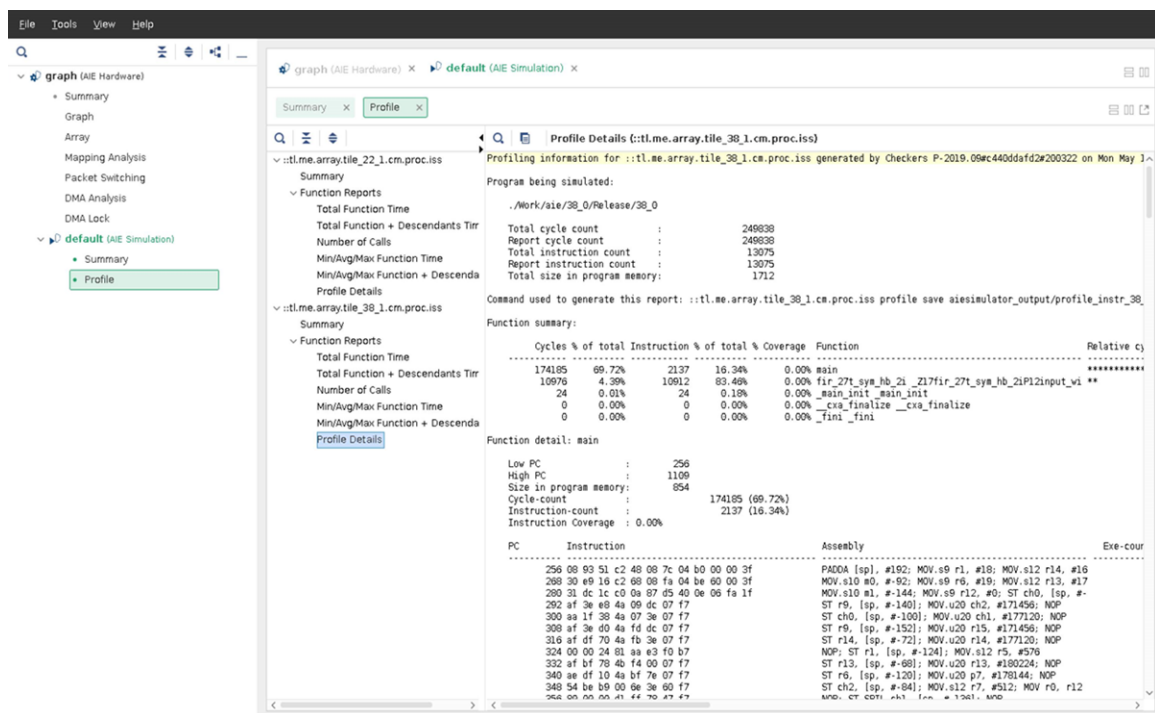
This is the top-level of the report, and reports the details of the run, such as date, tool version, and the command-line used to launch the simulator.

Profile Summary

When the `aiesimulator --profile` option is specified, the simulator collects profiling data on the AI Engine graph and kernels presenting a high-level view of the AI Engine graphs, kernels-mapped to processors, with tables and graphic presentation of metric data.

The Profile Summary provides annotated details regarding the overall application performance. All data generated during the execution of the application is grouped into categories. The Profile Summary lets you examine processor/DMA memory stalls, deadlock, interference, critical paths, and maximum contention. This is useful for system-level performance tuning and debug. System performance is presented in terms of latency (number of cycles taken to execute the system) and throughput (data/time taken). Sub-optimal system performance forces you to examine and control (thru constraints) mapping and buffer packing, stream and packet switch allocation, interaction with neighboring processors, and external interfaces. An example of the raw Profile Summary report is shown.

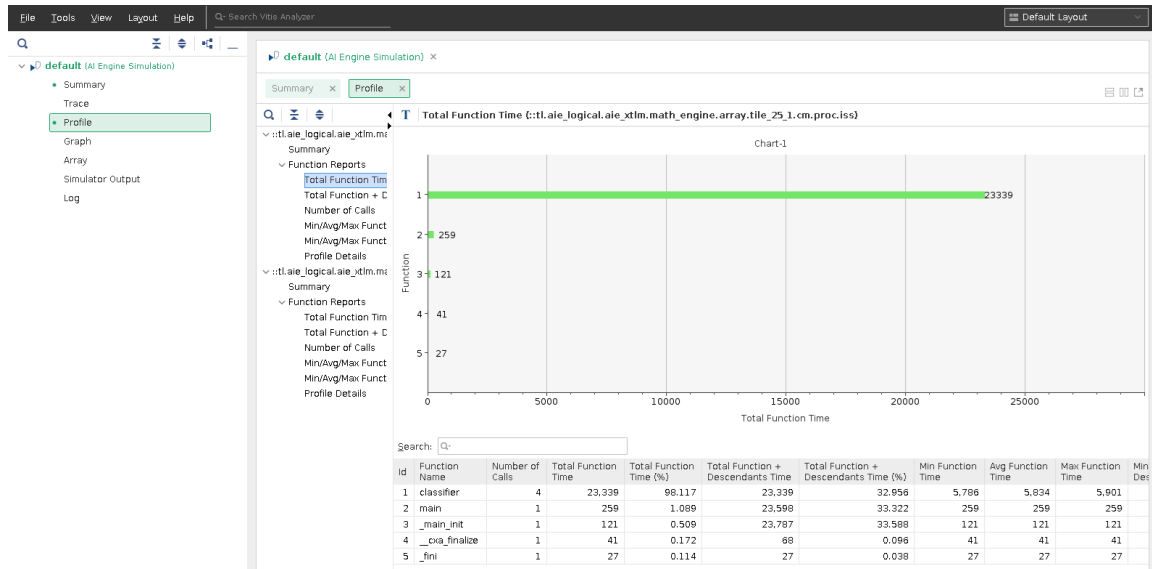
Figure 20: Profile Summary



Note: The row value of tile number from profile report is one above actual tile number. For example tile_38_1 is tile(38, 0) from the previous screen shot.

Specific tables can be used to see profile information specific to the kernels. This is shown as a chart with a table showing what is running on the tiles. The following is an example chart.

Figure 21: Example Chart

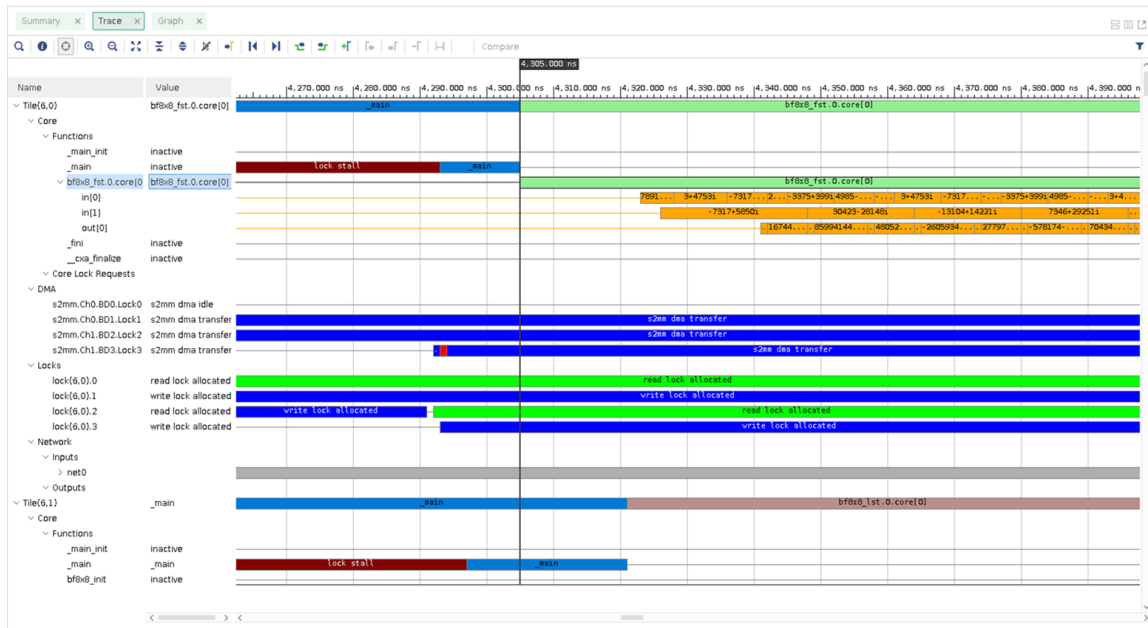


In this view, you can see a chart that shows a Total Function Time which is the total cycles the function used in running the graph. The y-axis shows the id of the function that can be referenced in the following table. This information can be useful in determining where time is being spent in a function and helps with potential optimization or debug.

Trace Report

Issues such as missing or mismatching locks, buffer overruns, and incorrect programming of DMA buffers are difficult to debug using traditional interactive debug techniques. Event trace provides a systematic way of collecting system level traces for the program events, providing direct support for generation, collection, and streaming of hardware events as a trace. The following image shows the Trace report open in the Vitis analyzer.

Figure 22: Trace Report



Note: This example illustrates kernel function and functions that are added by the compiler:

- **_main:** Core `main` function. This is different from the function used in the top-level file.
- **_main_init:** Kernel `init` function that runs once per graph execution.
- **_cxa_finalize:** Calls destructors of global C++ objects.
- **_fini:** This section holds executable instructions that terminate the process. When a program exits normally, the system runs the code in this section.

Note: If the VCD file is too large and it takes too much time for the Vitis analyzer to analyze the VCD and open the Trace view, you can do an online analysis of the VCD when running the AI Engine simulator. The Vitis analyzer then opens the existing WDB and CTF files instead of analyzing the VCD file. The command for AI Engine simulator is as follows.

```
aiesimulator --pkg-dir=./Work --online -wdb -ctf
```

Features of the trace report include the following.

- Each tile is reported. Within each tile the report includes core, DMA, locks, and I/O if there are PL blocks in the graph.
- There is a separate timeline for each kernel mapped to a core. It shows when the kernel is executing (blue) or stalled (red) due to memory conflicts or waiting for stream data.
- By using lock IDs in the core, DMA, and locks sections you can identify how cores and DMAs interact with one another by acquiring and releasing locks.

- The lock section shows the activities of the locks in the tile, both the allocation and release for read and write lock requests. A particular lock can be allocated by nearby tiles. Thus, this section does not necessarily match the core lock requests of the core shown in the left pane of the image.
- If a lock is not released, a red bar extends through the end of simulation time.
- Clicking the left or right arrows takes you to the start and end of a state, respectively.
- The data view shows the data flowing through stream switch network with slave entry points and master exit points at each hop. This is most useful in finding the routing delays, as well as network congestion effects with packet switching, where one packet might get delayed behind another packet when sharing the same stream channel.

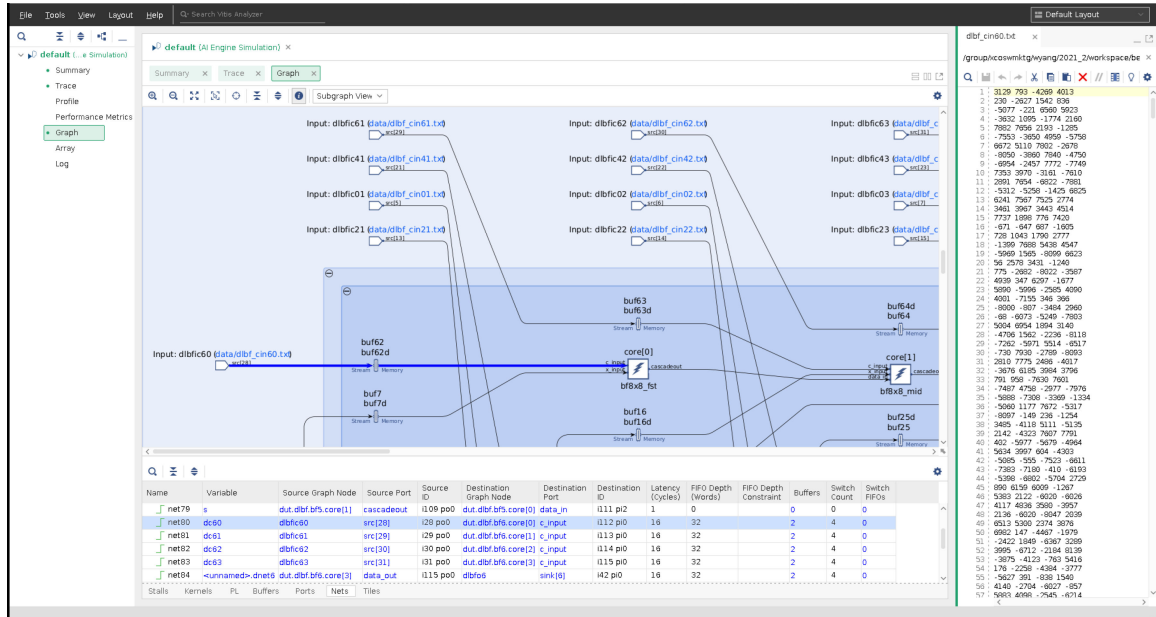
Trace View Data Visualization

Trace view data visualization allows you a side-by-side view of events at the I/O ports, allowing you to look into the design and examine the relative event timing. This provides insight into how the design works in a multi-core environment.

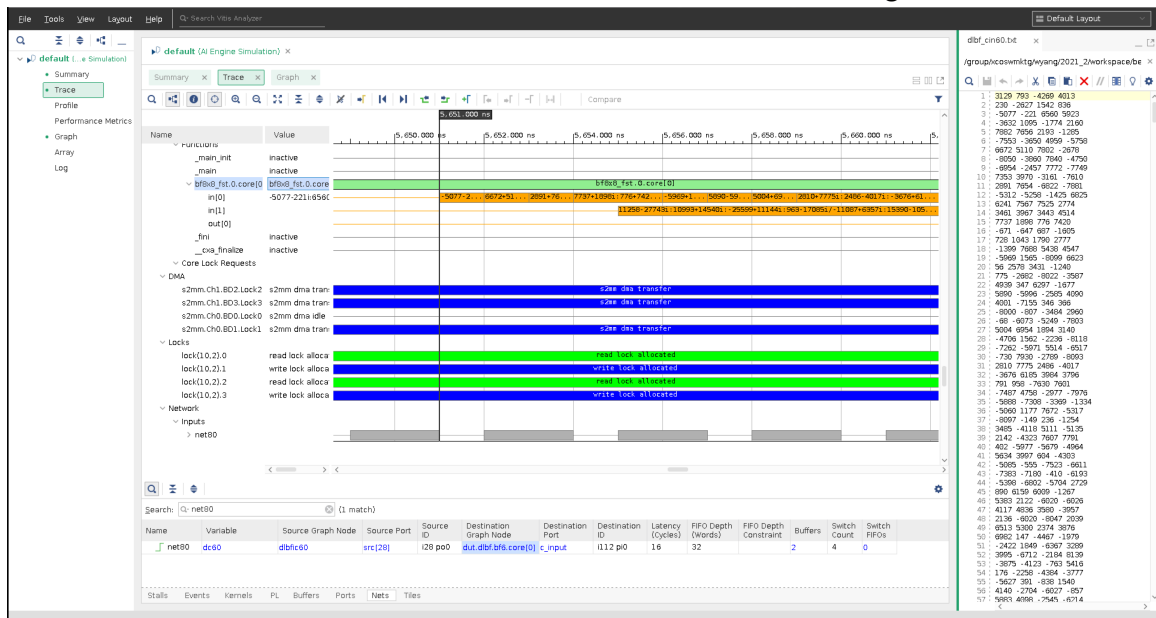
Window Data Analysis

Window data analysis in the trace view allows you to cross probe from a specific input/output window net connections in the graph view to the appropriate position in the event trace view. You can trace window input data as it is being buffered prior to the kernel starting execution. To perform data analysis on a window interface, use the following steps.

1. The following image shows a net connection that has been selected in the Graph view. The selected net (`net80`) is highlighted both in graph view and in the nets table at the bottom of the window. In addition the Graph view also associates the simulation input and output files with the appropriate nets. The simulation input file associated with `net80` is displayed on the right.



2. In the graph view, select the file source that connects to `net 80`, as shown in the previous image. The input data is displayed at the right hand side of the window.
3. Switch to the Trace view to view the events and detailed event timing.



4. In the events table at the bottom of the screen, select **Net** from the column selector.
5. Type `net 81` in the adjacent (filter) box.
6. Net events that match `net 80` are highlighted in the events table. Browse through the events if there are multiple events using the **Previous/Next** toolbar buttons to show the matched events.
7. The highlighted data matches the data from the input file, shown in the right-hand pane.

Note: The AI Engine kernel does not start until all the data is available at the window interface.

Using these steps, you can inspect I/O data for the required window connections to ensure the correctness of the I/O data to and from a kernel.

Supported Window Data Types

Table 49: Supported Window Data Types

Data Types	Display Format Example
int8 ¹	±123
int16	±12345
int32	±1234567890
int64	±1234567890123456789
uint8	123
uint16	12345
uint32	1234567890
uint64	12345678901234567890
cint16	±12345±12345i
cint32	±1234567890±1234567890i
float	±1.234567
cfloat	±1.234567±1.234567i
v4cint16	±1+2i:3+4i:5+6i:7+8i
v4int32	±1:2:3:4
v4cint32	±1+2i:3+4i
v4int64 ⁴	±1:2/3:4
v4float	±1.000000:2.000000:3.000000:4.000000
v4cfloat	±1.000000+2.000000i:3.000000+4.000000i
v8int16	±1:2:3:4:5:6:7:8
v8cint16	±1+2i:3+4i:5+6i:7+8i
v8int32	±1:2:3:4:5:6:7:8
v8float	±1.000000:2.000000:3.000000:4.000000:5.000000:6.000000:7.000000:8.000000
v16int8	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16
v16uint8	1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16
v16int16 ²	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16
v16uint16 ²	1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16
v16int32	±1:2:3:4:5:6:7:8
v16float ²	±1.000000:2.000000:3.000000:4.000000:5.000000:6.000000:7.000000:8.000000:9.000000:10.000000:11.000000:12.000000:13.000000:14.000000:15.000000:16.000000
v16cfloat ²	±1.000000+2.000000i:3.000000+4.000000i:5.000000+6.000000i:7.000000+8.000000i:9.000000+10.000000i:11.000000+12.000000i:13.000000+14.000000i:15.000000+16.000000i
v32int8 ³	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32
v32uint8 ³	1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32

Table 49: Supported Window Data Types (cont'd)

Data Types	Display Format Example
v32int16	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16
v32cint16	±1+2i:3+4i:5+6i:7+8i:9+10i:11+12i:13+14i:15+16i
v32int32	±1:2:3:4:5:6:7:8
v32float	±1.000000:2.000000:3.000000:4.000000:5.000000:6.000000:7.000000:8.000000
v64int8	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32 ±33:34:35:36:37:38:39:40:41:42:43:44:45:46:47:48:49:50:51:52:53:54:55:56:57:58:59:60:61:62:63:64
v64uint8	1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32 33:34:35:36:37:38:39:40:41:42:43:44:45:46:47:48:49:50:51:52:53:54:55:56:57:58:59:60:61:62:63:64
v64int16	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16 ±17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32

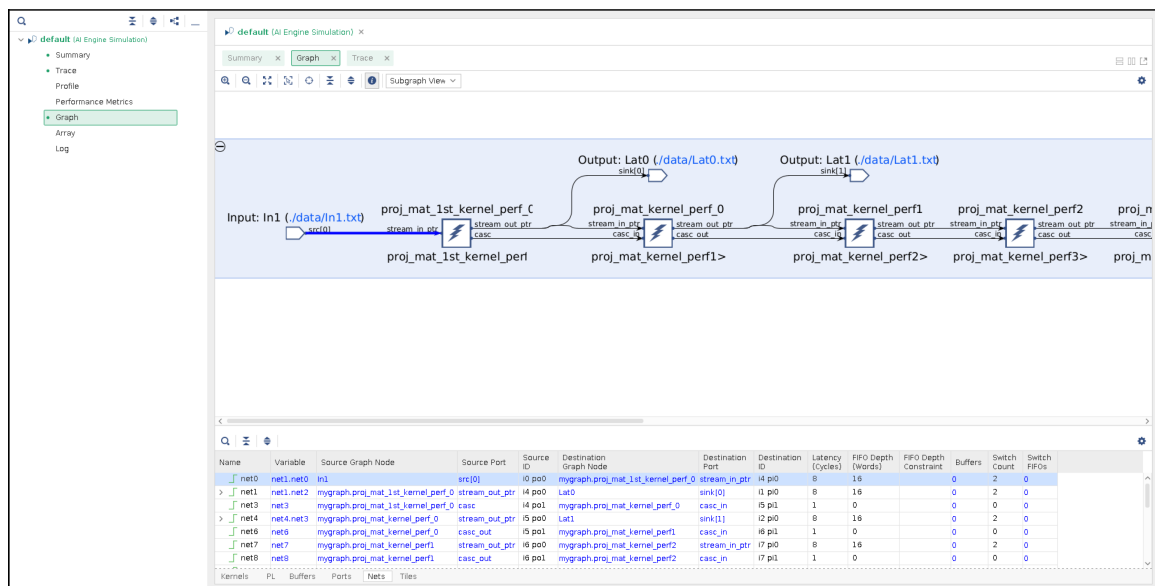
Notes:

1. Preceding 0 and '+' not displayed.
2. Currently data misaligned or out of order in timeline view and event table.
3. In the case of 256-bit data, each 128-bit data is separated by ' / ' instead of ' : '. For example, 1:2:3:4 / 5:6:7:8.
4. Two samples at timestamp x+1, next two samples at timestamp x.

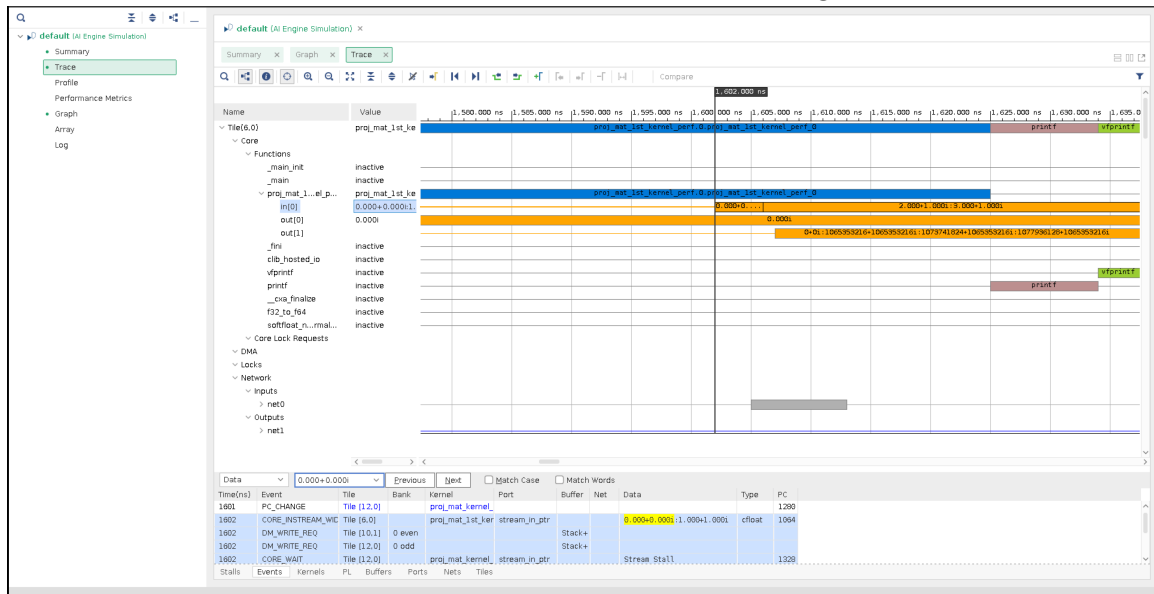
Stream Data Analysis

Stream data analysis in the trace view allows you to cross probe from a specific input/output stream net connection in the graph view to the appropriate position in the event trace view. You can trace stream input data as it is being received during the kernel execution. To perform data analysis on a stream interface, use the following steps.

1. The following image shows a kernel port that has been selected in the Graph view. The selected net with ID, `net0`, is highlighted both in the Graph view and in the net table at bottom of the window.



2. Switch to the Trace to view the events and detailed event timing.



3. Move the time marker to the beginning of the stream data by dragging the vertical line (time marker) in the upper part of the window.
4. In the upper part of the window you can examine the data associated with the selected input port.
5. In the events table at the bottom of the screen, select **Data** from the drop-down menu.
6. Type $0.000+0.000i$ into the adjacent (filter) box and click **Next**. The expected time and data are highlighted in the events table.

Note: The input value depends on data type. $0.000+0.000i$ in this example is for a complex float type.

Using these steps, you can inspect I/O data for the required stream connections to ensure the correctness of the I/O data to and from a kernel.

Supported Stream Data Types

Table 50: Supported Stream Data Types

Data Types	Display Format
int8 ¹	±123
int16	±12345
int32	±1234567890
int64 ³	±1234567890123456789
uint8	123
uint16	12345
uint32	1234567890
uint64 ³	12345678901234567890

Table 50: Supported Stream Data Types (cont'd)

Data Types	Display Format
cint16	$\pm 12345 \pm 12345i$
cint32 ²	$\pm 1234567890 \pm 1234567890$
float	± 1.234567
cfloat ²	$\pm 1.234567 \pm 1.234567i$
acc48	$\pm 1:2:3:4:5:6:7:8$
cacc48	$\pm 1+1i_2+2i_3+3i_4+4i$
acc80 ⁴	N/A
cacc80 ⁴	N/A
accfloat	$\pm 1.000000:2.000000:3.000000:4.000000:5.000000:6.000000:7.000000:8.000000$
caccfloat	$1.000000+2.000000i:3.000000+4.000000i:5.000000+6.000000i:7.000000+8.000000i$
v2cint32	$\pm 1+2i:3+4i$
v4cint16	$\pm 1+2i:3+4i:5+6i:7+8i$
v4int32	$\pm 1:2:3:4$
v4float	$\pm 1.000000:2.000000:3.000000:4.000000$
v8int16	$\pm 1:2:3:4:5:6:7:8$
v16int8	$\pm 1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16$
v16uint8	$1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16$
v8acc48	$\pm 1:2:3:4:5:6:7:8$
v4cacc48	$\pm 1+2i_3+4i_5+6i_7+8i$

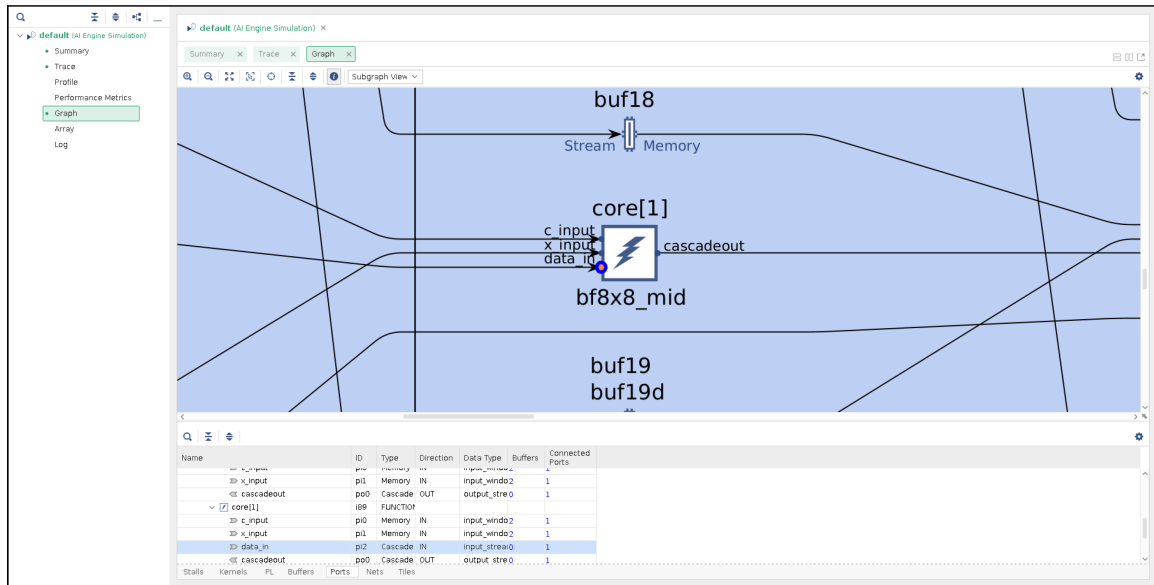
Notes:

1. Preceding 0 and '+' not displayed.
2. Real part followed by imaginary part at x and x+1 timestamp.
3. Least significant 32 bits value and Most significant 32 bits value are displayed at x and x+1 timestamp.
4. Currently under development.

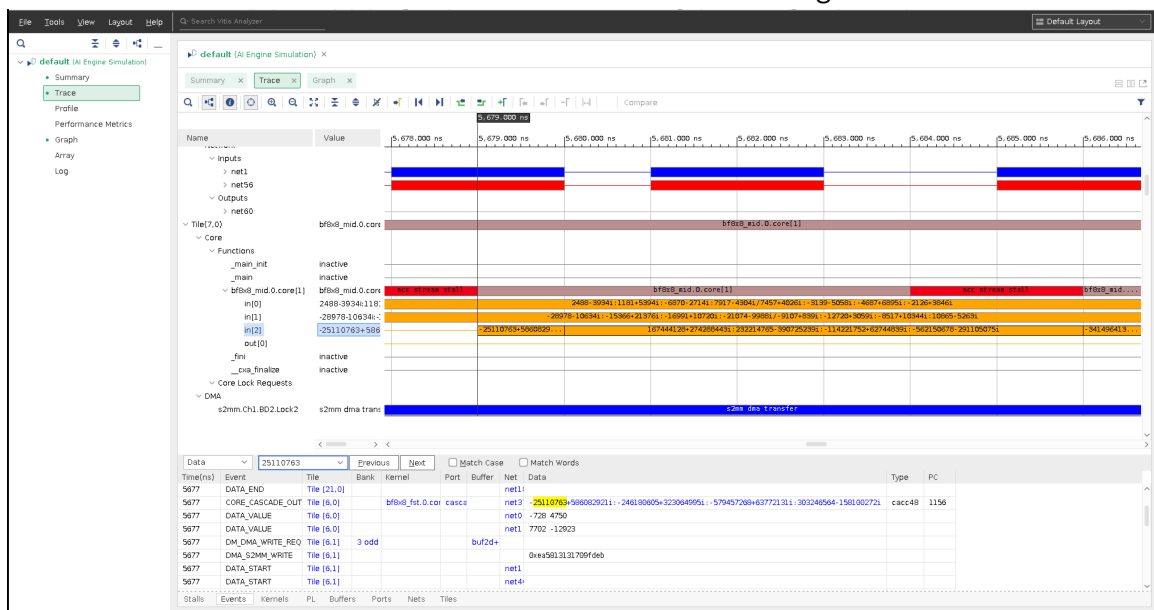
Cascade Data Analysis

Tracing cascaded data is similar to trace stream data. Cascade data analysis in the trace view allows you to cross probe from a specific input/output cascade connection in the Graph view to the appropriate position in the event trace view. You can trace cascade input data as it is being received during the kernel execution. To perform data analysis on a cascade interface, use the following steps.

1. The following image shows a kernel port that has been selected in the Graph view. The selected port with ID, `pi2`, is highlighted both in the Graph view and in the ports table at bottom of the window.



2. Switch to the Trace to view the events and detailed event timing.



- View the data associated with the selected input port.
- In the events table at the bottom of the screen, select **Data** from the column selector drop-down menu.
- Type 25110763 into the adjacent (filter) box and click **Next**. The expected time and data are shown in the events table.

Note: The input value depends on data type. -25110763+586082921i in this example is for a complex integer data type.

Using these steps, you can inspect I/O data for the required cascade connections to ensure the correctness of the I/O data to and from a kernel.

Data Display Limitations

The limitations of the trace view data visualization feature are as follows.

- 64-bit non-vector window data types are displayed as two 32-bit high and low values.

For example, an unsigned, 64-bit integer `0x00000000100000002` is displayed as 2L and 1H separately. L and H represent low and high 32 bits.

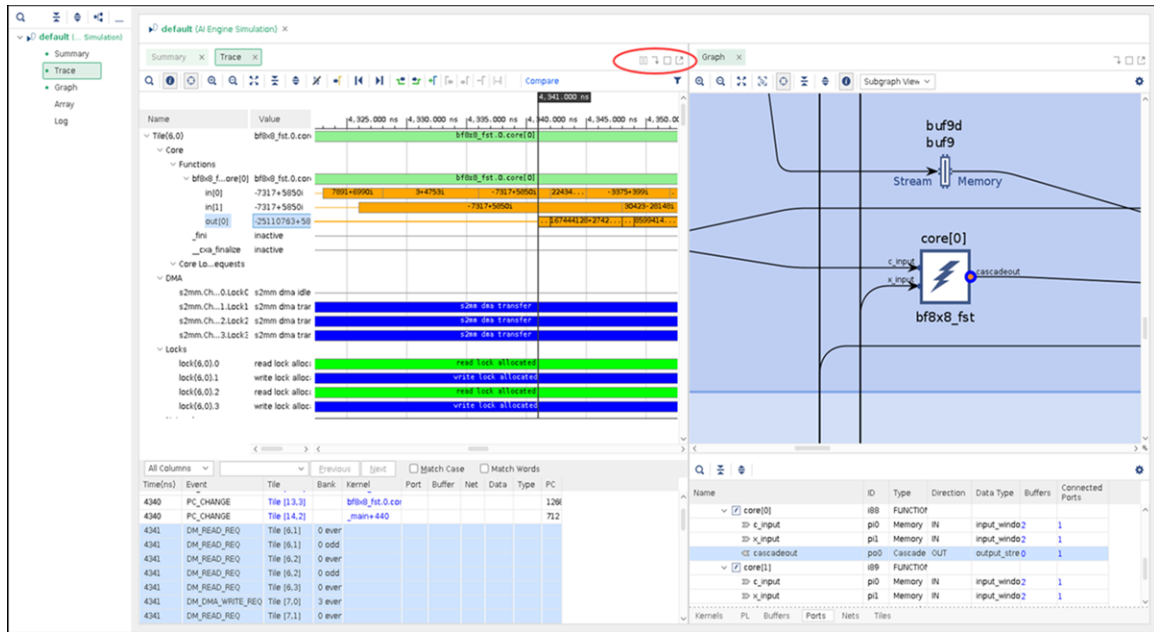
Another example is a complex 32-bit value where the real 32 bits are displayed first, followed by the imaginary 32 bits.

- The `input_pktstream/output_pktstream` data type is displayed in integer format.
- Kernel names and instance names can be displayed using the Vitis analyzer for templated class. However, templated class name is not supported.
- Kernel functions are not visible in the trace if they are inlined by the AI Engine compiler.
- Helper functions called inside the kernel function are not visible in the trace.

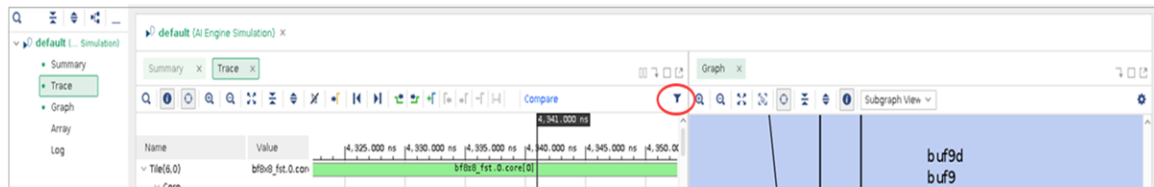
Cross-Probing

Debugging a design is a complicated process and often needs to switch between views/reports. The Vitis analyzer supports this functionality and allows detailed inspection of events and their associated timing. Cross-probing allows you to probe data in different views allowing you to view I/O data from different perspectives, including data on a particular tile or port, and at what time an event occurred, all within the same window. The Trace and Graph views can be shown in the same window and moving the time marker in the Trace view or selecting object(s) in the Graph view applies to both views simultaneously.

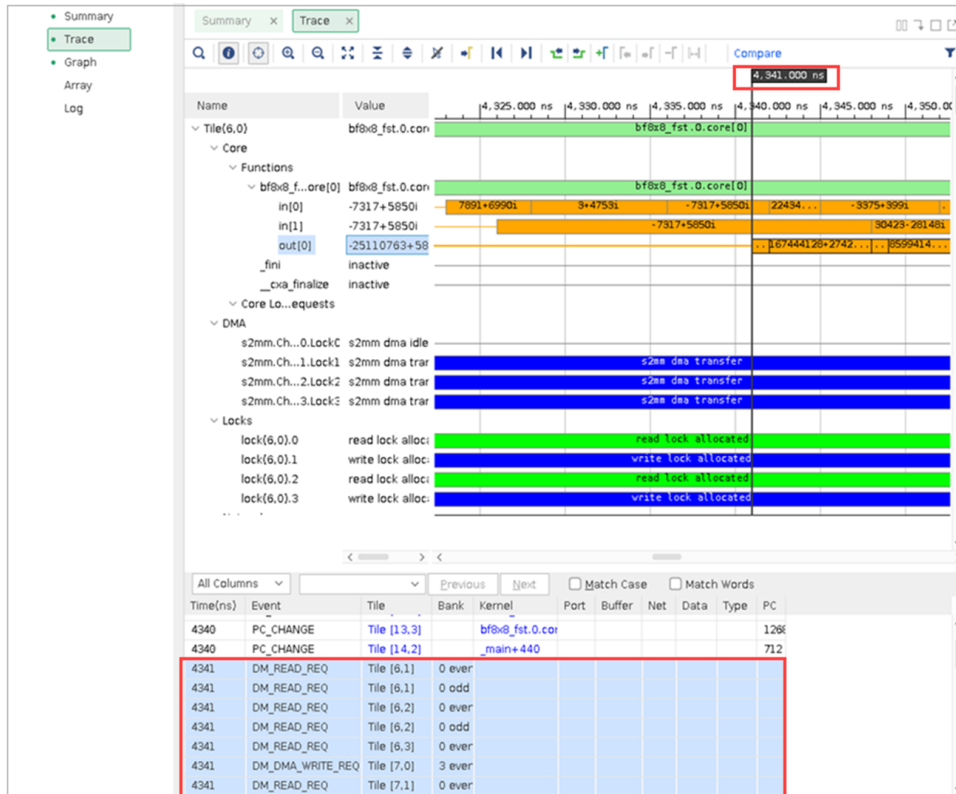
1. Use the Window Layout toolbar (circled in the following image) to manage views/reports. In the following image, the Trace and Graph view are selected to appear in the same window.



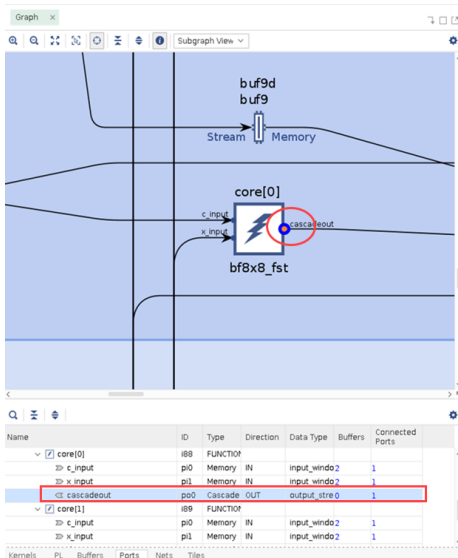
2. The filter button function, shown circled in the following image, selects tiles, functions, input/output ports, DMA, locks to be included or excluded from trace view to focus on areas of interest.



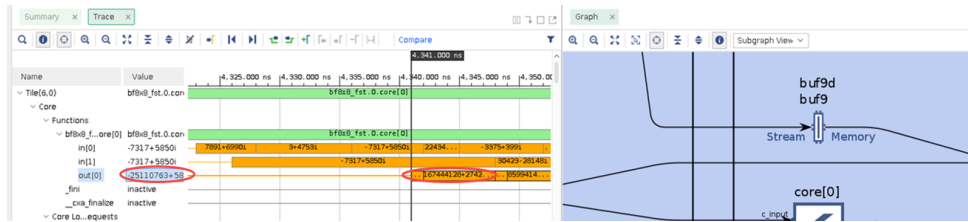
3. Drag the time marker to move the time backwards and forwards to evaluate events at a given time. Events occurring after the selected time are highlighted in the events table of the Trace view at the bottom of the screen.



4. Select objects in the Graph view to map graph, tiles, I/O ports, and net connections to see the object ID, type, direction, data type, buffers, and connected ports.



5. Values of I/O ports are available in the Trace view, shown circled in the following image. This example design uses complex 16 bits value type (c_int16).

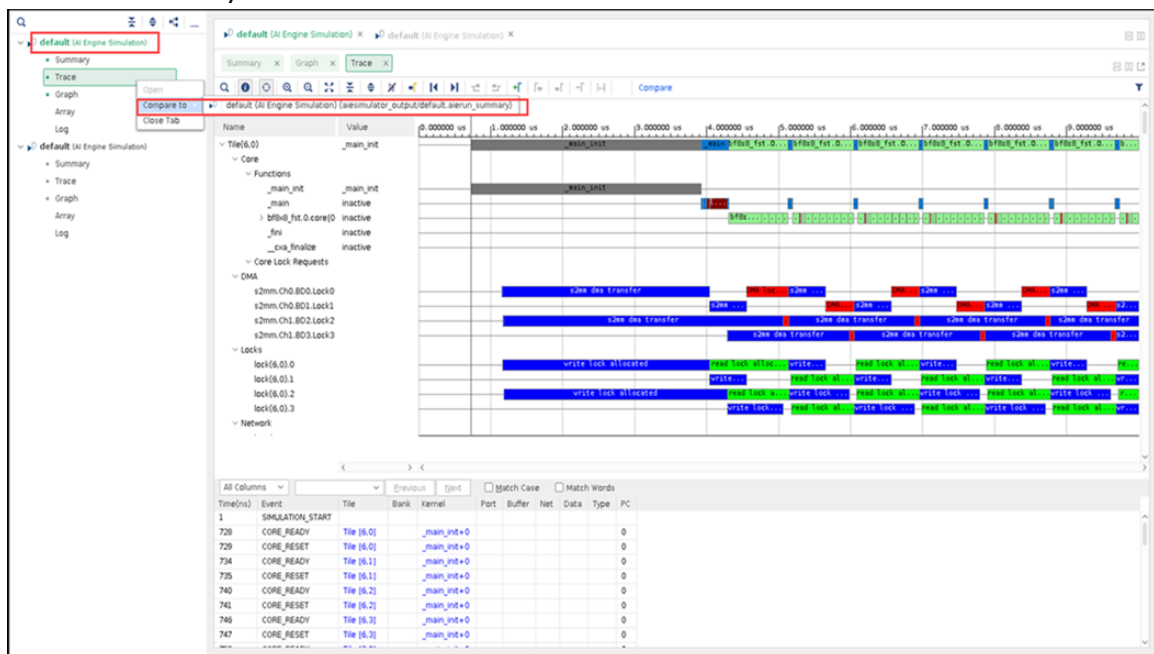


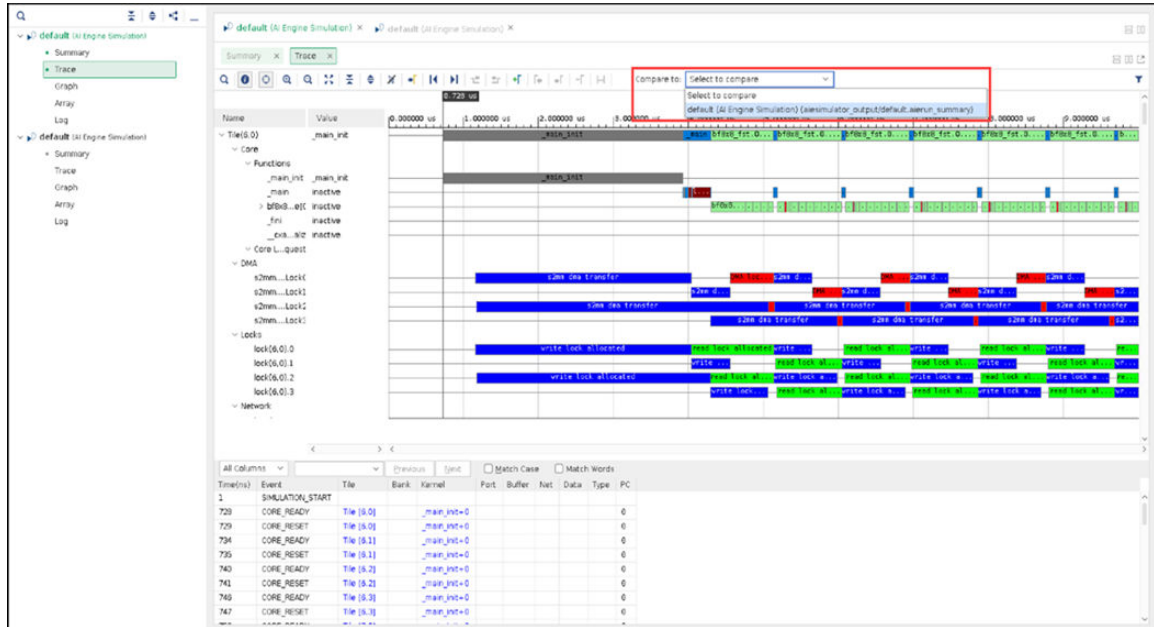
These steps show an output port object in the Graph view (on the right side) at 4,341.000 ns into the design execution and in the Trace view (on the left side) where the output data is available. Moving the time marker to a later time in the Trace view, the events in the events table (lower portion of the Trace view) are highlighted to indicate which events happened at that time. This information is useful to relate events in a multi-processor environment. Other examples can leverage this cross-probing feature in the Vitis analyzer tool.

Trace Compare

Trace Compare in the Vitis analyzer allows the comparison of two different design executions. The comparison between two design runs helps to see the impact of one or more variances introduced in a design run. This allows you to inspect performance difference regardless of what variables are adjusted.

1. Open the two summary files to be compared.
2. Select the Trace view for either design, right-click and select **Compare to** → **<filename>_summary** for the other the design, or click on the **Compare** link. The compare can start from any trace.





Note: In Trace Compare mode, features in the Vitis analyzer apply to both Trace views.

This example uses the same design but one has added `printf()` statements in the kernel. By looking at this Trace Compare example, you can see that the upper design expended time on the `printf()` calls that slowed down the overall execution of the design.

AI Engine Stall Analysis in Vitis Analyzer

AI Engine execution can be stalled by stall logic from multiple sources include the following.

- **External memory-mapped AXI4 master:** Any external AXI4 master (for example, PS) can issue a stall signal to a specific AI Engine.
- **Lock modules:** AI Engine has access to locks for hardware synchronization. When acquiring a lock, if the core does not get a lock, then the AI Engine will be stalled until the lock becomes unlocked.
- **Empty or full AXI4-Stream interfaces:** The AI Engine can be stalled when reading from an empty input FIFO or writing to a full output FIFO.
- **Data memory collisions:** Memory stall can occur between two different AI Engines or when one AI Engine tries to access a single memory bank (for example, trying to load and store in the same cycle to the same memory bank).
- **Event actions from the event unit:** The AI Engine can be stalled by event actions from the event unit.

When an AI Engine stalls, all memory interfaces to AI Engine including program memory interface are stalled. The stall is resolved when the cause of the stall has been fixed.

Vitis analyzer can use the VCD trace from AI Engine simulation to do stall analysis that shows an overview of the stall status in metrics. It also helps you detect where the stall happens and the possible causes.

For Vitis analyzer to do stall analysis, it is advised to run AI Engine simulator with `--online -wdb -ctf` options to generate event trace information in the background.

```
aiesimulator --pkg-dir=./Work --online -wdb -ctf
```

Note: If the VCD file is needed for analysis, `--dump-vcd` option of AI Engine simulator can be used instead. However, note that Vitis analyzer takes time to generate event trace from the VCD file. Especially when the design is large, the time becomes non-negligible. Therefore, it is recommended to use `vcdanalyze` together with AI Engine simulator to prepare event trace for Vitis analyzer to do AI Engine stall analysis.

```
aiesimulator --pkg-dir=./Work --dump-vcd foo
vcdanalyze --vcd=foo.vcd --wdb --ctf
```

Note: Do not use `--outdir` option of `vcdanalyze` to place output data in directories other than default.

For steps to launch the Vitis analyzer to view the AI Engine simulation result, see [Viewing the Run Summary in the Vitis Analyzer](#).

```
vitis_analyzer ./aiesimulator_output/default.aierun_summary
```

The following additional settings are required for AI Engine stall analysis in Vitis analyzer working in the HW emulation flow.

1. Write a simulator option file (for example, `sim_options.txt`) with content:

```
AIE_DUMP_VCD=foo
```

2. Launch HW emulation with options:

```
./launch_hw_emu.sh -aie-sim-options ./sim_options.txt -add-env
AIE_COMPILER_WORKDIR=<ABSOLUTE_PRJ_PATH>/Work
```

3. Optionally run `vcdanalyze`:

```
cd ./sim/behav_waveform/xsim/; vcdanalyze --pkg-dir=../../Work --
vcd=foo.vcd --wdb -ctf
```

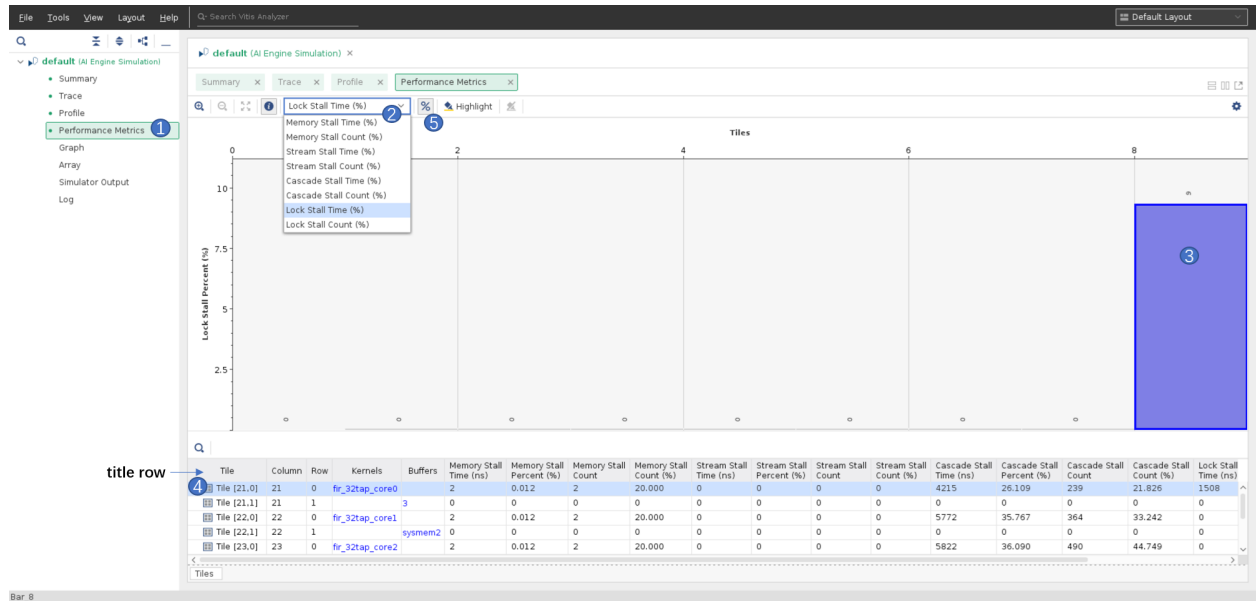
4. Launch Vitis analyzer:

```
vitis_analyzer ./sim/behav_waveform/xsim/default.aierun_summary
```


Performance Metrics

The Performance Metrics view shows the percentage of each type of stall against total simulation time or total stall count.

Figure 23: Performance Metrics View



1. Click on the Performance Metrics view.
2. Choose the stall type from the drop-down list. The metrics separate different types of stalls. It will show the existing stalls in the simulation result.
 - **Lock Stall Time (%):** Percentage of lock stall time during which buffers in the AI Engine tile are being acquired.
 - **Lock Stall Count (%):** Percentage of lock stall count in a specific tile against total lock stall count in all AI Engine tiles.
 - **Memory Stall Time (%):** Percentage of memory stall time that memory accesses in the AI Engine tile have conflicts.
 - **Memory Stall Count (%):** Percentage of memory stall count in a specific tile against total memory stall count in all AI Engine tiles.
 - **Stream Stall Time (%):** Percentage of stream stall time due to full or empty stream.
 - **Stream Stall Count (%):** Percentage of stream stall count in a specific tile against total stream stall count in all AI Engine tiles.
 - **Cascade Stall Time (%):** Percentage of cascade stream stall time that either due cascade stream is full or empty.
 - **Cascade Stall Count (%):** Percentage of cascade stream stall count in specific tile against total cascade stream stall count in all AI Engine tiles.

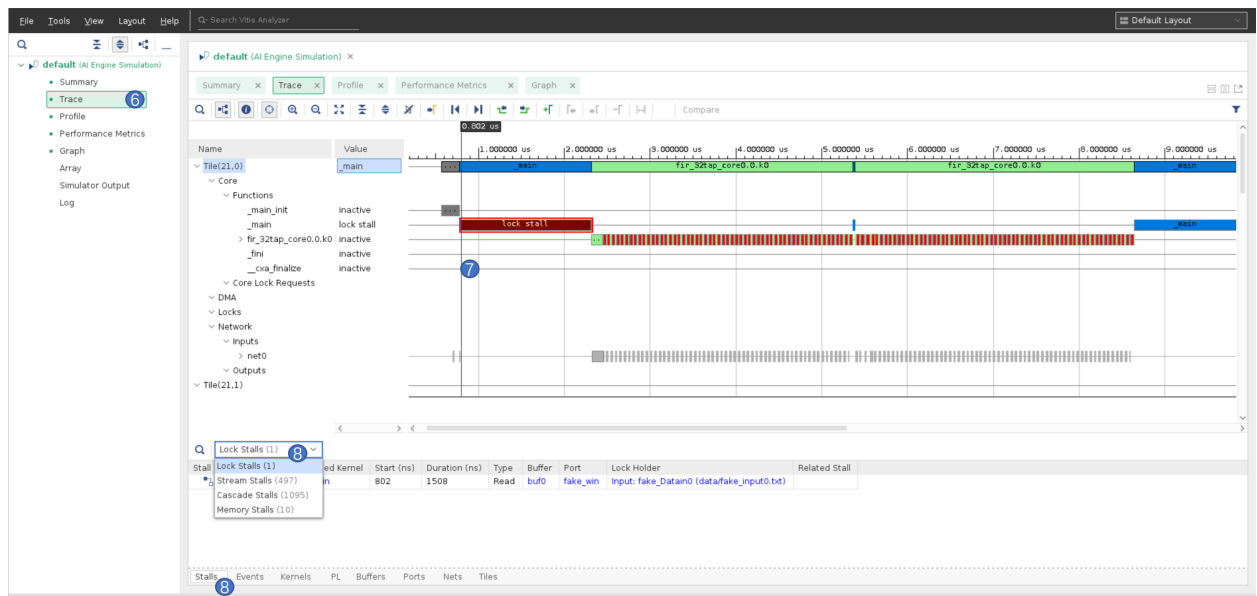
- Each AI Engine tile is shown as a bar in the Performance Metrics view. The higher the stall percentage, the more towards the right the bar will be. More attention should be paid to the highest bars. Click one of the bars to select the tile to focus.
- The Tiles view at the bottom lists all the AI Engine tiles with the information about column, row, kernels, buffers, and all the stall times, percentage, as well as count. You can click on the title row to sort on a specific column.



TIP: When the number or information in the view is in blue, it can be cross-probed with other views.

- There is a % button beside the drop-down list. By clicking on this button, you can switch between displaying the stall time in percentage or nano seconds (ns). While exploring the information in the Performance Metrics view, it is usually helpful to cross-probe with other views. For example:

Figure 24: Trace View



- Choose the tile that has the largest stall time, then go to Trace view to see the position and frequency of the stalls.
- Zoom in and out of the Trace view to have a better position of the stall in the timeline.
- Click on the Stalls view and from the drop-down list, select the type of stalls to examine. Select a stall in the Stalls view to highlight it in the Trace view.

The Performance Metrics view, Trace view, Graph view, and Array view can be cross-probed between each other. Graph view helps in understanding where the stall happens in the graph, and Array view helps in viewing the positions of the objects in hardware. Additional details about analyzing each type of stall are explained in the following sections.

Lock Stall Analysis

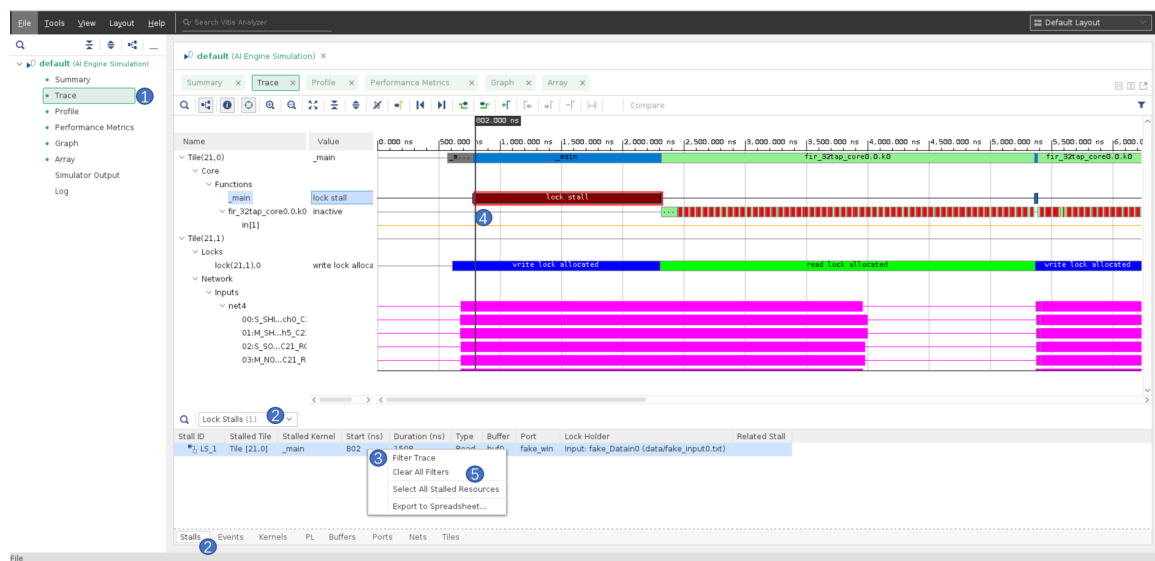
The Performance Metrics analysis tab can help identify if the lock stall needs to be analyzed. The following steps illustrate how a lock stall can be analyzed starting in the Vitis analyzer.

1. Select the Trace view.
2. Select the Stalls view and select **Lock Stalls** from the drop-down list.



TIP: The Stalls view is available with the Trace view, Graph view, and Array view.

Figure 25: Lock Stall in Trace View



Each stall has the following information associated with it.

- **Stall ID:** The lock stall is named LS_<NUM>. The number is unique across all types of stalls. The earlier the stall happens, the smaller the number.
- **Stalled Tile:** The AI Engine tile where the stalled kernel is located.
- **Stalled Kernel:** The kernel that is stalled. It is named <Kernel_function_name>.<Schedule_ID>.<Graph_instance_name>. Sometimes it is shown as _main and then cross-probe is required to find the real kernel function.
- **Start (ns):** The start time of the stall.
- **Duration (ns):** The duration of the stall.
- **Type:** The stalled kernel tries to read or write the buffer.
- **Buffer:** The buffer that the stalled kernel tries to read or write.
- **Port:** The port of the stalled kernel that tries to read or write the buffer.

- **Lock Holder:** The source that is holding the lock of the buffer.
- **Related Stall:** Other stalls that can cause a stall.



TIP: The items in blue can be cross-probed with other views.

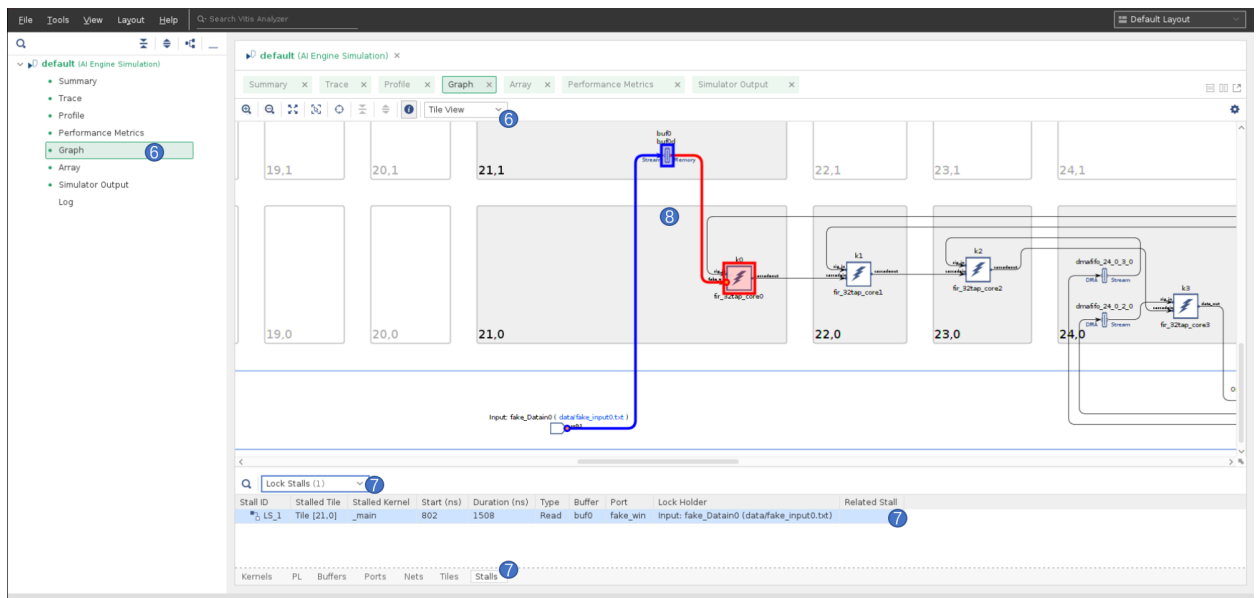
3. Select one row of the stall. It will go to the start of the stall in Trace view. It is optional to filter all the signals that are related to the stall by right-clicking the stall and choosing **Filter Trace**. In Trace view, the signals related to the stall are shown. Non-related signals are hidden. Exploring the trace is easier when the design is large.



TIP: Filter Trace may not show signals related to the related stalls.

4. Trace view can be used to view the lock stall in timeline. For a specific lock stall, it can be seen when the write lock and read lock are allocated. From the position of the stall and events before and after the stall, the reason for the stall can be analyzed. For example, if write lock has already been allocated and the lock type is *Read*, then it indicates that the buffer has not been released by the producer. The consumer is waiting for it to be readable. The producer can be found in the Lock Holder.
5. To clear the previously filtered trace, right-click and choose **Clear All Filters**.
6. It is usually helpful to have an overview of the stall path in Graph view. Select Graph view and then select **Tile View** from the drop-down list in the view. Tile view of Graph view shows graph in AI Engine tiles.

Figure 26: Lock Stall in Graph View



7. If the Stalls view is not shown, select **Lock Stalls** from the drop-down list and choose the stall to be analyzed. It will highlight the related paths in the Tile view of the Graph view. The red path shows where the stall occurs. The blue path shows the source to where the stall occurs.

The following table lists scenarios that can cause a lock stall and possible solutions.

Table 51: Lock Stall Scenarios and Solutions

Source	Target	Destination	Stall Type	Possible Solution
AI Engine kernel	Lock of sync window	AI Engine kernel	Lock stall	<ul style="list-style-type: none"> If single buffer is used, use PING-PONG buffer (default) or place kernels into same AI Engine tile. If the kernels are unbalanced in execution time, balance throughput between kernels.
AI Engine kernel	Lock of async window (<code>window_acquire</code> and <code>window_release</code> API)	AI Engine kernel	Lock stall	<ul style="list-style-type: none"> If a single buffer is used, use a PING-PONG buffer (default). Acquire and release buffer in-time. Use a local buffer as needed.
PL interface	Lock of window	AI Engine kernel	Lock stall	<ul style="list-style-type: none"> Ensure that the PL interface throughput matches the AI Engine throughput. Check that the PL interface frequency and width are set properly. See Chapter 7: AI Engine/Programmable Logic Integration.
AI Engine	Lock of window	PL interface	Lock stall	<ul style="list-style-type: none"> Ensure that the PL interface throughput matches AI Engine throughput. Check that the PL interface frequency and width are set properly. See Chapter 7: AI Engine/Programmable Logic Integration.

Note: DMA lock stall is not included in the Vitis analyzer lock stall analysis.

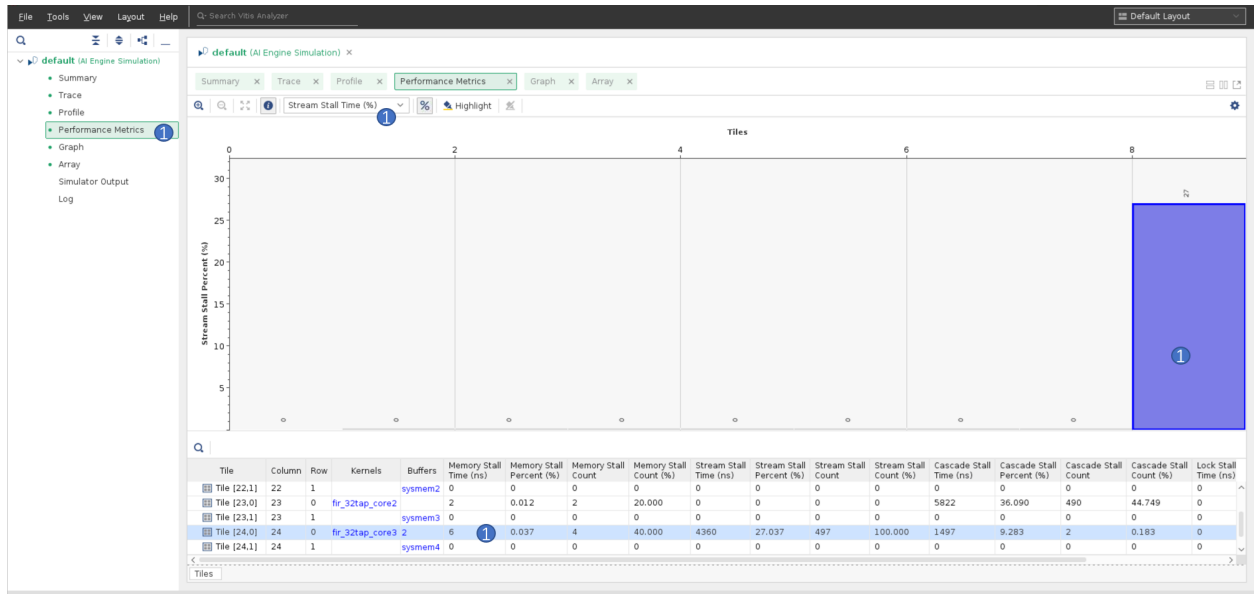
Stream Stall Analysis

From the Performance Metrics view, you can identify if a stream stall needs to be analyzed, and also the tile/tiles that are causing the stall.

The following steps illustrate how a stream stall can be analyzed starting in the Performance Metrics tab in Vitis analyzer.

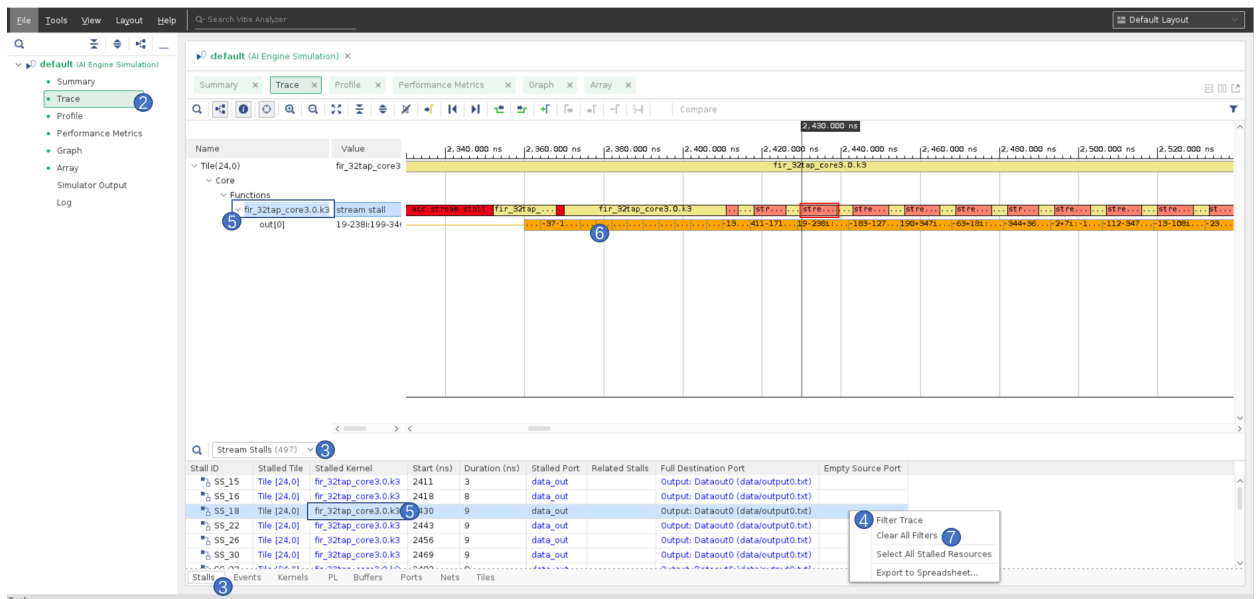
1. In the Performance Metrics view, select **Stream Stall Time (%)** to view stream stalls across all tiles. Identify the tile(s) to be analyzed. Note that the objects in Performance Metrics view can be cross-probed with Trace view, Graph view, and Array view. For example, selecting the tile in the Performance Metrics view to highlight the tile in Trace view can help quickly locate the tile.

Figure 27: Stream Stall in Performance Metrics View



2. Choose Trace view.

Figure 28: Stream Stall in Trace View

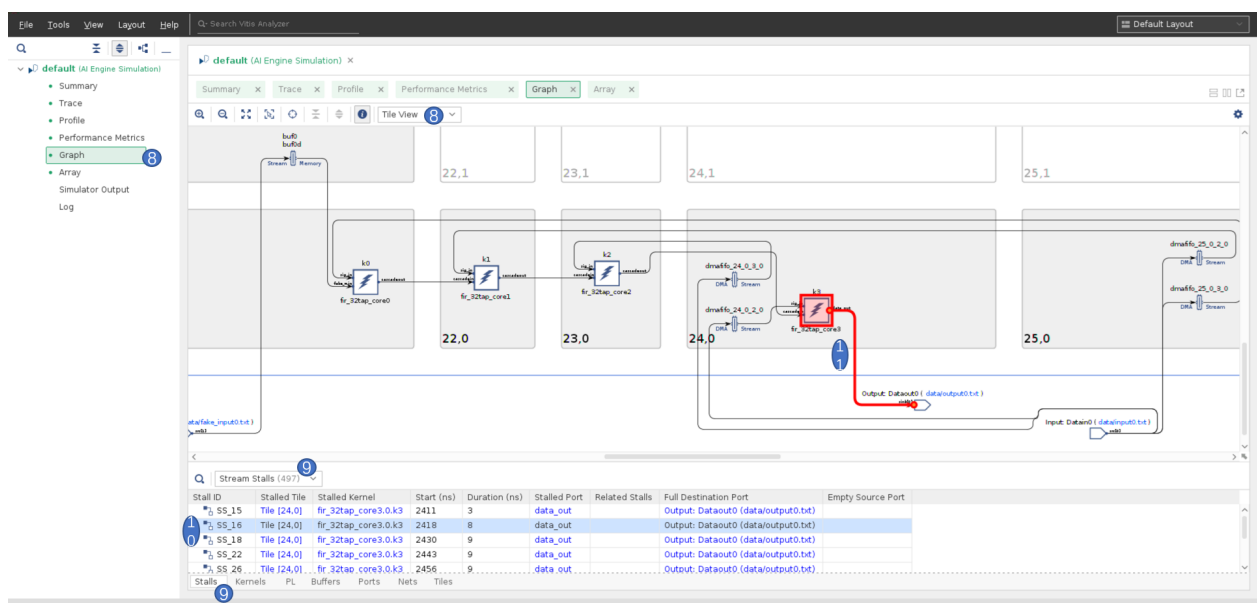


3. From the Stalls view, select **Stream Stalls** from the drop-down list. In the Stalls view, stream stalls have the following information and the objects in blue can be cross-probed with other views by clicking on it.

- **Stall ID:** The stream stall is named SS_<NUM>. The earlier the stall happens, the smaller the number. The number is unique across all types of stalls.
- **Stalled Tile:** The AI Engine tile where the stalled kernel is located.

- **Stalled Kernel:** The kernel that is stalled. It is named `<Kernel_function_name>.<Schedule_ID>.<Graph_instance_name>`. Sometimes it is shown as `_main` and then cross-probe is required to find the real kernel function.
 - **Start (ns):** The start time of the stall.
 - **Duration (ns):** The duration of the stall.
 - **Stalled Port:** The port of the stalled kernel.
 - **Related Stalls:** Other stalls that can cause the stall.
 - **Full Destination Port:** The port that the stalled kernel cannot write into because it is full.
 - **Empty Source Port:** The port that the stalled kernel cannot read from because it is empty.
4. Clicking on a stream stall in Stalls view will go to the start of the stall in Trace view. Right-click the stall and select **Filter Trace** as needed. After filtering trace, the signals related to the stall are shown in the Trace view. Non-related signals are hidden. Exploring the trace using filter trace is clearer when the design is large.
 5. Objects in blue in the Stalls view can be clicked and cross-probed. For example, clicking the kernel in Stalls view will highlight the kernel in Trace view.
 6. Zoom in and out of the Trace view to explore the stalls. From the position of the stall, how frequently do similar stalls occur, events before the stall and related stalls (if any), can give you a hint on why the stall occurs.
 7. To clear the previously filtered trace, right-click and select **Clear All Filters**.
 8. It is helpful to have an overview of the stall path in Graph view. Select **Graph** view and then select **Tile View** from the drop-down list.

Figure 29: Stall Path in Graph View

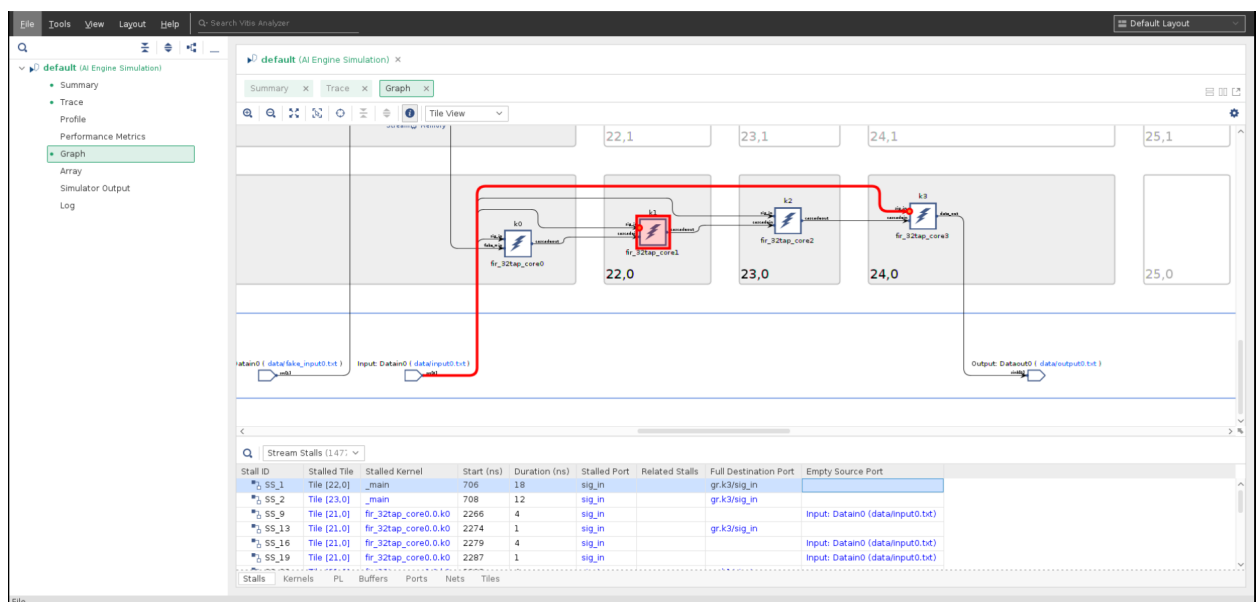


9. Select **Stalls** view and then select **Stream Stalls** from the drop-down list.
10. Explore the stream stalls in the Stalls view. Click on a stream stall in the Stalls view to have an overview of the stall in the graph. The red path shows where the stall occurs. It can be from a stalled kernel to full destination port, or from an empty source port to the stalled kernel.



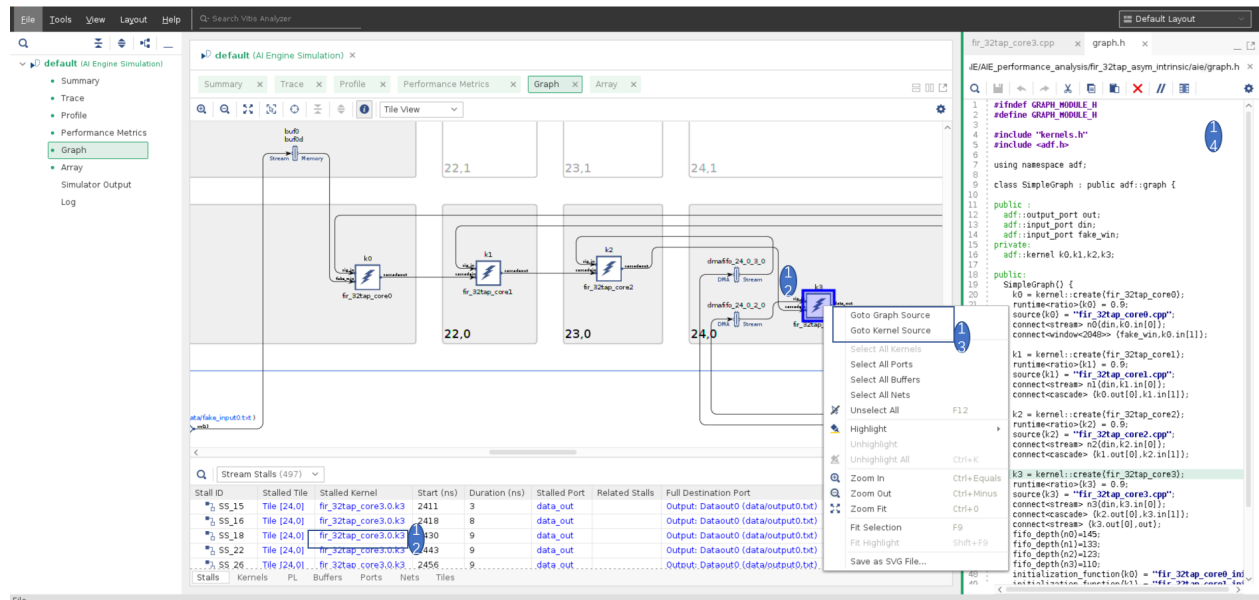
TIP: If a stream multicasts to multiple destinations and a stream stall occurs when the stream does not have enough FIFO for all destinations, the highlighted stalled kernel and stalled net may not be connected (separately in red). Meaning that all destinations of the multicast stream as a whole must be analyzed for the stream stall. An example of multicast stream stall in Graph view is shown in the following figure.

Figure 30: Multicast Stream Stall Path



11. It can open graph source code or kernel source code from Graph view or Array view. Select the kernel instance by clicking the kernel object in the Stalls view or clicking the kernel in the Graph view.

Figure 31: Viewing Graph Code and Kernel Code



12. Right-click the kernel instance in the Graph view, and select either **Goto Graph Source** or **Goto Kernel Source**. It will open the graph source code or kernel source code.
13. Correlate the graph source code and kernel source code with the stalls analyzed, editing the source code as needed.

The following table lists some possible scenarios that can cause a stream stall and possible solutions.

Table 52: Stream Stall Scenarios and Solutions

Source	Destination	Stall Type	Possible Solution	Notes
Stream	Stream	Stream stall	<ul style="list-style-type: none"> Increase FIFO depth. See FIFO Depth Constraints. Adjust stream read and write instructions in source or destination kernels. 	
Stream	Multiple streams	Stream stall	<ul style="list-style-type: none"> Increase FIFO depth. See FIFO Depth Constraints. Insert DMA FIFO or set different FIFO depth for different destination nets. See FIFO Depth Constraints. 	Multicast
Stream	Multiple streams of multiple kernels in same AI Engine	Stream stall	<ul style="list-style-type: none"> Put multiple kernels into different AI Engines Add enough FIFO to streams to the kernels. 	Multicast
Multiple streams	Multiple streams	Stream stall	<ul style="list-style-type: none"> Adjust instructions to match between different streams. Increase FIFO depth (ssFIFO or DMA FIFO). 	

Table 52: Stream Stall Scenarios and Solutions (cont'd)

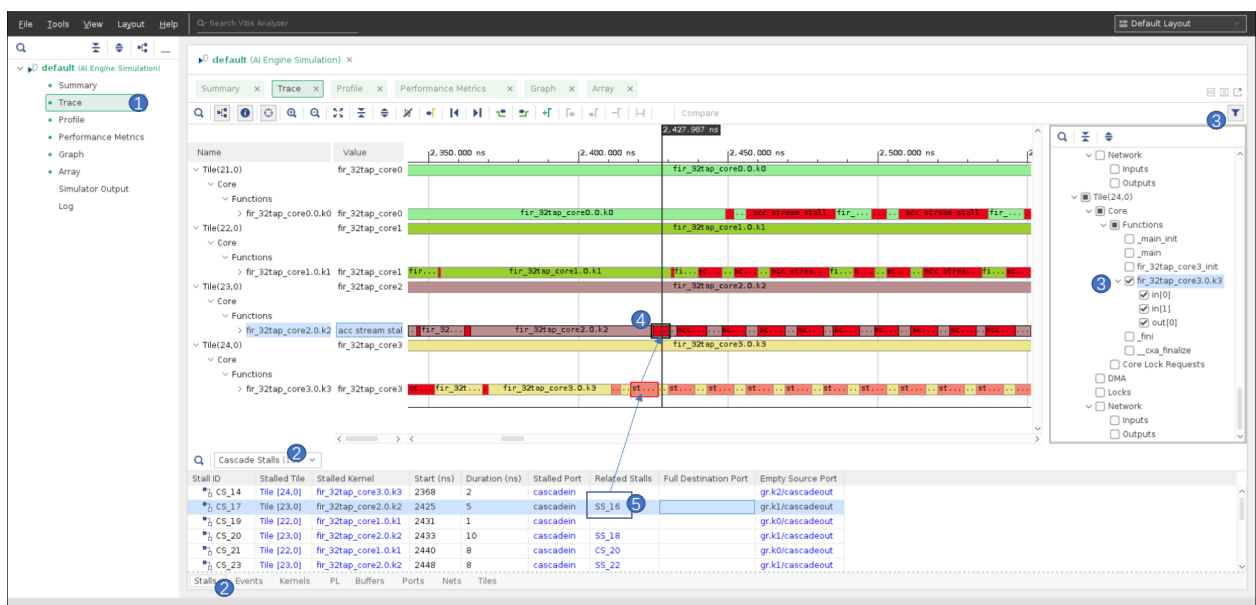
Source	Destination	Stall Type	Possible Solution	Notes
PLIO	Stream	Stream stall	<ul style="list-style-type: none"> Maximize AI Engine-PL interface bandwidth. For example, 64-bit interface, highest frequency (1/2 AI Engine frequency) for PL. Or 128-bit interface (note this uses two 64-bit channels for a 128-bit interface). See AI Engine-PL Interface Performance. 	
Stream	PLIO	Stream stall	Same as above.	
Stream (32 bits per iteration)	PLIO (64 bits)	Stream stall	Send TLAST for each 32 bits. See AI Engine-PL Interface Performance .	

Cascade Stall Analysis

From Performance Metrics analysis, you can identify the cascade stall that needs to be analyzed.

1. Choose Trace view.
2. Choose Stalls view and select **Cascade Stalls** from the drop-down list.
3. The filter button on the upper-right can be used to select the signals in the Trace view. It is helpful to show all cascade stalls in a window to better understand how they occur in a timeline because cascade stalls are usually correlated to each other in the cascade chain. Click on the filter button, de-select **All** at the top and select all kernels of interest.
4. Select **Cascade stall** to highlight the stall in Trace view.

Figure 32: Cascade Stall in Trace View



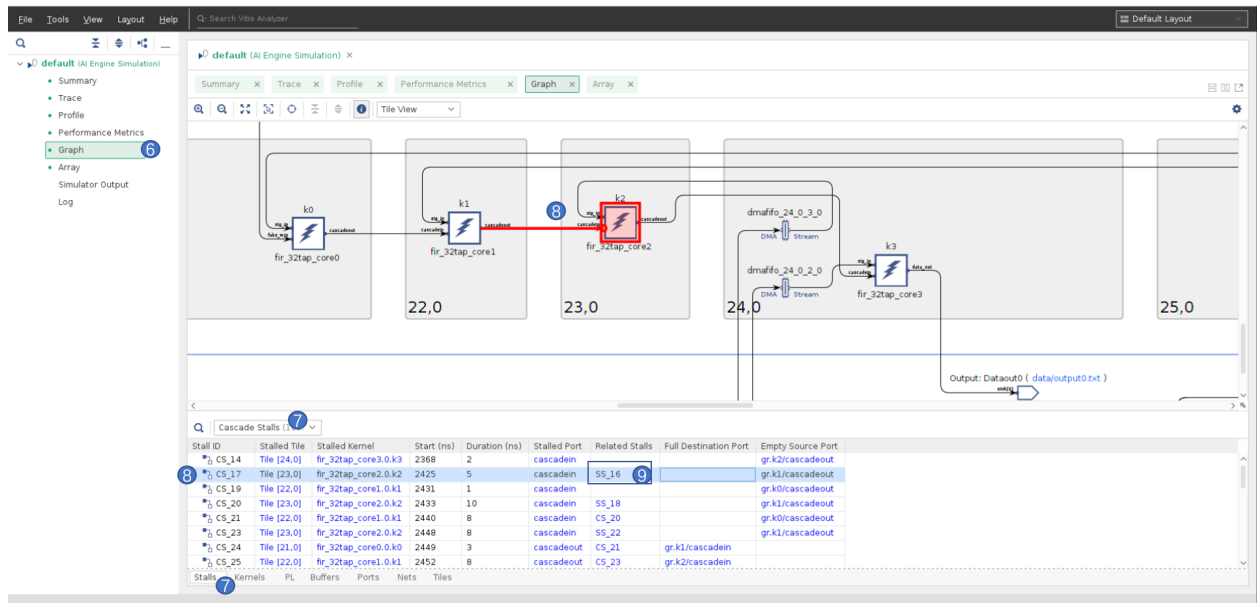
Each cascade stall has the following information.

