Software Defined Radio with Reconfigurable Hardware and Software

A Framework for a TV Broadcast Receiver

by

Peter Ryser

peter.ryser@xilinx.com

#442 at Embedded Systems Conference 2005 – San Francisco

Abstract

With the increasing computing power of modern microprocessors it becomes feasible to process radio signals completely in software reducing the complexity of the hardware. Using embedded microprocessors in an FPGA and combining them with other advanced features of the fabric allows for a powerful solution on a single, reprogrammable chip.

In this paper we present the system architecture of a Software Defined Radio system running on embedded Linux in a single FPGA at the example of a digital TV broadcast receiver. We discuss aspects of hardware and software partitioning between the FPGA fabric and the embedded processors as well as between the processors themselves and show the advantages of Software Defined Radio on reconfigurable computing systems over existing solutions. The result is a working SDR application using hardware, software, and a fully embedded operating system.

Introduction

While traditional hardware is still in common use and will be for quite some time to come Software Define Radio (SDR) systems look very promising for the future. The flexibility of software based systems with regards to various use cases and adaptability to a variable environment will make them mainstream for many different applications.

They will be capable of replacing traditional hardware systems like FM radio or TV broadcast as well as cell based phone systems. Actually, it is quite possible that these different categories of radio systems will merge over time in a similar manner as we see cable based systems taking on new functions.

Cognitive radio systems automatically adjust to frequencies and power levels. However, whether such systems will ever become mainstream is under debate. We expect that development in that area will happen slowly as resistance from normative and controlling agencies will be immense. Even from a technical point of view many questions are unresolved.

Overall, moving a system from hardware centric to software centric is a major undertaking. We believe that FPGAs will not only be a big help in that transition but actually will become commodity components in such systems. The main argument for this to happen is their flexibility to partition hardware and software, in certain cases even dynamically. With that the transition will not need to occur at once or in a single monolithic step but can happen slowly in a controlled manner. The system designer moves components piece by piece and is capable of choosing the optimal way to implement either in soft hardware through the FPGA fabric or software altogether with embedded processors. Such an approach requires strong building blocks that are available in modern FPGAs.

Building Blocks

To successfully build a SDR system you need powerful building blocks. These building blocks can either be implemented in software in a traditional sense or in soft hardware with an FPGA. In the following few sections I will introduce the various building blocks.

Embedded Processors

Processors have been embedded into FPGAs for quite some time and a variety of processors are available ranging from eight bit microcontrollers up to 32 bit microprocessors from many different sources [1][2][3][4][5]. Most of these processors are implemented in soft logic by using the FPGA fabric. They are written in a high-level hardware description language like Verilog or VHDL. The number of soft-processors within an FPGA is only limited by the amount of logic resources available. A user can easily build a multi-processor solution.

High-end FPGAs embed hardened CPU cores into the FPGA fabric. At no extra cost these CPU cores offer advanced features like a memory management unit, large instruction and data side caches, and high-speed access ports to the FPGA fabric. The embedded CPU cores run at up to 450 MHz and comparing key numbers like Dhrystone MIPS/MHz and Dhrystone Mips/mW outperform soft processors by far [6]. An additional advantage of the embedded CPU cores lies in their compatibility with existent processor architectures. Tool sets like compilers and debuggers as well as embedded software libraries and operating systems are readily available from various vendors, which simplifies a migration from an external processor to the embedded CPU.

Naturally, the advantages of hard- and soft-processors can be combined using the high performance and advanced features of the hard processor for compute intensive tasks and applications where memory protection and/or virtual memory is key while soft-processors solve dedicated and localized tasks.

In either case, whether a soft processor solution, or a hardened CPU core, or a combination thereof is used, processors within the FPGA fabric offer a system architect much more flexibility with regards to the design of the complete system. The system architect decides on the implementation of the interconnection architecture based on performance and cost considerations. Lower performance and lower cost systems might build upon a bus architecture while high performance and low latency but higher cost systems use a switched interconnect combined with multi-ported memory subsystems [7].

