Xcell SOFTWARE Journal

SOFTWARE SOLUTIONS FOR
A PROGRAMMABLE WORLD

ISSUE 3
FIRST QUARTER 2016

Running Doom on the Zynq MPSoC

Speed SDR Development with Avnet PicoZED SDR's Automated Workflows

Linux/RTOS AMP Brings Best of Both Worlds for Embedded

Tap Embedded Community's Accumulated Knowledge to Build Better Systems

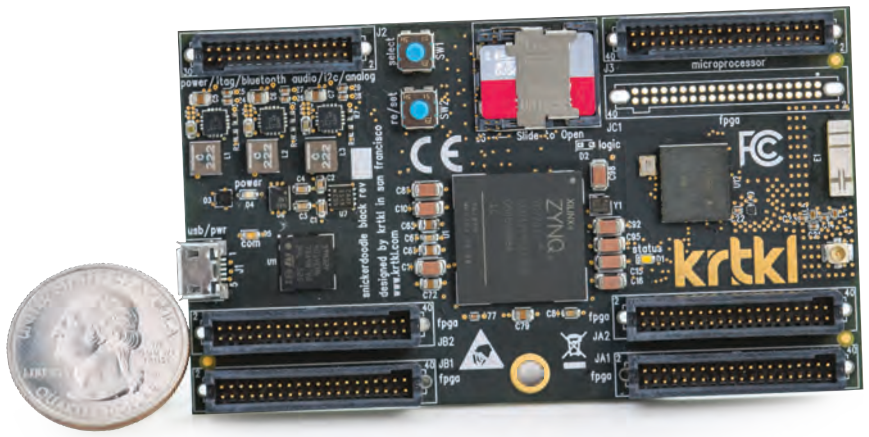
FPGA-based Infrastructure Poised to Bring Neuroplasticity to the Cloud

XILINX
ALL PROGRAMMABLE.

www.xilinx.com/xcell

snickerdoodle

create something different


krtkl
ZYNQ
snickerdoodle black rev
designed by krtkl in san francisco
www.krtkl.com
power/jtag/bluetooth audio/i2c/analog
microprocessor
fpga

Zynq®. Wi-Fi®. Bluetooth®. $55.
Any questions?

order yours today at
snickerdoodle.io

krtkl®

Xcell SOFTWARE journal

PUBLISHER	Mike Santarini
ACTING EDITOR IN CHIEF	Steve Leibson steve.leibson@xilinx.com 408-879-2716
EDITOR	Diana Scheben
ART DIRECTOR	Scott Blair
DESIGN/PRODUCTION	Teie, Gelwicks & Assoc. 1-408-842-2627
ADVERTISING SALES	Judy Gelwicks 1-408-842-2627 xcelladsales@aol.com
INTERNATIONAL	Melissa Zhang, Asia Pacific melissa.zhang@xilinx.com Christelle Moraga, Europe/Middle East/Africa christelle.moraga@xilinx.com Tomoko Suto, Japan tomoko@xilinx.com
REPRINT ORDERS	1-408-842-2627
EDITORIAL ADVISERS	Tomas Evensen Lawrence Getman Mark Jensen

Xilinx, Inc.
2100 Logic Drive
San Jose, CA 95124-3400
Phone: 408-559-7778
FAX: 408-879-4780
www.xilinx.com/xcell/

Xcell PUBLICATIONS

Letter from the Publisher

Xilinx Ecosystem of Development Environments Gives You Great Choices

Xilinx® has provided hardware designers with FPGA-based development tools for more than 30 years. In the past few years, it has also created the SDx™ line of tools—including the SDSoC™, SDAccel™ and SDNet™ development environments—that target developers unfamiliar with hardware description languages like Verilog and VHDL, allowing them to program Xilinx devices using software languages such as C/C++ and OpenCL™. Xilinx is not the only source for such tools; notable third parties delivering popular design environments that accommodate Xilinx devices include MathWorks® (MATLAB™ and Simulink™), National Instruments (LabVIEW and LabVIEW FPGA), and Topic Embedded Products (Dyplo).

I watched a demo of Topic's Dyplo for the Xilinx Zynq®-7000 SoC at this winter's Embedded World conference in Nuremberg, Germany, and came away impressed. Dyplo, which stands for Dynamic Process Loader, allows your program to swap custom-compiled hardware accelerator blocks in and out of the Zynq-7000 SoC's programmable logic programmable using the SoC's partial-reconfiguration capability. As I wrote in an Xcell Daily blog post, "This is a really neat trick." (See "Topic Embedded's Dyplo framework turns Zynq-7000 SoCs into multitasking hardware/sofware execution engines.")

If you have a preconceived notion of how you're "supposed" to develop code for Xilinx devices, you really should shake up those assumptions. Go and check out SDSoC and the other Xilinx SDx development environments and, by all means, check out the offerings from MathWorks, National Instruments and Topic Embedded Products.

Kudos and Goodbye to Mike Santarini

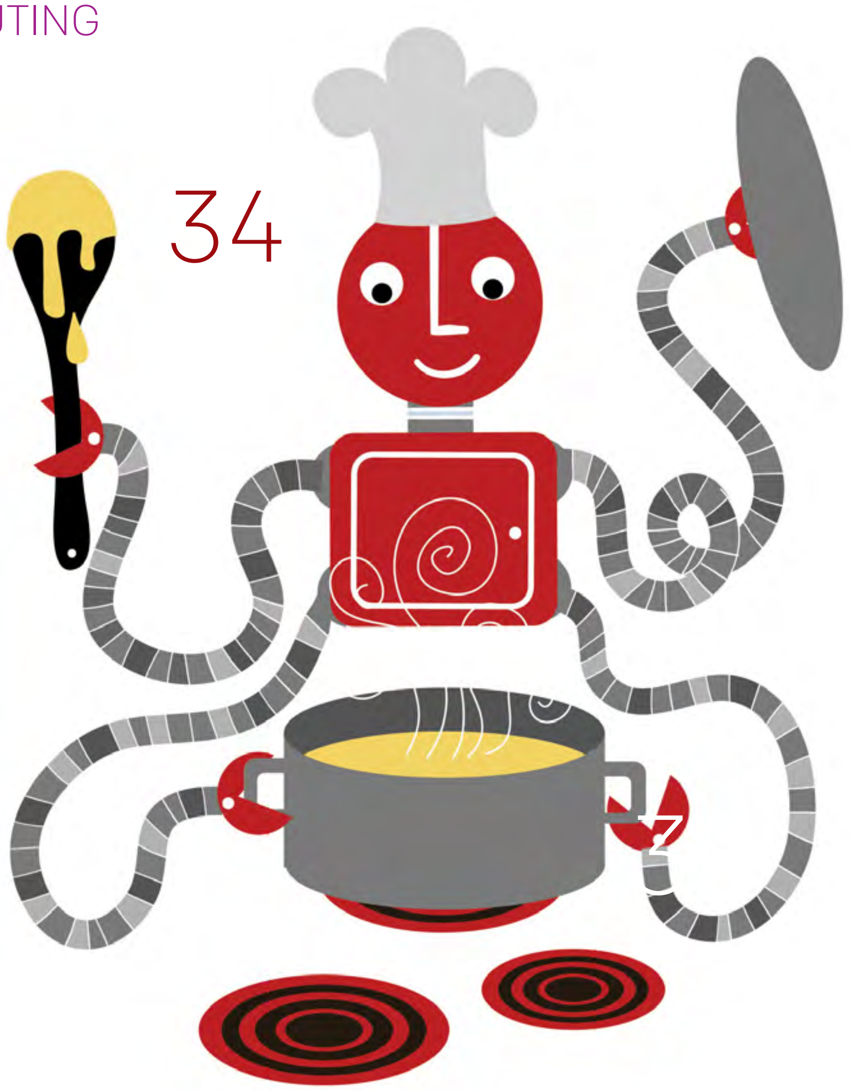
Xcell Software Journal and Xilinx owe a great debt to Mike Santarini, who served as the publisher of *Xcell Journal* for the past eight years and who started this magazine as well. Mike has moved on to other things, but the contribution he made was significant. Mike, all of us here at Xilinx say "Thank you!" and wish you good luck in your future endeavors.