- **Stall ID:** The stream stall is named CS_<NUM>. The earlier the stall happens, the smaller the number. The number is unique across all types of stall.
- **Stalled Tile:** The AI Engine tile where the stalled kernel is located.
- **Stalled Kernel:** The kernel that is stalled. It is named `<Kernel_function_name>.<Schedule_ID>.<Graph_instance_name>`. Sometimes it is shown as `_main` and then cross-probe is required to find the real kernel function.
- **Start (ns):** The start time that the stall happens
- **Duration (ns):** The duration of the stall.
- **Stalled Port:** Which port of the stalled kernel is stalled.
- **Related Stalls:** Other stalls that might cause the stall.
- **Full Destination Port:** The port that the stalled kernel cannot write into because it is full.
- **Empty Source Port:** The port that the stalled kernel cannot read from because it is empty.

Explore the cascade stalls in the Trace view and see how they are related. Related stalls in the Stalls view show which stall causes the cascade stall.

5. It is possible that other types of stalls cause the cascade stall. Explore the other types of stalls to analyze why that happened.
6. Viewing the stalls happen in Graph view can help identify the cause of the stall. Select **Graph view**.
7. Select **Stalls view** and then select **Cascade Stalls** from the drop-down list.
8. Click a path in the Stalls view and the stall path is shown in red in Graph view.
9. Exploring the cascade stall and its related stalls, can help in finding a hint about the cause of the stall. Clicking **Related Stalls** can also show the related stall in red. For example, by clicking SS_16 in Related Stalls, you can see the path in red, which gives a hint about where the stall starts.

Figure 33: Cascade Stall in Graph View



The following table lists some possible scenarios that cause cascade stall and possible solutions.

Table 53: Cascade Stall Scenarios and Solutions

Source	Destination	Stall type	Possible solution
Cascade Stream	Cascade Stream	Cascade Stall	<ul style="list-style-type: none"> Adjust cascade stream read/write instructions in kernel execution cycles to match between source and destination kernels Analyze other sources that cause the cascade stall
Stream + Cascade Streams	Stream + Cascade Streams	Cascade Stall	<ul style="list-style-type: none"> Adjust instructions to match between different streams Analyze other sources that cause the cascade stall

Memory Stall Analysis

AI Engine can perform several vector load or store operations per cycle. However, for the load or store operations to be executed in parallel, they must target different memory banks. Memory stall happens when multiple access on the same bank of memory on the same cycle.

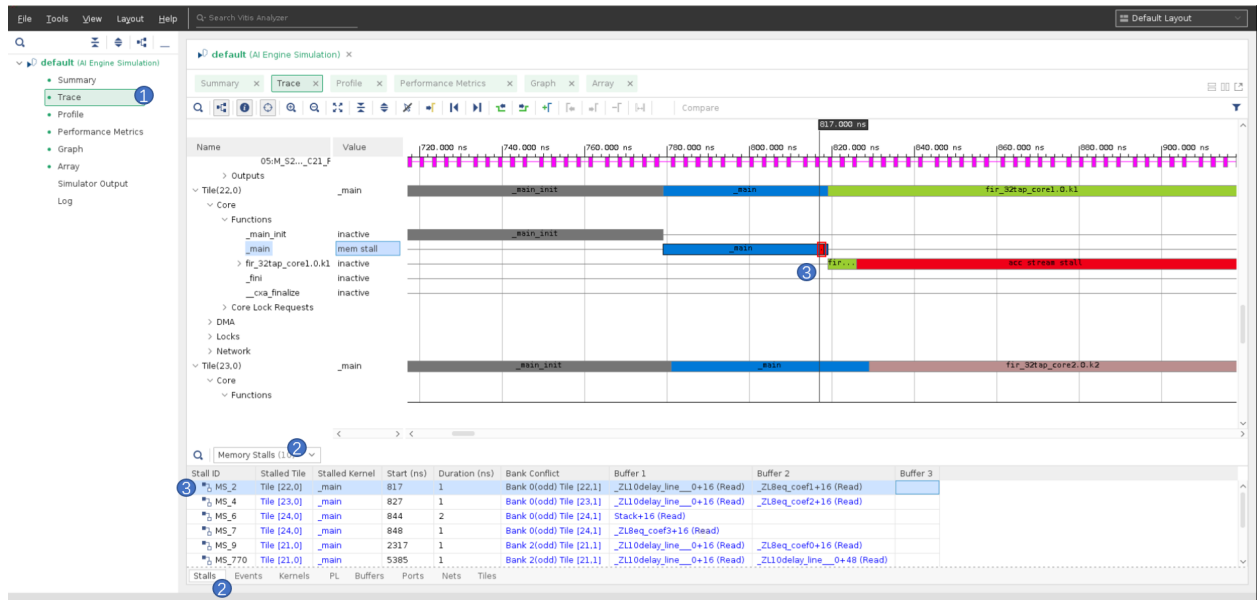
The kinds of memory include window buffer and DMA FIFO between kernels, RTP buffer, and system memory. System memory includes kernel synchronization information in the first 32 bytes, stack, and heap. Static variables are in the heap and function control logics are in the stack. System memory occupies continuous memory banks. Tool can automatically or manually place window buffer, RTP buffer, DMA FIFO, and system memory on specific banks. To alleviate memory stalls between these memories, try to place them into separate banks if possible. But memory stall can still happen between these kinds of memories if separate banks cannot be found for all the memories, or multiple accesses are happening on the same memory.

In general, the compiler tries to schedule many memory accesses in the same cycle when possible, but there are some exceptions. Memory accesses coming from the same pointer are scheduled on different cycles. If the compiler schedules the operations on multiple variables or pointers in the same cycle, memory bank conflicts can occur. Each memory bank has its arbitrator to arbitrate between all requests, and the arbitration is round-robin. After every request has been addressed, the memory stall is released.

From Performance Metrics analysis, you can identify if the memory stall needs to be analyzed.

1. Select Trace view.
2. Choose Stalls view in the bottom and select **Memory Stalls** from the drop-down list.

Figure 34: Memory Stall in Trace View



The stall is named as MS_<NUM>. The number is increased by time. Each stall has the following associated information.

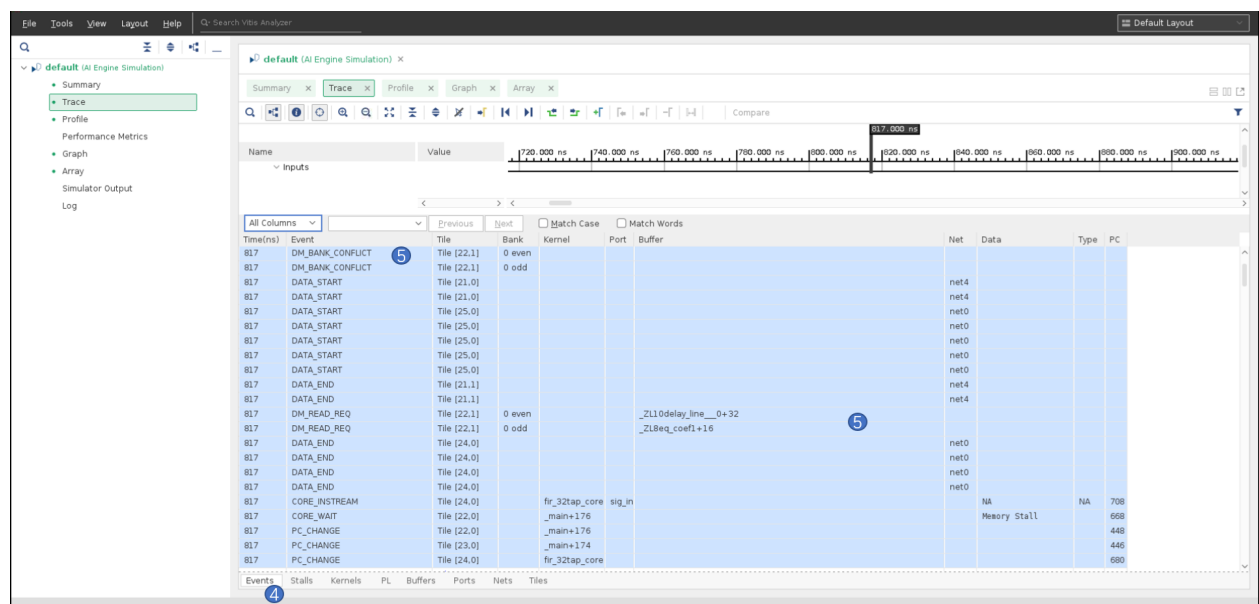
- **Stall ID:** The memory stall id. The earlier the stall happens, the smaller the number. The number is unique across all types of stall.
- **Stalled Tile:** The AI Engine tile where the stalled kernel is located.
- **Stalled Kernel:** The kernel that is stalled. It is named <Kernel_function_name>.<Schedule_ID>.<Graph_instance_name>. Sometimes it is shown as _main and then cross-probe is required to find the real kernel function.
- **Start (ns):** The start time that the stall happens
- **Duration (ns):** The duration of the stall.
- **Bank Conflict:** The memory bank where the stall happens on.

- **Buffer 1, Buffer 2, Buffer 3:** The buffers that cause the memory stall. It can be one buffer or multiple buffers.
3. When you click each line of the stalls in the Stalls view, it goes to the start of the memory stall in Trace view. Zoom in and out of the Trace view to observe how frequently the memory stalls occur and the position of the stall in kernel running.

Note: If large number of memory stalls occur repeatedly in the running kernel, it indicates that the stalls can happen inside loop. It is best to investigate and resolve. If memory stalls only happen once at the start of the kernel running, or a very small number of stalls happen in kernel running, it can usually be neglected. From the name of the buffers that cause the stall, it can be identified whether it is window buffer or system buffer or something else. If it is a window buffer or RTP buffer that can be controlled in the graph, one way is to place it manually using constraints if better placement can be identified. If it is system memory (named `system<NUM>+<NUM>`), it is required to identify the variables that are involved in the stall.

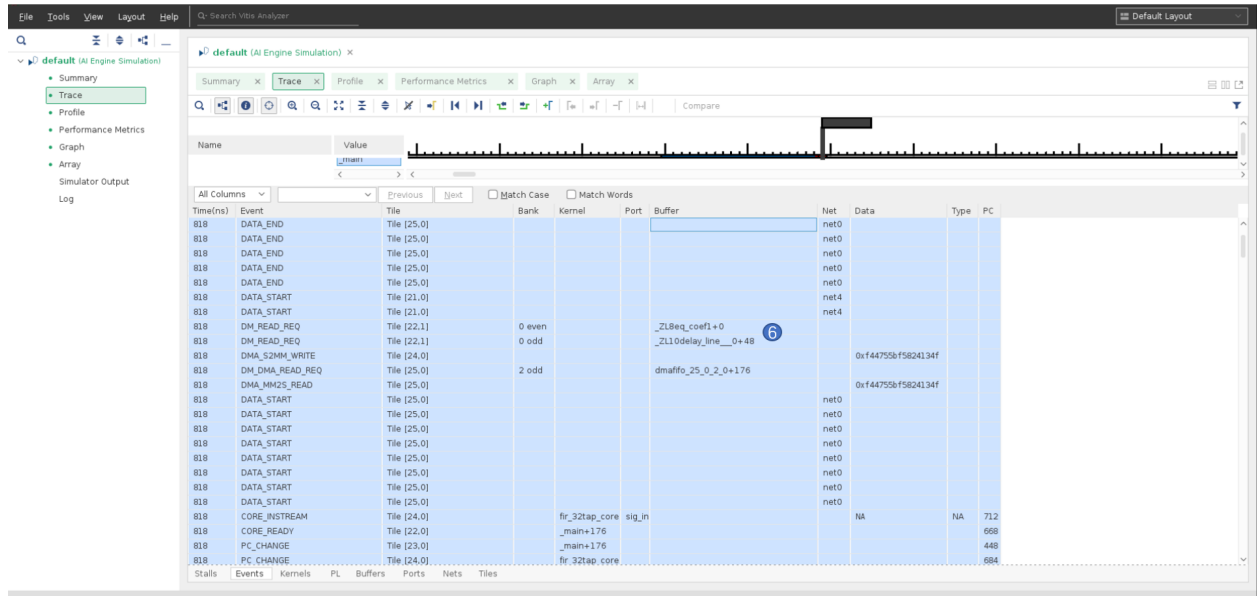
4. Click the row of the specific stall and switch to Events view.

Figure 35: Events View of Memory Stall



5. The Events view shows the events that happen in the device. The cycle where the memory stall happens is highlighted. You can see the tile on which the `DM_BANK_CONFLICT` event has occurred and the variables that are being read or written. In the previous figure, the variables `delay_line` and `eq_coef1` are being read at the same cycle.
6. Try to explore some cycles before or after the stall cycle to find more hints. For example, the following figure shows the events happened after the previous stall cycle. It is seen that `delay_line` and `eq_coef1` are also being read at the same cycle but different part (128 bits). By examining the source code and assembly code, it can be found that `delay_line` and `eq_coef1` are both issued 256 bits at the same cycle, and that causes the memory stall. The two 256 bits memory access are split into two cycles due to the memory stall.

Figure 36: Events View of Memory Stall



One way to resolve it is documented in the Load and Store with Virtual Resource Annotations section in *AI Engine Kernel Coding Best Practices Guide* (UG1079). For example, redefine points to `eq_coef0` and `delay_line` and annotate them with `__aie_dm_resource_a`. Use the following new points instead.

```
const v8cint16 __aie_dm_resource_a* __restrict coeff = (v8cint16
__aie_dm_resource_a*) eq_coef0; const v8cint16 coe = *coeff;

v16cint16 __aie_dm_resource_a* __restrict p_buff = (v16cint16
__aie_dm_resource_a*) &delay_line; v16cint16 buff=*p_buff;
```

The following table lists some possible scenarios that cause memory stalls and possible solutions.

Table 54: Memory Stall Scenarios and Solutions

Source	Target	Stall Type	Possible Solution	Note
One kernel	Buffers on one memory bank	Memory stall	<ul style="list-style-type: none"> Dispatch memories to different banks (memories include system memory, RTP, window buffer, DMA, and FIFO). See Memory Stalls. Guide compiler scheduling with virtual memory annotations. See Load and Store with Virtual Resource Annotations section in <i>AI Engine Kernel Coding Best Practices Guide</i> (UG1079) 	<p>One kernel accesses memories on the same bank.</p> <p>Or one kernel has multiple accesses on one memory on the same bank.</p> <p>(a cycle can have two loads and one store)</p>

Table 54: Memory Stall Scenarios and Solutions (cont'd)

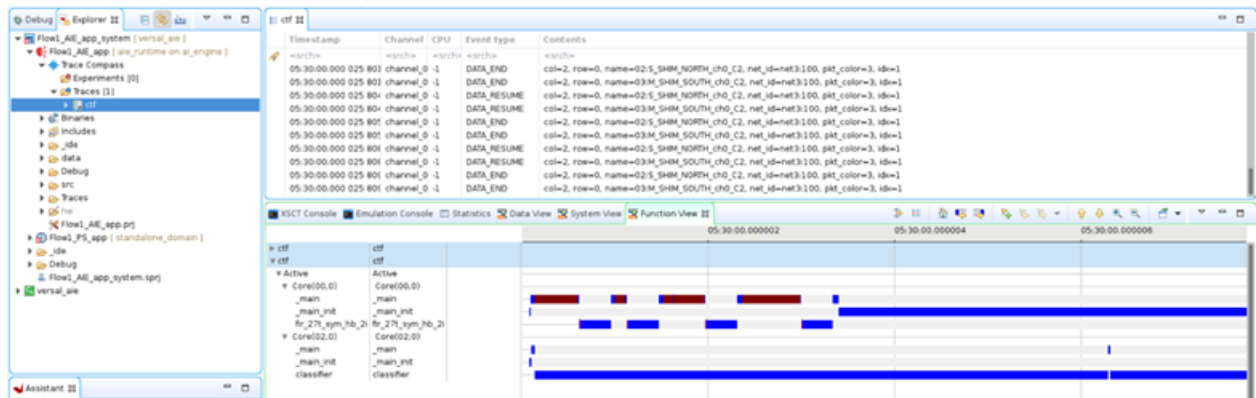
Source	Target	Stall Type	Possible Solution	Note
Multiple kernels on adjacent AI Engine tiles	Multiple buffers on one bank	Memory stall	<ul style="list-style-type: none"> Dispatch memories to different banks (memories include system memory, RTP, window buffer, DMA, and FIFO). BufferOptLevel. See Mapper and Router Options. If memory banks are exhausted, do profile and AI Engine stall analysis to find better solution with less kernel execution time or less stall percentage. 	Multiple kernels accessing multiple memories on the same bank.

Using Trace Compass to Visualize AI Engine Traces

The CTF-based AI Engine trace can be visualized using the free eclipse tool called Trace Compass. This tool is already integrated into the Vitis IDE as a plug-in, allowing you to visualize your traces from the Vitis IDE main panel.

Note: You can download standalone versions and documentation of Trace Compass from their website at <http://tracecompass.org>.

1. In the Vitis IDE, after capturing trace data during simulation you can right-click on your project then select **Analyze AIE Events**. This imports the event data from the simulation and create various views to analyze them. Your screen could look as follows:



2. To see various views, toggle between the Statistics, Data View, System View, and Function View tabs.

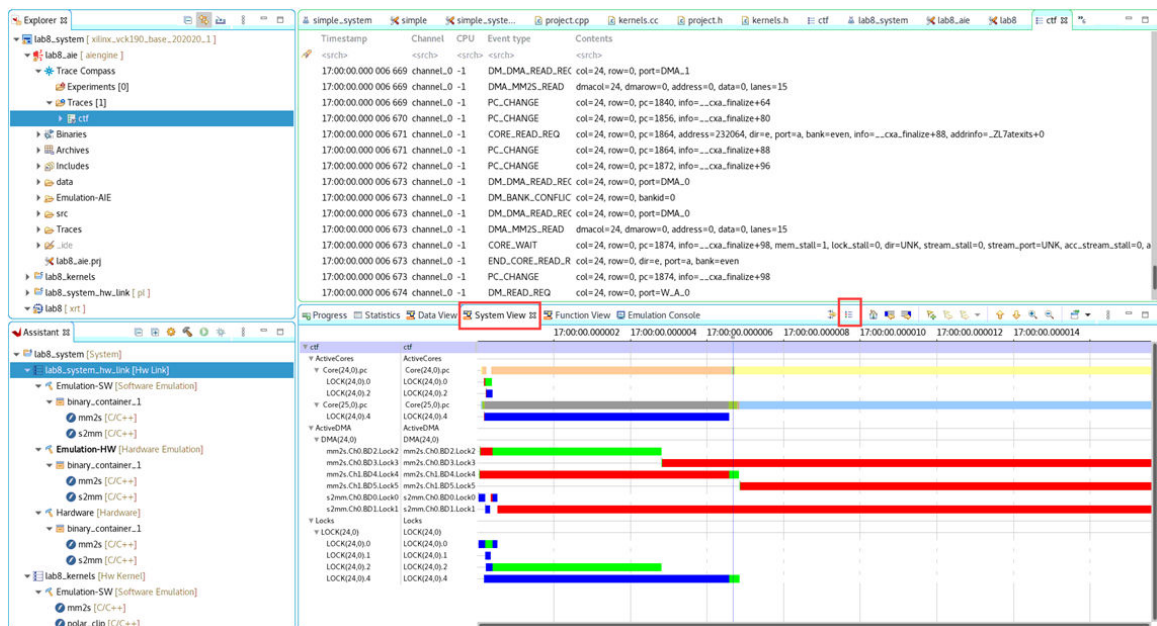
Trace Views

The trace reports support several views:

- The top window shows a textual list of events in chronological order with various event types and other relevant information. The top row in each column allows you to filter events based on textual patterns. In the bottom window, there are multiple tabs providing different views relating to the execution.
- The Statistics tab shows the aggregate event statistics based on the selected set of events or a time slice.
- The System View tab represents the state of system resources such as AI Engines, locks, and DMAs.
- The Function View tab represents the state of various kernels executing on an AI Engine (core).
- The Data View tab represents the state of data flowing through the stream switch network.

The following are screen shots of the function view, system view, and data view. The top bar of a view has several options: A legend explaining the colors, zoom in and zoom out, going to beginning and end of state, and correlating it to a textual event that causes the state change. Each view consists of a series of aligned timelines depicting the state of a certain resource or program object. Various events are represented in each timeline. You can hover over the timeline to see the information collected. Clicking on the timeline in one view creates a time bar that allows you to see the corresponding channel events at that time in other views.

Figure 37: System View with AI Engines, Locks, and DMAs



As shown in the system view, there are four sections: ActiveCores, ActiveDMA, and Locks. If there are PL blocks used in the application, the system view will also show the ActivePLPorts. By using lock IDs in the ActiveCores, ActiveDMA, and Locks sections you can identify how the AI Engines and DMAs interact with one another by acquiring and releasing locks. The currently executing function name is shown when hovering over the `Core(0,0).pc` bar. The color coding is shown in the legend that opens with a click on the legend icon (🔍, left of the home icon, which resets the timescale to default). Clicking the left or right arrows takes you to the beginning and end of a state, respectively. A text window shows you the event that caused the state change. In this example, all locks are properly acquired and released. If a lock is not released, you will see a red bar that extends through the end of simulation time.

Figure 38: Legend

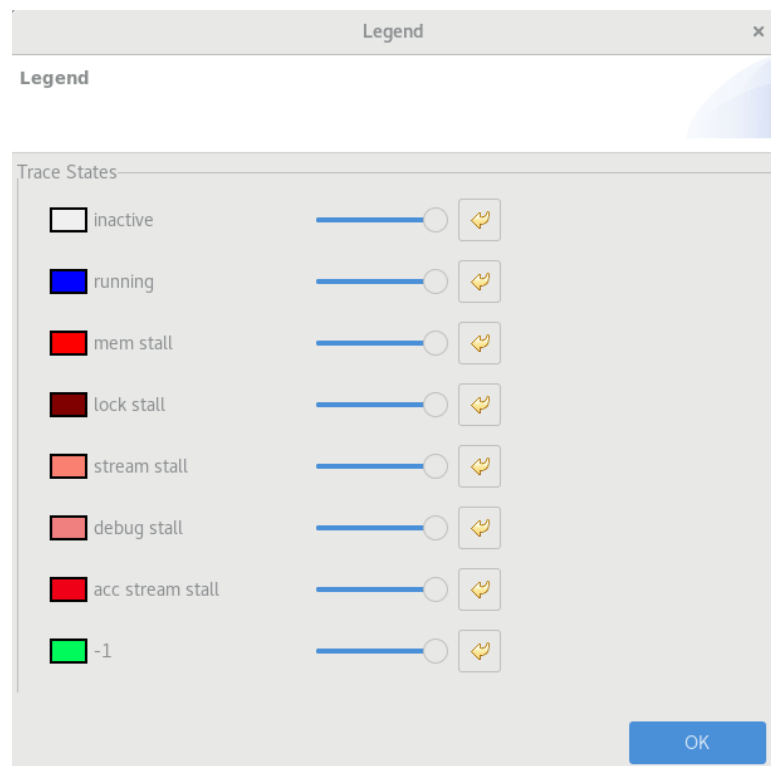
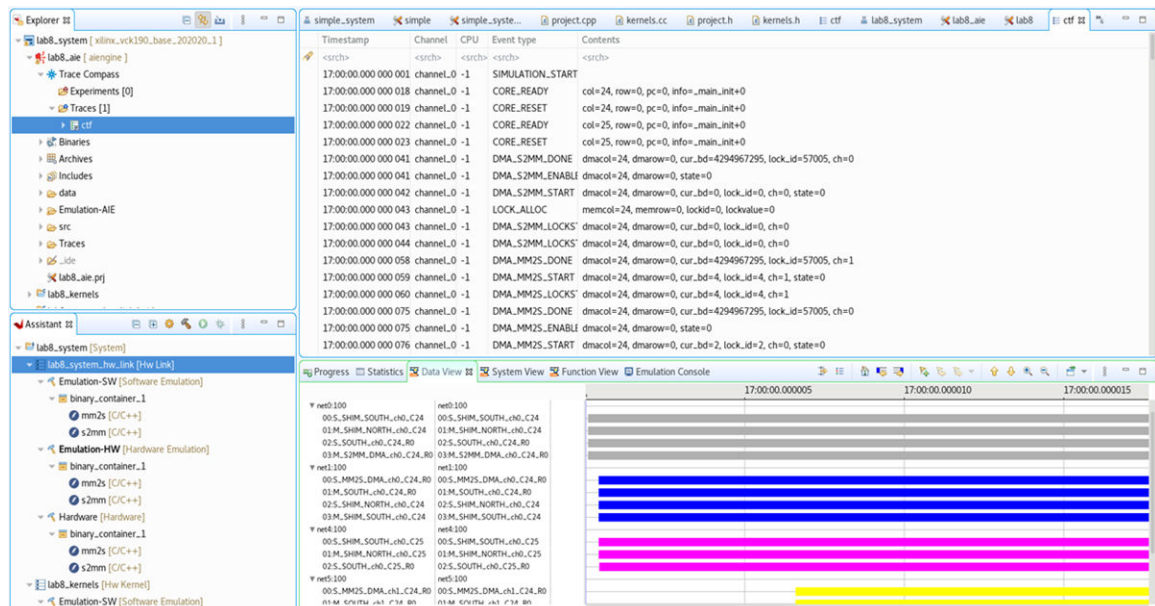


Figure 39: Function View Showing Running and Stalled Kernels and Main on Each AI Engine



The function view is most useful when analyzing the application from the program standpoint. There is a separate timeline for each kernel mapped to an AI Engine (core), and the view shows when the kernel is executing (blue) or stalled. A detailed pop-up window with details such as the types of stall and duration comes up when you hover over the stalls in the function view.

Figure 40: Data View Showing Data Flowing in the Stream Switch Network



The data view shows the data flowing through the stream switch network with slave entry points and master exit points at each hop. This is most useful in finding the routing delays, as well as network congestion effects with packet switching, when one packet might get delayed behind another packet when sharing the same stream channel.

Performance Analysis of AI Engine Graph Application on Hardware

A system-level view of program execution can be helpful in identifying problems during program execution including functional correctness and performance related challenges. While system level hardware and software emulation help during the development phase of the system, Xilinx offers flows around functional and performance debug in hardware as well. The AI Engine architecture has support for generation, collection, and streaming of profile related data, and events as trace data during hardware execution.

Profiling the AI Engine

Profiling is a form of dynamic program analysis that measures the input/output throughput, the space (memory), time complexity of a program, the usage of specific instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization, and more specifically, performance tuning.

You can obtain profiling data when you run your design in simulation or in hardware at run-time. Analyzing this data helps you gauge the efficiency of the kernels, the stall and active times associated with each AI Engine, and pinpoint the AI Engine kernel whose performance might not be optimal. This also allows you to collect data on design latency, throughput, and bandwidth by using run-time event APIs in your PS host code.

You have two options to gather this information; the first option is built on top of the second option.

1. Use event APIs to profile the AI Engine graph inputs and outputs from the graph host application. This feature can be used in simulation and hardware flows.
2. Use performance counters built into the hardware to monitor AI Engine and memory module events. This feature can only be used in hardware.

Event Profile APIs for Graph Inputs and Outputs

You can collect profile statistics of your design by calling event APIs in your PS host code. These event APIs are available both during simulation and when you run the design in hardware.

The AI Engine has hardware performance counters and can be configured to count hardware events for measuring performance metrics. You can use the run-time event API together with the graph control API to profile certain performance metrics during a controlled period of graph execution. The event API supports only platform I/O ports (PLIO) to measure performance metrics such as platform I/O port bandwidth, graph throughput, and graph latency.



IMPORTANT! *These APIs can be used in AI Engine simulation, hardware emulation, and hardware flows.*

Profiling Platform I/O Port Bandwidth

The bandwidth of a platform I/O port can be defined as the average number of bytes transferred per second, which can be derived as the total number of bytes transferred divided by the time when the port is transferring or is stalled (for example, due to back pressure). The following example shows how to profile I/O port bandwidth using the event API. In the example, `gr` is the application graph object, `plio_out` is the PLIO object connecting to the graph output port, and the graph is designed to produce 256 int32 data samples in eight iterations.

```
gr.init();
event::handle handle = event::start_profiling(plio_out,
event::io_total_stream_running_to_idle_cycles);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
gr.run(8);
gr.wait();
long long cycle_count = event::read_profiling(handle);
event::stop_profiling(handle);
double bandwidth = (double)256 * sizeof(int32) / (cycle_count * 1e-9); //
byte per second
```

In the example, after the graph is initialized, the `event::start_profiling` is called to configure the AI Engine to count the accumulated clock cycles between the stream running event and the stream idle event. In other words, it counts the number of cycles when the stream port is in running or in stall state. The first argument in `event::start_profiling` can be a PLIO or a GMIO object, in this case, it is `plio_out`. The second argument is `event::io_profiling_option` enumeration, and in this case, the enumeration is set to `event::io_total_stream_running_to_idle_cycles`. `event::start_profiling` returns a handle, which will be used later to read the counter value and to stop the profile. After the graph finishes eight iterations, you can call `event::read_profiling` to read the counter value by supplying the handle. After profiling is done, it is recommended to stop the performance counter by calling `event::stop_profiling` with the handle so the hardware resources configured to do the profile can be released for other uses. Finally, the bandwidth is derived by dividing the total number of bytes transferred ($256 \times \text{sizeof}(\text{int32})$) by the time spent when the stream port is active ($\text{cycle_count} \times 1e^{-9}$, assuming the AI Engine is running at 1 GHz).

Profiling Graph Throughput

Graph throughput can be defined as the average number of bytes produced (or consumed) per second. The following example shows how to profile graph throughput using the event API. In the example, `gr` is the application graph object, `plio_out` is the PLIO object connecting to the graph output port, and the graph is designed to produce 256 int32 data in eight iterations.

```
gr.init();
event::handle handle = event::start_profiling(plio_out,
event::io_stream_start_to_bytes_transferred_cycles, 256*sizeof(int32));
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
gr.run(8);
gr.wait();
long long cycle_count = event::read_profiling(handle);
event::stop_profiling(handle);
double throughput = (double)256 * sizeof(int32) / (cycle_count * 1e-9); //
byte per second
```

In the example, after the graph is initialized, `event::start_profiling` is called to configure the AI Engine to count the clock cycles from the stream start event to the event that indicates $256 \times \text{sizeof}(\text{int32})$ bytes have been transferred, assuming that the stream stops right after the specified number of bytes are transferred. If the stream continues after the number of bytes transferred, the counter continues and never ends. The first argument in `event::start_profiling` is `plio_out`, the second argument is set to `event::io_stream_start_to_bytes_transferred_cycles`, and the third argument specifies the number of bytes to be transferred before stopping the counter. The graph throughput is derived by dividing the total number of bytes produced in eight iterations ($256 \times \text{sizeof}(\text{int32})$) by the time spent from the first output data to the last output data ($\text{cycle_count} \times 1e^{-9}$, assuming the AI Engine is running at 1 GHz).

Note: The DMA for the input window from PL is active after configuration. When the PL output is asserted, the input window can start receiving data even before `graph::run`. One way to profile a PLIO input is to assert the PL after `event::start_profiling`. Otherwise, call `event::read_profiling` immediately after `graph::wait` because the performance counter may not hit the stop condition and will run continuously.

Profiling Graph Latency

Graph latency can be defined as the time spent from receiving the first input data to producing the first output data. The following example shows how to profile graph throughput using the event API. In the example, `gr` is the application graph object, `plio_out` is the PLIO object connecting to the graph output port, and `gmio_in` is the GMIO object connecting to the graph input port.

```
gr.init();
event::handle handle = event::start_profiling(gmio_in, plio_out,
event::io_stream_start_difference_cycles);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
gr.run(8);
gr.wait();
long long latency_in_cycles = event::read_profiling(handle);
event::stop_profiling(handle);
```

In the example, after graph is initialized, `event::start_profiling` is called to configure the AI Engine to count the clock cycles from the stream start event of the input I/O port to the stream start event of the output I/O port. The first and the second argument in `event::start_profiling` can be GMIO or PLIO ports, representing the input and the output I/O port respectively. In this example, `gmio_in` is the input I/O port and `plio_out` is the output I/O port. The third argument is set to `event::io_stream_start_difference_cycles` enumeration. The counter value simply indicates the graph latency in cycles.

Profiling Port Throughput

Port throughput can be measured by a count of the number of samples are sent in a specific time. Xilinx provides `event::io_stream_running_event_count` enumeration to count the running event, which corresponds to the number of samples sent.

After the graph runs, and data transfer from or to the port is stable, the following code can be inserted in the host code to measure the port throughput.

```
int wait_time_us=2000000;
event::handle handle = event::start_profiling(*plio_port,
event::io_stream_running_event_count);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
long long count0 = event::read_profiling(handle);
usleep(wait_time_us);
long long count1 = event::read_profiling(handle);
```



```
event::stop_profiling(handle);
long long samples = count1 - count0;
std::cout << "num running samples: " << samples << std::endl;
std::cout << " Throughput: " << samples / wait_time_us << " MSPS " <<
std::endl;
```

This method can be used for an infinite running graph, or just to count how many samples are sent or received before the graph is stalled (for whatever reason).

To minimize the variance of accuracy, it is advised to run for many seconds in hardware. Accuracy of this method can vary in hardware emulation.

For the AI Engine simulator, this profiling method applies too. You need to replace `usleep` with the `wait` function in SystemC, and the wait time needs to be much smaller, because it is much slower in simulation. For example, the `sleep` function in the preceding code can be replaced with following function call for the AI Engine simulator.

```
wait(20, SC_US);
```

Run-Time Event API Performance Counters

Run-time event APIs use the performance counters in the AI Engine-PL interface tiles and the AI Engine-NoC interface tiles. There are two performance counters in each column of the interface tiles. This section lists the number of performance counter - used by each run-time event API. If the total number of performance counters used exceeds the availability of performance counters in a column of the interface tile, the API that cannot acquire the performance counter fails with the following error message in AI Engine simulator.

```
[AIE WARNING]: Unable to request resources. RscType: 0
ERROR: event::start_profiling: Failed to request performance counter
resources.
```

For hardware emulation or hardware flows, the following error message is used.

```
[XRT] ERROR: ERROR: event::start_profiling: Failed to request performance
counter resources.: Resource temporarily unavailable
```

Table 55: Run-Time Event API Performance Counters

Run-time Event Enumeration	Number of Performance Counters
<code>event::io_total_stream_running_to_idle_cycles</code>	1
<code>event::io_stream_start_to_bytes_transferred_cycles</code>	2
<code>event::io_stream_start_difference_cycles</code>	1 for input port, 1 for output port
<code>event::io_stream_running_event_count</code>	1

Note: Performance counters are released after `event::stop_profiling`. The run-time event API can acquire the same performance counters again after they are released.

Note: When multiple graph ports are mapped into the same interface tile, if run-time event APIs are used on these ports, they will compete for the performance counters in the same interface tile.

Profiling the AI Engine in Hardware

An AI Engine event gives information about the system at a specific instant in time. An event that is associated with a timestamp, type and set of data values is referred to as a payload. The interpretation of the payload depends on the type of the event. A timestamp allows ordering of events, computation of causality relationships, and implementation of verifiers on a sequence of events.

For event modeling purposes, the key modules of the AI Engine array are processor, DMA, lock modules, memory, and I/O streams. Each module can be viewed as an event generator/responder. Each module receives an event and responds to the event. As a response, new events could be generated. Events are classified based on the generator of the events. The timestamp is not mentioned explicitly in the event definitions. Each event is described with a payload, which is a couple of values associated with the event. Each AI Engine, memory, DMA, or lock is addressable by a two-dimensional index <col, row>, which is the column and row index in the AI Engine array. Some of the AI Engine events are shown in the following tables.

Events generated by the AI Engine and their corresponding IDs are listed in the following table.

Table 56: AI Engine Events

Event Number	Event Name	Comment
22	Group Core Stall	Any or all the events from event number 23 to 31 can trigger this event
23	Memory Stall	Event generated when core gets stalled due to memory conflict
24	Stream Stall	Event generated when core gets stalled either due to no data at input or due to backpressure on the stream output from core
25	Cascade Stall	Event generated when core gets stalled either due to no data at Input or due to backpressure on the stream output from core
26	Lock Stall	Event generated when core gets stalled due to lock being acquired already
28	Active	Event generated when core changes state from disabled to active
32	Group Core Program Flow	Any or all the events from event number 33 to 45 can trigger this event
37	Instr Vector	Event generated when ME core executes a "vector" instruction
38	Instr Load	Event generated when ME core executes a "load" instruction
39	Instr Store	Event generated when ME core executes a "store" instruction

Table 56: AI Engine Events (cont'd)

Event Number	Event Name	Comment
40	Instr Stream Get	Event generated when ME core executes a "read from stream" instruction
41	Instr Stream Put	Event generated when ME core executes a "write to stream" instruction
42	Instr Cascade Get	Event generated when ME core executes a "read from cascade stream" instruction
43	Instr Cascade Put	Event generated when ME core executes a "write to cascade stream" instruction
50	FP Overflow	Event generated when floating point overflow exception flag bit gets set
51	FP Underflow	Event generated when floating point underflow exception flag bit gets set
52	FP Invalid	Event generated when floating point invalid exception flag bit gets set
53	FP Div by Zero	Event generated when floating point div by zero exception flag bit gets set

Events generated by the AI Engine memory module and their corresponding IDs are listed in the following table.

Table 57: Memory Module Events

Event Number	Event Name	Comment
20	Group DMA Activity	Any or all the events from event number 21 to 40 can trigger this event
33	DMA S2MM 0 stalled lock acquire	Event generated when S2MM channel 0 is stalled on lock acquire
34	DMA S2MM 1 stalled lock acquire	Event generated when S2MM channel 1 is stalled on lock acquire
35	DMA MM2S 0 stalled lock acquire	Event generated when MM2S channel 0 is stalled on lock acquire
36	DMA MM2S 1 stalled lock acquire	Event generated when MM2S channel 1 is stalled on lock acquire
43	Group Lock	Any or all the events from event number 44 to 75 can trigger this event
76	Group Memory Conflict	Any or all the events from event number 77 to 84 can trigger this event
86	Group Errors	Any or all the events from event number 87 to 100 can trigger this event

Profiling the AI Engine and Memory Modules

There are two types of performance counters, run-time event performance counters for the AI Engine modules and run-time memory counters for memory modules. These performance counters can be configured to track a variety of events in the AI Engine and the memory module. Various features like error-correction code (ECC) scrubbing, event trace and profiling can use these performance counters. Performance counters count occurrences of a given event in a profile configuration. The profile feature offers several different configurations of these performance counters that can be dynamically applied at run-time to collect various profiling statistics.

No changes are required in PS host code when using performance counters. These counters can be configured, read and collected at run-time while the design is executing in hardware. The following table lists the number of performance counters that are available at different configurations.

Table 58: Table of Counters

Event Trace Used?	ECC Scrubbing Used?	Counters Available for Profiling	
		Core Module	Memory Module
No	No	4	2
No	Yes	3	2
Yes	No	2	0
Yes	Yes	1	0

The ECC scrubbing is ON by default and it can be turned ON/OFF using the AI Engine compiler option. For more information, see [AI Engine Compiler Options](#). When ECC scrubbing is enabled, three counters are available for profiling.

When performance counters are used for ECC scrubbing, event trace and profiling in the same execution, allocated performance counters cannot meet the requirements of all the requested features at the same time. The following warning messages indicate this situation.

Figure 41: Warning Message

```

Loading: 'a.xclbin'
XAIEFAL: INFO: Resource group Avail is created.
XAIEFAL: INFO: Resource group Static is created.
XAIEFAL: INFO: Resource group Generic is created.
XAIEFAL: INFO: Resource group Avail is created.
XAIEFAL: INFO: Resource group Static is created.
XAIEFAL: INFO: Resource group Generic is created.
XRT build version: 2.12.0
Build hash: 2719b6027e185000fc49783171631db03fc0ef79
Build date: 2021-10-09 04:14:02
Git branch: 2021.2
PID: 1234
UID: 0
[Tue Nov 23 13:34:03 2021 GMT]
HOST: versal-rootfs-common-2021_2
EXE: /media/sd-mmchlk0p1/host.exe
[XRT] WARNING: Only 1 out of 4 metrics were available for aie core module profil
ing due to resource constraints. AIE profiling uses performance counters which c
ould be already used by AIE trace, ECC, etc.
Available metrics : ACTIVE_CORE
Unavailable metrics : GROUP_CORE_STALL_CORE INSTR_VECTOR_CORE GROUP_CORE_PROGRAM
FLOW
[XRT] WARNING: Only 0 out of 2 metrics were available for aie memory module prof
iling due to resource constraints. AIE profiling uses performance counters which
could be already used by AIE trace, ECC, etc.
Available metrics :
Unavailable metrics : GROUP_MEMORY_CONFLICT_MEM GROUP_ERRORS_MEM

```

Profiling for AI Engine

The following tables list the pre-defined metric set configurations available for AI Engine, in order of priority by which they are assigned to the available counters.

Table 59: Heat_map

Metric Name	Event ID	Description
Active Time	28	Time AI Engine was active since it was enabled.
Stall Time	22	Time AI Engine was stalled. This stall includes AI Engine memory, stream, cascade, and lock stalls.
Vector Instruction Time	37	Time AI Engine spent executing instructions in the vector processor.
Cumulative Instruction Time	32	Time AI Engine spent executing load/store, stream get/put, lock acquire/release instructions.

These indicators help you understand the efficiency of the kernels that are implemented in the AI Engines. You can compare stall time with active time to determine if there is a data communication issue for each AI Engine.

Table 60: Stalls

Metric Name	Event ID	Description
Memory Stall Time	23	Time the AI Engine was not active due to a memory stall.
Stream Stall Time	24	Time the AI Engine was not active due to a stream stall.
Lock Stall Time	26	Time the AI Engine was in a lock stall.
Cascade Stall Time	25	Time the AI Engine was in a cascade stall.

A stall in an AI Engine can occur in various situations:

- A memory stall happens when multiple accesses to the same memory bank are requested from one core, multiple cores, and/or DMAs.
- Stream stalls occur when data production and consumption on a stream do not have the same rate, leading to input stream starvation or output stream overflow.
- A cascade stall is generated when the cascade writer does not have the same rate as the cascade reader.
- A lock stall happens if the window data producer does not have the same iteration rate as the window consumer.

Table 61: Execution

Metric Name	Event ID	Description
Vector Instruction Time	37	Time spent by the AI Engine on vector instructions: vector processor instruction and vector data load/store
Load Instruction Time	38	Time spent by the AI Engine on load instructions (move data from memory to registers)
Store Instruction Time	39	Time spent by the AI Engine on store instructions (move data from registers to memory)
Cumulative Instruction Time	32	Time spent by the AI Engine on memory and stream accesses and lock acquire/release

All these indicators allow you to estimate the efficiency of your kernel. To increase efficiency, you should optimize data access, favor vector instructions over scalar instructions, and use 128-bit access to streams whenever possible.

Table 62: Floating-Point

Metric Name	Event ID	Description
Floating-Point Overflow Exception	50	Number of floating-point overflow exceptions generated by AI Engine
Floating-Point Underflow Exception	51	Number of floating-point underflow exceptions generated by AI Engine
Floating-Point Invalid Exception	52	Number of floating-point Invalid exceptions generated by AI Engine
Floating-point Divide by Zero Exception	53	Number of floating-point divide by zero exceptions generated by AI Engine

Floating-point exceptions lead to erroneous results. You might have to recode your floating-point algorithm if you get too many exceptions, or even a single in a critical area of the code.

Table 63: Stream_put_get

Metric Name	Event ID	Description
Cascade Read Instruction Time	42	Time AI Engine spent executing read instructions on the cascade stream.
Cascade Write Instruction Time	43	Time AI Engine spent executing write instructions on the cascade stream.
Stream Read Instruction Time	40	Time AI Engine spent executing read instructions on data streams.
Stream Write Instruction Time	41	Time AI Engine spent executing write instructions on data streams.

Profiling for Memory Modules

The following tables list the pre-defined metric set configurations available for memory modules.

Table 64: Conflicts

Metric Name	Event ID	Description
Memory Conflict	76	Time taken due to data memory conflicts on any of the 8 banks of the memory module.
Cumulative Memory Errors	86	Time taken due to ECC errors in any of the Data Memory banks, as well as the 2x MM2S and the 2x S2MM DMAs.

Memory conflicts happen when two memory chunks reside in the same memory bank and are accessed either by the same AI Engine (using the two read ports) or by two different AI Engines. A potential solution is to constrain the locations of these memories to different banks. In order to get more details about which bank is causing these conflicts, you should analyze the events from an Emulation-AIE simulation.

Table 65: DMA_locks

Metric Name	Event ID	Description
Cumulative DMA Activity	20	Time taken due to stalled lock acquires on both the MM2S and S2MM channels of the DMA.
Cumulative DMA Lock Count	43	Lock Stall count on the DMA channels.

The four DMA channels (2xS2MM and 2xMM2S) are driven by Buffer Descriptors (BDs). The Cumulative DMA Activity is a count of the time taken due to stalled lock acquire events on all channels. All these DMA events will help you understand why some connections through the device are slower than expected.

Table 66: DMA_stalls_s2mm

Metric Name	Event ID	Description
S2MM Channel 0 Stalls	33	Time S2MM channel 0 was stalled on lock acquire.
S2MM Channel 1 Stalls	34	Time S2MM channel 1 was stalled on lock acquire

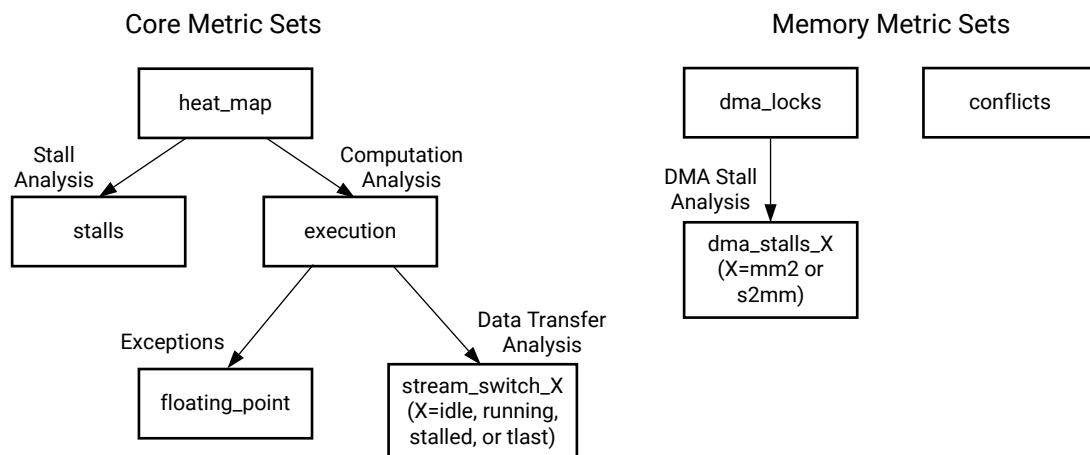
Table 67: DMA_stalls_mm2s

Metric Name	Event ID	Description
MM2S Channel 0 Stalls	35	Time MM2S channel 0 stalled on lock acquire
MM2S Channel 1 Stalls	36	Time MM2S channel 1 stalled on lock acquire

Core versus Memory Module Metric Sets

The following figure shows metric sets for both core modules and memory modules. These metric sets are mutually exclusive and can be used in a combination based on the requirements of your design. Initially, it is recommended to use the heat_map metric for core modules and conflicts metric for memory modules. These settings are specified at run-time and therefore, you can run as many times as you would like with your preferences. However, many of these metrics sets are interconnected because some use group events and some use solely individual events. For example, the heat_map metric contains a group stalls event and two events for core execution, vector instructions and group core instruction using all data transfer instructions (for example, load/store, stream and cascade put/get, and lock acquire/release). To get a better view of which stall type(s) are prevalent, re-run with the stalls set. To better understand core execution, re-run with the execution set.

Figure 42: Core vs Memory Modules Metric Sets



X26040-120121

Profiling Flows

The two different flows for configuring and capturing AI Engine profile counters are as follows:

- **XSDB Flow:** This flow provides a low-overhead method of communicating with the board, regardless of the operating system used on the host.
- **XRT Flow:** This flow is seamlessly integrated into Xilinx Runtime (XRT).

XSDB Flow

Set up `xsdb` as described in the following steps to connect to the device hardware.

1. When running the application, the profile data is captured in counters that can be retrieved by the debugging and profiling IP. To capture and evaluate this data, you must connect to the hardware device using `xsdb`. This command is typically used to program the device and debug applications. Connect your system to the hardware platform or device over JTAG, launch the `xsdb` command in a command shell, and run the following sequence of commands:

```
xsdb% connect
xsdb% ta 1
xsdb% source $::env(XILINX_VITIS)/scripts/vitis/util/aie_profile.tcl
xsdb% aieprofile start -graphs {myGraph1 myGraph2} \
    -work-dir <path to aiecompiler Work directory>
    -core-metrics heat_map \
    -memory-metrics dma_locks \
    -interval 20 -samples 100
```

where:

- `connect`: Launches the `hw_server` and connects `xsdb` to the device.
 - `source $::env(XILINX_VITIS)/scripts/vitis/util/aie_profile.tcl`: Sources the Tcl trace command to set up the `xsdb` environment.
 - `aieprofile start`: Instructs the DPA IP to begin capturing profile data based on configuration.
 - `-graph`: the graph profile data to be captured.
 - `-core-metrics`: the core metrics to be captured.
 - `-memory-metrics`: the memory metrics to be captured.
 - `-interval`: sample interval in milliseconds (default 20).
 - `-samples`: number of counter samples (default 100).
2. Run the design on hardware to generate hardware profile data. The XSDB aie profile script should still be running and sampling while the design application runs and until application finishes in order to capture valid statistics. The `-samples` and `-interval` switches can be adjusted accordingly to ensure this.

3. Use `vitis_analyzer` to import and analyze the data, as described [Viewing Profiling Results using Vitis Analyzer](#).

aieprofile Command Synopsis

```
aieprofile start [options] -tiles <tile-list>
```

```
aieprofile start [options] -graphs <graph-list>
```

Table 68: aieprofile Options

Option Name	Description
<code>-config-file <json-file></code>	JSON file generated by the AI Engine compiler at <code>Work/ps/c_rts/aie_control_config.json</code> with profile configuration data. This file can also be created for custom profile configuration. You can alternatively specify the <code>-work-dir</code> option instead of <code>config-file</code> .
<code>-core-metrics <metric-set></code>	This option allows you to set the core module counter metrics set. Valid values are <code>heat_map</code> , <code>stalls</code> , and <code>execution</code> .
<code>-memory-metrics <metric-set></code>	This option allows you to set the memory module counter metrics set. Valid values are <code>dma_locks</code> or <code>conflicts</code> .
<code>-graphs <graph-list></code>	This option allows you to set one or more graphs used in the AI Engine application.
<code>-tiles <tile-list></code>	This option allows you to set one or more tiles used in the AI Engine application. Tiles can be in the format of <code>col, row</code> (for example, <code>0,0, 0,1</code>). All the tiles in a row or column can be specified using wildcard <code>'*'</code> (for example, <code>0,*, *,1, *,*</code>).
<code>-work-dir <dir-path></code>	This option lets you set project Work directory.
<code>-interval</code>	This option lets you set the number of sample interval in milliseconds (default 20).
<code>-samples</code>	This option lets you set the exact number of counter samples (default 100) used in profiling the application.

XRT Flow

1. Burn the generated `sd_card.img` to the physical SD card.
2. Create the `xrt.ini` file in the same location where the PS host application is located. An example of `xrt.ini` file is as follows:

```
[Debug]
#
# Profile Counters
#
aie_profile = true
# Sample interval (in usec)
aie_profile_interval_us = 100
# All tiles
aie_profile_core_metrics = heat_map
aie_profile_memory_metrics = dma_locks
```

Where:

- [Debug]: specifies debug/profile settings for XRT, this is case sensitive.
 - aie_profile: enables profile configuration.
 - aie_profile_interval_us: profile data collection interval in micro seconds.
 - aie_profile_core_metrics: configure core metric to be applied.
 - aie_profile_memory_metrics: configure memory metric to be applied.
3. Run the design on hardware to capture profile data.
 4. Copy the generated profile files, `aie_profile_edge.csv`, `summary.csv`, and `xrt.run_summary`, from the SD card to your design at the same level as the `Work` directory. These generated files are at the same location as the host application on the SD card.
 5. Use `vitis_analyzer` to import and analyze the data as described in [Viewing Profiling Results using Vitis Analyzer](#).

xrt.ini Specification

```
[Debug]
#
# Profile Counters
#
aie_profile = true
# Sample interval (in usec)
aie_profile_interval_us = 100

# All tiles
aie_profile_core_metrics = <heat_map|stalls|execution|floating_point|
stream_put_get>
aie_profile_memory_metrics = <conflicts|dma_locks|dma_stalls_s2mm|
dma_stalls_mm2s>;
# Single tile
aie_profile_core_metrics = {<column>,<row>}:<heat_map|stalls|execution|
floating_point|stream_put_get>
aie_profile_memory_metrics = {<column>,<row>}:<conflicts|dma_locks|
dma_stalls_s2mm|dma_stalls_mm2s>;
# Range of tiles
aie_profile_core_metrics = {<mincolumn>,<minrow>}:
{<maxcolumn>,<maxrow>}:<heat_map|stalls|execution|
floating_point|stream_put_get>
aie_profile_memory_metrics = {<mincolumn>,<minrow>}:
{<maxcolumn>,<maxrow>}:<conflicts|
dma_locks|dma_stalls_s2mm|dma_stalls_mm2s>
```

Table 69: xrt.ini Options

Option name	Description
[Debug]	This option specifies debug/profile settings for XRT, this is case sensitive.
aie_profile	Enables profile configuration.
aie_profile_interval_us	This option lets you set the number of sample interval in micro-seconds (default 20).
aie_profile_samples	This option lets you set the exact number of counter samples (default 100) used in profiling the application.
aie_profile_core_metrics	Configure core metric to be applied for all tiles.
aie_profile_memory_metrics	Configure memory metric to be applied for all tiles.
aie_profile_core_metrics = {<column>,<row>}:	Configure core metric to be applied for single tile
aie_profile_memory_metrics = {<column>,<row>}:	Configure memory metric to be applied for single tile.
aie_profile_core_metrics = {<mincolumn>,<minrow>}:{<maxcolumn>,<maxrow>}:	Configure core metric to be applied for tiles in specified range.
aie_profile_memory_metrics = {<mincolumn>,<minrow>}:{<maxcolumn>,<maxrow>}:	Configure memory metric to be applied for tiles in specified range.

Note: If the total number of performance counters used exceeds the number of available performance counters, the API will not acquire a performance counter and fail with the following error message displayed on console.

```
[AIE ERROR]: Failed to request resource 0
[AIE WARNING]: Unable to request resources. RscType: 0
XAIEFAL: WARN: perfcount _reserve (6,1) Expect Mod= 1 resource not
available.
...
```

By default, `aie_profile_core_metrics` is set to `heat_map` core metric. You can specify an empty string in `xrt.ini` to disable this default setting as shown in the following example.

```
aie_profile_core_metrics =
```

Viewing Profiling Results using Vitis Analyzer

To launch the `vitis_analyzer` to view the profiling information in the XRT flow, use the following command.

```
vitis_analyzer xrt.run_summary
```

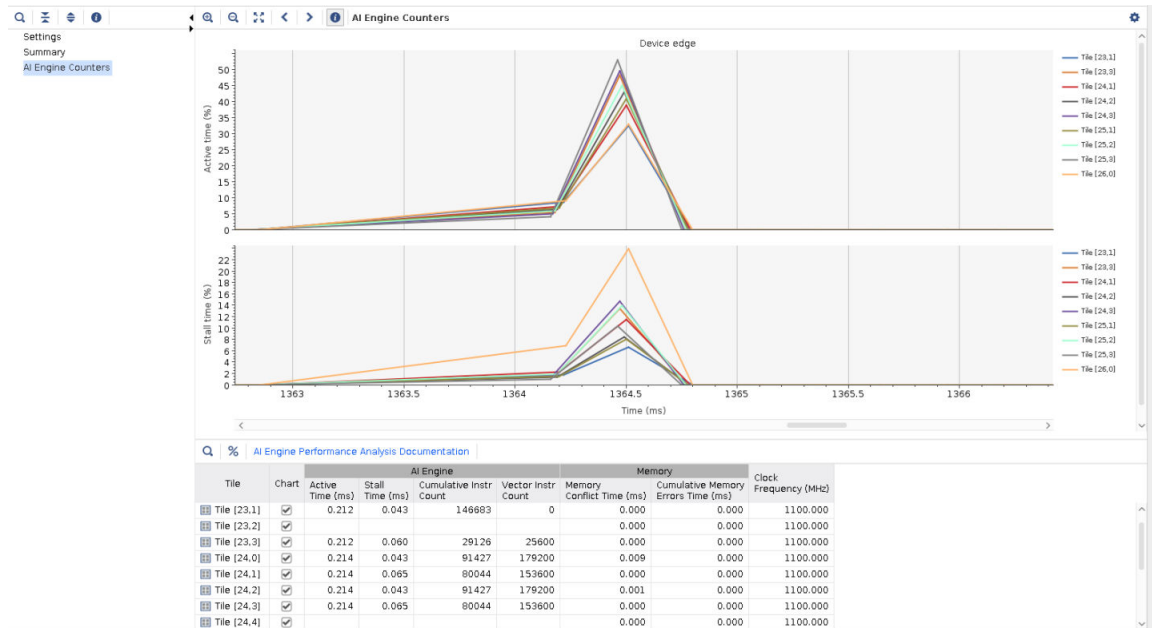
To launch the `vitis_analyzer` to view the profiling information in the XSDB flow, use the following command.

```
vitis_analyzer aie_trace_profile.run_summary
```

Example of Core_metrics: heat_map and Memory_metrics: conflicts

The following image shows the design's active time, stall time, cumulative instruction count, and vector_instruction_count as part of heat_map metric and memory conflict time, as well as cumulative memory error time of conflicts metrics for ten tiles of an example design.

Figure 43: Example of Core_metrics: heat_map and Memory_metrics: conflicts



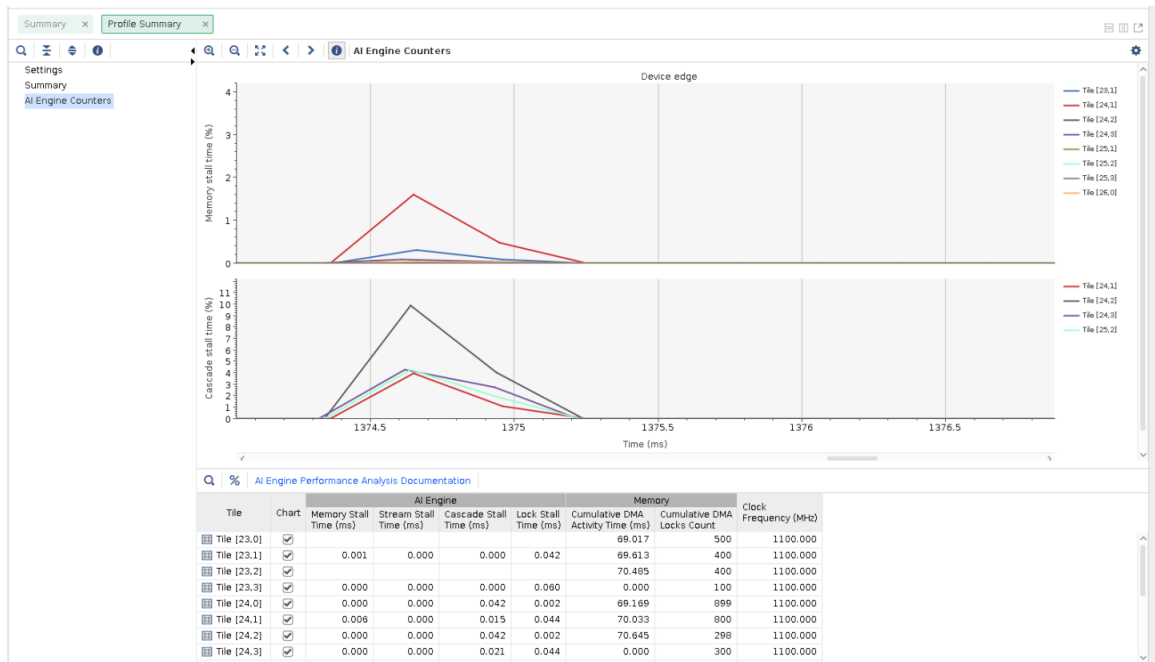
Note: Click on this icon  in the upper-right corner to enable/disable charts.

Consider the AI Engine located in (24,2). The stall time (.043 ms) is 20% of the active time (.214 ms). During this active time, it performs 179200 vector instructions, which represents 95% of the active time. This is an excellent performance that indicates a well optimized core.

Example of Core_metrics: stalls and Memory_metrics: dma_locks

The following image shows the design's memory stall time, stream stall time, cascade stall time, and lock stall time as part of stalls_metrics and cumulative DMA activity time, as well as cumulative DMA locks count of dma_locks_metrics for ten tiles of an example design.

Figure 44: Example of Core_metrics: stalls and Memory_metrics: dma_locks

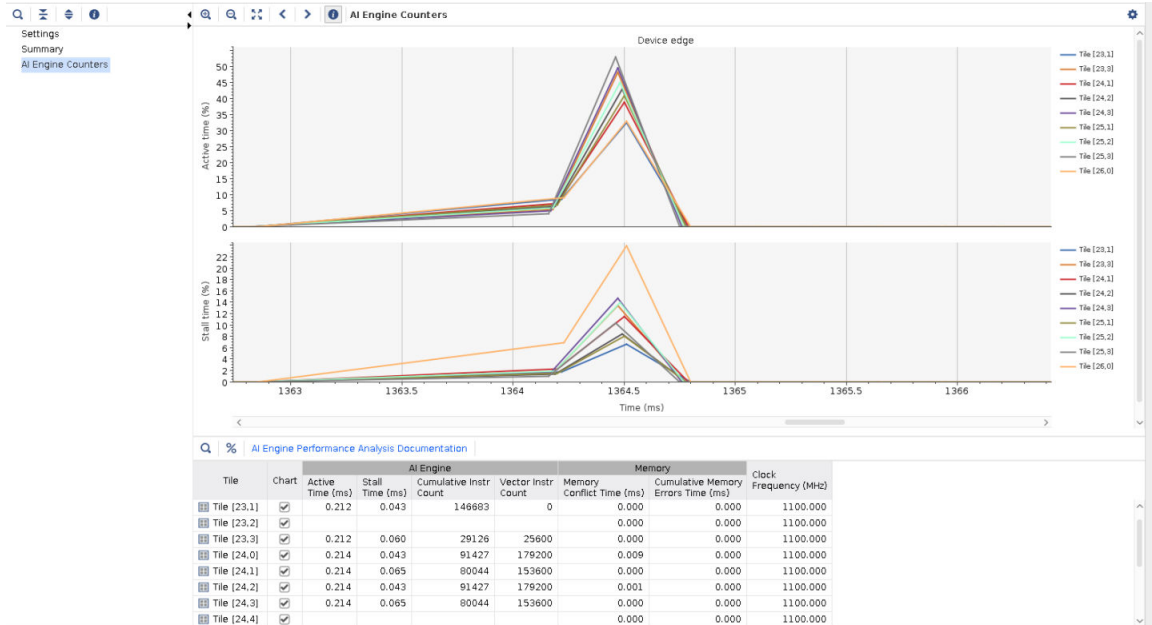


On the core (24,2), the DMA has been active for 70.645 ms (77.8 millions instructions), but has been stalled during 298 times. This does not indicate stalls in 298 instructions, because a stall can last multiple clock cycles.

Example of Core_metrics : execution and Memory_metrics: conflicts

The following image shows the design's cumulative instruction count, vector instruction count, load instruction count, and store instruction count as part of execution_metrics and memory conflict time, as well as cumulative memory error time of conflicts_metrics for ten tiles of an example design.

Figure 45: Example of Core_metrics : execution and Memory_metrics: conflicts

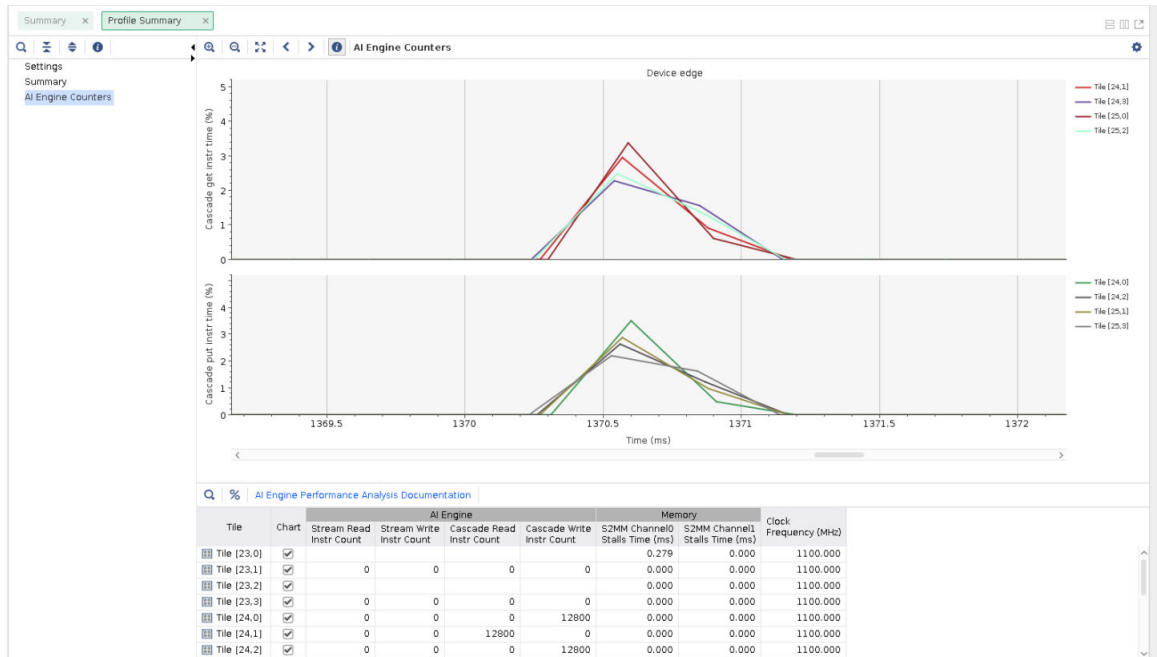


Although they are minor, core (24,2) suffers from some memory conflicts that must be identified. The occurrence being very small might be due to some DMA or some other kernel access interference.

Example of Core_metrics : stream_put_get and Memory_metrics: dma_stalls_s2mm

The following image shows the design's stream read instruction count, cascade read instruction count, and cascade write instruction count as part of stream_put_get_metrics and s2mm channel0 stalls time time, as well as s2mm channel1 stalls time of dma_stalls_s2mm_metrics for ten tiles of an example design.

Figure 46: Example of Core_metrics : stream_put_get and Memory_metrics: dma_stalls_s2mm

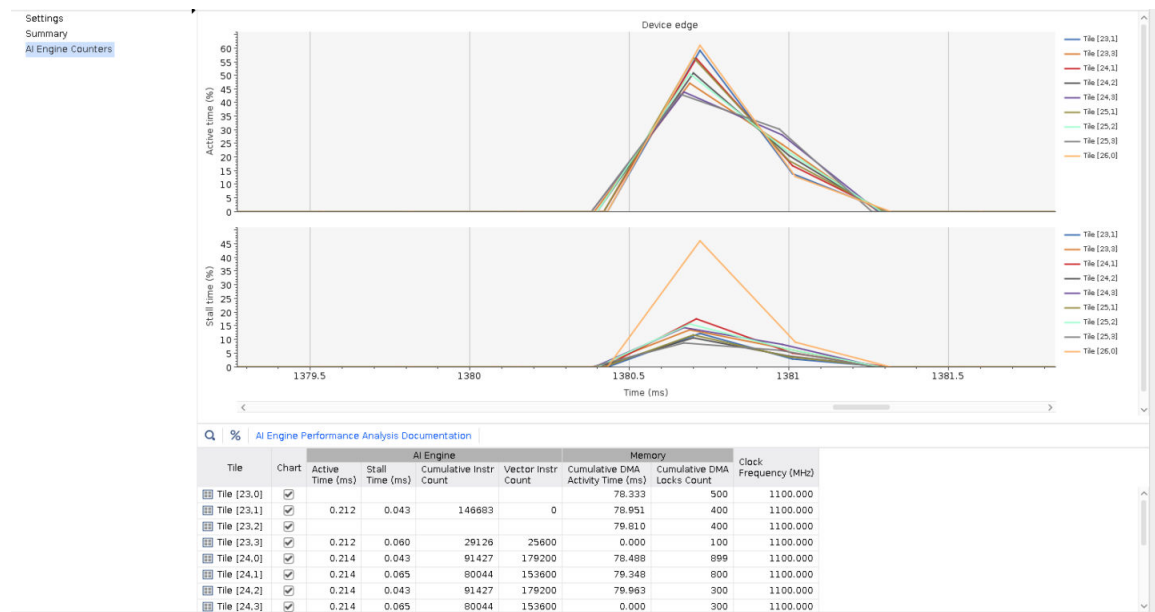


The graph shows that the core (25,1) writes to the cascade stream 3% of the time. (24,1) is the reading for the same amount of time from this cascade stream.

Example of Core_metrics : heat_map and Memory_metrics: dma_locks

The following image shows the design's active time, stall time, cumulative instruction count and vector_instruction_count as part of heat_metrics and cumulative DMA activity time, as well as cumulative DMA locks count of dma_lock_metrics for ten tiles of an example design.

Figure 47: Example of Core_metrics : heat_map and Memory_metrics: dma_locks



The cumulative DMA Activity time jointly with the Cumulative DMA Locks count allows you to see if there is any discrepancy between lock acquisition number and the number of data transferred through the DMAs. The relative number of locks count can also be used to interpret the relative number of iterations of each core.

Event Tracing in Hardware

AI Engine event trace tools provide an in-depth investigation into the operation and performance of a design. In support of this, a number of settings are required to capture trace data at run-time. In hardware, you must prepare the design when compiling the AI Engine graph application to ensure the `libadf.a` supports capturing trace data at run-time. Event tracing in hardware builds begins with the `aiecompiler` command, using the `--event-trace` option. This option sets up the hardware device to capture AI Engine run-time trace data by informing the `v++` linker to add and configure debugging and profiling IP (DPA) to the PL region of the device.

The event trace flow consists of three parts as follows.

- Event trace build flow
- Running the design in hardware and capturing trace data at run-time
- Viewing and analyzing the trace data using the Vitis analyzer

Event Trace Build Flow

The event trace build flow is as follows.

1. Compile the graph with `--event-trace` and other appropriate flags.

An example of the AI Engine compiler command for event tracing is as follows:

```
aielcompiler --verbose --pl-freq=100 --workdir=./myWork --event-trace-
port=gpio \
--event-trace=runtime --num-trace-streams=1 --broadcast-enable-core=true
\
--xlopt=0 --include="${XILINX_HLS}/include" \
--include="." --include="./src" --include="./src/kernels" --include="." \
data" \
./src/graph.cpp
```

Note:

- The preceding example illustrates compiling the design with `--event-trace=runtime` configuration. When you use this option, you can configure the type of events that AI Engine captures during runtime.
- The `--event-trace-port=gpio` option uses GPIO instead of PLIO to capture event trace data. Xilinx recommends that you use the GPIO option as the event-trace-port configuration. This option uses the AI Engine-to-NoC event trace pathway. This avoids the usage of programming logic resources and avoids timing errors caused by the PL resource usage.
- Xilinx recommends using `gpio` option as the event-trace-port configuration. This option uses the AI Engine to NoC event trace pathway. This avoids the usage of programming logic resources and avoids timing errors caused by the PL resource usage.
- GPIO = AI Engine NoC event pathway
- PLIO = AI Engine to PL event trace pathway, in addition to also using PL resources to capture trace data

2. Compile and link the design using the Vitis compiler.

After compiling the AI Engine graph application, you must build the other elements of the system as described in [Chapter 14: Integrating the Application Using the Vitis Tools Flow](#). With `--event-trace` enabled in the `libadf.a` file from the AI Engine compiler, the system hardware generated by the Vitis compiler includes the compiled ELF file for the PS application, the compiled ELF files for the AI Engine processors, and the XCLBIN file for the PL. These are the elements you need to run the system on hardware.

- After linking to create the device binary, run the Vitis compiler `--package` step to create the `sd_card` folder and files needed to boot the device, as described in [Packaging](#). This step packages everything needed to build the `BOOT.BIN` file for the system. When packaging the boot files for the device, you must also specify the `--package.defer_aie_run` to load the AI Engine application with the ELF file, but wait to run it until `graph.run` directs it, as described in [Graph Execution Control](#).

A feature of the `aiecompiler --event-trace` option is that you can compile your design to capture data using the `runtime` argument. Using this option, you can compile the AI Engine graph to be set up for event trace, and specify the type of profile data to capture at run-time: `functions`, `functions_partial_stalls`, and `functions_all_stalls`. Therefore, you do not need to recompile the design to capture a different type of data. Using this feature reduces the need to recompile the graph and re-package the design.

Table 70: Events Supported in Predefined Event Trace Levels

Event Type	Predefined Event Trace Level		
	<code>functions</code>	<code>functions_partial_stalls</code>	<code>functions_all_stalls</code>
Functions Calls/Returns	Captured	Captured	Captured
Stream Stalls	NA	Captured	Captured
Cascade Stalls	NA	Captured	Captured
Lock Stalls	NA	Captured	Captured
Memory Stalls	NA	NA	Captured

- Functions Calls/Returns:** Event generated when kernel functions are being invoked and returned.
- Stream Stalls:** Event generated when core gets stalled. This can be due to either NO data at Input or backpressure on the stream output from the core.
- Cascade Stalls:** Event generated when core gets stalled. This can be due to either NO data at Input or due to backpressure on the stream output from the core.
- Lock Stalls:** Event generated when the core gets stalled due to the lock already being acquired.
- Memory Stalls:** Event generated when core gets stalled due to a memory conflict.

Running the Design in Hardware and Capturing Trace Data at Run-Time

Xilinx® Runtime (XRT) and Xilinx Software Debugger (XSDB) are two ways to run the design on the Arm® processor in hardware and capture trace data at run time. XRT is supported on the Linux platform, whereas XSDB is supported on both bare metal and Linux platforms. The table below highlights the features supported in both flows.

Table 71: XRT versus XSDB

	Bare Metal	PetaLinux	Bandwidth per Trace Stream (bits/s)	Ease of Use
XSDB	Yes	Yes	PL clock-rate * 64	This flow involves manually setting up the environment, sourcing scripts and running the event trace flow. (Details can be found in the XSDB section below)
XRT	No	Yes	PL clock-rate * 64	This flow is easier than XSDB, because event trace can be set up in the <code>xrt.ini</code> file.

XSDB Flow

The XSDB flow is as follows:

1. Set up `xsdb` as described in the following steps to connect to the device hardware.

When running the application, the trace data is stored in DDR memory by the debugging and profiling IP. To capture and evaluate this data, you must connect to the hardware device using `xsdb`. This command is typically used to program the device and debug bare-metal applications. Connect your system to the hardware platform or device over JTAG, launch the `xsdb` command in a command shell, and run the following sequence of commands:

```
xsdb% connect
xsdb% ta
xsdb% ta 1
xsdb% source $::env(XILINX_VITIS)/scripts/vitis/util/aie_trace.tcl
xsdb% aietrace start -graphs mygraph -config-level functions_all_stalls -
work-dir ./Work -link-summary $PROJECT/xclbin.link_summary -base-address
0x900000000 -depth 0x800000

# Execute the PS host application (.elf) on Linux
## After the application completes processing.
xsdb% aietrace stop
```

where:

- **connect:** Launches the `hw_server` and connects `xsdb` to the device.
- **source \$::env(XILINX_VITIS)/scripts/vitis/util/aie_trace.tcl:** Sources the Tcl trace command to set up the `xsdb` environment.
- **aietrace start PROJECT/xclbin.link_summary -base-address 0x900000000 -depth 0x800000:** Initializes the DPA IP to begin capturing trace data. The `-config-level` specifies functions, functions_partial_stalls, or functions_all_stalls of event trace level to be captured. The values `-base-address 0x900000000 -depth 0x800000` specify the starting address to write trace data into the AI Engine and the amount of data to store.



IMPORTANT! The DDR memory address used in `-base-address 0x900000000` must be a high address to limit any chance of running into memory conflicts with the OS on the `xilinx_vck190_base_202120_1` platform or the application. For a custom platform, make sure you know how much DDR memory is being used and plan accordingly.

- `aietrace stop`: Instructs the DPA IP to offload the trace event data from the DDR memory. This command must wait until after the application completes. The data is written to the `event_trace<N>.txt` file in the current working directory from where `xsdb` was launched. An `aie_trace_profile.run_summary` file is also created. It can be opened in the Vitis analyzer as explained in [Viewing the Run Summary in the Vitis Analyzer](#).



TIP: If you do not remove the `event_trace<N>.txt` when running the graph again, the old files will be overwritten by the new run results.

2. Run the design on hardware to trace hardware events.
3. Offload the captured trace data.
4. Launch the Vitis analyzer to import and analyze data with this command.

```
vitis_analyzer aie_trace_profile.run_summary
```

XRT Flow

The XRT flow is as follows:

1. Burn the generated `sd_card.img` to the physical SD card.
2. Create the `xrt.ini` file in the `sd_card` folder as described in this section to enable `xrt` flow.

An example `xrt.ini` file is shown in the following.

```
[Debug]
aie_trace=true
aie_trace_buffer_size=100M
aie_trace_metrics = functions_all_stalls
```

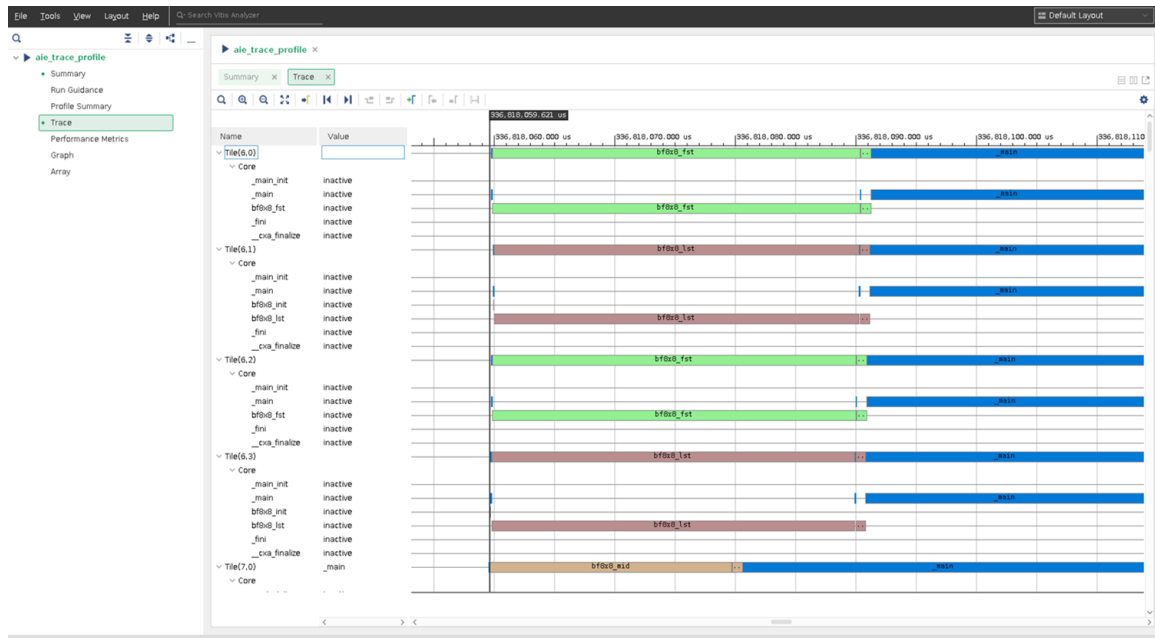
3. Run the design on hardware to trace hardware events.
4. Copy the captured trace data from the `sd_card` folder to your design at same level as the design `Work` directory. The trace data is generated in the same location as the host application on the SD card. They are `xrt.run_summary`, `aie_event_runtime_config.json`, and `aie_trace-N.txt`.
5. Use the Vitis analyzer to import and analyze data with this command.

```
vitis_analyzer xrt.run_summary
```

Viewing and Analyzing the Trace Data Using Vitis Analyzer

The Vitis analyzer must be used to view and analyze trace data. After the trace data has been captured either using XRT or XSDB, you should have all the data needed to open the Event Trace view in the Vitis analyzer.

Figure 48: Event Trace in Vitis Analyzer



Open the run summary file with the Vitis analyzer to view event trace data. An example of the XSDB flow is as follows:

```
vitis_analyzer ./aie_trace_profile.run_summary
```

An example of the XRT flow is as follows:

```
vitis_analyzer ./xrt.run_summary
```

Limitations

- Due to limited resources, overruns can be seen from event trace. Refer to [Using Multiple Event Trace Streams](#) to configure the number of trace streams to minimize the overruns issue.
- For detailed event trace information, the `--xlopt=0` option must be specified in compile. If omitted, the default setting `--xlopt=1` is used which might cause functions to be in-lined and limits debug capability.
- Run forever applications are supported by the XSDB flow only.

- Host code is required to invoke `graph.end()` to ensure that the XRT flow completes properly.

Using Multiple Event Trace Streams

As AI Engine designs grow larger, tracking the events produced while running the design can be useful to identify performance bottlenecks as well as understanding how the overall AI Engine is operating for the design. Of course, with larger designs more and more events will be produced causing a bottleneck of the events being recorded by the trace IP being used. To capture all this data effectively, and quickly, you should consider instantiating multiple event trace streams. These streams will spread out the event data coming from the AI Engine, letting it store them correctly and in a timely manner.

To increase the trace streams in a design, use the `aiecompiler --num-trace-streams` option, which can have a value in the range of 1 to 16. The following table provides guidance on the number of trace streams to use, depending on the size of the design.

Table 72: Number of Event Trace Streams Methodology

Number of AI Engines	Recommended Number of Streams
Less than 10	1
Between 10 and 20	2
Between 20 and 40	4
Between 40 and 80	8
Larger than 80	16

Notes:

1. It is recommended to only use up to 16 trace streams due to the resource utilization impact on the PL and DMA channel resources.

After the change to the AI Engine compiler option, recompile and re-link the XCLBIN file and `libadf.a` using the Vitis compiler with a `config` file as described in [Linking the System](#).

```
v++ -l --config system.cfg ...
```

Troubleshooting Event Trace in Hardware

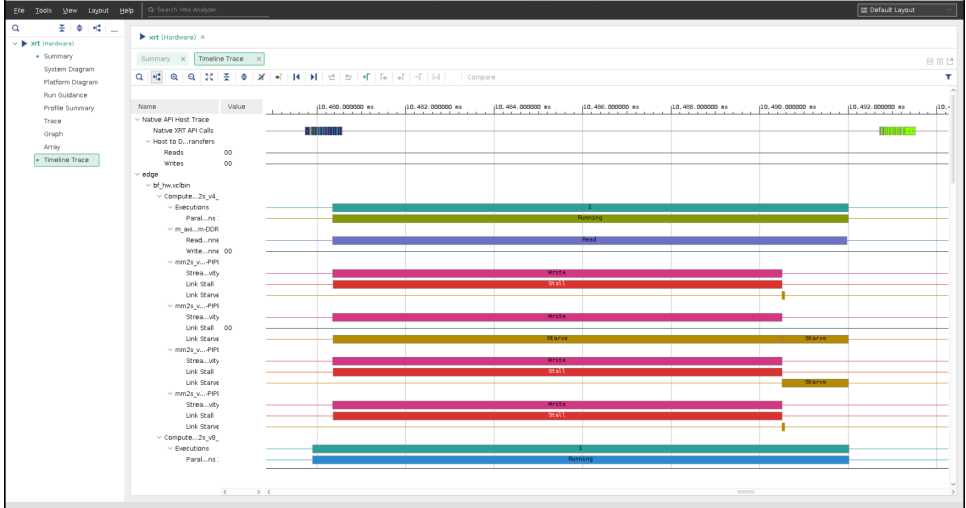
Table 73: Troubleshooting Event Trace in Hardware

Issue-	Resolution
Trace packets are being dropped on certain streams. (This can be determined by observing black bars on event trace streams in the Vitis Analyzer GUI.)	<p>Increase the number of trace streams coming out of AI Engine using the option below.</p> <pre>aiecompiler --num-trace-streams=<N></pre>

Table 73: Troubleshooting Event Trace in Hardware (cont'd)

Issue-	Resolution
Trace packets are being dropped on certain streams and the number of trace stream is 16 in a PLIO event trace flow. (This can be determined by observing black bars on event trace streams in the Vitis Analyzer GUI.)	<p>Use advanced options in v++ <code>system.cfg</code> file to set the aie trace clock frequency.</p> <pre>[advanced]param=compiler.aieTraceClockSelect=fastest</pre> <p>where <code>compiler.aieTraceClockSelect</code> is the trace clock frequency on the PL trace kernel. The exact frequency setting is dependent on the platform used. Using the "fastest" option could impact PL timing results.</p>
Kernel function names are not visible from trace.	<p>This could be because the kernel is inlined either by the compiler or via an attribute. Apply <code>--xlopt=0</code> to compile the design or specify kernel functions with <code>--attribute__((noinline))</code> attribute to disable kernel in-lining.</p>
Kernels start time are off by more than one hundred cycles.	<p>Apply <code>--broadcast-enable-core=true</code> option to the compiler to ensure the design starts all kernels in a few clock cycles range.</p>
Tool issues a warning message indicating trace buffer is full.	<p>Increase event trace buffer size.</p> <p>For the XSDB flow, specify larger value with the <code>-depth</code> option in the AI Engine trace start command.</p> <pre>%xsdb aietrace start -graphs dut -config-level functions_all_stalls -work-dir ./Work -link-summary ./bf_hw.xclbin.xclbin.link-summary -base-address 0x900000000 -depth 0x8000000</pre> <p>For the XRT flow, update <code>xrt.ini</code> file and <code>aie_trace_buffer_size</code> line.</p> <pre>[Debug] aie_trace=true aie_trace_buffer_size=100M aie_trace_metrics = functions_all_stalls</pre> <p>Note: The following warning message is issued by both XSDB and XRT when the trace buffer is full.</p> <p>Warning message: AI Engine Trace Buffer size is full, Device trace could be incomplete.</p>

Table 73: Troubleshooting Event Trace in Hardware (cont'd)

Issue-	Resolution
Need to know if the PL is sending or receiving data to/from the AI Engine as expected.	<p>Add monitors on PL kernels and their memory-mapped AXI4 masters.</p> <pre>v++ -l --profile_kernel <data:[kernel_name all]:[compute_unit_name all]:[interface_name all](:[counters all])><[stall exec]:[kernel_name all]:[compute_unit_name all](:[counters all])></pre> <p>For example, to monitor every kernel master, add the following:</p> <pre>v++ -l --profile_kernel data:all:all:all</pre> <p>Running on hardware, add the following lines in the <code>xrt.ini</code> file.</p> <pre>[Debug] profile=true native_xrt_trace=true data_transfer_trace=coarse</pre> <p>After running the application on hardware, <code>device_trace_0.csv</code>, <code>native_trace.csv</code>, <code>summary.csv</code>, and <code>xrt.run_summary</code> files are generated on <code>sd_card</code>. Copy those files to where your project is and at same level as the project's Work directory. Issue the command <code>vitis_analyzer xrt.run_summary</code> and select the timeline Trace view to inspect PL trace.</p> 

Programming the PS Host Application

In [Creating a Data Flow Graph \(Including Kernels\)](#) the discussion was centered around a very simple AI Engine graph application. The top-level application initialized the graph, ran the graph, and ended the graph. However, for actual AI Engine graph applications the host code must do much more than those simple tasks. The top-level PS application running on the Cortex®-A72, controls the graph and PL kernels: manage data inputs to the graph, handle data outputs from the graph, and control any PL kernels working with the graph.