User-Defined Instructions

Advanced processors embedded into the FPGA fabric allow the system designer to make use of user defined instructions (UDI). UDIs are custom instructions that enhance the capabilities of the regular processor by outsourcing functionality from the processor into the FPGA fabric.

Figure 1 identifies the key components of a processor architecture that supports UDIs at the example of the Auxiliary Processor Unit (APU) Controller and the PowerPC CPU embedded into the Virtex-4 FX FPGA family in relation to the Fabric Coprocessor

Module (FCM) soft coprocessor implemented in FPGA logic. To explain the operation of the APU controller and the interactions of the processor related to the execution units in soft logic, we can trace the step-by-step sequence of events that occur when an instruction is fetched from cache or memory.

Once the instruction reaches the decode stage, it is simultaneously presented to both the CPU and APU decode blocks. If the instruction is detected as a CPU instruction, the CPU will continue to execute the instruction as it would normally. Otherwise, within the same cycle, the CPU will look for a response from the APU controller. If the APU controller recognizes the instruction, it will provide the necessary information back to the CPU.

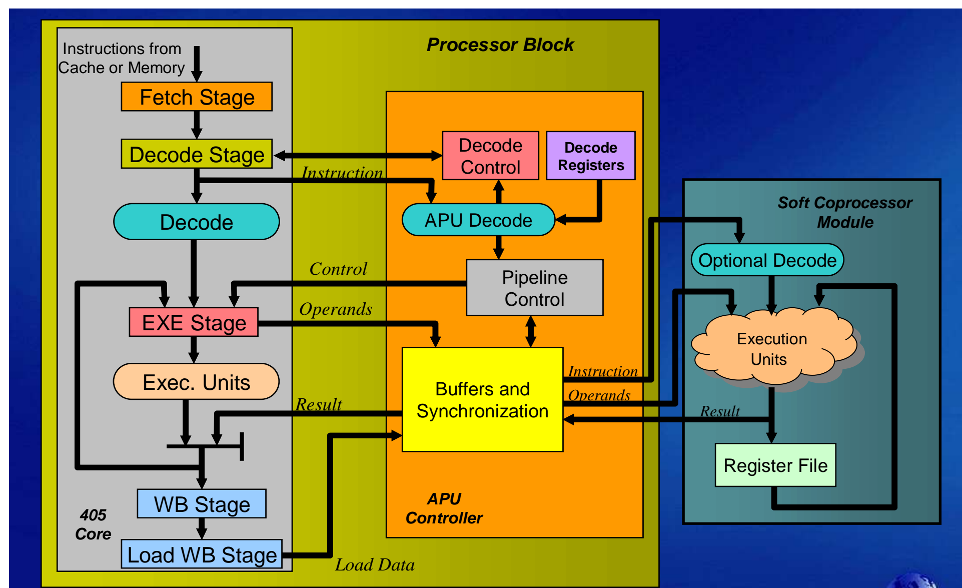

Figure 1: Integration of User Defined Instructions into the Virtex-4 FX FPGA family through the APU Interface

If the APU controller does not respond within that same cycle, the CPU generates an invalid instruction exception. If the instruction is a valid and recognized instruction, the necessary operands are fetched from the processor and passed to the FCM for processing. Because the PowerPC processor and the FCM reside in two separate clock domains, synchronization modules of the APU controller manage the speed difference. This allows the FCM to run at a slower speed than the processor. In this instance, the APU controller would receive the resultant data from the co-processor and at the proper execution time send the data back to the processor. The APU controller knows in advance, based on instruction type, if or when it will get the result.

Autonomous and Non-Autonomous Instructions

Two major categories of instructions exist - autonomous and non-autonomous. For autonomous instructions, the CPU continues issuing instructions and does not stall while the FCM operates on an instruction. This overlap of execution allows you to achieve high performance through techniques such as software pipelining.

On the other hand, during the synchronized execution, the CPU pipeline stalls while the FCM operates on an instruction. This feature allows you to implement synchronization semantics to pace the software execution with the hardware FCM latency.