CONTENTS

FIRST QUARTER
2016
ISSUE 3

28

40


34

Running Doom on the Zynq MPSoC

Use this fun tutorial to become familiar with the Xen hypervisor running on Xilinx's Zynq UltraScale+ MPSoC.

by Zach Pfeffer
Director of Embedded Software Development
Xilinx, Inc.
zachp@xilinx.com

Edgar Iglesias
Sr. Staff Software Engineer
Xilinx, Inc.
edgari@xilinx.com

Alistair Francis
Software Engineer
Xilinx, Inc.
alistai@xilinx.com

Nathalie Chan King Choy
Staff Software Engineer
Xilinx, Inc.
nathalie@xilinx.com

Rob Armstrong Jr.
Embedded Specialist Field Application Engineer
Xilinx, Inc.
ra@xilinx.com

W

When the System Software team at Xilinx® and DornerWorks brought up the Xen Project hypervisor on Xilinx's Zynq® Ultrascale+™ MPSoC, we found that we could run the popular 1993 videogame Doom to demonstrate the system and test it. The visually striking game allowed the team to visit Xen engineering topics with the aim of passing on knowledge and experience to future hypervisor users.

Our team used an emulation model of the Zynq UltraScale+ MPSoC available for QEMU (the open-source Quick Emulator) to prepare the software for the Doom demonstration, enabling us to bring it up in hours, not days, when silicon arrived.

Before we detail the steps of how to run Doom on Xen on top of QEMU for the Zynq UltraScale+ MPSoC, let's review what hypervisors are and how they work in relation to the processors on the Zynq UltraScale+ MPSoC.

HYPERVISORS AND HOW THEY WORK

A hypervisor is a computer program that virtualizes processors. Applications and operating systems running on the virtualized processors appear to own the system completely, but in fact the hypervisor manages the virtual processors' access to the physical machine resources, such as memory and processing cores. Hypervisors are popular because they provide design compartmentalization and isolation between the independent software elements running on the system.

To support virtualization, the physical processor must provide a special "mode" in which the hypervisor will run. Thus, describing a processor mode is a useful place to start in understanding how a hypervisor accomplishes this processor magic trick.

All processors have instructions that operate on values stored in registers and can read and write memory. A processor's mode is a collection of its instructions and registers, along with rules for using those instructions to access registers and memory. For this explanation, we will describe a generic processor as an example and will use architecture-agnostic terminology.

In our example, the processor has specific registers, instructions and modes. Registers include RegisterA, RegisterB, RegisterC, UserProgramCounter, RegisterSuper and SuperProgramCounter. Instructions include the following.

ADD Register3 Register1 Register2 adds Register1 to Register2 and puts the result in Register3, i.e., Register3 = Register1 + Register2.

MOVTO Register2 Register1 moves the contents of the memory at the address in Register1 to Register2.

MOVFROM Register2 Register1 moves the contents of Register1 to the memory at the address in Register2.

ENTERSUPER enters the SUPER mode of the processor.

EXITSUPER leaves SUPER mode and enters USER mode.

In USER mode, the processor's instructions are limited in what they can do. In our example, the instructions can read and write (operate on) all of the registers except for RegisterSuper and SuperProgramCounter, and the processor can execute all of the instructions except EXITSUPER. In addition, in USER mode all of the instructions can only read and write a subset of memory—for example, from address 0x0000_0100 to 0x0FFF_FFFF. In USER mode, if a program tries to execute an instruction it shouldn't or access a register or memory location to which it doesn't have access, the processor will halt on the offending instruction.

In SUPER mode, the processor's instructions can read and write all of the registers indicated above, including RegisterSuper and SuperProgramCounter. All of the instructions listed above, including EXITSUPER, can execute, as can an additional instruction, ENTERHYPER (more on that instruction later). Further, in SUPER mode the instructions can access all memory (from 0x0000_0000 to 0x7FFF_FFFF) in our system.

Having a processor with modes lets us use design compartmentalization to simplify solving software engineering problems. In the example above, there is only one way to enter SUPER mode: Execute the ENTERSUPER instruction. Likewise, there is only one way to leave SUPER mode: Execute EXITSUPER. In addition, programs can access only a subset of the machine's memory while in USER mode. With this scheme, we could write a program that would allow a processor to run multiple USER mode programs at the same time. This "operating system" (OS) program would run in SUPER mode and manage programs running in USER mode.

When the OS runs, it would look at all the USER mode programs that it needs to run, pick one to run and then instruct the processor to switch into USER mode to run it with an instruction such as EXITSUPER. The selected program would run until an event caused the processor to switch back into SUPER mode. Such an event could be an ENTERSUPER instruction from the USER mode program or an external event, such as a timer that would switch the processor into SUPER mode without alerting the program that was running in USER mode. Regardless of how the switch happens, we could construct the OS to select and run USER

HYPER mode is useful because it allows many SUPER programs to run. Each of the programs in SUPER mode could be an OS; those OSes themselves would allow many USER programs to run in parallel.

programs according to some policy, one after another, each time the event occurs. When the switch happens quickly, the user perceives USER programs to be running at the same time.

The SUPER processor mode also prevents USER programs from interfering with the programs running in SUPER mode or other USER mode programs. Any errors or misbehavior on the part of a USER mode program can be contained to just its own instance, and not corrupt or interfere with the system memory and registers reserved for SUPER mode operation.

Sounds good—but can we gain something with another mode?

Expanding our machine a bit, we can introduce HYPER mode. HYPER mode can read and write all of the original registers (RegisterA, RegisterB, RegisterC, UserProgramCounter, RegisterSuper and SuperProgramCounter) as well as two additional registers: RegisterHyper and HyperProgramCounter. The instructions in HYPER mode include the original set and the italicized additions below.

ADD Register3 Register1 Register2 adds Register1 to Register2 and puts the result in Register3, i.e., Register3 = Register1 + Register2.

MOVTO Register2 Register1 moves the contents of the memory at the address in Register1 to Register2.

MOVFROM Register2 Register1 moves the contents of Register1 to the memory at the address in Register2.

MOVTOPHYS Register2 Register1 moves the contents of the memory at the physical address in Register1 to Register2.

MOVFROMPHYS Register2 Register1 moves the contents of Register1 to the physical memory at the address in Register2.

ENTERSUPER enters the SUPER mode of the processor.

EXITSUPER leaves SUPER mode and enters USER mode.

ENTERHYPER enters the HYPER mode of the processor.

EXITHYPER exits the HYPER mode of the processor.

SWITCHSUPER RegisterHyper switches to the SUPER program that will execute next using the value in RegisterHyper.

The additional instructions and registers in HYPER mode allow the processor to switch which program is running in SUPER mode, just as the SUPER mode allows the processor to switch which program is running in USER mode. One feature of HYPER mode is the ability to switch which memory SUPER modes see; when a program running in HYPER mode executes SWITCHSUPER RegisterHyper, the underlying memory completely switches out. This means that when the next SUPER program runs after the program in HYPER mode executes EXITHYPER, the actual physical memory that the SUPER mode sees will differ from the physical memory used by another program running in SUPER mode. The SUPER mode program will still access the memory using the same address, but that address will point to a different physical location. Figure 1 shows the processor's view of memory before and after it executes SWITCHSUPER RegisterHyper.