In addition, AI Engine graph applications can be run on Linux operating systems or on bare-metal systems. The requirements for programming in these two systems are significantly different as outlined in the following topics. Xilinx provides drivers used by the API calls in the host program to control the graph and PL kernels based on the operating system. In Linux this is provided by the `libadf_api_xrt` library, in bare-metal the AI Engine kernels are controlled using the graph APIs, and PL kernels are controlled using `libUIO` driver calls.

Preventing Multiple Graph Executions

In cases where your graph is implemented in a PS-based host application, you must define a conditional pragma (`#ifdef`) for your `graph.cpp` code to ensure the graph is only initialized once, or run only once. The following example code is the simple application defined in [Creating a Data Flow Graph \(Including Kernels\)](#) with the additional guard macro `__AIESIM__` and `__X86SIM__`.

```
#include "project.h"

simpleGraph mygraph;
simulation::platform<1,1> platform("input.txt","output.txt");
connect<> net0(platform.src[0], mygraph.in);
connect<> net1(mygraph.out, platform.sink[0]);

#if defined(__AIESIM__) || defined(__X86SIM__)

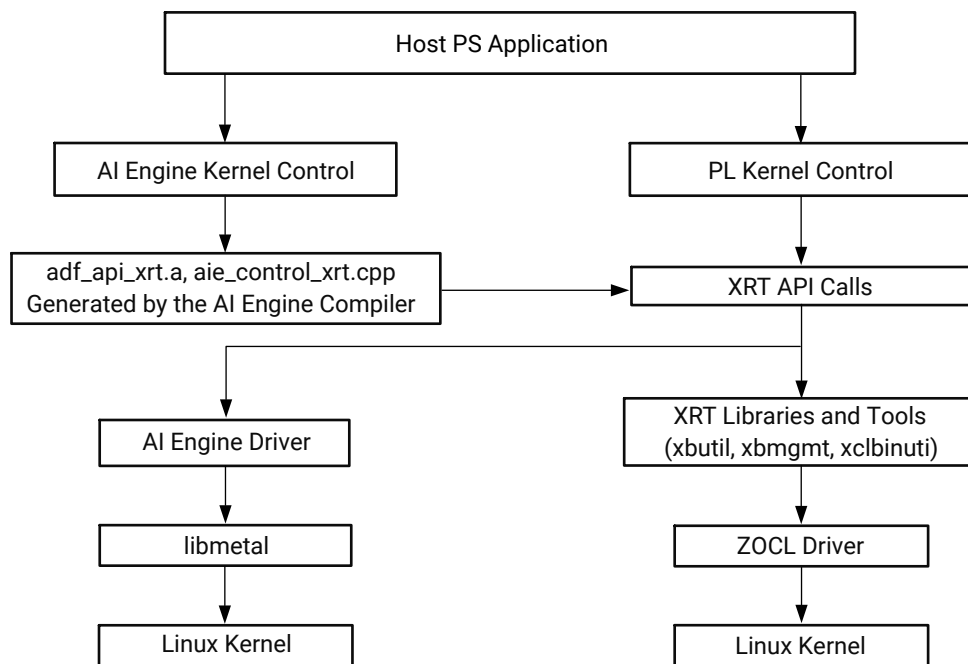
int main(void) {
    mygraph.init();
    mygraph.run(<number_of_iterations>);
    mygraph.end();
    return 0;
}
#endif
```

This conditional directive compiles the application only for use with the AI Engine simulator. It prevents the graph from being initialized or run multiple times, from both the graph and PS host application. The directive lets the `graph.cpp` run in simulation or in hardware emulation of the system design, which also runs in the AI Engine simulator and the x86 simulator. However when running in hardware, the graph is initialized and run from the PS application, rather than from the `graph.cpp`.

Host Programming on Linux

In Linux operating systems, the ADF API controls the AI Engine graph. The Xilinx Runtime (XRT) API is used to control PL kernels. The Xilinx Runtime (XRT) API can also be used to control the AI Engine graph. The following figure shows the APIs and drivers required in this system.

Figure 49: AI Engine XRT Software Stack



X24068-100920

Controlling the AI Engine Graph with the ADF API

ADF APIs are used to control graph execution in the top-level application, or host code, as described in [Chapter 3: Introduction to AI Engine Programming](#). For example, the following code is for a synchronous update of run-time parameters for AI Engine kernels in the graph:

```
// ADF API:run and update graph parameters (RTP)
gr.run(4);
gr.update(gr.trigger,10);
gr.update(gr.trigger,10);
gr.update(gr.trigger,100);
gr.update(gr.trigger,100);
gr.wait();
```



TIP: The use of `graph.end()` terminates the graph. It will not recover after `end()` has been called. Instead, you can use `graph.wait()` to wait for runs to be completed.

In the host application (`host.cpp`), the `graph.update()` function is called to update the RTPs, and `graph.run()` is called to launch the AI Engine kernels in the graph. In hardware emulation and hardware flows, the ADF API is calling the XRT API, and `adf::registerXRT()` is used to manage the relationship between them.



IMPORTANT! `adf::registerXRT()` must be called before any ADF API control or interaction with the graph.

The following is example code showing the RTP update and execution by the ADF API.

```
// update graph parameters (RTP) & run
adf::registerXRT(dhdl, uuid);
gr.update(gr.size, 1024);//update RTP
gr.run(16);//start AIE kernel
gr.wait();
```

In the preceding example, `gr.run(16)` specifies a run of 16 iterations.

In `graph.wait()`, the application waits for the AI Engine kernels to complete.

The code example shows that `adf::registerXRT()` requires the device handle (`dhdl`) and UUID of the XCLBIN image. They can be obtained using the XRT APIs:

```
auto dhdl = xrtDeviceOpen(0);//device index=0
xrtDeviceLoadXclbinFile(dhdl,xclbinFilename);
xuid_t uuid;
xrtDeviceGetXclbinUUID(dhdl, uuid);
```

Controlling the PL Kernel with the XRT API

Xilinx provides an OpenSource XRT API for controlling the execution of PL kernels when programming the host code for Linux.

The execution model for the XRT API controlling PL kernels is as follows:

1. Get device handle and load the XCLBIN. Get the `uuid` as needed.
2. Allocate buffer objects and map to host memory. Process and transfer data from host memory to device memory.
3. Get kernel and run handles, set arguments for kernels, and launch kernels.
4. Wait for kernel completion.
5. Transfer data from global memory in the device back to host memory.
6. Host code continues processing using the new data in the host memory.

When using the native XRT API, the host application looks like the following.

```
1.// Open device, load xclbin, and get uuid
auto dhdl = xrtDeviceOpen(0); //device index=0

xrtDeviceLoadXclbinFile(dhdl,xclbinFilename);
xuid_t uuid;
xrtDeviceGetXclbinUUID(dhdl, uuid);

2. Allocate output buffer objects and map to host memory

xrtBufferHandle out_bohdl = xrtBOAlloc(dhdl, output_size_in_bytes, 0, /
*BANK= */0);
std::complex<short> *host_out = (std::complex<short>*)xrtBOMap(out_bohdl);

3. Get kernel and run handles, set arguments for kernel, and launch kernel.
xrtKernelHandle s2mm_khdl = xrtPLKernelOpen(dhdl, top->m_header.uuid,
"s2mm"); // Open kernel handle
xrtRunHandle s2mm_rhdl = xrtRunOpen(s2mm_khdl);
xrtRunSetArg(s2mm_rhdl, 0, out_bohdl); // set kernel arg
xrtRunSetArg(s2mm_rhdl, 2, OUTPUT_SIZE); // set kernel arg
xrtRunStart(s2mm_rhdl); //launch s2mm kernel

// ADF API:run, update graph parameters (RTP) and so on
.....

4. Wait for kernel completion.
auto state = xrtRunWait(s2mm_rhdl);

5. Sync output device buffer objects to host memory.

xrtBOSync(out_bohdl, XCL_BO_SYNC_BO_FROM_DEVICE , output_size_in_bytes,/
*OFFSET= */ 0);

//6. post-processing on host memory - "host_out"
```

After post-processing the data, release the allocated objects:

```
graph.end();
xrtRunClose(s2mm_rhdl);
xrtKernelClose(s2mm_khdl);

xrtBOFree(out_bohdl);
xrtDeviceClose(dhdl);
```



TIP: After `graph.end()`, the AI Engine kernels will not recover again. To run the host multiple times, you can comment out `graph.end()` if the host does not depend on `graph.end()` for synchronization purpose, or replace `graph.end()` with `graph.wait()` to do synchronization.

Controlling the AI Engine Graph with the XRT C API

A Xilinx provided OpenSource XRT API can also be used for controlling execution of AI Engine graph when programming the host code for Linux. To control the AI Engine graph, XRT provides APIs through the header file `experimental/xrt_graph.h`.



TIP: The header file `experimental/xrt_graph.h` is included in the header file `experimental/xrt_kernel.h`. So, including `experimental/xrt_kernel.h` is sufficient to use the XRT API to control the AI Engine graph.

XRT graph APIs contain both a C and C++ version. Example code to control the AI Engine graph using the XRT C API is as follows:

```
int narrow_filter[12] = {180, 89, -80, -391, -720, -834, -478, 505, 2063,
3896, 5535, 6504};
int wide_filter[12] = {-21, -249, 319, -78, -511, 977, -610, -844,
2574, -2754, -1066, 18539};
xrtGraphOpen(dhdl, top->m_header.uuid, "gr");
if(!ghdl){
int size=1024;
xrtGraphUpdateRTP(ghdl, "gr.fir24.in[1]",
(char*)narrow_filter, 12*sizeof(int));
xrtGraphRun(ghdl, 16);
xrtGraphWait(ghdl, 0);
xrtGraphUpdateRTP(ghdl, "gr.fir24.in[1]",
(char*)wide_filter, 12*sizeof(int));
xrtGraphRun(ghdl, 16);
.....
xrtGraphEnd(ghdl, 0);
xrtGraphClose(ghdl);
```



TIP: The file `Work/ps/c_rts/aie_control_xrt.cpp` contains information about graph, RTP, GMIO, and initialize configurations. You can find the information for these XRT APIs as required.

Controlling the AI Engine Graph with the XRT C++ API

As stated in the previous section, XRT provides C and C++ APIs through the header file `experimental/xrt_graph.h` to control the AI Engine graphs.

XRT provides class `graph` in the name space `xrt` and its member functions to control the graph. Example code to control the AI Engine graph using the XRT C++ API is as follows:

```
using namespace adf;
// Open xclbin
auto device = xrt::device(0); //device index=0
auto uuid = device.load_xclbin(xclbinFilename);
auto dhdl = xrtDeviceOpenFromXcl(device);
...
int coeffs_readback[12];
int narrow_filter[12] = {180, 89, -80, -391, -720, -834, -478, 505, 2063,
3896, 5535, 6504};
int wide_filter[12] = {-21, -249, 319, -78, -511, 977, -610, -844, 2574,
-2754, -1066, 18539};
auto ghdl=xrt::graph(device,uuid,"gr");
ghdl.update("gr.fir24.in[1]",narrow_filter);
ghdl.run(16);
ghdl.wait();
ghdl.read("gr.fir24.inout[0]",coeffs_readback);//Read after graph::wait.
RTP update effective
ghdl.update("gr.fir24.in[1]",wide_filter);
ghdl.run(16);
ghdl.read("gr.fir24.inout[0]", coeffs_readback);//Async read
ghdl.end();
```

Note: The file `Work/ps/c_rts/aie_control_xrt.cpp` contains information about the graph, RTP, GMIO, and initialize configurations. For example in the code above, you can find the RTP port name in the RTP Configurations section in `aie_control_xrt.cpp` called `gr.fir24.in[1]`.

Multi-Process and Multi-Thread Support for Controlling the AI Engine Graph

Xilinx® XRT APIs provide multi-process support for controlling the AI Engine array and graphs. It supports three operating modes on AI Engine array and graphs.

- **Exclusive Mode:** Can fully access the AI Engine array or graph. No other process can access it.
- **Primary Mode:** Can fully access the AI Engine array or graph. Other processes can get non-destructive access to the AI Engine array or graph.
- **Shared Mode:** Can only get non-destructive access to the AI Engine array or graph.

Xilinx XRT provides the following APIs for opening the AI Engine array in three modes.

- `xrtAIEDeviceOpenExclusive` (Exclusive mode)
- `xrtAIEDeviceOpen` (Primary mode)

- `xrtAIEDeviceOpenShared` (Shared mode)

Note: If the application does not call `xrtAIEDeviceOpen*` to obtain device handle, as a default, it will try to acquire primary context while trying to access the AI Engine array through XRT APIs.

Xilinx XRT also provides the following APIs for opening the graph after the AI Engine array is opened.

- `xrtGraphOpenExclusive` (Exclusive mode)
- `xrtGraphOpen` (Primary mode)
- `xrtGraphOpenShared` (Shared mode)

The following APIs on the AI Engine array in exclusive and primary modes allow loading, resetting, and GMIO data transferring.

- `xrtDeviceLoadXclbin`
- `xrtAIEResetArray`
- `xrtAIESyncBO`
- `xrtDeviceClose`

The following APIs on the AI Engine array in shared modes allow nondestructive operations.

- `xrtDeviceLoadXclbin`: Read metadata from xclbin.
- `xrtDeviceClose`

The allowed APIs on the AI Engine graph in exclusive and primary modes include the following.

- `xrtGraphRun`
- `xrtGraphWait`
- `xrtGraphEnd`
- `xrtGraphUpdateRTP`
- `xrtGraphReadRTP`
- `xrtGraphTimeStamp`
- `xrtGraphClose`

The allowed APIs on the AI Engine array and graphs in shared mode include the following.

- `xrtGraphUpdateRTP` (asynchronous mode)
- `xrtGraphReadRTP`
- `xrtGraphTimeStamp`
- `xrtGraphClose`

The following rules apply for the AI Engine array multi-process support.

- Only one process can open AI Engine array in the exclusive mode. If the AI Engine array is opened in exclusive mode, it cannot be opened again in any mode in the same process or other processes.
- Only one process can open AI Engine array in the primary mode. If the AI Engine array is opened in primary mode, it cannot be opened again in exclusive mode or primary mode, but it can be opened in shared mode again.

The following rules apply for the AI Engine graph multi-process support.

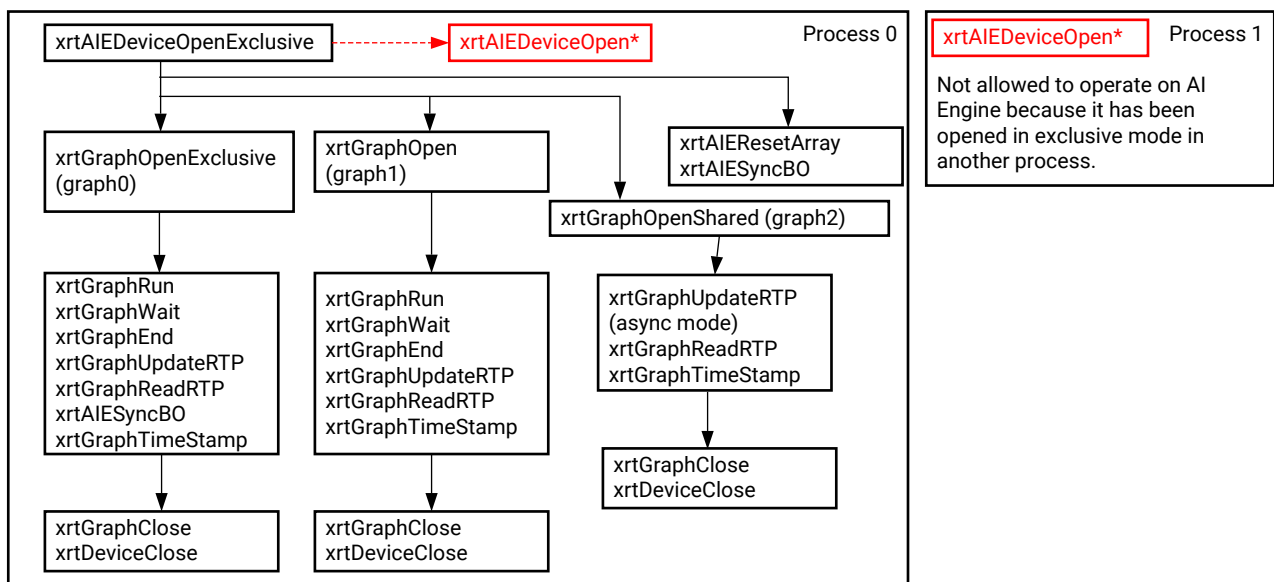
- Graph is not supposed to be opened multiple times in a process.
- If an AI Engine graph is opened in the exclusive mode, the graph cannot be opened in any mode again.
- There can be multiple processes to open graph in the shared mode, but the graph is allowed only one process in the primary mode.

Graph is to be closed before closing the array. After all open AI Engine arrays and graphs are closed, the AI Engine arrays and graphs can be opened again, and the preceding rules apply again.

Note: When a graph or AI Engine array is closed, only the opened context is closed. It does not destroy hardware in the AI Engine array.

The following figure summarizes the multi-process support in exclusive mode (black: supported, red: not supported, "***": any mode).

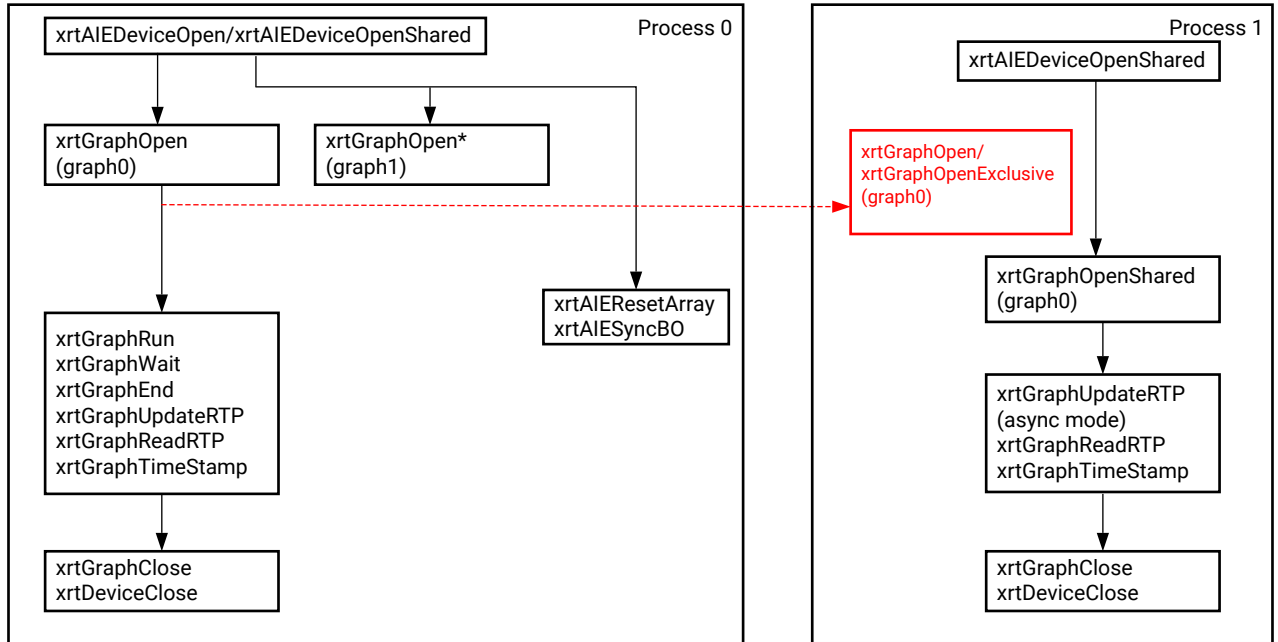
Figure 50: Multi-Process Support in Exclusive Mode



X25364-091421

The following figure summarizes the multi-process support in primary and shared modes (black: supported, red: not supported).

Figure 51: Multi-Process Support in Primary and Shared Modes



X25365-091321

It is recommended that multiple threads use the same model as multiple processes. However, because the AI Engine device handle and graph handle are sharable between threads, it is legal to use the same device handle or graph handle between threads. The host application is responsible for synchronizing the AI Engine array state and graph state between threads, especially when multiple threads are the exclusive or primary owner of the AI Engine array or graph.

A sample code to use multi-process is as follows.

```
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include "adf/adf_api/XRTConfig.h"
#include "experimental/xrt_kernel.h"

#include "graph.cpp"

//8192 matches 32 iterations of graph::run
#define OUTPUT_SIZE 8192
int value1[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
int value2[16] = {-1,-2,-3,-4,-5,-6,-7,-8,-9,-10,-11,-12,-13,-14,-15,-16};

using namespace adf;

int run(int argc, char* argv[],int id){
    std::cout<<"Child process "<<id<<" start"<<std::endl;
```

```
//TARGET_DEVICE macro needs to be passed from gcc command line
if(argc != 2) {
    std::cout << "Usage: " << argv[0] << " <xclbin>" << std::endl;
    return EXIT_FAILURE;
}
char* xclbinFilename = argv[1];
std::string graph_name=std::string("gr["+std::to_string(id)+"]");
std::string rtp_inout_name=std::string("gr["+std::to_string(id)
+std::string("].k.inout[0]");

int ret;
int value_readback[16]={0};
if(fork()==0){//child child process
    xrtDeviceHandle dhdl2=xrtAIEDeviceOpenShared(0);
    ret=xrtDeviceLoadXclbinFile(dhdl2,xclbinFilename);
    if(ret){
        printf("child child Xclbin Load fail\n");
    }
    if(!dhdl2){
        std::cout<<"child child device open error"<<std::endl;
        return 1;
    }else{
        std::cout<<"child child device open pass"<<std::endl;
    }
    xuid_t uuid2;
    ret=xrtDeviceGetXclbinUUID(dhdl2, uuid2);
    if(ret){
        std::cout<<"child child get xclbin uuid error"<<std::endl;
        return 1;
    }else{
        std::cout<<"child child get xclbin uuid pass"<<std::endl;
    }
    auto ghdl2=xrtGraphOpenShared(dhdl2,uuid2,graph_name.data());
    if(!ghdl2){
        std::cout<<"child child graph open error"<<std::endl;
        return 1;
    }else{
        std::cout<<"child child graph open pass"<<std::endl;
    }

    ret=xrtGraphReadRTP(ghdl2, rtp_inout_name.data(),
(char*)value_readback, 16*sizeof(int));
    if(ret){
        std::cout<<"child child Graph RTP read fail"<<std::endl;
        return 1;
    }
    std::cout<<"Add value read back are:";
    for(int i=0;i<16;i++){
        std::cout<<value_readback[i]<<","<<"\t";
    }
    std::cout<<std::endl;
    xrtGraphClose(ghdl2);
    xrtDeviceClose(dhdl2);
    std::cout<<"child child process exit"<<std::endl;
    exit(0);
}

xrtDeviceHandle dhdl=xrtAIEDeviceOpen(0);
ret=xrtDeviceLoadXclbinFile(dhdl,xclbinFilename);
if(ret){
    printf("Xclbin Load fail\n");
}
```

```

        xuid_t uuid;
        xrtDeviceGetXclbinUUID(dhdl, uuid);

        auto ghdl=xrtGraphOpen(dhdl,uuid,graph_name.data());
        if(!ghdl){
            std::cout << "Graph Open error" << std::endl;
        }else{
            std::cout << "Graph Open ok" <<std::endl;
        }
        std::string rtp_in_name=std::string("gr[")+std::to_string(id)
+std::string("].k.in[1]");
        ret=xrtGraphUpdateRTP(ghdl,rtp_in_name.data(),
(char*)value1,16*sizeof(int));
        if(ret){
            std::cout<<"Graph RTP update fail"<<std::endl;;
            return 1;
        }
        xrtGraphRun(ghdl,16);

        xrtGraphWait(ghdl,0);
        std::cout<<"Graph wait done"<<std::endl;

        //second run
        ret=xrtGraphUpdateRTP(ghdl,rtp_in_name.data(),
(char*)value2,16*sizeof(int));
        if(ret!=0){
            std::cout<<"Graph RTP update fail"<<std::endl;
            return 1;
        }else{
            std::cout<<"Graph RTP update pass"<<std::endl;
        }
        xrtGraphRun(ghdl,16);

        while(wait(NULL)>0){//Wait for child child process
        }

        ret=xrtGraphWait(ghdl,0);
        if(ret){
            std::cout << "Graph wait error" << std::endl;
        }else{
            std::cout<<"Graph done"<<std::endl;
        }
        xrtGraphClose(ghdl);
        xrtDeviceClose(dhdl);
        std::cout<<"Child process:"<<id<<" done"<<std::endl;
        return 0;
    }

    int main(int argc, char* argv[])
    {
        try {
            for(int i=0;i<GRAPH_NUM;i++){
                if(fork()==0){//child
                    auto match = run(argc, argv,i);
                    std::cout << "TEST child " <<i<< (match ? " FAILED" : "
PASSED") << "\n";
                    return (match ? EXIT_FAILURE : EXIT_SUCCESS);
                }else{
                    size_t output_size_in_bytes = OUTPUT_SIZE * sizeof(int);
                    //TARGET_DEVICE macro needs to be passed from gcc command
line
                    if(argc != 2) {
                        std::cout << "Usage: " << argv[0] <<" <xclbin>" <<

```

```
std::endl;
        return EXIT_FAILURE;
    }
    char* xclbinFilename = argv[1];

    int ret;
    // Open xclbin
    auto device = xrt::device(0); //device index=0
    auto uuid = device.load_xclbin(xclbinFilename);

    // s2mm & data_generator kernel handle
    std::string s2mm_kernel_name=std::string("s2mm:{s2mm_")
+std::to_string(i+1)+std::string("}");
    xrt::kernel s2mm = xrt::kernel(device, uuid,
s2mm_kernel_name.data());
    std::string
data_generator_kernel_name=std::string("data_generator:{data_generator_")
+std::to_string(i+1)+std::string("}");
    xrt::kernel data_generator = xrt::kernel(device, uuid,
data_generator_kernel_name.data());

    // output memory
    auto out_bo=xrt::bo(device,
output_size_in_bytes,s2mm.group_id(0));
    auto host_out=out_bo.map<int*>();
    auto s2mm_run = s2mm(out_bo, nullptr, OUTPUT_SIZE);//1st
run for s2mm has started
    auto data_generator_run = data_generator(nullptr,
OUTPUT_SIZE);

    // wait for s2mm done
    std::cout<<"Waiting s2mm to complete"<<std::endl;
    auto state = s2mm_run.wait();
    std::cout << "s2mm "<<" completed with status(" << state <<
")"<<std::endl;
    out_bo.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

    int match = 0;
    int counter=0;
    for (int i = 0; i < OUTPUT_SIZE/2/16; i++) {
        for(int j=0;j<16;j++){
            if(host_out[i*16+j]!=counter+value1[j]){
                std::cout<<"ERROR: num="<<i*16+j<<"
out="<<host_out[i*16+j]<<std::endl;
                match=1;
                break;
            }
            counter++;
        }
    }
    for(int i=OUTPUT_SIZE/2/16;i<OUTPUT_SIZE/16;i++){
        for(int j=0;j<16;j++){
            if(host_out[i*16+j]!=counter+value2[j]){
                std::cout<<"ERROR: num="<<i*16+j<<"
out="<<host_out[i*16+j]<<std::endl;
                match=1;
                break;
            }
            counter++;
        }
    }

    std::cout << "TEST " <<i<< (match ? " FAILED" : " PASSED")
```

```
<< "\n";
        while(wait(NULL)>0){//Wait for all child process
        }
        std::cout<<"all done"<<std::endl;
        return (match ? EXIT_FAILURE : EXIT_SUCCESS);
    }
}

catch (std::exception const& e) {
    std::cout << "Exception: " << e.what() << "\n";
    std::cout << "FAILED TEST\n";
    return 1;
}
}
```

C++ XRT API for Multi-Process Support

The XRT C++ API extends `xrt::aie::device` class to support access mode in https://github.com/Xilinx/XRT/blob/master/src/runtime_src/core/include/xrt/xrt_aie.h as shown in the following code.

```
namespace xrt { namespace aie {

/**
 * @enum access_mode - AIE array access mode
 *
 * @var exclusive
 * Exclusive access to AIE array. No other process will have
 * access to the AIE array.
 * @var primary
 * Primary access to AIE array provides same capabilities as exclusive
 * access, but other processes will be allowed shared access as well.
 * @var shared
 * Shared non destructive access to AIE array, a limited number of APIs
 * can be called.
 * @var none
 * For internal use only, to be removed.
 *
 * By default the AIE array is opened in primary access mode.
 */
enum class access_mode : uint8_t { exclusive = 0, primary = 1, shared = 2,
none = 3 };

class device : public xrt::device
{
public:
    using access_mode = xrt::aie::access_mode;

    /**
     * device() - Construct device with specified access mode
     *
     * @param args
     * Arguments to construct a device (xrt_device.h).
     * @param am
     * Open the AIE device is specified access mode (default primary)
     *
     * The default access mode is primary.
     */
    template
    device(ArgType&& arg, access_mode am = access_mode::primary)
```

```

        : xrt::device(std::forward(arg))
    {
        open_context(am);
    }
    ...
};

}} // namespace aie, xrt

```

The XRT C++ API extends `xrt::graph` class to support access mode in https://github.com/Xilinx/XRT/blob/master/src/runtime_src/core/include/xrt/xrt_graph.h as shown in the following code.

```

namespace xrt {

class graph
{
public:
    /**
     * @enum access_mode - graph access mode
     *
     * @var exclusive
     * Exclusive access to graph and all graph APIs. No other process
     * will have access to the graph.
     * @var primary
     * Primary access to graph provides same capabilities as exclusive
     * access, but other processes will be allowed shared access as well.
     * @var shared
     * Shared none destructive access to graph, a limited number of APIs
     * can be called.
     *
     * By default a graph is opened in primary access mode.
     */
    enum class access_mode : uint8_t { exclusive = 0, primary = 1, shared =
2 };

    /**
     * graph() - Constructor from a device, xclbin and graph name
     *
     * @param device
     * Device on which the graph should execute
     * @param xclbin_id
     * UUID of the xclbin with the graph
     * @param name
     * Name of graph to construct
     * @param am
     * Open the graph with specified access (default primary)
     */
    graph(const xrt::device& device, const xrt::uuid& xclbin_id, const
std::string& name,
        access_mode am = access_mode::primary);

    ...
};

} // namespace xrt

```

The corresponding C++ version of sample code is as follows.

```
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include "adf/adf_api/XRTConfig.h"
#include "experimental/xrt_aie.h"
#include "experimental/xrt_graph.h"
#include "experimental/xrt_kernel.h"

#include "graph.cpp"

//8192 matches 32 iterations of graph::run
#define OUTPUT_SIZE 8192
int value1[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
int value2[16] = {-1,-2,-3,-4,-5,-6,-7,-8,-9,-10,-11,-12,-13,-14,-15,-16};

using namespace adf;

int run(int argc, char* argv[],int id){
    std::cout<<"Child process "<<id<<" start"<<std::endl;

    //TARGET_DEVICE macro needs to be passed from gcc command line
    if(argc != 2) {
        std::cout << "Usage: " << argv[0] << " <xclbin>" << std::endl;
        return EXIT_FAILURE;
    }
    char* xclbinFilename = argv[1];
    std::string graph_name=std::string("gr["+std::to_string(id)+"]");
    std::string rtp_inout_name=std::string("gr["+std::to_string(id)
+std::string("].k.inout[0]");

    int ret;
    int value_readback[16]={0};
    if(fork()==0){//child child process
        xrt::aie::device device{0, xrt::aie::device::access_mode::shared};
        auto uuid = device.load_xclbin(xclbinFilename);
        xrt::graph graph{device, uuid, graph_name,
xrt::graph::access_mode::shared};

        graph.read(rtp_inout_name, value_readback);
        std::cout<<"Add value read back are:";
        for(int i=0;i<16;i++){
            std::cout<<value_readback[i]<<","<<t";
        }
        std::cout<<std::endl;
        std::cout<<"child child process exit"<<std::endl;
        exit(0);
    }

    xrt::aie::device device{0}; // default primary context
    auto uuid = device.load_xclbin(xclbinFilename);
    xrt::graph graph{device, uuid, graph_name}; // default primary context

    std::string rtp_in_name=std::string("gr["+std::to_string(id)
+std::string("].k.in[1]");
    graph.update(rtp_in_name, value1);
    graph.run(16); // 16 iterations

    graph.wait(0); // wait 0 => wait till graph is done
    std::cout<<"Graph wait done"<<std::endl;
```



```

//second run
graph.update(rtp_in_name.data(), value2);
graph.run(16); // 16 iterations;

while(wait(NULL)>0){//Wait for child child process
}

graph.wait(0); // wait 0 => wait till graph is done
std::cout<<"Child process:"<<i<<" done"<<std::endl;
return 0;
}

int main(int argc, char* argv[])
{
    try {
        for(int i=0;i<GRAPH_NUM;i++){
            if(fork()==0){//child
                auto match = run(argc, argv,i);
                std::cout << "TEST child " <<i<< (match ? " FAILED" : "
PASSED") << "\n";
                return (match ? EXIT_FAILURE : EXIT_SUCCESS);
            }else{
                size_t output_size_in_bytes = OUTPUT_SIZE * sizeof(int);
                //TARGET_DEVICE macro needs to be passed from gcc command
line
                if(argc != 2) {
                    std::cout << "Usage: " << argv[0] <<" <xclbin>" <<
std::endl;
                    return EXIT_FAILURE;
                }
                char* xclbinFilename = argv[1];

                int ret;
                // Open xclbin
                auto device = xrt::device(0); //device index=0
                auto uuid = device.load_xclbin(xclbinFilename);

                // s2mm & data_generator kernel handle
                std::string s2mm_kernel_name=std::string("s2mm:{s2mm_")
+std::to_string(i+1)+std::string("}");
                xrt::kernel s2mm = xrt::kernel(device, uuid,
s2mm_kernel_name.data());
                std::string
data_generator_kernel_name=std::string("data_generator:{data_generator_")
+std::to_string(i+1)+std::string("}");
                xrt::kernel data_generator = xrt::kernel(device, uuid,
data_generator_kernel_name.data());

                // output memory
                auto out_bo=xrt::bo(device,
output_size_in_bytes,s2mm.group_id(0));
                auto host_out=out_bo.map<int*>();
                auto s2mm_run = s2mm(out_bo, nullptr, OUTPUT_SIZE);//1st
run for s2mm has started
                auto data_generator_run = data_generator(nullptr,
OUTPUT_SIZE);

                // wait for s2mm done
                std::cout<<"Waiting s2mm to complete"<<std::endl;
                auto state = s2mm_run.wait();
                std::cout << "s2mm "<<" completed with status(" << state <<
")"<<std::endl;

```

```

        out_bo.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

        int match = 0;
        int counter=0;
        for (int i = 0; i < OUTPUT_SIZE/2/16; i++) {
            for(int j=0;j<16;j++){
                if(host_out[i*16+j]!=counter+value1[j]){
                    std::cout<<"ERROR: num="<<i*16+j<<"
out="<<host_out[i*16+j]<<std::endl;
                    match=1;
                    break;
                }
                counter++;
            }
        }
        for(int i=OUTPUT_SIZE/2/16;i<OUTPUT_SIZE/16;i++){
            for(int j=0;j<16;j++){
                if(host_out[i*16+j]!=counter+value2[j]){
                    std::cout<<"ERROR: num="<<i*16+j<<"
out="<<host_out[i*16+j]<<std::endl;
                    match=1;
                    break;
                }
                counter++;
            }
        }

        std::cout << "TEST " <<i<< (match ? " FAILED" : " PASSED")
<< "\n";

        while(wait(NULL)>0){//Wait for all child process
        }
        std::cout<<"all done"<<std::endl;
        return (match ? EXIT_FAILURE : EXIT_SUCCESS);
    }
}

catch (std::exception const& e) {
    std::cout << "Exception: " << e.what() << "\n";
    std::cout << "FAILED TEST\n";
    return 1;
}
}

```

Error Reporting Through the XRT API

XRT provides error reporting APIs. The error reporting APIs can be categorized into two types: synchronous and asynchronous APIs. Synchronous errors are errors that can be detected during the XRT run-time function call. It is POSIX-compliant. For example:

```

rval = xclSyncBO(devHandle, boHandle, XCL_BO_SYNC_BO_TO_DEVICE, size, offset); // Synchronous error captured by xclSyncBO call
if (rval != 0) {
    /* code to handle xclSyncBO fail */
}
// -EINVAL: Invalid arguments, e.g. invalid sync dir, invalid size or offset
// -ENOENT: No such file or directory, e.g. invalid bo handle
// -EOPNOTSUPP: Operation not supported, e.g. BO is not syncable
// -EBUSY: Device or resource busy, e.g. No available DMA channel
// -ENOMEM: Out of memory, e.g. No available free memory for sync BO
// -EIO: I/O error, e.g. DMA error
// "dmesg" or look at system log file (/var/log/syslog) might give you more information

```

An asynchronous error might not be related to the current XRT function call or the application that is running. Asynchronous errors are cached in driver subsystems and can be accessed by the user application through the asynchronous error reporting APIs. Cached errors are persistent until explicitly cleared. Persistent errors are not necessarily indicative of the current system state, for example, a board might have been reset and be functioning correctly while previously cached errors are still available. To avoid current state confusion, asynchronous errors have a timestamp attached indicating when the error occurred. The timestamp can be compared to, for example, the timestamp for last `xbutil` reset.

The errors cached by the driver contain a system error code and additional meta data as defined in `xrt_error_code.h`, which is shared between the user space and the kernel space.

The error code format for asynchronous errors is as shown here:

```
/**
 * xrtErrorCode layout
 *
 * This layout is internal to XRT (akin to a POSIX error code).
 *
 * The error code is populated by driver and consumed by XRT
 * implementation where it is translated into an actual error / info /
 * warning that is propagated to the end user.
 *
 * 63 - 48  47 - 40  39 - 32  31 - 24  23 - 16  15 - 0
 * -----
 * |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | xrtErrorNum
 * |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | xrtErrorDriver
 * |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | xrtErrorSeverity
 * |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | xrtErrorModule
 * |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | xrtErrorClass
 * |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | reserved
 *
 */
typedef uint64_t xrtErrorCode;
typedef uint64_t xrtErrorTime;

#define XRT_ERROR_NUM_MASK          0xFFFFFUL
#define XRT_ERROR_NUM_SHIFT        0
#define XRT_ERROR_DRIVER_MASK      0xFUL
#define XRT_ERROR_DRIVER_SHIFT     16
#define XRT_ERROR_SEVERITY_MASK    0xFUL
#define XRT_ERROR_SEVERITY_SHIFT   24
#define XRT_ERROR_MODULE_MASK      0xFUL
#define XRT_ERROR_MODULE_SHIFT     32
#define XRT_ERROR_CLASS_MASK       0xFUL
#define XRT_ERROR_CLASS_SHIFT      40

#define XRT_ERROR_CODE_BUILD(num, driver, severity, module, eclass) \
    (((num) & XRT_ERROR_NUM_MASK) << XRT_ERROR_NUM_SHIFT) | \
    (((driver) & XRT_ERROR_DRIVER_MASK) << XRT_ERROR_DRIVER_SHIFT) | \
    (((severity) & XRT_ERROR_SEVERITY_MASK) << XRT_ERROR_SEVERITY_SHIFT) | \
    (((module) & XRT_ERROR_MODULE_MASK) << XRT_ERROR_MODULE_SHIFT) | \
    (((eclass) & XRT_ERROR_CLASS_MASK) << XRT_ERROR_CLASS_SHIFT)

#define XRT_ERROR_NUM(code) (((code) >> XRT_ERROR_NUM_SHIFT) & \
    XRT_ERROR_NUM_MASK)
#define XRT_ERROR_DRIVER(code) (((code) >> XRT_ERROR_DRIVER_SHIFT) & \
    XRT_ERROR_DRIVER_MASK)
```

```
#define XRT_ERROR_SEVERITY(code) (((code) >> XRT_ERROR_SEVERITY_SHIFT) &
XRT_ERROR_SEVERITY_MASK)
#define XRT_ERROR_MODULE(code) (((code) >> XRT_ERROR_MODULE_SHIFT) &
XRT_ERROR_MODULE_MASK)
#define XRT_ERROR_CLASS(code) (((code) >> XRT_ERROR_CLASS_SHIFT) &
XRT_ERROR_CLASS_MASK)

/**
 * xrt_error_num - XRT specific error numbers
 */

enum xrtErrorNum {
XRT_ERROR_NUM_FIREWALL_TRIP = 1,
XRT_ERROR_NUM_TEMP_HIGH,
XRT_ERROR_NUM_AIE_SATURATION,
XRT_ERROR_NUM_AIE_FP,
XRT_ERROR_NUM_AIE_STREAM,
XRT_ERROR_NUM_AIE_ACCESS,
XRT_ERROR_NUM_AIE_BUS,
XRT_ERROR_NUM_AIE_INSTRUCTION,
XRT_ERROR_NUM_AIE_ECC,
XRT_ERROR_NUM_AIE_LOCK,
XRT_ERROR_NUM_AIE_DMA,
XRT_ERROR_NUM_AIE_MEM_PARITY,
XRT_ERROR_NUM_UNKNOWN
};

enum xrtErrorDriver {
XRT_ERROR_DRIVER_XOCL,
XRT_ERROR_DRIVER_XCLMGMT,
XRT_ERROR_DRIVER_ZOCL,
XRT_ERROR_DRIVER_AIE,
XRT_ERROR_DRIVER_UNKNOWN
};

enum xrtErrorSeverity {
XRT_ERROR_SEVERITY_EMERGENCY = 0,
XRT_ERROR_SEVERITY_ALERT,
XRT_ERROR_SEVERITY_CRITICAL,
XRT_ERROR_SEVERITY_ERROR,
XRT_ERROR_SEVERITY_WARNING,
XRT_ERROR_SEVERITY_NOTICE,
XRT_ERROR_SEVERITY_INFO,
XRT_ERROR_SEVERITY_DEBUG,
XRT_ERROR_SEVERITY_UNKNOWN
};

enum xrtErrorModule {
XRT_ERROR_MODULE_FIREWALL = 0,
XRT_ERROR_MODULE_CMC,
XRT_ERROR_MODULE_AIE_CORE,
XRT_ERROR_MODULE_AIE_MEMORY,
XRT_ERROR_MODULE_AIE_SHIM,
XRT_ERROR_MODULE_AIE_NOC,
XRT_ERROR_MODULE_AIE_PL,
XRT_ERROR_MODULE_AIE_UNKNOWN
};

enum xrtErrorClass {
XRT_ERROR_CLASS_FIRST_ENTRY = 1,
XRT_ERROR_CLASS_SYSTEM = XRT_ERROR_CLASS_FIRST_ENTRY,
```

```
XRT_ERROR_CLASS_AIE,
XRT_ERROR_CLASS_HARDWARE,
XRT_ERROR_CLASS_UNKNOWN,
XRT_ERROR_CLASS_LAST_ENTRY = XRT_ERROR_CLASS_UNKNOWN
};
```

The API header file `experimental/xrt_error.h` defines the APIs for accessing currently cached errors. It provides `xrtErrorGetLast()` and `xrtErrorGetString()` APIs to retrieve the system level asynchronous errors.

```
/**
 * xrtErrorGetLast - Get the last error code and its timestamp of a given
 * error class.
 *
 * @handle:      Device handle.
 * @class:       Error Class for the last error to get.
 * @error:       Returned XRT error code.
 * @timestamp:   The timestamp when the error generated
 *
 * Return:       0 on success or appropriate XRT error code.
 */
int
xrtErrorGetLast(xrtDeviceHandle handle, xrtErrorClass ecl, xrtErrorCode*
error, uint64_t* timestamp);

/**
 * xrtErrorGetString - Get the description string of a given error code.
 *
 * @handle:      Device handle.
 * @error:       XRT error code.
 * @out:         Preallocated output buffer for the error string.
 * @len:         Length of output buffer.
 * @out_len:     Output of length of message, ignored if null.
 *
 * Return:       0 on success or appropriate XRT error code.
 *
 * Specifying out_len while passing nullptr for output buffer will
 * return the message length, which can then be used to allocate the
 * output buffer itself.
 */
int
xrtErrorGetString(xrtDeviceHandle, xrtErrorCode error, char* out, size_t
len, size_t* out_len);
```

The application can call `xrtErrorGetLast()` with a given error class to get the latest error code. The application can call `xrtErrorGetString()` with a given error code to get the error string corresponding to this error code. XRT maintains the latest error for each class and an associated timestamp for when the error was generated.

`xbutil` can be used to report errors. The error report accumulates *all* the errors from the various classes and sorts them by timestamp. The report queries the drivers as to when the last reset was requested. This reset will be merged (using the timestamp) into the report listing.

```
$ xbutil examine -r error -d 0000:00:00.0
Asynchronous Errors
  Time                               Class
Module          Driver              Severity      Error
Code
```

```

2020-Oct-08 16:40:02          CLASS_SYSTEM
MODULE_FIREWALL          DRIVER_XOCL          SEVERITY_EMERGENCY  FIREWALL_TRIP

$ xbutil2 examine -r error -f JSON-2020.2 -o <OUTPUT_FILE> -d 0000:00:00.0
{
  "schema_version": {
    "schema": "JSON",
    "creation_date": "Fri Oct  9 11:04:24 2020 GMT"
  },
  "devices": [
    {
      "asynchronous_errors": [
        {
          "timestamp": "1602175202572070700",
          "class": "CLASS_SYSTEM",
          "module": "MODULE_FIREWALL",
          "severity": "SEVERITY_EMERGENCY",
          "driver": "DRIVER_XOCL",
          "error_code": {
            "error_id": "1",
            "error_msg": "FIREWALL_TRIP"
          }
        }
      ]
    }
  ]
}

```

xbutil can also be used to report AI Engine running status and read registers for debug purposes. For example, the following command reads the status of kernels after the graph has executed.

```

$ xbutil examine -r aie -d 0000:00:00.0

-----
1/1 [0000:00:00.0] : edge
-----
Aie
  Aie_Metadata
    GRAPH[ 0] Name : gr
              Status : running
    SNo. Core [C:R] Iteration_Memory [C:R] Iteration_Memory_Addresses
    [ 0] 23:1 23:1 16388
    [ 1] 23:2 23:0 6980
    [ 2] 23:3 23:1 4
    [ 3] 24:1 24:0 4
    [ 4] 24:2 24:2 4
    [ 5] 24:3 24:1 4
    [ 6] 25:1 25:1 4

Core [ 0]
  Column : 23
  Row : 1
  Core:
    Status : core_done
    Program Counter : 0x00000308
    Link Register : 0x00000290
    Stack Pointer : 0x000340a0
  DMA:

```

```

MM2S:
  Channel:
    Id : 0
    Channel Status : idle
    Queue Size : 0
    Queue Status : okay
    Current BD : 0

    Id : 1
    Channel Status : idle
    Queue Size : 0
    Queue Status : okay
    Current BD : 0

S2MM:
  Channel:
    Id : 0
    Channel Status : idle
    Queue Size : 0
    Queue Status : okay
    Current BD : 0

    Id : 1
    Channel Status : idle
    Queue Size : 0
    Queue Status : okay
    Current BD : 0

Locks:
  0 : released_for_write
  1 : released_for_write
  2 : released_for_write
  3 : released_for_write
  4 : released_for_write
  5 : released_for_write
  6 : released_for_write
  7 : released_for_write
  8 : released_for_write
  9 : released_for_write
  10 : released_for_write
  11 : released_for_write
  12 : released_for_write
  13 : released_for_write
  14 : released_for_write
  15 : released_for_write

Events:
  core : 1, 2, 5, 22, 23, 24, 28, 29, 31, 32, 35, 36, 38, 39, 40, 44, 45,
47, 68
  memory : 1, 43, 44, 45, 106, 113

.....

Core [ 6]
  Column : 25
  Row : 1
  Core:
    Status : enabled, east_lock_stall
    Program Counter : 0x000001e6
    Link Register : 0x000000b0
    Stack Pointer : 0x00030020

```

```

DMA:
  MM2S:
    Channel:
      Id : 0
      Channel Status : stalled_on_requesting_lock
      Queue Size : 0
      Queue Status : okay
      Current BD : 2

      Id : 1
      Channel Status : idle
      Queue Size : 0
      Queue Status : okay
      Current BD : 0

  S2MM:
    Channel:
      Id : 0
      Channel Status : running
      Queue Size : 0
      Queue Status : okay
      Current BD : 0

      Id : 1
      Channel Status : idle
      Queue Size : 0
      Queue Status : okay
      Current BD : 0

Locks:
  0 : acquired_for_write
  1 : released_for_write
  2 : released_for_write
  3 : released_for_write
  4 : released_for_write
  5 : released_for_write
  6 : released_for_write
  7 : released_for_write
  8 : released_for_write
  9 : released_for_write
  10 : released_for_write
  11 : released_for_write
  12 : released_for_write
  13 : released_for_write
  14 : released_for_write
  15 : released_for_write

Events:
  core : 1, 2, 5, 22, 26, 28, 29, 31, 32, 35, 38, 39, 44
  memory : 1, 20, 21, 23, 35, 43, 44, 106, 113

```

The following command can be used to read specific registers for debug purposes.

```

$ xbutil advanced --read-aie-reg -d 0000:00:0 0 25 Core_Status
Register Core_Status Value of Row:0 Column:25 is 0x00000201

```

For AI Engine register definitions, see the *Versal ACAP AI Engine Register Reference* (AM015). For details on xbutil command use, see [Xilinx Runtime \(XRT\) Architecture](#).

AI Engine Error Events

This section provides error and related debug information for the errors obtained using the XRT error reporting APIs described previously. These are errors propagated from the AI Engine array and can be used to debug application specific errors in hardware. For errors with class `XRT_ERROR_CLASS_AIE`, you can obtain additional information by enabling the `dmesg` logs, which provide the causes of the error (and are described in the following tables). An example log is shown here:

```
[ 6616.963964] aie aie0: Asserted tile error event 56 at col 6 row 7
[DLBF] Completed reading 4 iterat[ 6616.970234] aie aie0: Asserted tile
error event 56 at col 7 row 8
[ 6616.979187] aie aie0: Asserted tile error event 56 at col 8 row 5
```

Note: Note the tile location is indicated by the `col` and `row` number. Row 0 is the SHIM (interface) tile, AI Engines start from row 1.

The following tables list the various categories of error, in addition to the exact error number, description, and tips on the next steps to debug and resolve the errors.

Table 74: CORE Module Error Events

Error Group	No.	Name	Description	Debug Tips
Instruction Errors	59	Instruction Decompression Error	Event generated when AI Engine cannot decompress instruction fetched. This can happen if the program instructions are corrupt. Validate ELF generation.	Regenerate the ELF file with the Vitis compiler (V++) <code>--package</code> command. If the issue persists, contact Xilinx support.
Access Errors	55	PM Reg Access Failure	This error can happen on bank access conflict to PM by the memory mapped AXI interface and AI Engine.	Contact Xilinx support.
	60	DM address out of range	Event generated if AI Engine tries to access a memory location outside of 0x20000 – 0x3FFFF.	Run AI Engine simulator (<code>aiesimulator</code>) with <code>--enable-memory-check</code> that will flag any access violations. Alternatively run <code>x86simulator</code> with <code>--valgrind</code> that will flag any access violations.
	65	PM address out of range	Event generated if PC is out of range	Run AI Engine simulator (<code>aiesimulator</code>) with <code>--enable-memory-check</code> that will flag any access violations. Alternatively run <code>x86simulator</code> with <code>--valgrind</code> that will flag any access violations.
	66	DM access to unavailable	Event generated if AI Engine issues an access to the isolated tile in neighborhood.	Check if the kernel runs on AI Engine accesses data memory of an isolated tile (a different partition). If the issue persists, contact Xilinx support.

Table 74: CORE Module Error Events (cont'd)

Error Group	No.	Name	Description	Debug Tips
Bus Errors	58	AXI MM Slave Error	Event generated if the memory mapped AXI interface slave read/write request is for an address which does not exist in the AI Engine tile.	If the PL IP is accessing the AI Engine registers using the memory mapped AXI interface, check the PL IP to see if it access invalid registers. If the issue persists, contact Xilinx support.
	54	TLAST in WSS words 0-2	Event generated if TLAST is not on the fourth word of a wide stream.	If PL IP is used to generate the stream, check if it generates TLAST correctly. If the issue persists, contact Xilinx support.
	56	Stream Pkt Parity Error	Event generated if there is any parity error in the packet header.	Check the data source such as PL IP which generates the packets to see if the packet is valid and if the parity bit is correctly calculated. If the data is from PL IP, check the packet header generated from the PL IP.
Stream Errors	57	Control Pkt Error	Control Packet Error	Check the data source, such as PL IP which generates the packets to see if it generates the packets correctly. If the issue persists, contact Xilinx support.
	64	PM ECC Error 2bit	Event generated when 2 bit ECC error is detected	Re-run the application. If the issue persists, contact Xilinx support.
	62	PM ECC Error Scrub 2bit	Event generated if ECC scrubber detects 2 Bit ECC error	Re-run the application. If the issue persists, contact Xilinx support.
ECC Errors	67	Lock Access to unavailable	Event generated if AI Engine issues an access to the isolated tile in neighborhood.	Run AI Engine simulator (<code>aiesimulator</code>) with <code>--enable-memory-check</code> that will flag any access violations. If the issue persists, contact Xilinx support. Alternatively run <code>x86simulator</code> with <code>--valgrind</code> that will flag any access violations.
Lock Errors				

Notes:

- CORE refers to the AI Engine in the AI Engine tile.

Table 75: MEMORY Module Error Events

Errors Group	No.	Name	Description	Debug Tips
ECC Errors	88	DM ECC Error Scrub 2bit	Event generated when ECC scrubber detects 2-bit ECC error in bank 0 or bank 1 of DM.	Re-run the application. If the issue persists, contact Xilinx support.
	90	DM ECC Error 2bit	Event generated when 2-bit ECC error is detected during access to bank 0 or 1 of DM. This data memory ECC error can be caused by DM access from the AI Engine, tile DMA, or memory mapped AXI interface.	Re-run the application. If the issue persists, contact Xilinx support.

Table 75: MEMORY Module Error Events (cont'd)

Errors Group	No.	Name	Description	Debug Tips
Memory Parity Errors	91	DM Parity Error Bank 2	Event generated when a parity error is detected during access to DM bank 2. This data memory parity error can be caused by DM access from the AI Engine, tile DMA, or memory mapped AXI interface.	Re-run the application. If the issue persists, contact Xilinx support.
	92	DM Parity Error Bank 3	Event generated when a parity error is detected during access to DM bank 3. This data memory parity error can be caused by DM access from the AI Engine, tile DMA, or memory mapped AXI interface.	Re-run the application. If the issue persists, contact Xilinx support.
	93	DM Parity Error Bank 4	Event generated when a parity error is detected during access to DM bank 4. This data memory parity error can be caused by DM access from the AI Engine, tile DMA, or memory mapped AXI interface.	Re-run the application. If the issue persists, contact Xilinx support.
	94	DM Parity Error Bank 5	Event generated when a parity error is detected during access to DM bank 5. This data memory parity error can be caused by DM access from the AI Engine, tile DMA, or memory mapped AXI interface.	Re-run the application. If the issue persists, contact Xilinx support.
	95	DM Parity Error Bank 6	Event generated when a parity error is detected during access to DM bank 6. This data memory parity error can be caused by DM access from the AI Engine, tile DMA, or memory mapped AXI interface.	Re-run the application. If the issue persists, contact Xilinx support.
	96	DM Parity Error Bank 7	Event generated when a parity error is detected during access to DM bank 7. This data memory parity error can be caused by DM access from the AI Engine, tile DMA, or memory mapped AXI interface.	Re-run the application. If the issue persists, contact Xilinx support.

Table 75: MEMORY Module Error Events (cont'd)

Errors Group	No.	Name	Description	Debug Tips
DMA Errors	97	DMA S2MM 0 Error	This error can be caused by writing to the BD task queue of S2MM channel 0 when it is full.	If you manage buffer descriptors in your application, verify that you are not pushing new buffer descriptors when the queue is full. If the issue persists, contact Xilinx support.
	98	DMA S2MM 1 Error	This error can be caused by writing to the BD task queue of S2MM channel 1 when it is full.	If you manage buffer descriptors in your application, verify that you are not pushing new buffer descriptors when the queue is full. If the issue persists, contact Xilinx support.
	99	DMA MM2S 0 Error	This error can be caused by writing to the BD task queue of MM2S channel 0 when it is full.	If you manage buffer descriptors in your application, verify that you are not pushing new buffer descriptors when the queue is full. If the issue persists, contact Xilinx support.
	100	DMA MM2S 1 Error	This error can be caused by writing to the BD task queue of MM2S channel 1 when it is full.	If you manage buffer descriptors in your application, verify that you are not pushing new buffer descriptors when the queue is full. If the issue persists, contact Xilinx support.

Table 76: SHIM Module Error Events

Error Group	No.	Name	Description	Debug Tips
Bus Errors	62	AXI MM Slave Tile Error	Event generated if a memory mapped AXI interface slave request comes to an interface tile but the address is invalid.	If using the PL IP to access the AI Engine register with the memory mapped AXI interface, check if the IP tries to access the wrong address. If the issue persists, contact Xilinx support.
	64	AXI MM Decode NSU Error	The memory mapped AXI interface traffic internally has responded with a DECERR. For example, if a column, set of tiles are clock gated, a decode error is generated internally and travels on the memory mapped AXI interface to the interface tile to generate this event.	If using the PL IP to access the AI Engine register using the memory mapped AXI interface, check if the IP tries to access tile which is gated. If the issue persists, contact Xilinx support.
	65	AXI MM Slave NSU Error	The memory mapped AXI interface traffic internally has responded with a SLVERR. For example, an AI Engine tile in that interface tile column has responded with a slave error. That slave error will travel over the memory mapped AXI interface to the interface tile as a slave error.	If using the PL IP to access the AI Engine register with the memory mapped AXI interface, check if the IP tries to access wrong address. If the issue persists, contact Xilinx support.
	66	AXI MM Unsupported Traffic	The memory mapped AXI interface from the NoC has made a request that the AI Engine does not support.	If using the PL IP to access the AI Engine register with the memory mapped AXI interface, check if the IP generates unsupported memory mapped AXI interface requests.
	67	AXI MM Unsecure Access in Secure Mode	The memory mapped AXI interface from the NoC is violating the secure mode (trying to route unsecured traffic when AI Engine only supports secure traffic).	Check if the AI Engine array is configured in secure mode.
	68	AXI MM Byte Strobe Error	The memory mapped AXI interface from the NoC is writing with non-complete 32-bit words (within a 32-bit word all byte strobes must be set).	If the PL IP is accessing the AI Engine using the memory mapped AXI interface, check if all byte strobes are set for a 32-bit word.
Stream Error	63	Control Pkt Error	Control Packet Error	If the PL IP is generating the control packets, check if the IP generates packets properly. If the issue persists, contact Xilinx support.

Table 76: SHIM Module Error Events (cont'd)

Error Group	No.	Name	Description	Debug Tips
DMA Error	69	DMA S2MM 0 Error	This DMA error is for DMA S2MM channel 0. It can be caused by: <ul style="list-style-type: none"> writing to the BD task queue when it is full; decode error when it tries to access the memory slave error when it tries to access the memory 	If you manage buffer descriptors in your application, verify that you are not pushing new buffer descriptors when the queue is full. If you manage buffer descriptors in your application, check if the memory address sent to the interface tile DMA buffer descriptor is invalid. If the issue persists, contact Xilinx support.
	70	DMA S2MM 1 Error	This DMA error is for DMA S2MM channel 1. It can be caused by: <ul style="list-style-type: none"> writing to the BD task queue when it is full; decode error when it tries to access the memory slave error when it tries to access the memory 	If you manage buffer descriptors in your application, verify that you are not pushing new buffer descriptors when the queue is full. If you manage buffer descriptors in your application, check if memory address sent to the interface tile DMA buffer descriptor is invalid. If the issue persists, contact Xilinx support.
	71	DMA MM2S 0 Error	This DMA error is for DMA MM2S channel 0. It can be caused by: <ul style="list-style-type: none"> writing to the BD task queue when it is full; decode error when it tries to access the memory slave error when it tries to access the memory 	If you manage buffer descriptors in your application, verify that you are not pushing new buffer descriptors when the queue is full. If you manage buffer descriptors in your application, check if memory address sent to the interface tile DMA buffer descriptor is invalid. If the issue persists, contact Xilinx support.
	72	DMA MM2S 1 Error	This DMA error is for DMA MM2S channel 1. It can be caused by: <ul style="list-style-type: none"> writing to the BD task queue when it is full; decode error when it tries to access the memory slave error when it tries to access the memory 	If you manage buffer descriptors in your application, verify that you are not pushing new buffer descriptors when the queue is full. If you manage buffer descriptors in your application, check if memory address sent to the interface tile DMA buffer descriptor is invalid. If the issue persists, contact Xilinx support.

Notes:

- SHIM refers to the interface tiles in the AI Engine array.

Host Code Reference with ADF API and XRT API

This section provides a summary of the XRT APIs that control the PL kernels and graph as well as a mapping relationship between the ADF API and XRT API. Complete host code using the ADF API or the XRT API to control the graph is also provided for reference.

Note: This section only lists part of APIs. See <https://github.com/xilinx/xrt> for the latest and more detailed information about the XRT API.

Table 77: XRT APIs

XRT API	Description
Category: Device handle (experimental/xrt_device.h)	
<code>xrtDeviceHandle xrtDeviceOpen(unsigned int index);</code>	Open a device and obtain its handle.
<code>xrtDeviceHandle xrtDeviceOpenFromXcl(xclDeviceHandle xhdl);</code>	Get a device handle from <code>xclDeviceHandle</code> .
<code>int xrtDeviceClose(xrtDeviceHandle dhd1);</code>	Close an opened device.
<code>int xrtDeviceLoadXclbinFile(xrtDeviceHandle dhd1, const char* xclbin_fnm);</code>	Read and load an XCLBIN file.
<code>void xrtDeviceGetXclbinUUID(xrtDeviceHandle dhd1, xuid_t out);</code>	Get UUID of XCLBIN image loaded on device.
Category: PL kernel handle (experimental/xrt_kernel.h)	
<code>xrtKernelHandle xrtPLKernelOpen(xrtDeviceHandle deviceHandle, const xuid_t xclbinId, const char *name);</code>	Open a PL kernel and obtain its handle.
<code>int xrtKernelClose(xrtKernelHandle kernelHandle);</code>	Close an opened kernel.
<code>xrtRunHandle xrtKernelRun(xrtKernelHandle kernelHandle, ...);</code>	Start a kernel execution.
<code>xrtRunHandle xrtRunOpen(xrtKernelHandle kernelHandle);</code>	Open a new run handle for a kernel without starting kernel.
<code>int xrtRunSetArg(xrtRunHandle rhdl, int index, ...);</code>	Set a specific kernel argument for this run.
<code>int xrtRunUpdateArg(xrtRunHandle rhdl, int index, ...);</code>	Asynchronous update of kernel argument.
<code>int xrtRunStart(xrtRunHandle rhdl);</code>	Start existing run handle.
<code>enum ert_cmd_state xrtRunWait(xrtRunHandle rhdl);</code>	Wait for a run to complete.
<code>int xrtRunClose(xrtRunHandle rhdl);</code>	Close a run handle.
Category: Graph handle (experimental/xrt_graph.h)	
<code>xrtGraphHandle xrtGraphOpen(xrtDeviceHandle handle, const uuid_t xclbinUUID, const char *graphName);</code>	Open a graph and obtain its handle.
<code>void xrtGraphClose(xrtGraphHandle gh);</code>	Close an open graph.
<code>int xrtGraphRun(xrtGraphHandle gh, int iterations);</code>	Start a graph execution.
<code>int xrtGraphWait(xrtGraphHandle gh, uint64_t cycle);</code>	Wait a set number of AI Engine cycles since the last <code>xrtGraphRun</code> and then stop the graph. If <code>cycle</code> is 0, wait until the graph is finished. If the graph has already run more than the set number of cycles, stop the graph immediately.
<code>int xrtGraphResume(xrtGraphHandle gh);</code>	Resume a suspended graph.
<code>int xrtGraphEnd(xrtGraphHandle gh, uint64_t cycle);</code>	Wait a set number of AI Engine cycles since the last <code>xrtGraphRun</code> and then end the graph. If <code>cycle</code> is 0, wait until the graph is finished before ending the graph. If the graph has already run more than the set number of cycles, stop the graph immediately and end it.
<code>int xrtGraphUpdateRTP(xrtGraphHandle gh, const char *hierPathPort, const char *buffer, size_t size);</code>	Update RTP value of port with hierarchical name.
<code>int xrtGraphReadRTP(xrtGraphHandle gh, const char *hierPathPort, char *buffer, size_t size);</code>	Read RTP value of port with hierarchical name.

Table 77: XRT APIs (cont'd)

XRT API	Description
Category: AIE handle (experimental/xrt_aie.h)	
<code>int xrtAIESyncBO(xrtDeviceHandle handle, xrtBufferHandle bohdl, const char *gmioName, enum xclBOSyncDirection dir, size_t size, size_t offset);</code>	Transfer data between DDR memory and interface tile DMA channel.
Category: Buffer object handle (experimental/xrt_bo.h)	
<code>xrtBufferHandle xrtBOAlloc(xrtDeviceHandle dhd1, size_t size, xrtBufferFlags flags, xrtMemoryGroup grp);</code>	Allocate a BO of requested size with appropriate flags.
<code>int xrtBOFree(xrtBufferHandle bhd1);</code>	Free/Deallocate the allocated BO.
<code>int xrtBOSync(xrtBufferHandle bhd1, enum xclBOSyncDirection dir, size_t size, size_t offset);</code>	Synchronize buffer contents in requested direction.
<code>void* xrtBOMap(xrtBufferHandle bhd1);</code>	Memory map BO into user address space.
Category: Error reporting (experimental/xrt_error.h)	
<code>int xrtErrorGetLast(xrtDeviceHandle handle, xrtErrorClass ecl, xrtErrorCode* error, uint64_t* timestamp);</code>	Get the last error code and its timestamp of a given error class.
<code>int xrtErrorGetString(xrtDeviceHandle, xrtErrorCode error, char* out, size_t len, size_t* out_len);</code>	Get the description string of a given error code.

The following table lists the mapping between the ADF API and XRT API. The `xrtGraphOpen()`, `xrtPLKernelOpen()`, `xrtRunOpen()`, `xrtKernelClose()` XRT APIs are called inside the ADF APIs when required and there is no corresponding mapping listed.

Table 78: ADF API and XRT API mapping

Graph API	XRT API
<code>graph::run()</code>	<code>xrtGraphRun(xrtGraphHandle, 0)</code> for AI Engine.
<code>graph::run(iterations)</code>	<code>xrtGraphRun(xrtGraphHandle, iterations)</code> for AI Engine.
<code>graph::wait()</code>	<code>xrtGraphWait(xrtGraphHandle, 0)</code> for AI Engine.
<code>graph::wait(aie_cycles)</code>	<code>xrtGraphWait(xrtGraphHandle aie_cycles)</code> , for AI Engine.
<code>graph::resume()</code>	<code>xrtGraphResume(xrtGraphHandle)</code>
<code>graph::end()</code>	<code>xrtGraphEnd(xrtGraphHandle, 0)</code> and then <code>xrtGraphClose(xrtGraphHandle)</code> for AI Engine.
<code>graph::end(aie_cycles)</code>	<code>xrtGraphEnd(xrtGraphHandle, aie_cycles)</code> and then <code>xrtGraphClose(xrtGraphHandle)</code> for AI Engine.
<code>graph::update()</code>	<code>xrtGraphUpdateRTP()</code> for AI Engine;
<code>graph::read()</code>	<code>xrtGraphReadRTP()</code> for AI Engine;
<code>GMIO::malloc()</code>	<code>xrtBOAlloc()</code> , <code>xrtBOMap()</code>
<code>GMIO::free()</code>	<code>xrtBOFree()</code>
<code>GMIO::gm2aie_nb()</code>	N/A
<code>GMIO::aie2gm_nb()</code>	N/A
<code>GMIO::wait()</code>	N/A

Table 78: ADF API and XRT API mapping (cont'd)

Graph API	XRT API
GMIO::gm2aie()	xrtSyncBOAIE(...,XCL_BO_SYNC_BO_GMIO_TO_AIE,...)
GMIO::aie2gm()	xrtSyncBOAIE(...,XCL_BO_SYNC_BO_AIE_TO_GMIO,...)
adf::event APIs for profiling and event trace	N/A

The following is host code using the ADF API and XRT API for reference. The `__USE_ADF_API__` is a user-defined macro in the code that can be used to switch between the ADF API and XRT API to control the AI Engine graph.

```
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include "host.h"
#include <unistd.h>
#include <complex>
#include "adf/adf_api/XRTConfig.h"
#include "experimental/xrt_kernel.h"

#include "graph.cpp"

#define OUTPUT_SIZE 2048

using namespace adf;

int main(int argc, char* argv[]) {

    size_t output_size_in_bytes = OUTPUT_SIZE * sizeof(int);

    //TARGET_DEVICE macro needs to be passed from gcc command line
    if(argc != 2) {
        printf("Usage: %d <xclbin>\r\n",argv[0]);
        return EXIT_FAILURE;
    }
    char* xclbinFilename = argv[1];

    int ret;
    // Open xclbin
    auto dhdl = xrtDeviceOpen(0);//device index=0
    if(!dhdl){
        printf("Device open error\n");
    }
    ret=xrtDeviceLoadXclbinFile(dhdl,xclbinFilename);
    if(ret){
        printf("Xclbin Load fail\n");
    }
    xuid_t uuid;
    xrtDeviceGetXclbinUUID(dhdl, uuid);

    // output memory
    xrtBufferHandle out_bohdl = xrtBOAlloc(dhdl, output_size_in_bytes, 0, /*BANK=*/0);
    std::complex<short> *host_out = (std::complex<short>*)xrtBOMap(out_bohdl);

    // s2mm ip
    xrtKernelHandle s2mm_khdl = xrtPLKernelOpen(dhdl, uuid, "s2mm");
    xrtRunHandle s2mm_rhdl = xrtRunOpen(s2mm_khdl);
    xrtRunSetArg(s2mm_rhdl, 0, out_bohdl);
    xrtRunSetArg(s2mm_rhdl, 2, OUTPUT_SIZE);
    xrtRunStart(s2mm_rhdl);
    printf("run s2mm\n");
```

```

#if __USE_ADF_API__
// update graph parameters (RTP) & run
adf::registerXRT(dhdl, uuid);
printf("Register XRT\r\n");
int narrow_filter[12] = {180, 89, -80, -391, -720, -834, -478, 505, 2063, 3896, 5535,
6504};
int wide_filter[12] = {-21, -249, 319, -78, -511, 977, -610, -844, 2574, -2754, -1066,
18539};
gr.run(16); //start AIE kernel
gr.update(gr.fir24.in[1], narrow_filter, 12); //update AIE kernel RTP
printf("Update fir24 done\r\n");
printf("Graph run done\r\n");
gr.wait(); // wait for AIE kernel to complete
printf("Graph wait done\r\n");
gr.update(gr.fir24.in[1], wide_filter, 12); //Update AIE kernel RTP
printf("Update fir24 done\r\n");
gr.run(16); //start AIE kernel
printf("Graph run done\r\n");
#else
int narrow_filter[12] = {180, 89, -80, -391, -720, -834, -478, 505, 2063, 3896, 5535,
6504};
int wide_filter[12] = {-21, -249, 319, -78, -511, 977, -610, -844, 2574, -2754, -1066,
18539};
auto ghdl=xrtGraphOpen(dhdl,uuid,"gr");
if(!ghdl){
printf("Graph Open error\r\n");
}else{
printf("Graph Open ok\r\n");
}
int size=1024;
xrtKernelHandle noisegen_khdl = xrtPLKernelOpen(dhdl, uuid, "random_noise");
xrtRunHandle noisegen_rhdl = xrtRunOpen(noisegen_khdl);
xrtRunSetArg(noisegen_rhdl, 1, size);
xrtRunStart(noisegen_rhdl);
printf("run noisegen\n");
ret=xrtGraphUpdateRTP(ghdl,"gr.fir24.in[1]",(char*)narrow_filter,12*sizeof(int));
if(ret!=0){
printf("Graph RTP update fail\n");
return 1;
}
ret=xrtGraphRun(ghdl,16);
if(ret){
printf("Graph run error\r\n");
}else{
printf("Graph run ok\r\n");
}
ret=xrtGraphWait(ghdl,0);
if(ret){
printf("Graph wait error\r\n");
}else{
printf("Graph wait ok\r\n");
}
xrtRunWait(noisegen_rhdl);
xrtRunSetArg(noisegen_rhdl, 1, size);
xrtRunStart(noisegen_rhdl);
printf("run noisegen\n");
ret=xrtGraphUpdateRTP(ghdl,"gr.fir24.in[1]",(char*)wide_filter,12*sizeof(int));
if(ret!=0){
printf("Graph RTP update fail\n");
return 1;
}
ret=xrtGraphRun(ghdl,16);
if(ret){
printf("Graph run error\r\n");
}else{
printf("Graph run ok\r\n");
}
}
#endif
// wait for s2mm done
auto state = xrtRunWait(s2mm_rhdl);
printf("s2mm completed with status %d\r\n",state);

```

```
xrtBOSync(out_bohdl, XCL_BO_SYNC_BO_FROM_DEVICE , output_size_in_bytes, /*OFFSET=*/ 0);

std::ofstream out("out.txt", std::ofstream::out);
std::ifstream golden("data/filtered.txt", std::ifstream::in);
short g_real=0, g_imag=0;
int match = 0;
for (int i = 0; i < OUTPUT_SIZE; i++) {
    golden >> std::dec >> g_real;
    golden >> std::dec >> g_imag;
    if(g_real!=host_out[i].real() || g_imag!=host_out[i].imag()){

        printf("ERROR: i=%d gold.real=%d gold.imag=%d out.real=%d out.imag=%d\r\n", i, g_real, g_imag, host_out[i].real(), host_out[i].imag());
        match=1;
    }
    out<<host_out[i].real()<<" "<<host_out[i].imag()<<" "<<std::endl;
}
out.close();
golden.close();

#if __USE_ADF_API__
    gr.end();
#else
    ret=xrtGraphEnd(ghdl, 0);
    if(ret){
        printf("Graph end error\r\n");
    }
    xrtRunClose(noisegen_rhdl);
    xrtKernelClose(noisegen_khdl);
    xrtGraphClose(ghdl);
#endif
    xrtRunClose(s2mm_rhdl);
    xrtKernelClose(s2mm_khdl);
    xrtBOFree(out_bohdl);
    xrtDeviceClose(dhdl);

    char pPass[] = "PASSED";
    char pFail[] = "FAILED";
    char* presult;
    presult = (match ? pFail : pPass);
    printf("TEST %s\r\n", presult);

    return (match ? EXIT_FAILURE : EXIT_SUCCESS);
}
```

The XRT API has C and C++ versions for controlling the PL kernels. For more information about the C++ version of the XRT API, see [XRT Native APIs](#).

Host Programming for Bare-Metal Systems

In a bare-metal/standalone environment, Xilinx provides standalone board support package (BSP), drivers, and libraries for applications to use to reduce development effort. As described in [Host Programming on Linux](#), the top-level application for bare-metal systems must also integrate and manage the AI Engine graph and PL kernels.



TIP: The steps to integrate a bare-metal system with the AI Engine graph and PL kernels is described in [Building a Bare-metal System](#), or in [Building a Bare-metal AI Engine in the Vitis IDE](#).

The following is an example top-level application (`main.cpp`) for a bare-metal system:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include "platform.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xil_cache.h"
#include "input.h"
#include "golden.h"
...
void InitData(int32_t** out, int size)
{
    int i;
    *out = (int32_t*)malloc(sizeof(int32_t) * size);

    if(!out) {
        printf("Allocation of memory failed \n");
        exit(-1);
    }

    for(i = 0; i < size; i++) {
        (*out)[i] = 0xABCDEF00;
    }
}

int RunTest(uint64_t mm2s_base, uint64_t s2mm_base, int32_t* in, int32_t*
golden,
    int32_t* out, int input_size, int output_size)
{
    int i;
    int errCount = 0;
    uint64_t memAddr = (uint64_t)in;
    uint64_t mem_outAddr = (uint64_t)out;

    printf("Starting test w/ cu\n");

    Xil_Out32(mm2s_base + MEM_OFFSET, (uint32_t) memAddr);
    Xil_Out32(mm2s_base + MEM_OFFSET + 4, 0);
    Xil_Out32(s2mm_base + MEM_OFFSET, (uint32_t) mem_outAddr);
    Xil_Out32(s2mm_base + MEM_OFFSET + 4, 0);
    Xil_Out32(mm2s_base + SIZE_OFFSET, input_size);
    Xil_Out32(s2mm_base + SIZE_OFFSET, output_size);
    Xil_Out32(mm2s_base + CTRL_OFFSET, 1);
    Xil_Out32(s2mm_base + CTRL_OFFSET, 1);

    printf("GRAPH INIT\n");
    clipgraph.init();

    printf("GRAPH RUN\n");
    clipgraph.run();

    while(1) {
        uint32_t v = Xil_In32(s2mm_base + CTRL_OFFSET);
        if(v & 6) {
            break;
        }
    }

    printf("PLIO IP DONE!\n");
}
```

```

        for(i = 0; i < output_size; i++) {
            if((((int32_t*)out)[i] != ((int32_t*)golden)[i]) ) {
                printf("Error found in sample %d != to the golden %d\n", i+1,
                    ((int32_t*)out)[i], ((int32_t*)golden)[i]);
                errCount++;
            }
            else
                printf("%d\n ", ((int32_t*)out)[i]);
        }

        printf("Ending test w/ cu\n");
        return errCount;
    }

int main()
{
    int i;
    int32_t* out;
    int errCount;

    Xil_DCacheDisable();
    init_platform();
    sleep(1);

    printf("Beginning test\n");
    InitData(&out, OUTPUT_SIZE);
    errCount = RunTest(MM2S_BASE, S2MM_BASE, (int32_t*)cint16input,
        int32golden, out, INPUT_SIZE, OUTPUT_SIZE);

    if(errCount == 0)
        printf("Test passed. \n");
    else
        printf("Test failed! Error count: %d \n",errCount);

    cleanup_platform();
    return errCount;
}

```

The following are the steps in the code example:

- The `main()` function initializes the platform, data, runs the test, verifies the return code, and return the error code.
- `InitData()` allocates `size` of memory space and initializes successfully allocated memory space to known data.
- `RunTest()` passes necessary data to the kernel to process and return a result.
- `clipgraph.init()` initializes the tiles that kernels will be run on.
- `clipgraph.run()` starts kernels running on associated tiles.

The preceding code example references `xparameters.h` which is automatically generated from the bare-metal BSP. The application needs to ensure the bare-metal BSP is properly generated so that the memory mapped addresses for all drivers are correctly assigned.

`xil_io.h` contains general driver I/O APIs. This is the preferred method for accessing drivers.

Addressing Kernels in Bare-metal Applications

For bare-metal applications, when addressing the PL kernels from the embedded application, you must use the control registers, or read and write to the kernel at the appropriate base address and offset as it is implemented in hardware. Looking at the application discussed earlier, the embedded application can deliver data to the MM2S kernel, to introduce it to the AI Engine graph for the Interpolator and Classifier kernels, and read data from the S2MM kernel to continue processing in the embedded application. In this case, address the MM2S and S2MM kernels as they are implemented in the PL region of the fixed platform.

The `main.cpp` of the example shows the `#define` statements for the kernel base address and the address offset for specific registers. For example:

```
#define MM2S_BASE XPAR_MM2S_S_AXI_CONTROL_BASEADDR
#define S2MM_BASE XPAR_S2MM_S_AXI_CONTROL_BASEADDR

#define MEM_OFFSET 0x10
#define SIZE_OFFSET 0x1C
#define CTRL_OFFSET 0x0
```

To determine the address and offsets for the kernels, examine some of the files in the fixed platform. The location of the base address for the implemented kernels is located in the fixed platform `xparameters.h` file, that is located in the `<platform_name>/standalone_domain/bsp/include/include` folder. For the example design, use the following entries in `xparameters.h` to determine the base addresses of these kernels.

```
/* Definitions for peripheral MM2S */
#define XPAR_MM2S_S_AXI_CONTROL_BASEADDR 0xA4020000
#define XPAR_MM2S_S_AXI_CONTROL_HIGHADDR 0xA402FFFF

/* Definitions for peripheral S2MM */
#define XPAR_S2MM_S_AXI_CONTROL_BASEADDR 0xA4030000
#define XPAR_S2MM_S_AXI_CONTROL_HIGHADDR 0xA403FFFF
```

Note: The `xparameters.h` file is generated and addressing is dynamic. It is best to reference the address macros for kernels than to hard code them.

The location of the address offset is located in the `<kernel_driver>_hw.h` file of the `_x/<kernel>/<kernel>/<kernel>/solution/impl/ip/drivers/<kernel>_v1_0/src` folder of the compiled kernel produced by the Vitis™ compiler. For example, the MM2S kernel driver, `xmm2s_mm2s_hw.h` displays the following data.

```
#define XMM2S_MM2S_CONTROL_ADDR_AP_CTRL 0x00
#define XMM2S_MM2S_CONTROL_ADDR_GIE 0x04
#define XMM2S_MM2S_CONTROL_ADDR_IER 0x08
#define XMM2S_MM2S_CONTROL_ADDR_ISR 0x0c
#define XMM2S_MM2S_CONTROL_ADDR_MEM_V_DATA 0x10
#define XMM2S_MM2S_CONTROL_BITS_MEM_V_DATA 64
#define XMM2S_MM2S_CONTROL_ADDR_SIZE_DATA 0x1c
#define XMM2S_MM2S_CONTROL_BITS_SIZE_DATA 32
```

Use these offsets when reading or writing to the kernels. For instance, from the example application `main.cpp` file use the following to write to the memory location.