Non-autonomous instruction types are further divided into blocking and non-blocking. If blocking, asynchronous exceptions or interrupts are blocked until the FCM instruction completes. Otherwise, if non-blocking, an exception or interrupt is taken and the FCM is flushed.

Software Description

Software engineers can access the FCM from within assembler or C code. On one side, compilers are capable of generating code that uses an FCM floating-point unit to calculate floating-point operations. Furthermore, assembler mnemonics are available for UDIs and the pre-defined load/store instructions, enabling you to place hardware-accelerated functions into the regular program flow. For the ultimate level of flexibility, you can define your own instructions designed specifically for the hardware functionality of the FCM.

You can easily use the pre-defined load/store instructions through high-level C macros. For example, in an application where the FCM is used to convert pixel data into the frequency domain, 8 pixels of 16 bits are transferred from main memory to an FCM register with a simple program:

```
unsigned short pixel_row[8];          // 8 pixels, each pixel has a size of 16 bits
lqfcm(0, pixel_row);                  // transfer a row of pixels to FCM register zero
```

The quadword load operation maintains cache coherency as the data is moved through the cache, if caching is enabled for the corresponding address space.

The FCM operation on the pixel data can start on an explicit command - for example, a UDI. However, for many applications the operation starts immediately after the FCM hardware detects the completion of the load instruction. The latter approach has many advantages:

- Simple software – A load operation moves the data from the memory to the FCM and starts the operation. A subsequent store instruction retrieves the result of the operation and stores it back to main memory.
- High data transfer rates – Quadword load and store operations take just a few cycles to complete. A single operation moves 16 bytes within that timeframe.
- Low latency – FCM load operations are simple to use. The processor completes the operation in a single cycle.

The principle of the RISC architecture uses a number of simple instructions on data stored in general purpose registers (GPR) to compute complex operations. User-defined instructions fall into this category but take the concept a step further in that the system architect defines the complexity of the operation on data stored in GPRs and FCM registers (FCR).

Again, from a software point of view the engineer codes user-defined instructions through C macros. A compiler capable of parsing UDIs recognizes mnemonics such as udi0fcm as a user-defined operation of the general form:

```
udi0fcm <FCRT5/RT5>, <FCRA5/RA5/imm>, <FCRB5/RB5/imm>
```

The target of the operation is either a GPR or a FCR. The operands are GPRs, FCRs, immediate values, or a combination. As you can see, the semantics are not defined by the instruction and depend on your intentions and the implementation in the FCM.

The following code sequence demonstrates the use of a user-defined instruction as an example of a complex add operation:

```
struct complex {
        int r, i;           // 32 bit integer for real and imaginary parts
};

complex a, b, r;

ldfcm(0, &a);           // load complex number a into FCM register 0
ldfcm(1, &b);           // load complex number b into FCM register 1
udi0fcm(2, 1, 0);       // udi0fcm computes r = a + b, where r is stored in FCM register 2
stdfcm(&r, 2);          // store complex result from FCM register 2 to variable r
```

To increase readability of the code you can redefine the user-defined instruction with regular C preprocessor constructs. Instead of using the udi0fcm() macro, you can redefine it to a more comprehensible complex_add() macro with

```
#define complex_add(r, a, b) udi0fcm(r, a, b)
```

and change the listing to call complex_add(2, 1, 0) instead of udi0fcm(2, 1, 0).

System architects can partition their tasks into hardware- and software-executed pieces that are efficiently and precisely interfaced to one another through the use of the APU controller. This partitioning can be done statically during the initial system configuration or dynamically during the program execution. Using the direct processor/FPGA coupling presented by the APU controller and its high throughput interfaces, hardware/software synchronization is greatly simplified and performance is significantly improved.

High-End Signal Processing Power

While you can implement high-performance signal processing algorithms in the FPGA fabric, state-of-the-art FPGAs provide dedicated function blocks for fast and wide multiplication or even multiply-accumulate operations. Figure 2 shows the block diagram of a digital signal processing (DSP) slice.