HYPER mode is useful because it allows many SUPER programs to run. Each of the programs in SUPER mode could be an OS; those OSes themselves would allow many USER programs to run in parallel. This would mean, for example, that we could run multiple OSes, such as Windows and Linux, on the same hardware; 20 instances of Linux on one processor; or any combination in between. Since each instance of a virtualized OS cannot see the other OS instances, if one crashes, it doesn't crash the other instances. The features of HYPER mode have other applications: We can partition system resources

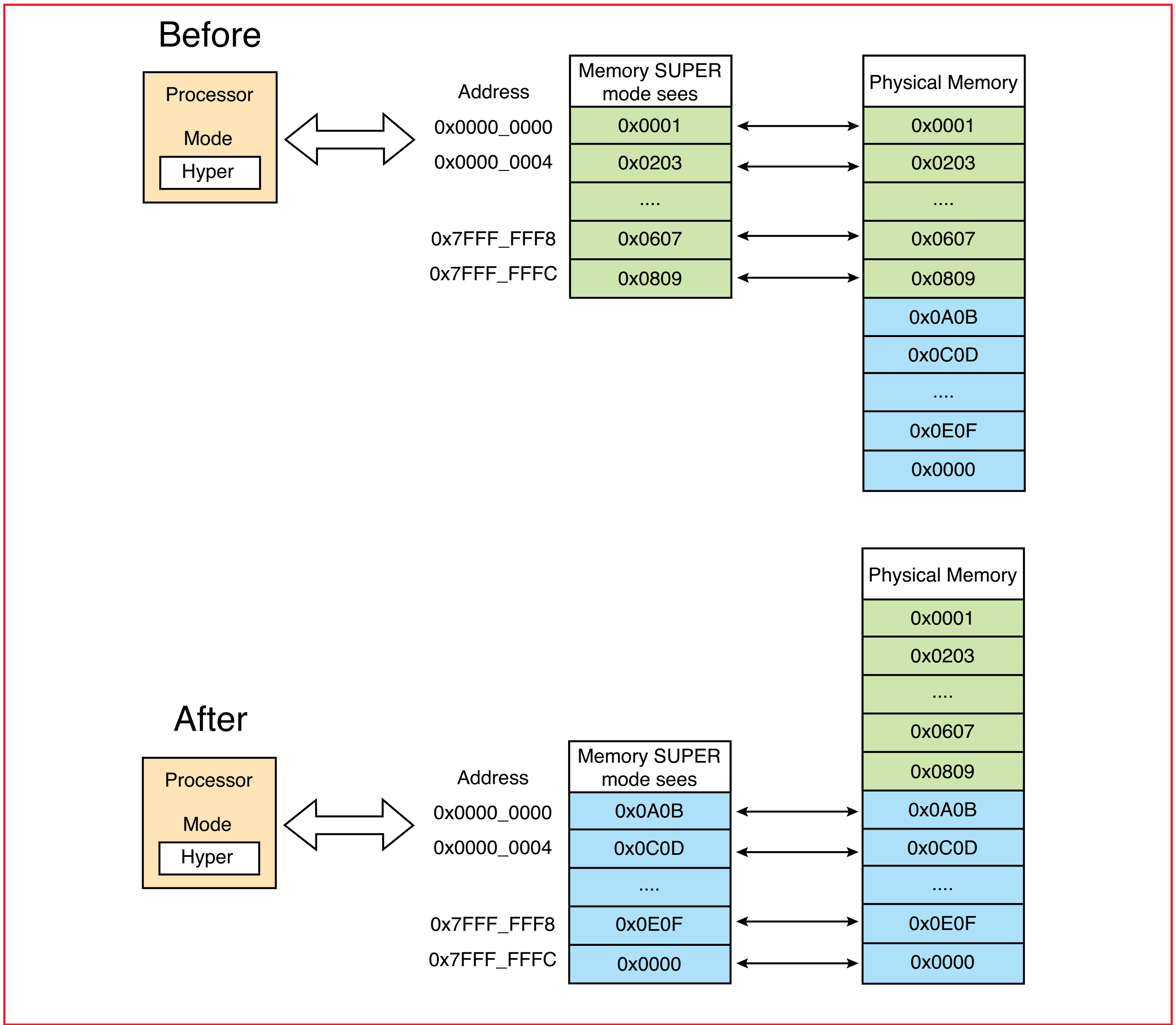


Figure 1 — Before and after execution of SWITCHSUPER RegisterHyper in HYPER mode

between multiple OSes, monitor the execution of each OS from the HYPER mode to restart it if it crashes, and keep tabs on system status while the virtualized OSes are running.

As a processor moves from USER to SUPER mode and then from SUPER to HYPER mode, the machine gives more privileges to the executing code. In our example, USER mode programs have the privilege to use only four registers (RegisterA, RegisterB, RegisterC and UserProgramCounter) and four instructions: (ADD, MOVTO, MOVFROM and ENTERSUPER). In addition, USER programs can only read and write memory from 0x0000_0100 to 0x0FFF_FFFF. Once in SUPER mode, the processor allows instructions to talk to RegisterSuper and SuperProgramCounter, and allows the execution of EXITSUPER and ENTERHYPER. In addition, SUPER programs can access memory from 0x0000_0000 to 0x7FFF_FFFF.

Finally, once the processor enters HYPER mode, its instructions can act on RegisterHyper and HyperProgramCounter, and programs can execute SWITCH-

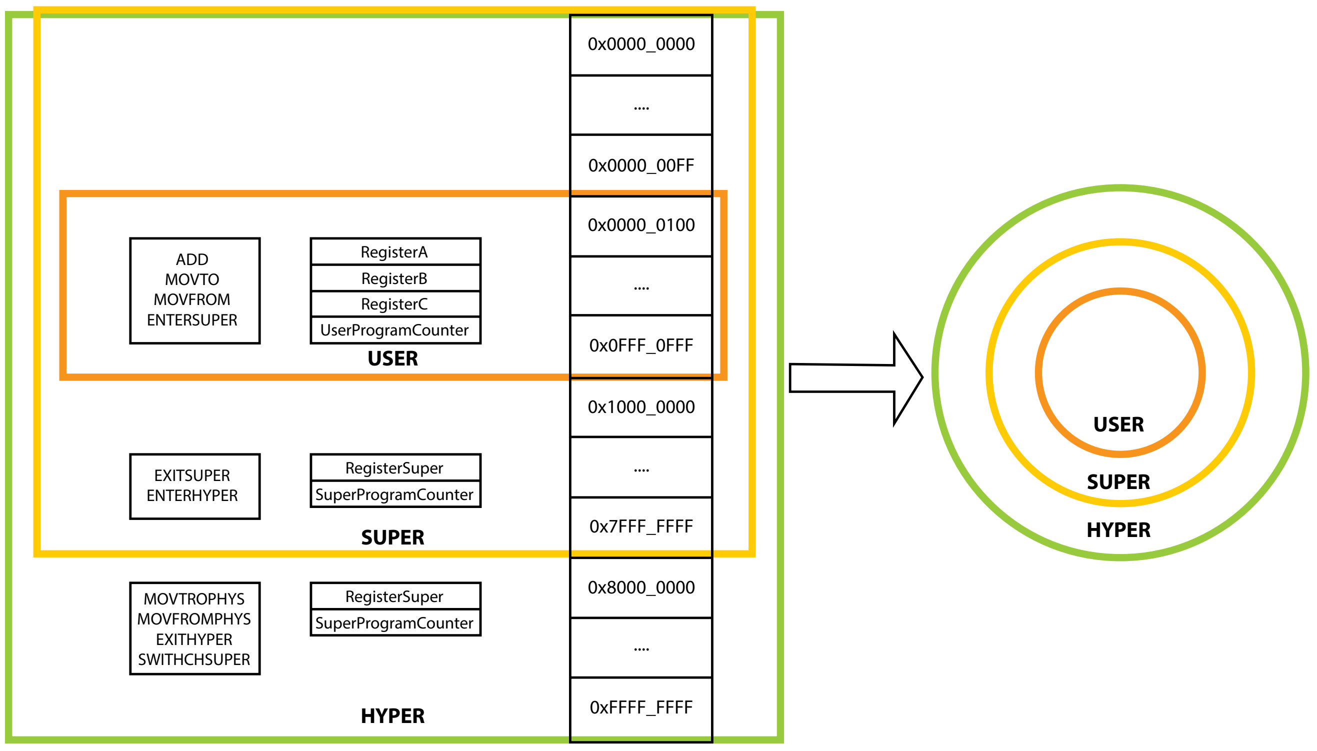


Figure 2 — Modes visualized as rings

SUPER and EXITHYPER. HYPER mode also allows the processor to read and write all virtual memory, 0x0000_0000 to 0xFFFF_FFFF, and to read and write the actual physical memory. These levels of privilege are typically visualized as rings (Figure 2). The master, HYPER ring grants permissions to the lower rings and ultimately can control the whole system.