```
Xil_Out32(MM2S_BASE + MEM_OFFSET, (uint32_t) memAddr);
```

Integrating the Application Using the Vitis Tools Flow

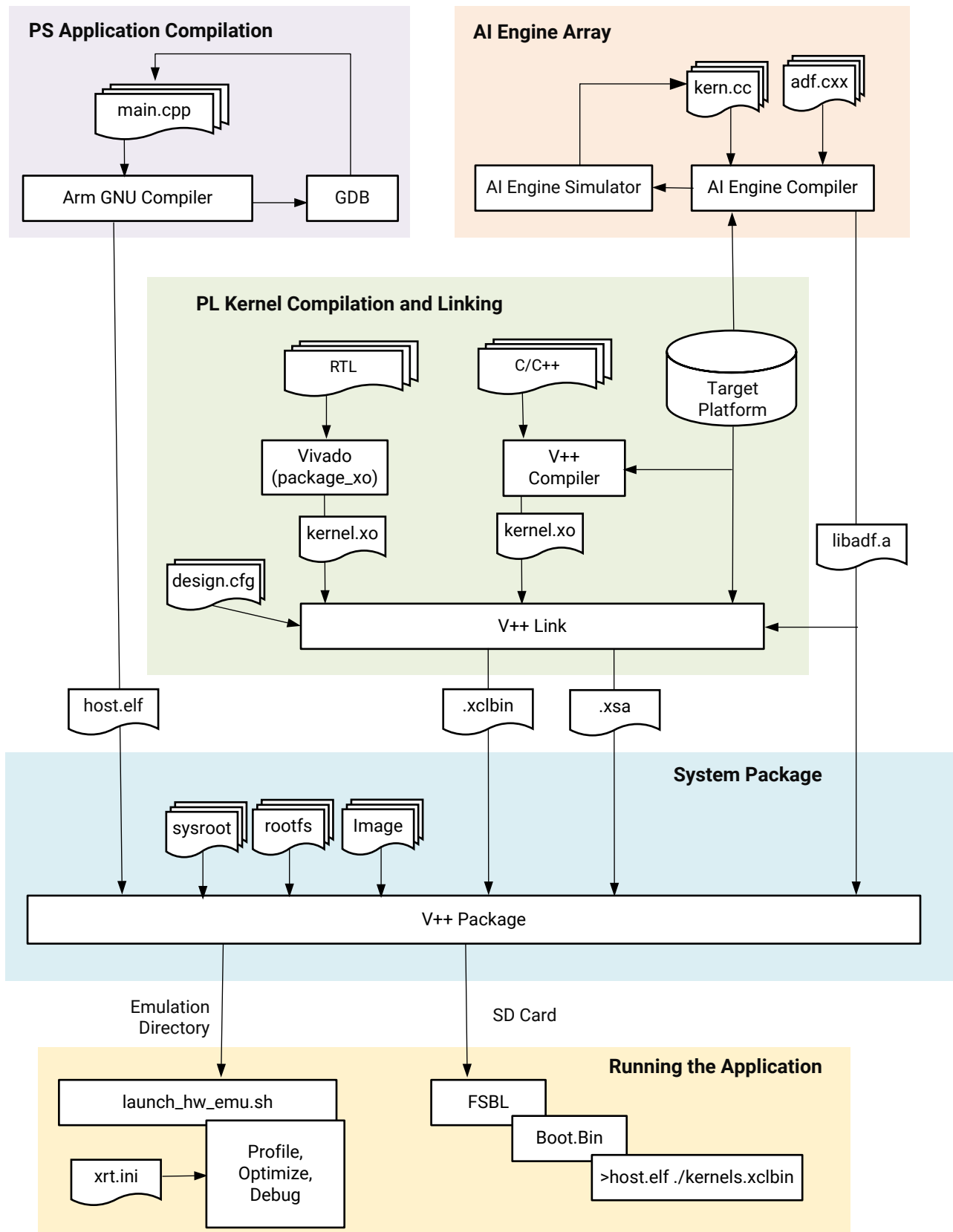
While developing an AI Engine design graph, many design iterations are typically performed using the AI Engine compiler or AI Engine simulator tools. This method provides quick design iterations when focused on developing the AI Engine application. When ready, the AI Engine design can be integrated into a larger system design using the flow described in this chapter.

The Vitis™ tools flow simplifies hardware design and integration with a software-like compilation and linking flow, integrating the three domains of the Versal® device: the AI Engine array, the programmable logic (PL) region, and the processing system (PS). The Vitis compiler flow lets you integrate your compiled AI Engine design graph (`libadf.a`) with additional kernels implemented in the PL region of the device, including HLS and RTL kernels, and link them for use on a target platform. You can call these compiled hardware functions from a host program running in the Arm® processor in the Versal device.

The following figure shows the high-level steps required to use the Vitis tools flow to integrate your application. The command-line process to run this flow is described here.

Note: You can also use this flow from within the Vitis IDE as explained in [Chapter 15: Using the Vitis IDE](#).

Figure 52: Vitis Tools Flow



X24782-060321



IMPORTANT! Using Vitis tools and AI Engine tools require the setup described in [Setting Up the Vitis Tool Environment](#).

The following steps can be adapted to any AI Engine design in a Versal device.

1. As described in [Chapter 9: Compiling an AI Engine Graph Application](#), the first step is to create and compile the AI Engine graph into a `libadf.a` file using the AI Engine compiler. You can iterate between the AI Engine compiler, and the AI Engine simulator to develop the graph, until you are ready to proceed.
2. [Compiling PL Kernels](#): PL kernels are compiled for implementation in the PL region of the target platform using the `v++ --compile` command. These kernels can be C/C++ kernels or RTL kernels, in compiled Xilinx object (`.xo`) form.
3. [Linking the System](#): Link the compiled AI Engine graph with the C/C++ kernels and RTL kernels onto a target platform. The process creates an XCLBIN file to load and run an AI Engine graph and PL kernels code in the target platform.
4. [Compile the Embedded Application for the Cortex-A72 Processor](#): Optionally compile a host application to run on the Cortex®-A72 core processor using the GNU Arm cross-compiler to create an ELF file. The host program interacts with the AI Engine kernels and kernels in the PL region. This compilation step is optional because there are several ways to deploy and interact with the AI Engine kernels, and the host program running in the PS is one way.
5. Packaging the system: Use the `v++ --package` process to gather the required files to configure and boot the system, to load and run the application, including the AI Engine graph and PL kernels. This builds the necessary package to run emulation and debug, or run your application on hardware.

Platforms

A platform is a fully contained image that defines both the hardware (XSA) as well as the software (bare metal, Linux, or both). The XSA contains the hardware description of the platform, which is defined in the Vivado Design Suite, and the software is defined with the use of a bare-metal setup, or a Linux image defined through PetaLinux.

Types of Platforms

There are two types of platforms: base platform, or custom platform. A base platform is one that is provided by Xilinx (for example, the `xilinx_vck190_base_202120_1`) typically targeting Xilinx boards, and a custom platform is one that you can create, either by extending or re-customizing a base platform or creating a new platform. When starting platform development, it can be useful to use a base platform as a reference development platform to create your custom platform.

Custom Platforms

You can create platforms that can re-customize an existing base platform (for example, changing the AI Engine clock frequency, clocks available in the programmable logic (PL), changing memory controller settings) or create a new platform targeting Xilinx or non-Xilinx boards. Creating a platform allows you to provide your own IP or subsystems to meet your needs. The process to create a platform can be found in [Creating Embedded Platforms in Vitis](#) in the *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393).

Platform Clocking

Platforms have a variety of clocking: processor, PL, and AI Engine clocking. The following table explains the clocking for each.

Table 79: Platform Clocks

Clock	Description
AI Engine	Can be configured in the platform in the AI Engine IP.
Processor	Can be configured in the platform in the CIPS IP.
Programmable Logic (PL)	Can have multiple clocks and can be configured in the platform.
NoC	Device dependent and can be configured in the platform in the CIPS and NoC IP.

Notes:

- These clocks are derived from the platform and are affected by the device, speed grade and operating voltage.

For more information related to platform clocking, see *Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#)). For information on Versal device clocks, see *Versal AI Core Series Data Sheet: DC and AC Switching Characteristics* ([DS957](#)).

PL Kernels

PL kernels can take the form of HLS kernels, written in C/C++, or RTL kernels packaged in the Vivado Design Suite. These kernels must be separately compiled to produce the Xilinx object files (XO) used in integrating the system design on the target platform.

HLS kernels, written in C/C++, can be written and compiled from within the Vitis HLS tool directly, or as part of the Vitis application acceleration development flow.

For information on creating and building RTL kernels, see [RTL Kernels](#) in the *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393).

Compiling PL Kernels

To compile kernels using the Vitis compiler command as described in the [Compiling Kernels with Vitis Compiler](#) in the *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393), use the following command syntax:

```
v++ --compile -t hw_emu --platform xilinx_vck190_base_202120_1 -g \
-k <kernel_name> <kernel>.cpp -o <kernel_name>.xo --save-temps
```

The `v++` command uses the options described in the following table.

Table 80: Vitis Compiler Options

Option	Description
<code>--compile</code>	Specifies compilation mode.
<code>-t hw_emu</code>	Specifies the build target for the compilation process. For more information, see the <i>Build Targets</i> section in <i>Vitis Unified Software Platform Documentation: Application Acceleration Development</i> (UG1393).
<code>--platform</code>	Specifies the path and name of the target platform. For this example command-line option, it is assumed to have PLATFORM_REPO_PATHS set to the right platform path.
<code>-g</code>	Enables the debug features. This is required for emulation modes.
<code>-k</code>	Specifies the kernel name. This must match the function name in the specified kernel source file.
<code>-o</code>	Specifies the output file name of the compiled Xilinx object file (.xo).
<code>--save-temps</code>	Saves the temporary files generated during the compilation process. This is optional.

Clocking the PL Kernels

PLIO represents an ADF graph interface to the PL. This PL could be a PL kernel, a platform IP representing a signal source or sink, or it could be a data mover to interface the ADF graph to memory. You should provide clock frequency values for these interfaces to ensure simulation results match the results from running the design in hardware. In addition, when you link the ADF graph into the platform, at the Vitis linker (`v++ -link`) step, you have the ability to provide more accurate clock values depending on the specific clock frequency values supported by the platform. To set the exact frequency of a PLIO interface in the graph and the clock frequency of the corresponding PL kernel you must specify the clock frequency in three locations:

- ADF graph (Optional)
- Vitis compilation of a PL kernel (`v++ -c`)
- Vitis linking (`v++ -l`)

You must specify the clocking depending on where the kernels are located. The following table describes the default clocks based on the kernel location.

Table 81: Default Kernel Clocks

Kernel Location	Description
AI Engine kernels	Clocked per the AI Engine clock frequency. All cores run with the same clock frequency.
PL kernels connected to AI Engine graph	HLS: Default frequency for all HLS kernels - 150 MHz RTL: Frequency is set to the frequency that the XO file was compiled with. AI Engine: Set in the PLIO constructor in the AI Engine graph. Setting the frequency here is optional. See Appendix A: Adaptive Data Flow Graph Specification Reference for more information. ¹
PL kernels added to platform using the Vitis linker	Platforms have a default clock. If no clocking option is set at the command line or configuration file the default clock is used. This default can be overridden depending on the design and required clock value, as shown in the following table.

Notes:

1. If the PLIO frequency is not provided in the PLIO constructor the AI Engine compiler defaults the frequency to one quarter of the AI Engine clock frequency. When you determine the target platform, Xilinx recommends setting the PLIO clock frequencies explicitly, to make your AI Engine simulations more representative of your application.

Note: The maximum supported PLIO interface clock frequency is half the AI Engine clock frequency, depending on the device speed-grade. If you specify a higher frequency, the Vitis linker will cap the frequency to the maximum supported frequency and will issue a critical warning in linker stage.

Setting the clocks at the Vitis linker step allows you to choose a frequency based on the platform. The following table describes the Vitis compiler clocking options during the link step.

Table 82: Vitis Linking Clock Options

[clock] Options	Description
--clock.defaultFreqHz arg	Specify a default clock frequency to use in Hz.
--clock.defaultId arg	Specify a default clock reference ID to use.
--clock.defaultTolerance arg	Specify a default clock tolerance to use.
--clock.freqHz arg	<frequency_in_Hz>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]] Specify a clock frequency in Hz and a list of associated compute unit names and optionally their clock pins.
--clock.id arg	<reference_ID>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]] Specify a clock reference ID and a list of associated compute unit names and optionally their clock pins.
--clock.tolerance arg	<tolerance>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]] Specify a clock tolerance and a list of associated compute unit names and optionally their clock pins.

The following table describes the steps to set clock frequencies for PLIOs that interface to the platform, including to PL kernels specified outside of the ADF graph.

Table 83: Compiling PL Kernels with Non-default Clocking

PL Kernel Location	Clock Specification
PLIO interface specified in ADF graph	<p>Specify the clock frequency per PLIO interface in the graph. For the PLIO interface, you can optionally specify <code>FreqMHz</code>.</p> <pre>adf::PLIO *(<input> = new adf::PLIO(<logical_name>, <plio_width>, <file>, <FreqMHz>);</pre>
HLS kernels	<p>Compile the HLS code using the Vitis compiler.</p> <pre>v++ -c -k kernelName kernel.cpp --hls.clock freqHz:kernelName</pre> <p>To change the frequency at which HLS kernels are compiled use: <code>--hls.clock arg:kernelName</code>. <code>arg</code> must be in Hz (for example, <code>250000000Hz</code> is 250 MHz). Per kernel, specify the clock in the Vitis linker.</p> <pre>v++ -l ... --clock.freqHz <freqHz>:kernelName.ap_clk</pre>
RTL kernels	<p>Per kernel, specify the clock in the Vitis linker.</p> <pre>v++ -l ... --clock.freqHz <freqHz>:kernelName.ap_clk</pre>

Note: Clock frequencies for PL kernels at the Vitis linker stage take precedence over Vitis compile time clock frequency values. However the clock frequency value specified at the Vitis linker stage should not exceed the Vitis compiler clock frequency value significantly, because the Vitis compiler generates RTL based on the target frequency specified.

See *Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)* for more detailed information on how to compile kernels for specific platform clocks and clocking information.

Linking the System

After the AI Engine graph and the C/C++ kernels are compiled, and any RTL kernels are packaged, the Vitis `v++ --link` command links them with the target platform to build the device binary (XCLBIN), used to program the hardware. For more information, see [Linking the Kernels](#) in the Application Acceleration Development flow of the *Vitis Unified Software Platform Documentation*.

The following is an example of the linking command for the Vitis compiler in the AI Engine design flow.

```
v++ --link -t hw_emu --platform xilinx_vck190_base_202120_1 -g \
<pl_kernel1>.xo <pl_kernel2>.xo ../libadf.a -o vck190_aie_graph.xclbin \
--config ../system.cfg --save-temps
```

The `v++` command uses the options in the following table.

Table 84: Vitis Compiler Link Options

Option	Description
<code>--link</code>	Specifies the linking process.
<code>-t hw_emu</code>	Specifies the build target of the link process. For the AI Engine kernel flow, the target can be either <code>hw_emu</code> for emulation and test, or <code>hw</code> to build the system hardware. IMPORTANT! <i>The <code>v++</code> compilation and linking commands must use both the same build target (<code>-t</code>) and the same target platform (<code>--platform</code>).</i>
<code>--platform</code>	Specifies the path to the target platform.
<code>-g</code>	Specifies the addition of debugging logic required to enable debug (for hardware emulation) and to capture waveform data.
<code><pl_kernel1>.xo</code> <code><pl_kernel2>.xo</code>	Specifies the input compiled PL kernel object files (<code>.xo</code>) to link with the AI Engine graph and the target platform.
<code>../libadf.a</code>	Specifies the input compiled AI Engine graph application to link with the PL kernels and the target platform.
<code>-o</code>	Specifies the device binary (XCLBIN) file that is the output of the linking process.
<code>--config</code>	Specifies a configuration file to define some of the compilation or linking options. ¹
<code>--save-temps</code>	Indicates that the temporary files created during the build process should be preserved for later examination or use. This includes output files created by Vitis HLS and the Vivado Design Suite.

Notes:

1. The `--config` option is used to simplify the `v++` command line by moving many commands with extended syntax into a file that can be specified from the command line. For more information, see the [Vitis Compiler Configuration file](#) in the *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393).



TIP: The `config` file requirements for the command line are different from the requirements of the Vitis IDE, as discussed in [Configuring the HW-Link Project](#).

For the AI Engine kernel flow, the Vitis compiler requires two specific sections in the configuration file: `[connectivity]` and `[advanced]`. The following is an example configuration file.

```
[connectivity]
nk=mm2s:1:mm2s
nk=s2mm:1:s2mm
stream_connect=mm2s.s:ai_engine_0.DataIn1
stream_connect=ai_engine_0.DataOut1:s2mm.s
[advanced]
param=compiler.addOutputTypes=hw_export
```

The `[connectivity]` section of the configuration file has options described in the following table.

Table 85: Connectivity Section Options

Option	Description
<code>nk</code>	<p>Specifies the number of kernel- instances or CUs the <code>v++</code> command adds to the device binary (XCLBIN). The <code>nk</code> option specifies the kernel name, the number of instances, or CUs of that kernel, and the CU name for each instance. In the example, <code>nk=mm2s:1:mm2s</code> specifies the kernel <code>mm2s</code> should only have one instance and that instance should be called <code>mm2s</code>.</p> <p>Multiple instances of the kernels are specified as <code>nk=mm2s:2:mm2s_1:mm2s_2</code>. This indicates that <code>mm2s</code> should have two CUs called <code>mm2s_1</code> and <code>mm2s_2</code>. For more information, see Creating Multiple Instances of a Kernel in the <i>Vitis Unified Software Platform Documentation: Application Acceleration Development</i> (UG1393).</p>
<code>sc</code>	<p>Defines AXI4-Stream connections between the ports of the AI Engine graph and the streaming ports of the PL kernels. Connections can be defined as the streaming output of one kernel connecting to the streaming input of a second kernel, or to a streaming input port on an IP implemented in the target platform. For more information, see --connectivity Options in the <i>Vitis Unified Software Platform Documentation: Application Acceleration Development</i> (UG1393).</p> <p>The example <code>stream_connect=mm2s.s:ai_engine_0.DataIn1</code> from the <code>config</code> file, defines a connection between the streaming output of the <code>mm2s</code> PL kernel, and the <code>DataIn1</code> input port of the AI Engine graph.</p> <p>The example <code>stream_connect=ai_engine_0.DataOut1:s2mm.s</code> defines a connection between the <code>DataOut1</code> output port of the AI Engine graph, to the input port <code>s</code> of the PL kernel <code>s2mm</code>. For more information, see Specify Streaming Connections Between Compute Units in the <i>Vitis Unified Software Platform Documentation: Application Acceleration Development</i> (UG1393).</p>

To instruct the Vitis linker to generate an XSA archive of the generated hardware design, add the following parameter to a configuration file.

```
[advanced]param=compiler.addOutputTypes=hw_export
```



TIP: The exported XSA is required for building the fixed platform in the bare-metal flow as described in [Building a Bare-metal System](#).

Alternatively, for a custom platform, add the following Tcl command prior to invoking `write_hw_platform`.

```
set_property compiler.default_output_type=hw_export
```

This directs the Vitis linker to always generate an XSA for any design.

During the linking process, the Vitis compiler invokes the Vivado Design Suite to generate the device binary (XCLBIN) for the target platform. The XCLBIN file is used to program the device and includes the following information.

- **PDI:** Programming information for the AI Engine array
- **Debug data:** Debug information when included in the build
- **Memory topology:** Defines the memory resources and structure for the target platform
- **IP Layout:** Defines layout information for the implemented hardware design
- **Metadata:** Various elements of platform meta data to let the tool load and run the XCLBIN file on the target platform

For more information on the XRT use of the XCLBIN file, see [XRT](#).

Compile the Embedded Application for the Cortex-A72 Processor

After linking the AI Engine graph and PL kernels, the focus moves to the embedded application running in the PS that interacts with the AI Engine graph and kernels. The PS application is written in C/C++, using API calls to control the initialization, running, and closing of the AI Engine graph as described in [Chapter 5: Run-Time Graph Control API](#).

You compile the embedded application by following the typical cross-compilation flow for the Arm Cortex-A72 processor. The following are example commands for compiling and linking the PS application:

```
aarch64-xilinx-linux-g++ -std=c++14 -O0 -g -Wall -c \
-I<platform_path>/sysroots/aarch64-xilinx-linux/usr/include/xrt \
--sysroot=<platform_path>/sysroots/aarch64-xilinx-linux/ \
-I./ -I./src -I${XILINX_HLS}/include/ -I${XILINX_VITIS}/aie-tools/include -o
sw/host.o sw/host.cpp

aarch64-xilinx-linux-g++ -std=c++14 -O0 -g -Wall -c \
-I<platform_path>/sysroots/aarch64-xilinx-linux/usr/include/xrt \
--sysroot=<platform_path>/sysroots/aarch64-xilinx-linux/ \
-I./ -I./src -I${XILINX_HLS}/include/ -I${XILINX_VITIS}/aie-tools/include -o
sw/aie_control_xrt.o Work/ps/c_rts/aie_control_xrt.cpp
```

Many of the options in the preceding command are standard and can be found in a description of the `g++` command. The more important options are listed as follows.

- `-std=c++14`
- `-I<platform_path>/sysroots/aarch64-xilinx-linux/usr/include/xrt`
- `--sysroot=<platform_path>/sysroots/aarch64-xilinx-linux/`
- `-I./ -I./src`
- `-I${XILINX_HLS}/include/`
- `-I${XILINX_VITIS}/aie-tools/include`
- `-o sw/host.o sw/host.cpp`

The cross compiler `aarch64-xilinx-linux-g++` is used to compile the Linux host code. `aie_control_xrt.cpp` is copied from the directory `Work/ps/c_rts`.

```
aarch64-xilinx-linux-g++ -ladf_api_xrt -lgcc -lc -lpthread -lrt -ldl \
-lcrypt -lstdc++ -lxrt_coreutil \
-L<platform_path>/sysroots/aarch64-xilinx-linux/usr/lib \
--sysroot=<platform_path>/sysroots/aarch64-xilinx-linux \
-L${XILINX_VITIS}/aie-tools/lib/aarch64.o -o sw/host.exe sw/host.o sw/
aie_control_xrt.o
```

Note in the preceding linker script that it links the `adf_api_xrt` libraries, which is necessary for the ADF API to work with the XRT API.

`xrt_coreutil` are required libraries for XRT and for the XRT API.

While many of the options can be found in a description of the `g++` command, some of the more important options are listed in the following table.

Table 86: Command Options

Option	Description
<code>-ladf_api_xrt</code>	Required for the ADF API. For more information, see Host Programming on Linux . This is used to control the AI Engine through XRT. If not controlling with XRT, use <code>-ladf_api</code> with the path <code>-L\${XILINX_VITIS}/aie-tools/lib/aarch64none.so</code> . For more information see Host Programming for Bare-Metal Systems .
<code>-lxrt_coreutil</code>	Required for the XRT API.
<code>-L<platform_path>/sysroots/aarch64-xilinx-linux/usr/lib</code>	
<code>--sysroot=<platform_path>/aarch64-xilinx-linux</code>	
<code>-L\${XILINX_VITIS}/aie-tools/lib/aarch64.o</code>	
<code>-o sw/host.exe</code>	

Packaging

The AI Engine compiler generates output in the form of a library file, `libadf.a`, which contains ELF and CDO files, as well as tool-specific data and metadata, for hardware and hardware emulation flows. To create a loadable image binary, this data must be combined with PL-based configuration data, boot loaders, and other binaries. The Vitis™ packager performs this function, combining information from `libadf.a` and the Vitis linker generated XSA file.

This requires the use of the Vitis packaging command (`v++ --package`) as described in [Vitis Compiler Command](#) in the *Vitis Unified Software Platform Documentation: Application Acceleration Development* (UG1393).

For Versal ACAPs, the programmable device image (PDI) file is used to boot and program the hardware device. For hardware emulation the `--package` command adds the PDI and `EMULATION_DATA` sections to the XCLBIN file, and outputs a new XCLBIN file. For hardware builds, the package process creates an XCLBIN file containing ELF files and graph configuration data objects (CDOs) for the AI Engine application.

In the Vitis IDE, the package process is automated and the tool creates the required files based on the build target, platform, and OS. However, in the command line flow, you must specify the Vitis packaging command (`v++ --package`) with the correct options for the job.

Packaging the System

For both hardware and hardware emulation, the `v++ --package` command takes the XCLBIN file and `libadf.a` as input, produces a script to launch hardware emulation (`launch_hw_emu.sh`), and writes the required support files. An example command line follows:

```
v++ --package --config package.cfg ./aie_graph/libadf.a \
./project.xclbin -o aie_graph.xclbin
```

where, the `--config package.cfg` option specifies a configuration file with the following options:

```
platform=xilinx_vck190_base_202120_1
target=hw_emu
save-temps=1

[package]
boot_mode=sd
out_dir=./emulation
enable_aie_debug=1
rootfs=<path_to_platform>/sw/versal/xilinx-versal-common-v2021.2/rootfs.ext4
image_format=ext4
kernel_image=<path_to_platform>/sw/versal/xilinx-versal-common-v2021.2/Image
sd_file=host.exe
```

The following table explains the options for both hardware and hardware emulation.

Table 87: Hardware and Hardware Emulation Options

Command-line Flag	Hardware	Hardware Emulation	Details
<code>platform</code>	Target platform	Target platform	Either a base platform, or a custom platform that meets AI Engine flow requirements.
<code>target</code>	<code>hw</code>	<code>hw_emu</code>	Specifies the hardware emulation build target. Specifying <code>hw_emu</code> as the target causes a number of files to be generated, including the PDI to boot the device, as well as files required for emulation. Specifying <code>hw</code> only generates the PDI file required to configure and boot the hardware.
<code>save-temps</code>			Causes the Vitis compiler to save intermediate files created during the build and package process.
Package Options			
<code>boot_mode¹</code>	<code>sd</code>	<code>sd</code>	Indicates the device boots from an SD card or from a QSPI image in flash memory. Values can be: <code>sd</code> or <code>qspi</code> .
<code>out-dir</code>	<code><path></code>	<code><path></code>	Specifies a directory where output files should be created. If <code>out-dir</code> is not specified, the files are written to the current working directory.
<code>kernel_image</code>	<code><path>/Image</code>	<code><path>/Image</code>	Specifies the image file that is specified as part of the linking command. The file here should be the same for both targets.
<code>rootfs</code>	<code><path>/rootfs.cpio</code>	<code><path>/rootfs.cpio</code>	Specifies the path to the Root FS file that is required as part of the linking command. The file should be the same for both targets.
<code>enable_aie_debug</code>			Generate debug features for the AI Engine kernels. This can be used in both hardware and emulation builds.
<code>defer_aie_run</code>			The AI Engines will be enabled by the PS application. When unset, generate the CDO commands to enable AI Engines during PDI load instead. Only valid if <code>libadf.a</code> is an input file and the platform is of a Versal platform.
<code>ps_elf</code>	<code><file>,core</code>	<code><file>,core</code>	Used only for bare-metal designs. Automatically programs the PS core to run. For example, <code>host.elf,a72-0</code>
<code>domain</code>	<code>aiengine</code>	<code>aiengine</code>	Specifies the domain to be run. For AI Engine designs, this should always be <code>aiengine</code> .
<code>sd_file</code>	<code><file></code>	<code><file></code>	Copies the ELF for the main application that will run on the Cortex-A72 processor for bare metal, and any files needed to run on Linux. The XCLBIN file is automatically copied to the <code>out-dir</code> or <code>sd_card</code> folder. To have more files copied to the <code>sd_card</code> folder, you must specify this option multiple times.

Notes:

1. The `xilinx_vck190_v202120_1` platform does not support the `qspi` option. Custom platforms that are configured to support it will work.

The following table shows the output defined by `-out-dir` produced when building for both hardware and hardware emulation.

Table 88: Table of Outputs

Build	Output
Hardware	<pre> -- BOOT.BIN -- boot_image.bif -- sd_card -- BOOT.BIN -- boot.scr -- aie_graph.xclbin -- host.exe -- Image -- init.sh -- platform_desc.txt -- sd_card.img </pre>
Hardware Emulation	<pre> -- BOOT_bh.bin //Boot header -- BOOT.BIN //Boot File -- boot_image.bif -- launch_hw_emu.sh //Hardware emulation launch script -- libadf //AIE emulation data folder -- cfg -- aie.control.config.json -- aie.partial.aiecompile_summary -- aie.shim.solution.aiesol -- aie.sim.config.txt -- aie.xpe -- plm.bin //PLM boot file -- pmc_args.txt //PMC command argument specification file -- pmc_cdo.bin //PMC boot file -- qemu_args.txt //QEMU command argument specification file -- sd_card -- BOOT.BIN -- boot.scr -- aie_graph.xclbin -- host.exe -- Image -- init.sh -- platform_desc.txt -- sd_card.img -- sim //Vivado simulation folder </pre>

For hardware emulation, the key output file is the `launch_hw_emu.sh` script used to launch emulation. The `sd_card.img` image includes the `BOOT.BIN` (U-Boot to boot Linux, PDI boot data, etc.), Image (kernel image), XCLBIN file, user application (`host.exe`), and other files. For example, all generated files are placed in a folder called `emulation`.

To use the `sd_card.img` file on a Linux host, use the `dd` command to write the image to the SD card. If you are targeting Linux but with `package.image_format=fat32`, copy the `sd_card` folder to an SD card formatted for FAT32. This is not needed for hardware emulation.



TIP: The PS host application is included in the `sd_card` output, however, it is not incorporated into the `rootfs`. If you want to include the executable images in `rootfs`, you must rebuild the `rootfs` before running the `v++ --package` command.

If the design needs to be programmed to a local flash memory, make sure `--package.boot_mode qspi` is used. This allows the use of the `program_flash` command or the use of the Vitis IDE to program the device or program the flash memory, described in [Chapter 15: Using the Vitis IDE](#).

Building a Bare-metal System

Building a bare-metal system requires a few additional steps from the standard application flow previously described. The specific steps required are described here.

1. Build the bare-metal platform.

Building bare-metal applications requires a bare-metal domain in the platform. The base platform `xilinx_vck190_base_202120_1` does not have a bare-metal domain, which mean you must create a platform with one. Starting from the `v++` linking process as described in [Linking the System](#), you must create a custom platform because the PS application needs drivers for the PL kernels in the design.

Use the XSA generated during the link process to create a new platform using the following command:

```
generate-platform.sh -name vck190_baremetal -hw <filename>.xsa \
                    -domain psv_cortexa72_0:standalone
```

where:

- `-name vck190_baremetal`: Specifies a name for the platform that will be created. The platform will be created according to the specified name. In this example it will be written to: `./vck190_baremetal/export/vck190_baremetal`
- `-hw <filename>.xsa`: Specifies the name of the input XSA file generated during the `v++ --link` command. The `<filename>` will be the same as the file name specified for the `.xclbin` output.
- `-domain psv_cortexa72_0:standalone`: Specifies the processor domain and operating system to apply to the new platform.

You can add the new platform to your platform repository by adding the file location to your `$PLATFORM_REPO_PATHS` environment variable. This makes it accessible to the Vitis IDE for instance, or allows you to specify the platform in command-lines by simply referring to the name rather than the whole path.

Note: The generated platform will be used only for building the bare-metal PS application and is not used any other places across the flow.

2. Compile and link the PS application.

To build the PS application for the bare-metal flow, use the platform generated in the prior step. You need the PS application (`main.cpp`), and the bare-metal AI Engine control file (`aie_control.cpp`), which is created by the `aiecompiler`. To build the PS command and can be found in the `./Work/ps/c_rts` folder.

Compile the `main.cpp` file using the following command:

```
aarch64-none-elf-gcc -I.. -I. -I../src \
-I./vck190_baremetal/export/vck190_baremetal/sw/vck190_baremetal/
standalone_domain/bspinclude/include \
-g -c -std=c++11 -o main.o main.cpp
```

Note: You must include the BSP include files for the generated platform, located at: `./vck190_baremetal/export/vck190_baremetal/sw/vck190_baremetal/standalone_domain/bspinclude/include`

Compile the `aie_control.cpp` file using the following command:

```
aarch64-none-elf-gcc -I.. -I. -I../src \
-I./vck190_baremetal/export/vck190_baremetal/sw/vck190_baremetal/
standalone_domain/bspinclude/include \
-g -c -std=c++11 -o aie_control.o ../Work/ps/c_rts/aie_control.cpp
```

Link the PS application using the two compiled object files:

```
aarch64-none-elf-gcc main.o aie_control.o -g -mcpu=cortex-a72 -Wl,-T -
Wl,./lscript.ld \

-L./vck190_baremetal/export/vck190_baremetal/sw/vck190_baremetal/
standalone_domain/bsplib/lib \
-ladf_api -Wl,--start-group,-lxil,-lgcc,-lc,-lstdc++,--end-group -o
main.elf
```

Note: You also need the BSP `libxil.a` located at `./vck190_baremetal/export/vck190_baremetal/standalone_domain/bsplib/lib` during linking. Here the assumption is the AI Engine are enabled during the platform management controller (PMC) boot.

3. Package the System

Finally, you must run the package process to generate the final boot-able image (PDI) for running the design on the bare-metal platform. This command produces the SD card content for booting the device and running the application. Refer to [Packaging](#) for more information. This requires the use of the `v++ --package` command as follows:

```
v++ -p -t hw \
-f xilinx_vck190_base_202120_1 \
libadf.a project.xclbin \
--package.out_dir ./sd_card \
--package.domain aiengine \
--package.defer_aie_run \
--package.boot_mode sd \
--package.ps_elf main.elf,a72-0 \
-o aie_graph.xclbin
```



TIP: For bare-metal ELF files running on PS cores, you should also add the `package.ps_elf` option to the `--package` command.

The use of `--package.defer_aie_run` is related to the way the AI Engine graph is run. If the application is loaded and launched at boot time, these options are not required. If your host application launches and controls the graph, then you need to use these options when compiling and packaging your system as described in [Deploying the System](#).

The `./sd_card` folder, specified by the `--out_dir` option, contains the following files produced for the hardware build:

```
|-- BOOT.BIN      //BOOT.BIN file containing PDI and the application ELF
|-- boot_image.bif  //bootgen input file used to create BOOT.BIN
|-- sd_card        //SD card folder
    |-- aie_graph.xclbin  //xclbin output file (not used)
    |-- BOOT.BIN      //BOOT.BIN file containing PDI and the
application ELF
```

Copy the contents of the `sd_card` folder to an SD card to create a boot device for your system.

Now you have built the bare-metal system, you can run it or debug it.

Running the System in Hardware

Running the system depends on the build target. The process of running the hardware emulation build is different from running the hardware build.

For the hardware build, copy the contents of the `sd_card` folder produced by the package process to an actual SD card. The SD card becomes the boot device for your system. Boot your system and launch your application as designed. To capture event trace data when running the hardware, see [Event Tracing in Hardware](#). To debug the running hardware, see [Chapter 16: Debugging the AI Engine Application](#).

Running Software Emulation

In the typical AI Engine development flow, the steps will be as follows.

1. X86 simulator
2. Software emulation of the system
3. AI Engine simulator
4. Hardware emulation of the system
5. Test and debug on hardware

Generally, software emulation is the first step to building and testing the system in a functional process through the custom host code you create rather than the simulator test host code.

Software emulation for a system with the AI Engine can be useful in:

- Checking initial system behavior with a limited known data set
- Functional integration and debugging of PS, PL, and ADF graph using GDB
- Testing the system with external traffic generator using Python, or C++
- Running system with C-based models for RTL kernels
- Applying AI Engine simulation options through the `x86.options` file in `Work/options`



IMPORTANT! *Software emulation requires all PL kernels to be HLS based, or contain C/C++ models.*

Note:

- When writing host code to execute in software emulation, make sure all buffer objects (`xrtBO`) are synced with `xrtBOSync` API call.
- To run free running PL kernels more than once, you can run it inside a while (1) loop.

To build the project for software emulation confirm that the target option of the `aiecompiler` is `-target=x86sim`. The `v++` link and `v++` package command-s are set to `-target=sw_emu`. The `v++` package command generates the `launch_sw_emu.sh` script as part of the process for packaging the system. This script launches the QEMU emulation environment for the AI Engine application for test and debug purposes. Software emulation runs separate x86 processes for the PL kernels and AI Engine kernels, and runs QEMU for the PS host application.

Use the following command to run software emulation:

```
./launch_sw_emu.sh
```



IMPORTANT! *For HLS free running kernels, the HLS kernel code needs to be changed to run indefinitely in software emulation. Refer to Free Running Kernels in Vitis Unified Software Platform Documentation: Application Acceleration Development ([UG1393](#)).*

When launching software emulation, you can specify options for the `x86simulator` simulator that runs the graph application. The options can be specified from the `launch_sw_emu.sh` script using the `-x86-sim-options` as described in [Reusing x86 Simulator Options](#).

When the emulation is fully booted and the Linux prompt is up, make sure to set the following environment variables in the QEMU environment.

```
export XILINX_XRT=/usr
export LD_LIBRARY_PATH=/mnt/sd*1:/tmp:$LD_LIBRARY_PATH
export XCL_EMULATION_MODE=sw_emu
```

This ensures that the host application works. Now the application can be executed on the Linux prompt.

Running Hardware Emulation

To build the project for hardware emulation confirm that the target option of the `V++` link command is `target=hw_emu`. Next, the `v++ --package` command generates the `launch_hw_emu.sh` script as part of the process for packaging the system. This script launches the emulation environment for the AI Engine application for test and debug purposes. Hardware emulation runs the AI Engine simulator for the graph application, runs the Vivado logic simulator for the PL kernels, and runs QEMU for the PS host application.

Use the following command to launch hardware emulation from the command line.

```
./launch_hw_emu.sh --graphic-xsim
```

Note: The `--graphic-xsim` switch is optional and launches the Vivado logic simulator window where you can specify what signals from the design you want to view. It does not include internal AI Engine signals. Here, you must click the **Run All** button in the window to continue execution.

The `launch_hw_emu.sh` script launches QEMU in system mode, and loads and runs the AI Engine application, running the PL kernels in the Vivado simulator. If the emulation flow completes successfully, at the end of the emulation you should see something like the following:

```
[LAUNCH_EMULATOR] INFO: 09:44:09 : PS-QEMU exited
[LAUNCH_EMULATOR] INFO: 09:44:09 : PMU/PMC-QEMU exited
[LAUNCH_EMULATOR] INFO: 09:44:09 : Simulation exited
pmu_path /scratch/aie_test1/hw_emu_pmu.log
pl-sim_dir /scratch/aie_test1/sim/behav_waveform/xsim
Please refer PS /simulate logs at /scratch/aie_test1 for more details.
DONE!
INFO: Emulation ran successfully
```

When launching hardware emulation, you can specify options for the AI Engine simulator that runs the graph application. The options can be specified from the `launch_hw_emu.sh` script using the `-aie-sim-options` as described in [Reusing AI Engine Simulator Options](#).

When the emulation is fully booted and the Linux prompt is up, make sure to set the following environment variables in the QEMU environment.

```
export XILINX_XRT=/usr
export LD_LIBRARY_PATH=/mnt/sd*1:
export XCL_EMULATION_MODE=hw_emu
```

This ensures that the host application works. Note that this also must be done when running on hardware.

Generating Traffic for Emulation

This section describes how to provide input and capture the output from the AI Engine array in hardware and software emulation using AXI traffic generators. In the AI Engine simulator the input data stimulus was provided using the simulation platform construct.

```
PLIO("DataIn", adf::plio_32_bits, "data/input.txt")
```

For hardware and software emulation an equivalent feature exists that emulates the behavior of this PLIO and AXI4-Stream interface. Both Python and C++ APIs are provided to make this easier to use.

The primary external data interfaces for the AI Engine array are AXI4-Stream interfaces. These are known as PLIOs and allow the AI Engine to receive data, operate on the data, and send data back on a separate AXI4-Stream interface. The input interface to the AI Engine is an AXI4-Stream slave and the output is an AXI4-Stream master. To interact with these top level interfaces during hardware and software emulation complementary AXI4-Stream modules are provided. These complementary modules are referred to as the AXI traffic generators.

Note: The width of a PLIO interface is an important system level design decision. The wider the interface the more data can be sent per PL clock cycle.

AXI Traffic Generators

The AXI Traffic generators are provided as XO files which need to be linked to your simulation platform using the Vitis compiler (v++). These XO files are called

`sim_ipc_axis_master_XY.xo` and `sim_ipc_axis_slave_ZW.xo` where XY and ZW correspond to the number of bits in the PLIO interface. For example

`sim_ipc_axis_master_128.xo` provides an AXI4-Stream master data bus that is 128 bits wide. A wider interface allows the PL to achieve the same throughput at a lower clock frequency and allows the AI Engine array to maximize its memory bandwidth. However, the PLIO interface tiles are each 64 bits wide and they are a limited resource. Using one 64-bit PLIO interface at twice the clock speed provides an equivalent bandwidth to a 128-bit PLIO while using only one PLIO tile. This requires the PL to run at twice the clock speed and the optimal choice will vary from application to application.

Two steps are required to use the traffic generators with the Vitis compiler. First, make the connections between the `sim_ipc` modules and their corresponding AXI4-Stream ports on the AI Engine array. This is typically done in the `system.cfg` file. Here is an example:

```
[connectivity]
nk=sim_ipc_axis_master:1:inst_sim_ipc_axis_master
nk=sim_ipc_axis_slave:1:inst_sim_ipc_axis_slave
stream_connect=sim_ipc_axis_master.M00_AXIS:ai_engine_0.DataIn
stream_connect=ai_engine_0.DataOut:sim_ipc_axis_slave.S00_AXIS
```

The syntax for connecting the `sim_ipc_axis` XO files is as follows.

```
nk=sim_ipc_axis_master:<Number Of Masters>:<your_instance_name_1>
nk=sim_ipc_axis_slave:<Number Of Slaves>:<your_instance_name_2>
```

The `sim_ipc_axis_master/slave` specifies the type of XO file and the instance name should be meaningful to your application.

Next, add the XO files to the Vitis link command. Note the `sim_ipc` XO files can only be used with target `hw_emu`.

```
v++ -l --platform <platform.xpfm> sim_ipc_axis_master_128.xo
sim_ipc_axis_slave_128.xo libadf.a -target hw_emu --config system.cfg
```

For additional information on how to use XO files with the Vitis compiler see https://github.com/Xilinx/Vitis-Tutorials/tree/master/AI_Engine_Development/Feature_Tutorials/05-AI-engine-versal-integration.

Note: To use multiple AXI4-Stream masters at the same time change the `Number of Masters` field in the `system.cfg` file from 1 to as many as needed (up to 8).

Formatting Data with Traffic Generators in Python

To emulate AXI4-Stream transactions AXI Traffic Generators require the payload data to be broken into appropriately sized bursts. For example, to send 128 bytes with a PLIO width of 32 bits (4 bytes) requires 128 bytes/4 bytes = 32 AXI4-Stream transactions. Converting between bytes arrays and AXI transactions can be handled in Python.

The Python `struct` library provides a mechanism to convert between Python and C data types. Specifically, the `struct.pack` and `struct.unpack` functions pack and unpack byte arrays according to a format string argument. The following table shows format strings for common C data types and PLIO widths.

For more information see: <https://docs.python.org/3/library/struct.html>

Table 89: Format Strings for C Data Types and PLIO Widths

Data Type	PLIO Width	Python Code Snippet
cfloat	PLIO32	N/A
	PLIO64	<code>rVec = np.real(data)</code> <code>iVec = np.imag(data)</code>
	PLIO128	<code>out2column = np.zeros((L,2)).astype(np.single)</code> <code>out2column.tobytes()</code> <code>formatString = "<" + str(len(byte_array)//4) + "f"</code>
cint16	PLIO32	<code>rVec = np.real(data).astype(np.int16)</code>
	PLIO64	<code>iVec = np.imag(data).astype(np.int16)</code>
	PLIO128	<code>formatString = "<" + str(len(byte_array)//2) + "h"</code>

Table 89: Format Strings for C Data Types and PLIO Widths (cont'd)

Data Type	PLIO Width	Python Code Snippet
int8	PLIO32	<pre>intvec = np.real(data).astype(np.int8) formatString = "<" + str(len(byte_array)//1) + "b"</pre>
	PLIO64	
	PLIO128	
int32	PLIO32	<pre>intvec = np.real(data).astype(np.int32) formatString = "<" + str(len(byte_array)//4) + "i"</pre>
	PLIO64	
	PLIO128	

The remaining aspects of Python libraries, interacting with the `sim_ipc` Python object and providing and receiving data are beyond the scope of this document.

A significant benefit of this feature is that it enables you to integrate your AI Engine design with a larger system while also minimizing the amount of PS code required. This is useful during development where not all domains of the system are ready to integrate.

Because the data source and sink are kept completely within the simulated PL domain the host only needs to provide setup and control functionality. For example the `main` in a minimal `host.cpp` might look like the following.

```
int main(int argc, char ** argv)
{
    //////////////////////////////////////
    // Open xclbin
    //////////////////////////////////////
    auto dhdl = xrtDeviceOpen(0); // Open Device the local device
    if(dhdl == nullptr)
        throw std::runtime_error("No valid device handle found. Make sure
using right xclOpen          index.");
    auto xclbin = load_xclbin(dhdl, "a.xclbin");
    auto top = reinterpret_cast(xclbin.data());
    adf::registerXRT(dhdl, top->m_header.uuid);

    //////////////////////////////////////
    // graph execution for AIE
    //////////////////////////////////////

    printf("graph init.\n");
    mygraph_top.init();

    printf("graph run\n");
    mygraph_top.run(1);

    mygraph_top.end();
    printf("graph end\n");
    xrtDeviceClose(dhdl);

    return 0;
}
```

Deploying the System

The Vitis design execution model has multiple considerations that impact how the AI Engine graph is loaded onto the board, run, reset, and reloaded. Depending on the needs of the application you have a choice of loading the AI Engine graph at board boot up time, or using the PS host application. In addition, you can also control running the graph as soon as the graph is loaded or defer it to a later time. You also have the option of running the graph infinitely or for a fixed number of iterations or cycles.

AI Engine Graph Load and Run

The AI Engine graph can be loaded and run immediately at boot, or it can be loaded by the host PS application. Additionally, you also have the option of deferring the running of the graph after the graph has been loaded using the `graph.run()` host API XRT call. By default, the Xilinx® platform management controller (PMC) loads and runs the graph. However the `v++ --package.defer_aie_run` option will let you defer the graph run until after the graph has been loaded using the `graph.run()` API call. The following table lists the deployment options.

Table 90: Deploying the AI Engine Graph

Host Control	Run Forever
Specify <code>v++ --package.defer_aie_run</code> to stop the AI Engine from starting at boot-up.	Enable it in the PDI and let the graph run forever.
Enable the graph from the PS program using <code>graph.run()</code>	

AI Engine Run Iterations

The AI Engine graph can run for a limited number of iterations or infinitely. By default, the graph runs infinitely. You can use the `graph.run(run_iterations)` or `graph.end(cycles)` to limit the number of graph runs to a specific number of iterations or for a specific number of cycles. See [Chapter 5: Run-Time Graph Control API](#).

Using the Vitis IDE

The Vitis™ tool flow, as described in [Chapter 14: Integrating the Application Using the Vitis Tools Flow](#), is also available in the Vitis IDE. The different steps involved in building the system project, with an AI Engine graph, PL kernels, and PS application, are described in the following sections.

Before using the Vitis IDE, you must first set up the development environment, as described in [Setting Up the Vitis Tool Environment](#).

Creating the AI Engine Graph Project and Top-Level System Project

The Vitis integrated development environment (IDE) is available for building, running, and debugging AI Engine programs. This section provides screen captures that correspond to the command-line steps previously described.

1. Start the Vitis IDE by entering `vitis` from the command line. When starting for the first time, you must specify a workspace directory to store your projects. You can also specify the workspace when launching the tool:

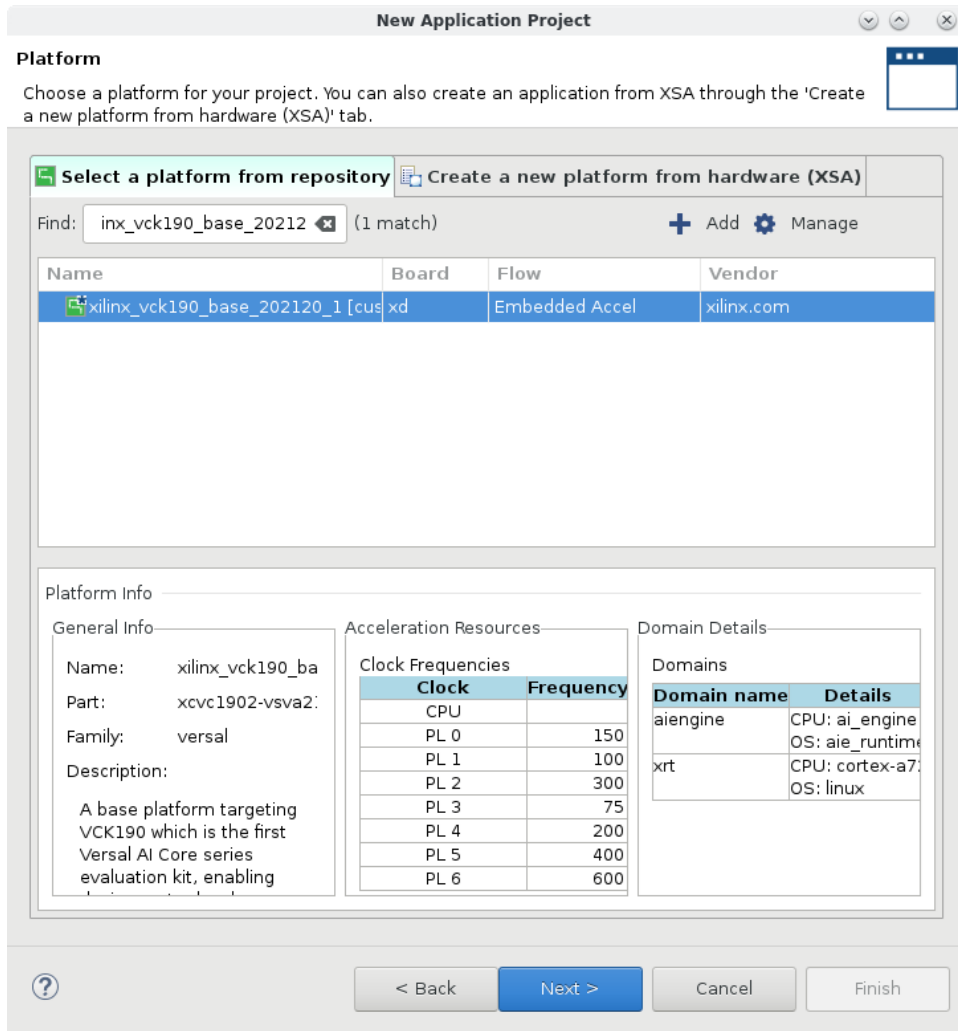
```
vitis -workspace ./myWorkspace
```

2. Create a new AI Engine project by selecting **File → New → Application Project**.



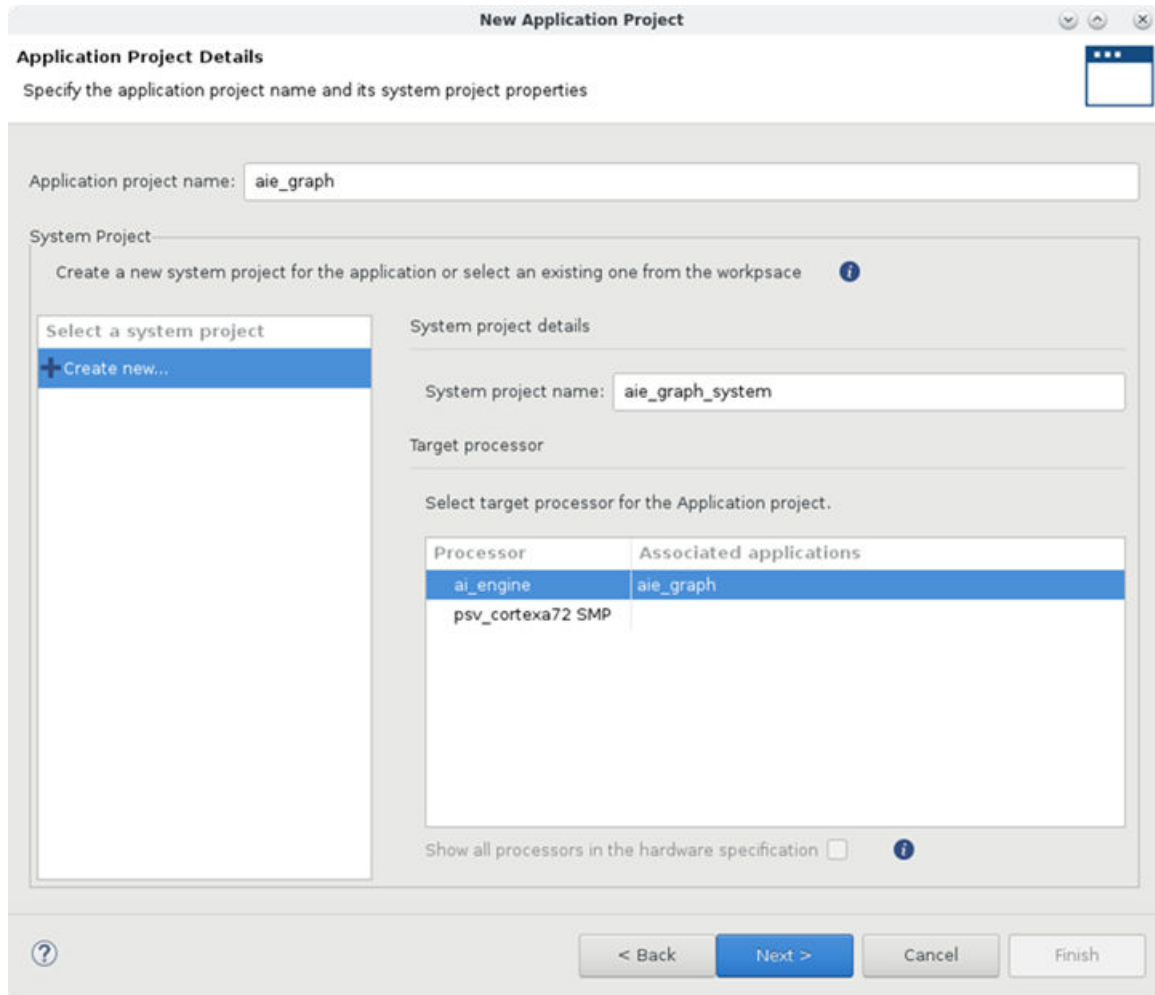
TIP: If this is your first time launching the Vitis IDE, a Welcome screen opens. In this case, click on **Create Application Project**.

The New Application Project wizard opens with an introductory page that describes the flow used to create a new project. Select **Next** to open the Platform selection page as shown in the following figure.



Select the `xilinx_vck190_base_202120_1` platform and click **Next**.

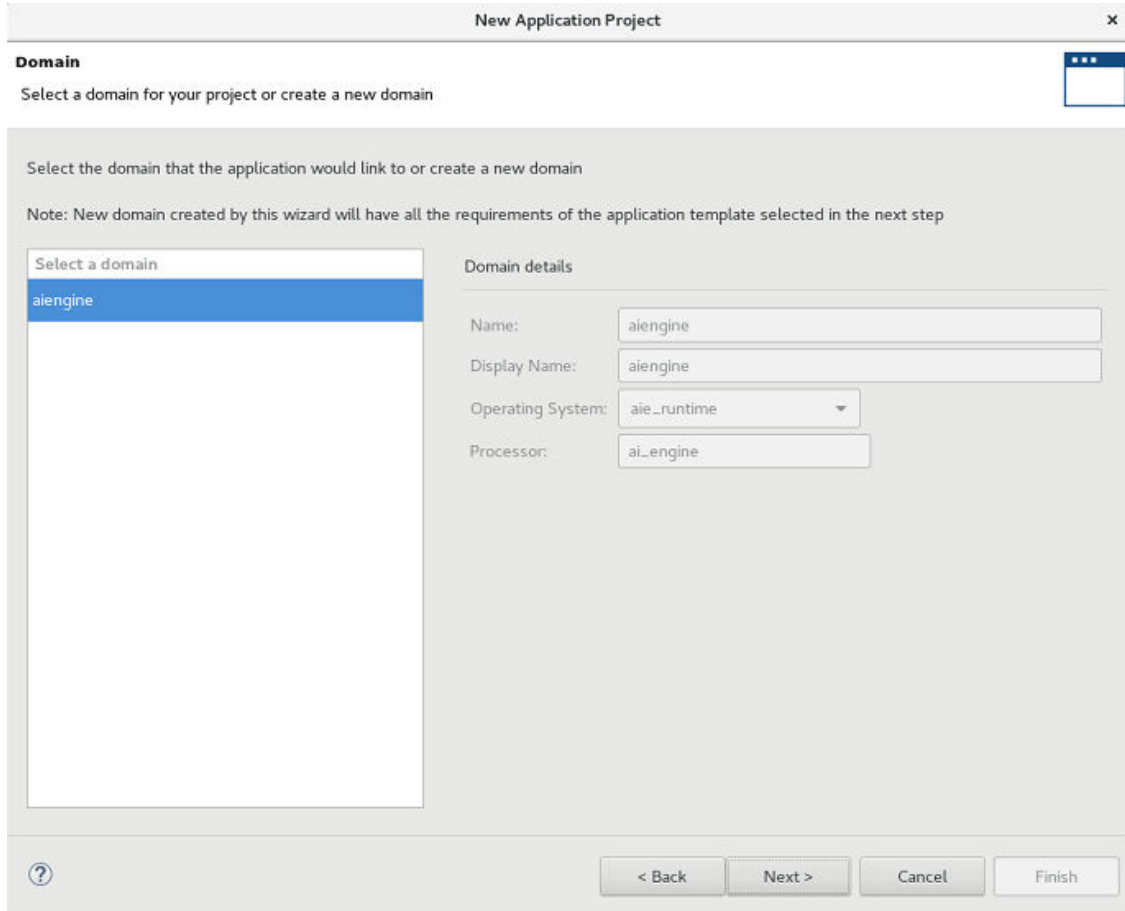
- The Application Project Details page opens as shown in the following figure.



Here you specify the **Application project name**. Under System project, you can select an existing system project to add your application project to (if one exists), or let the Vitis IDE create a new system project. When creating a new system project, the name is automatically generated based on the application project name you specified. However, you can enter a new name in the **Project name** field.

Next you specify the Processor to associate with your new project. In this example an AI Engine graph project is being created. Select the `ai_engine` Processor, and click **Next**.

4. The Domain page opens as shown in the following figure.



New Application Project

Domain

Select a domain for your project or create a new domain

Select the domain that the application would link to or create a new domain

Note: New domain created by this wizard will have all the requirements of the application template selected in the next step

Select a domain

- aiengine

Domain details

Name: aiengine

Display Name: aiengine

Operating System: aie_runtime

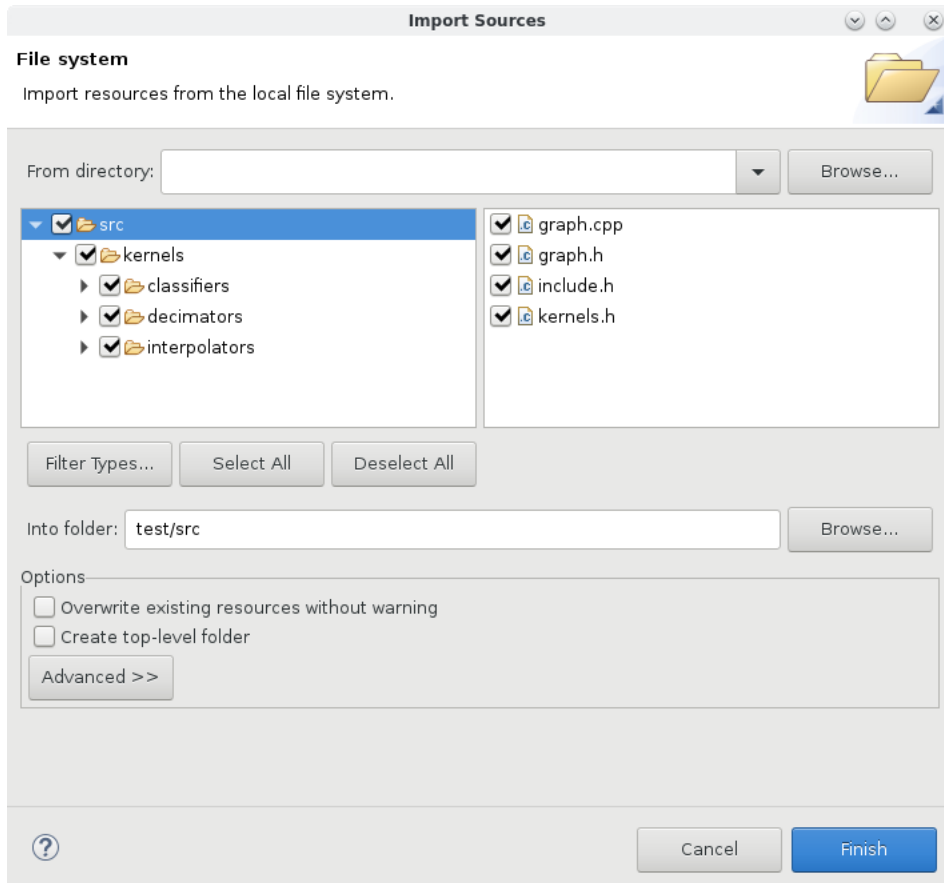
Processor: ai_engine

< Back Next > Cancel Finish

On the Domain selection page you can specify a processor domain for your new application project. In this example, because the `aiengine` Processor is already specified, there is only one domain listed and it is preselected. Click **Next** to proceed.

5. The Templates page opens, as described in [Vitis Tools Template Examples](#).

In this example you are creating a new custom project. Select **Empty Application** template (default), and click **Finish** to create the project. The new AI Engine graph project is created and the Vitis IDE opens the project in the Design perspective.



IMPORTANT! The key to this dialog box is to ensure the *Into folder* field points to the `src` folder of your graph project, as shown for the `aie_graph` project.

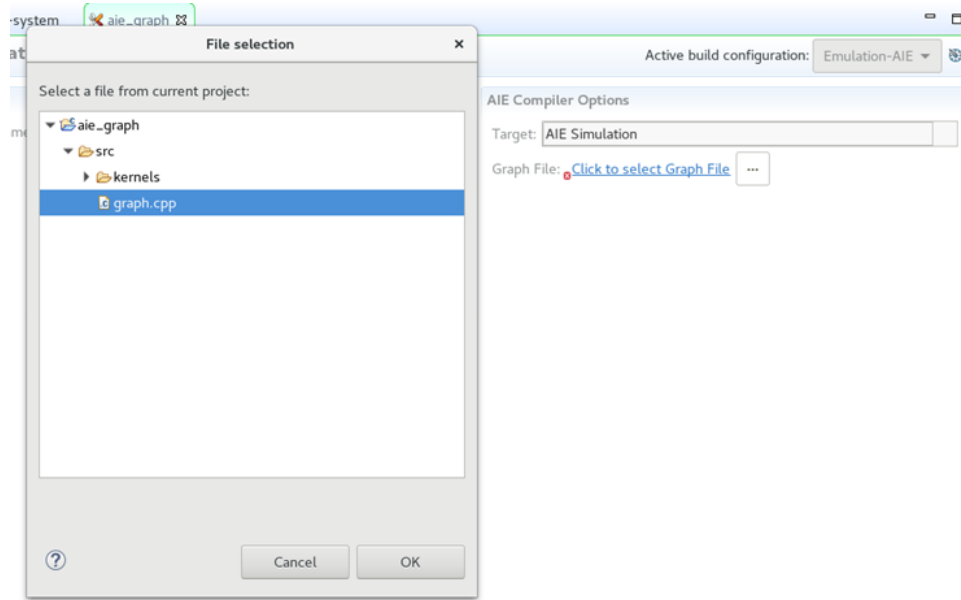
Use the Browse command to navigate to the folder containing the source files for your AI Engine graph. Select the graph and header files required for your graph application, and click **Finish**.

2. Import data files required for your graph application using a similar process. Instead of the `src` folder, import files into the `data` folder of your AI Engine graph project.



TIP: Data files can include input source files to exercise your graph or golden output data files to compare the results of simulation.

3. After importing source and data files, you need to define the top-level of your graph. On the right side of the Project Editor window, click the **Click to select Graph File** link to open the File Selection dialog box as shown in the following figure. Navigate through the application project hierarchy, to the `src` folder and select the C/C++ code file containing your graph application.



The project is now ready to build.

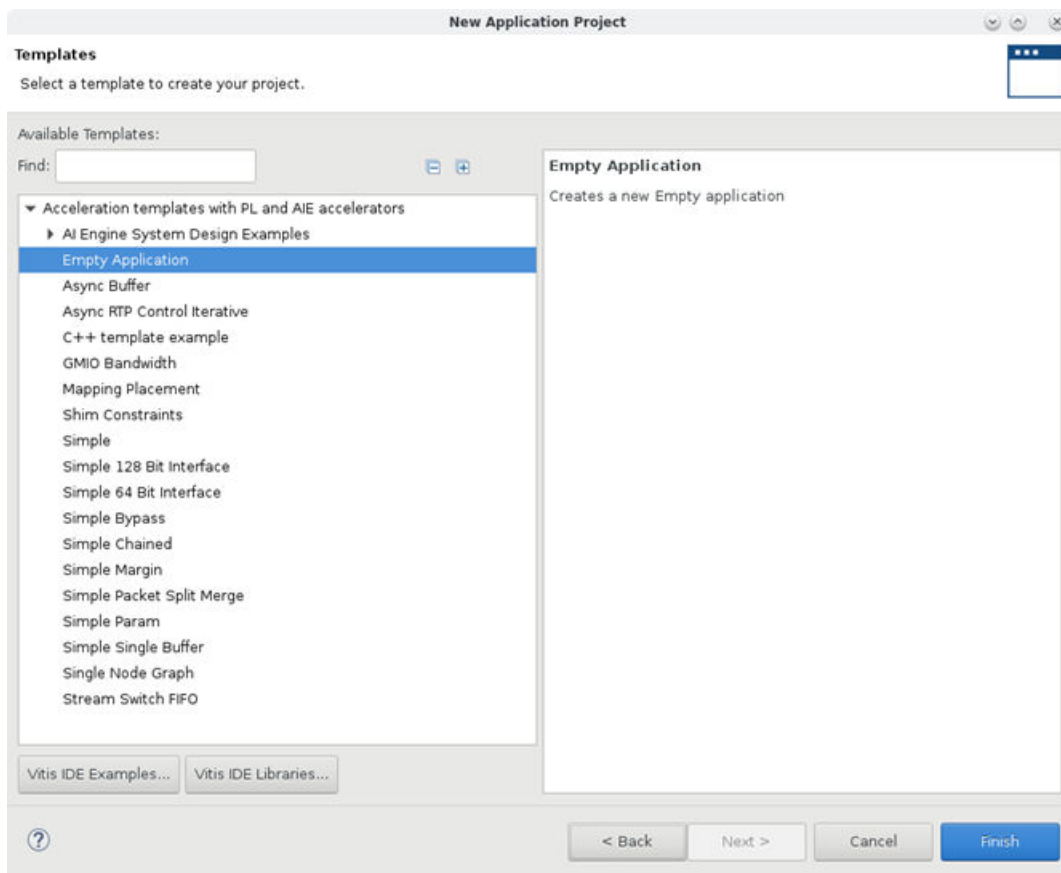


TIP: The previous steps apply to both single kernel and multiple kernel programming. A Pipeline view is available in the Vitis IDE to provide single kernel debug (see [Pipeline View for Single Kernel Debug](#)).

Vitis Tools Template Examples

The last page of the New Application Project wizard in the Vitis IDE (shown in the following figure), displays a list of application templates that can be used for your design. Selecting a template creates a sample AI Engine graph application, and imports the necessary source code to let you build and examine different elements of the application simulation design.

Figure 53: AI Engine Application Templates



The template projects illustrate the basic features of AI Engine programming. You can study these templates, use them as a starting point for your own projects, or mix and match the features to create your own complex computation graphs. The following table describes some of the templates.

Table 91: Application Template Examples

Template Name	Description	Further Information
AI Engine, PL and PS System Design	This design demonstrates integrating the AI Engine array with the Programmable Logic and the Processing System in a system. It performs hardware co-simulation and hardware implementation.	Chapter 14: Integrating the Application Using the Vitis Tools Flow
Async Buffer	A graph to demonstrate asynchronous window APIs.	Asynchronous Window Access
Async RTP Control Iterative	A graph to demonstrate simple use of asynchronous RTP update and run with specified test iterations.	Graph Execution Control
C++ template example	An example demonstrating C++ templated data types and state encapsulation.	C++ Template Support
GMIO Bandwidth	A graph to demonstrate GMIO performance profiling.	GMIO Attributes
Mapping Placement	A templated graph with relocatable mapping and location constraints for kernels.	Location Constraints

Table 91: Application Template Examples (cont'd)

Template Name	Description	Further Information
Shim Constraints	A graph to demonstrate physical channel allocation constraints on the AI Engine to PL interface boundary.	Chapter 7: AI Engine/Programmable Logic Integration
Simple	A simple 2-kernel graph with window based data communication.	Window-based Access
Simple 128 Bit Interface	A graph to demonstrate 128-bit interface between the AI Engine and PL.	PLIO Attributes
Simple 64 Bit Interface	A graph to demonstrate 64-bit interface between the AI Engine and PL.	PLIO Attributes
Simple Bypass	A graph demonstrating the use of bypass for kernels.	Kernel Bypass
Simple Chained	A simple 2-kernel graph with triggered array parameter communication between the kernels.	Chained Updates Between AI Engine Kernels
Simple Margin	A graph demonstrating the use of margin in windows (overlapping windows).	Window-based Access
Simple Packet Split Merge	A graph to demonstrate simple split and merge of packet stream data.	Explicit Packet Switching
Simple Param	A simple 1-kernel graph with scalar parameter update using external trigger.	Specifying Run-Time Data Parameters
Simple Single Buffer	A graph demonstrating single buffer constraint on connections.	Buffer Allocation Control
Single Node Graph	A simple single node graph with demonstration window (single buffer and double buffer), stream and RTP array connections.	Single Kernel Development
Stream Switch FIFO	A graph to demonstrate use of stream switch FIFO to avoid deadlocks with reconvergent streams.	FIFO Depth

Building the AI Engine Graph


Take a look at your project in the Assistant view. It initially includes three projects: the top-level system project that is created by the Vitis IDE when you define an application project, the application project itself, and a hw_link project that links the AI Engine domain and the PL region domain into a single device binary (XCLBIN) or platform file (XSA).


The application project in this case is your AI Engine graph application. When you created the graph project, the Vitis IDE created the top-level system project, and the hw-link project, which is discussed further in [Configuring the HW-Link Project](#).

Expand the AI Engine graph application project to see that it contains three build targets:

- **Emulation-SW:** The functional simulation build target. Compiled by the AI Engine compiler, run in the x86simulator.
- **Emulation-AIE:** The hardware emulation build. Compiled in the AI Engine compiler, and run in the AI Engine SystemC simulator (`aiesimulator`).
- **Hardware:** The hardware build, compiled by the AI Engine compiler for use in the actual device.

All of these build targets are built by the AI Engine compiler tool as described in [Chapter 9: Compiling an AI Engine Graph Application](#), and simulated by the AI Engine simulator or the x86simulator as described in [Chapter 10: Simulating an AI Engine Graph Application](#).

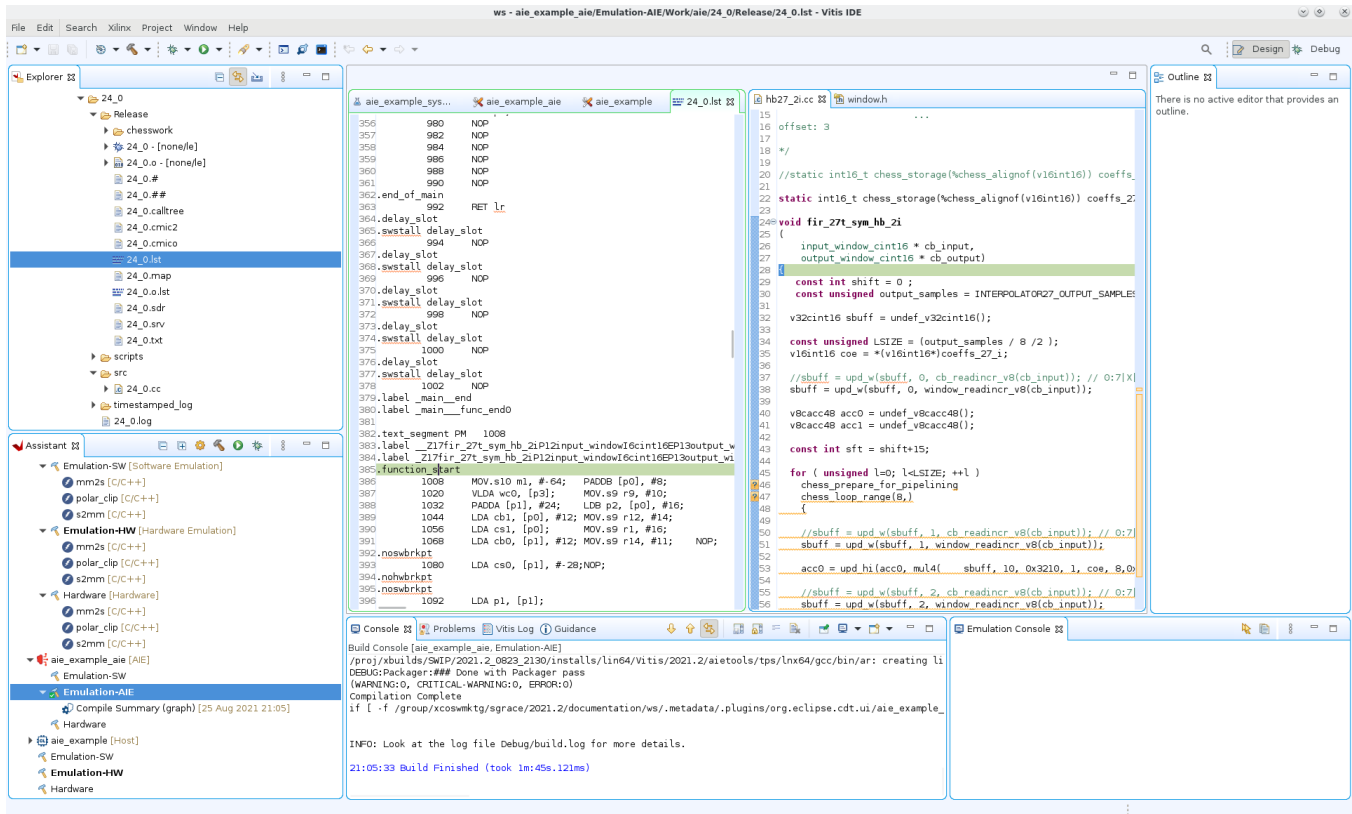
If you select one of these build targets in the **Assistant** view, and select the **Settings** command () , you can see that there are no build settings for the compiler. The Vitis IDE configures the AI Engine compiler as required for the build.

Build the target by making it active in the Assistant view, or the Project Editor view, and click the **Build** icon () .

Viewing Microcode

After the build is complete, you can view the microcode produced by the AI Engine compiler by right-clicking the project in the Explorer view and selecting **Open Disassembly View**. In the **Select Active Core** dialog box, select the core (AI Engine). This opens the LST (`.lst`) file which is microcode that corresponds to the kernel code scheduled to be executed on that AI Engine tile. The kernel code is embedded in the LST file after the `main_end` function. You can select parts of the microcode and cross-probe to the corresponding line in the kernel source file. This allows you to examine the number of cycles taken by specific lines of kernel code and see where further optimizations and efficiencies can be made. In the following image you can see the microcode in the LST file in the center left and the corresponding kernel code file to the right. Clicking in either view shows the corresponding code in the other view.

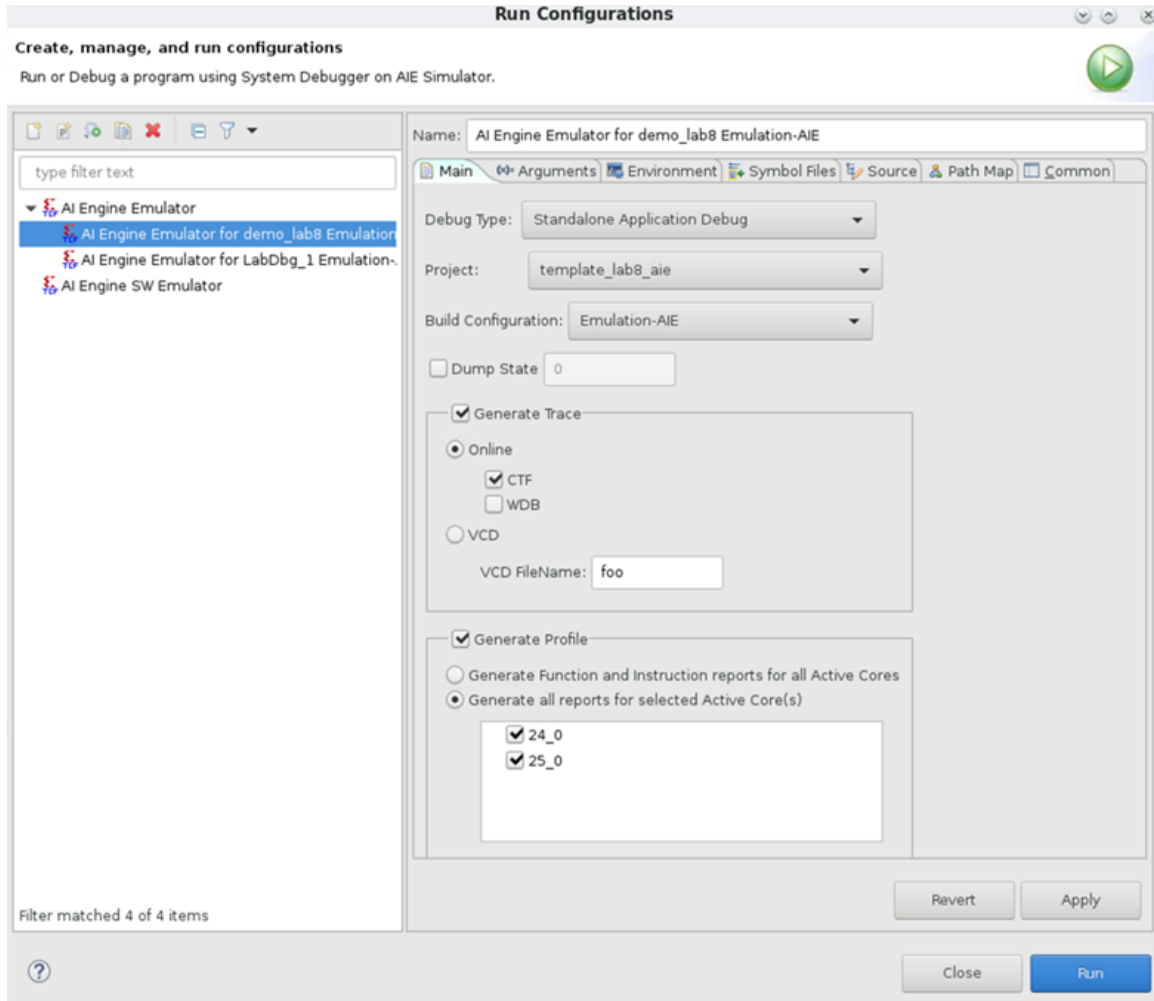
Figure 54: Cross-Probe View



Running and Analyzing the Graph

After the build is successfully finished, you can run the Emulation-SW build, or Emulation-AIE build in the Vitis IDE. You can examine various reports generated by the compiler and simulate your design. For the hardware build, you can copy the SD card output for instance and then use it to boot and run the application on the hardware card.

1. In the Assistant view expand a specific build target, right-click the **Compile Summary** (graph) and select **Open in Vitis Analyzer**. This opens the Compile Summary report as described in [Viewing Compilation Results in the Vitis Analyzer](#).
2. Find additional compiler-generated reports in the Explorer view by navigating from your project and select **Emulation AIE** → **Work** → **Reports**.
3. To run the program for hardware emulation, in the Assistant view click the Run button (🏃) and select **Run Configurations**. This opens the Run Configurations dialog box to create a new run configuration or edit an existing one as shown.



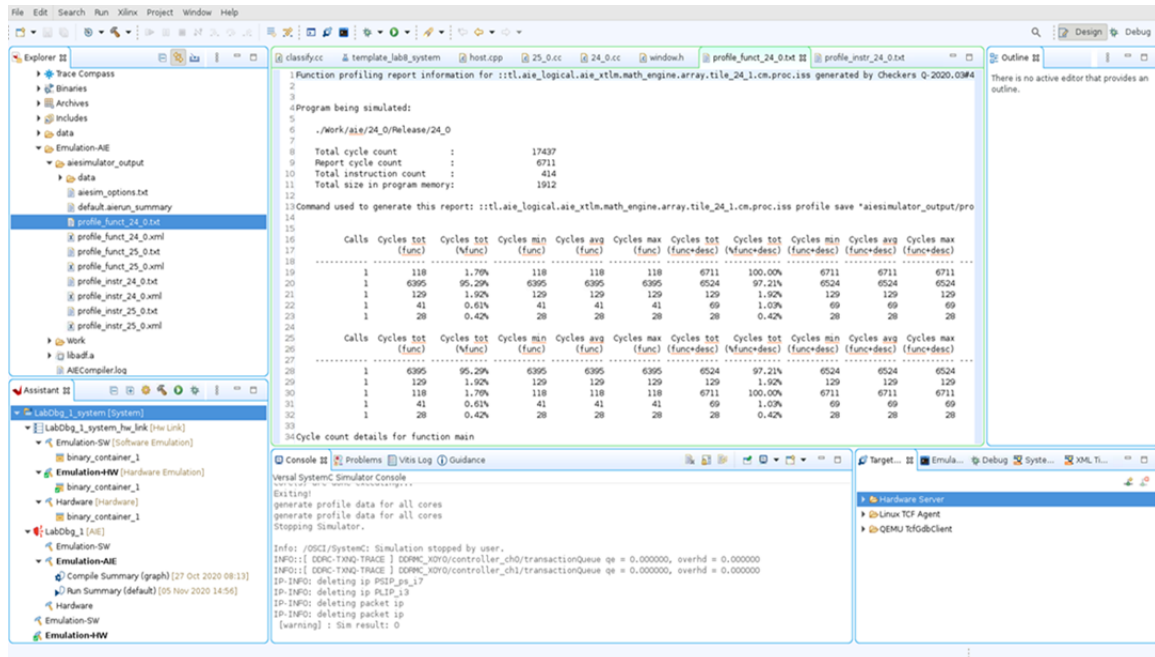
- You can specify a name for the configuration, which allows you to create multiple configurations to apply at different times, or to different build targets as your design flow progresses.
- You can enable **Generate Trace**, and enable event trace for the emulation build using `--dump-vcd` in the AI Engine simulator. See [Chapter 11: Performance Analysis of AI Engine Graph Application during Simulation](#) for more details.
- You can enable **Generate Profile** to specify the `--profile` option in the AI Engine simulator and trigger a profile for all AI Engine processors or selected tiles. Reports are generated in the project `./Emulation-AIE/aiesimulator_output` directory.



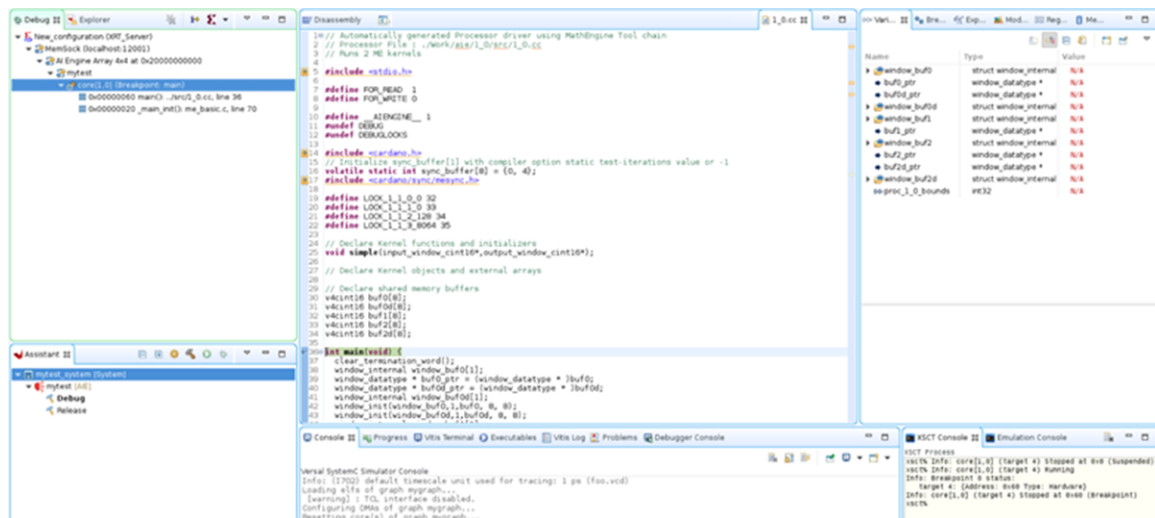
TIP: Clock cycle count reports are generated with this option enabled for the selected tiles.

- To add additional AI Engine simulator options, select the **Arguments** tab and enter the option as you would from the command line.

When ready to run emulation, select **Apply** → **Run**.



4. To debug the program, right-click on the application and select **Debug As → Launch AIE Emulator**. The simulator starts in debug mode with the AI Engines stopping at their respective `main()`. You can set breakpoints, single-step, and resume execution, as well as examine registers, local variables, and memory data structures. See [Hardware Emulation Debug from the Vitis IDE](#) for more information.



Single Kernel Development

To achieve the highest performance on the AI Engine, the primary goal of single kernel programming is to ensure that the use of the vector processor approaches its theoretical maximum. Vectorization of the algorithm is important, but managing the vector registers, memory access, and software pipelining are also required. Because the vector processor is capable of an operation every clock cycle, the programmer must strive to make the data for the next operation load during the current operation. When implementing an algorithm for the AI Engine, it is important to start vectorization based on the data types and the vector intrinsic functions that operate on those data types. Depending on the data type, the various intrinsic functions operate on two or more elements at the same time. When the inner loop has sequential or loop carried dependencies it might be possible to unroll an outer loop and compute multiple values in parallel. There are many creative ways to use the vector intrinsic functions to solve problems. When implementing an algorithm for a Versal® ACAP, it is important to understand what the AI Engine does well and what would be better implemented in the other engines, for example, the Scalar, Adaptable and DSP engines.

To support AI Engine single kernel development, the Vitis IDE supports AI Engine kernel development in addition to traditional processor support. The Vitis IDE provides a single node graph example that can be used as a starting point for single kernel development. The Vitis IDE has a debug view which displays registers, variables, available breakpoints, variables to register/memory mapping, internal/external memory contents, and an instruction pipeline (pipeline view) for each individual kernel.

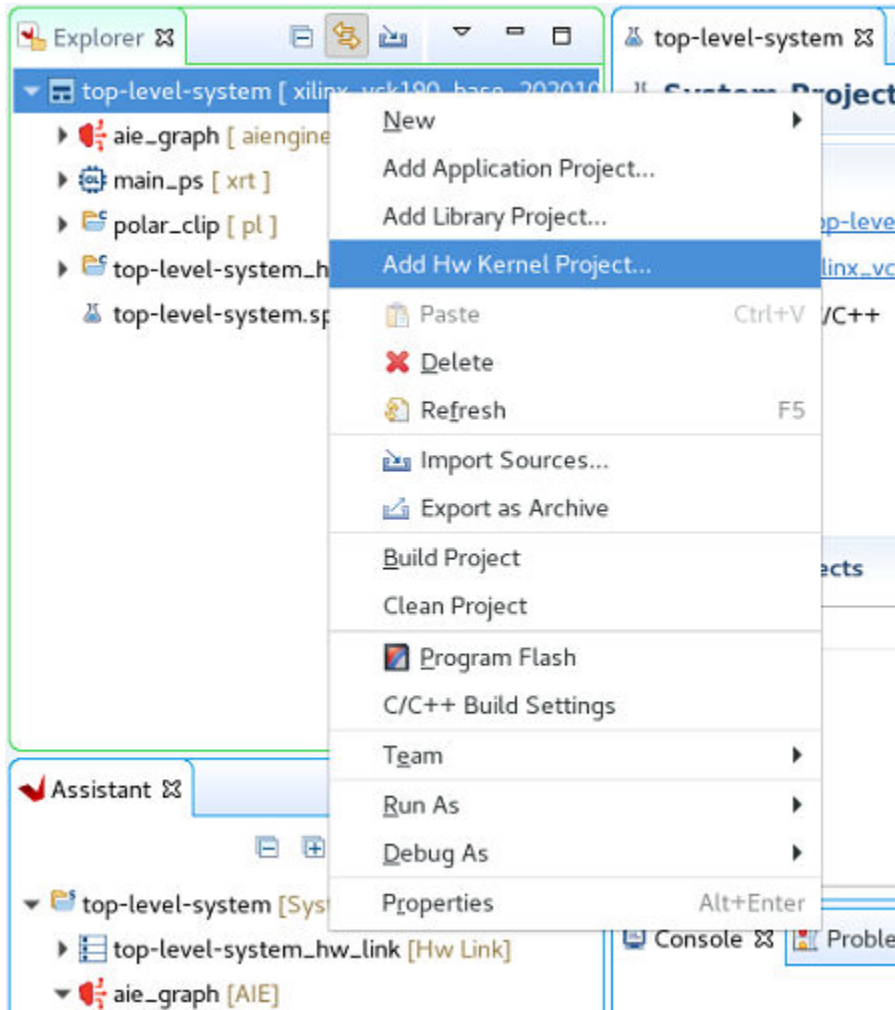


TIP: Clock cycle count reports are generated with the `--profile` option enabled after the emulation run.

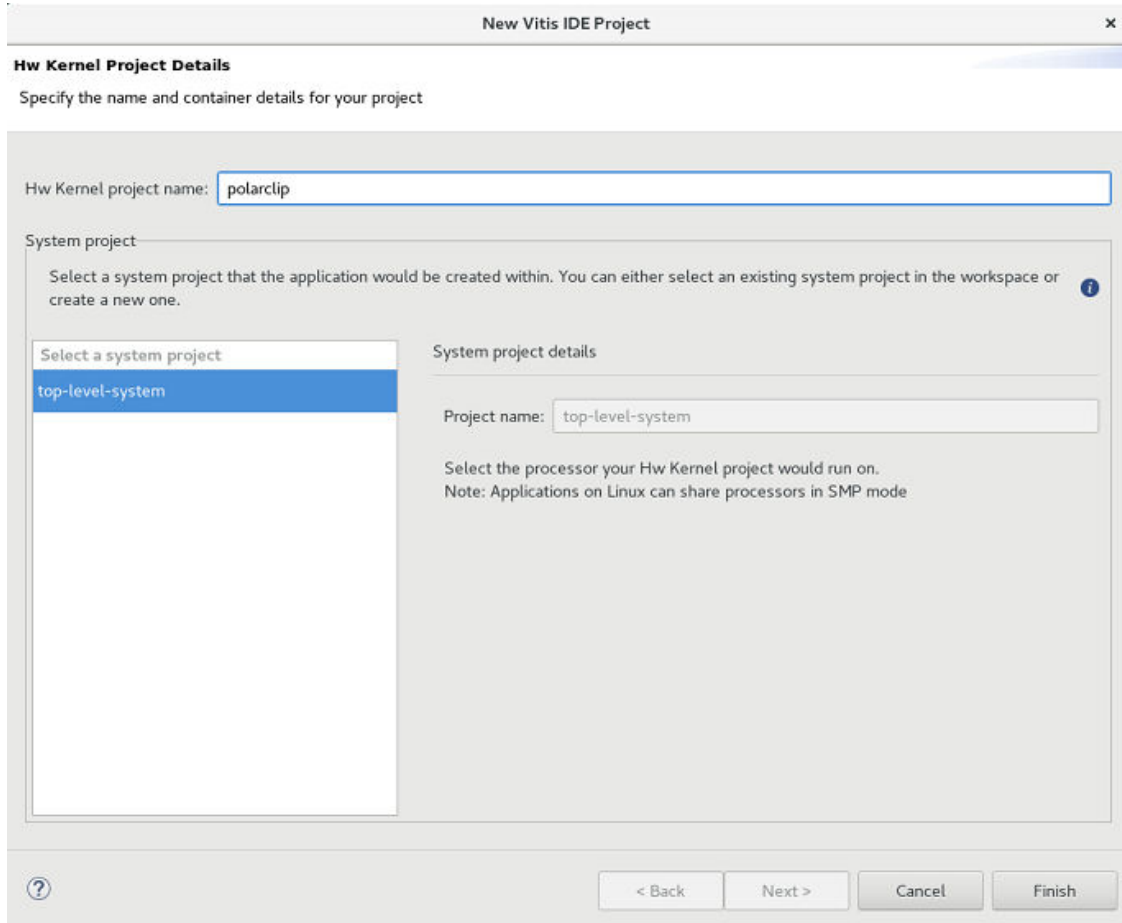
Adding a PL Kernel Project to the System

After creating the AI Engine graph project, which also creates the top-level system project, and a hw-link project, you can create PL kernel projects to add your system. You need to create a PL application project and add it to your system project using the following process.

1. In the Explorer view, select the top-level system project to create a new PL project to add to it. Right-click on the system project and select the **Add Hw Kernel Project** command, as shown in the following figure.



2. This displays the Hw Kernel Project Details page of the New Vitis IDE Project wizard as shown in the following figure.



New Vitis IDE Project

Hw Kernel Project Details

Specify the name and container details for your project

Hw Kernel project name:

System project

Select a system project that the application would be created within. You can either select an existing system project in the workspace or create a new one.

Select a system project

- top-level-system

System project details


Project name:

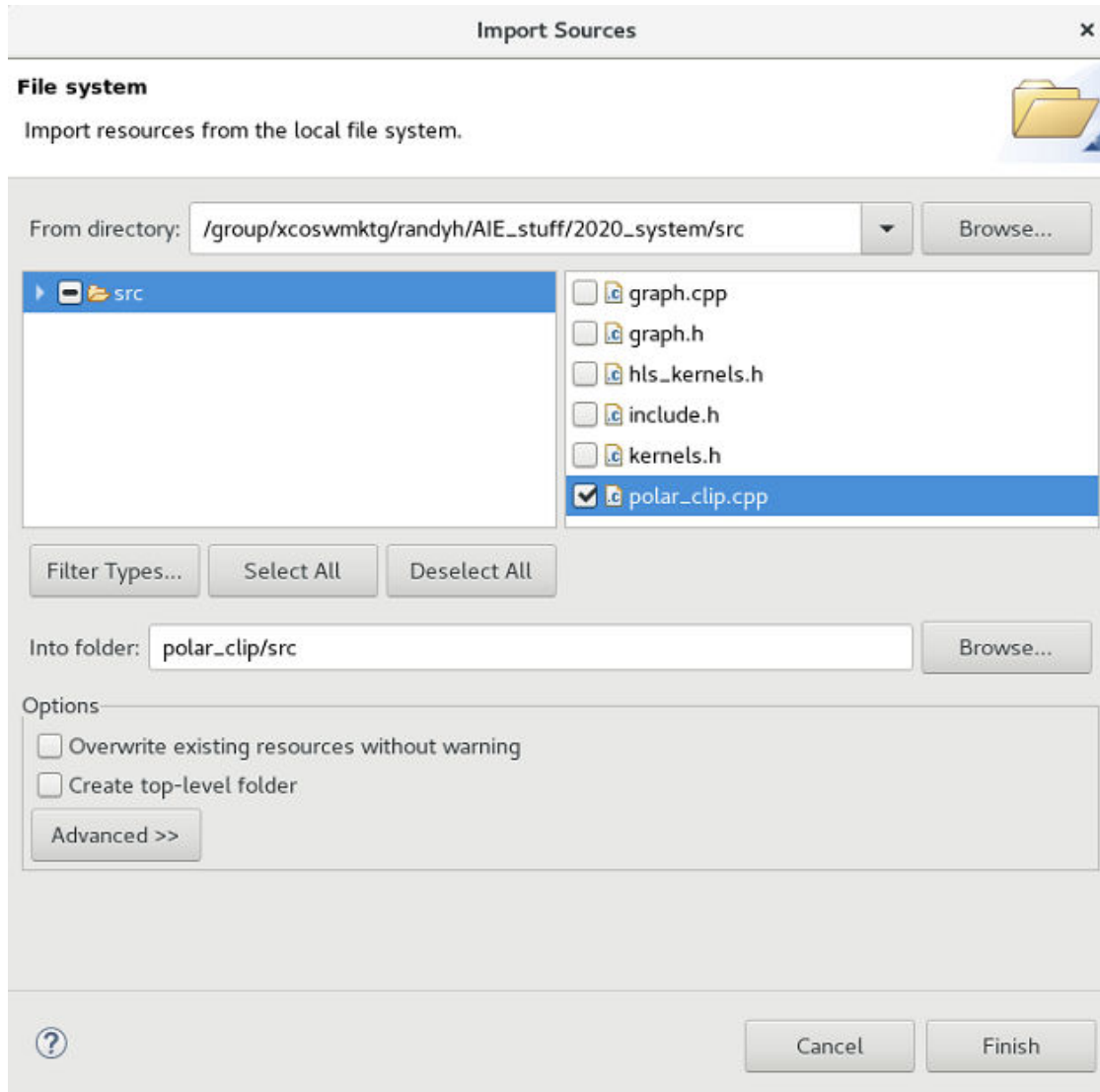
Select the processor your Hw Kernel project would run on.

Note: Applications on Linux can share processors in SMP mode


< Back Next > Cancel Finish

Make sure the project is assigned to the existing system project, which also contains your AI Engine graph project. Specify the HW Kernel project name. Click **Next** to proceed.

- This creates the PL kernel project and adds it to the hierarchy of the top-level system project. Next you must add the source code for your kernel. In the Explorer view, select the `src` folder of the PL kernel and click the **Import Sources** command () to open the dialog box shown in the following figure.



Browse to and select the necessary source files for your PL kernel. Click **Finish** to import the source files to your HW kernel project.

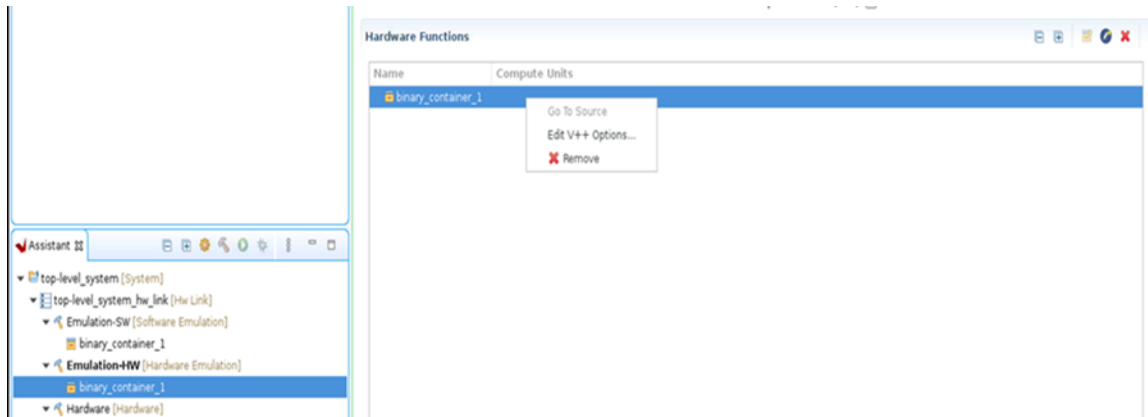
4. With the source files added to the project, you must define the HW function to place into the PL region. In the Project Editor window, select the **Add Hardware Function** command () and specify the name of the function to implement in the PL region.

With the PL kernel project created and the HW function defined, you can build the kernel project for the Emulation-SW, Emulation-HW, or Hardware targets. You can build these targets directly in the PL kernel project or as part of building the top-level system project. The top-level project uses an incremental build approach that recognizes the state of the sub-projects and only rebuilds projects that need to be updated.

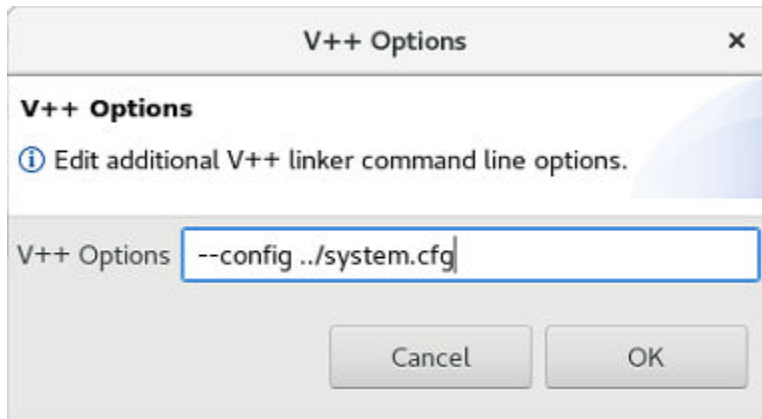
Configuring the HW-Link Project

With the various domain application projects added to the top-level system project, only the connections between the AI Engine graph and the PL kernels need to be defined using the `hw_link` project. This project is automatically generated during the creation of the AI Engine graph project.

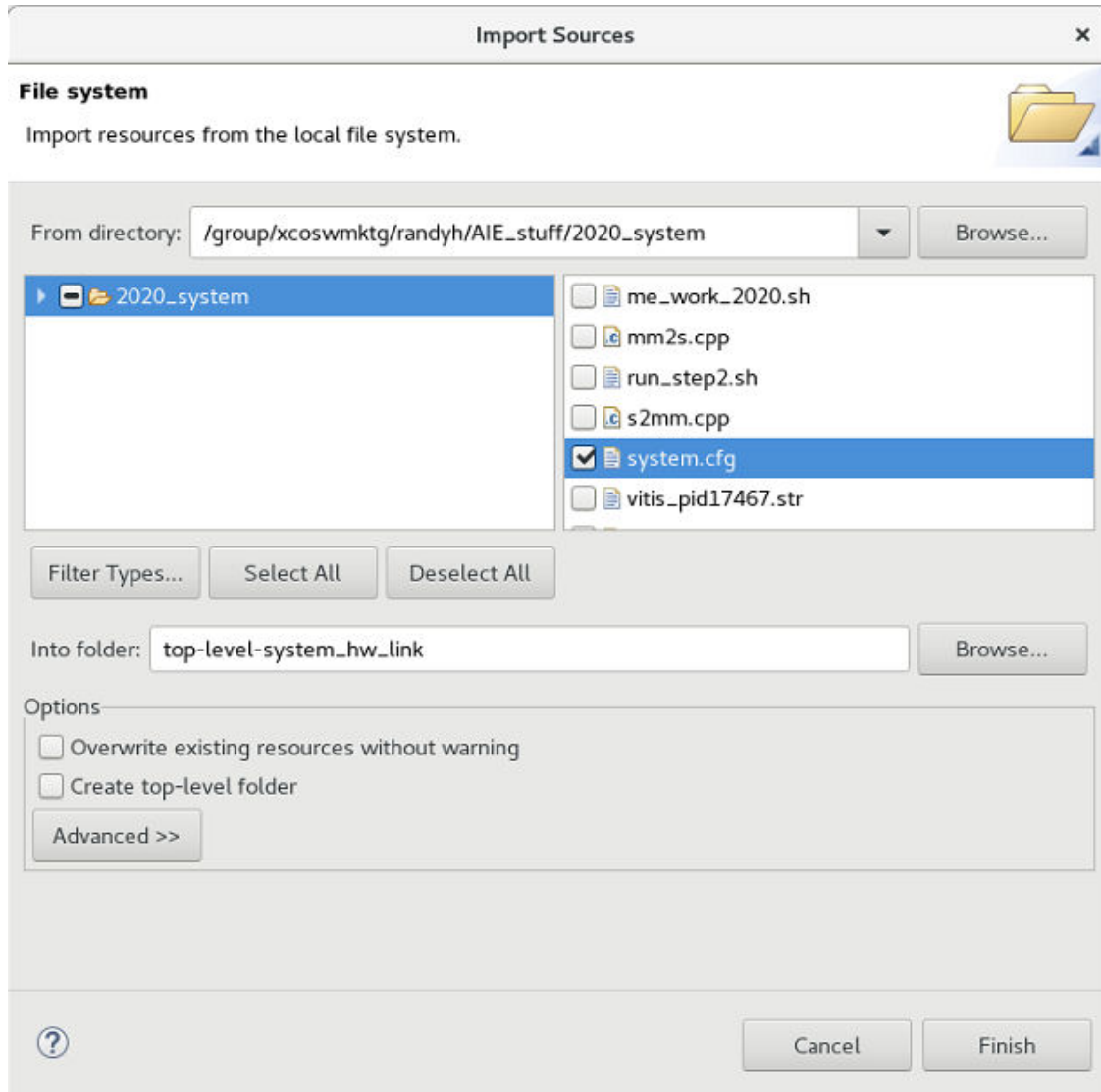
1. Open the `hw_link` project by double-clicking on the project in the **Explorer** view. Right-click on the `binary_container` in the Hardware Functions window, and select the **Edit V++ Options** command.



2. Edit the V++ Options field. It is important to add the correct path to the `config` file because the `v++` command needs it for the Vitis IDE workspace.



3. Import the specified `config` file into the `hw_link` project folder.



TIP: The system configuration file is added to the `hw_link` project folder and not to a `src` folder.

As explained in [Linking the System](#), for AI Engine graph applications the Vitis compiler needs some instruction on how to connect PL kernels to the graph. The number of kernels to be instantiated is already configured in the IDE. However the definition of the connections between the PL kernels and the graph must be specified.

For the Vitis IDE the following is an example configuration file:

```
[connectivity]
stream_connect=mm2s_1.s:ai_engine_0.DataIn1
stream_connect=ai_engine_0.clip_in:polar_clip_1.in_sample
stream_connect=polar_clip_1.out_sample:ai_engine_0.clip_out
stream_connect=ai_engine_0.DataOut1:s2mm_1.s
[advanced]
param=compiler.addOutputTypes=hw_export
```

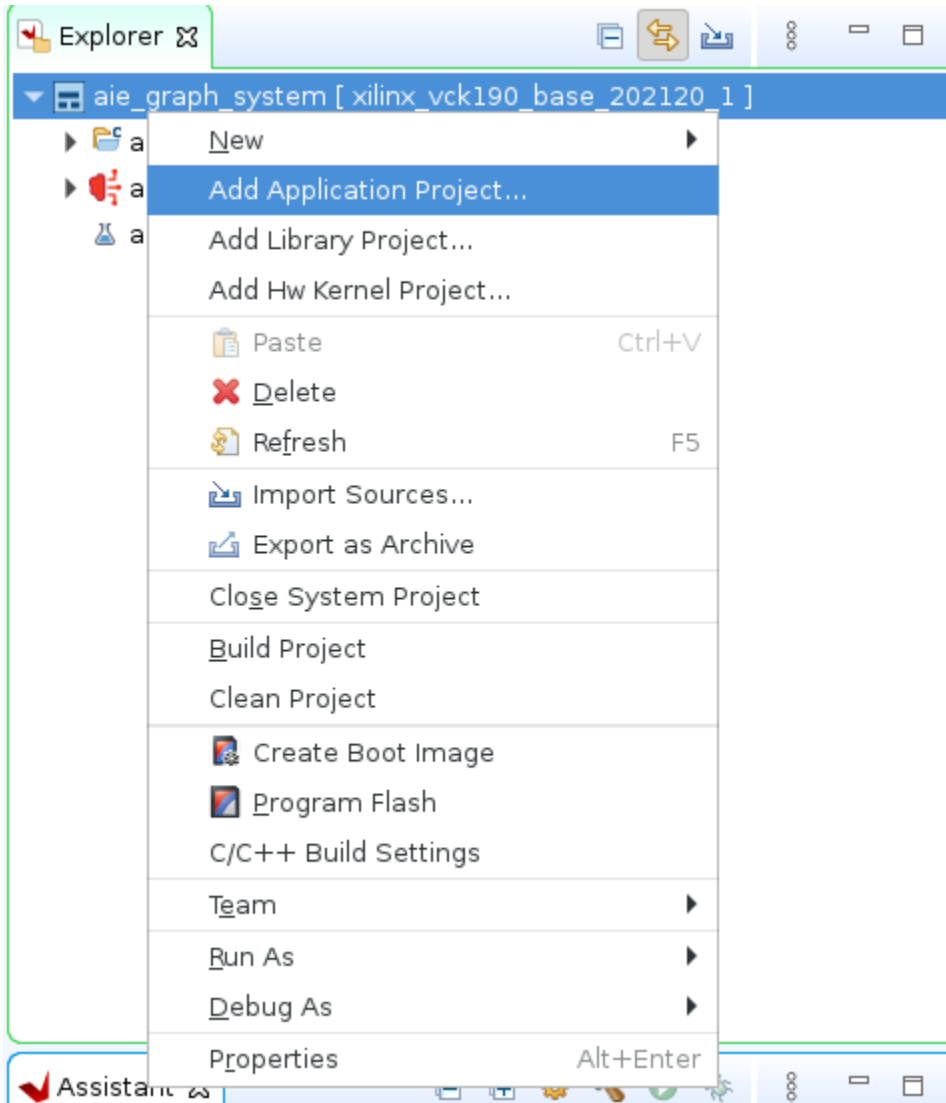
The connectivity `sc` option defines connections between the ports of the AI Engine graph and streaming ports of PL kernels. Connections can be defined as the streaming output of one kernel connecting to the streaming input of a second kernel, or to a streaming input port on an IP implemented in the target platform.

The `system.cfg` file has some differences between the Vitis IDE and the command-line flow as described in [Linking the System](#). The primary difference is that the IDE provides the connectivity `nk` options of the `config` file, instantiating a specified number of compute units (CUs) per kernels, as specified in the build settings. In addition, the IDE uses a naming convention for CUs in the form `<kernel>_#`, where `#` indicates the CU instance. In the case where there is only one CU instance, it has the `_1` extension. This means that for the Vitis IDE, the `system.cfg` should not specify the `nk` option, and the `sc` option should use the CU instance name from the IDE.

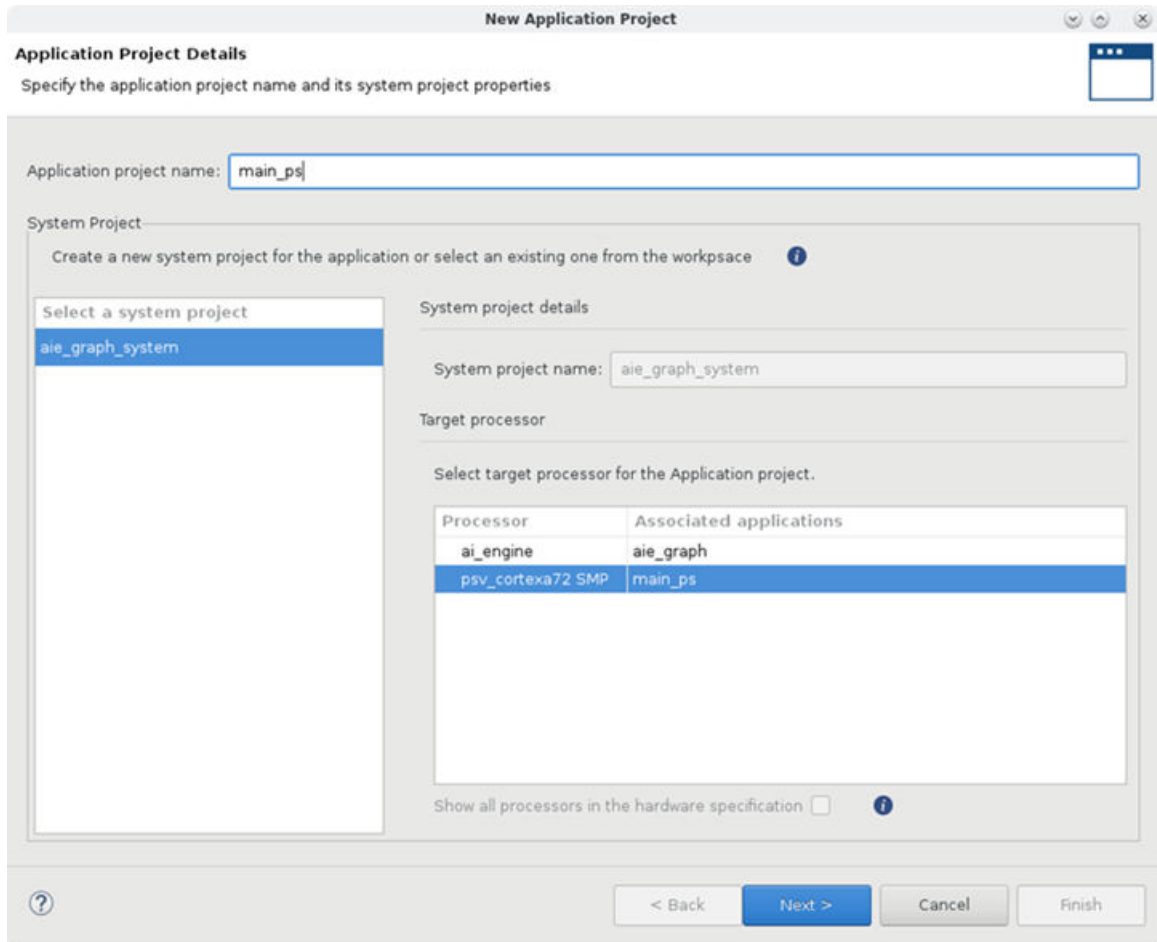
Adding a PS Application to the System

Your top-level system project can also have an application to run in the PS domain of the Versal ACAP to load and run the AI Engine graph and PL kernels. Use the following process to create a PS application project and add it to your system project.

1. In the Explorer view, right-click on the system project and select the **Add Application Project** command, as shown in the following figure.



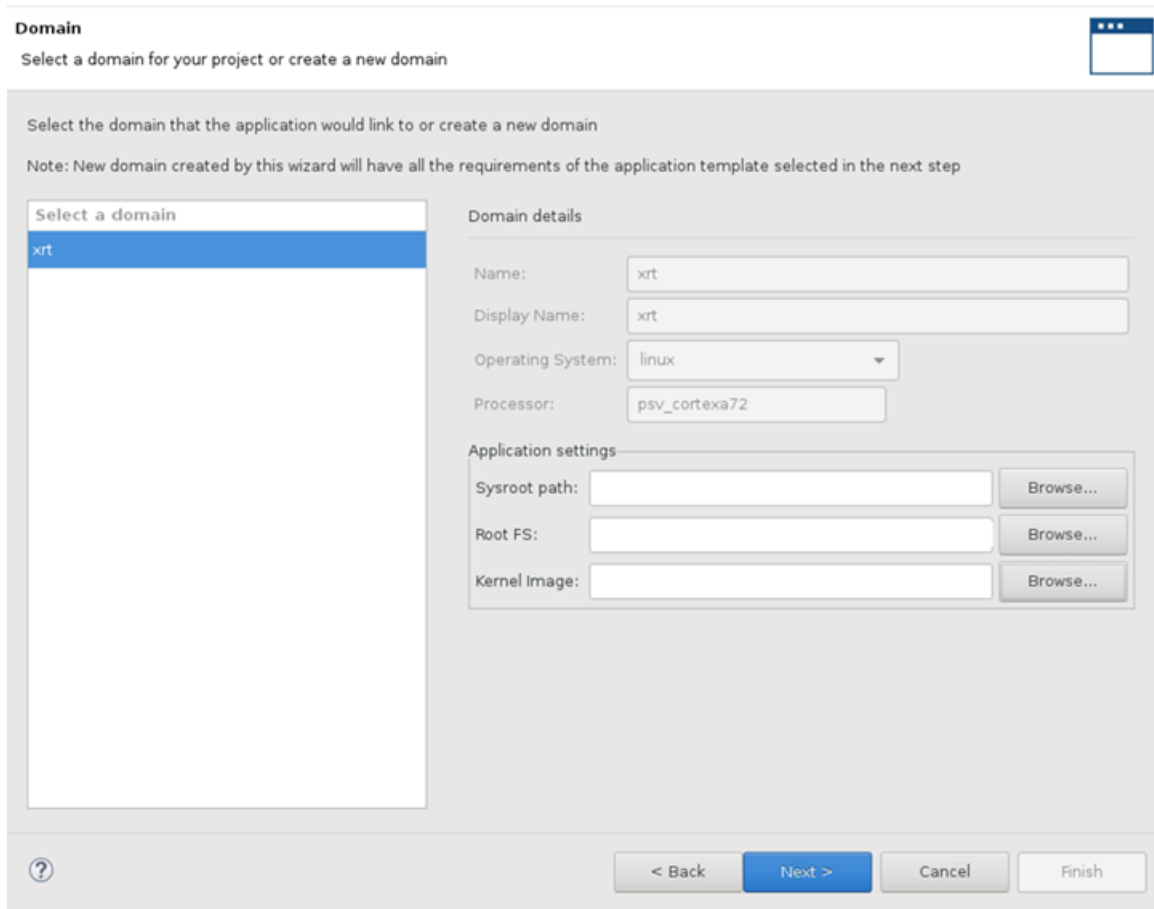
2. This displays the Application Project Details page as shown in the following figure.



Make sure the project is assigned to the existing system project, which also contains your AI Engine graph project. Specify the **Application project name**.

Select the Cortex[®]-A72 processor core, and click **Next** to proceed.

- This displays the Domain page of the project wizard as shown below.



Domain

Select a domain for your project or create a new domain

Select the domain that the application would link to or create a new domain

Note: New domain created by this wizard will have all the requirements of the application template selected in the next step

Select a domain

- xrt

Domain details

Name: xrt

Display Name: xrt

Operating System: linux

Processor: psv_cortexa72

Application settings

Sysroot path: Browse...

Root FS: Browse...

Kernel Image: Browse...

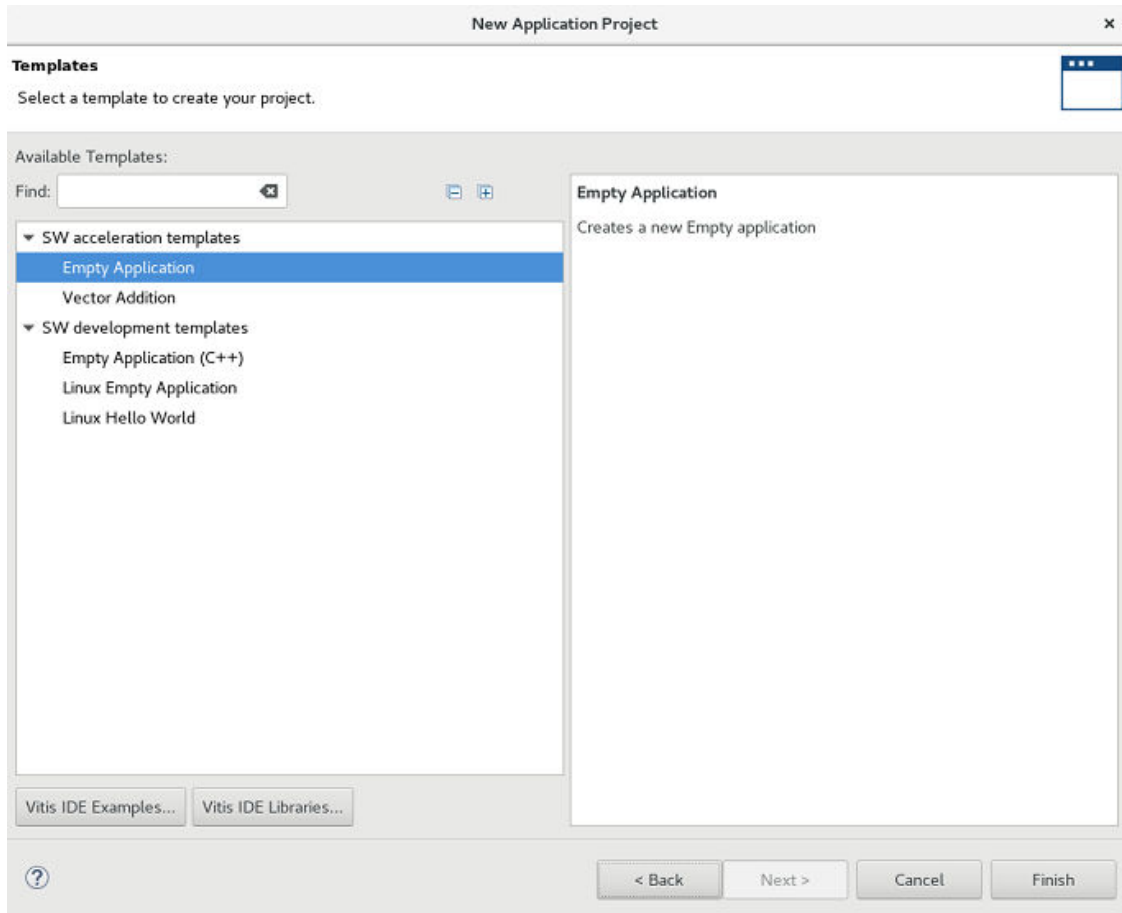
< Back Next > Cancel Finish

Because you have selected the Cortex®-A72 processor core, the XRT domain is the only available option. This indicates the domain includes the Linux operating system and the XRT library. You must also specify the following three elements of the embedded platform:


- Sysroot
- Root FS
- Kernel Image

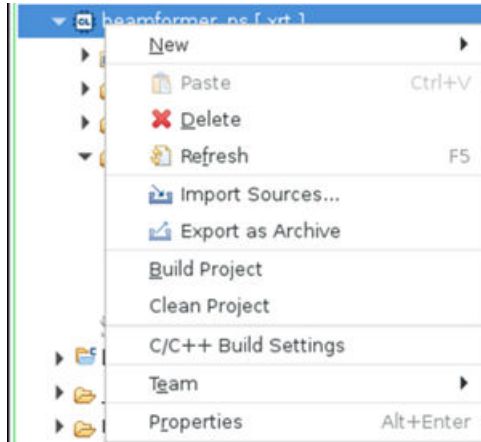
These items are required for loading and booting the operating system and run-time drivers. These files are available from the Embedded Platforms download page. Specify the files and click **Next** to proceed.

4. The Templates page opens for the PS Application project, as shown in the following figure.



In this case, where you are creating a new PS project, select the **Empty Application** template (default), and click **Finish** to create the project. The new PS project is created and added to the top-level system project that you have been working on.

5. Add the source code for the PS application. In the Explorer view, right-click the `src` folder of the PS project and click the **Import Sources** command (). Browse to and select the necessary source files for your PS application. Click **Finish** to import the selected source files.
6. Add include directories, link libraries, C++ standard, and build configurations, as required, by right-clicking on the PS project and selecting **C/C++ Build Settings**. You can configure the project properties for the PS project.



With the PS application project created, as well as the AI Engine graph and the PL kernels, your top-level system project is almost ready to build from the top-down.

Building and Running the System

With the top-level system project defined, the AI Engine graph application added, the PL kernels added, the PS application added, and the HW-Link project configured, you are now ready to build and run the system.

The system project supports three different build targets: Emulation-SW, Emulation-HW, and Hardware. You can build the top-level system project using the following steps.

1. Double-click the `<project>.sprj` file in the Explorer view to open the system project in the editor area.
2. Set the **Active build configuration** in the Project Editor window to either Emulation-SW, Emulation-HW, or Hardware to select a specific build target.
3. In the Assistant view, select the top-level project and click the **Settings** command (⚙️) to display the System Project Settings dialog box and make any needed changes prior to the build.
4. Click the **Build** command (🔨) in the toolbar menu to start the build process for the active build configuration. The build process in the Vitis IDE is incremental and will only build elements of the project that have been updated and need to be rebuilt after the last build. You can build the individual elements of the system project, such as the AI Engine graph or PL kernels, and the tool recognizes whether these elements need to be rebuilt.

Note: The build process for hardware takes considerably longer to run than the build for emulation. This is why it is important to debug your design in the emulation build before moving on to the hardware build.

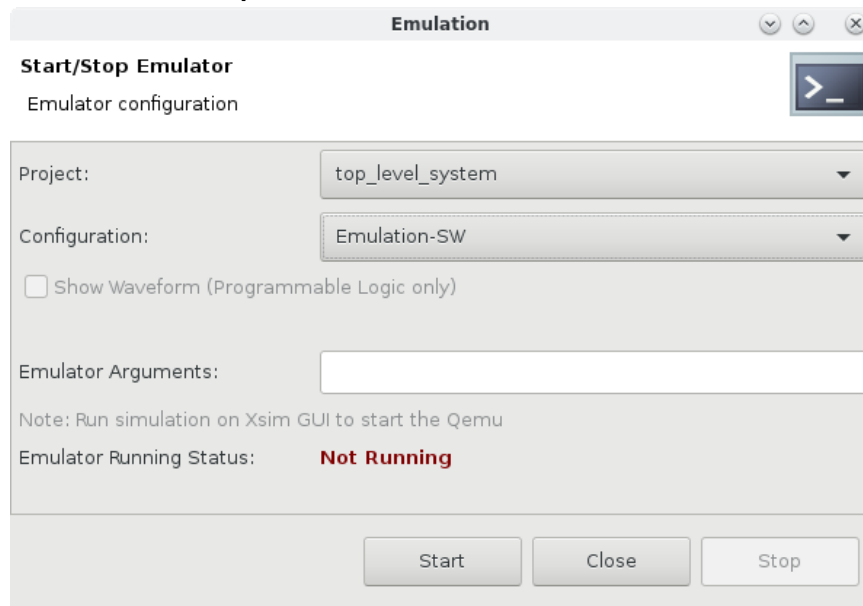
5. When the build completes, examine the contents of the Emulation-SW, Emulation-HW, or Hardware build folders in the Explorer view. You can select and expand the folders of the build directories. You can see the output files of the Vitis compiler package process (`v++ --package`) in the output hierarchy. The build process generates the emulation data and boot files needed for the system, and writes them to the `sd_card` folder.

Note: Two folders are created under the Hardware folder, `package` and `package_no_aie_debug`. The `sd_card.img` file within the `package` folder is for hardware debug purposes whereas the `sd_card.img` file in the `package_no_aie_debug` folder is for regular application execution.



IMPORTANT! In an AI Engine system project you can debug and run the system-level project, or debug and run the AI Engine project. You cannot debug and run the PS or PL projects except as part of the top-level system project.

6. For software emulation builds, start the QEMU emulation environment by selecting the **Xilinx → Start/Stop Emulator** command.

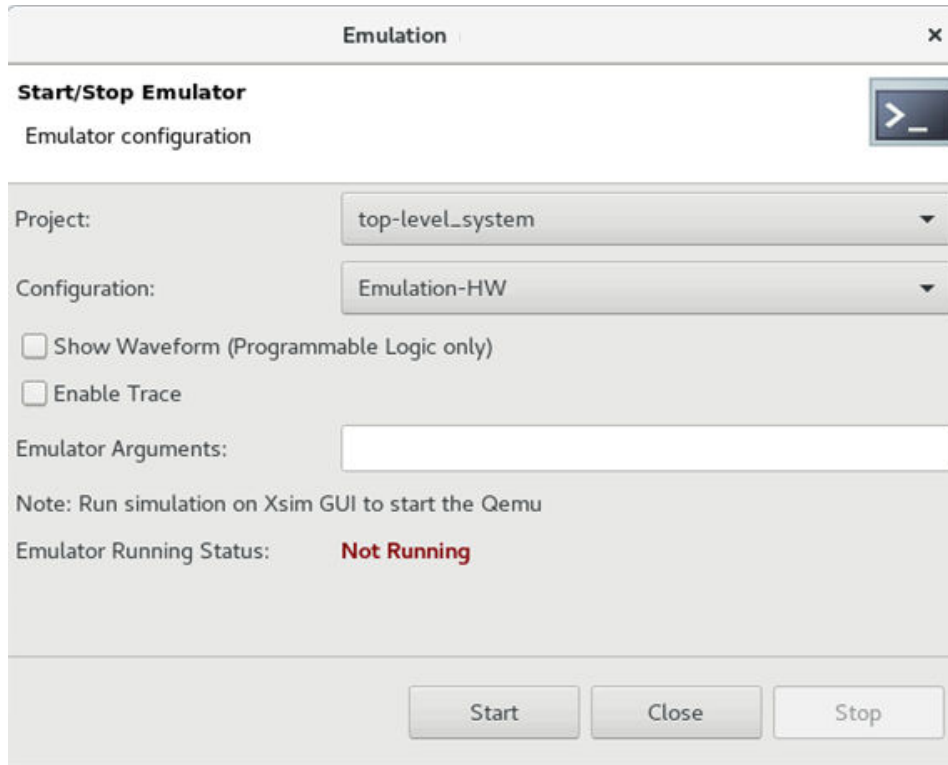


This launches the emulator and then waits until Linux is booted within the QEMU. The Emulation console shows a transcript of the QEMU launch and Linux boot process. You can tell when the process has completed when the progress dialog closes and the `qemu%` prompt is black. You can examine the transcript for details of this process.

When launching software emulation, you can specify options for the AI Engine simulator that runs the graph application, as described in [Reusing x86 Simulator Options](#). The options can be specified in the Emulator Arguments field shown in the preceding figure by specifying the following command.

```
-aie-sim-options ${FULL_PATH}/x86sim.options
```

7. For hardware emulation builds, start the QEMU emulation environment by selecting the **Xilinx → Start/Stop Emulator** command.



This launches the emulator and then waits until Linux is booted within the QEMU. The Emulation console shows a transcript of the QEMU launch and Linux boot process. You can tell when the process has completed when the progress dialog closes and the `qemu%` prompt is black. You can examine the transcript for details of this process.

When launching hardware emulation, you can specify options for the AI Engine simulator that runs the graph application, as described in [Reusing AI Engine Simulator Options](#). The options can be specified in the Emulator Arguments field shown in the preceding figure by specifying the following command.


```
-aie-sim-options ${FULL_PATH}/aiesim_options.txt
```

8. From the Run Configurations dialog box, select **Run** to proceed.

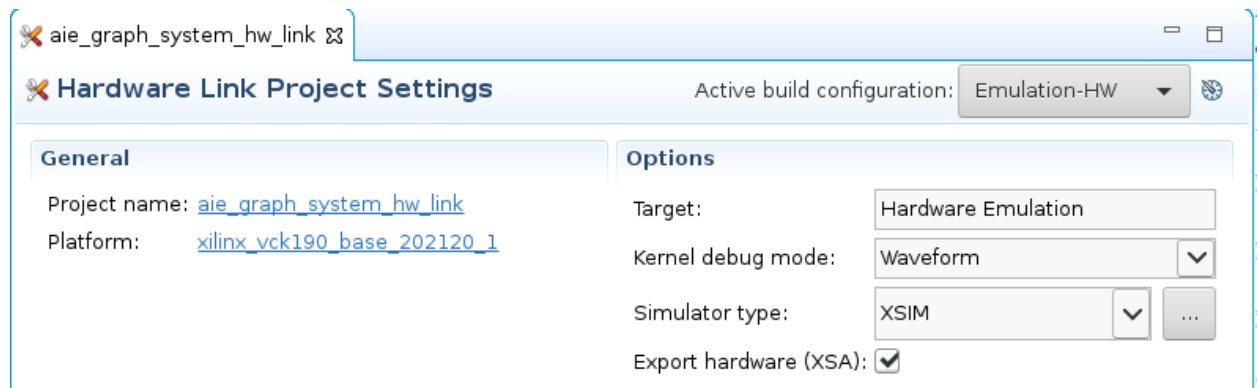
Building a Bare-metal AI Engine in the Vitis IDE

Building a bare-metal system requires a deviation from the standard application flow previously described. The branch in the process occurs at the [Configuring the HW-Link Project](#) step, with the bare-metal system requiring an addition to that step and then following a new process. The specific steps required are detailed below.

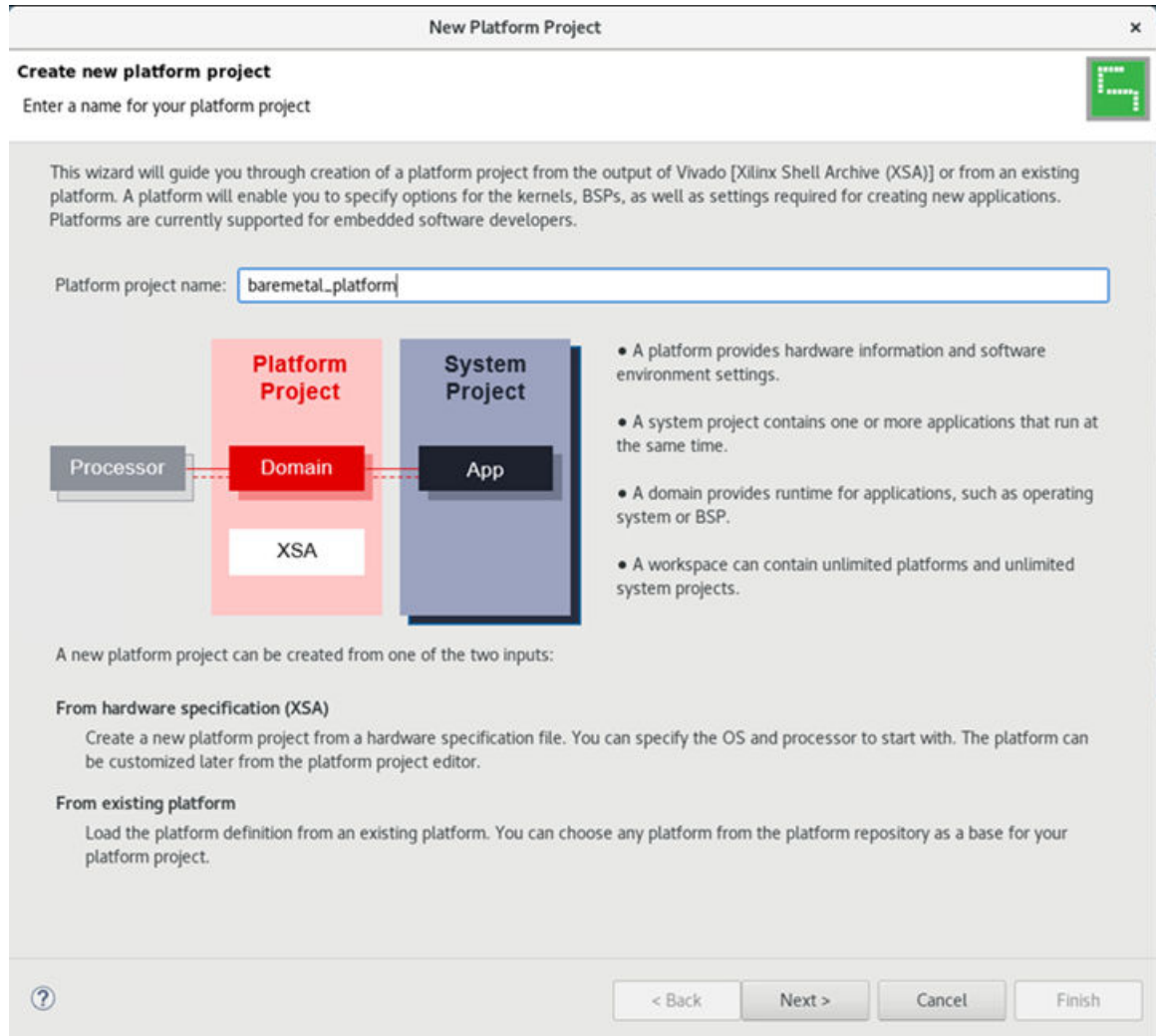
1. For bare-metal systems you must follow the process described in [Configuring the HW-Link Project](#) with the added step of enabling the Export Hardware (XSA) option for Emulation-HW and Hardware builds. This triggers the creation of a fixed-XSA file used for building a bare-metal platform using data found in the AI Engine graph and PL regions of the design.

 **IMPORTANT!** *The fixed-XSA matches the specified build target of the Vitis compiler. So a hardware emulation capable fixed-XSA is generated from the `hw_emu` build target, and a hardware capable fixed-XSA is generated from the `hw` build target.*

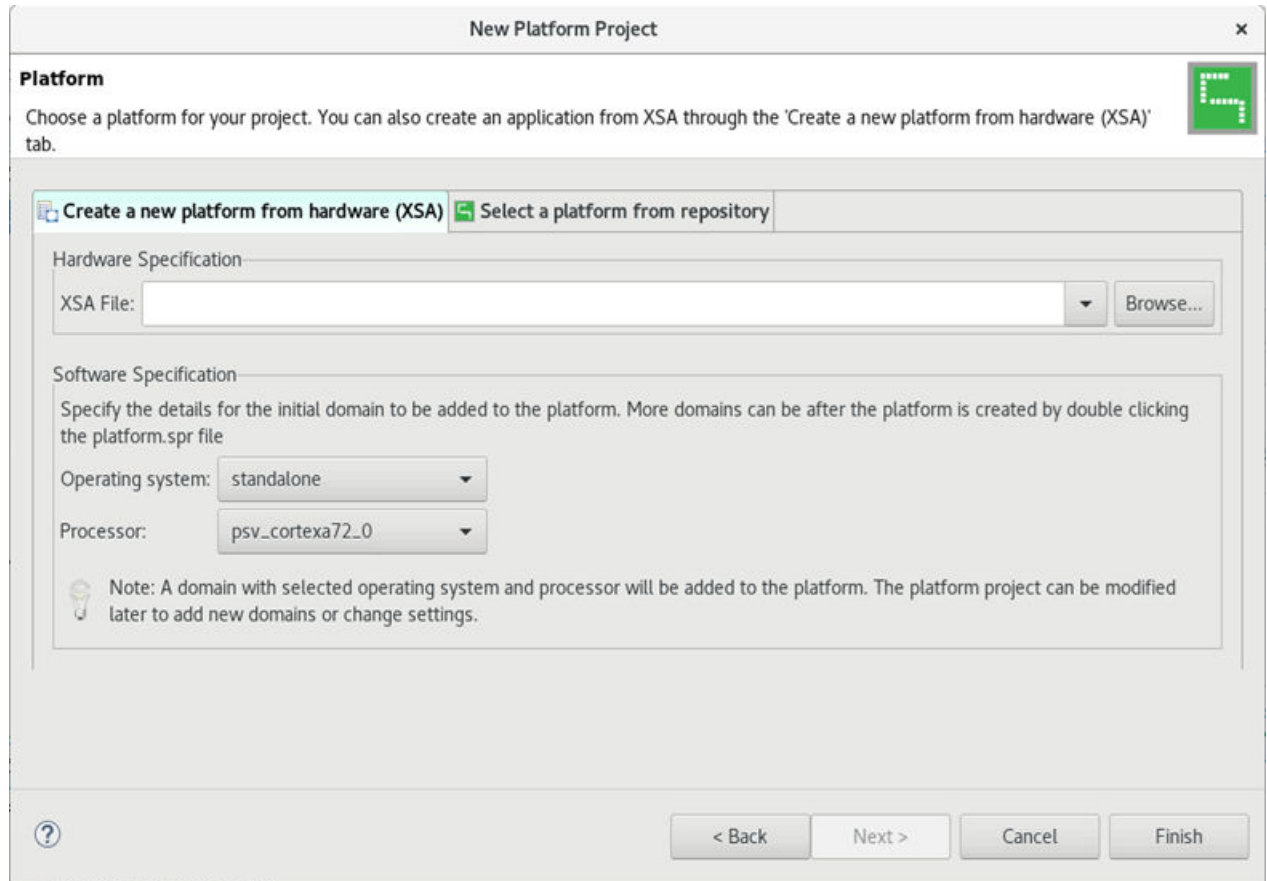
After enabling this option, build the emulation or hardware build as usual. The fixed-XSA file will be written to the output folder for the build.



2. Create a bare-metal platform. Building bare-metal applications requires a platform with a bare-metal domain. Because the `xilinx_vck190_base_202120_1` base platform does not have one, you must create a custom platform with a bare-metal domain using the fixed-XSA file exported during the build process described above.
 - a. Select the **File → New → Platform Project** command in the Vitis IDE. This opens the New Platform Project wizard as shown.



- b. Specify a Platform project name and click **Next** to proceed. This displays the Platform page of the wizard where you specify an XSA to create the new platform. Use the `binary_container_1.xsa` exported from the HW_Link project as described in Step 1.

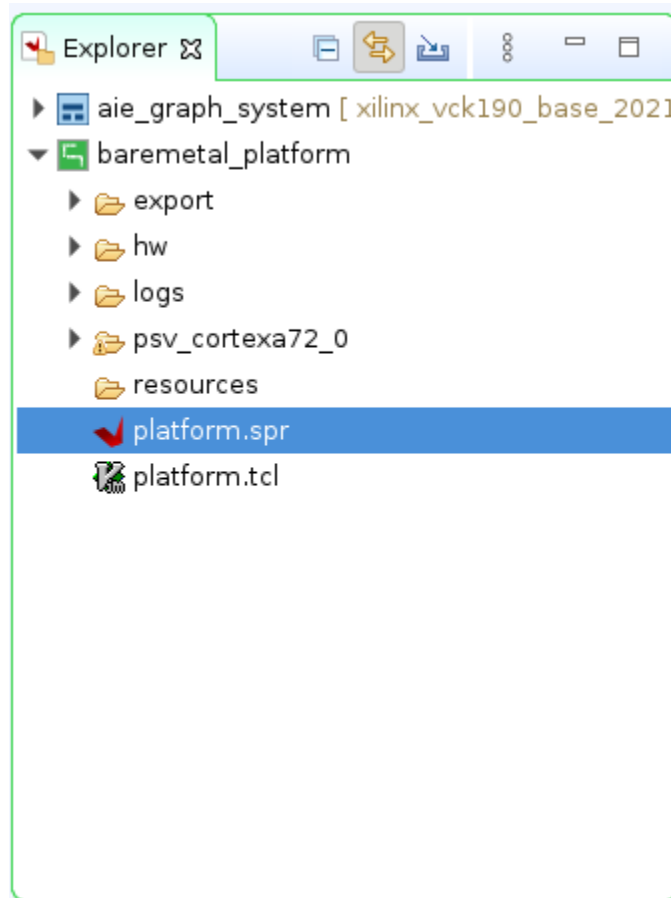


- c. After you select the XSA, the Vitis IDE reads the file, determines the Operating system and Processor for the domain defined by the XSA, and populates it in the dialog box. Click **Finish** to create the platform project.



TIP: The bare-metal platform is valid for either hardware emulation or hardware builds depending on the fixed-XSA selected for the platform.

- d. Click the **Build** command to build the platform. A copy of the completed platform is written to the export folder of the project, and shows in the Explorer view for the project. As shown in the following figure, the exported platform has the `platform.xpfm` meta-data file, as well as the `./hw` and `./sw` folders for the different elements of the platform.

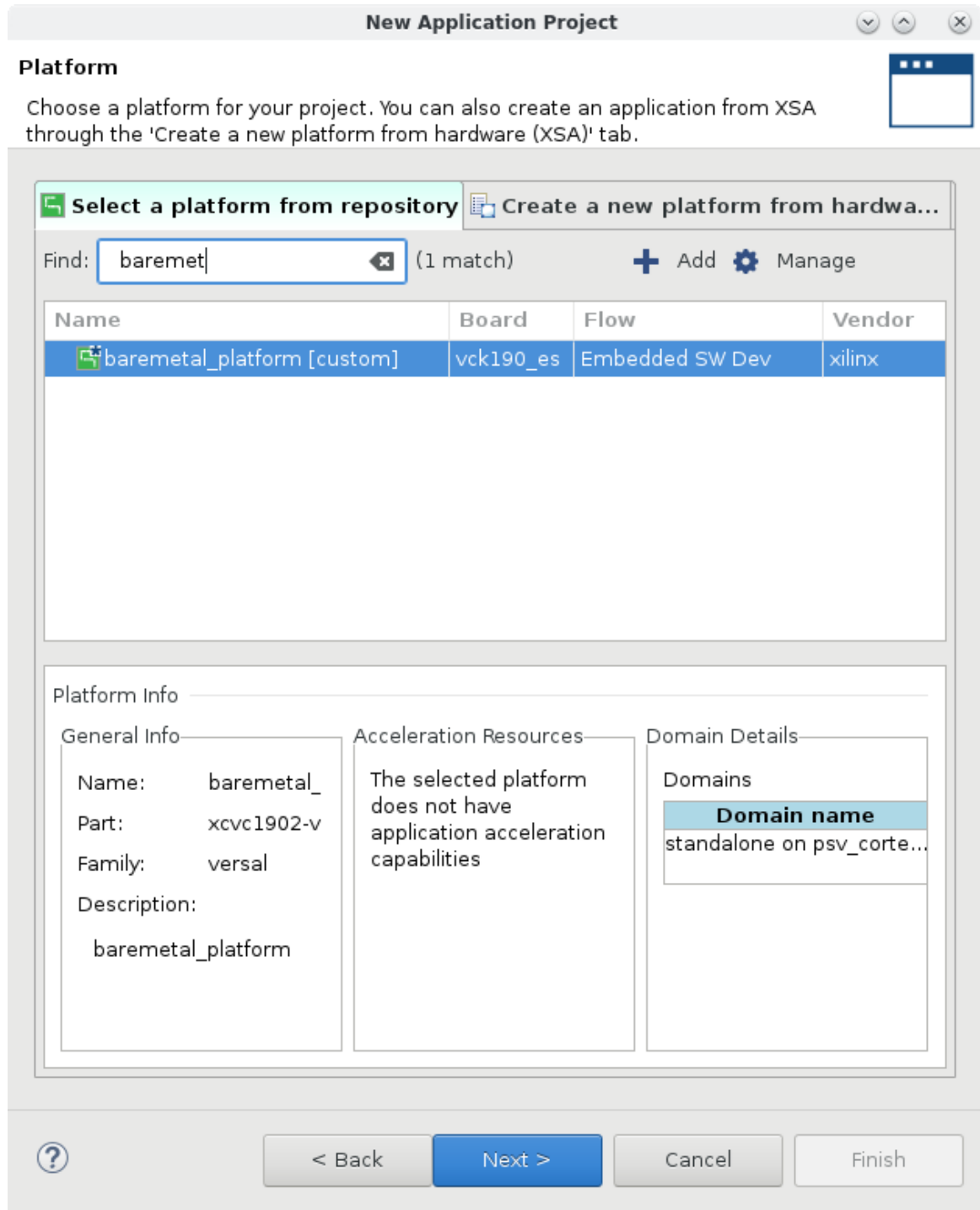


The Vitis IDE automatically adds the new platform to your platform repository making it available to use in new projects. You can also add the file location to your `$PLATFORM_REPO_PATHS` environment variable. This makes the platform accessible to the Vitis IDE, or allows you to specify the platform in command-lines by referring to the platform name rather than the whole path.




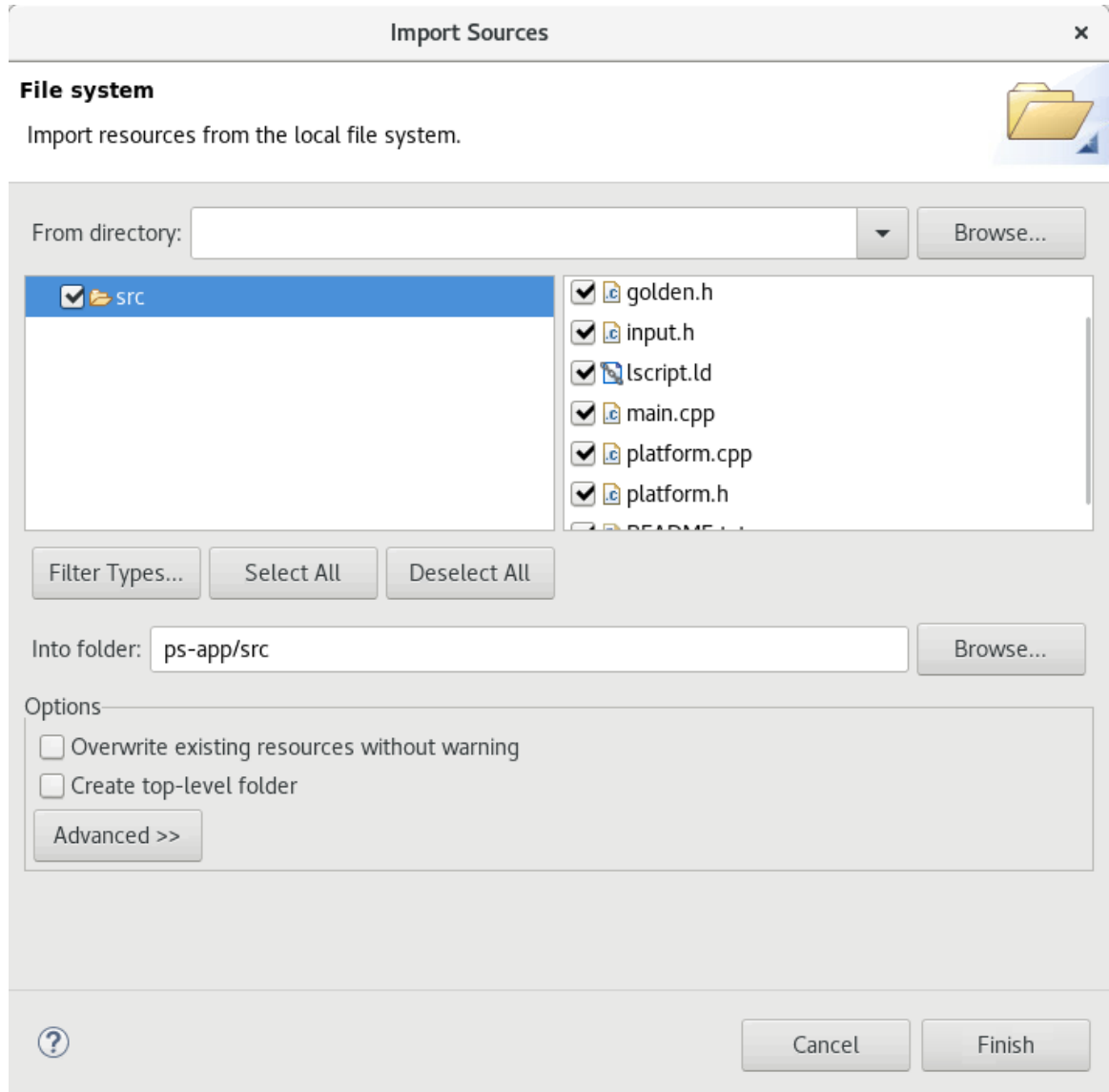
IMPORTANT! *The generated platform will be used only for building the bare-metal PS application and is not used any other places in the flow.*


3. Create a new PS application project.
 - a. Select the **File → New → Application Project** command in the Vitis IDE. This opens the New Application Project wizard.
 - b. Click **Next** to skip past the first page, and display the Platform page as shown.
 - c. Select the **baremetal_platform** you created in the last step, and click **Next** to proceed.

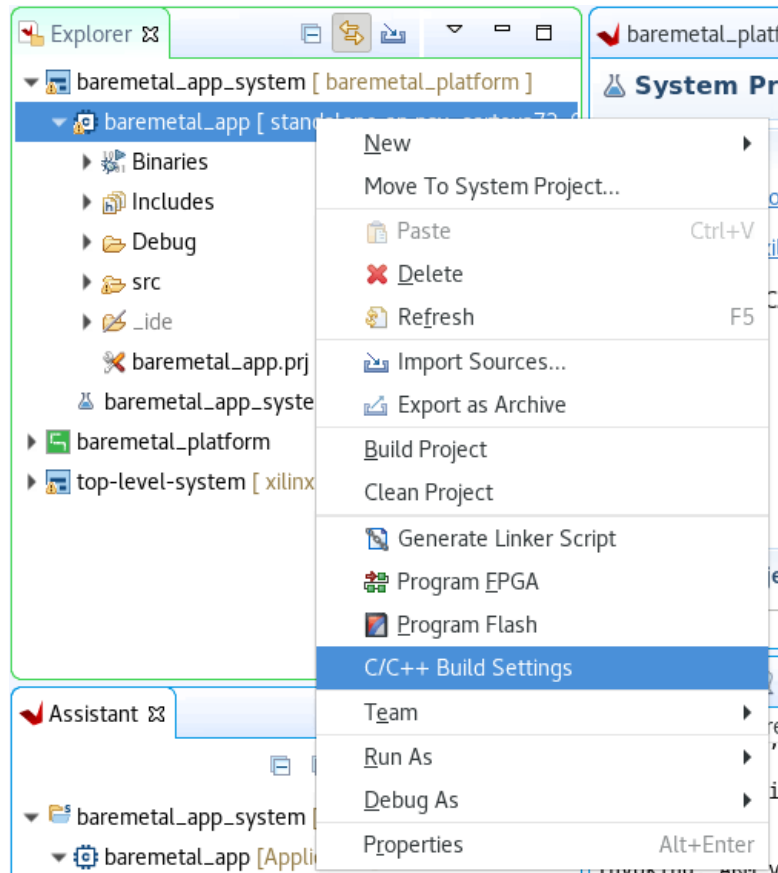


- Provide an **Application project name** and click **Next**.
- Review the Domain page and click **Next** to proceed.
- On the Templates page, select **Empty Application** and click **Finish** to create the project. The project is opened in the Vitis IDE.

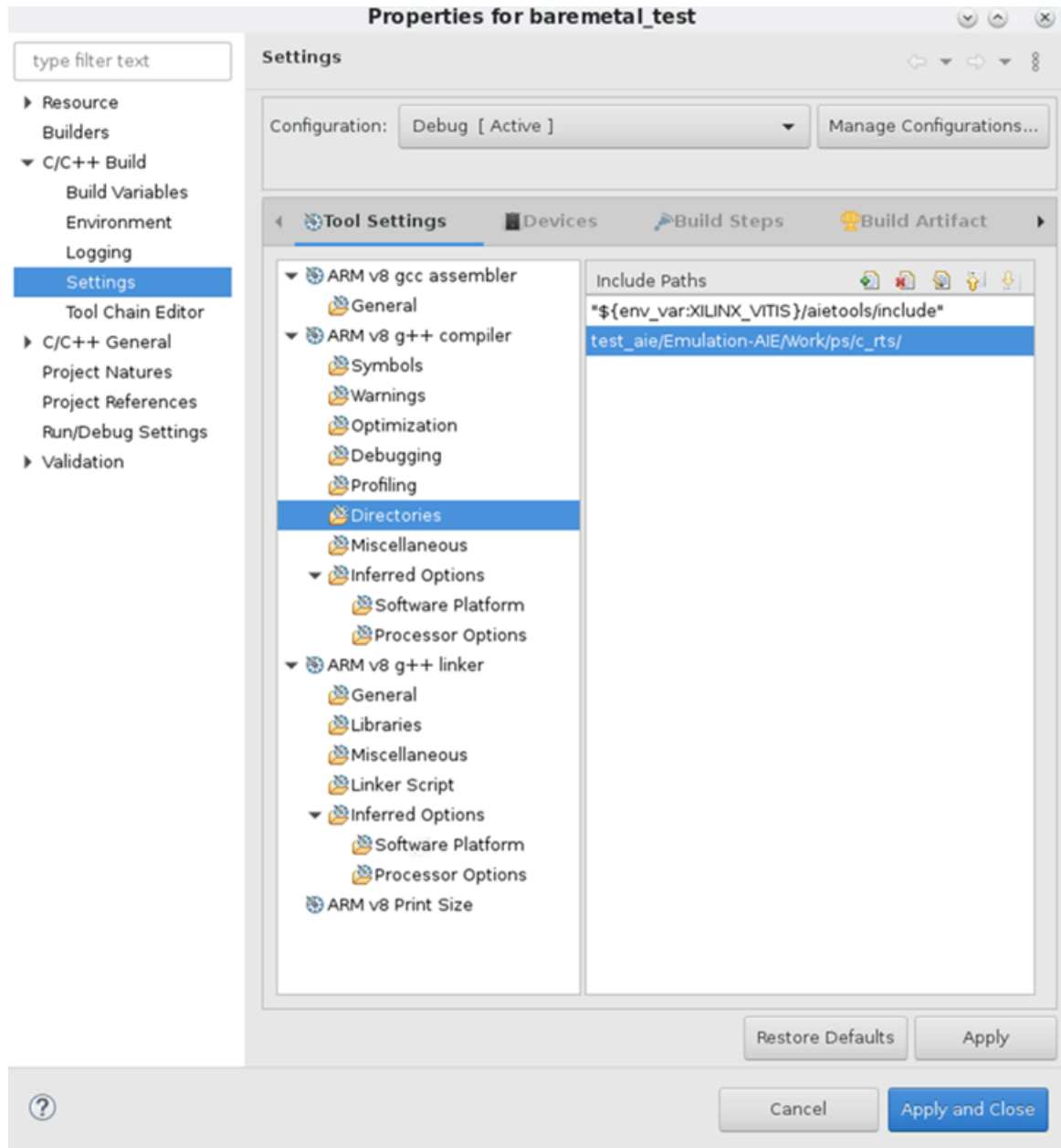
- g. Add the source code for the PS application, `main.cpp`, `platform.cpp`, and associated files written specifically for the bare-metal project. Select the project in the Explorer view, expand the folders, right-click the `src` folder, and click the **Import Sources** command () to open the dialog box shown in the following figure. Select the files to add and click **Finish**.




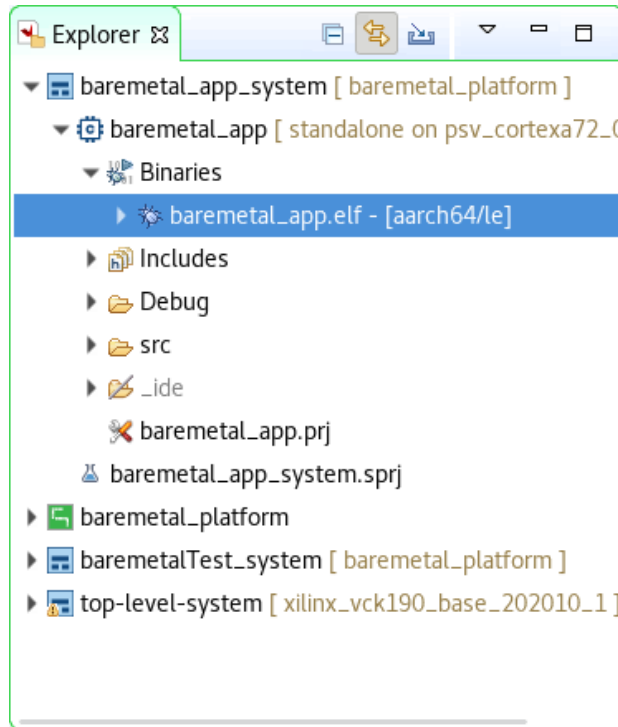
- h. You must also add the bare-metal AI Engine control file (`aie_control.cpp`), which is created by the `aiecompiler` command, and can be found in the `./Work/ps/c_rts` folder. Select the `src` folder again, and click the **Import Sources** icon () to open the dialog box and add the `aie_control.cpp` file to the project.
- i. Finally, you must add to the Include paths for your project. Right-click the bare-metal application project and select **C/C++ Build Settings** as shown.



This selection opens the Build Settings dialog box, as shown in the following figure. Select the **Directories** option as shown, and select the **Add** command (📁) to add the new include paths. You will need to add an entry for the source files for your original AI Engine graph application. This is the `src` folder for the project described in [Creating the AI Engine Graph Project and Top-Level System Project](#). This should point to a folder containing your AI Engine graph source files. Click **Apply and Close** to finish defining the include path.



With the updated Include path, select the **Build** () icon to build your project. When the build completes, you should see the ELF file for your bare-metal application.



4. Package the system.

- With the ELF file produced for the PS application, you are ready to build the system-level project, and package the system for the bare-metal platform. You must run the package process to generate the final bootable image (PDI), and write the SD card content for booting the device and running the application. See [Packaging](#) for more information.
- In the system project created that was created when you set up your AI Engine graph project as discussed in [Creating the AI Engine Graph Project and Top-Level System Project](#), add the following to the **Packaging options** field:

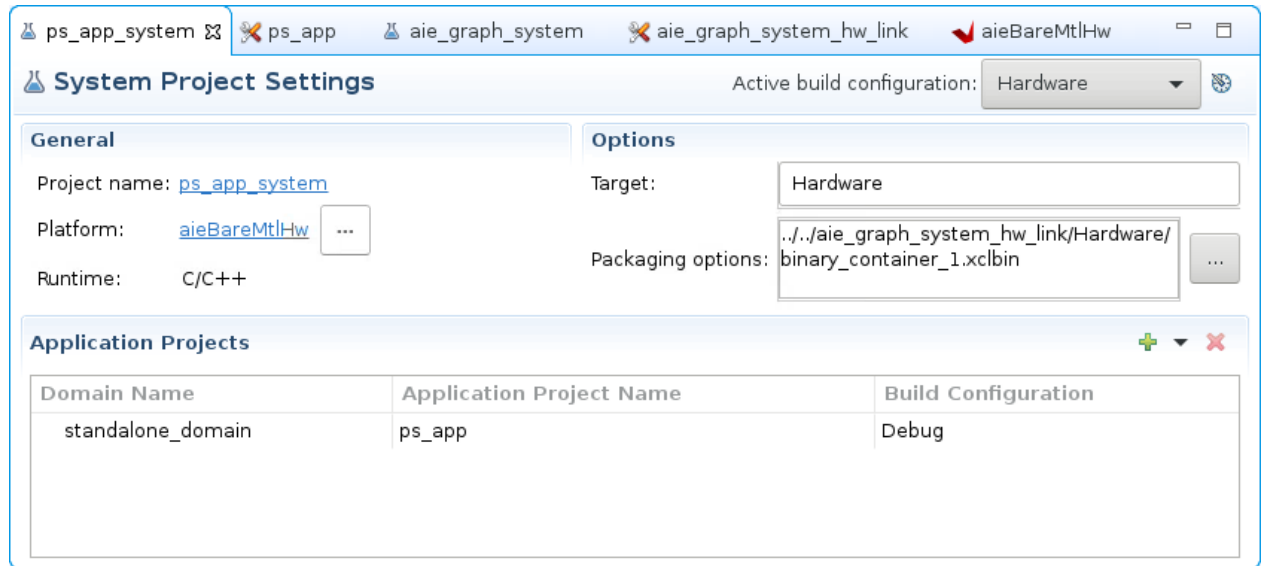
```
--package.ps_elf ../..../baremetal_app/Debug/baremetal_app.elf,a72-0
```

Note: For debugging both the AI Engine graph, together with the bare-metal PS application, you should not add the `--package` option specified in the previous example. If you want to debug only the AI Engine graph, then you need to add the option as described.

- For the hardware build of the bare-metal application system project you need to specify an XCLBIN file for the project in the Packaging options field as shown in the following figure.



TIP: You can copy or reference the required file from the top-level system project, which is the source of the fixed-XSA described in [Step 1](#).



- d. Now build (🔨) the top-level system project.

This adds the ELF file you created in the bare-metal application project, assigns it to a processor core, and builds the system project. See the `--package` option in the [Vitis Compiler Command](#) in the *Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)* for more information.

Note: In the Linux system, you added a PS application into the system project to build and debug as part of the system. Here you are building the PS application as part of a separate bare-metal project and adding it as a boot file for the package process in your top-level system.

Now that you have built the bare-metal system, you can continue to run or debug the application.



IMPORTANT! You cannot debug the hardware emulation build for bare-metal projects in the Vitis IDE.

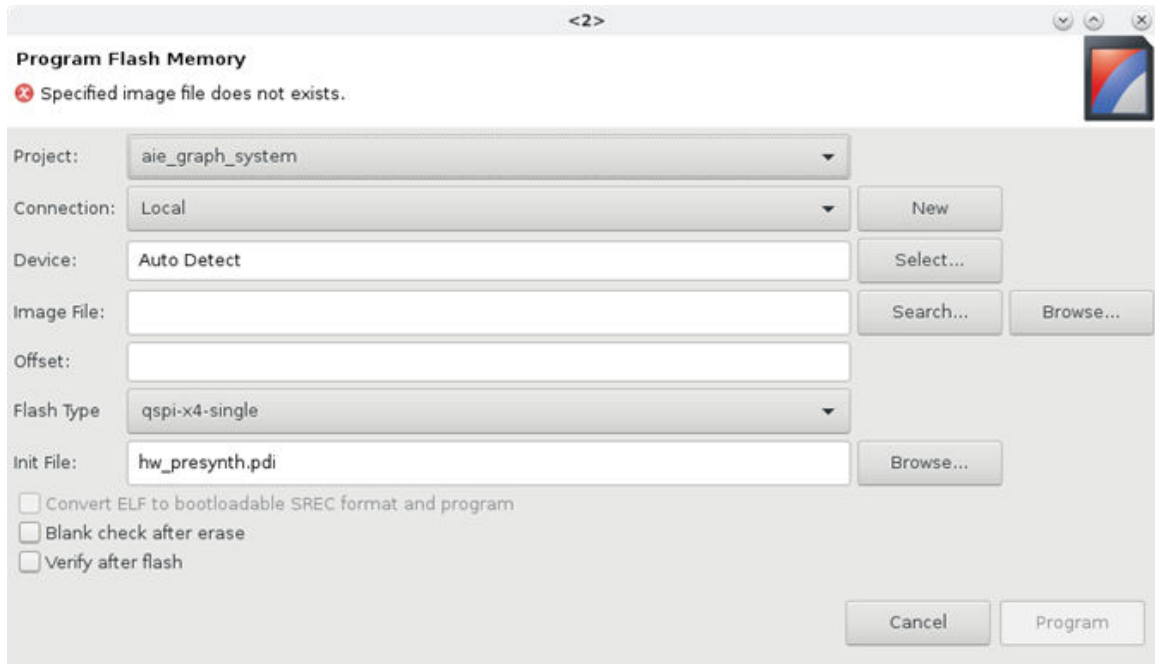
Programming Device and Flash Memory

The Vitis IDE has a feature for programming the board/part using JTAG, allowing you to bypass the need to copy the contents of `sd_card` to an actual SD card to boot on the board.

Programming Flash

After successfully building a design for hardware, right-click **System Project** and select **Program Flash**. You should see a dialog box similar to the following:

Figure 55: Program Flash Memory Dialog Box



In this screen specify the **Image File**, which is generally the `BOOT.BIN`, any offset (if necessary for the flash memory), as well as the **Init File**. The Init File is the partial PDI from the platform you are targeting. To make sure the flash is programmed properly, check the **Verify after flash** check box to confirm it was programmed properly. If the flash memory already has existing data, you can do a more secure erase by checking the **Blank check after erase** check box to make sure it is fully erased before programming.

Make sure that the **Packaging options** in the **System Project Settings** window has -- `package.boot_mode=qspi` set.

Programming the Device

If you do not want to program the flash memory and just want to program the device through JTAG, there is an option under the **Xilinx** menu to select **Program Device**.

Debugging the AI Engine Application

You can debug the AI Engine graph application in a standalone capacity to run and debug just the AI Engine. Or, you can debug the system project, including the top-level PS application, as well as the AI Engine graph. Within this framework, you can also debug applications built from the command line, or system projects built in the Vitis™ IDE. You can debug applications running on the Linux OS, or in bare-metal systems. Finally, you can debug hardware emulation builds that let you simulate the application, or debug the actual application running on hardware. All of these configurations are addressed in the following topics.

Note: It is recommended to follow the following steps while debugging AI Engine applications.

1. Use x86 simulation for both single and multi-kernel debug. It supports breakpoints and single stepping using the GNU debugger.
2. Use AI Engine simulation to verify timing and to check that it will fit within the program memory and stack/heaps size within the available hardware memory space.
3. Software emulation for system level functional verification including host code, along with any c-models for user PL code.
4. Hardware emulation for complete integration testing including PL, PS, and AI Engine domains as applicable.
5. It is recommended to use the compiler optimization option `--xlopt = 0` because higher compiler optimizations will reduce debug visibility.

If code changes are applicable, repeat steps 1 through 4.

Launching Debug from the Vitis IDE

As described in [Chapter 15: Using the Vitis IDE](#) you can build a system level project incorporating the kernels running in the AI Engine domain, the PL domain, and an application running in the PS domain. These system-level projects can be debugged together, form within the Vitis IDE, or you can debug the AI Engine graph application, focusing just on that particular problem. This section discusses running the Debug Environment from the Vitis IDE for hardware emulation and hardware builds.

Using printf for Event Tracing

The simplest form of tracing is to use a formatted `printf()` statement in the code for printing debug messages. Visual inspection of intermediate values, addresses, etc. can help you understand the progress of program execution. No additional include files are necessary for using `printf()` other than standard C/C++ includes (`stdio.h`). You can add `printf()` statements to your code to be processed during simulation, or hardware emulation, and remove them or comment them out for hardware builds.

Adding `printf` statements to your AI Engine kernel code will increase the compiled size of the AI Engine program. Be careful that the compiled size of your kernel code does not exceed the per-AI Engine processor memory limit of 16 KB.



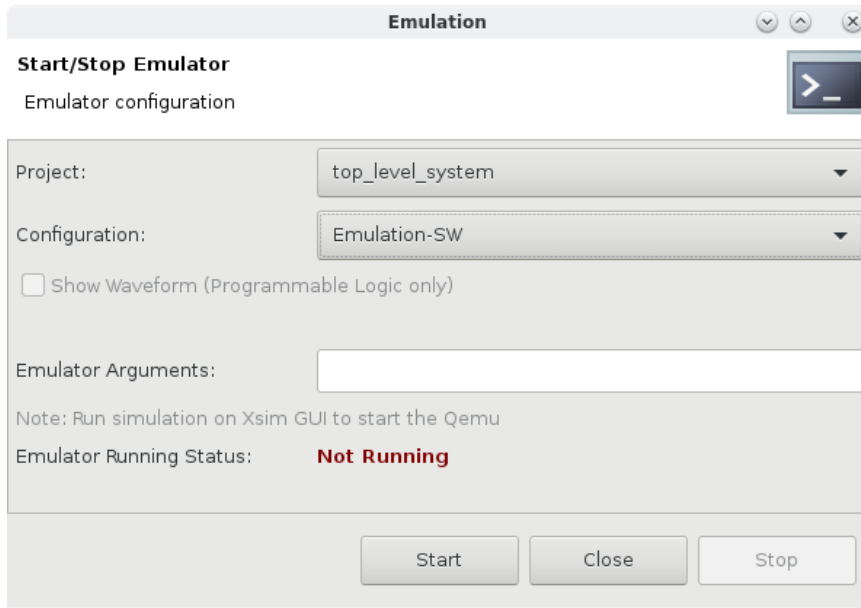
IMPORTANT! You must use the `aiesimulator --profile` command to enable the `printf()` execution during a simulator run. If `--profile` is not specified, the `printf()` function is ignored.

A separate driver and binary is used for this functionality to allow the main simulator to remain as fast as possible. Using the debug simulator driver produces a per-tile profile report under the output directory which gives detailed cycle-level statistics of kernel execution. In addition, using the `--profile` option generates a `run_summary` file that is written to the `./aiesimulator_output` folder that can be viewed as described in [Viewing the Run Summary in the Vitis Analyzer](#).

Software Emulation Debug from the Vitis IDE

Prior to running the application in software emulation, you need to build the system project using the Emulation-SW build target. To run and debug the application in the Emulation-SW build target, you must use the following steps.

1. Start the QEMU emulation environment by selecting the **Xilinx → Start/Stop Emulator**.



This will launch the emulator and then wait until Linux is booted within QEMU. The Emulation console shows a transcript of the QEMU launch and Linux boot process. You can tell when the process has completed when the progress dialog closes and the Emulation Console window shows a blank `qemu%` prompt. You can examine the transcript for details of the process.

When launching software emulation, you can specify options for the AI Engine simulator that runs the graph application. As described in [Reusing x86 Simulator Options](#), the options can be specified in the Emulator Arguments field shown in the previous image by specifying the following command.