It contains a 2's complement signed 18x18-bit multiplier followed by a three-input adder/subtractor. Feedback paths and configurable operation modes allow for more than

forty different operations such as addition, multiplication, accumulation, and multiplexing. You can cascade multiple DSP slices without using the FPGA fabric at optimal speed.

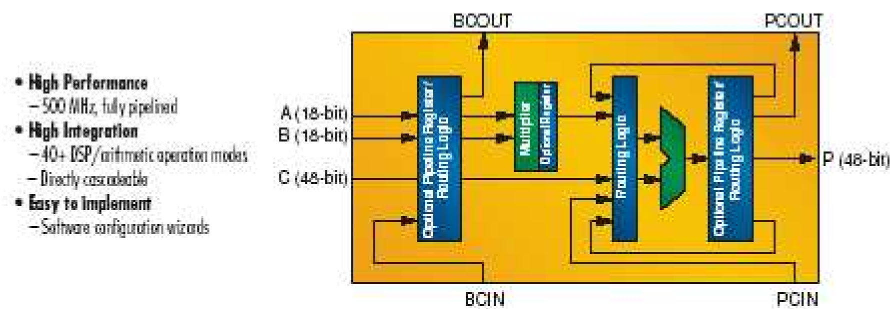

Figure 2: High-Performance Digital Signal Processing Slice in the Virtex-4 FPGA family

The DSP slices run at speeds up to 500 MHz and use only 57 uW/MMAC. All these characteristics make them ideal building blocks for software defined radio.

Partial Reconfiguration

The commonly accepted terminology for partial reconfiguration of an FPGA is the replacement of one or multiple functional blocks with a different implementation. The reconfigured block can be as small as a single lookup table (LUT) or flip-flop (FF) but typically consists of a multitude of basic building blocks. The parts that are reconfigured are contained in frames.

In the past a frame span a complete column, i.e. from top to bottom of the FPGA. This architecture made it tricky to maintain the integrity of wires that were routed through the frame and that should not have been affected by the reconfiguration. In certain cases such wires are actually needed for the partial reconfiguration. Current FPGA families allow partial reconfiguration on rectangular shapes [8]. Other functionality can be completely blocked out from these areas. Virtual ports into and out of the reconfigurable area provide the communication mechanisms to and from the rest of the FPGA fabric.

With embedded processors in the same FPGA as the hardware logic the terminology for partial reconfiguration needs to be extended. It is now possible to replace pieces or the entirety of a software program or data with a different implementation. This is mainly useful in, but not restricted to, a multi-processor environment where the main processor reconfigures the secondary processor in a running system. The reconfigured processor acts as a small controller for a part of the system. Using partial reconfiguration on the software keeps the code size small for a particular application or task and fast since cache size is limited. Applications can be engineered to completely fit into caches and execute at maximum speed.

While the initial configuration of most FPGAs is initiated through external components like a serial EEPROM or an external microcontroller, partial reconfiguration can now be initiated through internal configuration access ports. With that, parts of the running FPGA, either hardware or software or a combination of the two, can be reconfigured from a processor embedded within the FPGA.

Operating System

The operating system (OS) is an integral component of the whole system. Many different embedded operating systems are available on the market and the designer is free to chose amongst them. Examples of embedded operating systems running on hard or soft processors within the FPGA are MontaVista Linux [9], VxWorks [10], or QNX Neutrino [11]. Some OS are intentionally kept small and provide only very limited functionality. Their strongest offering is their real-time capability.

Other OS are rather large but offer more features. Linux currently falls into the second category. While efforts are under way to improve the real-time characteristics of embedded Linux it is clearly not its largest benefit. The strength of Linux, and with that of embedded Linux, is in the wealth of available libraries, middle-ware, and applications. For almost all possible problems and environments a solution already exists. Source code exists for most libraries and applications, which allows porting them to different processor architectures by a simple recompilation.