MAPPING THEORY TO PRACTICE

ARM® creates processor designs that ARM partners can use to build chips. An ARM processor contains one or more cores. Each core implements an ARM architecture.

For instance, the Zynq UltraScale+ MPSoC contains an ARM Cortex™-A53 processor complex with four physical ARMv8-A cores (Figure 3).

The distinction is important when looking at documentation and code for an ARM processor; to get a complete understanding of a "chip" with an ARM core, consult documentation on the architecture (for example, ARMv8-A) as well as the processor (for example, Cortex-A53).

There are four exception levels in the ARMv8 architecture (source: *ARM Architecture Reference Manual,* D1-1404):

1. Exception Level 0 (EL0), which executes without privilege;
2. Exception Level 1 (EL1), which executes an OS and anything else that executes privileged instructions;
3. Exception Level 2 (EL2), which allows the hardware to be virtualized; and
4. Exception Level 3 (EL3), which allows switching between secure and nonsecure processor states.

The following programs would typically run in these modes, as described in the *ARM Architecture Reference Manual* (D1–1404): EL0, applications; EL1, the OS kernel and associated functions that are typically described as

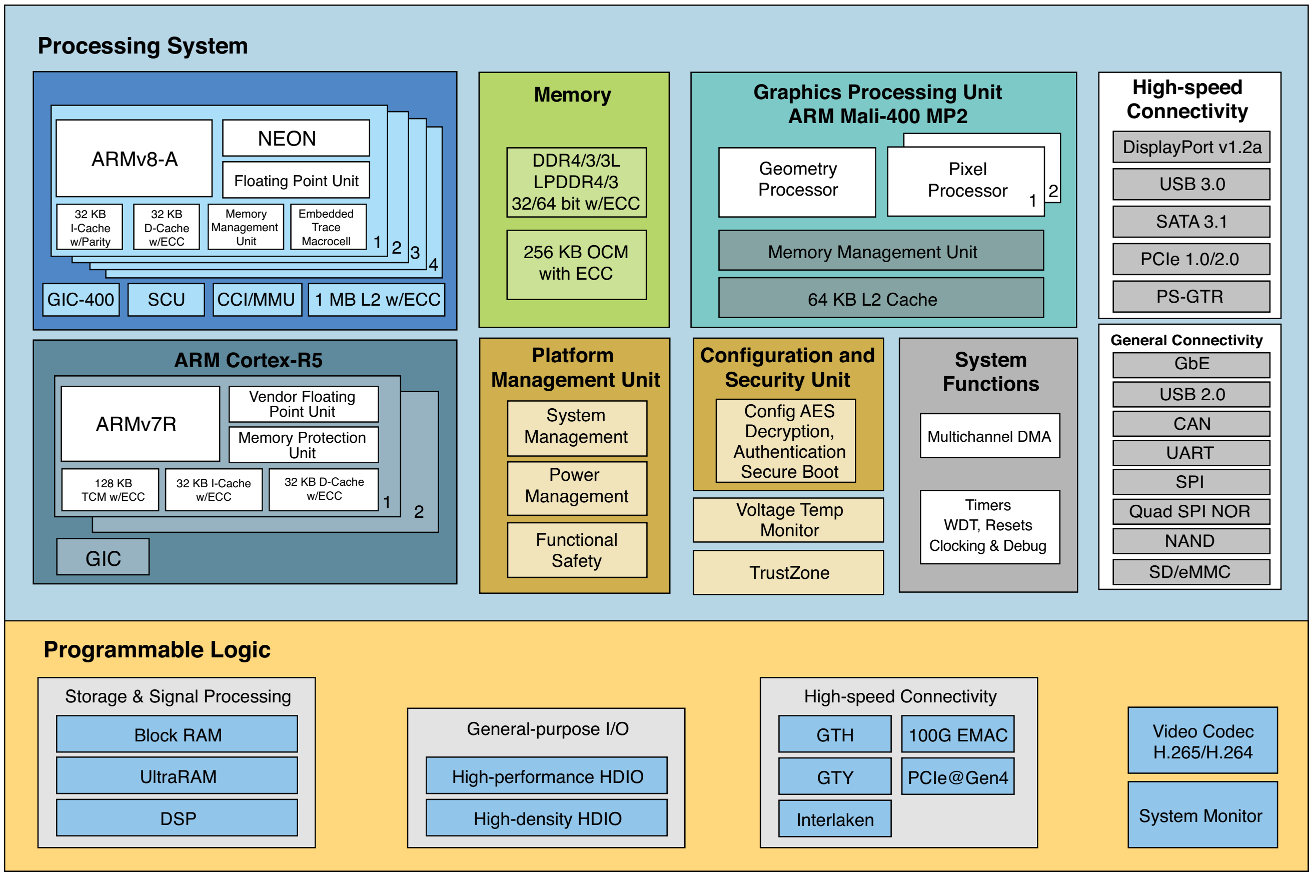


Figure 3 — Zynq UltraScale+ MPSoC

privileged; EL2, the hypervisor; and EL3, a secure monitor.

Our theoretical example maps directly onto ARMv8 execution modes EL0 to EL2: USER is EL0, SUPER is EL1 and HYPER is EL2. ARM adds a fourth privilege level, EL3, which we could use to switch EL0 and EL1 between operation in secure and nonsecure contexts. While the use of EL3 is an important topic that adds considerable capability to the architecture, for the purposes of this example we will ignore it and focus on EL0 to EL2 for virtualization with hypervisors. If you're curious about how a computer can secure a financial transaction, however, the ARMv8 EL3 documentation (free and open registration required) is a good place to get extremely specific details.

GETTING INTO AND OUT OF EXCEPTION MODES

In a real system, transitioning between modes is a bit more complex than in our example. ARM summarizes the behavior of the ARMv8-A architecture in the reference manual. It explains that execution can move between exception levels only upon taking an exception or returning from an exception. Upon taking an exception, the exception level can only increase or remain the same; upon returning from an exception, the exception level can only decrease or remain the same.

There are only three instructions that generate an exception targeting the next exception level: SVC (Supervisor Call), which generates an exception targeting EL1; HVC (Hypervisor Call), which generates an exception targeting EL2; and SMC (Secure Monitor Call), generating an exception targeting EL3. These instructions take values from 0 to 65,555, allowing for 2^{16} unique system calls per exception level. The instructions target the next exception level and are the only mechanisms by which a program running at a lower exception level can request something from a program running at a higher exception level. In our theoretical example, SVC is SWITCHSUPER and HVC is SWITCHHYPER.

The PetaLinux Tools contain a set of commands that allow users to create and extend Linux systems easily on Xilinx FPGAs and SoCs.

In the previous section, we talked about an event that would cause the program operating in USER mode (EL0) to enter SUPER mode (EL1). One event that most programs running in USER mode generate is a request for memory. When a user space program running in EL0 requests memory from an OS running in EL1, the C code for that user space program will likely call a function such as malloc(), which in turn will call mmap() or sbrk(), to request a pointer to available memory from the OS. On Linux in the ARMv8-A architecture, this will translate behind the scenes to an SVC system call. That system call will transition the processor to EL1, thus passing control back to the OS, which in turn will interpret the call and provide an appropriate response—in this case, a pointer to the requested memory region or an error indicating that no additional memory was available.