```
-x86-sim-options ${FULL_PATH}/x86sim.options
```

Note: `${FULL_PATH}` is the full path to the location of the `x86sim.options` file.




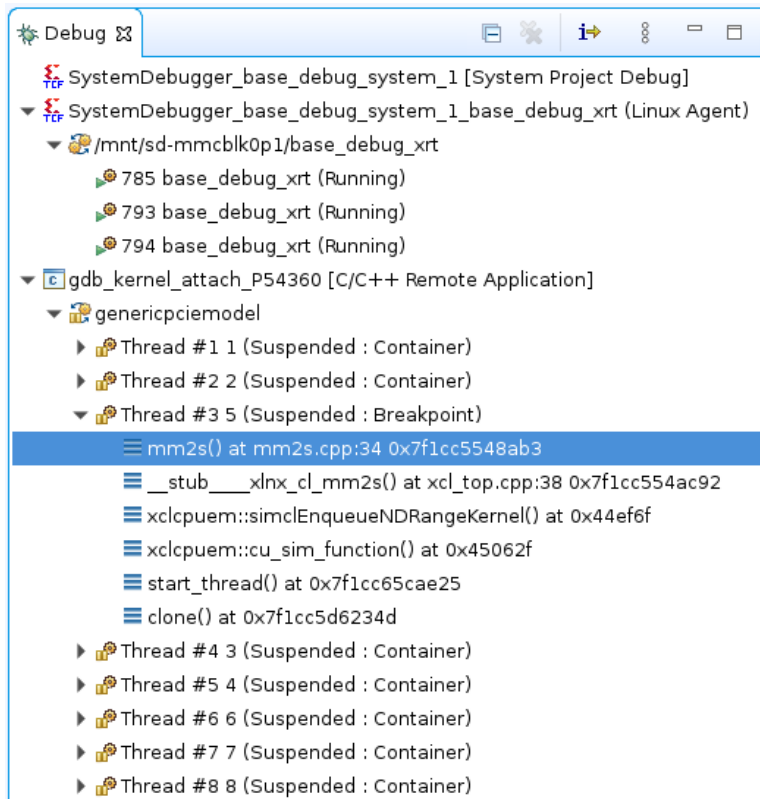
IMPORTANT! Open up the host application source file. Right-click on the side of the `int main()` to set a breakpoint because the IDE does not automatically set the breakpoint at `main()`.

2. Right-click on the top-level system project, and select the **Debug As → Launch SW Emulator** command. This opens the Debug Configurations dialog box.
3. Click **Debug** to proceed.

This opens the Debug perspective in the Vitis IDE, and connects to the PS application and AI Engine graph running on their respective threads in the QEMU.

Note: See [Vitis IDE Layout for Software Emulation Debug](#) for additional details on the views used during debugging.

4. Click the **Resume** button  to go to the next breakpoint. Looking at the Debug view, you can see the various threads spawned.



Note: The Debug view also shows which thread has hit a breakpoint with the info (Suspended : Breakpoint).

5. Review the Variables view. This view shows all the variables and objects, and in the following figure, an AI Engine kernel. For example, the following figure shows the expanded sbuff shows how a v32cint16 datatype is formatted.

The screenshot shows the 'Variables' tab in the Xilinx IDE. The 'sbuff' variable is expanded, showing its internal structure. The table below represents the data shown in the screenshot:

Name	Type	Value
cb_input	input_window_cint16 *	0x1b123c0
cb_output	output_window_cint16	0x1b12470
shift	const int	0
output_samples	const unsigned int	28384456
sbuff	v32cint16	{...}
bits	const int	1024
isSigned	const bool	false
val	v32cint16::BitType	{...}
VBitVectorBase<VBit>	VBitVectorBase<VBit>	{...}
B	const int	32
S	const bool	false
auxiliarySize	const int	<optimized out>
data	std::__1::array<v1cint16, 32>	{...}
__elems_	v1cint16 [32]	0x7f1ca3ffecc0
__elems_[0]	v1cint16	{...}
bits	const int	32
isSigned	const bool	false

Below the table, the details for the 'sbuff' variable are shown:

```

Name : sbuff
Details:{static bits = 1024, static isSigned = false, val =
Default:{...}
Decimal:{...}
Hex:{...}
Binary:{...}
Octal:{...}

```

- Click the **Registers** view. With a breakpoint triggered in an AI Engine kernel, this view shows all of the relevant registers.

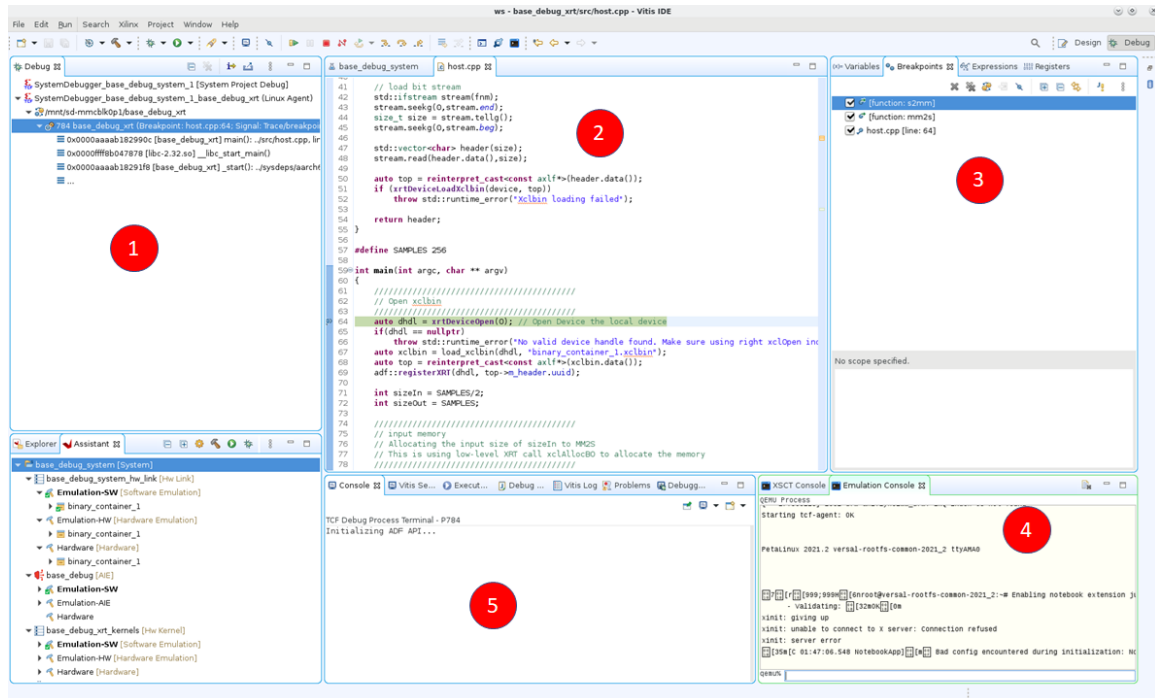
(x)= Variables Breakpoints Expressions Registers		
Name	Value	Description
General Registers		
rax	28386240	General Purpose and FPU Re
rbx	28386416	
rcx	1	
rdx	28384960	
rsi	28386416	
rdi	28386240	
rbp	0x7f1ca3ffed80	
rsp	0x7f1ca3ffd720	
r8	0	
r9	1	
r10	1397615332860	
r11	1397615308695	
r12	1	
r13	28398368	
r14	1397615637931	
r15	28398688	
Name : rax		
Hex:0x1b123c0		
Decimal:28386240		
Octal:0154221700		
Binary:1101100010010001111000000		
Default:28386240		

Note: If a breakpoint is triggered in the host application source, it shows the PS registers.

Vitis IDE Layout for Software Emulation Debug

This section describes all the aspects of the Vitis IDE for effective debugging for Emulation-SW.

The following figure shows the typical layout of the Vitis IDE when in the Debug perspective.



The numbered sections are as follows:

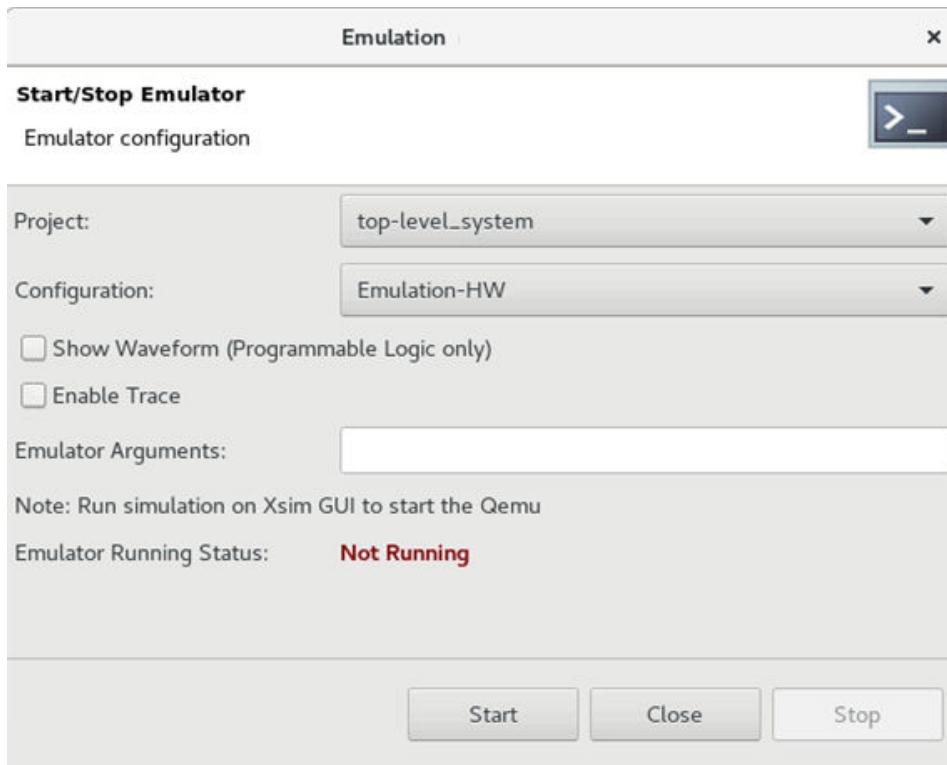
1. The Debug view displays the list of threads that are associated with the running the PL, AI Engine, and PS applications as the PS host application, PL kernels, and AI Engine kernels.
2. The code window. This window opens up the source file when a breakpoint in that file is triggered.
3. The Variables, Breakpoints, Expressions, and Registers views.
 - Variables view shows all the available variables used in the perspective of the triggered breakpoint. For example, if a breakpoint is in the host application source file, it will display all objects and variables associated with that code.
 - Breakpoints view shows all breakpoints, enabled or disabled, for the design that is being debugged.
 - Expressions allows for specific expressions to be written in which to trigger a pause in code execution. For example, checking and breaking when a for loop variable reaches a certain value.
 - Registers view shows the registers of the Cortex®-A72 when a breakpoint is triggered in the host application source code, and the AI Engine when a breakpoint is triggered in the AI Engine kernel.
4. The Emulation Console and the XSCT Console views.
 - Emulation Console provides a transcript of QEMU, as well as allows you to run Linux commands.

- XSCT Console allows you to see the program memory and snippet of code when a breakpoint is triggered.

Hardware Emulation Debug from the Vitis IDE

To run and debug the application in the Emulation-HW build target, you must use the following steps:

1. Start the QEMU emulation environment by selecting the **Xilinx → Start/Stop Emulator** command.



This launches the emulator and then waits until Linux is booted within the QEMU. The Emulation console shows a transcript of the QEMU launch and Linux boot process. You can tell when the process has completed when the progress dialog closes and the `qemu%` prompt is black. You can examine the transcript for details of this process.

When launching hardware emulation, you can specify options for the AI Engine simulator that runs the graph application, as described in [Reusing AI Engine Simulator Options](#). The options can be specified in the Emulator Arguments field shown in the prior figure by specifying the following command:

```
-aie-sim-options ../aiesim_options.txt
```

2. Right-click on the top-level system project, and select the **Debug As → Launch HW Emulator** command. This opens the Debug Configurations dialog box.
3. Press **Debug** to proceed.

This opens the Debug perspective in the Vitis IDE, and connects to the PS application and AI Engine graph running on their respective cores in the QEMU. The application automatically breaks at the `main()` function for all the ELF files.

From this point you can do all the debug activities like step in/step over/viewing variables/plan break points in the emulation environment. Refer to [Using the Debug Environment](#) for more information.

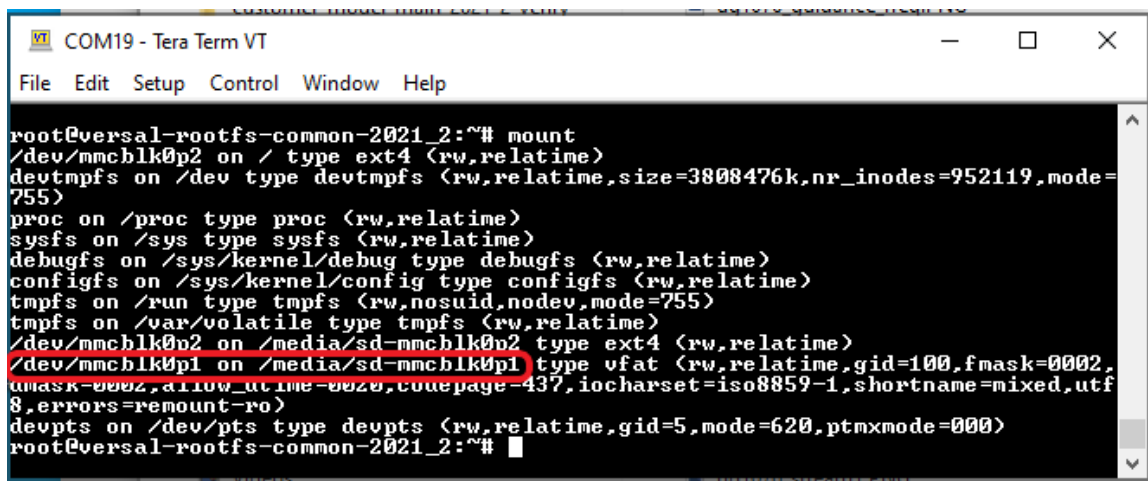
Hardware Debug from the Vitis IDE

With the top-level system project built, you must use the following steps to debug the application in the Hardware build target.

1. Burn `<project>/Hardware/package/sd_card.img` to a physical SD card. This creates a boot-able medium for your target platform.
2. Insert the SD card into the card reader of the [VCK190](#) evaluation kit.
3. Change the boot-mode settings of the card to SD boot mode, and power up the board.
4. After the VCK190 is booted, enter the `mount` command at the command prompt to get a list of mount points. As shown in the following figure, the `mount` command displays mounting information for the system.



TIP: Be sure to capture the proper path for the `cd` command in the next step, and subsequent commands, based on the results of the `mount` command.



```

root@versal-rootfs-common-2021_2:~# mount
/dev/mmcbllk0p2 on / type ext4 (rw,relatime)
devtmpfs on /dev type devtmpfs (rw,relatime,size=3808476k,nr_inodes=952119,mode=755)
proc on /proc type proc (rw,relatime)
sysfs on /sys type sysfs (rw,relatime)
debugfs on /sys/kernel/debug type debugfs (rw,relatime)
configs on /sys/kernel/config type configs (rw,relatime)
tmpfs on /run type tmpfs (rw,nosuid,nodev,mode=755)
tmpfs on /var/volatile type tmpfs (rw,relatime)
/dev/mmcbllk0p2 on /media/sd-mmcbllk0p2 type ext4 (rw,relatime)
/dev/mmcbllk0p1 on /media/sd-mmcbllk0p1 type vfat (rw,relatime,gid=100,fmask=0002,umask=0002,allow_utime=0000,codepage=437,ioccharset=iso8859-1,shortname=mixed,utf8,errors=remount-ro)
depts on /dev/pts type depts (rw,relatime,gid=5,mode=620,ptmxmode=000)
root@versal-rootfs-common-2021_2:~#

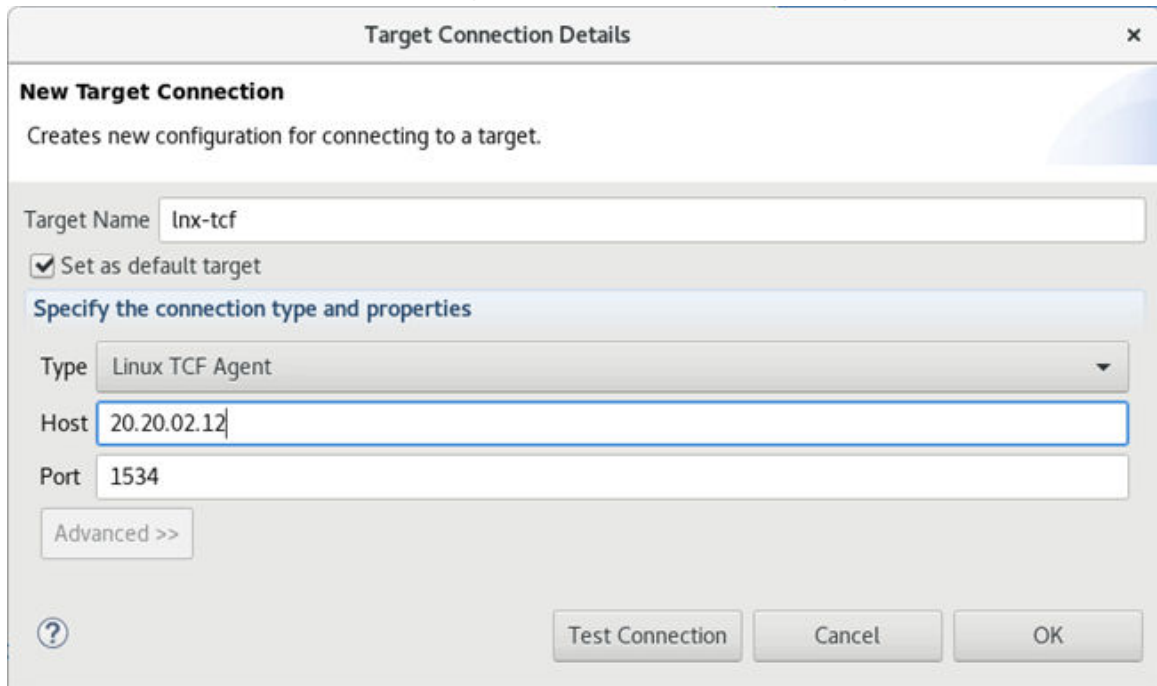
```

5. Execute the following commands:

```
cd /mnt/sd-mmcbllk0p1
```

6. Run `ifconfig` to get the IP address of the target card. The IP address is used to set up a TCF agent connection in Vitis IDE. The target needs to connect to the network assigned IP address.

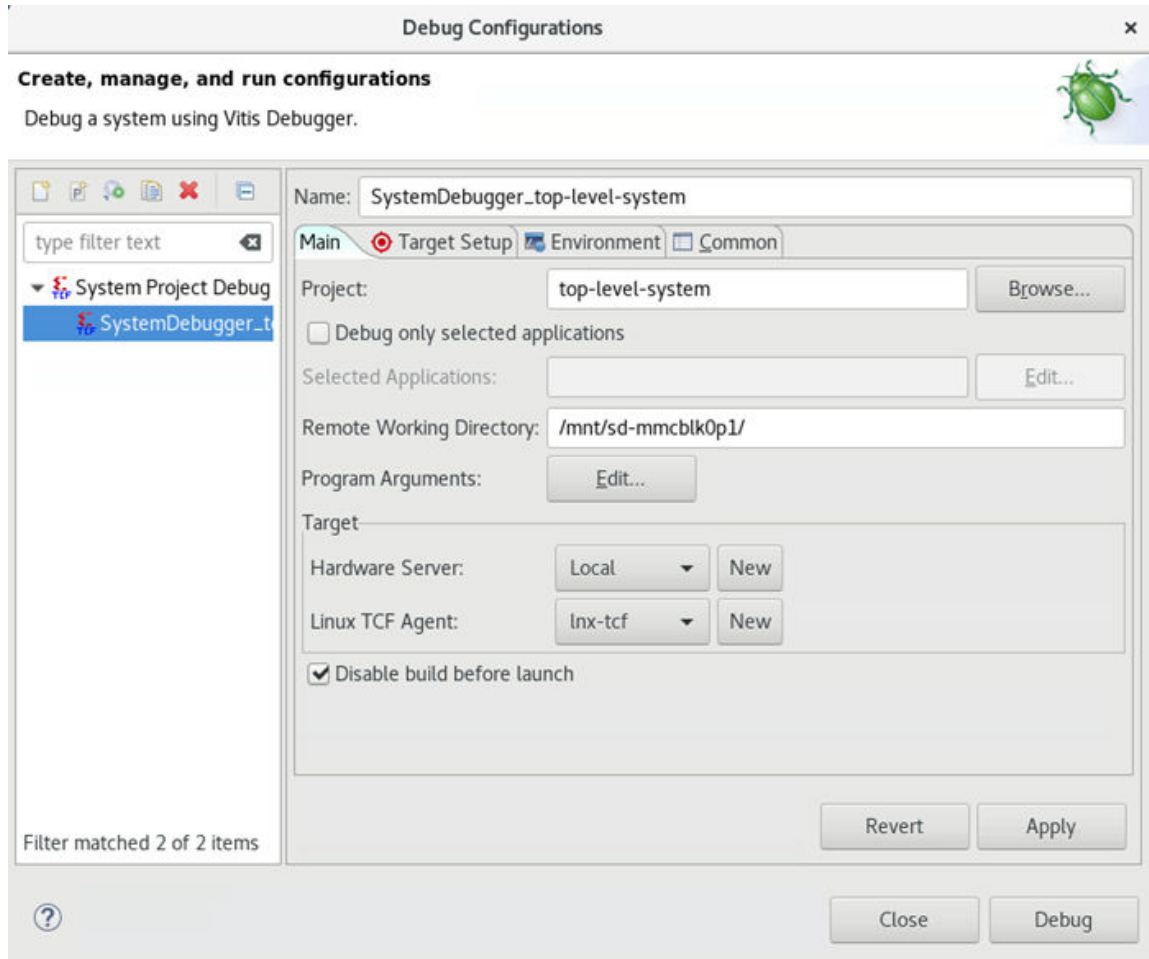
7. Create a target connection to the remote accelerator card. Select the **Window → Show view → Xilinx → Target connections** command to open the Target Connections view.
8. In the Target Connections view, right-click on the **Linux TCF Agent** and select the **New Target** command to open the New Target Connection dialog box.
9. Specify the Target Name, enable the **Set as default target** check box, and specify the Host IP address of the accelerator card that you obtained in an earlier step.



The image shows the 'Target Connection Details' dialog box. It has a title bar with a close button. The main area is titled 'New Target Connection' with a subtitle 'Creates new configuration for connecting to a target.' Below this, there is a 'Target Name' text field containing 'lnx-tcf'. A checkbox labeled 'Set as default target' is checked. A section titled 'Specify the connection type and properties' contains a 'Type' dropdown menu set to 'Linux TCF Agent', a 'Host' text field containing '20.20.02.12', and a 'Port' text field containing '1534'. At the bottom left is an 'Advanced >>' button. At the bottom right are three buttons: 'Test Connection', 'Cancel', and 'OK'. A help icon (?) is located at the bottom left of the dialog area.

10. Click **OK** to close the dialog box and continue.
11. Right-click on the top-level system project and select the **Debug As → Debug Configurations** command.

This opens the Debug Configurations dialog box to let you set up the tool. For the Hardware builds, you will need to create two *launch* configurations: one for the top-level system project, and a second for the PS application.
12. In the Debug Configurations dialog box select the **New Launch Configuration** (📄) command to open the Debug Configurations dialog box as shown.



Be sure to set the following fields on the dialog box as shown in the preceding figure.

- Remote Working Directory: Specify the remote mount location from the accelerator card as determined in an earlier step.
- Linux TCF Agent: Select the new agent you built with the specified IP address for the accelerator card.
- Disable build before launch: This is necessary because without this step, the tool will try to build your system before running the application.

13. Select **Apply** to apply your changes and select **Debug** to start the process.

This opens the Debug perspective in the Vitis IDE, and connects to the PS application and AI Engine graph running on their respective cores in the QEMU. The application automatically breaks at the `main()` function for all the ELF files.

From this point you can do all the debug activities such as, step in, step over, viewing variables, apply break points in the emulation environment. See [Using the Debug Environment](#) for more information.

Bare-Metal Debug from the Vitis IDE

With the bare-metal system project built as described in [Building a Bare-metal AI Engine in the Vitis IDE](#), you must use the following steps to debug both the AI Engine graph, together with the bare-metal PS application on the Hardware build target.

This process is a bit more complex than debugging a Linux application in the Vitis IDE. In that case the PS host application was built as a part of the top-level system project. Here the PS application is built as a standalone project, and must be separately included in the debug configurations when launching the debug environment. This is detailed in the following steps.

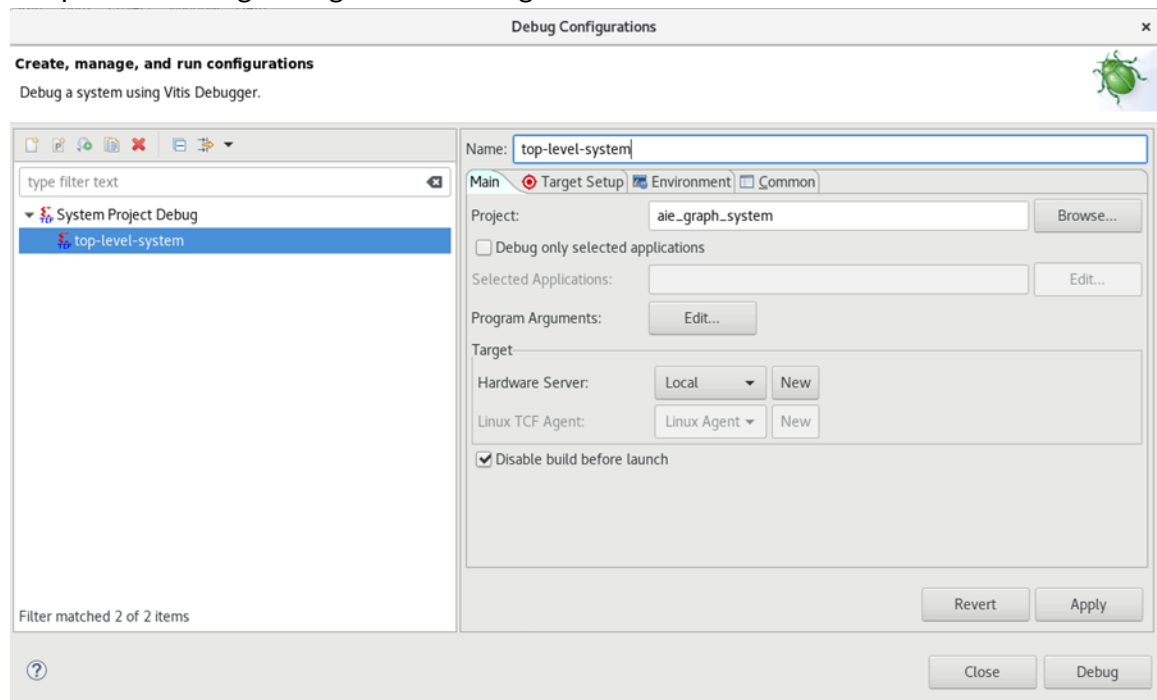
1. Right-click on the top-level system project and select the **Debug As → Debug Configurations** command.

This opens the Debug Configurations dialog box to let you set up the tool.



IMPORTANT! For the Hardware build, you will need to create two Debug configurations: one for the top-level system project, and a second for the bare-metal PS application.

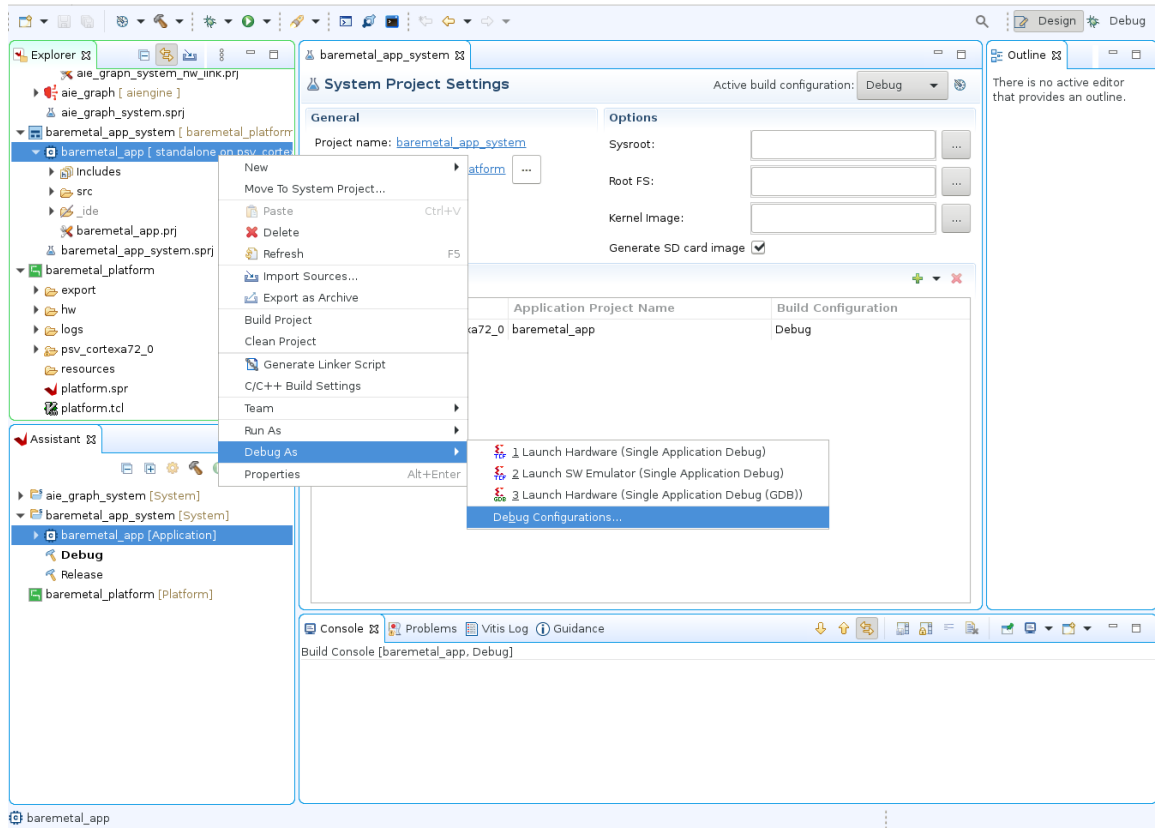
2. In the Debug Configurations dialog box select the **New Launch Configuration** (📄) command to open the Debug Configurations dialog box as shown.



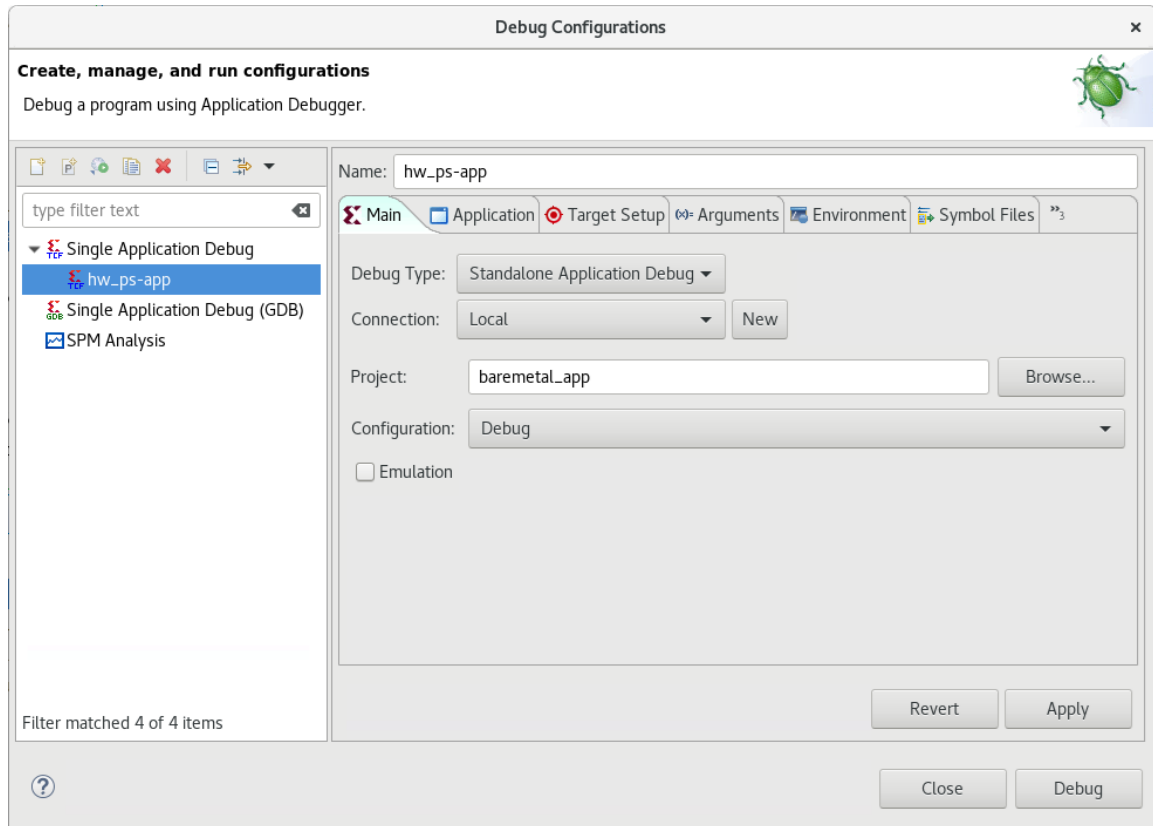
Notice the following fields on the Debug Configurations dialog box:

- **Project:** Reflects the name of the top-level system project which includes the AI Engine graph application, the PL kernels, and the HW-Link projects.
- **Hardware Server:** Specifies a local connection to the board. You can configure this differently for a remotely connected board.

- Linux TCF Agent: Is disabled for bare-metal systems.
 - Disable build before launch: Enable this to prevent the tool from building your system before launching the application.
3. Select **Apply** to save and apply your changes, and select **Close** to close the dialog box.
 4. Right-click the **baremetal_app** project in the Explorer view, and select **Debug As → Debug Configurations** as shown in the following image.



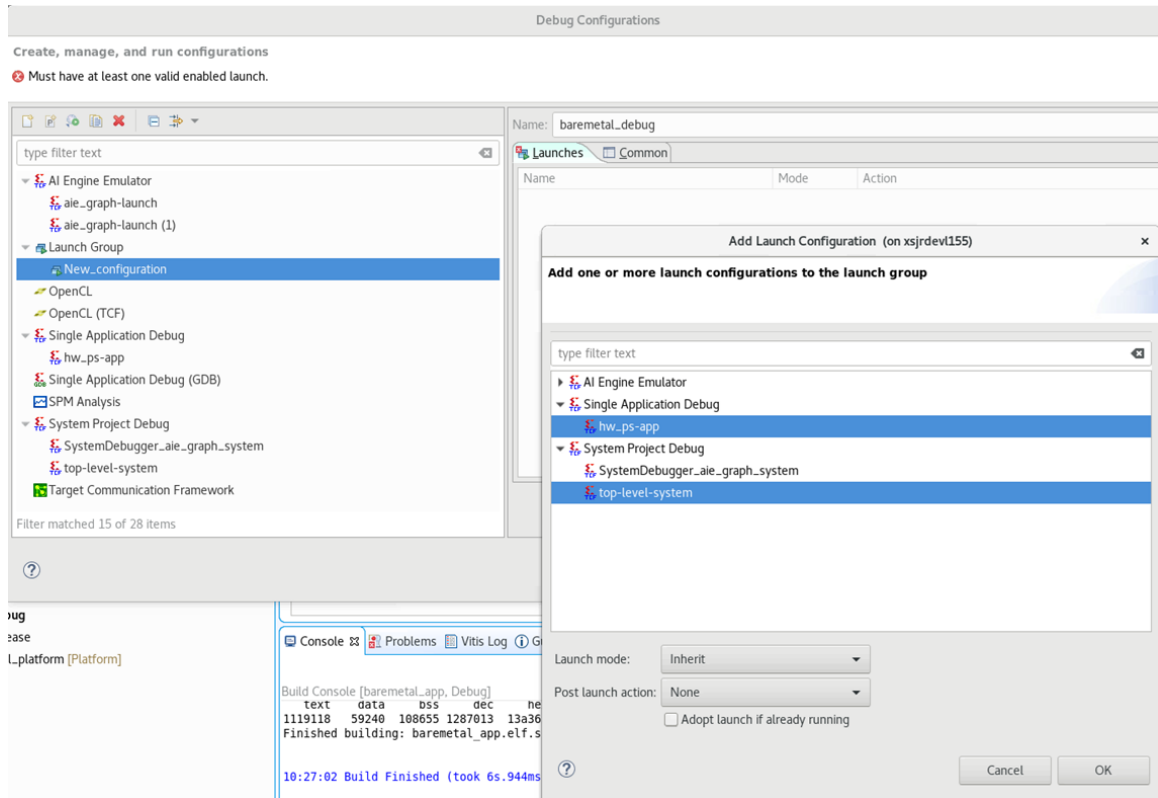
5. The Single Application Debug Configurations dialog box opens as shown in the following figure.



6. Specify a **Name** to identify the configuration as applying to the PS application. The **Connection** is set up automatically for the local hardware.
7. Change to select the **Target Setup** tab of the Debug Configurations dialog box, and deselect the **Reset entire system** and **Program Device** check boxes in the dialog box.
8. Click **Apply** to proceed.
9. You also need to create a launch group containing the two debug configurations you just created. The launch group lets you launch multiple configurations as a group. Use the main toolbar menu command by selecting **Debug → Debug Configurations** to open the Debug Configurations dialog box, and then select **Launch Group** as shown.

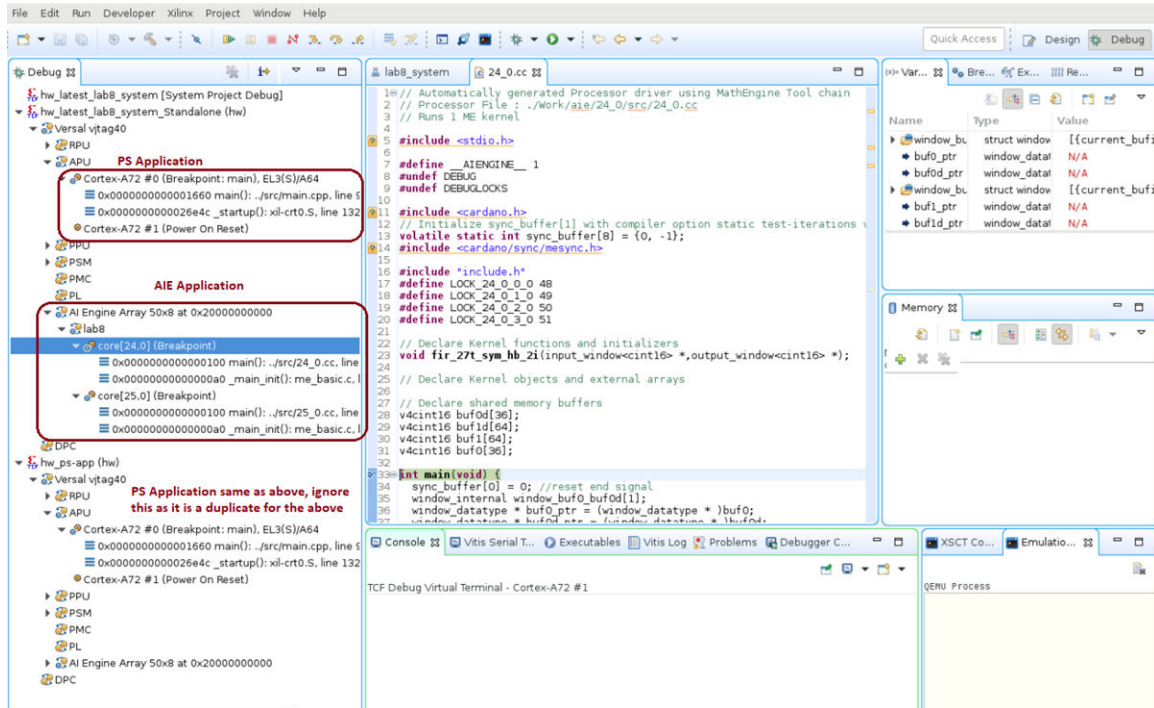


IMPORTANT! You must use the main toolbar menu **Debug** command because it provides the complete set of Debug Configuration options rather than the reduced options found in the Explorer or Assistant views, which are limited by your selections.



Click on **Launch Group**, and provide a **Name** for the new group. Click **Add** the top-level project and PS application Debug configurations to the group as shown in the previous image. Click **OK** to create the launch group.

10. The Launch Group is displayed with both debug configurations added. Click **Debug** to launch the Debug perspective.
11. Because you are launching both the top-level project and PS debug configurations on a single target connection, the Vitis IDE will display a Launch Config Conflict message. Click **No** on the conflict message to proceed.
12. The Debug perspective is opened, as shown in the following image. Because there are two debug configurations, you will see two instances of the PS application. You can ignore the second instance of the PS application.



From this point you can do all the debug activities: step in, step over, viewing variables, apply break points in the emulation environment. Refer to [Using the Debug Environment](#) for more information.

Launching Debug from the Command Line

Debugging projects built from the command line is a special challenge because the various elements of the system, the compiled graph application (`libadf.a`), the device binary (XCLBIN), and the top-level application (`host.cpp`), must be gathered together and presented as a system. In the Vitis IDE this is done through the system-level project that is created when you first created the graph project. In the command line flow, this service is provided by the Vitis compiler package process. As described in [Packaging](#), the `v++ --package` command gathers the elements of the system together to create a boot container for either hardware emulation or hardware builds.

In addition, you must separately manage the different elements of the debug environment, running the QEMU emulation environment, or running the `xrt_server` for connecting to the hardware, and launching the Vitis IDE to run the debug process. The steps and requirements for command line debug are detailed in the following topics.

Software Emulation from Command Line

With the Software Emulation build complete, you can use the following steps to debug the system. This process will launch a new terminal that will allow for `gdb` commands to be used, as well as visual of the files for code stepping.

1. Launch QEMU emulator environment using `launch_sw_emu.sh` script that is generated during the `v++ --package` process.
 2. Using a specific command-line option, `-kernel-dbg` and set it to `true`.
 3. Specify the kernels, PL or AI Engine kernels.
1. To launch the emulation environment with debug use the following command from your build directory.

```
./emulation/launch_sw_emu.sh -kernel-dbg true
```

Where,

- `./emulation` is the output directory of the packaging process.
- `-kernel-dbg true` will setup the emulator to run `gdb` at the execution of the application.

2. Run the following commands in the QEMU shell once you see the `qemu%` prompt.

```
export LD_LIBRARY_PATH=/mnt/sd*1:/tmp:$LD_LIBRARY_PATH
export XCL_EMULATION_MODE=sw_emu
export XILINX_XRT=/usr
```

3. Run the PS application. For example, `./host.exe a.xclbin`.

This will start running the host application and launch `gdb` in a separate terminal. Here you can do all the debug activities such as, breakpoint, next (step), and continue on PL kernels and AI Engine kernels.

Note: To get a textual user interface (TUI) of `gdb`, enable with `Ctrl+x Ctrl+a`.

Hardware Emulation Debug from the Command Line

With the Hardware Emulation build complete, including the AI Engine graph, the PL region kernels, and the PS application, you can use the following steps to debug the system design. This process makes use of the Vitis IDE to launch the debug environment from the command line.

- Launch the QEMU emulator environment using the `launch_hw_emu.sh` script, that is generated during the `--package` process.
- Launch the Vitis IDE in standalone debug mode using the `vitis -debug` option.
- Configure the debug environment to connect to the PS and AI Engine domains within the system.

1. For AI Engine platforms, the files required for emulation of the system are defined by the `--package` command, including the emulation script. To launch the emulation environment use the following command from your build directory:

```
./emulation/launch_hw_emu.sh -pid-file emulation.pid -no-reboot \
-add-env ENABLE_RDWR_DEBUG=true -add-env RDWR_DEBUG_PORT=10100 -forward-
port 1440 1534
```

Where:

- `./emulation` is the output directory of the package process as described in [Packaging](#), and is where the `launch_hw_emu.sh` script can be found.
- `-add-env RDWR_DEBUG_PORT=${aie_mem_sock_port}` defines the port for communicating with the AI Engine domain. In the previous example, it is 10100.
- `-forward-port ${linux_tcf_agent_port} 1534` defines the port for the Linux TCF agent. In the previous example, it is 1440, which is the default.



TIP: Any free ports can be used for `aie_mem_sock_port` and `linux_tcf_agent_port` in the above command template. However, these port are mandatory for enabling the AI Engine application and Linux application debug respectively.

This command launches the emulator and then waits until Linux is booted within the QEMU. The QEMU shell shows a transcript of the QEMU launch and Linux boot process. You can tell when the process has completed when the `qemu%` prompt is displayed. At that time you are ready to proceed.

2. Run the following command in the QEMU shell at the `qemu%` prompt:

```
source /mnt/sd-mmcbk0p1/init.sh
```

Note: The mount drive is reported in your QEMU transcript, or can be found by typing `mount` at the `qemu%` prompt. The drive above should be standard for Xilinx platforms.

3. In a second terminal window, launch the XRT server application using the following command:

```
xrt_server -I300 -S -s tcp::4352
```

where:

- `-I300` defines an idle timeout, in which the server quits if there is no response.
- `-S` specifies print server properties in JSON format to stdout.
- `-s tcp::${xrt_server_port}` defines the agent listening protocol and port. In the previous example, it is 4352, but can be any free port.

4. Create a Tcl script with the name `aie_app_debug.tcl` to set up the AI Engine debug environment:

```
#Set up the required environment
# The aie_mem_socket and xrt_server ports must match what was specified
in earlier commands.
set aie_work_dir "<AIE_Project>/Work"
set aie_mem_sock_port "10100"
set xrt_server_port "4352"
set app_name "aie_graph"

#Echo the environment setup
puts "Vitis install: $XILINX_VITIS"
puts "Application: $app_name, Work Directory: $aie_work_dir"
puts "XRT Server Port: $xrt_server_port, AIE Port: $aie_mem_sock_port"

#Set up AIE Debug environment
set source_tcl_cmd "source $XILINX_VITIS/scripts/vitis/util/
aie_debug_init.tcl"
puts "$source_tcl_cmd"
eval $source_tcl_cmd

##run the command to connect and display debug targets
set aie_debug_cmd "init_aie_debug -work-dir $aie_work_dir -url
tcp::$xrt_server_port \
-memsock-url localhost:$aie_mem_sock_port -sim-type memserver -name
$app_name -full-program"
puts "$aie_debug_cmd"
eval $aie_debug_cmd
```

Note: This script requires the `$XILINX_VITIS` environment variable to be set up.

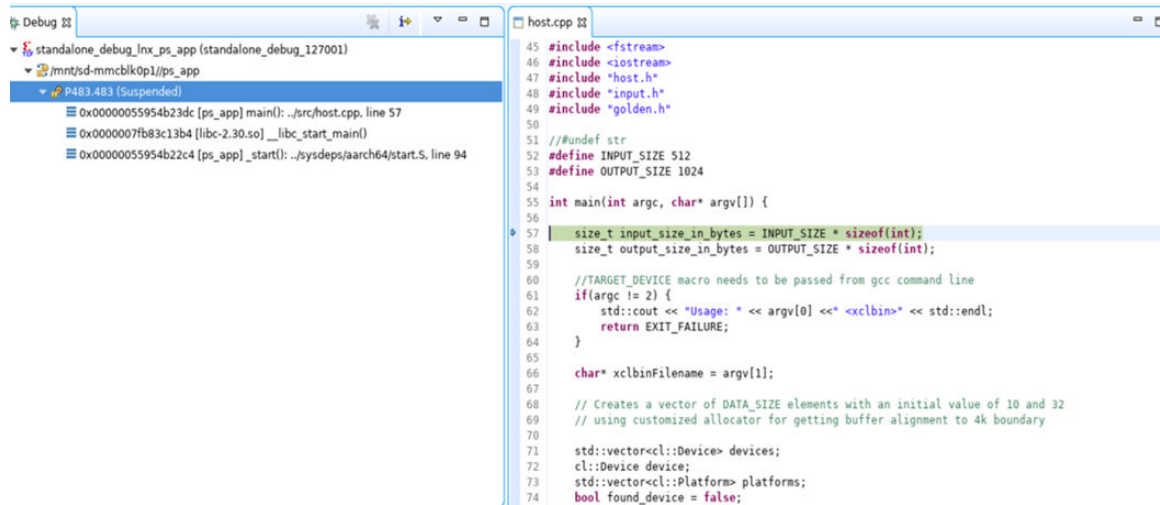
5. After the QEMU environment and `xrt_server` are up and running, you can launch the Vitis IDE in stand-alone debug mode *in a third terminal window*:

```
vitis -debug -flow embedded_accel -target hw_emu -exe ./ps_app \
-program-args ${xcl_bin_dir}/binary_container_1.xclbin -port 1440
```

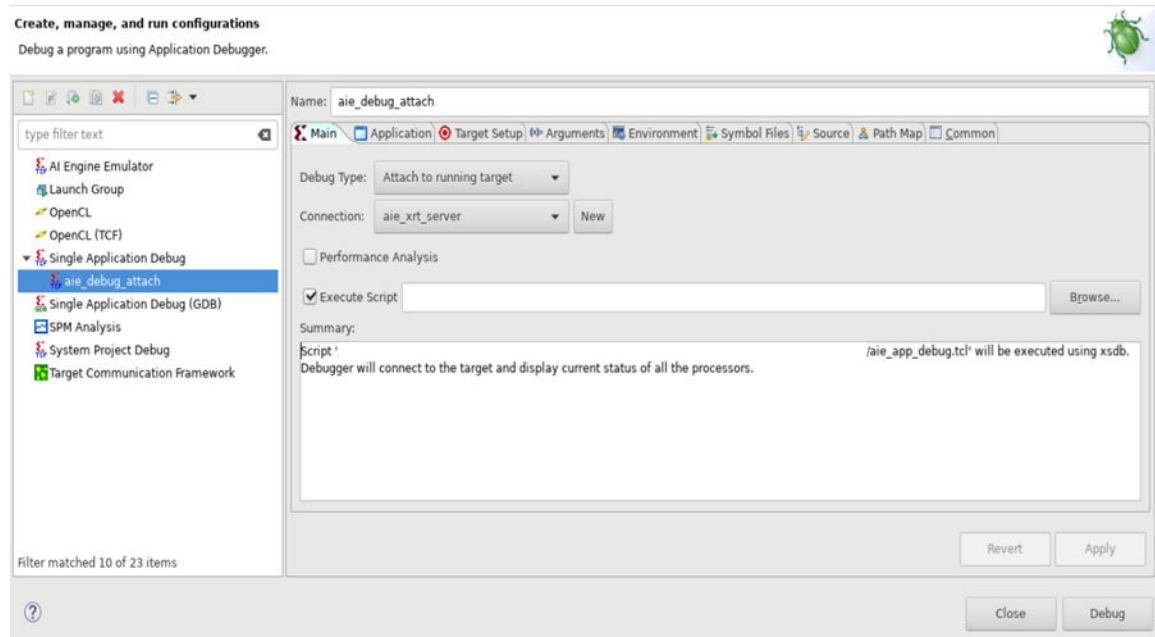
where:

- `vitis -debug`: Launches the Vitis IDE in stand-alone debug mode.
- `-flow embedded_accel`: Specifies the embedded processor application acceleration flow.
- `-target hw_emu`: Indicates the target build being debugged.
- `-exe ./ps_app`: Indicates the PS application to run and debug.
- `-program-args ${xcl_bin_dir}/binary_container_1.xclbin`: Refers to the location of the XCLBIN file to be loaded as an argument to the executable.
- `-port 1440`: Specifies the `${linux_tcf_agent_port}` as discussed previously.

This opens the Vitis IDE with the Debug perspective displayed, and the debug configuration for the PS application loaded.

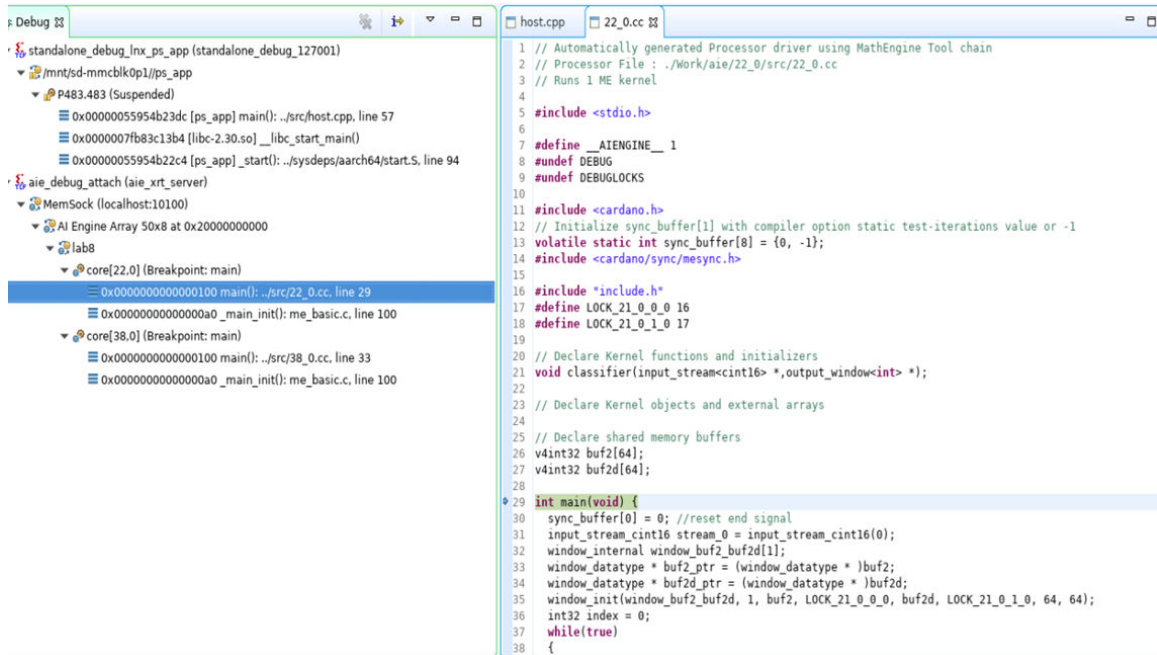


6. In the Debug perspective of the Vitis IDE, create a new target connection of type Hardware Server with the name `aie_xrt_server`. Specify `localhost` as host, and `xrt_server_port`, 4352 in the previous example, as the port.
7. Create a new Debug configuration of type Single Application Debug as shown.



- **Debug Type:** Attach to running target
 - **Connection:** `aie_xrt_server`
 - **Execute Script:** Specify the path to `aie_app_debug.tcl` defined in Step 5.
8. Press **Debug** to proceed.

This connects to the PS application and AI Engine graph running on their respective cores in the QEMU. The application automatically breaks at the `main()` function for all the ELF files.



From this point you can do all the debug activities like step in/step over/viewing variables/plant break points in the emulation environment. Refer to [Using the Debug Environment](#) for more information.

Debugging Only the AI Engine Graph

To debug just the AI Engine domain without the PS, follow steps 1 to 4 in [Hardware Emulation Debug from the Command Line](#). Then, instead of step 5, continue by using the following step:

1. After the QEMU environment and xrt_server are up and running, you can launch the Vitis IDE in standalone debug mode *in a third terminal window*:

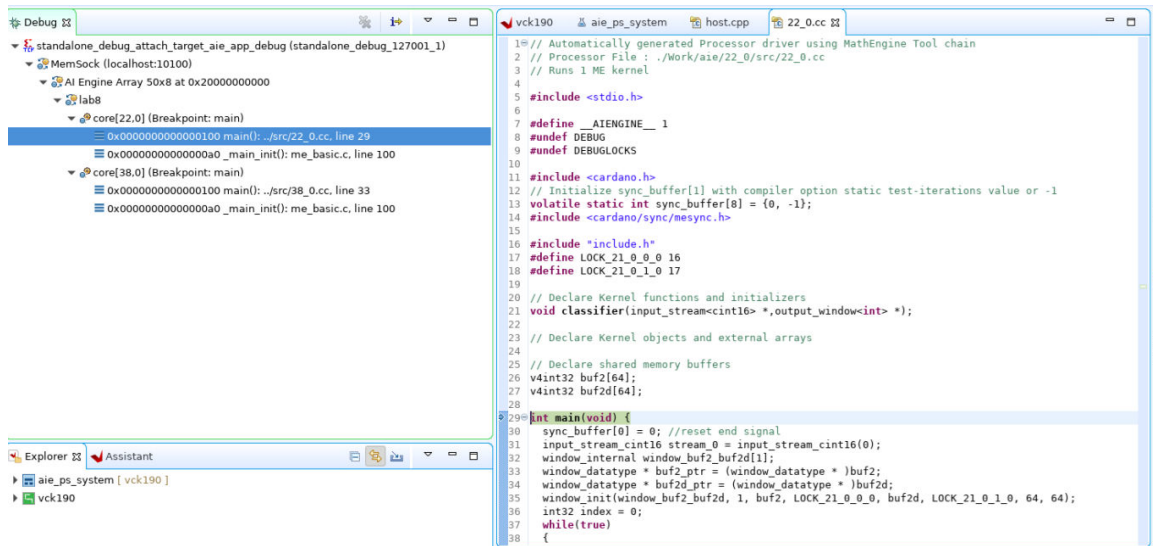
```
vitis -debug -flow embedded -os baremetal -port 4352 \
-launch-script <aie_project>/aie_app_debug.tcl
```

where:

- **vitis -debug**: Launches the Vitis IDE in standalone debug mode.
- **-flow embedded**: Specifies the embedded processor flow for the AI Engine processor.
- **-os baremetal**: For bare-metal OS of the AI Engine domain.
- **-port 4352**: Specifies the xrt_server port as discussed in Step 3.
- **-launch-script <aie_project>/aie_app_debug.tcl**: Specifies the Tcl script from Step 4, which sets up the AI Engine debug environment.

This opens the Vitis IDE with the Debug perspective displayed, and the debug configuration for the AI Engine application loaded.

Figure 56: AI Engine Debug Configuration



Hardware Debug from the Command Line

With the Hardware build complete, including the AI Engine graph, the PL region kernels, and the PS application, you can use the following steps to debug the system design. This process makes use of the Vitis IDE to launch the debug environment from the command line.

- Launch the Vitis IDE in standalone debug mode using the `vitis -debug` option.
 - Connect to the board using the hardware server (`hw_server`) command.
 - Configure the debug environment for the AI Engine domain.
1. Burn `<project>/Hardware/package/sd_card.img` to a physical SD card. This creates a bootable medium for your target platform.
 2. Insert the SD card into the card reader of the [VCK190](#) evaluation kit.
 3. Change the boot-mode settings of the card to SD boot mode, and power up the board.
 4. Run `ifconfig` to get the IP address of the target card. The IP address is used to set up a the hardware serve connection in Vitis IDE. The target needs to connect to the network assigned IP address.
 5. In a terminal window, launch the hardware server to connect to the board.
 6. Create a Tcl script with the name `aie_app_debug.tcl` to set up the AI Engine debug environment:

```
#Set up the required environment
# The aie_mem_socket and xrt_server ports must match what was specified
in earlier commands.
set aie_work_dir "<AIE-Project>/Work"
set hw_server_host "gandalf"
set app_name "aie_graph"
```

```
#Printing the information
puts "Install: $$XILINX_VITIS"
puts "Application: $app_name, Work Directory: $aie_work_dir"
puts "Hardware Server: $hw_server_host"

set source_tcl_cmd "source ${vitis_install}/scripts/vitis/util/
aie_debug_init.tcl"
puts "$source_tcl_cmd"
eval $source_tcl_cmd

##run the command to connect and display debug targets
set aie_debug_cmd "init_aie_debug -work-dir $aie_work_dir -url
tcp:$hw_server_host:3121 -jtag -name $app_name"
puts "$aie_debug_cmd"
eval $aie_debug_cmd
```

Note: This script requires setting up the \$XILINX_VITIS environment variable.

7. After the board system and hw_server are up and running, launch the Vitis IDE in standalone debug mode *in a second terminal window*:

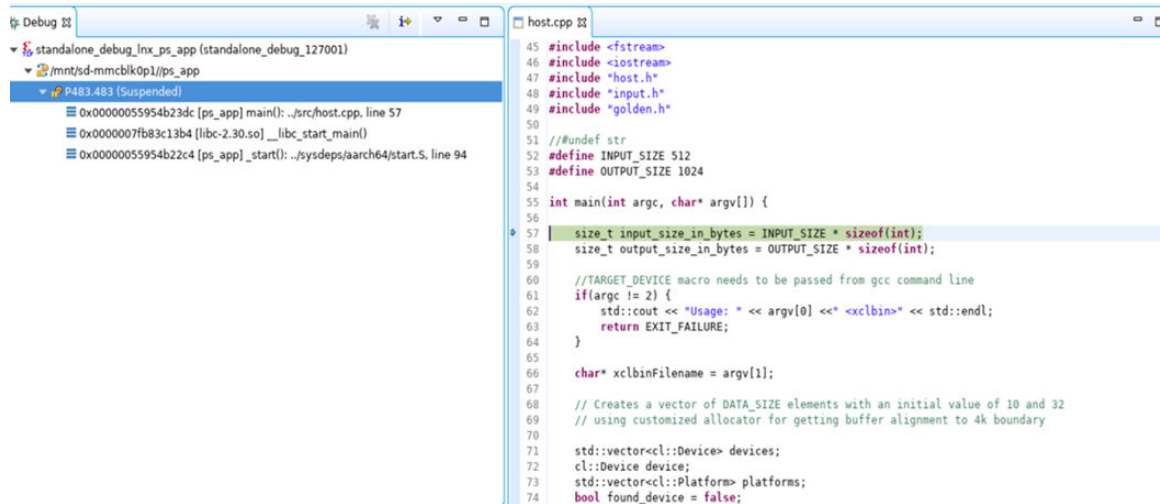
```
vitis -debug -flow embedded_accel -target hw -exe ./ps_app \
-program-args ${xcl_bin_dir}/binary_container_1.xclbin -host $
{linux_tcf_agent_host} \
-port 1534
```

where:

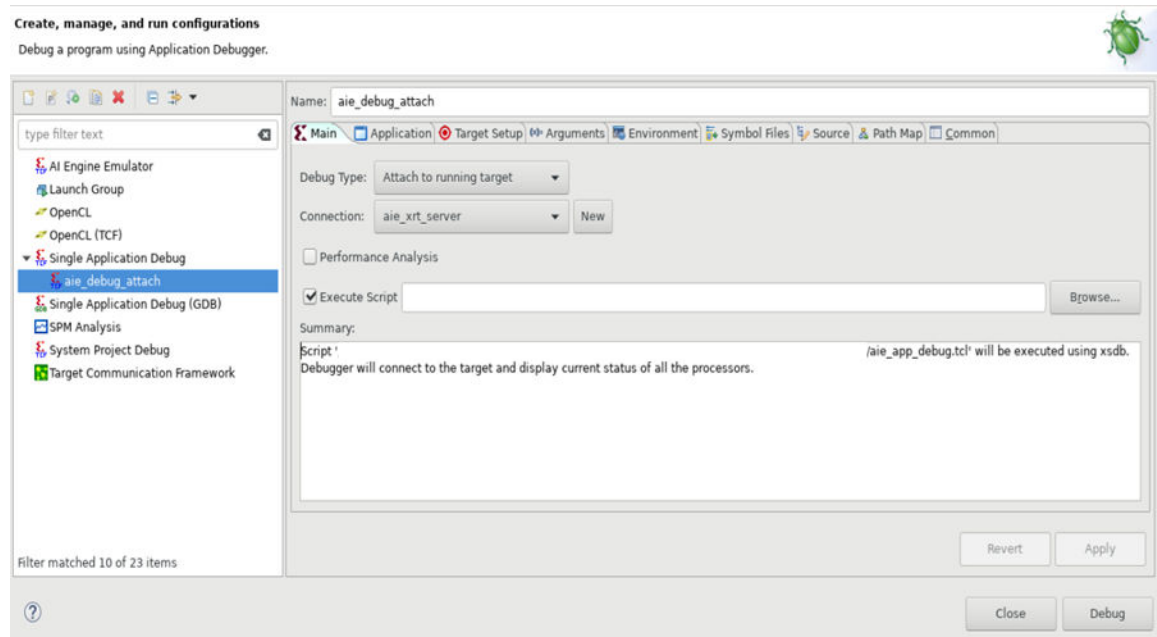
- **vitis -debug:** Launches the Vitis IDE in standalone debug mode.
- **-flow embedded_accel:** Specifies the embedded processor application acceleration flow.
- **-target hw:** Defines the hardware build as being debugged.
- **-exe ./ps_app:** Indicates the PS application to run and debug.
- **-program-args \${xcl_bin_dir}/binary_container_1.xclbin:** Refers to the location of the XCLBIN file to be loaded as an argument to the executable.
- **-host \${linux_tcf_agent_host}:** Specifies the host name or IP address obtained from Linux running on the board, \${linux_tcf_agent_host}.
- **-port 1534:** Specifies the port the Linux TCF agent is running on, \${linux_tcf_agent_port}. In this case 1534.

This opens the Vitis IDE with the Debug perspective displayed, and the debug configuration for the PS application loaded.

Figure 57: PS Debug Configuration



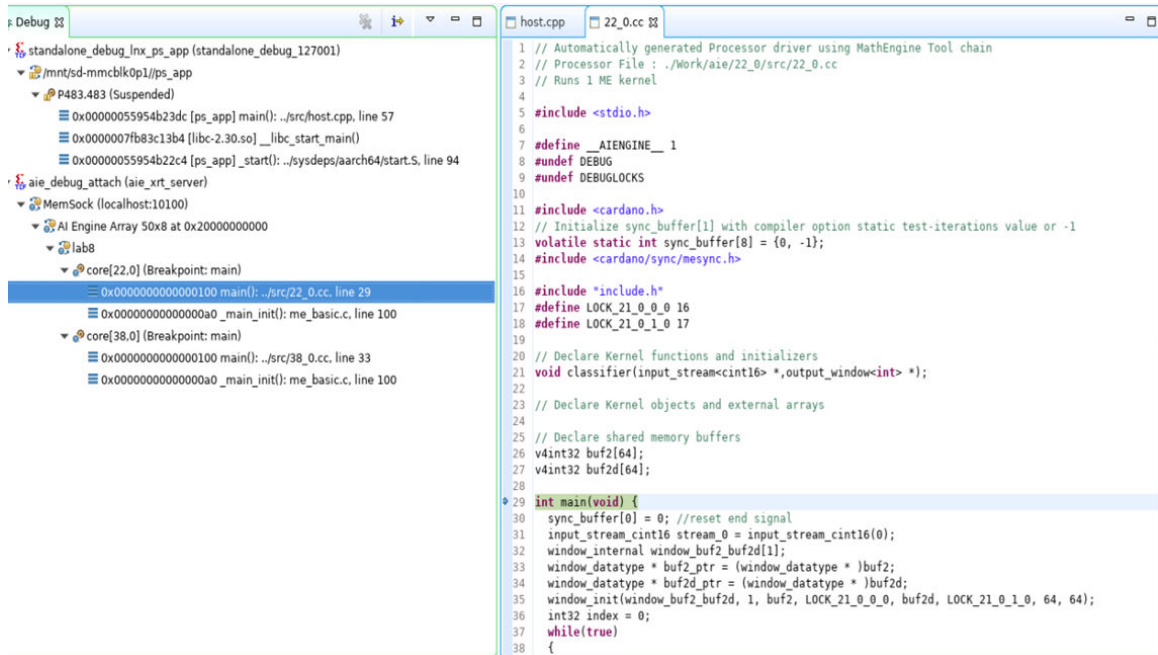
8. Create a new debug configuration of type Single Application Debug as shown.



- **Debug Type:** Attach to running target
- **Connection:** hw_server defined in Step 5.
- **Execute Script:** Specify the path to aie_app_debug.tcl defined in Step 6.

9. Click **Debug** to proceed.

This connects to the PS application and AI Engine graph running on their respective cores in the QEMU. The application automatically breaks at the `main()` function for all the ELF files.



From this point you can do all the debug activities such as step in/step over/viewing variables/ plant break points in the debug environment. Refer to [Using the Debug Environment](#) for more information.

Debugging Only the AI Engine Hardware

To debug just the AI Engine domain without the PS, follow steps 1 to 6 in [Hardware Debug from the Command Line](#). Then, instead of step 7, continue by using the following step:

1. After the board and the hw_server are up and running, launch the Vitis IDE in stand-alone debug mode *in a separate terminal window*:

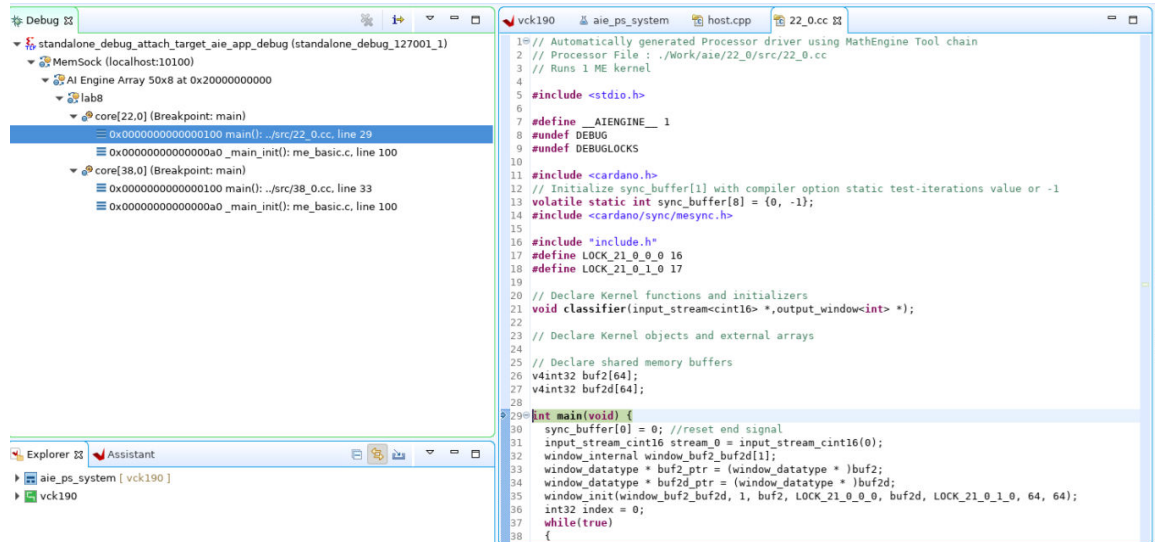
```
vitis -debug -flow embedded -os baremetal -host gandalf \
-launch-script <aie_project>/aie_app_debug.tcl
```

Where:

- **vitis -debug**: Launches the Vitis IDE in standalone debug mode.
- **-flow embedded**: Specifies the embedded processor flow for the AI Engine processor.
- **-os baremetal**: For the baremetal OS of the AI Engine domain.
- **-host gandalf**: Specifies the host name of the hw_server (see Step 5 of [Hardware Debug from the Command Line](#)).
- **-launch-script <aie_project>/aie_app_debug.tcl**: Specifies the Tcl script from Step 6 of [Hardware Debug from the Command Line](#), which sets up the AI Engine debug environment.

This opens the Vitis IDE, displays the Debug perspective, and loads the debug configuration for the AI Engine application.

Figure 58: AI Engine Debug Configuration



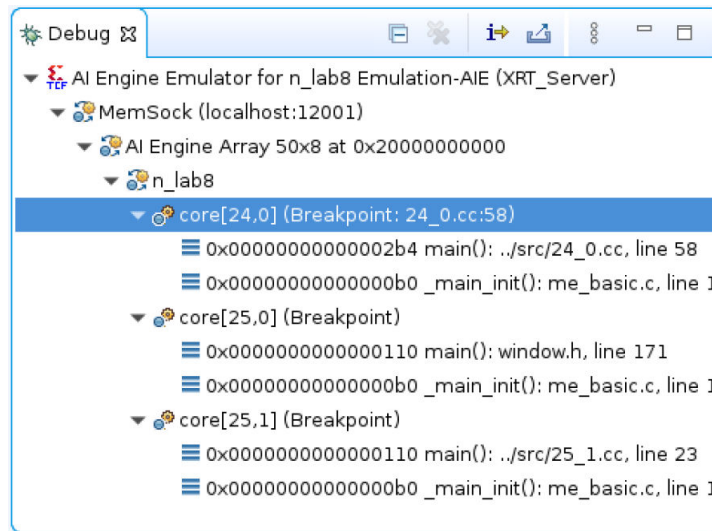
Using the Debug Environment

The Vitis IDE debug environment has many features found in traditional GUI-based debug environments, such as GDB. You can add break points to the code, step over or step into specific lines of codes, loops, or functions, and examine the state of variables and force them to specific values. These are just a few of the features in the Vitis IDE debug environment.

After you have launched the Debug perspective, you will see several windows or views displayed, such as the Debug view in the upper left of the display, as shown in the following figure. During the debug process, several windows show the debug state that include code at breakpoint, step over states, breakpoints view, variables view, registers view, disassemble view, and pipeline view (single kernel only).

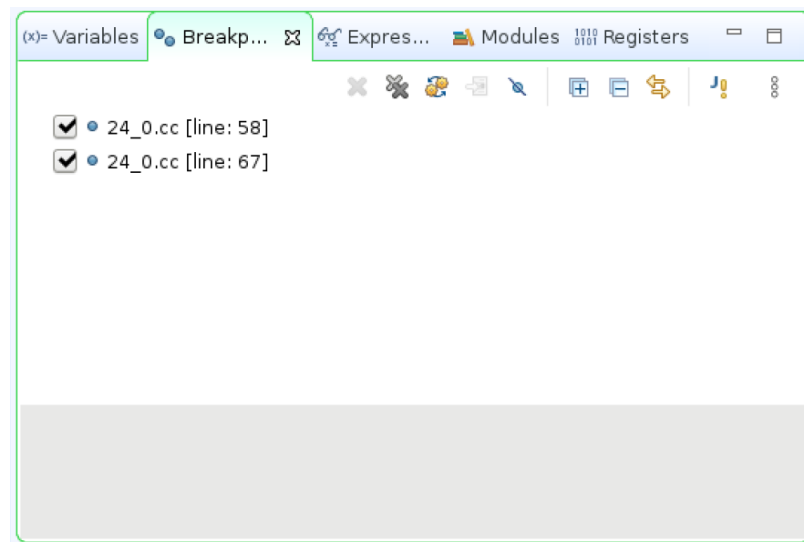
The Debug view shows states of cores that are under debug. It shows where (which file and which line of source code of the file) debugger stops at and with what action it is taking (breakpoint/step over/...) as shown in the following figure.

Figure 59: Debug View



The following figure shows the Breakpoints information with current setup breakpoints. The square with a check mark indicates the breakpoint is enabled. Click on the check mark to clear the check mark and disable breakpoint during debugging. This lets you manage breakpoints without having to remove them or add them back into the code.

Figure 60: Breakpoints View



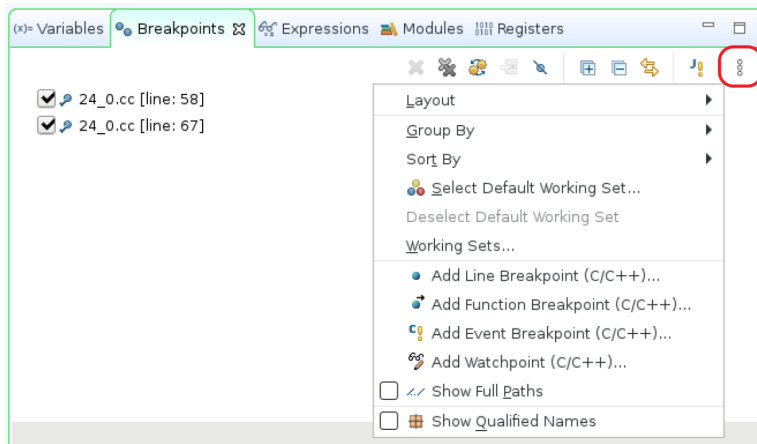
IMPORTANT! Each AI Engine tile supports four breakpoints when debugging on the AI Engine simulator, or co-simulation. The TCF framework stops at the AI Engine kernel `main()` by default. Breakpoints attached to a `while` statement consume two breakpoint resources. A workaround is to attach the breakpoint inside the `while` loop. This only consumes one breakpoint.

Watchpoints

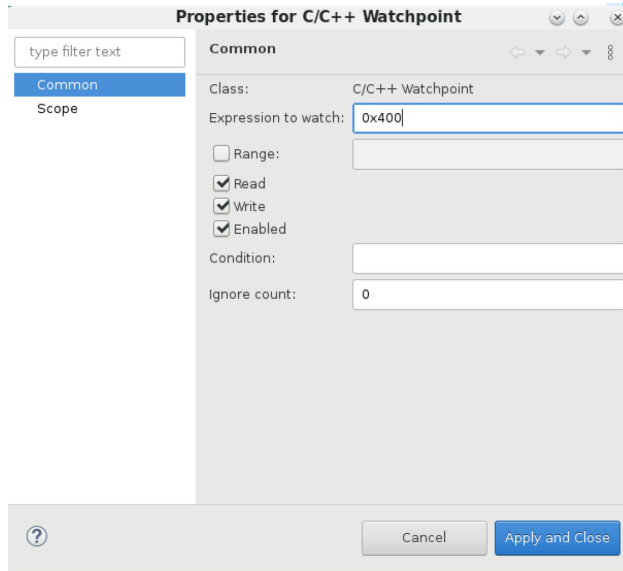
A watchpoint is a type of breakpoint. Watchpoints can be used to stop the execution of a core when the value at an address changes. These are useful in determining the place where the value changes. For example, a variable value can be overwritten unexpectedly, which can break the program flow. Watchpoints help in detecting such cases.

AI Engine architecture supports read/write watchpoints. This means, a watchpoint is triggered on a read or write access, or both (based on your configuration). Each AI Engine tile supports two watchpoints per memory bank. Because every AI Engine core (excluding tiles on boundaries) has access to memory banks in adjacent tiles, a maximum of eight watchpoints can be used per core. However, because the memory banks are shared, the number of available watchpoints per core depends on how many watchpoints are already used from the memory bank. For example, if a core uses all eight watchpoints from its four adjacent memory banks, the other cores which share those four memory banks cannot use watchpoints from the shared memory banks. Debugger keeps track of watchpoints that are allocated from each bank, and throws an error if there are no unused watchpoints in that tile.

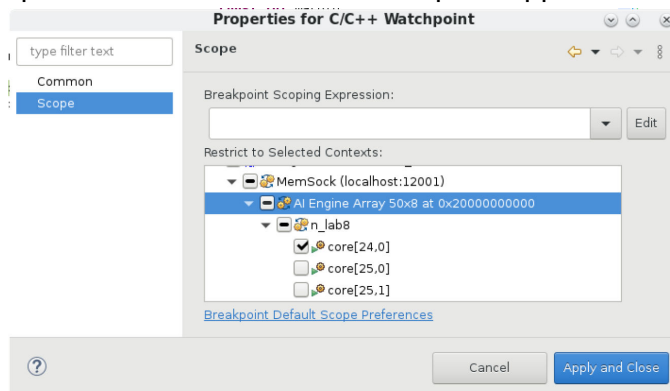
1. To add a watchpoint in the Vitis IDE, click on the three dots at top-right corner of the Breakpoints view and select **Add Watchpoint (C/C++)**.



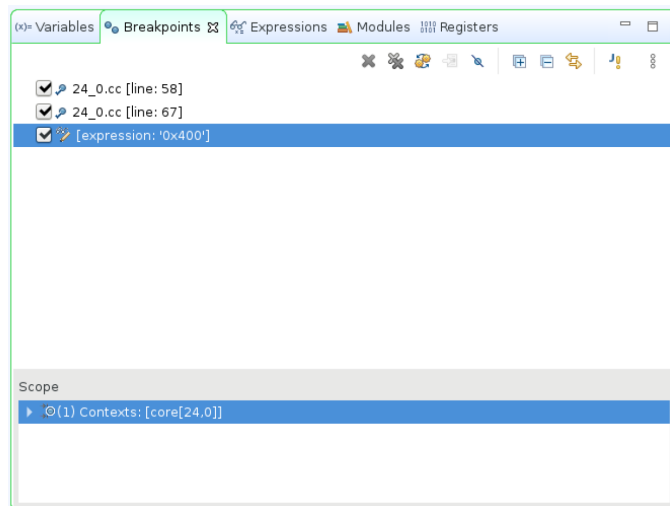
2. In the Watchpoint Properties dialog box, enter the memory address or the variable name of interest. Select read/write modes. If **Enabled** is unchecked, the watchpoint is not enabled regardless of the read/write mode during the debug session. This is to preserve the watchpoint configuration without removing it and adding it back when needed.



- By default, watchpoints are added to all debug cores. To apply the watchpoint to a specific core, click on **Scope** in the left pane of the Watchpoint Properties window, and select only specific cores to which the watchpoint applies.



- Verify the watchpoint in the Breakpoints view.



Triggering of Watchpoints

A watchpoint is triggered on a read access or write access, or both, based on how the watchpoint is configured. A triggered watchpoint causes the core to stop at the instruction that accessed the address. Debugger detects the core has stopped because of the watchpoint and reports that the core has stopped because of a watchpoint.



IMPORTANT!

- Watchpoints work on hardware only. AI Engine system C model is not supported.
- Memory banks are shared. It may not be possible to add watchpoints for a core in a specific bank, if another core uses both the watchpoints from that bank.
- Watchpoints must not be set for memory addresses which fall in unused tiles/memory banks. Setting watchpoints to unused tiles can cause AXI errors. Used AI Engine and memory tiles can be found at `Work/aie/active_cores.json`.
- Watchpoints are triggered for memory access in full 16-byte aligned address range.
- Debugger uses two broadcast channels to handle watchpoint events. When enabling watchpoints during debug, ensure that there are no conflicts in using these two broadcast channels.

The Variables view shows the kernel variables values. Depending on variable type, clicking on a variable shows its type, value, and the address of the variable. For array/structure variables, click on the arrow of the variable expands array/structure content of the array.

Figure 61: Variables View

Name	Type	Value
▶ window_buf0_buf0d	struct window_internal [1]	[{current_bufid=0x00000000, i
▶ buf0_ptr	window_datatype *	"\001"
▶ buf0d_ptr	window_datatype *	"\001"
▶ window_buf1_buf1d	struct window_internal [1]	[{current_bufid=0x00000000, i
▶ buf1_ptr	window_datatype *	"\001"
▶ buf1d_ptr	window_datatype *	"\001"
c*	window_datatype	'\000'
lockid_i2_pi0	int	N/A
lockid_i2_po0	int	N/A
▶ input_window_i2_pi0	input_window_cint16 *	{current_bufid=0x00010001, ir
▶ output_window_i2_po0	output_window_cint16 *	{current_bufid=0x00010001, ir
index	int32	N/A
proc_24_0_bounds	int32	14

Hex: 00, Dec: 0, Oct: 0
Bin: 0000,0000
Size: 1 byte, Type: window_datatype
Address: 0x400

Address	0 - 3	4 - 7	8 - B	C - F
0000000000000400	00010001	3ACE0003	802607F7	80000091
0000000000000410	00010001	00010001	00370001	20018000
0000000000000420	E7F77AFE	00C7E7FC	67F78000	00C7D3FB
0000000000000430	E7F78000	00C7BFBF	E7F78000	00C7BFFF

Click on a variable to obtain address information and then click on the + from the Memory view and enter the memory address of the variable. The value of that variable shows up along with other memory content. Variable values can be changed by clicking on the variable's value field, entering the new value, and pressing the Enter key.

You can specify the format of the data presented in the Memory window by clicking on the **New Renderings** tab, specifying the format of the data to present, and click **Add Rendering(s)**.



TIP: You can export the contents of the Memory window by selecting the **Copy to Clipboard** command from the right-click menu to copy and paste the contents into a text editor and save to a file.

Figure 62: Registers View

Name	Hex	Decimal	Description	Mnemonic
r3	00000014	20	General purp	
r4	00000030	48	General purp	
r5	00032460	205920	General purp	
r6	00000000	0	General purp	
r7	00000000	0	General purp	
r8	00000007	7	General purp	
r9	0000000a	10	General purp	
r10	000323e4	205796	General purp	
r11	00000001	1	General purp	
r12	0000000e	14	General purp	
r13	00000003	3	General purp	
r14	0000000b	11	General purp	
r15	00000001	1	General purp	
pc	000002d0	720	Program Cou	
fc	00000320	800	Fetch Counte	

r0: General purpose register 0

In the Registers view, the values are updated during the debug process and are highlighted in the Vitis IDE.

Figure 63: Disassembly View

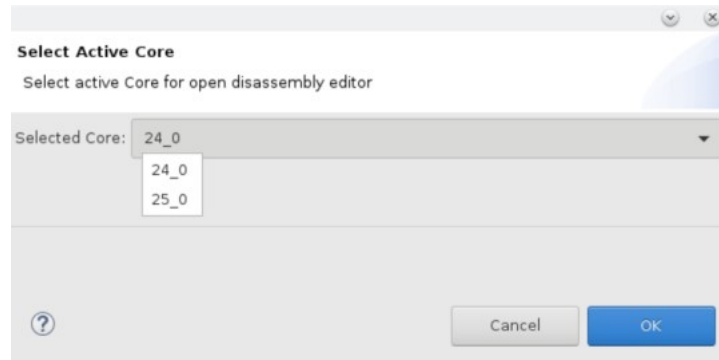
```