The openness of Linux also makes it an ideal research object to explore into new ideas or emerging technologies as SDR. Open source projects as GNU Radio use Linux as one of their building blocks [12].

When working with an operating system the designer needs to be careful on how he architects the system. For performance reasons he must make sure that the operating system influences the control path of an application and stays away from the data path. The FPGA fabric handles the high-bandwidth and low latency mainstream functionality of the application where the processor running the operating system influences the behavior of the system, processes user input and output, and handles exceptions caused by special circumstances during data processing.

Design Flexibility

A non-material but nonetheless important building block is design flexibility. The same framework can be used for different environments. You can integrate the intellectual property into the software or the FPGA fabric without changing the hardware around it. With that, you can support customer requirements with the same platform and make additional functionality available to them according to their needs or service plan.

Further, in-field upgrades allow adapting to changing and emerging standards even after a system has been deployed to the field. You simply upgrade the firmware of the system, either soft hardware or software, and easily make new features available.

As a logical consequence, production and shipment can start earlier when compared with conventional systems. In parallel with the production process additional functionality can be added and made available to the deployed systems at the same time when the product ships. While this is a well-known process for driver updates in PCs, FPGAs take it a step further and make the same process available for hardware functions and building blocks.

System Level View

We have long learnt to look at a problem from some distance at a system level point of view. However, very often we immediately start to draw a hard line of what we think needs to be implemented in hardware and/or software. In many cases we actually do not have a choice because traditionally available solutions dictate the approach we have to take. Once a decision has been made it is difficult to come back to it or even overturn it. The design process might already be on going for several months and changes at that time are not feasible for risk management reasons or economic aspects.

FPGAs allow for a different design approach in that they allow partitioning the system and building a framework that can be filled in with hardware or software as required.

Partitioning

The key to building the system is to take it apart. The system needs to be split into logical pieces. We introduced these building blocks earlier in this paper.

Typically, the data that we are interested in is encapsulated in different layers of abstraction. Let us take the example of an audio/video feed. The source can be a mobile news unit in the field reporting on a live event. The destination is an analog TV. The camera captures the event in analog or digital form and transforms it into an encapsulated format, for example an MPEG-2 transport stream. In a next step the transport stream is modulated and transmitted over satellite to the broadcast studio. After demodulation of the feed, high-end audio/video processing units decode the MPEG-2 transport stream, improve picture quality, add commentaries, captions or other additional information and re-encode into a new MPEG-2 transport stream. The signal is modulated and broadcasted over the air to the antenna of a home receiver. There the signal is demodulated, decoded, and finally displayed on the TV unit.

More steps can be necessary if data is encrypted for example for Pay-TV or for security relevant applications.

Each of these steps is a building block in the whole system. Ideally, these building blocks can be dynamically assembled as needed according to the current requirements.

Framework

The advantage of a software defined radio system is the capability to adapt to a varying environment. An FPGA based system is the ideal platform to implement such a system. Without programming the FPGA is a container with zero functionality but infinite potential. Only the programming information makes it solve a given problem. Naturally, the FPGA depends on external functionality like large memories, A/D converters and analog circuits where applicable.

To explore the potential of FPGAs in a SDR environment we decided to use existing boards and emulate missing functionality for a start. For example, instead of capturing real signals we decided to start off with an embedded function generator. Further, we decided to start at the sink and work our way backward to the source. The main reason for this approach lies in our team being more knowledgeable on video processing, i.e.

the sink of the system, than on the TV signal decoding, the source, allowing us to get to results in a shorter amount of time.

Without restricting the framework to a single application we decided to concentrate on using digital broadcast TV as a means of demonstrating the capabilities of an FPGA in a SDR environment. Digital broadcast TV is encapsulated in multiple layers of transport mechanisms as we have seen earlier. Figure 3 shows the current system under development.