DEMO CREATION AND TOOLS

Now let's turn to the steps our team used to run Doom on the Zynq UltraScale+ QEMU Model. The steps show how to get and construct each component required to run the demo, how to run each component and in what order, and how to interact with the demonstration. After successfully completing this demo, you will have an environment on which to experiment with the Xen hypervisor running on an emulated Zynq UltraScale+ MPSoC. Porting this to Zynq UltraScale+ MPSoC silicon is left as an exercise for you to undertake.

To make this process easier, Xilinx has provided a base root file system, which spares users the time and hassle of building it themselves. All downloads required for this demo are available at: *www.wiki.xilinx.com/Doom+on+Xen+Demo*.

The demo starts with updating a precompiled root File System (rootFS) provided by Xilinx to include the required components. We then leverage Xilinx's PetaLinux Tools to run the demo. A rootFS contains most of the programs that run on a Linux system—specifically, a set of scripts to bring up the system, and the set of applications and libraries that implement the system. The two tools we use to extend the base rootFS in this demo are Buildroot and PetaLinux. We use Buildroot to build the Doom binaries for the base rootFS supplied by Xilinx, and we use PetaLinux to create the rest of the rootFS and boot the demo.

Buildroot

Buildroot is a simple build system for creating a rootFS for a Linux system. It uses a make menuconfig interface, a popular method that configures the Linux kernel itself. Buildroot includes default support for PrBoom, which helps with this demo. (PrBoom is the GNU General Public License [GPL] version of the Doom game we are using. We will use the terms PrBoom and Doom interchangeably here.) Buildroot doesn't have native support for building Xen (though it does create all the necessary libraries and toolchains to build Xen), so Xilinx has provided Xen, Xen Tools and the Xen libraries precompiled for users, as well as some other required libraries to make the process straightforward.

PetaLinux

The PetaLinux Tools contain a set of commands that allow users to create and extend Linux systems easily on Xilinx FPGAs and SoCs. This demo leverages the petalinux-build and petalinux-boot commands. The petalinux-build command creates all of the required components. The petalinux-boot command (plus a few arguments) starts all of the components running on the QEMU emulator. Descriptions of all of the commands in the PetaLinux Tools is beyond the scope of this article, but with the demo system it should be easy to explore what these and other commands can do. Consult *PetaLinux Tools Documentation — Reference Guide* UG1144 (v2015.4) for more information.

Project prerequisites

This project requires a workstation or virtual machine running Linux with an environment meeting the PetaLinux Tools Installation Requirements outlined in UG1144 (v2015.4) and with Xilinx PetaLinux Tools v2015.4 installed in this environment.

Once you see Doom start up, you should be able to control the game using your keyboard and mouse. Remember that you might need to hit the escape key to start a game.

STEP 1: BUILDING THE ROOTFS

First, we have to build the rootFS. Download the doom_demo.tar.gz from Xilinx and open a terminal in the download directory; you can find all required files at *www.wiki.xilinx.com/Doom+on+Xen+Demo*. We will call that directory the <down_dir>.

We unpack the archive.

```
$ cd <down_dir>
```

```
$ tar -xzf doom_demo.tar.gz && cd doom_demo
```

We will see one folder, which we will build into our root file systems (one for Dom0 and the other for DomU). Now we need to build PrBoom and copy it into the rootFS.

First we need to download the Linux kernel so that we can build the rootFS later. We are using the v4.3 tag.

```
$ git clone -b v4.3 https://github.com/tor-
valds/linux.git
```

We download the Buildroot source and change to the Buildroot directory.

```
$ git clone https://git.buildroot.net/buildroot
&& cd buildroot
```

Now we need to configure Buildroot to build packages we can use.

```
$ make menuconfig
```

We select the following options:

Target options ---> Target Architecture ---> AArch64 (little endian)

Target packages ---> Games ---> prboom ---> [*]

Target packages ---> Games ---> shareware Doom WAD file ---> [*]

All required libraries should automatically be selected.

`$ make #` (This could take a few minutes, depending on your machine.)

Now we copy all the PrBoom related files into the targetfs directory, making sure we are in the ./output/target/ directory under the buildroot directory.

```
$ for i in $(find ./ -name '*oom*'); do cp ${i}
<down_dir>/doom_demo/targetfs/${i}; done
```

We have now finished with Buildroot. We go up one directory to the doom_demo directory.

`$ make # Build the host and guest rootFS.` (This could take a few minutes, depending on your machine.)

Note: Depending on which kernel version you use, there might be extra config options that are not preselected by our supplied config. You should be fine using the default options (just press enter).

STEP 2: BUILDING THE BASE SETUP

Next we will build the rest of the embedded system software for the platform, including the boot loader, ARM Trusted Firmware (ATF), Linux kernel and device trees. Xilinx's PetaLinux Tools make this process straightforward. We create a PetaLinux project targeting the Xilinx ZCU102 board. Reference quick-start material for QEMU and PetaLinux for MPSoC in 2015.4 UG1144 and AR#66249. Go to *www.xilinx.com* and download the ZCU102 BSP (Board Support Package) to a location that we will refer to as <petalinux_bsp_dir>.

```
$ cd <down_dir>
$ petalinux-create --type project -s <petali-
nux_bsp_dir>/ Xilinx-ZCU102-v2015.4-final.bsp
--name doom_demo_zynqMP
```

This will create our PetaLinux project in <down_dir>/doom_demo_zynqMP.

We go to the PetaLinux project and build PetaLinux.

```
$ cd <down_dir>/doom_demo_zynqMP
```

```
$ petalinux-build
```

Now we need to edit the device tree manually for our use case.

Edit the xen-overlay.dtsi file (subsystems/linux/configs/device-tree/xen-overlay.dtsi).

Replace

'reg = <0x0 0x80000 0x3100000>;'

under dom0 with

'reg = <0x0 0x80000 0x4100000>;'

Replace

'xen,xen-bootargs = "console=dtuart dtuart=serial0 dom0_mem=512M bootscrub=0 maxcpus=1 time r_slop=0";'

under chosen with

'xen,xen-bootargs = "console=dtuart dtuart=serial0 dom0_mem=512M bootscrub=0 maxcpus=4 timer_slop=0";'

Replace

'xen,dom0-bootargs = "console=hvc0 earlycon=xen earlyprintk=xen maxcpus=1";'

under chosen with

'xen,dom0-bootargs = "rdinit=/bin/sh console=hvc0 earlycon=xen earlyprintk=xen maxcpus=4";'

Edit the zynqmp.dtsi file (subsystems/linux/configs/device-tree/zynqmp.dtsi).

Replace

'compatible = "cdns,uart-r1p12";'

under uart0 with

'compatible = "cdns,uart-r1p8", "cdns,uart-r1p12";'

Now manually build the Xen device tree.

```
$ dtc -I dts -O dtb -i ./subsystems/linux/con-
figs/device-tree/ -o ./images/linux/xen.dtb ./
subsystems/linux/configs/device-tree/xen.dts
```

Finally, we need to replace the rootFS built by PetaLinux with the one we built before. This is required because PetaLinux doesn't include PrBoom, so we are supplying our own rootFS. We also need to replace the xen.ub image with one prebuilt by Xilinx, as the Xen and Xen tool versions must match.

```
$ rm <down_dir>/doom_demo_zynqMP/images/linux/
Image && rm <down_dir>/doom_demo_zynqMP/images/
linux/xen.ub

$ cp <down_dir>/doom_demo/Image <down_dir>/doom_
demo_zynqMP/images/linux/Image && cp <down_dir>/
doom_demo/xen.ub <down_dir>/doom_demo_zynqMP/im-
ages/linux/xen.ub
```

Boot using u-boot bootloader.

```
$ petalinux-boot --qemu --u-boot --qemuargs="-
net nic -net nic -net nic -net nic -net us-
er,net=192.168.129.0,dhcpstart=192.16
8.129.50,host=192.168.129.1,hostfwd=t
cp:127.0.0.1:5900-192.168.129.50:5900"
```

```
> setenv serverip 192.168.129.1
> tftpb 4000000 xen.dtb; tftpb 0x80000 Image; tftpb 6000000 xen.ub; bootm 6000000 - 4000000
```

```
# /boot.sh
# /xen-doom.sh 1
```

STEP 3: FIRING IT UP

Now we can fire up a virtual network computing (VNC) viewer and, on the machine running QEMU, connect to localhost:5900 to see the Doom game. (Note: The command line above will only redirect port 5900, which will only allow you to connect to the first instance of Doom when you fire up your demo. If you would like to connect to multiple instances, add more hostfwd arguments to QEMU and connect to the next available port [5901 for the next instance, 5902 for the one after that and so on], and then connect to those instances.)