000000000000118: 0x38 0x00 0x00 0x00 0x15 0x43 0x00 0x04 0xb0 0x00 0x00 0x3f MOV.s10 r0, #0; NOP; ST ch0, [sp, #-92];
472 | w[i].heads[0] = w[i].buffers[0];
473 | #ifdef NEW_X86SIM
474 | w[i].conn = nullptr;
475 | else
476 | 0x000000000000124: 0xaa 0x1f 0x30 0x4a 0xfc 0x00 0x07 0xf7 ST ch0, [sp, #-184]; MOV.u20 r15, #163840;
477 | w[i].size = size;
478 | 0x00000000000012c: 0xaf 0x0f 0x08 0x00 0x00 0x40 0x07 0xf7 ST r14, [sp, #-76]; MOV.u20 ch1, #1624;
479 | sync_buffer[0] = 0; //reset end signal
480 | input_stream_cint16_stream_0 = input_stream_cint16(0);
481 | window_internal_window_buf2_buf2d[1];
482 | window_datatype * buf2_ptr = (window_datatype * )buf2;
483 | window_datatype * buf2d_ptr = (window_datatype * )buf2d;
484 | window_init(window_buf2_buf2d, 1, buf2, LOCK_21_0_0_0, buf2d, LOCK_21_0_1_0, 64, 64);
485 | int32 index = 0;
486 | while(true)
487 | {
488 | 0x000000000000134: 0xaf 0x0f 0x0a 0x00 0x00 0x07 0xf7 ST r13, [sp, #-72]; MOV.u20 p0, #172032;
489 | 0x00000000000013c: 0x5f 0x28 0x00 0x00 0x01 0x00 0x00 0xf7 ST r12, [p0, #4]; MOV r8, sp
490 | w[i].buffer = w[i].buffers[0];
491 | 0x000000000000144: 0x54 0x0e 0x00 0x00 0x00 0xc0 0x04 0xf7 PADDA [p2], #0; ST ch2, [sp, #-88]; NOP; MOV p2, r8
492 | 0x00000000000014c: 0x08 0x00 0x00 0x00 0x00 0xc0 0x04 0xf7 PADDA [p7], #1; ST ch2, [sp, #-100]; MOV p7, p2
493 | w[i].current_bufid = 0;
494 | 0x000000000000154: 0xb8 0x03 0xc0 0x00 0x15 0xf3 0xe2 0x04 0xb0 0x00 0x00 0x3f ST r12, [sp, #-120]; MOV.u20 r13, #180224;
495 | w[i].buffer = w[i].buffers[0];
496 | 0x000000000000160: 0xaf 0x1e 0x08 0x03 NOP; ST r15, [p2]
497 | w[i].head = w[i].heads[0];
498 | 0x000000000000164: 0x80 0x00 0x00 0x00 0xff 0x94 0x07 0xf7 ST.SPIL ch0, [sp, #-10]
499 | w[i].ptr = w[i].head;
500 | 0x00000000000016c: 0x80 0x00 0x00 0x00 0xff 0x90 0x07 0xf7 ST.SPIL ch0, [sp, #-11];
501 | w[i].size = size;
502 | w[i].win_size = win_size;
503 | }
504 | }
505 | static inline void window_init(window_internal *w, int const channels, window_datatype *buffer, int const size, int const win_size)
506 | {
507 | for (int i = 0; i < channels; ++i)
508 | {
509 | w[i].buffer = buffersize+1;
510 | w[i].ptr = w[i].buffer;
511 | w[i].head = w[i].buffer;
512 | w[i].size = size;
513 | w[i].win_size = win_size;
514 | }
515 | }
516 | #endif NEW_WINDOW_H

```

The Disassembly view displays machine code and assembly code. C/C++ source code is also embedded between the lines for source code referencing. In the **Explorer** view, select the AI Engine project and right-click and select **Open Disassembly View**. It will bring up the following window where you will be able to select the core for which you want to see disassembled code.

Figure 64: Select Core



Note: In the Disassembly view, NOP instructions are skipped by default. To step the debugger through the NOP instructions, click the **Instruction Stepping Mode** icon, circled in the following image.

Figure 65: Step Through NOP Instructions

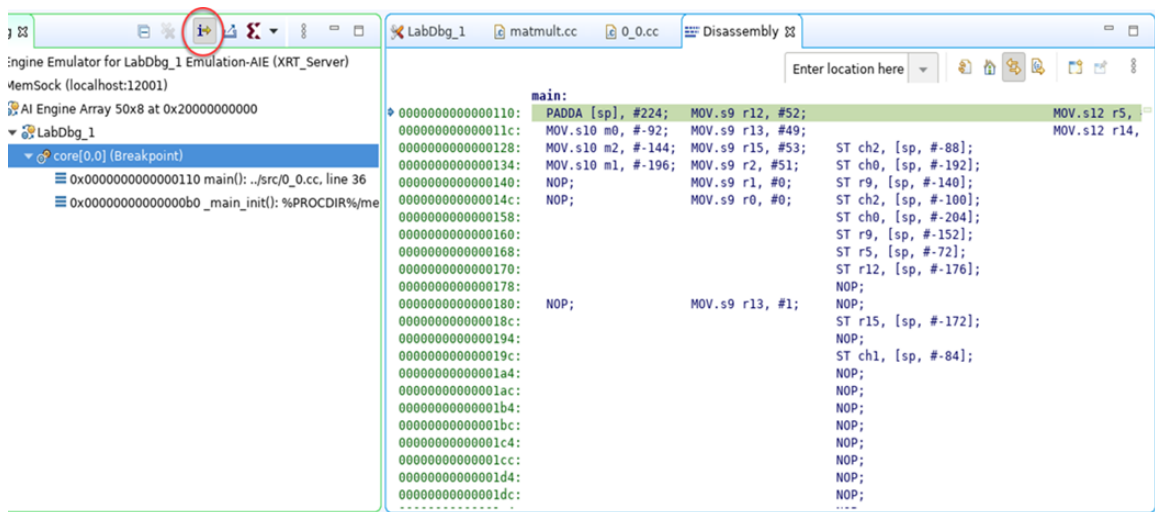
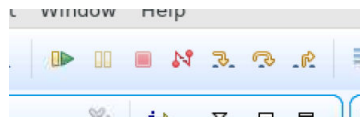


Figure 66: Debugging Controls



The debug control commands let you step into/over/return from the source code. Other commands let you resume, stop, terminate, or disconnect the debugger from the Vitis IDE.

The Vitis IDE supports multi-kernel debug and multi-domain (PS and AI Engine) debug. Depending on the application, there can be hundreds of kernels being debugged. Having granular control of each core as well as all cores within one domain is important. Select one core from the Vitis IDE and click the **Resume** icon to resume debug execution for the core only. Select one level above all cores within the AI Engine domain, and click the **Resume** icon to resume all AI Engine core executions. Resuming execution for all the kernels in the graph is dependent on a variety of conditions such as the availability of data to each of the kernels to avoid stalls, or the setup of individual kernel breakpoints. You must also ensure that the kernels not being debugged are free to run.

Figure 67: Selection to Resume Multiple AI Engine Debug

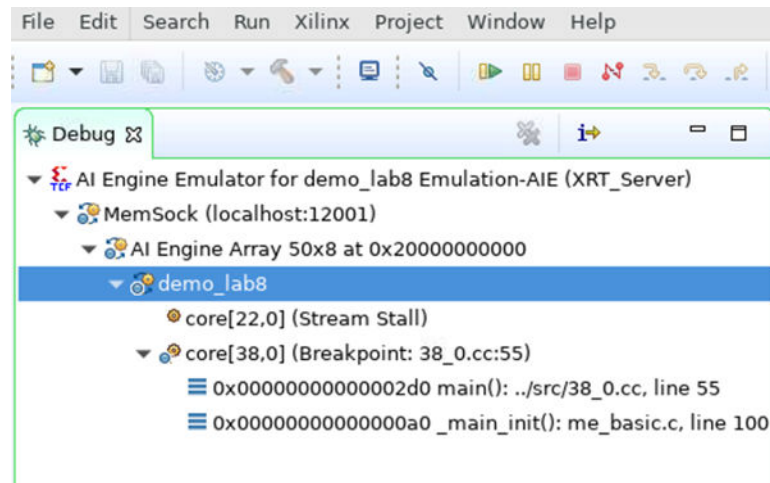
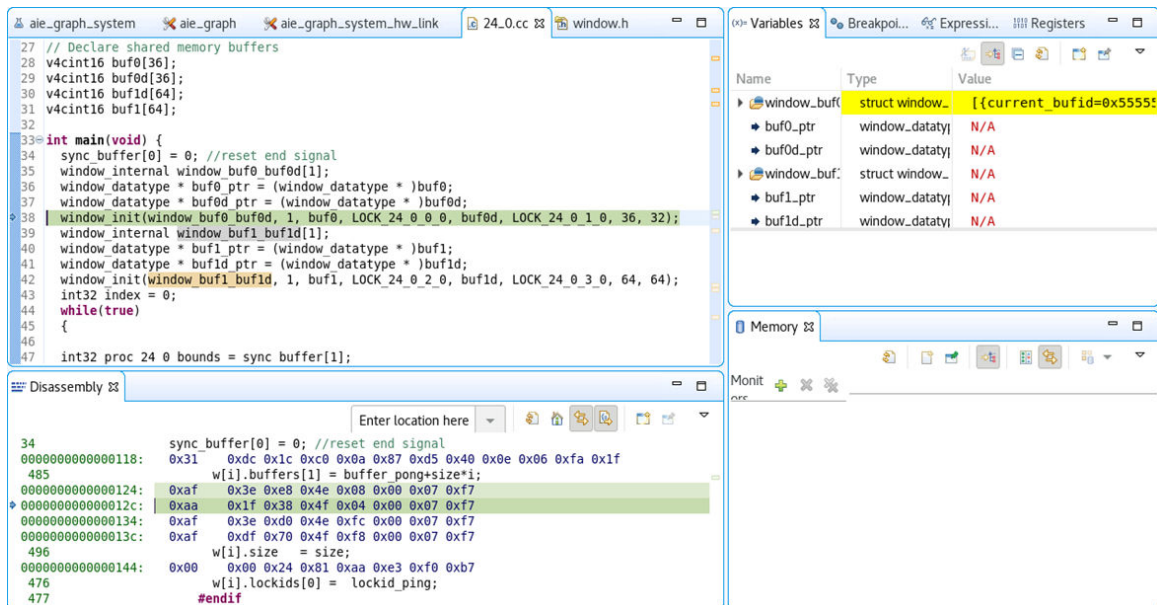


Figure 68: Debug Perspective

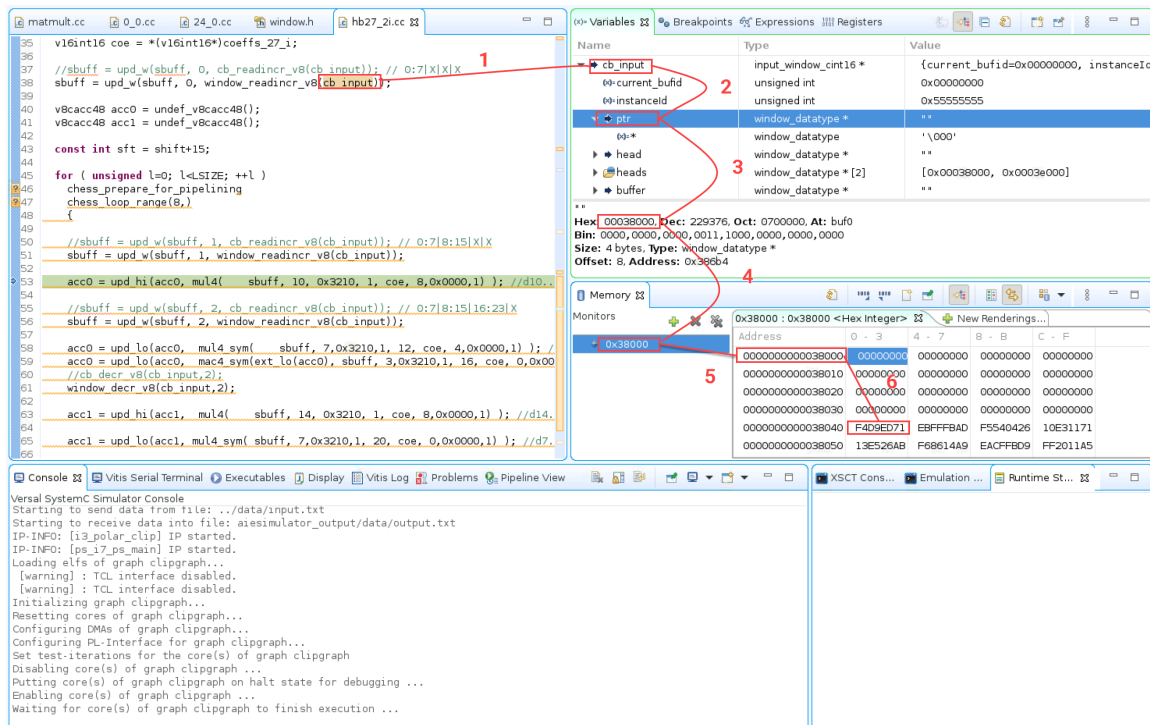


Note: The `main()` function is created automatically by the AI Engine compiler for each core that wraps around the kernel implementation. The Debug view displays the complete source code including the inserted portion.

Viewing Data from Window Interfaces

As described in [Chapter 4: Window and Streaming Data API](#) the data flows into and out of an AI Engine kernel through access windows, or a stream interface. During debug you might want to see the value of these data access windows as data is passing through the kernels. In the following figure you can trace the series of objects that you must work through to examine the contents of the variables.

Figure 69: Data Access Chain



1. In the code example shown at Step 1 in the figure, you can see that the variable `cb_input` is a pointer of type `input_window_cint16`. This shows the declaration of an input window carrying complex integers where the real part is 16-bits wide, and the imaginary part is 16 bits.
2. Move to the `cb_input` variable in the Variables view. This is the pointer representation of the data access window that holds the input data for the kernel. However, the kernel functions merely operate on pointers to the window data structures passed to them as arguments. The input window data structure is defined by a pointer called `ptr`. This `ptr` is defined by the input data window API as described in [Window Operations for Kernels](#).
3. Examine the contents of the variable `ptr`. It is an address with value `0x00038000`.

4. In the Memory window, click on the + sign to input the address 0x38000 as shown in Step 4.
5. The Memory window displays the content at address of 0x38000.
6. This is the data contained in the data access window defined by the `cb_input` variable. The previous example has a 64-byte margin size, so the actual data starts from 0x38000 plus 0x40 = 0x38040. You can examine the data contents, display it in a specific data format, or copy it to the clipboard and export it to a separate file as described in [Using the Debug Environment](#).

Notice the HEX values, such as F4D9ED71, presented in the Memory window as shown in the previous figure represent a `c_int16` type complex number with a 16-bit signed integer real number and a 16-bit signed imaginary number with the following values read from an input file:

```
-4751 -2855
-1107 -5121
1062 -2732
4465 4323
9899 5093
5289 -2426
-1063 -5425
...
```



TIP: If the connection to a data access window has a margin specified, the margin data is included in the memory window. The total memory allocated equals the window size plus the margin size.

Viewing Data from Stream Interfaces

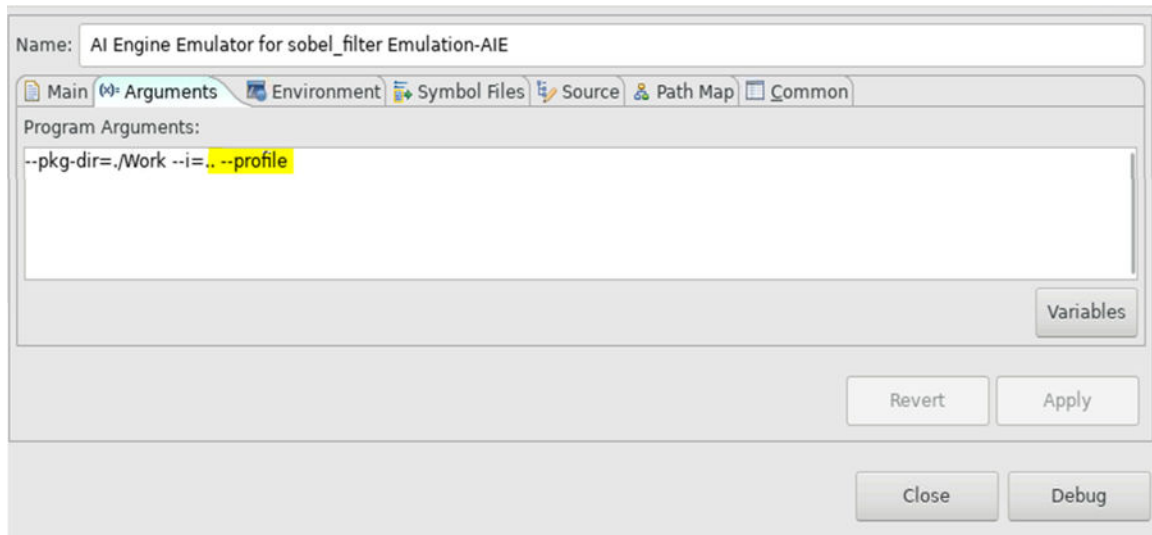
Data flows into and out of an AI Engine kernel through access windows, or a stream interface. During debug you might want to see the value of these data access windows as data is passing through the kernels. In the case of data windows, the Vitis IDE debug environment provides methods to view and access the data as described in [Viewing Data from Window Interfaces](#). In the case of stream interface connections, it is recommended to add `printf()` statements to your code to let you examine the data passing through the kernel.



IMPORTANT! Adding `printf()` statements to the code suppresses compiler optimizations, and results in a larger kernel executable program that might not fit into the available memory of the AI Engine processor.

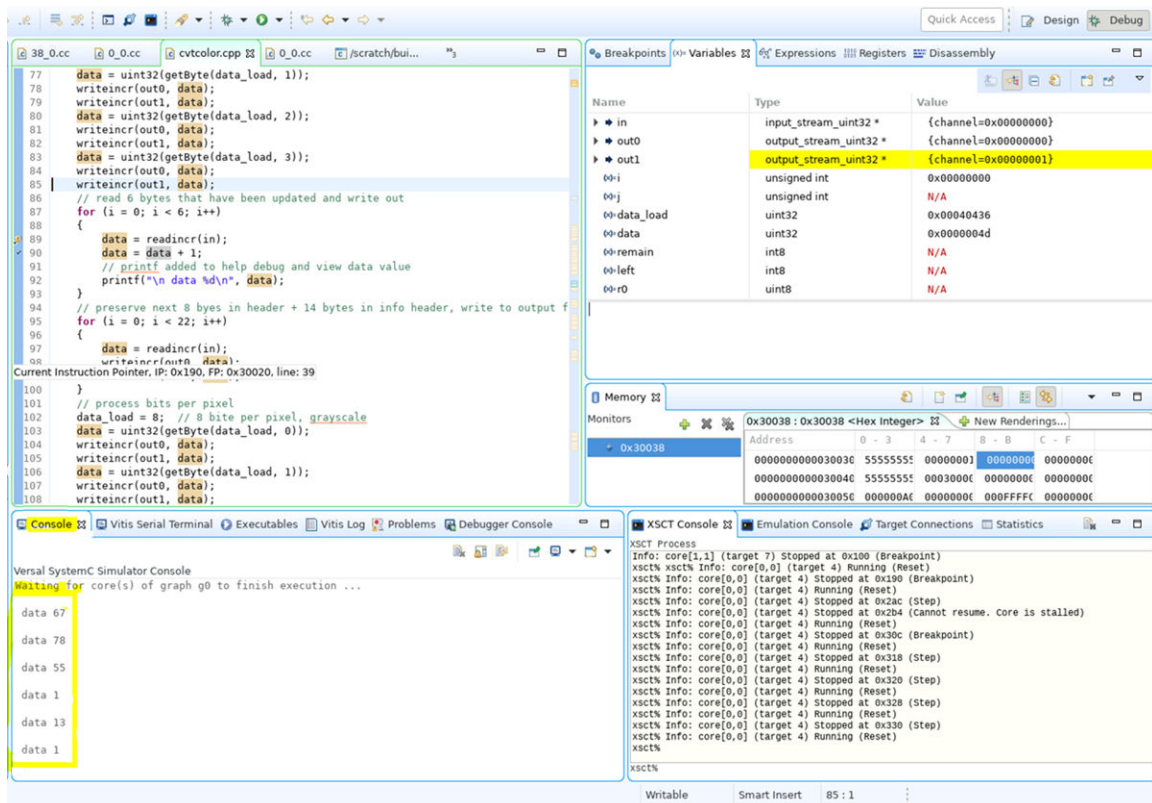
When adding the `printf()` statement in your kernel code you must also add the `--profile` option in the Run Configurations or Debug Configurations dialog box in the Vitis IDE. Add `--profile` to the Arguments tab of the Debug Configuration, along with whatever other options are already specified, as shown in the following figure.

Figure 70: Debug Configuration Arguments



Adding the `printf()` statements to your source code, results in the output of streaming data as it is processed by the kernel. The following figure shows an example of such output in the console window. This provides visibility to capture and debug the dataflow through streaming interfaces.

Figure 71: Printing Data from Stream Interfaces



Pipeline View for Single Kernel Debug

The AI Engine Pipeline view in the Vitis IDE allows you to correlate instructions executed in a specific clock cycle with the labels in the Disassembly view. The underlying AI Engine pipeline is exposed in debug mode using the pipeline view. The Vitis IDE only supports pipeline view for graphs containing single kernels.

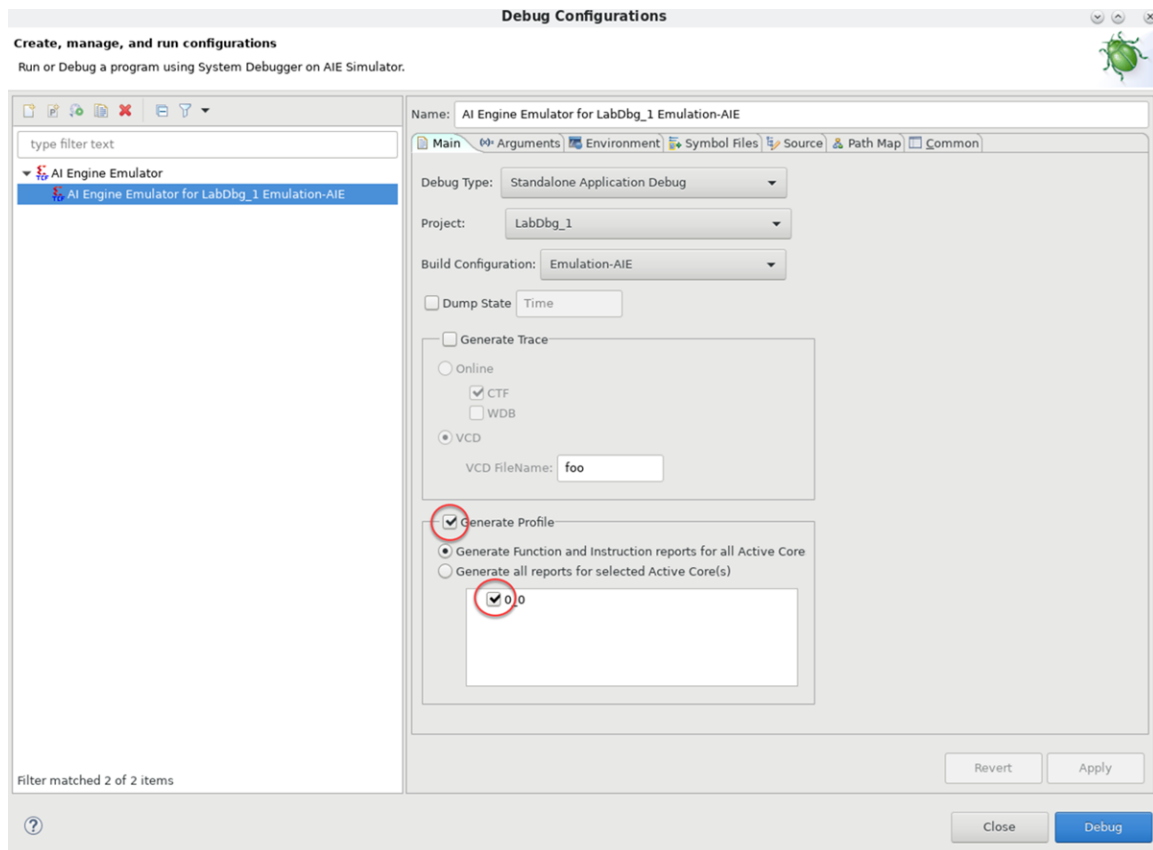
To enable the Pipeline view on graphs with a single kernel, select **Generate Profile** from the project debug configuration after the project has been built successfully.



IMPORTANT!

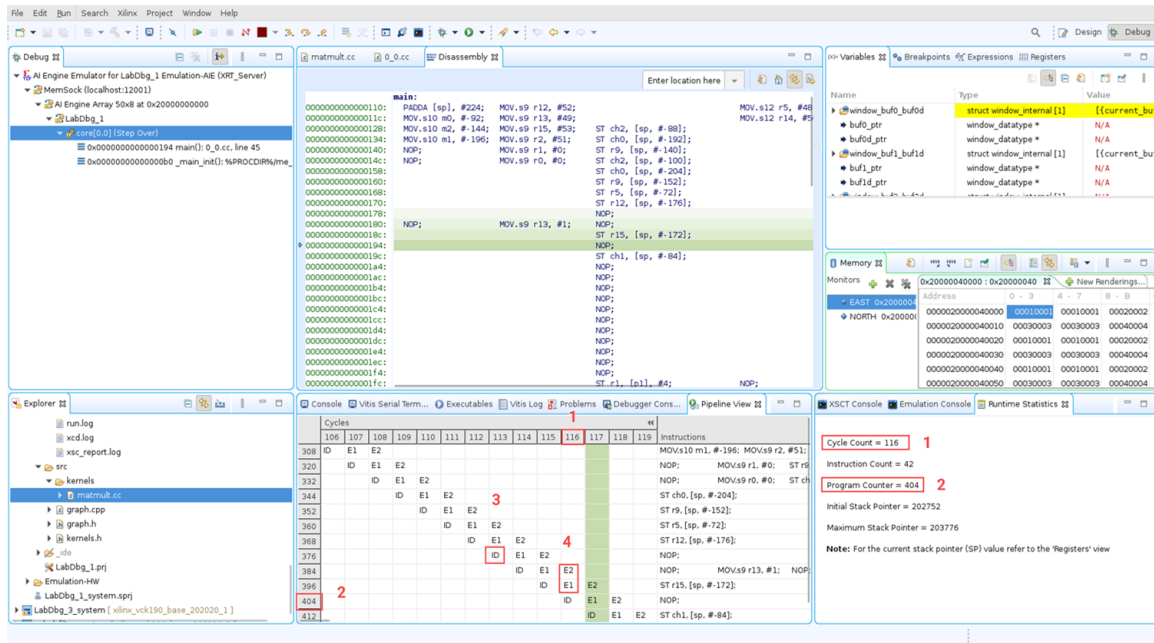
- If your graph has multiple tiles this view does not appear; however the case with multiple kernels in a single tile is supported.
- The pipeline view support is available with the AI Engine simulator only.

Figure 72: Debug Configuration Dialog Box



Click on **Debug** to start debugging the application. Note the Pipeline view shows up automatically in the Debugger Console window.

Figure 73: Pipeline View



Run-time statistics of the kernel in the pipeline view are highlighted in the previous figure.

1. AI Engine kernel cycle count
2. Program counter
3. ID = Instruction decode
4. E1-E7 are the AI Engine execution stages. Almost all operations in the scalar unit are scheduled in E1 stage of the pipeline besides non-linear operations. The vector unit scheduling spans from the ID stage to the E6 stage. Address Generation Units (AGUs) span over two pipeline stages. The address is ready in the E2 stage of the pipeline. For load units, the data will be available in the AI Engine from the memory module in the E7 stage. For the store unit, the data will be sent out from the AI Engine to the memory module in the E5 or E6 stage of the pipeline depending on the type of instruction.

Mapper/Router Methodology

Design Convergence

This section describes the process of handling a failure in the AI Engine compiler during the mapper (place) or router steps.

Mapping Solution Not Found

Mapper failures typically have two modes. Either the failure happens during a pre-check phase or it might happen during the actual mapping phase.

Pre-check failures have explicit error messages which point to exact reason for failure, as shown in the following example.

```
ERROR: [aiecompiler 47-772] Inst g.kernel_a is in conflicting pblocks:(0,0)
(5,5) and (20,0) (25,5).
```

You can trace such errors to the design element or the constraints.

Mapping phase failures typically have an error message that looks like the following.

```
ERROR: [aiecompiler 47-51] AIE Mapper failed to find a legal solution.
Please try to relax constraints and/or try alternate strategies like
disableFloorplanning.
```

In this case, use the following steps to either narrow down the cause of the failure if it is design-related or help the tool to find a solution if the failure is tool-related.

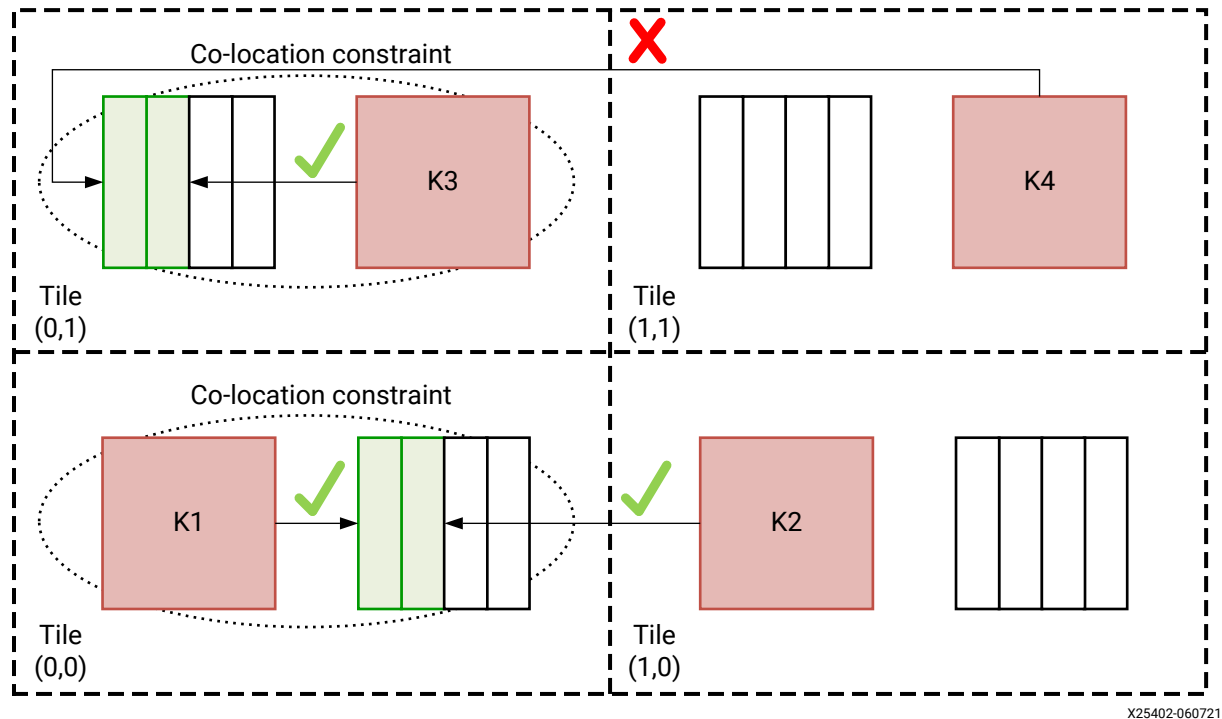
Checking User-Defined Constraints

Incorrectly defined user constraints can cause mapper failure. Some of these will be caught by the mapper pre-checker. However, not all of them can be caught in the pre-check phase. In such cases, check for the following conditions.

1. If you have a large number of absolute location or co-location constraints in the graph, check that these constraints do not give conflicting directives to the mapper. This might occur because of the checkerboard nature of the AI Engine array as shown in the following diagram.

In this diagram, the kernels in red have absolute location constraints while the window buffer (green) between them has a co-location constraint with the first kernel. This will result in mapper failure.

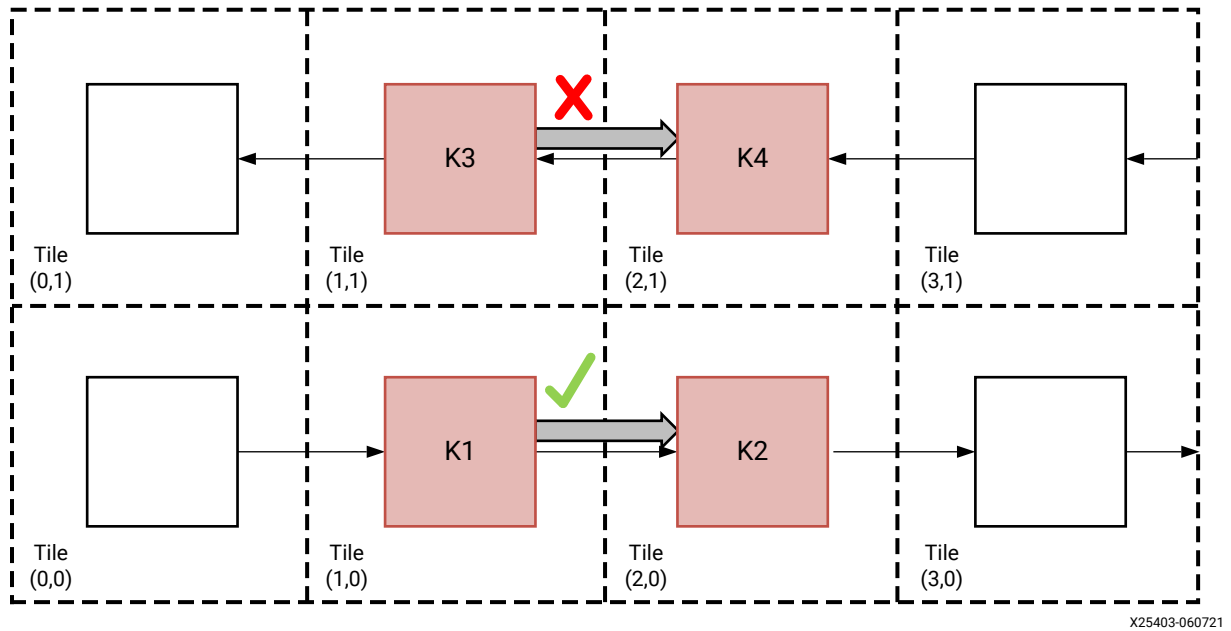
Figure 74: **Conflicting Absolute Location/Co-location Constraints**



X25402-060721

- If you have absolute location constraints for kernels that are part of a cascade chain, check that these constraints are compatible with the architecture. In AI Engine architecture, the direction of the cascade changes in each row. If you have absolute constraints and cascades as shown in the following diagram, it will cause mapper failure.

Figure 75: Conflicting Cascade Direction



3. In some cases the size of the buffers being constrained to a particular tile might exceed the memory capacity of the tile (32 KB in AI Engine architecture). This will result in mapper failure.
4. Each tile in the AI Engine architecture has two input and two output DMA channels. If you have constrained buffers in such a way that a particular tile needs more than this number of DMA channels, it will result in mapper failure.

Reducing Window Buffer Sizes for Very High Memory Density Designs

One of the main considerations when determining the window sizes for a design is that the number of cycles required for data loading is balanced with the number of compute cycles required by the kernel. This helps to pipeline the ping and pong buffer data loading with the kernel compute. For very high memory density designs, it makes sense to have smaller window sizes which can still balance the kernel compute because having larger window sizes might lead to mapper failure.

The following table shows the number of cycles required for the matrix multiplication of two matrices with 16-bit data. Example 1 and Example 2 have different matrix sizes, but both have their compute and data loading balanced. Note that only the larger of the A or B matrix size determines the data loading time whereas the time of kernel compute is determined by both sizes. This shows that Example 1 has smaller window sizes than Example 2, but the compute and data loading are balanced and can be pipelined.

Table 92: Matrix Multiplication Examples

	Matrix A Size	Matrix B Size	# of Multiplication Operations (MultOps)	#Cycles for Compute 32 ops/ cycle	#Cycles for Data Loading 32 bits/ cycle
Example 1	16x64	64x16	16384	512 (16384/32)	512 (64x16x16/32)
Example 2	16x64	64x32	32768	1024 (32768/32)	1024 (64x32x16/32)

Providing User Guidance to Mapper

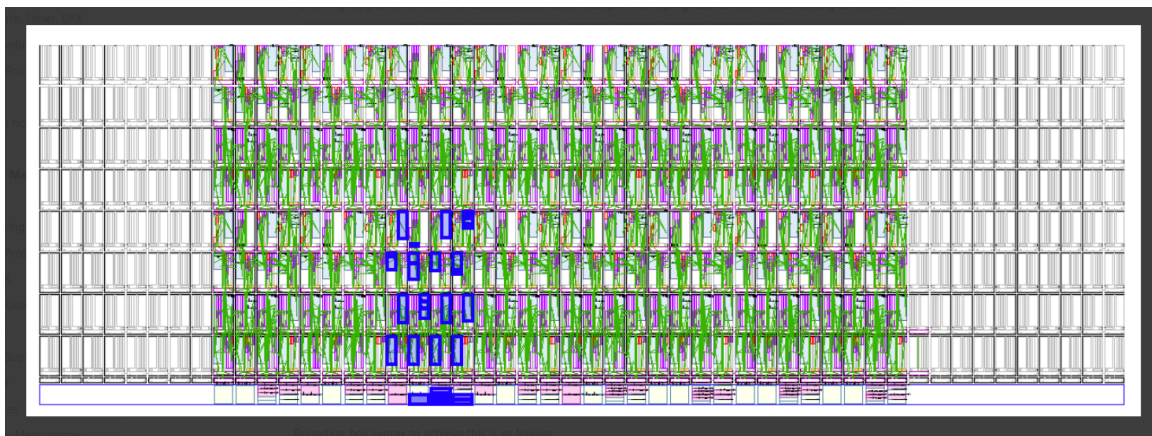
In some cases, the mapper error might be due to limitations of the tool. In such cases, it might be useful to provide guidance to the tool.

1. Turn off automatic AI Engine compiler floorplanning using the command line switch.

```
--Xmapper=disableFloorplanning
```

2. Create your own floorplanning by using either bounding box constraints in the graph or `areaGroup` constraints in the constraints file. This is particularly useful if there are multiple disjoint graphs in the design. Each of these separate graphs can be constrained to a particular region of the array. This technique not only helps design convergence but can also help improve performance by minimizing interference between different graph buffers. The following diagram shows an example using a 16 antenna transmit chain design. The highlighted kernels in blue belong to antenna 4.

Figure 76: Kernels in Antenna 4



The constraints file syntax to achieve this is as follows.

```
{
  "GlobalConstraints": {
    "areaGroup": {
      "name": "ant4_cores",
      "nodeGroup": [ "tx_chain4.*" ],
      "tileGroup": [ "(16,0):(19,3)" ]
    }
  }
}
```

The bounding box syntax to achieve this is as follows.

```
location<graph>(tx_chain4) = bounding_box(16,0,19,3);
```

3. Add co-location or absolute location constraints if necessary. If this guidance does not get the design to converge, you can try adding co-location or absolute location constraints. Co-location constraints can be added between kernels and buffers or system memory that you expect to be mapped to the same tile, as shown in the following example.

```
location<buffer>(kernel_1.out[0]) = location<kernel>(kernel_1);
location<stack>(kernel_1) = location<kernel>(kernel_1);
```

Absolute location constraints can also be added to certain key kernels or buffers to act as anchors and guide the mapper's placement of other components, as shown in the following example.

```
location<kernel>(kernel_1) = tile(20, 0);
location<buffer>(kernel_1.in[0]) =
    { address(19, 0, 0x0),
      address(19, 0, 0x2000) }; // double buffer needs two locations
```

Routing Solution Not Found

In this section you can see how to check router congestion to see whether the mapper has put the router into an impossible situation and how to fix that. Also you can see if packet switching disable is affecting the congestion.

Initially, check the user-defined constraints, which include the FIFO depth constraints and the area group constraints and then check the mapping results.

FIFO Depth Constraints

There are limited switch and DMA FIFOs on the device. When deciding on `fifo_depth` constraints it is important to consider the amount of FIFOs you specify for an area. This includes taking into account if the nets that have `fifo_depth` constraints also have area group constraints. In this case make sure that all `fifo_depth` constraints can be met within the specified area.

If there is a high contention for switch FIFOs, consider moving to DMA FIFOs. Without changing the `fifo_depth` you can specify the DMA FIFO type using the following constraint.

```
location<fifo>(net1) = { dma_fifo() }
```

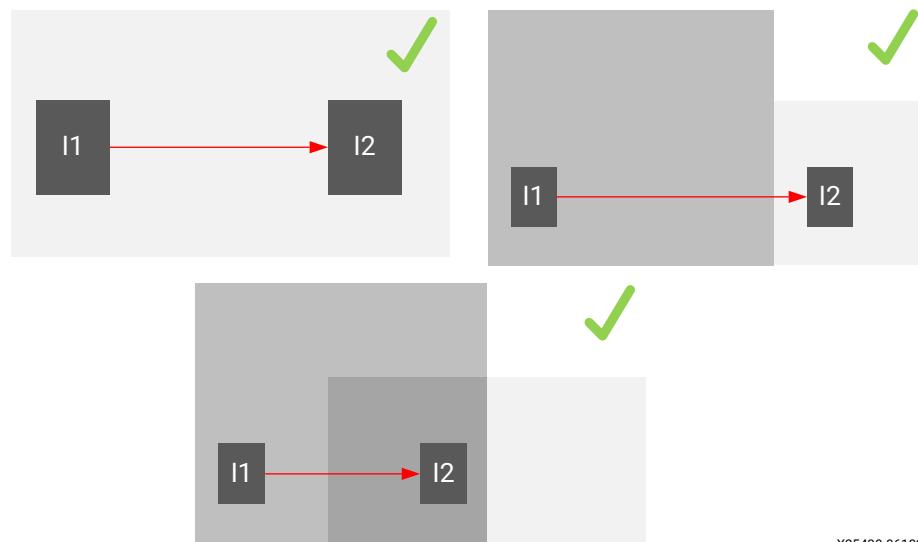
With careful consideration FIFO locations constraints can be applied, as shown in the following example.

```
location<fifo>(net2) = { dma_fifo(aie_tile, 15, 0, 0x3100, 32) };
location<fifo>(net3) = { ss_fifo(shim_tile, 16, 0, 0), dma_fifo(aie_tile,
17, 0, 0x3100, 48) }
```

Area Group Constraints

Sometimes the placement of objects is only considered when defining the area group constraints. This can leave routing without the ability to form all its connections. The following image shows a variety of area group constraints that all allow routing to form connections. In all three of these cases the routing never has to leave the defined area groups to complete its routing.

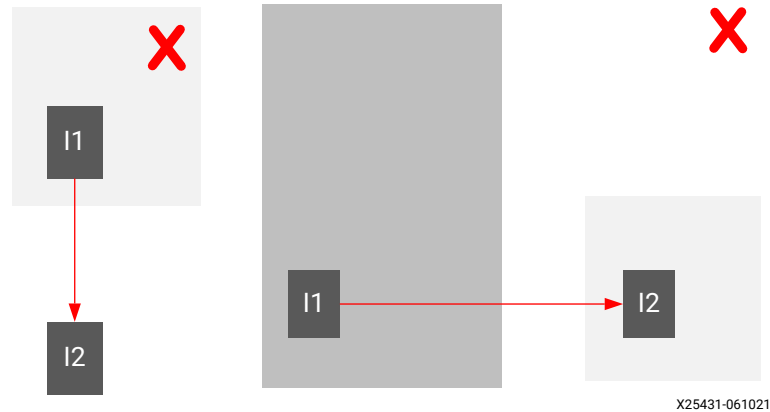
Figure 77: Routing in Defined Area Groups



X25430-061021

In contrast there are two common errors with routing and area group constraints. The left side of the following image shows a missing area group for an object. The second case is one where all objects are contained within separate area groups but the two groups are not adjacent. In this case the router has no way to complete the routing of its nets without violating an area group constraint and the router will fail to find a legal solution.

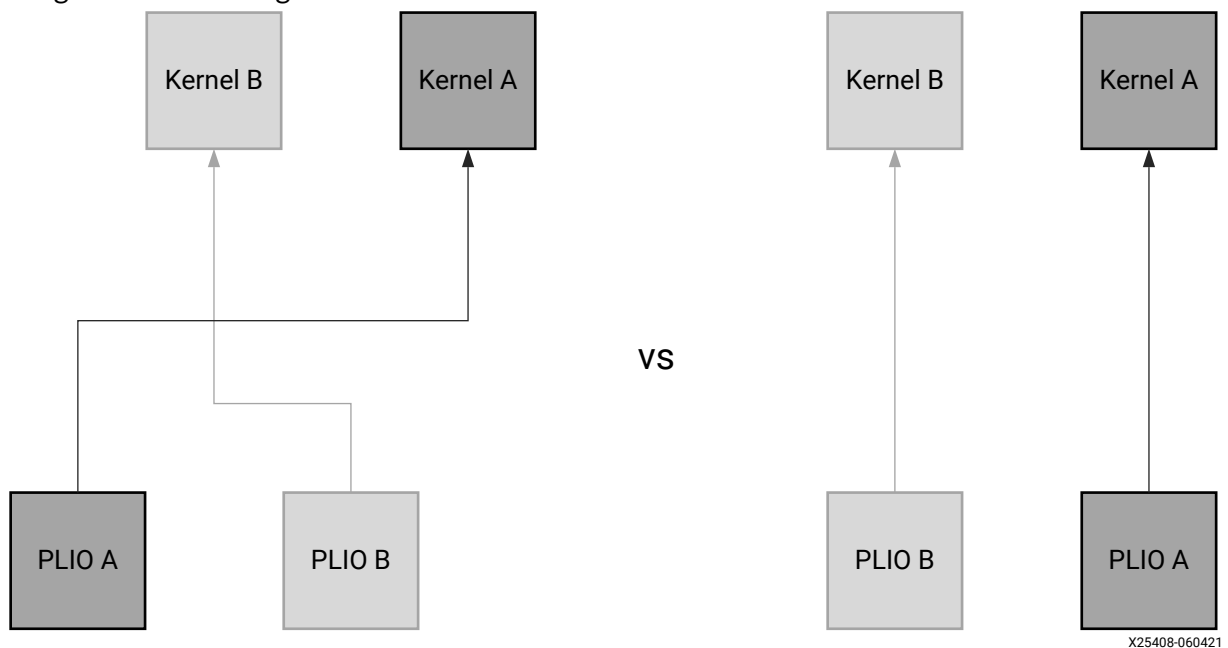
Figure 78: Routing Errors



Checking Mapping Results

Check PLIO placement to see if it is causing routing congestion. The most common congestion area for router is in the interface tile region. There are more PLIO channels than there are Interface Channels to connect them into the AI Engine array.

1. If PLIOs are being constrained make sure adequate resources into the AI Engine array are open to handle the locked PLIOs.
2. If possible limit PLIO placement near shadow regions to decrease congestion of routing resources.
3. Check PLIO placement/constraints to verify that crossing such as shown in the following image is not occurring.



Improving Design Performance

Design throughput can be negatively impacted by memory or stream stalls and also by large skew between the graph output. These situations can be identified by viewing the simulator output in the Vitis™ analyzer tool. The following sections discuss techniques to be followed in each of these cases.

Memory Stalls

The objective of the mapper is to prevent buffer conflicts, where possible. It also has different buffer optimization levels that try to bloat or increase the size of buffers to prevent conflicts. These buffer optimization levels range from 0 (default) to 9. They are invoked with the `Xmapper` option, `--Xmapper=BufferOptLevel<level num>`. At the highest buffer optimization level (9), no two buffers can be placed in the same bank. However it is important to know that at the higher buffer optimization levels it might become impossible for the mapper to find a solution and it will error out. So the first option if you see a large number of memory stalls is to cycle through the `BufferOptLevel` options to see if fewer memory stalls are seen at higher `bufferOptLevels`.

Another possibility is that you can explicitly inform the mapper not to place two buffers in the same bank. If your simulation analysis indicates that a significant throughput degradation is caused by memory stall resulting from a bank conflict between buffer `kernel_0.in[0]` and `kernel_1.out[0]`, you can provide a directive to the mapper to not place these buffers in the same bank as follows.

```
not_equal(location<buffer>(kernel_0.in[0]),
location<buffer>(kernel_1.out[0]));
```

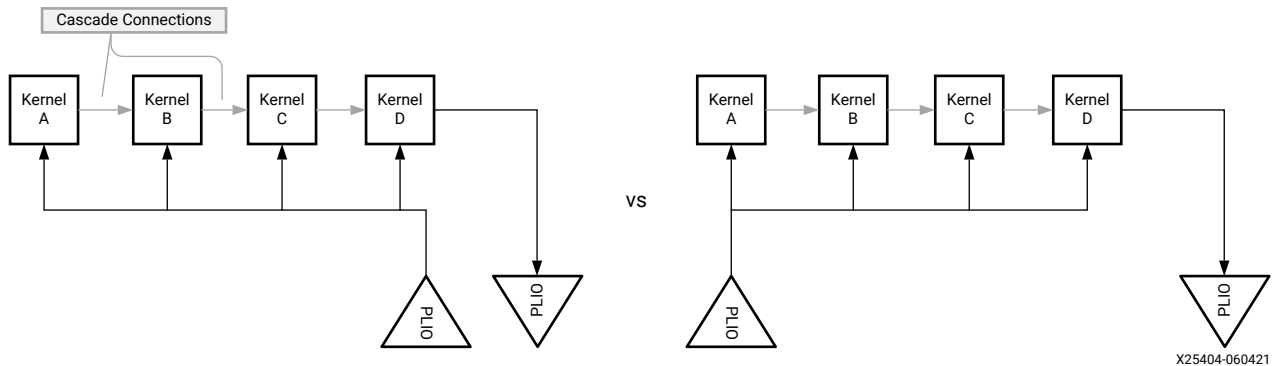
If DMA FIFOs are used in the design and they are placed in the same bank as other buffers then the `Xrouter` option `DMAFIFOsInFreeBankOnly` can force the router to place these FIFOs in free banks. This eliminates memory conflicts with the DMA FIFOs. If it is not possible to reserve an entire free bank for the DMA FIFO then location constraints can be used in coordination with outside knowledge of memory buffers. In this case it is important to have knowledge of which buffers might cause stalls when conflicting with DMA FIFOs. The constraints can look like the following.

```
location<fifo>(net2) = { dma_fifo(aie_tile, 15, 0, 0x3100, 32) };
```

Stream Stalls

The driver of a cascade chain should be placed closer to the head of the cascade chain, if possible. This reduces routing latency for these long latency paths and can reduce the need for expensively large switch/DMA FIFOs.

Figure 79: Cascade Chain



1. Make sure `fifo_depth` constraints have been specified for nets that require buffering.
2. Minimize streams and favor windows.

If `fifo_depth` has been specified you can check the log for the FIFO report section in `AIECompiler.log`.

Large Skew Between Identical Graph Outputs

If a design has multiple instantiations of the same graph, consider using the stamp and repeat flow in the mapper. In this flow, you provide input to the mapper that all the graphs should be mapped in an exactly identical manner in the area group region that you have provided for each graph. This not only simplifies the problem for the mapper, it also significantly reduces skew between the outputs of different graphs. This is particularly important for wireless communications designs with multiple antennas. Steps to use the stamp and repeat flow are as follows.

1. Define an area group for each graph. For example in the following add constraints in the `aiecost` file.

```
"GlobalConstraints": {
  "areaGroup": {
    "name": "ant0_cores",
    "nodeGroup": ["tx_chain0.*"],
    "tileGroup": ["(0,0):(3,3)"]
  },
  "areaGroup": {
    "name": "ant1_cores",
    "nodeGroup": ["tx_chain1.*"],
    "tileGroup": ["(4,0):(7,3)"]
  }
}
```

or add constraints in the graph:

```
location<graph>(tx_chain0) = bounding_box(0,0,3,3);
location<graph>(tx_chain1) = bounding_box(4,0,7,3);
```

2. Define stamp and repeat constraint in the `aiecost` file.

```
{
  "GlobalConstraints": {
    "isomorphicGraphGroup": {
      "name": "isoGroup",
      "referenceGraph": "tx_chain0",
      "stampedGraphs": ["tx_chain1"]
    }
  }
}
```

Note that the graph that is designated as the reference graph can also be given additional constraints such as co-location or absolute location constraints. These are automatically applied to other graphs with an appropriate offset.

Note: For more information see [Mapping Constraints](#).

Adaptive Data Flow Graph Specification Reference

Unless otherwise stated, all classes and their member functions belong to the `adf` name space.

Return Code

ADF APIs have defined return codes to indicate success or different kinds of failures in the `adf` namespace.

```
enum return_code
{
    ok = 0,
    user_error,
    aie_driver_error,
    xrt_error,
    internal_error,
    unsupported
};
```

The following defines the different return codes:

- **ok:** success
- **user error:** Possible invalid argument or use of the API in an unsupported way
- **aie_driver_error:** If the AI Engine driver returns errors, the graph API returns this error code.
- **xrt_error:** If XRT returns errors, the graph API returns this error code.
- **internal error:** This means something is wrong with the tool; contact Xilinx support.
- **unsupported:** Unsupported feature or unsupported scenario.

Graph Objects

graph

This is the main graph abstraction exported by the ADF tools. All user-defined graphs should be inherited from `class graph`.

Scope

All instances of those user-defined graph types that form part of a user design must be declared in global scope, but can be declared under any name space.

Member Functions

```
virtual return_code init() ;
```

This method loads and initializes a precompiled graph object onto the AI Engine array using a predetermined set of processor tiles. Currently, no relocation is supported. All existing information in the program memory, data memory, and stream switches belonging to the tiles being loaded is replaced. The loaded processors are left in a disabled state.

```
virtual return_code run();
virtual return_code run(unsigned int num_iterations);
```

This method enables the processors associated with a graph to start execution from the beginning of their respective `main` programs. Without any arguments, the graph will run forever. The API with arguments can set the number of iterations for each run differently. This is a non-blocking operation on the PS application.

```
virtual return_code end();
virtual return_code end(unsigned int cycle_timeout);
```

The `end` method is used to wait for the termination of the graph. A graph is considered to be terminated when all its active processors exit their `main` thread and disable themselves. This is a blocking operation for the PS application. This method also cleans up the state of the graph such as forcing the release of all locks and cleaning up the stream switch configurations used in the graph. The `end` method with cycle timeout terminates and cleans up the graph when the timeout expires rather than waiting for any graph related event. Attempting to `run` the graph after `end` without re-initializing it can give unpredictable results.

```
virtual return_code wait();
virtual return_code wait(unsigned int cycle_timeout);

virtual return_code resume();
```

The `wait` method is used to pause the graph execution temporarily without cleaning up its state so that it can be restarted with a `run` or `resume` method. The `wait` method without arguments is useful when waiting for a previous `run` with a fixed number of iterations to finish. This can be followed by another `run` with a new set of iterations. The `wait` method with cycle timeout pauses the graph execution when the timeout expires counted from a previous `run` or `resume` call. This should only be followed by a `resume` to let the graph to continue to execute. Attempting to `run` after a `wait` with cycle timeout call can lead to unpredictable results because the graph can be paused in an unpredictable state and the `run` can restart the processors from the beginning of their `main` programs.

```
virtual return_code update(input_port& pName, <type> value);

virtual return_code update(input_port& pName, const <type>* value, size_t
size);
```

These methods are various forms of run-time parameter update APIs that can be used to update scalar or array run-time parameter ports. The port name is a fully qualified path name such as `graph1.graph2.port` or `graph1.graph2.kernel.port`. The `<type>` can be one of `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `cint16`, `cint32`, `float`, `cfloat`. For array run-time parameter updates, a `size` argument specifies the number of elements in the array to be updated. This `size` must match the RTP array size defined in the graph, meaning that the full RTP array must be updated at one time.

```
virtual return_code read(input_port& pName, <type>& value);

virtual return_code read(input_port& pName, <type>* value, size_t size);
```

These methods are various forms of run-time parameter read APIs that can be used to read scalar or array run-time parameter ports. The port name is a fully qualified path name such as `graph1.graph2.port` or `graph1.graph2.kernel.port`. The `<type>` can be one of `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `cint16`, `cint32`, `float`, `cfloat`. For array run-time parameter reads, a `size` argument specifies the number of elements in the array to be read.

kernel

This class represents a single node of the graph. User-defined graph types contain kernel objects as member variables that wrap over some C function computation mapped to the AI Engine array.

Scope

`kernel` objects can be declared in class scope as member variables in a user-defined graph type (i.e., inside a class that inherits from `graph`).

`kernel` objects must be initialized by assignment in the graph constructor.

Member Functions

```
static kernel & create( function );
```

The static `create` method creates a kernel object from a C kernel function. It automatically determines how many input ports and output ports each kernel has and their appropriate element type. Any other arguments in a kernel function are treated as run-time parameters or lookup tables, passed to the kernel on each invocation. Run-time parameters are passed by value, while lookup tables are passed by reference each time the kernel is invoked by the compiler generated static-schedule.

```
kernel & operator()(...)
```

Takes one or more `parameter` objects as arguments. The number of `parameter` arguments must match the number of non-window formal arguments in the kernel function used to construct the `kernel`. When used in the body of a graph constructor to assign to a `kernel` member variable, the operator ensures that updated parameter arguments are passed to the kernel function on every invocation.

Member Variables

```
std::vector<port<input>>> in;
```

This variable provides access to the logical inputs of a kernel, allowing user graphs to specify connections between kernels in a graph. The *i*'th index selects the *i*'th input port (window, stream, or rtp) declared in the kernel function arguments.

```
std::vector<port<output>>> out;
```

This variable provides access to the logical outputs, allowing user graphs to specify connections between kernels in a graph. The *i*'th index selects the *i*'th output port (window or stream) declared in the kernel function arguments.

```
std::vector<port<inout>>> inout;
```

This variable provides access to the logical inout ports, allowing user graphs to specify connections between kernels in a graph. The *i*'th index selects the *i*'th inout port (rtp) declared in the kernel function arguments.

port<T>

Scope

Objects of type `port<T>` are port objects that can be declared in class scope as member variables of a user-defined graph type (i.e., member variables of a class that inherits from `graph`), or they are defined implicitly for a kernel according to its function signature. The template parameter `T` can be one of `input`, `output`, or `inout`.

Aliases

`input_port` is an alias for the type `port<input>`.

`output_port` is an alias for the type `port<output>`.

`inout_port` is an alias for the type `port<inout>`.

Purpose

Used to connect between kernels within a graph and across levels of hierarchy in customer specification containing platform, graphs, and subgraphs.

Operators

```
port<T>& negate(port<T>&)
```

When applied to a destination port within a connection, this operator inverts the Boolean semantics of the source port to which it is connected. Therefore, it has the effect of converting a 0 to 1 and 1 to 0.

```
port<T>& async(port<T>&)
```

When applied to a destination RTP port within a connection, this operator specifies an *asynchronous* update of the destination port's RTP buffer from the source port that it is connected to or from the external control application if the source is a graph port left unconnected. Therefore, the receiving kernel does not wait for the value for each invocation, rather it uses previous value stored in the corresponding buffer.

When applied to a source or destination window port, this operator specifies that the window object will not be synchronized upon kernel entry. Instead, the `window_acquire` and `window_release` APIs must be used to manage the window object synchronization explicitly within the kernel code.

```
port<T>& sync(port<T>&)
```

When applied to a source RTP port within a connection, this operator specifies a *synchronous* read of the source port's RTP buffer from the destination port that it is connected to or from the external control application if the destination is a graph port left unconnected. Therefore, the receiving kernel waits for a new value to be produced for each invocation of the producing kernel.

parameter

The `parameter` class contains two static member functions to allow you to associate globally declared variables with kernels.

Member Functions

```
static parameter & array(X)
```

Wrap around any extern declaration of an array to capture the size and type of that array variable.

```
static parameter & scalar(Y)
```

Wrap around any extern declaration of a scalar value (including user defined structs).

bypass

This class is a control flow encapsulator with data bypass. It wraps around an individual node or subgraph to create a bypass data path based on a dynamic control condition. The dynamic control is coded as a run-time parameter port `bp` (with integer value 0 or 1) that controls whether the input window (or stream) data will flow into the kernel encapsulated by the bypass (`bp=0`) or whether it will be directly bypassed into the output window (or stream) (`bp=1`).

Scope

`bypass` objects can be declared in class scope as member variables in a user-defined graph type (i.e., inside a class that inherits from `graph`).

`bypass` objects must be initialized by assignment in the graph constructor.

Member Functions

```
static bypass & create( kernel );
```

The static `create` method creates a `bypass` object around a given kernel object. The number of inputs and outputs of the bypass are inferred automatically from the corresponding ports of the kernel.

Graph Objects for Packet Processing

The following predefined object classes in the `adf` namespace are used to define the connectivity of packet streams.

```
template <int nway> class pktsplit { ... }
template <int nway> class pktmerge { ... }
```

Scope

Objects of type `pktsplit<n>` and `pktmerge<n>` can be declared as member variables in a user-defined graph type (i.e., inside a class that inherits from `graph`). The template parameter `n` must be a compile-time constant positive integer denoting the `n`-way degree of split or merge. These objects behave like ordinary nodes of the graph with input and output connections, but are only used for explicit packet routing.

Member Functions

```
static pktsplit<nway> & create();
static pktmerge<nway> & create();
```

The static `create` method for these classes work in the same way as `kernel` `create` method. The degree of split or merge is already specified in the template variable declaration.

Member Variables

```
std::vector<port<input>>> in;
```

This variable provides access to the logical inputs of the node. There is only one input for `pktsplit` nodes. For `pktmerge` nodes the `i`'th index selects the `i`'th input port.

```
std::vector<port<output>>> out;
```

This variable provides access to the logical outputs of the node. There is only one output for `pktmerge` nodes. For `pktsplit` nodes the `i`'th index selects the `i`'th output port.

Platform Objects

`platform<#in,#out>`

This templated class abstractly represents the external environment under which a top-level graph object executes. It provides a mechanism to source/sink input/output data that is consumed/produced during graph execution.

Constructor

```
simulation::platform<#in,#out> (IOAttr* in_0,..., IOAttr* out_0,...);
```

This platform constructor is provided for software simulation purposes. The template parameters `#in` and `#out` are non-negative integers specifying the number of input and output ports supported by this abstract platform object. The constructor takes as many I/O attribute specification arguments as the sum of input and output ports: first all output attributes, then all input attributes. Output platform attributes feed external data to graph inputs, and input platform attributes receive graph output data for external consumption.

An I/O attribute specification is either `FileIO`, `GMIO`, or `PLIO` objects, and is declared separately. A direct `std::string` argument can also be used to represent a `FileIO` attribute object.

Member Variables

```
std::vector<port<output>>> src;
```

This variable provides access to the output attributes of a platform in the form of an output port, allowing connections between platform outputs and graph inputs to be specified. The *i*'th index selects the *i*'th output port (window, stream, or RTP) declared in the platform constructor arguments.

```
std::vector<port<input>>> sink;
```

This variable provides access to the input attributes of a platform in the form of an input port, allowing connections between graph outputs and platform inputs for be specified. The *i*'th index selects the *i*'th input port (window, stream, or RTP) declared in the platform constructor arguments.

FileIO

This class represents the I/O port attribute specification used to connect an external file to a graph input or output port for simulation purposes.

Constructor

```
FileIO(std::string data_file);
FileIO(std::string logical_name, std::string data_file);
```

The `data_file` argument is the path name of an external file relative to the application project directory that is opened for input or output during simulation. The `logical_name` must be the same as the annotation field of the corresponding port as presented in the logical architecture interface specification.

GMIO

This class represents the I/O port attribute specification used to connect graph kernels to the external virtual platform ports representing global memory (DDR) connections.

Constructors

```
GMIO(const std::string& logical_name, int burst_length, int bandwidth);
```

This GMIO port attribute specification is used to connect AI Engine kernels to the DDR memory or connect PL blocks to the DDR memory. `logical_name` is the name of the port as presented in the interface data sheet. The `burst_length` is the length of the DDR memory burst transaction (can be 64, 128, or 256 bytes), and the `bandwidth` is the average expected throughput in MB/s.

Member Functions

```
static void* malloc(size_t size);
```

The `malloc` method allocates contiguous physical memory space and returns the corresponding virtual address. It accepts a parameter, `size`, to specify how many bytes to be allocated. If successful, the pointer to the allocated memory space is returned. `nullptr` is returned in the event of a failure.

```
static void free(void* address);
```

The `free` method frees memory space allocated by `GMIO::malloc`.

```
return_code gm2aie_nb(const void* address, size_t transaction_size);
```

The `gm2aie_nb` method initiates a DDR to AI Engine transfer. The memory space for the transaction is specified by the `address` pointer and the `transaction_size` parameter (in bytes). The transaction memory space needs to be within the total memory space allocated by the `GMIO::malloc` method. This method can only be used by platform *source* GMIO objects. It is a non-blocking function in that it does not wait for the read transaction to complete.

```
return_code aie2gm_nb(void* address, size_t transaction_size);
```

The `aie2gm_nb` method initiates an AI Engine to DDR transfer. The memory space for the transaction is specified by the `address` pointer and the `transaction_size` parameter (in bytes). The transaction memory space needs to be within the total memory space allocated by the `GMIO::malloc` method. This method can only be used by platform *sink* GMIO objects. It is a non-blocking function in that it does not wait for the write transaction to complete.

```
return_code wait();
```

The `wait` method blocks until all the previously issued transactions are complete. This method is only applicable for GMIO objects connected to AI Engine.

```
return_code gm2aie(const void* address, size_t transaction_size);
```

The `gm2aie` method is a blocking version of `gm2aie_nb`. It blocks until the AI Engine–DDR read transaction completes.

```
return_code aie2gm(void* address, size_t transaction_size);
```

The `aie2gm` method is a blocking version of `aie2gm_nb`. It blocks until the AI Engine–DDR write transaction completes.

In Linux, these GMIO member functions must use PS virtual memory addresses through `void*` pointers returned by `GMIO::malloc` in the PS program.

For bare metal, the virtual address and physical address are the same. There is no need to call `GMIO::malloc` and `GMIO::free`. But you can call it for consistency.

PLIO

This class represents the I/O port attribute specification used to connect AI Engine kernels to the external platform ports representing programmable logic.

Constructor

```
PLIO(std::string logical_name, std::string datafile);
```

The above PLIO port attribute specification is used to represent a single 32-bit input or output AXI4-Stream port at the AI Engine array interface as part of a virtual platform specification. The `logical_name` must be the same as the annotation field of the corresponding port as presented in the logical architecture interface specification. The `datafile` is an input or output file path that sources input data or receives output data for simulation purposes. This data could be captured separately during platform design and then replayed here for simulation.

```
PLIO(std::string logical_name, plio_type pliowidth, std::string datafile);
```

The above PLIO port attribute specification is used to represent a single 32-bit, 64-bit, or 128-bit input or output AXI4-Stream port at the AI Engine array interface as part of a virtual platform specification. Here the `pliowidth` can be one of `plio_32_bits` (default), `plio_64_bits`, or `plio_128_bits`.

```
PLIO(std::string logical_name, plio_type pliowidth, std::string datafile, double frequency);
```

The above PLIO port attribute specification is used to represent a single 32-bit, 64-bit, or 128-bit input or output AXI4-Stream port at the AI Engine array interface as part of a virtual platform specification. Here `pliowidth` can be one of `plio_32_bits` (default), `plio_64_bits`, or `plio_128_bits`. The frequency of the PLIO port can also be specified as part of the constructor.

```
PLIO(std::string logical_name, plio_type pliowidth, std::string datafile,
double frequency, bool binary, bool hex);
```

The above PLIO boolean port attribute specification is used to indicate if the contents in the input data file are in hex or binary formats.

The data in the data files must be organized according to the bus width of the PLIO attribute (32, 64, or 128) per line as well as the data type of the graph port it is connected to. For example, a 64-bit PLIO feeding a kernel port with data type `int32` requires file data organized as two columns. However, the same 64-bit PLIO feeding to a kernel port with data type `cint16` requires the data to be organized into four columns, each representing a 16-bit real or imaginary part of the complex data type.

Event API

The event API provides functions to configure AI Engine hardware resources for performance profiling and event tracing. In this release, a subset of performance profiling use cases are supported.

Enumeration

```
enum io_profiling_option
{
    io_total_stream_running_to_idle_cycles,
    io_stream_start_to_bytes_transferred_cycles,
    io_stream_start_difference_cycles,
    io_stream_running_event_count
};
```

The `io_profiling_option` contains the enumerated options for performance profiling using PLIO and GMIO objects. The `io_total_stream_running_to_idle_cycles` option represents the total accumulated clock cycles in between the stream running event and the stream idle event of the corresponding stream port in the interface tile. This option can be used to profile platform I/O bandwidth.

The `io_stream_start_to_bytes_transferred_cycles` option represents the clock cycles in between the first stream running event to the event that the specified number of bytes are transferred through the stream port in the interface tile. This option can be used to profile graph throughput.

The `io_stream_start_difference_cycles` option represents the clock cycles elapsed between the first stream running events of the two platform I/O objects. This option can be used to profile graph latency.

The `io_stream_running_event_count` option represents the number of stream running events. This option can be used to profile the graph throughput during a period of time for streaming applications.

Member Functions

```
static handle start_profiling(IoAttr& io, io_profiling_option option,
uint32 value = 0);
```

This function configures the performance counters in the AI Engine and starts profiling. `io` is the platform GMIO or PLIO object. `option` is one of the `io_profiling_option` enumerations described in the previous topic. If the `io_stream_start_to_bytes_transferred_cycles` option is used, the number of bytes can be specified in the `value` parameter. This function should be called after `graph::init()`. It returns a handle to be used by `read_profiling` and `stop_profiling`. If the specification is incorrect or there is insufficient hardware resources to perform the profiling, an `invalid_handle` is returned.

```
static handle start_profiling(IoAttr& io1, IoAttr& io2, io_profiling_option
option, uint32 value = 0);
```

This function configures the performance counters in the AI Engine and starts profiling. Parameters `io1` and `io2` specify the two platform I/O objects. This function should be called after `graph::init()`. It returns a handle to be used by `read_profiling` and `stop_profiling`. If the specification is incorrect or there is insufficient hardware resources to perform the profiling, an `invalid_handle` is returned.

```
static long long read_profiling(handle h);
```

This function returns the current performance counter value associated with the `handle`.

```
static void stop_profiling(handle h);
```

This function stops the performance profiling associated with the `handle` and releases the corresponding hardware resources.

Connections

The following template object constructors specify different types of connections between ports. Each of them support the appropriate overloading for input/output/inout ports. Specifying the connection object name while creating a connection is optional, but it is recommended for better debugging.

Connection Constructor Templates

```
template<int blocksize, int overlap> connect<stream , window<blocksize,
overlap> > [name](portA, portB)
```

Connects a stream port to a windowed buffer port of specified block size and overlap.

```
template<int blocksize> connect<stream , window<blocksize> > [name](portA,
portB)
```

Connects a stream port to a windowed buffer port of specified block size and zero overlap.

```
template<int blocksize> connect<window<blocksize>, stream> [name](portA,
portB)
```

Connects a windowed buffer port of specified block size to a stream port.

```
template<> connect<stream> [name](portA, portB)
```

Connects between two stream ports.

```
template<> connect<cascade> [name](portA, portB)
```

Connects between two AI Engine cascade ports.

```
template<> connect<> [name](portA, portB)
```

Connects between hierarchical ports between different levels of hierarchy.

```
template<> connect<parameter> [name](portA, portB)
```

Connects a parameter port to a kernel port.

```
template<> connect<> [name](parameter, kernel)
```

Connects a LUT parameter array object to a kernel.

Port Combinations

The port combinations used in the constructor templates are specified in the following table.

Table 93: Port Combinations

PortA	PortB	Comment
port <output>	port <output>	Connect a kernel output to a parent graph output
port <output>	port <input>	Connect a kernel output to a kernel input
port <output>	port <inout>	Connect an output of a kernel or a subgraph to an inout port of another kernel or a subgraph
port <input>	port <input>	Connect a graph input to a kernel input
port <input>	port <inout>	Connect an input of a parent graph to an inout port of a child subgraph or a kernel
port <inout> >	port <input>	Connect an inout port of a parent graph or a kernel to an input of another kernel or a subgraph
port <inout>	port <output>	Connect an inout port of a subgraph or a kernel to an output of a parent graph
parameter&	kernel&	Connect an initialized parameter variable to a kernel ensuring that the compiler allocates space for the variable in the memory around the kernel

Constraints

Constraints are user-defined properties for graph nodes that provide additional information to the compiler.

constraint<T>

This template class is used to build scalar data constraints on kernels, connections, and ports.

Scope

A constraint must appear inside a user graph constructor.

Member Functions

```
constraint<T> operator=(T)
```

This overloaded equality operator allows you to assign a value to a scalar constraint.

Constructors

The default constructor is not used. Instead the following special constructors are used with specific meaning.

```
void fabric<aiengine>(kernel&)
```

This constraint allows you to mark a kernel to be implemented on the AI Engine (default).

```
constraint<std::string>& initialization_function(kernel&)
```

This constraint allows you to set a specific initialization function for each kernel. The constraint expects a string denoting the name of the initialization function. Where multiple kernels are packed on a core, each initialization function packed on the core is called exactly once. No kernel functions are scheduled until all the initialization functions packed on a core are completed.

An initialization function cannot return a value and cannot have input/output arguments, that is, the function prototype must be as follows.

```
void init_function_name(void)
```

Note: The initialization function is called only once when the first `graph::run` API is called.

This function can be used to initialize global variables and set or clear rounding and saturation modes. It cannot use window or stream APIs to access memory or stream interfaces, but stream intrinsics (for example, `get_ss()`) can be used. See the *AI Engine Kernel Coding Best Practices Guide* ([UG1079](#)) for examples.

```
constraint<float>& runtime<ratio>(kernel&)
```

This constraint allows you to set a specific core usage fraction for a kernel. This is computed as a ratio of the number of cycles taken by one invocation of a kernel (processing one block of data) to the cycle budget. The cycle budget for an application is typically fixed according to the expected data throughput and the block size being processed.

```
constraint<std::string>& source(kernel&)
```

This constraint allows you to specify the source file containing the definition of each kernel function. A source constraint must be specified for each kernel.

```
constraint<int>& fifo_depth(connect&)= [<depth> | (depth)]
```

This constraint allows you to specify the amount of slack to be inserted on a streaming connection to allow deadlock free execution.

```
void single_buffer(port<T>&)
```

This constraint allows you to specify single buffer constraint on a window port. By default, a window port is double buffered.

```
void initial_value(async_AIE_RTP_port)
```

This constraint allows you to set the initial value for an asynchronous AI Engine input run-time parameter port. It allows the destination kernel to start asynchronously with the specified initial value. You can set both scalar and array run-time parameters using this constraint.

Example scalar: `initial_value(6).`

Example array: `initial_value({1,2,3})`

```
constraint<int> stack_size(adf::kernel& k);
```

This constraint allows you to set stack size for individual kernel.

```
constraint<int> heap_size(adf::kernel& k);
```

This constraint allows you to set heap size for individual kernel.

constraint< std::vector<T>>

This template class is used to build vector data constraints on kernels, connections, and ports.

Scope

Constraint must appear inside a user graph constructor.

Member Function

```
constraint<std::vector<T> > operator=(std::vector<T>)
```

Constraint must appear inside a user graph constructor.

Constructors

The default constructor is not used. Instead the following special constructors are used with specific meaning.

```
constraint <std::vector<std::string > >& headers (kernel&)
```

This constraint allows you to specify a set of header files for a kernel that define objects to be shared with other kernels and hence have to be included once in the corresponding `main` program. The kernel source file would instead include an `extern` declaration for that object.

Mapping Constraints

The following functions help to build various types of constraints on the physical mapping of the kernels and buffers onto the AI Engine array.

Scope

A constraint must appear inside a user graph constructor.

Kernel Location Constructors

```
location_constraint tile(int col, int row)
```

This location constructor points to a specific AI Engine tile located at specified column and row within the AI Engine array. The column and row values are zero based, where the zero'th row is counted from the bottom-most row with a compute processor and the zero'th column is counted from the left-most column. The previously used constructor `proc(col, row)` is now deprecated.

```
location_constraint location<kernel> (kernel&)
```

This constraint provides a handle to the location of a kernel so that it can be constrained to be located on a specific tile or co-located with another kernel using the following assignment operator.

Buffer Location Constructors

```
location_constraint address(int col, int row, int offset)
```

This location constructor points to a specific data memory address offset on a specific AI Engine tile. The offset address is relative to that tile starting at zero with a maximum value of 32768 (32K).

```
location_constraint bank(int col, int row, int bankid)
```

This location constructor points to a specific data memory bank on a specific AI Engine tile. The bank ID is relative to that tile and can take values 0, 1, 2, 3.

```
location_constraint offset(int offset_value)
```

This location constructor specifies data memory address offset. The offset address is between 0 and 32768 (32K) and is relative to a tile allocated by the compiler.

```
location_constraint location<buffer> (port<T>&)
```

This location constructor provides a handle to the location of a buffer attached to an input, output, or inout port of a kernel. It can be used to constrain the location of the buffer to a specific address or bank, or to be on the same tile as another kernel, or to be on the same bank as another buffer using the following assignment operator. It is an error to constrain two buffers to the same address. This constructor only applies to window kernel ports.

```
location_constraint location<stack> (kernel&)
```

This location constructor provides a handle to the location of the system memory (stack and heap) of the AI Engine where the specified kernel is mapped. This provides a mechanism to constrain the location of the system memory with respect to other buffers used by that kernel.



IMPORTANT! *The stack location offset must be in multiples of 32 bytes.*

```
location_constraint location<parameter> (parameter&)
```

This location constructor provides a handle to the location of the parameter array (for example, a lookup table) declared within a graph.

Bounding Box Constructor

```
location_constraint bounding_box(int column_min, int row_min, int
column_max, int row_max)
```

This bounding box constructor specifies a rectangular bounding box for a graph to be placed in AI Engine tiles, between columns from `column_min` to `column_max` and rows from `row_min` to `row_max`. Multiple bounding box location constraints can be used in an initializer list to specify an irregular shape bounding region.

Operator Functions

```
location_constraint& operator=(location_constraint)
```

This operator expresses the equality constraint between two location constructors. It allows various types of absolute or relative location constraints to be expressed.

The following example shows how to constrain a kernel to be placed on a specified AI Engine tile.

```
location<kernel>(k1) = tile(3,2);
```

The following template shows how to constrain the location of double buffers attached to a port that are to be placed on a specific address or a bankid. At most, two elements should be specified in the initializer list to constrain the location of the double banks. Furthermore, if these buffers are read or written by a DMA engine, then they must be on the same tile.

```
location<buffer>(port1) = { [address(c,r,o) | bank(c,r,id)] ,
[address(c,r,o) | bank(c,r,id)] };
```

The following template shows how to constrain the location of a parameter lookup table or the system memory of a kernel to be placed on a specific address or a bankid.

```
location<parameter>(param1) = [address(c,r,o) | bank(c,r,id)];
location<stack>(k1) = [address(c,r,o) | bank(c,r,id)];
```

The following example shows how to constrain two kernels to be placed relatively on the same AI Engine. This forces them to be sequenced in topological order and be able to share memory buffers without synchronization.

```
location<kernel>(k1) = location<kernel>(k2);
```

The following example shows how to constrain a buffer, stack, or parameter location to be on the same tile as that of a kernel. This ensures that the buffer, or parameter array can be accessed by the other kernel k1 without requiring a DMA.

```
location<buffer>(port1) = location<kernel>(k1);
location<stack>(k2) = location<kernel>(k1);
location<parameter>(param1) = location<kernel>(k1);
```

The following example shows how to constrain a buffer, stack, or parameter location to be on the same bank as that of another buffer, stack, or parameter. When two double buffers are co-located, this constrains both the ping buffers to be on one bank and both the pong buffers to be on another bank.

```
location<buffer>(port2) = location<buffer>(port1);
location<stack>(k1) = location<buffer>(port1);
location<parameter>(param1) = location<buffer>(port1);
```

The following example shows how to constraint a graph to be placed within a bounding box or a joint region among multiple bounding boxes.

```
location<graph>(g1) = bounding_box(1,1,2,2);
location<graph>(g2) = { bounding_box(3,3,4,4), bounding_box(5,5,6,6) };
```

The following example shows how to constrain FIFO locations using the DMA FIFO with `aie_tile/memory_tile/shim_tile`, tile column number, tile row number, memory address, and size as input parameters, and/or the stream switch FIFO with `aie_tile/memory_tile/shim_tile`, tile column number, tile row number, and FIFO identifier as input parameters.

```
location(net0) = { dma_fifo(tile_type0, col0, row0, address0, size0),
ss_fifo(tile_type1, col1, row1, fifo_id1), ... }
```

Non-Equality Function

```
void not_equal(location_constraint lhs, location_constraint rhs)
```

This function expresses $lhs \neq rhs$ for the two `location_constraint` parameters `lhs` and `rhs`. It allows relative non-collocation constraint to be specified. The `not_equal` buffer constraint only works for single buffers. This constraint should not be used with double buffers.

The following example shows how to specify two kernels, `k1` and `k2`, should not be mapped to the same AI Engine.

```
not_equal(location<kernel>(k1), location<kernel>(k2));
```


The following example shows how to specify two buffers, `port1` and `port2`, should not be mapped to the same memory bank.

```
not_equal(location<buffer>(port1), location<buffer>(port2));
```

Stamp and Repeat Constraint

The following example shows how to constraint a graph to be placed within a bounding box. In this case, the `tx_chain0` graph is the reference and its objects will be placed first and stamped to graph `tx_chain1` and `tx_chain2`. The number of rows for all identical graphs (reference plus stamp-able ones) must be the same, and must begin and end at the same parity of row, meaning if the reference graph's bounding box begins at an even row and ends at an odd row, then all of the stamped graphs must follow the same convention. This limitation occurs because of the mirrored tiles in AI Engine array. In one row the AI Engine is followed by a memory group, and in the next row the memory group is followed by an AI Engine within a tile.

```
location<graph>(tx_chain0) = bounding_box(0,0,3,3);
location<graph>(tx_chain1) = bounding_box(4,0,7,3);
location<graph>(tx_chain2) = bounding_box(0,4,3,7);

location<graph>(tx_chain1) = stamp(location<graph>(tx_chain0));
location<graph>(tx_chain2) = stamp(location<graph>(tx_chain0));
```

JSON Constraints

The constraints JSON file can contain one or more of the following sections:

- **NodeConstraints:** Constrain graph nodes, such as kernels
- **PortConstraints:** Constrain kernel ports and params
- **GlobalConstraints:** Specify global constraints, i.e., constraints that are not associated with a specific object

Node Constraints

The `NodeConstraints` section is used to constrain graph nodes. Constraints are grouped by node, such that one or more constraints can be specified per node.

Syntax

```
{
  "NodeConstraints": {
    "<node name>": {
      <constraint>,
      <constraint>,
      ...
    }
  }
}
```

```

<node name> ::= string
<constraint> ::= tile
                | shim
                | reserved_memory
                | colocated_nodes
                | not_colocated_nodes
                | colocated_reserved_memories
                | not_colocated_reserved_memories

```

Example

```

{
  "NodeConstraints": {
    "mygraph.k1": {
      "tile": {
        "column": 2,
        "row": 1
      },
      "reserved_memory": {
        "column": 2,
        "row": 1,
        "bankId": 3,
        "offset": 4128
      }
    },
    "mygraph.k2": {
      "tile": {
        "column": 2,
        "row": 2
      }
    }
  }
}

```

Node Names

Nodes must be specified by their fully qualified name, for example: <graph name>.<kernel name>.

In the following example, the graph name is `myGraph` and the kernel name is `k1`. The fully specified node name is `myGraph.k1`.

```

class graph : public adf::graph {
private:
    adf::kernel k1;
public:
    my_graph() {
        k1 = kernel::create(kernel1);
        source(k1) = "src/kernels/kernel1.cc";
    }
};
graph myGraph;

```

Whenever this kernel is referenced in the constraints JSON file it must be named `myGraph.k1`, as shown in the various examples throughout this document.

Tile Constraint

This constrains a kernel to a specific tile located at a specified column and row within the array. The column and row values are zero based, where the zeroth row is counted from the bottom-most compute processor and the zero-th column is counted from the left-most column.

Syntax

```
"tile": {
  "column": integer,
  "row": integer
}
```

Example

```
{
  "NodeConstraints": {
    "mygraph.k1": {
      "tile": {
        "column": 2,
        "row": 1
      }
    }
  }
}
```

Shim Constraint

This constrains a node (PLIO or GMIO) to a specific AI Engine array interface, which is specified by column and channel. The column and channel are zero based. The channel is optional, and if omitted, the compiler selects the optimal channel.

Note: PLIOs cannot be placed in every column. The availability of columns is device dependent. For example, columns 0-5 cannot be used for PLIO for the xcvc1902-vsva2197-2MP-e-S device. Refer to the relevant device data sheet for more information.

Syntax

```
"shim": {
  "column": integer,
  "channel": integer (optional)
}
```

Example

```
{
  "NodeConstraints": {
    "plioOut1": {
      "shim": {
        "column": 0,
        "channel": 1
      }
    }
  },
}
```

```

    "plioOut2": {
      "shim": {
        "column": 1
      }
    }
  }
}

```

Reserved Memory Constraint

This constrains the location of system memory (stack and heap) for a kernel to a specific address on a specific tile. The address can be specified in one of two different ways:

- Column, row, bankID and offset, where the tile is specified by column, row, bankID and the offset address is relative to the bankID, starting at zero with a maximum value of 8192 bytes of a bank.
- Column, row, and bankId, where the bank ID is relative to the tile and can take values 0, 1, 2, 3.

Syntax

```

"reserved_memory": <bank_address>
<bank_address> ::= {
  "column": integer,
  "row": integer,
  "bankId": integer,
  "offset": integer
}
<bank_address> ::= {
  "column": integer,
  "row": integer,
  "bankId": integer
}

```

Example

```

{
  "NodeConstraints": {
    "mygraph.k1": {
      "reserved_memory": {
        "column": 2,
        "row": 1,
        "bankId": 3,
        "offset": 4128
      }
    },
    "mygraph.k2": {
      "reserved_memory": {
        "column": 1,
        "row": 1,
        "bankId": 3
      }
    }
  }
}

```

Colocated Nodes Constraint

The colocated nodes constraint requires two or more kernels to be on the same tile and forces sequencing of the kernels in a topological order. It also allows them to share memory buffers without synchronization.

Syntax

```
"colocated_nodes": [<node list>]
<node list> ::= <node name>[,<node name>...]
<node name> ::= string
```

Example

```
{
  "NodeConstraints": {
    "mygraph.k2": {
      "colocated_nodes": ["mygraph.k1"]
    }
  }
}
```

Not Colocated Nodes Constraint

This constrains two or more kernels to not be on the same tile.

Syntax

```
"not_colocated_nodes": [<node list>]
<node list> ::= <node name>[,<node name>...]
<node name> ::= string
```

Example

```
{
  "NodeConstraints": {
    "mygraph.k2": {
      "not_colocated_nodes": ["mygraph.k1"]
    }
  }
}
```

Colocated Reserved Memories Constraint

This constrains a kernel location to be on the same tile as that of one or more stacks. This ensures that the stacks can be accessed by the kernel without requiring a DMA.

Syntax

```
"colocated_reserved_memories": [<node list>]
<node list> ::= <node name>[,<node name>...]
<node name> ::= string
```

Example

```
{
  "NodeConstraints": {
    "mygraph.k2": {
      "colocated_reserved_memories": ["mygraph.k1"]
    }
  }
}
```

Not Colocated Reserved Memories Constraint

This constrains a kernel location so that it will not be on the same tile as one or more stacks.

Syntax

```
"not_colocated_reserved_memories": [<node list>]
<node list> ::= <node name>[,<node name>...]
<node name> ::= string
```

Example

```
{
  "NodeConstraints": {
    "mygraph.k2": {
      "not_colocated_reserved_memories": ["mygraph.k1"]
    }
  }
}
```

Port Constraints

Port constraints are specified in the PortConstraints section. Constraints are grouped by port, such that one or more constraints can be specified per port.

Syntax

```
{
  "PortConstraints": {
    "<port name>": {
      <constraint>[,
      <constraint>...]
    }
  }
}
<port name> ::= string
<constraint> ::= buffers
                | colocated_nodes
                | not_colocated_nodes
                | colocated_ports
                | not_colocated_ports
                | exclusive_colocated_ports
                | colocated_reserved_memories
                | not_colocated_reserved_memories
```

Example

```
{
  "PortConstraints": {
    "mygraph.k1.in[0]": {
      "colocated_nodes": [ "mygraph.k1" ]
    },
    "mygraph.k2.in[0]": {
      "colocated_nodes": [ "mygraph.k2" ]
    },
    "mygraph.p1": {
      "buffers": [{
        "column": 2,
        "row": 1,
        "bankId": 2
      }]
    }
  }
}
```

Port Names

Ports must be specified by their fully qualified name: `<graph name>.<kernel name>.<port name>`. In the following example, the graph name is `myGraph`, the kernel name is `k1`, and the kernel has two ports named `in[0]` and `out[0]` (as specified in `kernel1.cc`). The fully specified port names are then `myGraph.k1.in[0]` and `myGraph.k1.out[0]`.