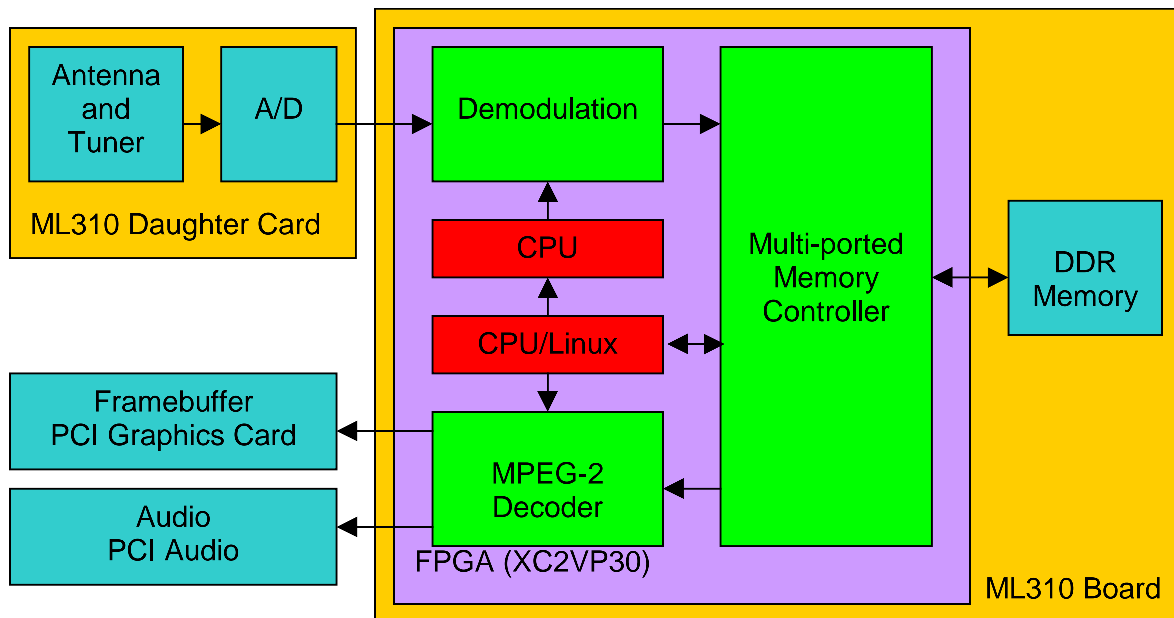

Figure 3: System for Broadcast TV Receiver

The system consists of three main components for signal input, processing and output.

A daughter card for the ML310 board contains the components needed for signal input. An antenna receives the broadcast video transmission and feeds it into a tuner that demodulates the signals to I/F at 5.75 MHz. An analog to digital converter samples the signals at 20MHz and 12 bits and forwards the data stream to the FPGA on the ML310 board.

The ML310 board is the processing unit for the SDR system [13]. Its main component is a powerful FPGA with two integrated PowerPC CPUs. We partitioned the FPGA design into two major parts.

A demodulation block extracts the MPEG-2 transport stream from the digitized data provided by the A/D converter and writes it through a multi-ported memory controller into the DDR memory on the ML310 board. One of the two CPUs supports the demodulation process.

The MPEG-2 decoder block supports the second CPU running Linux with the processing of the video and audio data. Most of it is implemented in software but the system allows improving performance by outsourcing functionality into the FPGA fabric. At this time we

use this capability to accelerate the calculation of the inverse discrete cosine transform (IDCT).

The video is displayed on a PCI graphics card plugged into one of the PCI slots of the ML310 board. The audio is played through the on-board AC97 audio controller.

We explore using partial reconfiguration in various places throughout the system; i.e. we imagine that all the building blocks are partially reconfigurable. With that the system becomes more flexible and adaptable to different standards in the real world. In a broadcast TV receiver similar building blocks require a different implementation:

- Countries use differing standards for terrestrial video broadcasting like ATSC 8-VSB, DVB-T COFDM, and ISDB-T BST-OFDM.
- Video signals formats can be PAL, NTSC, and/or HDTV.
- Audio formats are AC-3 and/or MPEG-2

Another example is the encryption and decryption of data streams. The mechanism can be replaced dynamically based on the current use case. The same system can be used in various environments. Video streams transmitted through broadcast networks and confidential corporate video conferencing can be encrypted with different modules. The framework used for the process does not change but the components do. The framework also allows for custom processing steps where the hardware plugins are proprietary.