Once you see Doom start up, you should be able to control the game using your keyboard and mouse. Remember that you might need to hit the escape key to start a game. Also remember that it's been a while

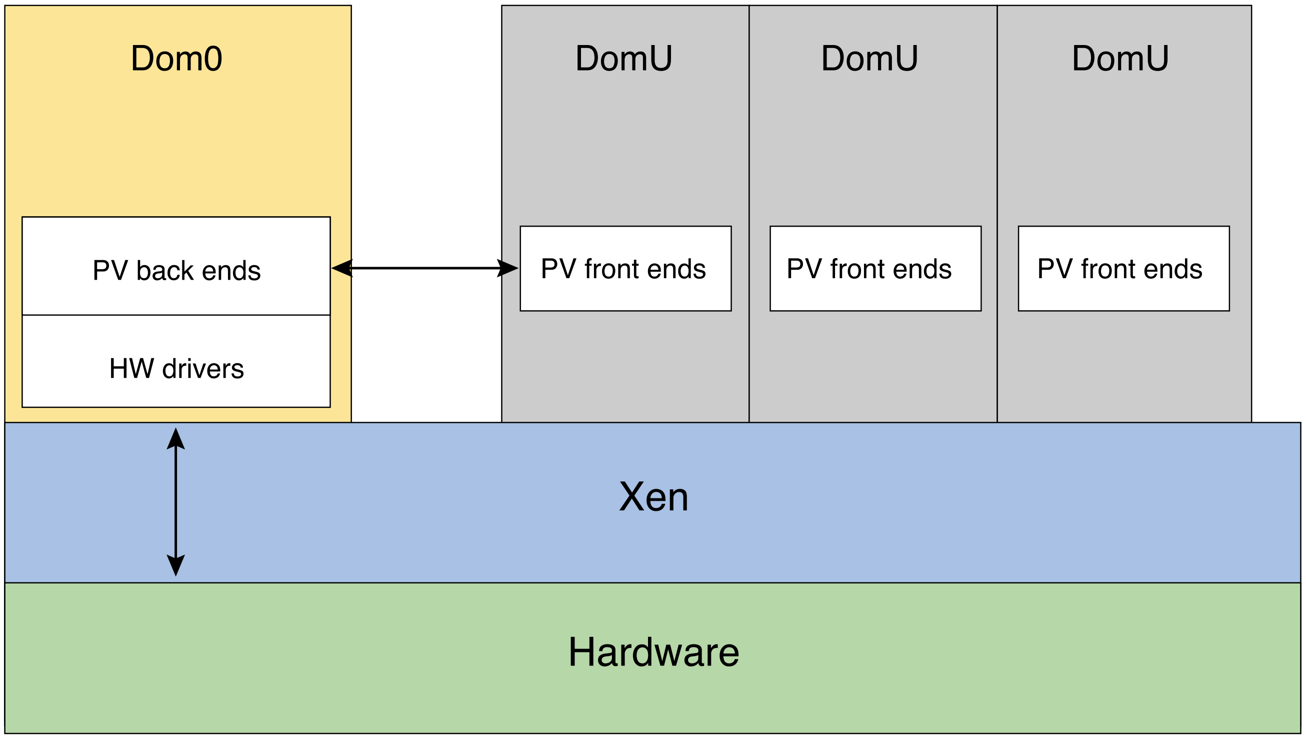


Figure 4 — As a Type 1 hypervisor, Xen runs natively on hardware, and virtual machines run on top of Xen (source: "Xen ARM with Virtualization Extensions" white paper).

since you've played Doom, so you may not make it very far. Don't feel discouraged. Working with the system you built is definitely "work."

XEN DEEPER DIVE

As described in "Zynq MPSoC Gets Xen Hypervisor Support" (*Xcell Journal*, Issue 93), a Type 1 hypervisor runs natively on the hardware, whereas a Type 2 hypervisor is not the lowest layer of software and gets hosted on an OS. Xen is a Type 1 hypervisor (Figure 4).

Earlier, we mentioned virtual processors (also known as virtual machines). In Xen, these are referred to as domains. The most privileged domain is called Dom0; the unprivileged guest domains are DomU domains.

Dom0 is the initial domain that the Xen hypervisor creates upon booting. It is privileged and drives the devices on the platform. Xen virtualizes CPUs, memory, interrupts and timers, providing virtual machines with one or more virtual CPUs, a portion of the memory of the system, a virtual interrupt controller and a virtual timer. Unless configured otherwise, Dom0 will get direct access to all devices and drive them. Dom0 also runs a set of drivers called paravirtualized (PV) back ends to give the unprivileged virtual machines access to disk, network and so on. Xen provides all the tools for discovery and initial communication setup. The OS running as DomU gets access to a set of generic virtual devices by running the corresponding PV front-end drivers. A single back end can service multiple front ends, depending on how many DomUs there are. A pair of PV drivers exists for all of the most common device classes (disk, network, console, frame buffer, mouse, keyboard, etc.). The PV drivers usually live in the OS kernel, i.e., Linux. A few PV back ends can also run in user space, usually in QEMU. The front ends connect to the back ends using a simple ring protocol over a shared page in memory.

Interacting with the hypervisor from Dom0 requires programs that use the defined hypervisor calls (similar to system calls). Xen provides a reference toolset with libraries called Xen Tools (also written as xen-tools). The xen-tools include a program called xl that, among other things, can inspect state and create guests.

With device paravirtualization, there is agreement between the hypervisor and the guest on how communications will progress. Popular communication protocols are Xen Bus and VirtIO.

The "create" command in xl takes a configuration file describing a guest, and if the configuration file specifies that the guests want a virtual frame buffer (VFB) backed by a VNC session, xl will automatically launch virtualization code in the Dom0 user space (one per guest in our demo).

The configuration file for the doom VM looks like this:

```
# Guest name

name = "guest1"

# Kernel image to boot

kernel = "/boot/Image"

# Kernel command line options

extra = "console=hvc0 rdinit=/doom.sh"

# Initial memory allocation (MB)

memory = 56

# Number of VCPUS

vcpus = 1

vfb = [ 'type=vnc, vnclisten=0.0.0.0' ]
```

DEVICES IN XEN

There are three common ways to expose a device to a guest: emulation, paravirtualization and pass-through (Figure 5).

With device emulation, when a guest writes to the memory of the emulated device, the write triggers a trap. The trap is typically a page fault. The trap allows the processor to switch into the hypervisor, which emulates the device. Emulation is flexible but slow because of all the traps, and someone must write models for all of the devices that require emulation. It's also hard to find tricks to speed emulation because there's little to no hardware acceleration; it's an all-software approach.