```
class graph : public adf::graph {
private:
  adf::kernel k1;
public:
  my_graph() {
    k1 = kernel::create(kernel1);
    source(k1) = "src/kernels/kernel1.cc";
  }
};
graph myGraph;
```

Anytime either of these ports are referenced in the constraints JSON file, they must be named `myGraph.k1.in[0]` and `myGraph.k1.out[0]`, as shown in the various examples throughout this document.

Buffers Constraint

This constrains a data buffer to a specific address on a specific tile. The data buffer can be attached to an input, output, or inout port of a kernel or param (e.g., a lookup table). The address can be specified in one of three different ways:

- Column, row, and offset, where the tile is specified by column and row and the offset address is relative to the tile, starting at zero with a maximum value of 32768 (32k).
- Column, row, and bankId, where the bank ID is relative to the tile and can take values 0, 1, 2, 3.

- Offset, that can be between zero and 32768 (32k) and is relative to the tile allocated by the compiler.

Note: One or two buffers can be constrained for a port.

Syntax

```
"buffers": [<address>, <(optional) address>]
<address> ::= <offset_address> | <bank_address> | <offset_address>
<tile_address> ::= {
  "column": integer,
  "row": integer,
  "offset": integer
}
<bank_address> ::= {
  "column": integer,
  "row": integer,
  "bankId": integer
}
<offset_address> ::= {
  "offset": integer
}
```

Example

```
{
  "PortConstraints": {
    "mygraph.k2.out[0]": {
      "buffers": [{
        "column": 2,
        "row": 2,
        "offset": 5632
      }, {
        "column": 2,
        "row": 2,
        "offset": 4608
      }]
    },
    "mygraph.k1.out[0]": {
      "buffers": [{
        "column": 2,
        "row": 3,
        "bankId": 2
      }, {
        "column": 2,
        "row": 3,
        "bankId": 3
      }]
    },
    "mygraph.p1": {
      "buffers": [{
        "offset": 512
      }]
    }
  }
}
```


Colocated Nodes Constraint

This constrains a port (i.e., the port buffer) location to be on the same tile as that of one or more kernels. This ensures that the data buffer can be accessed by the other kernels without requiring a DMA.

Syntax

```
"colocated_nodes": [<node list>]
<node list> ::= <node name>[,<node name>...]
<node name> ::= string
```

Example

```
{
  "PortConstraints": {
    "mygraph.k1.in[0]": {
      "colocated_nodes": ["mygraph.k1"]
    },
    "mygraph.k2.in[0]": {
      "colocated_nodes": ["mygraph.k2"]
    }
  }
}
```

Not Colocated Nodes Constraint

This constrains a port (i.e., the port buffer) location to not be on the same tile as that of one or more kernels.

Syntax

```
"not_colocated_nodes": [<node list>]
<node list> ::= <node name>[,<node name>...]
<node name> ::= string
```

Example

```
{
  "PortConstraints": {
    "mygraph.k2.in[0]": {
      "not_colocated_nodes": ["mygraph.k1"]
    }
  }
}
```

Colocated Ports Constraint

This constrains a ports buffer location to be on the same bank as that of one or more other port buffers. When two double buffers are co-located, this constraints both of the ping buffers to be on one bank and both of the pong buffers to be on another bank.

Syntax

```
"colocated_ports": [<port list>]
<port list> ::= <port name>[, <port name>...]
<port name> ::= string
```

Example

```
{
  "PortConstraints": {
    "mygraph.k2.in[0]": {
      "colocated_ports": ["mygraph.k2.out[0]"]
    }
  }
}
```

Not Colocated Ports Constraint

This constrains a port buffer location to not be on the same bank as that of one or more other port buffers.

Syntax

```
"not_colocated_ports": [<port list>]
<port list> ::= <port name>[, <port name>...]
<port name> ::= string
```

Example

```
{
  "PortConstraints": {
    "mygraph.k2.in[0]": {
      "not_colocated_ports": ["mygraph.k2.out[0]"]
    }
  }
}
```

Exclusive Colocated Ports Constraint

This constrains a port buffer location to be exclusively on the same bank as that of one or more other port buffers, meaning that no other port buffers can be on the same bank.

Syntax

```
"exclusive_colocated_ports": [<port list>]
<port list> ::= <port name>[, <port name>...]
<port name> ::= string
```

Example

```
{
  "PortConstraints": {
    "mygraph.k2.in[0]": {
      "exclusive_colocated_ports": ["mygraph.k2.out[0]"]
    }
  }
}
```

Colocated Reserved Memories Constraint

This constrains a ports buffer location to be on the same bank as that of one or more stacks.

Syntax

```
"colocated_reserved_memories": [<port list>]
<port list> ::= <port name>[, <port name>...]
<port name> ::= string
```

Example

```
{
  "PortConstraints": {
    "mygraph.k2.in[0]": {
      "colocated_reserved_memories": ["mygraph.k1"]
    }
  }
}
```

Not Colocated Reserved Memories Constraint

This constrains a ports buffer location to not be on the same bank as that of one or more stacks.

Syntax

```
"not_colocated_reserved_memories": [<port list>]
<port list> ::= <port name>[, <port name>...]
<port name> ::= string
```

Example

```
{
  "PortConstraints": {
    "mygraph.k2.in[0]": {
      "not_colocated_reserved_memories": ["mygraph.k1"]
    }
  }
}
```

FIFO Constraint

This constrains a FIFO to a specific tile located at a specified column and row within the array. The tile can be an AI Engine tile, memory tile, or interface tile. The bankId is optional; if omitted, the compiler selects an optimal bank.

The FIFO constraint can be specified in one of the following ways:

- Column, row, bankID and offset, where the tile is specified by column, row, bankID, and the offset address is relative to the bankID, starting at zero with a maximum value of 8188 32-bit words of a bank.
- Column, row, and bankId, where the bank ID is relative to the tile and can take values 0, 1, 2, or 3.

The following code block shows the syntax.

```
"PortConstraints": [<fifo list>]
<fifo list> ::= <fifo type>[, <fifo type>...]
<fifo type> ::= <dma_fifos> | <stream_fifos>
<dma_fifos> ::= <aie_tile>
<aie_tile> ::= {
  "fifo_id": string,
  "tile_type": "core",
  "column": integer,
  "row": integer,
  "size": integer,
  "offset": integer,
  "bankId": integer (optional)
}
<stream_fifos> ::= {
  "fifo_id": string,
  "tile_type": "shim",
  "column": integer,
  "row": integer,
  "channel": integer
}
```

The following code shows an example.

```
{
  "PortConstraints": {
    "fifo_locations_records": {
      "dma_fifos": {
        "r1": {
          "tile_type": "core",
          "row": 0,
          "column": 0,
          "size": 16,
          "offset": 8,
          "bankId": 2
        },
        "r2": {
          "tile_type": "core",
          "row": 0,
          "column": 1,
          "size": 16,
          "offset": 9
        }
      }
    }
  }
}
```

```

    },
    "r4": {
      "tile_type": "mem",
      "row": 2,
      "column": 4,
      "size": 16,
      "offset": 6,
      "bankId": 2
    }
  },
  "stream_fifos": {
    "r3": {
      "tile_type": "shim",
      "row": 1,
      "column": 3,
      "channel": 1
    }
  }
},
"mygraph.k2.in[0]": {
  "not_collocated_nodes": ["mygraph.k1"],
  "fifo_locations": ["r1", "r2", "r3"]
},
"mygraph.k4.in[0]": {
  "fifo_locations": ["r1", "r2", "r4"]
}
}
}

```

Global Constraints

Global constraints are specified in the GlobalConstraints section.

Syntax

```

{
  "GlobalConstraints": {
    <constraint>[,
    <constraint>...]
  }
}
<constraint> ::= areaGroup
               | IsomorphicGraphGroup

```

Example

```

{
  "GlobalConstraints": {
    "areaGroup": {
      "name": "root_area_group",
      "nodeGroup": ["mygraph.k1", "mygraph.k2"],
      "tileGroup": ["(2,0):(2,3)"],
      "shimGroup": ["0:3"]
    },
    "isomorphicGraphGroup": {
      "name": "isoGroup1",

```

```

    "referenceGraph": "clipGraph0",
    "stampedGraphs": [ "clipGraph1", "clipGraph2" ]
  }
}

```

Area Group Constraint

The area group constraint specifies a range of tile and/or shim locations to which a group of one or more nodes can be mapped. The area group constraint can be specified with the following properties.

- **groups:** Specify the collection of groups. The groups can be:
 - **tile-type:** Specify the tile-type for the group. Supported tile-types are `aie_tile`, `shim_tile`, or `memory_tile`.
 - **column_min:** Column index for lower left corner of the group.
 - **row_min:** Row index for lower left corner of the group.
 - **column_max:** Column index for upper right corner of the group.
 - **row_max:** Row index for upper right corner of the group.
- **contain_routing:** A boolean value that when specified true ensures all routing, including nets between nodes contained in the `nodeGroup`, is contained within the area group.
- **exclude_routing:** A boolean value that when specified true ensures all routing, excluding nets between nodes from the `nodeGroup`, is excluded from the area Group.
- **exclude_placement:** A boolean value that when specified true prevents all nodes not included in `nodeGroup` from being placed within the area group bounding box.

An AI Engine tile or memory tile range is in the form of `(column, row):(column, row)`, where the first tile is the bottom left corner and the second tile the upper right corner. The column and row values are zero based, where the zeroth row is counted from the bottom-most compute processor and the zeroth column is counted from the left-most column.

A shim range is in the form of `(column):(column)`, where the first value is the left-most column and the second value the right-most column. The column is zero based, where the zeroth row is counted from the bottom-most compute processor and the zeroth column is counted from the left-most column. The shim range also allows an optional channel to be specified, for example, `(column, channel):(column, channel)`.

The area group is used to exclude a range on the device from being used by the compiler for mapping and optionally routing as follows.

- To exclude a range from router, set `nodeGroup` and set **exclude_placement** to true.
- To exclude a range from mapper, set `nodeGroup` and set **exclude_routing** to true.

- To include all nets within a nodeGroup for router and mapper, set nodeGroup and set **contain_routing** to true.

Note: There can be any number of area group constraints in the Global Constraints section, as long as each constraint has a unique name.

Syntax

```
"areaGroup": {
  "name": string,

  "nodeGroup": [<node list>], (*optional)
  "tileGroup": [<tile list>], (*optional)
  "shimGroup": [<shim list>], (*optional)
  "contain_routing": bool, (*optional)
  "exclude_routing": bool, (*optional)
  "exclude_placement": bool, (*optional)
}

<node list> ::= <node name>[,<node name>...]

<tile array> ::= <tile value>[,<tile value>...]
<tile value> ::= <tile range> | <tile address>
<tile range> ::= "<tile address>[:<tile address>]"
<tile address> ::= (<column>, <row>)

<shim array> ::= <shim value>[,<shim value>...]
<shim value> ::= <shim range> | <shim address>
<shim range> ::= "<shim address>[:<shim address>]"
<shim address> ::= (<column>[,<channel>])

<node name> ::= string
<column> ::= integer
<row> ::= integer
<channel> ::= integer
```

Example

```
{
  "GlobalConstraints": {
    "areaGroup": {
      "name": "mygraph_area_group",
      "nodeGroup": ["mygraph.k1", "mygraph.k2"],
      "tileGroup": ["(2,0):(2,3)"],
      "shimGroup": ["0:3"]
    }
  }
}
```

Example of Exclude

```
{
  "GlobalConstraints": {
    "areaGroup": {
      "name": "mygraph_excluded_area_group",

      "tileGroup": ["(3,0):(4,3)"],
```

```

        "shimGroup": ["3:4"],
        "exclude_routing": true
    }
}

```

IsomorphicGraphGroup Constraint

The isomorphicGraphGroup constraint is used to specify isomorphic graphs that are used in the stamp and repeat flow.

Syntax

```

"isomorphicGraphGroup": {
    "name": string,
    "referenceGraph": <reference graph name>,
    "stampedGraphs": [<stamped graph name list>]
}

```

Example

```

"isomorphicGraphGroup": {
    "name": "isoGroup",
    "referenceGraph": "tx_chain0",
    "stampedGraphs": ["tx_chain1", "tx_chain2", "tx_chain3"]
}

```

General Description

The stamp and repeat feature of the AI Engine compiler can be used when the same graph has multiple instances that can be constrained to the same geometry in AI Engines. There are two main advantages to using this feature when the same graph is instantiated multiple times.

- **Small variation in performance:** All graphs will have very similar throughput because buffers and kernels are mapped identically with respect to each other. Throughput might not be exactly identical due to differences in routing. However, it will be much closer than when stamping is not used.
- **Smaller run time of AI Engine compiler:** Because the AI Engine compiler only solves a reference graph instead of the entire design, run time required will be significantly less than the default flow.

Capabilities and Limitations

If required, you are allowed to stamp multiple different graphs. For example, if a design contains four instances of a graph called `tx_chain` and four instances of `rx_chain`, then both sets of graphs can be independently stamped. This feature is only supported for designs which have one or more sets of isomorphic graphs, with no interaction between the different isomorphic graph sets. All reference and stamped graphs must have area group constraints. You must declare identical size area groups for each instance of the graph that needs to be stamped. All area groups must be non-overlapping. For example:

```
"areaGroup": {
  "name": "ant0_cores",
  "nodeGroup": [ "tx_chain0*" ],
  "tileGroup": [ "(0,0):(3,3)" ]
},
"areaGroup": {
  "name": "ant1_cores",
  "nodeGroup": [ "tx_chain1*" ],
  "tileGroup": [ "(0,4):(3,7)" ]
},
```

Note: The node group must contain all node instances in the graphs to be stamped. Pattern matching can be used as in shown in the previous example.

You must declare an `isomorphic_graph_group` in the constraints file that specifies the reference graph and the stamped graphs. For example:

```
"isomorphicGraphGroup": {
  "name": "isoGroup",
  "referenceGraph": "tx_chain0",
  "stampedGraphs": [ "tx_chain1", "tx_chain2" ]
},
```

In this case, the `tx_chain0` graph is the reference and its objects will be placed first and stamped to graph `tx_chain1` and `tx_chain2`. Area groups must follow these rules for number of rows: the number of rows for all identical graphs (reference + stamp-able ones) must be the same, and must begin and end at the same *parity* of row, meaning if the reference graph's `tileGroup` begins at an even row and ends at an odd row, then all of the stamped graphs must follow the same convention. This limitation occurs because of the mirrored tiles in AI Engine array. In one row, the AI Engine is followed by a memory group and in the next row the memory group is followed by an AI Engine within a tile.



RECOMMENDED: To add absolute, co-location, or other constraints to your design, only add constraints to the reference graph and all these constraints will automatically be applied to the stamped graphs.



IMPORTANT! Only top-level graphs can be stamped. You cannot instantiate a single graph at the top level and stamp graphs at a lower level of a hierarchy.

Event Trace Reference

Simulation Event Trace

An event in the AI Engine simulator gives information about a system at a specific instant in time. An event that is associated with a time stamp, type and set of data values, is referred to as a payload. The interpretation of the payload depends on the type of the event. A time stamp allows ordering of events, computation of causality relationships, and implementation of verifiers on a sequence of events.

For event modeling purposes, the key modules of the AI Engine array are processor cores, DMA, lock modules, memory and I/O streams. Each module can be viewed as an event generator/responder. Each module receives an event and responds to the event. As a response, new events could be generated. Events are classified based on the generator of the events. The time stamp is not mentioned, explicitly, in the event definitions. Each event is described with a payload, which is a couple of values associated with the event. Each AI Engine, memory, DMA, or lock is addressable by a 2-dimensional index <col, row>, which is the column and row index in the AI Engine array. Some of the AI Engine events are shown in the following table. All events have a time stamp, which is omitted from the table.

Table 94: AI Engine Events

Event Name	Values	Description
CORE_RESET	<col, row>	Core <col, row> is out of reset and starts executing code from PC0.
CORE_WAIT	<col, row>	Core <col, row> is waiting for lock, stream input, or memory response.
CORE_READY	<col, row>	Core <col, row> is ready after a stall is serviced.
ACQ_READ_LOCK_REQ ACQ_WRITE_LOCK_REQ	<col, row, lockid, dir, pc, info>	Core <col, row> requests for a lock to one of its neighboring memory blocks. The <pc> is the current program counter. The <info> is the symbolic information.
REL_LOCK_REQ	<col, row, lockid, dir>	Core <col, row> releases a lock in a neighboring lock module.

Table 94: AI Engine Events (cont'd)

Event Name	Values	Description
CORE_READ_REQ	<col, row, address, dir, port, bank, data, pcinfo, addrinfo>	Core <col, row> requests a read at a memory address. The address is a 15-bit address local to the memory. Together, the <dir, address> forms a 17-bit address. The <dir> field specifies [E, W, N, S] direction. The <port> specifies the load port (A,B), and <bank> specifies access to an odd or even bank. Symbolic information about program variables and the <pc> value is available in the event.
CORE_WRITE_REQ	<col, row, address, dir, bank, data, size.pcinfo, addrinfo>	Core <col, row> writes data at a particular 15-bit address. The <dir> specifies the direction of the memory being accessed. Symbolic information about the <pc> value and data symbol are available in the event.
PC_CHANGE	<col, row, pc, info>	Core <col, row> shows the change in the <pc> value. It is useful for identifying loops.
DMA_S2MM_ACQ_LOCK	<col, row, ch>	DMA requests a lock. The <ch> field represents the DMA channel number. The supported channels are 0 and 1.
DMA_S2MM_IDLE	<col, row, bd, ch>	DMA in an idle state. <bd> is the buffer descriptor.
DMA_S2MM_START	<col, row, bd, ch, start>	Start DMA transfer at the start address
DMA_S2MM_DONE	<col, row, bd, ch>	End DMA transfer at the end address
DMA_S2MM_LOCKSTALL	<col, row, bd, ch>	DMA is stalled to get a lock
DMA_S2MM_LOCKSTALL_RELEASE	<col, row, bd, ch>	DMA gets a lock and stall is released
DMA_MM2S_ACQ_LOCK	<col, row, ch>	DMA <col, row> requests a lock
DMA_MM2S_IDLE	<col, row, ch>	DMA <col, row> in an idle state
DMA_MM2S_START	<col, row, bd, ch, start>	Start DMA transfer at <col, row> at the start address, the <bd> is the buffer descriptor.
DMA_MM2S_DONE	<col, row, bd, ch>	End DMA transfer at <col, row> at the end address, the <bd> is the buffer descriptor.
DMA_MM2S_LOCKSTALL	<col, row, bd, ch>	DMA is stalled to get a lock
DMA_MM2S_LOCKSTALL_RELEASE	<col, row, bd, ch>	DMA gets a lock and stall is released
IO	<dir, id, data >	An I/O event represents new data on an input stream. The <dir> [E, W, N, S] and <id> [0-15] together represent a physical stream, the data field represents 32-bit data appearing on the stream.
DATA_HEAD	<col, row, name, netid, pktid, idx>	Core <col, row> transfers packet switched data header with routing <name>, on net<netid>. This packet has packet ID <pktid>. <idx> is reserved.
DATA_START	<col, row, name, netid, pktid, idx>	Data payload (either packet switched or circuit switched) start. All fields follow the meaning of DATA_HEAD fields.
DATA_RESUME	<col, row, name, netid, pktid, idx>	Data payload resumed after being stalled. All fields follow the meaning of DATA_HEAD fields.
DATA_STALL	<col, row, name, netid, idx>	Data payload stalled within packet payload. All fields follow the meaning of DATA_HEAD fields.
DATA_END	<col, row, name, netid, idx>	Data payload end (either packet switched or circuit switched). All fields follow the meaning of DATA_HEAD fields.
PL_TO_SHIM, SHIM_TO_PL	<name, col, channelId, data0, data1, tlast>	Data movement between the PL and AI Engine array interface (SHIM). Name shows the PL block name.

Table 94: AI Engine Events (cont'd)

Event Name	Values	Description
PL2PL, PL2PL_E	<name, port, data0, data1, tlast, tkeep>	PL port activity start and end. Name and port show PL block name/port.
DM_READ_REQ	<col, row, portname>	A read request event on a port of a memory module. A read port can be a core, DMA, or memory-mapped AXI4 interface read request port.
DM_WRITE_REQ	<col, row, portname>	A write request event on a port of a memory module. A write port can be a core, DMA, or memory-mapped AXI4 interface write request port.
DM_BANK_CONFLICT	<col, row, bankid, bank>	A bank conflict event shows a bank conflict because multiple requests are sent to the same bank of a memory module.

Hardware Event Trace

The following table contains all the possible events produced during hardware event tracing. If looking at the `ctf` data produced by `hwanalyze` the description provides information about the event being called.

An example of how to read the event is if the `functions` event is selected only the `FUN_CALL` and `FUN_RETURN` are produced and provide details about the tile the function resides on and the program counter (PC), and whether the event is calling the function, or if the function is being called.

Table 95: Hardware Event Trace

Event Name	Values	Description
FUN_CALL	<col, row, PC, val, info>	PC: function call instruction PC Val: 0 or 2 Info: calling function name with val 0 called function name with val 2
FUN_RETURN	<col, row, PC, val, info>	PC: function return instruction PC Val: 0 or 2 Info: Returning function name with val 0 Returned function name with val 2
MEM_STALL	<col, row, val, info>	Val: 23 (fixed) Info: Function name
MEM_STALL_RELEASE	<col, row, val, info>	Val: 2 (fixed) Info: Function name

Table 95: Hardware Event Trace (cont'd)

Event Name	Values	Description
STREAM_STALL	<col, row, val, info>	Val: 24 (fixed) Info: Function name
STREAM_STALL_RELEASE	<col, row, val, info>	Val: 2 (fixed) Info: Function name
CASCADE_STALL	<col, row, val, info>	Val: 25 (fixed) Info: Function name
CASCADE_STALL_RELEASE	<col, row, val, info>	Val: 2 (fixed) Info: Function name
LOCK_STALL	<col, row, val, info>	Val: 26 (fixed) Info: Function name
LOCK_STALL_RELEASE	<col, row, val, info>	Val: 2 (fixed) Info: Function name
DMA_S2MM_RUNNING	<col, row, ch, val>	Ch: channel ID Val: 0 (activity ends) 1 (activity starts)
DMA_MM2S_RUNNING	<col, row, ch, val>	Ch: channel ID Val: 0 (activity ends) 1 (activity starts)
OVERRUN	<col, row>	Overrun occurred in tile <col, row>

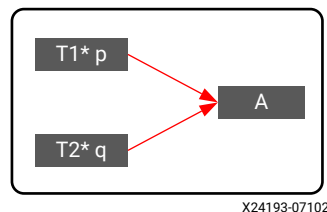
Using the Restrict Keyword in AI Engine Kernels

The use of restrict keyword (`__restrict`) is permitted in the AI Engine kernel C++ code. This appendix highlights Xilinx recommendations for using the restrict keyword in the context of AI Engine kernel code.

Pointer Aliasing

Pointer aliasing refers to the situation where the same memory location can be accessed using different pointer names. The strict aliasing rule in C/C++ means that pointers are assumed not to alias if they point to fundamentally different types. Aliasing introduces strong constraints on program execution order. The following shows the aliasing of `p` and `q`.

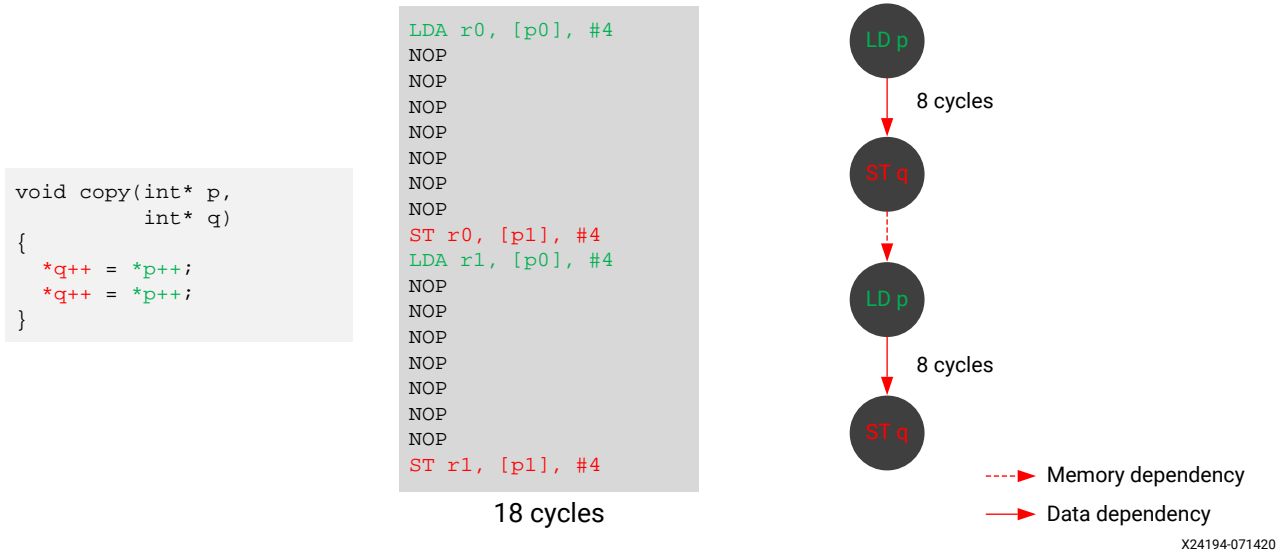
Figure 80: Pointer Aliasing



X24193-071020

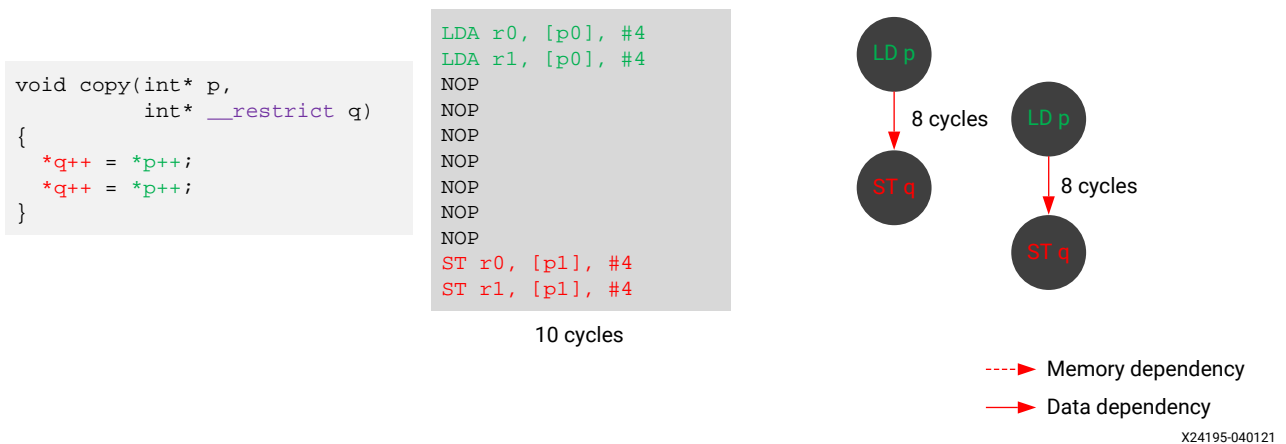
The following is an example of pointer aliasing, in which both the pointers `p` and `q` point to the same address. The assembly language code produced by the compiler is shown in the middle column, and the operations and clock cycles are shown on the right.

Figure 81: Aliasing Code Example



By adding the restrict keyword into this code example, the compiler can optimize the resulting assembly language to increase parallelization of the operations in hardware. The following example shows that using the restrict keyword to prevent aliasing uses fewer clock cycles to complete the same operation.

Figure 82: Use of Restrict Keyword to Avoid Aliasing



Memory Dependencies

Memory dependencies in the code can limit the kinds of optimizations attempted by the compiler. For example in the following code, `xyz` and pointers `p` and `q` might be unrelated. However, within the function code both pointer `p` and pointer `q` point to same global variable `xyz`. The compiler must guarantee the correct execution under both these conditions. Due to these kinds of memory dependencies the compiler needs to be conservative and limit optimizations.

Figure 83: Unrelated Pointers

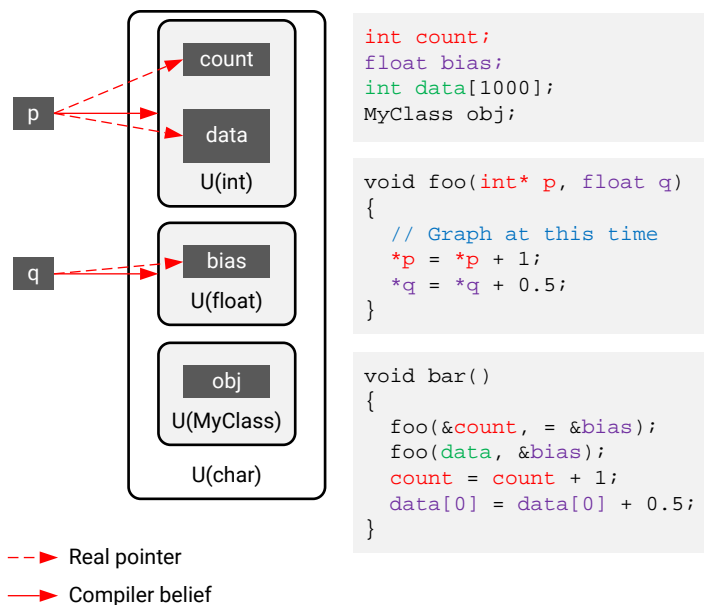
```
int xyz;
void bar(int *p, int *q)
{
    ...
    p=&xyz;
    ...
    q=&xyz;
    ...
}
```

X24245-071520

Strict Aliasing Rule

The strict aliasing rule dictates that pointers are assumed not to alias if they point to fundamentally different types, except for `char*` and `void*` which can alias to any other data type. This is shown in the following graphic which shows the object universes and the associated pointers.

Figure 84: Object Universes



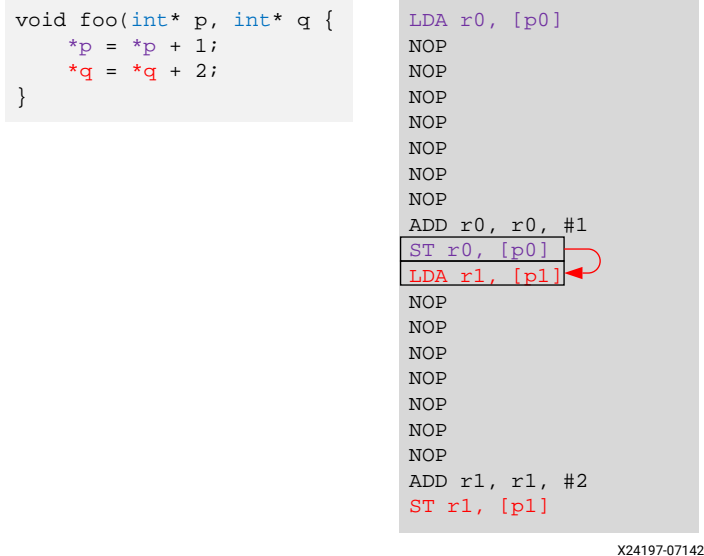
X24196-071020

- **Pointers are associated with a type universe: `U(T)`:** `T` is the template and in the preceding graphic the various templates are shown, including an `int` universe and a `float` universe; there is also a `MyClass` universe per design. Additionally there is a `char` universe that includes all universes by default.

- **Universes do not alias:** Pointer `p` can only point to any address within the `int` universe whereas pointer `q` can only point to any address within the `float` universe. Because of this pointer `p` and pointer `q` cannot be aliased.
- **Derived pointers point to the original universe:** Pointers derived from a restrict pointer are considered restrict pointers and point to the same restricted memory region. See [Derived Pointers](#).
- **`char*` universe contains all universes:** A `char` pointer can point to any variable in all universes.

For two pointers of the same type, as in the following, where both `p` and `q` are `int`, the compiler is conservative and aliasing is applied, resulting in loss of performance.

Figure 85: Loss of Performance



For two pointers of different types, as in the following example, where `p` is an `int` and `q` is `float`, the compiler applies the strict aliasing rule and an undefined behavior occurs if aliasing exists.

Figure 86: Two Pointers of Different Types

```
void foo(int* p, float* q {
    *p = *p + 1;
    *q = abs(*q);
}

LDA r0, [p0]; LDB r1, [p1]
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
ADD r0, r0, #1
ST r0, [p0]; AND r1, r1, r2
LDA r1, [p1]
```

X24198-071420

Restrict Keyword

The restrict keyword is mainly used in pointer declarations as a type qualifier for pointers. It does not add any new functionality. It allows you to tell the compiler about a potential optimization. Using `__restrict` with a pointer informs the compiler that the pointer is the only way to access the object pointed at, and the compiler does not need to perform any additional checks.

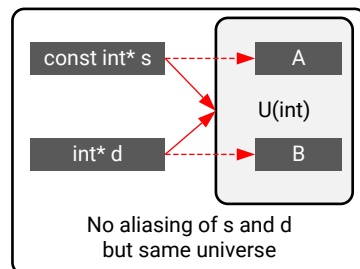
Note: If a programmer uses the restrict keyword and violates the above condition, undefined behavior can occur.

The following is another example with pointers that, by default, have no aliasing.

Figure 87: No Aliasing Example

```
void copy_block(int *d, const int *s) {
    for (int n = 0; n < 128; n++) {
        d[n] = s[n];
    }
}

MOV.s12 lc, #128
Loop_start:
    LDB r1, [p1]
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
ST r1, [p0] #4
```



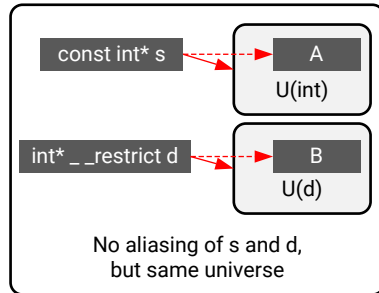
1153 cycles, 9 cycles loop

X24199-071520

Apply the restrict keyword for performance improvement. The following example shows no memory dependencies with other pointers.

Figure 88: No Memory Dependencies with Other Pointers

```
void copy_block(int * __restrict d, const int *s)
{
    for (int n = 0; n < 128; n++) {
        d[n] = s[n];
    }
}
```



```
MOV.s12 lc, #120
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
loop_start:
    LDA r1, [p1] #4; ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
```

137 cycles, 1 cycle loop

X24200-040121

Restrict Qualification

The C standard provides a specific pointer qualifier, `__restrict`, intended to allow more aggressive compiler optimization by explicitly stating data independence between whatever the pointer references and all other variables. For example :

```
int a; // global variable
void foo(int* __restrict p, int* q)
{
    for (...) { ... *p += a + *q; ...}
}
```

Now the analysis of `foo` can proceed with the knowledge that `*p` does not denote the same object as `*q` and `a`. So, `a` and `*q` can now be loaded once, before the loop.

Currently, the compiler front end does not disambiguate between different accesses to the same array. So, when updating one element of an array, it assumes that the complete array has changed value. The `__restrict` qualifier can be used to override this conservative assumption. This is useful when you want to obtain multiple independent pointers to the same array.

```
void foo(int A[])
{
    int* __restrict rA = A; // force independent access
    for (int i = ...)
        rA[i] = ... A[i];
}
```

In this example, the `__restrict` qualifier allows software pipelining of the loop; the next array element can already be loaded, while the previous one must still be stored. To maximize the impact of the `__restrict` qualifier, the compiler front end, by default, inserts a `chess_copy` operation in the initializer, as if was written:

```
int* __restrict rA = chess_copy(A);
```

This is needed to keep both pointers distinct within the optimizer (for example, no common subexpression elimination). This behavior can be disabled for the AI Engine compiler front end by means of the option `-mllvm -chess-implicit-chess_copy=false`. So, the `chess_copy` creates two pointers, while `__restrict` informs the compiler not to consider any mutual dependencies between the stores/loads through these pointers. For `__restrict` pointers having a local scope, the mutual independence assumption only holds during the lifetime of the `__restrict` pointer.

Pointers derived from a `__restrict` pointer (such as `rA+1` or through pointer intrinsics) keep the restrictness, that is, they are considered to point to the same restricted memory region.

Note: Details of `chess_copy` is available from the Chess Compiler User Manual, which can be found in the AI Engine [lounge](#).

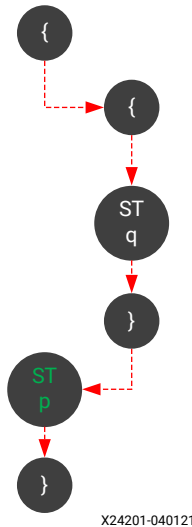
Undefined Behavior

Using the `restrict` keyword improves performance as shown in the previous topic. However, there are issues if the keyword is used inappropriately. The `__restrict` child pointers *must* be used in a different block-level scope than the parent pointers, such as pointer `p` and `q` as shown in the following example.

Working Example 1

Figure 89: Use of Restrict Keyword

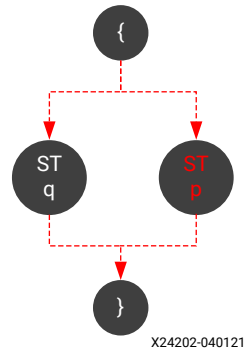
```
void foo(int *p)
{
    {
        int * __restrict q = p;
        *q = 3;
    }
    *p = 1; // The result is 1
}
```



Use of parent pointers in the same scope might break the `__restrict` contract which produces an undefined behavior, such as pointers `p` and `q` in the following example.

Figure 90: Undefined Behavior

```
void foo(int *p)
{
    int * __restrict q = p;
    *q = 3;
    *p = 1; // UNDEFINED!
}
```

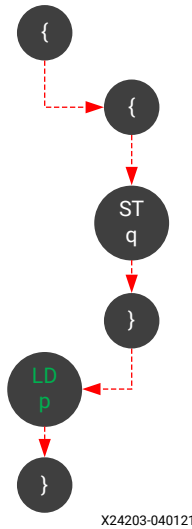


Working Example 2

This can also happen during the load operation, as shown in the green text (`return *p;`) in the following figure.

Figure 91: Load Operation

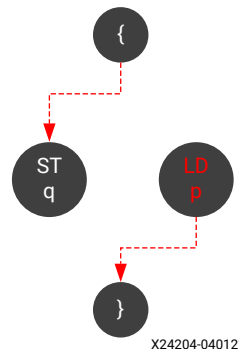
```
int bar(int *p)
{
    {
        int * __restrict q = p;
        *q = 3;
    }
    return *p; // Returns 3
}
```



The undefined behavior occurs when the restrict pointers are used within the same scope, such as pointers `p` and `q` in the following example.

Figure 92: Restrict Pointers in Same Scope

```
int bar(int *p)
{
    {
        int * __restrict q = p;
        *q = 3;
        return *p; // UNDEFINED!
    }
}
```



Working Example with Inline Function

The following code shows the working inline function call, in which pointer `p` and pointer `q` are used in different scopes.

Figure 93: Inline Function Calls

```
inline
void bar(int *q){
    *q = 3;
}

void foo(int *p)
{
    *p = 1;
    {
        int * __restrict q = p;
        bar(q);
    }
}
```

X24205-040121

The undefined behavior occurs when the restrict pointers are used within the same scope, such as pointers `p` and `q` in the following example.

Figure 94: Inline Function Calls in Same Scope

```
inline
void bar(int *q){
    *q = 3;
}

void foo(int *p)
{
    *p = 1;
    int * __restrict q = p;
    // UNDEFINED BEHAVIOR!
    bar(q);
}
```

X24206-040121

Scope of Restrict Keyword in Inline Function

When there are no other accesses within the scope, declaring the restrict pointer has no performance benefits.

Figure 95: Working Example with No Performance Benefits

```
inline int read(int *p) {
    int * __restrict q = p;
    return *q;
}

inline void write(int *p, int d){
    int * __restrict q = p;
    *q = d;
}

int foo(int *p1, int *p2) {
    write(p1, 4);
    return read(p2);
}
```

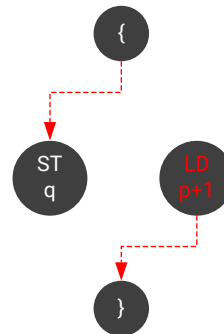
X24207-040121

In a special case, you can have non-aliasing accesses, as in the following example. Here the parent pointer, *p*, is used but points to a different location and therefore this is acceptable.

Figure 96: Special Case—Non-aliasing Accesses

```
int bar(int *p)
{
    int * __restrict q = p;
    *q = 3;
    return *(p+1); // Ok!
}
```

-----> Memory dependency
 -----> Data dependency

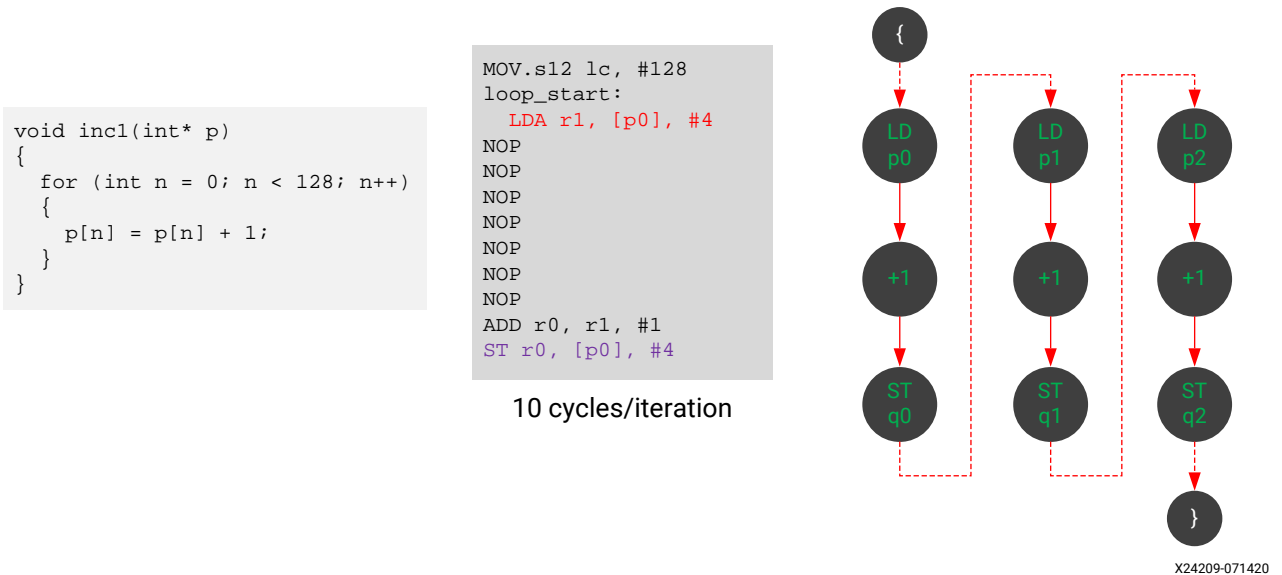


X24208-040121

Benefits of Using the Restrict Keyword for Read/Modify/Write Loops

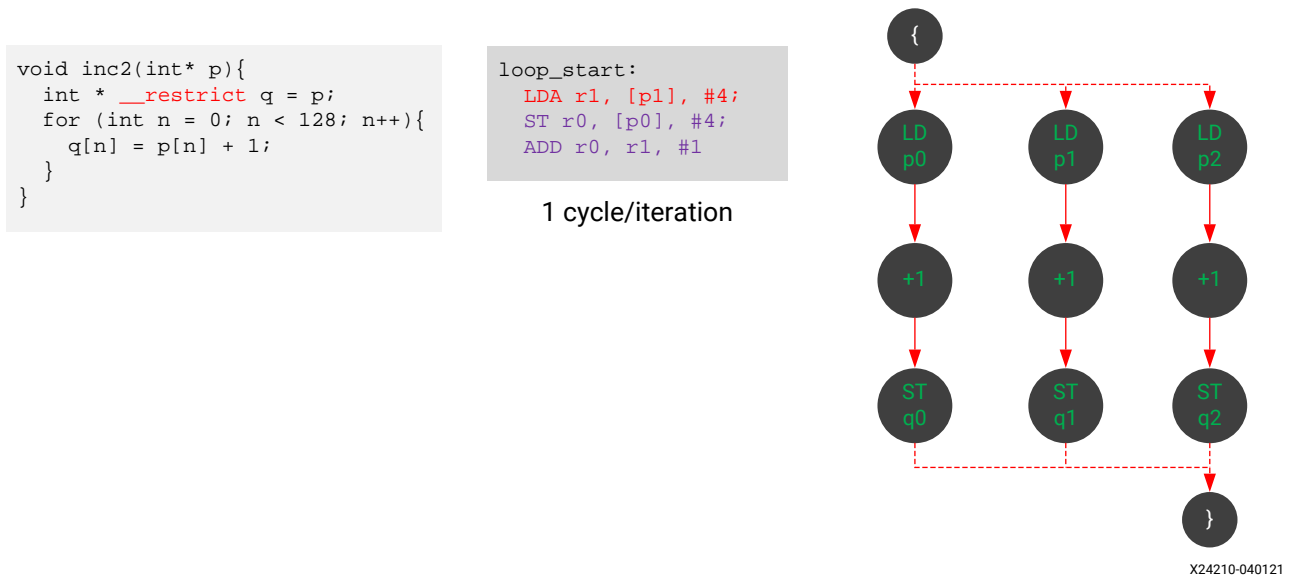
The following example works without the restrict keyword, but has poor performance.

Figure 97: Example Without Restrict Keyword



Adding the restrict keyword allows every iteration to access a different location where there is no aliasing between iterations (`__restrict`) and aliasing within iterations preserved by data dependency. The increased parallelization results in improved performance.

Figure 98: Add Restrict Keyword

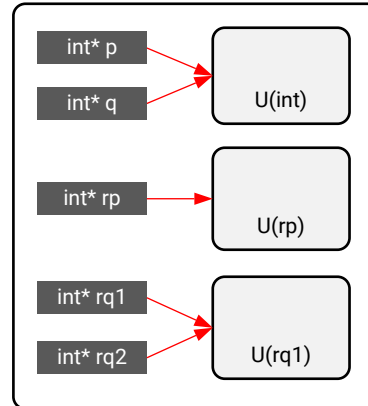


Derived Pointers

Pointers derived from a restrict pointer are considered restrict pointers and point to the same restricted memory region, as shown in the following example, where `rq2`, derived from `rq1` (defined as a restrict pointer) is also a restrict pointer and points to the same universe.

Figure 99: Pointers to Same Restricted Memory Region

```
void foo(int * p, int *q){
    int * __restrict rp = p;
    int * __restrict rq1 =
q;
    int * rq2 = rq1 + 3;
    ...
}
```

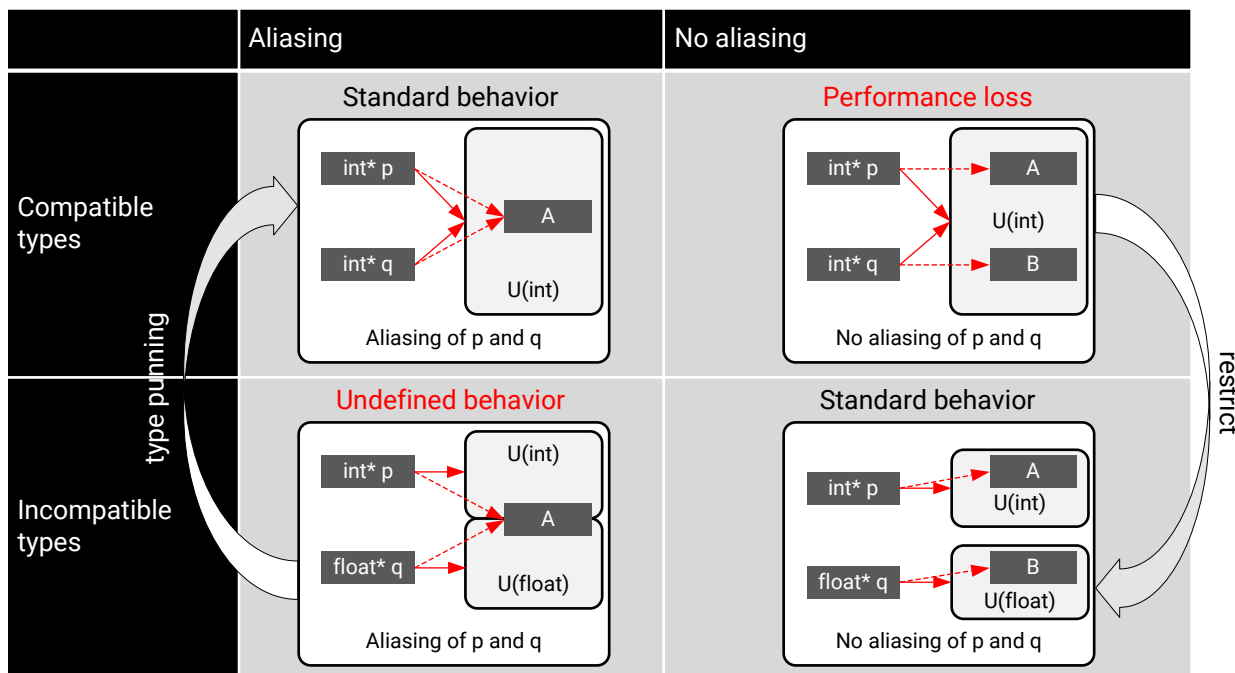


X24211-040121

Summary

Proper use of the restrict keyword (`__restrict`) in AI Engine kernel programming can result in performance gains and eliminate undefined behaviors in your code. However, be aware that when assigned to the same scope, the restrict pointers might result in undefined behavior in your design.

Figure 100: Restrict Keyword Use Summary



X24212-071420

Non-Templated Versions of Window and Stream APIs



IMPORTANT! Note that these APIs only work with intrinsic functions and not with AI Engine APIs.

Window Operations for Kernels

Window Data Types

Table 96: Supported Window Data Types

Input Window Types	Output Window Types
input_window_int8	output_window_int8
input_window_int16	output_window_int16
input_window_int32	output_window_int32
input_window_int64	output_window_int64
input_window_uint8	output_window_uint8
input_window_uint16	output_window_uint16
input_window_uint32	output_window_uint32
input_window_uint64	output_window_uint64
input_window_cint16	output_window_cint16
input_window_cint32	output_window_cint32
input_window_float	output_window_float
input_window_cfloat	output_window_cfloat

Moving the Current Read/Write Position Forward

In the following description, `<input_window_type>` stands for any of the allowed input window data types. Likewise, `<output_window_type>` stands for any of the allowed output window data types.

Purpose	Input Window Type	Output Window Type
To increase the current read/write position by the count times of the underlying window type.	<code>void window_incr(<input_window_type> *w, int count);</code>	<code>void window_incr(<output_window_type> *w, int count);</code>
To increase the current read/write position by four times the count times of the underlying window type.	<code>void window_incr_v4(<input_window_type> *w, int count);</code>	<code>void window_incr_v4(<output_window_type> *w, int count);</code>
To increase the current read/write position by eight times the count times of the underlying window type.	<code>void window_incr_v8(<input_window_type> *w, int count);</code>	<code>void window_incr_v8(<output_window_type> *w, int count);</code>
To increase the current read/write position by 16 times the count times of the underlying window type.	<code>void window_incr_v16(<input_window_type> *w, int count);</code>	<code>void window_incr_v16(<output_window_type> *w, int count);</code>
To increase the current read/write position by 32 times the count times of the underlying window type.	<code>void window_incr_v32(<input_window_type> *w, int count);</code>	<code>void window_incr_v32(<output_window_type> *w, int count);</code>
To increase the current read/write position by 64 times the count times of the underlying window type.	<code>void window_incr_v64(<input_window_type> *w, int count);</code>	<code>void window_incr_v64(<output_window_type> *w, int count);</code>

Moving the Current Read/Write Position Backward

In the following description, `<input_window_type>` stands for any of the allowed input window data types. Likewise, `<output_window_type>` stands for any of the allowed output window data types.

Purpose	Input Window Type	Output Window Type
To decrease the current read/write position by the count times of the underlying window type.	<code>void window_decr(<input_window_type> *w, int count);</code>	<code>void window_decr(<output_window_type> *w, int count);</code>
To decrease the current read/write position by four times the count times of the underlying window type.	<code>void window_decr_v4(<input_window_type> *w, int count);</code>	<code>void window_decr_v4(<output_window_type> *w, int count);</code>
To decrease the current read/write position by eight times the count times of the underlying window type.	<code>void window_decr_v8(<input_window_type> *w, int count);</code>	<code>void window_decr_v8(<output_window_type> *w, int count);</code>
To decrease the current read/write position by 16 times the count times of the underlying window type.	<code>void window_decr_v16(<input_window_type> *w, int count);</code>	<code>void window_decr_v16(<output_window_type> *w, int count);</code>
To decrease the current read/write position by 32 times the count times of the underlying window type.	<code>void window_decr_v32(<input_window_type> *w, int count);</code>	<code>void window_decr_v32(<output_window_type> *w, int count);</code>
To decrease the current read/write position by 64 times the count times of the underlying window type.	<code>void window_decr_v64(<input_window_type> *w, int count);</code>	<code>void window_decr_v64(<output_window_type> *w, int count);</code>

Reading Data from an Input Window

The following code reads a scalar typed value from an input window of the same type. The current position is not modified and both functional form (returns the value) and procedural form (modifies a reference argument) are provided.

```
int8 window_read(input_window_int8 *w);
int16 window_read(input_window_int16 *w);
int32 window_read(input_window_int32 *w);
int64 window_read(input_window_int64 *w);
uint8 window_read(input_window_uint8 *w);
uint16 window_read(input_window_uint16 *w);
uint32 window_read(input_window_uint32 *w);
uint64 window_read(input_window_uint64 *w);
cint16 window_read(input_window_cint16 *w);
cint32 window_read(input_window_cint32 *w);
float window_read(input_window_float *w);
cfloat window_read(input_window_cfloat *w);

void window_read(input_window_int8 *w, int8 &v );
void window_read(input_window_int16 *w, int16 &v );
void window_read(input_window_int32 *w, int32 &v );
void window_read(input_window_int64 *w, int64 &v );
void window_read(input_window_uint8 *w, uint8 &v );
void window_read(input_window_uint16 *w, uint16 &v );
void window_read(input_window_uint32 *w, uint32 &v );
void window_read(input_window_uint64 *w, uint64 &v );
void window_read(input_window_cint16 *w, cint16 &v );
void window_read(input_window_cint32 *w, cint32 &v );
void window_read(input_window_float *w, float &v );
void window_read(input_window_cfloat *w, cfloat &v );
```

The following code reads a 4-way vector of typed value from an input window of the same type. The current position is not modified and both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v4cint16 window_read_v4(input_window_cint16 *w);
v4int32 window_read_v4(input_window_int32 *w);
v4cint32 window_read_v4(input_window_cint32 *w);
v4int64 window_read_v4(input_window_int64 *w);
v4float window_read_v4(input_window_float *w);
v4cfloat window_read_v4(input_window_cfloat *w);

void window_read(input_window_cint16 *w, v4cint16 &v);
void window_read(input_window_int32 *w, v4int32 &v);
void window_read(input_window_cint32 *w, v4cint32 &v);
void window_read(input_window_int64 *w, v4int64 &v);
void window_read(input_window_float *w, v4float &v);
void window_read(input_window_cfloat *w, v4cfloat &v);
```

The following code reads an 8-way vector of typed value from an input window of the same type. The current position is not modified and both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v8int16 window_read_v8(input_window_int16 *w);
v8cint16 window_read_v8(input_window_cint16 *w);
v8int32 window_read_v8(input_window_int32 *w);
v8float window_read_v8(input_window_float *w);

void window_read(input_window_int16 *w, v8int16 &v);
void window_read(input_window_cint16 *w, v8cint16 &v);
void window_read(input_window_int32 *w, v8int32 &v);
void window_read(input_window_float *w, v8float &v);
```

The following code reads a 16-way vector of typed value from an input window of the same type. The current position is not modified and both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v16int8 window_read_v16(input_window_int8 *w);
v16uint8 window_read_v16(input_window_uint8 *w);
v16int16 window_read_v16(input_window_int16 *w);
v16cint16 window_read_v16(input_window_cint16 *w);
v16int32 window_read_v16(input_window_int32 *w);
v16cint32 window_read_v16(input_window_cint32 *w);
v16float window_read_v16(input_window_float *w);
v16cfloat window_read_v16(input_window_cfloat *w);

void window_read(input_window_int8 *w, v16int8 &v);
void window_read(input_window_uint8 *w, v16uint8 &v);
void window_read(input_window_int16 *w, v16int16 &v);
void window_read(input_window_cint16 *w, v16cint16 &v);
void window_read(input_window_int32 *w, v16int32 &v);
void window_read(input_window_cint32 *w, v16cint32 &v);
void window_read(input_window_float *w, v16float &v);
void window_read(input_window_cfloat *w, v16cfloat &v);
```

The following code reads a 32-way vector of typed value from an input window of the same type. The current position is not modified and both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v32int8 window_read_v32(input_window_int8 *w);
v32uint8 window_read_v32(input_window_uint8 *w);
v32int16 window_read_v32(input_window_int16 *w);
v32cint16 window_read_v32(input_window_cint16 *w);
v32int32 window_read_v32(input_window_int32 *w);
v32float window_read_v32(input_window_float *w);

void window_read(input_window_int8 *w, v32int8 &v);
void window_read(input_window_uint8 *w, v32uint8 &v);
void window_read(input_window_int16 *w, v32int16 &v);
void window_read(input_window_cint16 *w, v32cint16 &v);
void window_read(input_window_int32 *w, v32int32 &v);
void window_read(input_window_float *w, v32float &v);
```

The following code reads a 64-way vector of typed value from an input window of the same type. The current position is not modified and both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v64int8 window_read_v64(input_window_int8 *w);
v64uint8 window_read_v64(input_window_uint8 *w);
v64int16 window_read_v64(input_window_int16 *w);

void window_read(input_window_int8 *w, v64int8 &v);
void window_read(input_window_uint8 *w, v64uint8 &v);
void window_read(input_window_int16 *w, v64int16 &v);
```

Reading and Advancing an Input Window

The following code reads a scalar typed value from an input window of the same type and advances the window current position by one times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided.

```
int8 window_readincr(input_window_int8 *w);
int16 window_readincr(input_window_int16 *w);
int32 window_readincr(input_window_int32 *w);
int64 window_readincr(input_window_int64 *w);
uint8 window_readincr(input_window_uint8 *w);
uint16 window_readincr(input_window_uint16 *w);
uint32 window_readincr(input_window_uint32 *w);
uint64 window_readincr(input_window_uint64 *w);
cint16 window_readincr(input_window_cint16 *w);
cint32 window_readincr(input_window_cint32 *w);
float window_readincr(input_window_float *w);
cfloat window_readincr(input_window_cfloat *w);

void window_readincr(input_window_int8 *w, int8 &v );
void window_readincr(input_window_int16 *w, int16 &v );
void window_readincr(input_window_int32 *w, int32 &v );
void window_readincr(input_window_int64 *w, int64 &v );
void window_readincr(input_window_uint8 *w, uint8 &v );
void window_readincr(input_window_uint16 *w, uint16 &v );
void window_readincr(input_window_uint32 *w, uint32 &v );
void window_readincr(input_window_uint64 *w, uint64 &v );
void window_readincr(input_window_cint16 *w, cint16 &v );
void window_readincr(input_window_cint32 *w, cint32 &v );
void window_readincr(input_window_float *w, float &v );
void window_readincr(input_window_cfloat *w, cfloat &v );
```


The following code reads a 4-way vector of typed value from an input window of the same type and advances the window current position by four times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v4cint16 window_readincr_v4(input_window_cint16 *w);
v4int32 window_readincr_v4(input_window_int32 *w);
v4cint32 window_readincr_v4(input_window_cint32 *w);
v4int64 window_readincr_v4(input_window_int64 *w);
v4float window_readincr_v4(input_window_float *w);
v4cfloat window_readincr_v4(input_window_cfloat *w);

void window_readincr(input_window_cint16 *w, v4cint16 &v);
void window_readincr(input_window_int32 *w, v4int32 &v);
void window_readincr(input_window_cint32 *w, v4cint32 &v);
void window_readincr(input_window_int64 *w, v4int64 &v);
void window_readincr(input_window_float *w, v4float &v);
void window_readincr(input_window_cfloat *w, v4cfloat &v);
```

The following code reads an 8-way vector of typed value from an input window of the same type and advances the window current position by eight times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v8int16 window_readincr_v8(input_window_int16 *w);
v8cint16 window_readincr_v8(input_window_cint16 *w);
v8int32 window_readincr_v8(input_window_int32 *w);
v8float window_readincr_v8(input_window_float *w);

void window_readincr(input_window_int16 *w, v8int16 &v);
void window_readincr(input_window_cint16 *w, v8cint16 &v);
void window_readincr(input_window_int32 *w, v8int32 &v);
void window_readincr(input_window_float *w, v8float &v);
```

The following code reads a 16-way vector of typed value from an input window of the same type and advances the window current position by sixteen times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v16int8 window_readincr_v16(input_window_int8 *w);
v16uint8 window_readincr_v16(input_window_uint8 *w);
v16int16 window_readincr_v16(input_window_int16 *w);
v16cint16 window_readincr_v16(input_window_cint16 *w);
v16int32 window_readincr_v16(input_window_int32 *w);
v16cint32 window_readincr_v16(input_window_cint32 *w);
v16float window_readincr_v16(input_window_float *w);
v16cfloat window_readincr_v16(input_window_cfloat *w);

void window_readincr(input_window_int8 *w, v16int8 &v);
void window_readincr(input_window_uint8 *w, v16uint8 &v);
void window_readincr(input_window_int16 *w, v16int16 &v);
void window_readincr(input_window_cint16 *w, v16cint16 &v);
void window_readincr(input_window_int32 *w, v16int32 &v);
void window_readincr(input_window_cint32 *w, v16cint32 &v);
void window_readincr(input_window_float *w, v16float &v);
void window_readincr(input_window_cfloat *w, v16cfloat &v);
```

The following code reads a 32-way vector of typed value from an input window of the same type and advances the window current position by thirty-two times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v32int8 window_readincr_v32(input_window_int8 *w);
v32uint8 window_readincr_v32(input_window_uint8 *w);
v32int16 window_readincr_v32(input_window_int16 *w);
v32cint16 window_readincr_v32(input_window_cint16 *w);
v32int32 window_readincr_v32(input_window_int32 *w);
v32float window_readincr_v32(input_window_float *w);

void window_readincr(input_window_int8 *w, v32int8 &v);
void window_readincr(input_window_uint8 *w, v32uint8 &v);
void window_readincr(input_window_int16 *w, v32int16 &v);
void window_readincr(input_window_cint16 *w, v32cint16 &v);
void window_readincr(input_window_int32 *w, v32int32 &v);
void window_readincr(input_window_float *w, v32float &v);
```

The following code reads a 64-way vector of typed value from an input window of the same type and advances the window current position by sixty-four times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v64int8 window_readincr_v64(input_window_int8 *w);
v64uint8 window_readincr_v64(input_window_uint8 *w);
v64int16 window_readincr_v64(input_window_int16 *w);

void window_readincr(input_window_int8 *w, v64int8 &v);
void window_readincr(input_window_uint8 *w, v64uint8 &v);
void window_readincr(input_window_int16 *w, v64int16 &v);
```

Reading and Decrementing an Input Window

The following code reads a scalar typed value from an input window of the same type and decrements the window current position by one times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided.

```
int8 window_readdecr(input_window_int8 *w);
int16 window_readdecr(input_window_int16 *w);
int32 window_readdecr(input_window_int32 *w);
int64 window_readdecr(input_window_int64 *w);
uint8 window_readdecr(input_window_uint8 *w);
uint16 window_readdecr(input_window_uint16 *w);
uint32 window_readdecr(input_window_uint32 *w);
uint64 window_readdecr(input_window_uint64 *w);
cint16 window_readdecr(input_window_cint16 *w);
cint32 window_readdecr(input_window_cint32 *w);
float window_readdecr(input_window_float *w);
cfloat window_readdecr(input_window_cfloat *w);
```

```
void window_readdecr(input_window_int8 *w, int8 &v );
void window_readdecr(input_window_int16 *w, int16 &v );
void window_readdecr(input_window_int32 *w, int32 &v );
void window_readdecr(input_window_int64 *w, int64 &v );
void window_readdecr(input_window_uint8 *w, uint8 &v );
void window_readdecr(input_window_uint16 *w, uint16 &v );
void window_readdecr(input_window_uint32 *w, uint32 &v );
void window_readdecr(input_window_uint64 *w, uint64 &v );
void window_readdecr(input_window_cint16 *w, cint16 &v);
void window_readdecr(input_window_cint32 *w, cint32 &v);
void window_readdecr(input_window_float *w, float &v );
void window_readdecr(input_window_cfloat *w, cfloat &v);
```

The following code reads a 4-way vector of typed value from an input window of the same type and decrements the window current position by four times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v4cint16 window_readdecr_v4(input_window_cint16 *w);
v4int32 window_readdecr_v4(input_window_int32 *w);
v4cint32 window_readdecr_v4(input_window_cint32 *w);
v4int64 window_readdecr_v4(input_window_int64 *w);
v4float window_readdecr_v4(input_window_float *w);
v4cfloat window_readdecr_v4(input_window_cfloat *w);

void window_readdecr(input_window_cint16 *w, v4cint16 &v);
void window_readdecr(input_window_int32 *w, v4int32 &v);
void window_readdecr(input_window_cint32 *w, v4cint32 &v);
void window_readdecr(input_window_int64 *w, v4int64 &v);void
window_readdecr(input_window_float *w, v4float &v);
void window_readdecr(input_window_cfloat *w, v4cfloat &v);
```

The following code reads an 8-way vector of typed value from an input window of the same type and decrements the window current position by eight times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v8int16 window_readdecr_v8(input_window_int16 *w);
v8cint16 window_readdecr_v8(input_window_cint16 *w);
v8int32 window_readdecr_v8(input_window_int32 *w);
v8float window_readdecr_v8(input_window_float *w);

void window_readdecr(input_window_int16 *w, v8int16 &v);
void window_readdecr(input_window_cint16 *w, v8cint16 &v);
void window_readdecr(input_window_int32 *w, v8int32 &v);
void window_readdecr(input_window_float *w, v8float &v);
```

The following code reads a 16-way vector of typed value from an input window of the same type and decrements the window current position by sixteen times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v16int8 window_readdecr_v16(input_window_int8 *w);
v16uint8 window_readdecr_v16(input_window_uint8 *w);
v16int16 window_readdecr_v16(input_window_int16 *w);
v16cint16 window_readdecr_v16(input_window_cint16 *w);
v16int32 window_readdecr_v16(input_window_int32 *w);
v16cint32 window_readdecr_v16(input_window_cint32 *w);
v16float window_readdecr_v16(input_window_float *w);
v16cfloat window_readdecr_v16(input_window_cfloat *w);

void window_readdecr(input_window_int8 *w, v16int8 &v);
void window_readdecr(input_window_uint8 *w, v16uint8 &v);
void window_readdecr(input_window_int16 *w, v16int16 &v);
void window_readdecr(input_window_cint16 *w, v16cint16 &v);
void window_readdecr(input_window_int32 *w, v16int32 &v);
void window_readdecr(input_window_cint32 *w, v16cint32 &v);
void window_readdecr(input_window_float *w, v16float &v);
void window_readdecr(input_window_cfloat *w, v16cfloat &v);
```

The following code reads a 32-way vector of typed value from an input window of the same type and decrements the window current position by thirty-two times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v32int8 window_readdecr_v32(input_window_int8 *w);
v32uint8 window_readdecr_v32(input_window_uint8 *w);
v32int16 window_readdecr_v32(input_window_int16 *w);
v32cint16 window_readdecr_v32(input_window_cint16 *w);
v32int32 window_readdecr_v32(input_window_int32 *w);
v32float window_readdecr_v32(input_window_float *w);

void window_readdecr(input_window_int8 *w, v32int8 &v);
void window_readdecr(input_window_uint8 *w, v32uint8 &v);
void window_readdecr(input_window_int16 *w, v32int16 &v);
void window_readdecr(input_window_cint16 *w, v32cint16 &v);
void window_readdecr(input_window_int32 *w, v32int32 &v);
void window_readdecr(input_window_float *w, v32float &v);
```

The following code reads a 64-way vector of typed value from an input window of the same type and decrements the window current position by sixty-four times the size of the underlying data type. Both functional form (returns the value) and procedural form (modifies a reference argument) are provided. The memory data path is either 128-bits or 256-bits wide for vector operations.

```
v64int8 window_readdecr_v64(input_window_int8 *w);
v64uint8 window_readdecr_v64(input_window_uint8 *w);
v64int16 window_readdecr_v64(input_window_int16 *w);

void window_readdecr(input_window_int8 *w, v64int8 &v);
void window_readdecr(input_window_uint8 *w, v64uint8 &v);
void window_readdecr(input_window_int16 *w, v64int16 &v);
```

Writing Data to an Output Window

The following code writes a scalar typed value to an output window of the same type. The current position is not modified.

```
void window_write(output_window_int8 *w, int8 v);
void window_write(output_window_uint8 *w, uint8 v);
void window_write(output_window_int16 *w, int16 v);
void window_write(output_window_uint16 *w, uint16 v);
void window_write(output_window_cint16 *w, cint16 v);
void window_write(output_window_int32 *w, int32 v);
void window_write(output_window_uint32 *w, uint32 v);
void window_write(output_window_cint32 *w, cint32 v);
void window_write(output_window_int64 *w, int64 v);
void window_write(output_window_uint64 *w, uint64 v);
void window_write(output_window_float *w, float v);
void window_write(output_window_cfloat *w, cfloat v);
```

The following code writes a 4-way vector of a typed value to an output window of the same type. The current position is not modified.

```
void window_write(output_window_cint16 *w, v4cint16 v);
void window_write(output_window_int32 *w, v4int32 v);
void window_write(output_window_cint32 *w, v4cint32 v);
void window_write(output_window_int64 *w, v4int64 v);
void window_write(output_window_float *w, v4float v);
void window_write(output_window_cfloat *w, v4cfloat v);
```

The following code writes an 8-way vector of a typed value to an output window of the same type. The current position is not modified.

```
void window_write(output_window_int16 *w, v8int16 v);
void window_write(output_window_cint16 *w, v8cint16 v);
void window_write(output_window_int32 *w, v8int32 v);
void window_write(output_window_float *w, v8float v);
```

The following code writes a 16-way vector of a typed value to an output window of the same type. The current position is not modified.

```
void window_write(output_window_int8 *w, v16int8 v);
void window_write(output_window_uint8 *w, v16uint8 v);
void window_write(output_window_int16 *w, v16int16 v);
void window_write(output_window_cint16 *w, v16cint16 v);
void window_write(output_window_int32 *w, v16int32 v );
void window_write(output_window_cint32 *w, v16cint32 v);
void window_write(output_window_float *w, v16float v );
void window_write(output_window_cfloat *w, v16cfloat v);
```

The following code writes a 32-way vector of a typed value to an output window of the same type. The current position is not modified.

```
void window_write(output_window_int8 *w, v32int8 v);
void window_write(output_window_uint8 *w, v32uint8 v);
void window_write(output_window_int16 *w, v32int16 v);
void window_write(output_window_cint16 *w, v32cint16 v);
void window_write(output_window_int32 *w, v32int32 v );
void window_write(output_window_float *w, v32float v );
```

The following code writes a 64-way vector of a typed value to an output window of the same type. The current position is not modified.

```
void window_write(output_window_int8 *w, v64int8 v);
void window_write(output_window_uint8 *w, v64uint8 v);
void window_write(output_window_int16 *w, v64int16 v);
```

Writing and Advancing an Output Window

The following code writes a scalar typed value to an output window of the same type and advances the current position based upon that type.

```
void window_writeincr (output_window_int8 *w, int8 v);
void window_writeincr (output_window_uint8 *w, uint8 v);
void window_writeincr (output_window_int16 *w, int16 v);
void window_writeincr (output_window_uint16 *w, uint16 v);
void window_writeincr (output_window_cint16 *w, cint16 v);
void window_writeincr (output_window_int32 *w, int32 v );
void window_writeincr (output_window_uint32 *w, uint32 v );
void window_writeincr (output_window_cint32 *w, cint32 v);
void window_writeincr (output_window_int64 *w, int64 v);
void window_writeincr (output_window_uint64 *w, uint64 v);void
window_writeincr (output_window_float *w, float v );
void window_writeincr (output_window_cfloat *w, cfloat v);
```

The following code writes a 4-way vector of a typed value to an output window of the same type and advances the current position by four times the size of the underlying type.

```
void window_writeincr(output_window_cint16 *w, v4cint16 v);
void window_writeincr(output_window_int32 *w, v4int32 v );
void window_writeincr(output_window_cint32 *w, v4cint32 v);
void window_writeincr(output_window_int64 *w, v4int64 v );
void window_writeincr(output_window_float *w, v4float v );
void window_writeincr(output_window_cfloat *w, v4cfloat v);
```

The following code writes an 8-way vector of a typed value to an output window of the same type and advances the current position by eight times the size of the underlying type.

```
void window_writeincr(output_window_int16 *w, v8int16 v);
void window_writeincr(output_window_cint16 *w, v8cint16 v);
void window_writeincr(output_window_int32 *w, v8int32 v );
void window_writeincr(output_window_float *w, v8float v );
```

The following code writes a 16-way vector of a typed value to an output window of the same type and advances the current position by sixteen times the size of the underlying type.

```
void window_writeincr(output_window_int8 *w, v16int8 v);
void window_writeincr(output_window_uint8 *w, v16uint8 v);
void window_writeincr(output_window_int16 *w, v16int16 v);
void window_writeincr(output_window_cint16 *w, v16cint16 v);
void window_writeincr(output_window_int32 *w, v16int32 v );
void window_writeincr(output_window_cint32 *w, v16cint32 v);
void window_writeincr(output_window_float *w, v16float v );
void window_writeincr(output_window_cfloat *w, v16cfloat v);
```

The following code writes a 32-way vector of a typed value to an output window of the same type and advances the current position by thirty-two times the size of the underlying type.

```
void window_writeincr(output_window_int8 *w, v32int8 v);
void window_writeincr(output_window_uint8 *w, v32uint8 v);
void window_writeincr(output_window_int16 *w, v32int16 v);
void window_writeincr(output_window_cint16 *w, v32cint16 v);
void window_writeincr(output_window_int32 *w, v32int32 v );
void window_writeincr(output_window_float *w, v32float v );
```

The following code writes a 64-way vector of a typed value to an output window of the same type and advances the current position by sixty-four times the size of the underlying type.

```
void window_writeincr(output_window_int8 *w, v64int8 v);
void window_writeincr(output_window_uint8 *w, v64uint8 v);
void window_writeincr(output_window_int16 *w, v64int16 v);
```

Stream Operations for Kernels

Stream Data Types

Table 97: Supported Stream Data Types

Input Stream Types	Output Stream Types
input_stream_int8	output_stream_int8
input_stream_int16	output_stream_int16
input_stream_int32	output_stream_int32
input_stream_int64	output_stream_int64
input_stream_uint8	output_stream_uint8
input_stream_uint16	output_stream_uint16
input_stream_uint32	output_stream_uint32
input_stream_uint64	output_stream_uint64
input_stream_cint16	output_stream_cint16
input_stream_cint32	output_stream_cint32
input_stream_acc48	output_stream_acc48
input_stream_cacc48	output_stream_cacc48
input_stream_acc80	output_stream_acc80
input_stream_cacc80	output_stream_cacc80
input_stream_accfloat	output_stream_accfloat
input_stream_caccfloat	output_stream_caccfloat
input_stream_float	output_stream_float
input_stream_cfloat	output_stream_cfloat

Each of the data types in the table can be read or written from the AI Engine as either scalars or in vector groups. However, there are certain restrictions on valid groupings based on the bus data width supported on the AI Engine to programmable logic interface ports or through the stream-switch network. The valid combinations for AI Engine kernels are vector bundles totaling up to 32-bits or 128-bits. The accumulator data types are only used to specify cascade-stream connections between adjacent AI Engines. Its valid groupings are based on the 384-bit wide cascade channel between two processors.

Note: To use these data types, it is necessary to use `#include <adf.h>` in the kernel source file.

Reading and Advancing an Input Stream

AI Engine Operations

The following operations read data from the given input stream and advance the stream on the AI Engine. Because there are two input stream ports on the AI Engine, the physical port assignment is made by the AI Engine compiler automatically and conveyed as part of the stream data structure. Data values from the stream can be read one at a time or as a vector. In the latter case, unless all values are present, the stream operation stalls. The data groupings are based on the underlying single cycle, 32-bit stream operation or 4 cycle, 128-bit wide stream operation. The cascade connection reads all accumulator values in parallel.

```
int32 readincr(input_stream_int32 *w);
uint32 readincr(input_stream_uint32 *w);
cint16 readincr(input_stream_cint16 *w);
float readincr(input_stream_float *w);
cfloat readincr(input_stream_cfloat *w);

v16int8 readincr_v16(input_stream_int8 *w);
v16uint8 readincr_v16(input_stream_uint8 *w);
v8int16 readincr_v8(input_stream_int16 *w);
v4cint16 readincr_v4(input_stream_cint16 *w);
v4int32 readincr_v4(input_stream_int32 *w);
v2cint32 readincr_v2(input_stream_cint32 *w);
v4float readincr_v4(input_stream_float *w);

v8acc48 readincr_v8(input_stream_acc48 *w);
v4cacc48 readincr_v4(input_stream_cacc48 *w);
v4acc80 readincr_v4(input_stream_acc80 * str);
v2cacc80 readincr_v2(input_stream_cacc80 * str);
v8float readincr_v8(input_stream_accfloat * str);
v4cfloat readincr_v4(input_stream_caccfloat * str);
```

Writing and Advancing an Output Stream

AI Engine Operations

The following operations write data to the given output stream and advance the stream on the AI Engine. Because there are two output stream ports on the AI Engine, the physical port assignment is made by the AI Engine compiler automatically and conveyed as part of the stream data structure. Data values can be written to the output stream one at a time or as a vector. In the latter case, until all values are written, the stream operation stalls. The data groupings are based on the underlying single cycle, 32-bit stream operation or 4 cycle, 128-bit wide stream operation. Cascade connection writes all values in parallel.

```
void writeincr(output_stream_int32 *w, int32 v);
void writeincr(output_stream_uint32 *w, uint32 v);
void writeincr(output_stream_cint16 *w, cint16 v);
void writeincr(output_stream_float *w, float v);
void writeincr(output_stream_cfloat *w, cfloat v);

void writeincr_v16(output_stream_int8 *w, v16int8 v);
```

```
void writeincr_v16(output_stream_uint8 *w, v16uint8 v);
void writeincr_v8(output_stream_int16 *w, v8int16 v);
void writeincr_v4(output_stream_cint16 *w, v4cint16 v);
void writeincr_v4(output_stream_int32 *w, v4int32 v);
void writeincr_v2(output_stream_cint32 *w, v2cint32 v);
void writeincr_v4(output_stream_float *w, v4float v);

void writeincr_v8(output_stream_acc48 *w, v8acc48 v);
void writeincr_v4(output_stream_cacc48 *w, v4cacc48 v);
void writeincr_v4(output_stream_acc80* str, v4acc80 value);
void writeincr_v2(output_stream_cacc80* str, v2cacc80 value);
void writeincr_v8(output_stream_accfloat* str, v8float value);
void writeincr_v4(output_stream_caccfloat* str, v4cfloat value);
```

Using Streams in Parallel

For streaming input and output interfaces, when the performance is limited by the streaming interface, it is possible to use two streaming inputs or two streaming outputs in parallel. To use two parallel streams, it is recommended to use the following pairs of macros, where `idx1` and `idx2` are the two streams. Add the `restrict` keyword to the stream ports to optimize them for parallel processing.

```
READINCR(SS_rsrc1, idx1) and READINCR(SS_rsrc2, idx2)
READINCRW(WSS_rsrc1, idx1) and READINCRW(WSS_rsrc2, idx2)
WRITEINCR(MS_rsrc1, idx1, val) and WRITEINCR(MS_rsrc2, idx2, val)
WRITEINCRW(WMS_rsrc1, idx1, val) and WRITEINCRW(WMS_rsrc2, idx2, val)
```

The following example code shows two parallel input streams using pipelining with an interval of 1.

```
void simple( input_stream_int32 * restrict data0, input_stream_int32 *
restrict data1,
    output_stream_int32 * restrict out) {
    for(int i=0; i<1024; i++)
        chess_prepare_for_pipelining
    {
        int32_t d = READINCR(SS_rsrc1, data0) ;
        int32_t e = READINCR(SS_rsrc2, data1) ;
        WRITEINCR(MS_rsrc1,out,d+e);
    }
}
```

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

Documentation Navigator and Design Hubs

Xilinx[®] Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado[®] IDE, select **Help** → **Documentation and Tutorials**.
- On Windows, select **Start** → **All Programs** → **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

Note: For more information on DocNav, see the [Documentation Navigator](#) page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. Versal ACAP AI Engine Architecture Manual ([AM009](#))
2. Versal ACAP data sheets:
 - Versal Architecture and Product Data Sheet: Overview ([DS950](#))
 - Versal AI Core Series Data Sheet: DC and AC Switching Characteristics ([DS957](#))
3. Versal ACAP AI Engine Intrinsic Documentation ([UG1078](#))
4. Versal ACAP AI Engine Register Reference ([AM015](#))
5. Versal ACAP Design Guide ([UG1273](#))

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
12/17/2021 Version 2021.2	
Chapter 4: Window and Streaming Data API	Added more supported unsigned integer data types.
Specifying Run-Time Data Parameters	Clarified description.
Enhanced Programming Model	Changed heading of section.
AI Engine Compiler Options	Added Table 28: Optimization Options
Profiling the AI Engine	New section.
Profiling Graph Throughput	Added information.
Profiling the AI Engine in Hardware	New section.
Event Tracing in Hardware	New section.
Hardware Event Trace	New section.
Troubleshooting Event Trace in Hardware	New section.
10/22/2021 Version 2021.2	
AI Engine Tile Architecture	Updated Figure 1 .
Prepare the Kernels	Updated for AI Engine API.
Creating a Data Flow Graph (Including Kernels)	Figure representing graph connectivity added.
Chapter 4: Window and Streaming Data API	Updated data types for AI Engine API and template support.
Packet Switching Graph Constructs	Added an example for floating-point data.
Area Location Constraints	New section.
Hierarchical Constraints	Added information.
Enhanced Programming Model	New section.

Section	Revision Summary
AI Engine Compiler Options	New options added.
Chapter 10: Simulating an AI Engine Graph Application	Added simulation flow related information.
Data Snapshots	New section.
Deadlock Detection	New section.
Trace Report	New section.
Memory Access Violations and Valgrind	New section.
Memory Model	Updated information.
Simulation Output File Processing Considerations	New section.
adf::headers Constraint and aie_api Include Files	New section.
Software Emulation	New section.
Simulator Options	New option added.
Hardware Emulation	New section.
Reusing AI Engine Simulator Options	Added information about setting the AI Engine compiler <code>workdir</code> environment variable, as well as manual creation of the sim options file.
AI Engine Simulation-Based Profiling	New section.
Supported Window Data Types	Updated data types.
Supported Stream Data Types	
AI Engine Stall Analysis in Vitis Analyzer	New section.
Multi-Process and Multi-Thread Support for Controlling the AI Engine Graph	New section.
AI Engine Error Events	Updated errors, as well as debug tips.
Running Software Emulation	New section.
Area Group Constraint	Updated properties.
Creating the AI Engine Graph Project and Top-Level System Project	Updated screenshots.
Building and Running the System	Updated to add software emulation.
Chapter 16: Debugging the AI Engine Application	Debug information added.
Software Emulation Debug from the Vitis IDE	New section.
Software Emulation from Command Line	New section.
Using the Debug Environment	Updated screenshots.
Watchpoints	New section.
Vitis IDE Layout for Software Emulation Debug	New section.
Appendix D: Non-Templated Versions of Window and Stream APIs	Appendix describing non-templated version of window and stream data types and APIs.
07/19/2021 Version 2021.1	
FIFO Location Constraints	Updated FIFO constraints examples.
Supported Window Data Types	New topic.
Supported Stream Data Types	New topic.
Building a Bare-metal AI Engine in the Vitis IDE	Updated Step 4.
06/16/2021 Version 2021.1	
Run-Time Ratio	New topic.

Section	Revision Summary
Stream Data Types Reading and Advancing an Input Stream Writing and Advancing an Output Stream	New Stream types added.
Run-Time Parameter Support Summary	AI Engine RTP Support table added.
Stream Switch FIFO DMA FIFO AI Engine Tile DMA Performance	New FIFO topics.
Packet Switching Graph Constructs	Allowed number of packet streams updated.
Multicast Support	New topic.
Chapter 7: AI Engine/Programmable Logic Integration	Updated content.
Hardware Emulation and Hardware Flows	ADF_FRONTEND removed.
Performance Comparison Between AI Engine/PL and AI Engine/NoC Interfaces	New topic.
AI Engine Compiler Options	<ul style="list-style-type: none"> • Heap and Stack size updated. • <code>--broadcast-enable-core</code> CDO option added. • Tracing options updated. • <code>xlopt</code> updated.
Graph and Array Details	New section added.
AI Engine Compiler Guidance	New topic.
Reusing AI Engine Simulator Options	<code>--profile/AIE_PROFILE</code> added to options.
Enabling Third-Party Simulators	Simulators added and versions updated.
x86 Functional Simulator	Updated content and new sections added.
Viewing the Run Summary in the Vitis Analyzer	Content updated.
Trace View Data Visualization	New section.
Run-Time Event API Performance Counters	New topic.
Chapter 13: Programming the PS Host Application	ADF_FRONTEND removed.
Controlling the AI Engine Graph with the XRT C++ API	New topic.
Error Reporting Through the XRT API	<code>xbutil</code> scope updated.
Host Code Reference with ADF API and XRT API	Updated for <code>printf</code> .
Clocking the PL Kernels	Updated topic.
Compile the Embedded Application for the Cortex-A72 Processor	Code updated. <ul style="list-style-type: none"> • <code>aarch64-linux-gnu-g++</code> to <code>aarch64-xilinx-linux-g++</code>
Running Hardware Emulation	New section.
Chapter 15: Using the Vitis IDE	Screenshot updates.
Chapter 17: Mapper/Router Methodology	New chapter.
Event API	Removed extra Enumeration section.
FIFO Constraint	New topic.
Appendix C: Using the Restrict Keyword in AI Engine Kernels	Updated to C++.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2020-2021 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Kria, Spartan, Versal, Vitis, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. All other trademarks are the property of their respective owners.