Hardware Acceleration for 2D-IDCT

The 2D-IDCT transforms a block of 8 x 8 data points from the frequency domain into pixel information. A high-level diagram depicting the pixel decode by the APU controller, along with advantages, is shown in Figure 4. In this example, each data point has a resolution of 12 bits and is represented as a 16-bit integer value. The data structure is defined where each row of eight pixels consumes 16 bytes. This is an ideal size that allows optimal use of the FCM load and store instructions described earlier. In other words, eight FCM quadword load instructions are needed to load a data block into the 2D-IDCT hardware. Eight FCM quadword store instructions are sufficient to copy the pixel data back into the system memory.

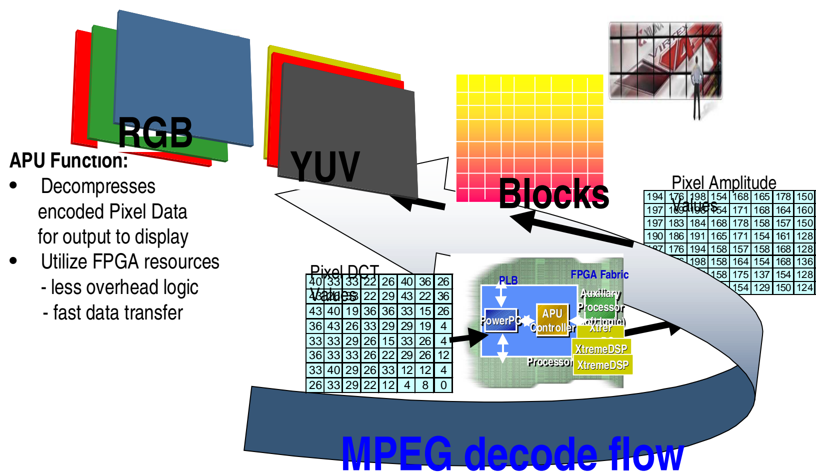

Figure 4: 2D-IDCT for MPEG-2 Video Decoding

The calculation of the 2D-IDCT in the FCM starts immediately after the first load, and the pixel data is available shortly after the last load operation. As shown in Figure 5, the 2D-IDCT makes uses of the DSP slices introduced as building blocks earlier in this paper.

A software-only implementation of a 2D-IDCT takes 11 multiplies and 29 additions together with a number of 32-bit load and store operations [14], while the hardware-accelerated version takes eight load and eight store operations. The reduced number of operations results in a speed-up of 20X in favor a 2D-IDCT FCM attached through the APU controller.

By comparison, if you connect the 2D-IDCT hardware block to the processor local bus, as it is done conventionally, the system performance will be degraded due to the increased, latency mainly caused by the bus arbitration overhead and the large number of 32-bit load and store instructions. This is illustrated schematically in Figure 5.