With device paravirtualization, there is agreement between the hypervisor and the guest on how communications will progress. There is typically a shared-memory area (plus protocol) that looks like a device, and the hypervisor services requests on this area. For example, to support a paravirtualized frame buffer on Linux, a Linux front-end driver would write the buffer of the frame it got from user space into a shared memory area; it would then signal the hypervisor using hypervisor calls to output the frame via a back-end driver. The guest can only talk to the host (Dom0) and other guests (DomU) through paravirtualized drivers. Some benefits of this approach are that you can share devices among many guests, it runs fast, and a guest can run a mostly unchanged kernel. The changes required are under standard interfaces, so to the applications and the rest of the kernel the front-end driver just looks like a normal network interface,

	Speed	Device Sharing	Security
Emulated	Slow	Yes	Yes
Paravirtualized	Medium	Yes	Yes
Pass-through	Fast	No	Yes (but only if you have an SMMU*)

*System Memory Management Unit

Figure 5 — Comparison of emulated, paravirtualized and pass-through approaches

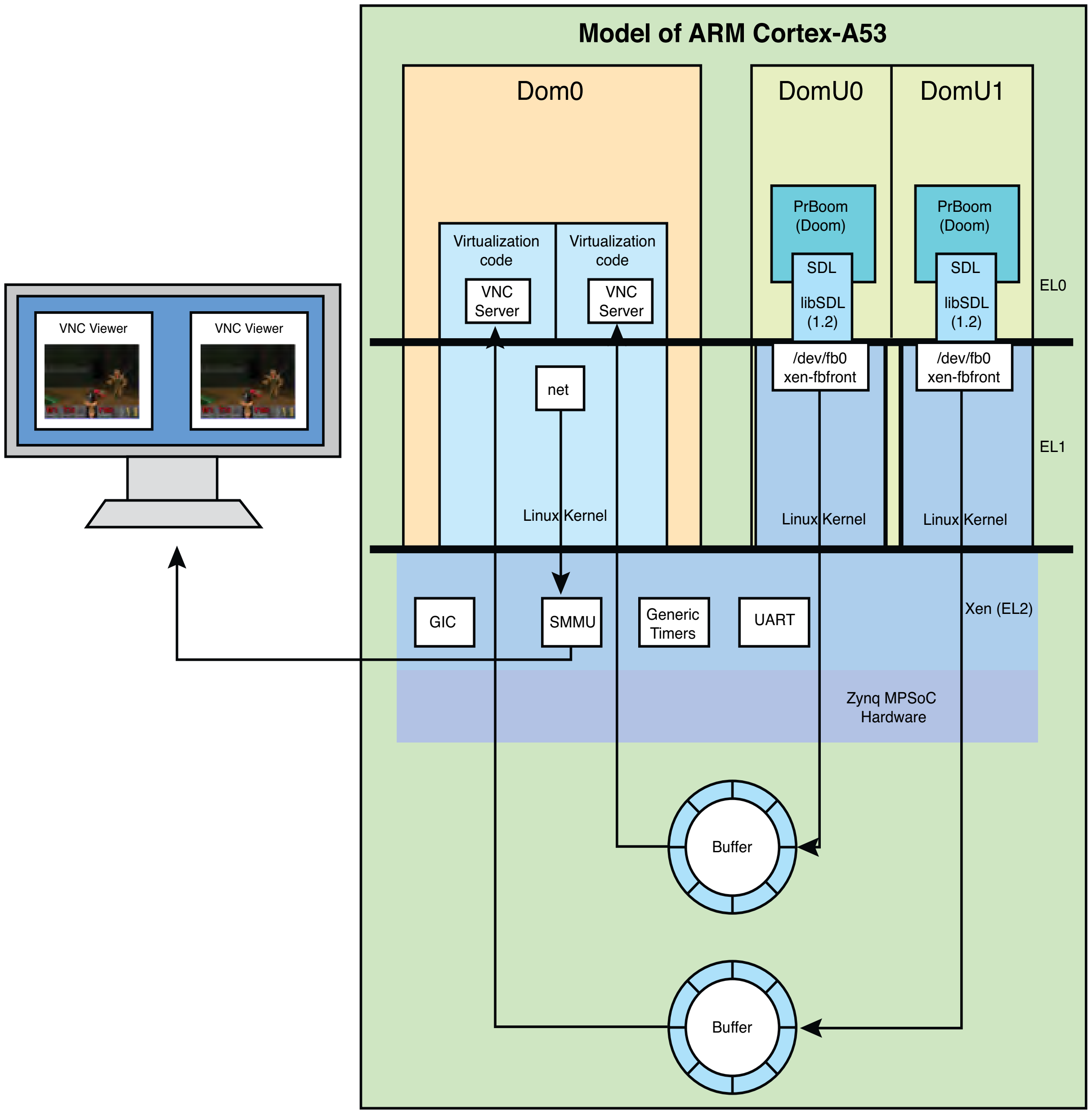


Figure 6 – Under the hood of the demo: QEMU launched from PetaLinux Tools running on an x86

disk or other device. Two popular protocols for supporting guest communication are Xen Bus and VirtIO.

In pass-through mode, the host "gives" a device to the guest. This means only one guest can use the device at a time.

DEVICE PERFORMANCE AND SECURITY

In general, emulated devices have lower performance than devices exposed via pass-through, and the paravirtualization approach tends to have sufficient performance. A benefit of the paravirtualized and emulation approaches is that the hypervisor can allow device access to multiple entities without exposing those entities to one another.

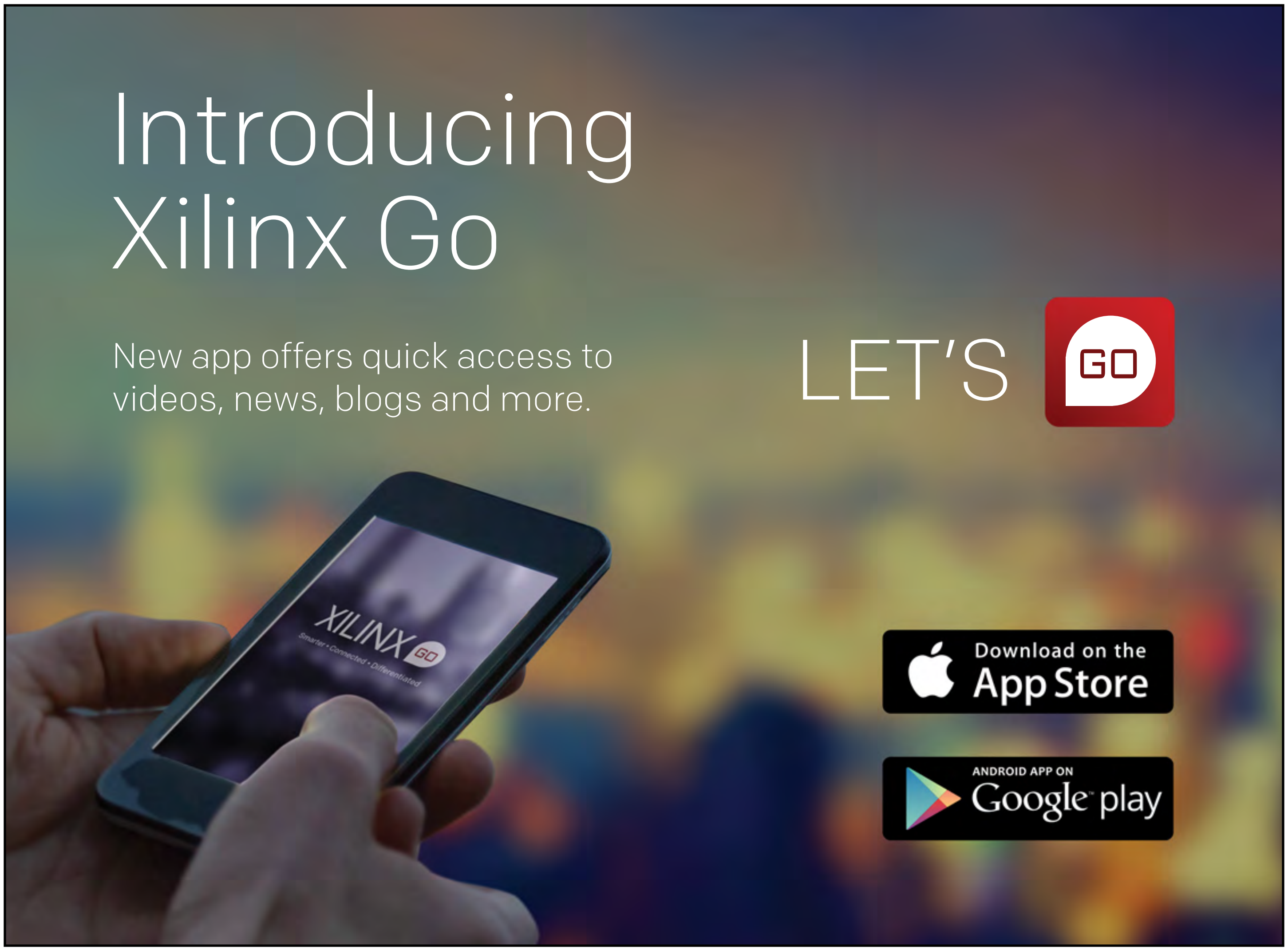
UNDER THE HOOD

The processing contexts of the Doom-on-Zynq UltraScale+ MPSoC are like an onion, with many layers (Figure 6). In the Cortex-A53 cluster are the four ARMv8 cores. On each core, the hypervisor runs in