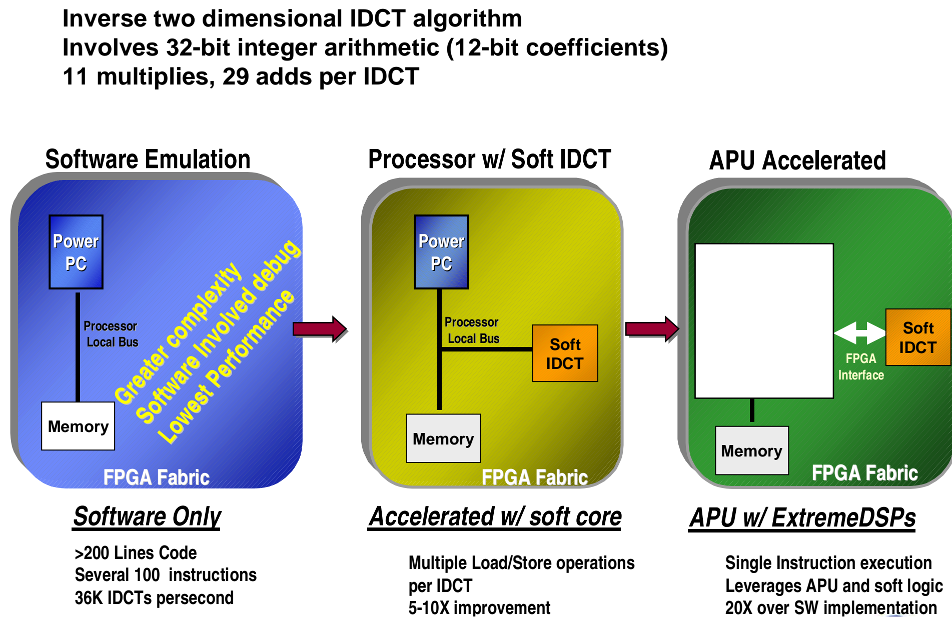

Figure 5: Comparison of Various 2D-IDCT Implementations

Status of Project and Future Work

So far we have designed the framework and identified the various building blocks. Further, we have implemented the video back-end system ready to use hardware acceleration to improve the performance when decoding the MPEG-2 encoded video stream. The hardware acceleration for the 2D-IDCT is implemented and proven out in simulation. Linux is up and running on the system and is capable of handling user input and decoding MPEG-2 transport streams.

As of the writing of this paper we are in the design process to build a new board that will integrate the video framebuffer into the FPGA and thus make the use of the PCI graphics card unnecessary. A daughter card for video input is also under development that we will use for the signal processing front-end.

So far we could identify various limitations that we will address as part of our future work on the project. For the MPEG-2 transport stream decoding we will focus even more on using FPGA fabric to gain a better speed-up. We believe that an extensive use of user-defined instructions and high-end fabric co-processor modules for functions like color-conversion and Huffman decoding will give us additional performance. These modules will make use of the DSP slices.

Conclusion

In this paper we have shown the framework for a SDR based system at the example of a digital TV broadcast receiver. We have shown that it is very well possible to implement a complete system consisting of the signal processing fronted and the MPEG-2 transport stream processing backend in a single FPGA based system.

We use modular building blocks like processors, operating systems, high-performance DSP slices, and partial reconfiguration to adapt the system to different environments and to keep it flexible with regards to changing and emerging standards.

References

[1] PicoBlaze 8-bit soft processor, Xilinx Inc., http://www.xilinx.com/picoblaze
[2] MicroBlaze 32-bit soft processor, Xilinx, Inc., http://www.xilinx.com/microblaze
[3] PowerPC FPGA Embedded 32-bit hard processor, Xilinx Inc., http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=v2p_powerpc
[4] Nios-II 32-bit soft processor, Altera, http://www.altera.com/nios
[5] OpenCores Microprocessors, Opencores, http://www.opencores.org/browse.cgi/filter/category_microprocessor
[6] Virtex-4 Performance Advantage, Xilinx Inc., 2004 http://www.xilinx.com/products/virtex4/overview/performance.htm
[7] High Performance Multi Port Memory Controller, Xilinx Application Note XAPP535, 2004, http://www.xilinx.com/bvdocs/appnotes/xapp535.pdf
[8] Virtex-4 Configuration Guide, Xilinx Inc, p. 81, 2004, http://direct.xilinx.com/bvdocs/userguides/ug071.pdf
[9] MontaVista Linux, http://www.mvista.com
[10] QNX Neutrino, http://www.qnx.com/products/rtos
[11] Wind River VxWorks, http://www.vxworks.com
[12] GNU Radio, http://www.gnuradio.org
[13] ML310 Board, Xilinx Inc, http://www.xilinx.com/ml310
[14] Inverse Two-Dimensional DCT, Chen-Wang Algorithm, IEEE ASSP-32, pp. 803-816, Aug. 1984