EL2, and the guests (Dom0 or DomU) run in EL0/EL1. Each DomU guest runs Linux; Doom (PrBoom) runs in the user space. Doom uses the Simple Direct Media Layer (SDL), which talks to a frame buffer front-end driver via the SVC instruction (eventually). The frame buffer front end writes the buffer into a shared memory area set up by Dom0. The front-end driver communicates with virtualization code running on Dom0 via a protocol such as Xen Bus or VirtIO using the HVC instruction (eventually). The virtualization code running on Dom0 provides a back end for display which then is encoded by the virtualization code's VNC server and sent over a network to a VNC client.

This information and the demo should provide a good foundation for further hypervisor study and experimentation. After you are able to run the demo in emulation on QEMU, you can use PetaLinux Tools to run it on Zynq UltraScale+ MPSoC silicon.

For more great developer resources, visit Xilinx's Software Developer Zone. ■


Introducing Xilinx Go
New app offers quick access to videos, news, blogs and more.
LET'S GO
XILINX GO
Smarter • Connected • Differentiated
Download on the App Store
ANDROID APP ON Google play

Software-Defined Radio from Concept to Deployment

Use Avnet PicoZed SDR's automated workflows to cut development time and differentiate your design.

by Robin Getz
Director of Engineering, Global Alliances Group
Analog Devices, Inc.
Robin.Getz@analog.com

Luc Langlois
Director, Global Solutions Team
Avnet Electronics Marketing
Luc.Langlois@avnet.com

Wireless communications will play a key role in a wide range of emerging technologies, from fleets of self-driving autonomous vehicles to heterogeneous networks connecting millions of industrial sensors. Such applications will demand reconfigurable software-defined radios (SDRs) capable of changing modulation schemes, frequency bands and system protocols on the fly. By integrating the critical RF signal path and high-speed programmable logic in a fully verified system-on-module (SOM), Avnet's PicoZed SDR delivers the flexibility of software-defined radio in a device the size of deck of cards, enabling frequency-agile, wideband 2x2 receive and transmit paths in the 70-MHz to 6.0-GHz range for diverse fixed and mobile SDR applications.

PicoZed SDR combines the Analog Devices AD9361 integrated RF Agile Transceiver™ with the Xilinx® Z-7035 Zynq®-7000 All Programmable SoC. [1] The architecture is ideal for mixed software-hardware implementations of complex applications, such as digital receivers, in which the digital front end (physical layer) is implemented in programmable logic, while the upper protocol layers run in software on dual ARM® Cortex™-A9 processors. Let's look at the software-related features of PicoZed SDR throughout the development process.

FAST PROOF OF CONCEPT WITH PICOZED SDR RADIO-IN-THE-LOOP

Leveraging the full potential of PicoZed SDR calls for a robust, multidomain simulation environment to model the entire signal chain, from the RF analog

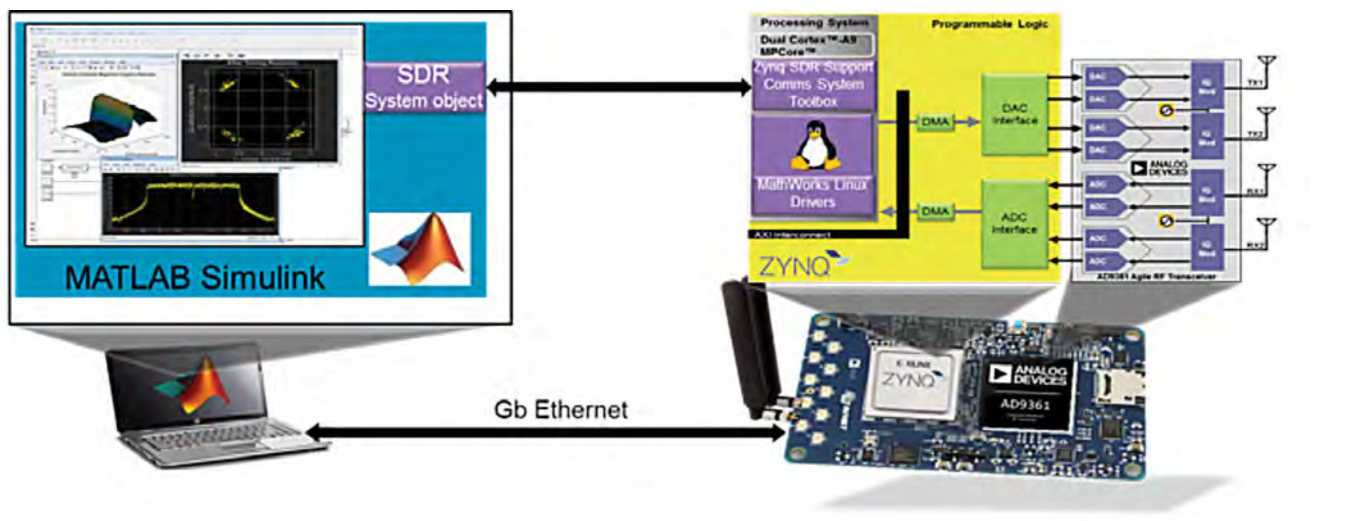


Figure 1 — Streaming data with PicoZed SDR using System objects

electronics to the baseband digital algorithms. This is the inherent value of Model-Based Design, a methodology from MathWorks® that places the system model at the center of the development process, spanning from requirements definition through design, code generation, implementation and testing. Avnet worked with Analog Devices and MathWorks to develop a support infrastructure for PicoZed SDR in each facet of the design process, starting at the initial prototyping phase. [2]

Faced with the constant pressures of shorter development cycles, engineers seek solutions for fast, accurate proof of concept on proven hardware to demonstrate the feasibility of a product under "real world" conditions. Using a MATLAB® software construct called System objects™, MathWorks created a support package for Xilinx Zynq-Based Radio that enables PicoZed SDR as an RF front end to prototype SDR designs right out of the box. Optimized for iterative computations that process large streams of data, System objects automate streaming data between PicoZed SDR and the MATLAB and Simulink® environments in a configuration known as radio-in-the-loop (Figure 1).

Akin to concepts of object-oriented programming, System objects are created by a constructor call to a class name, either in MATLAB code or as a Simulink block. Once a System object is instantiated, you can invoke various methods to stream data through the System object during simulation. The Communications System Toolbox™ Support Package for Xilinx Zynq-Based Radio from MathWorks contains predefined classes for the PicoZed SDR receiver and transmitter, each with tunable configuration attributes for the AD9361, such as RF center frequency and sampling rate. The code example in Figure 2 creates a PicoZed SDR receiver System object to receive data on a single channel, with the AD9361 local oscillator frequency set to 2.5 GHz and a baseband sampling rate of 1 megasample/second (Msps). The captured data is saved using a log.

LIBIIO LIBRARY

Analog Devices has developed the Libiio library [3, 4] to ease the development of software interfacing to Linux Industrial I/O (IIO) devices, such as the AD9361 on the PicoZed SDR SOM. The open-source (GNU Lesser General Public License V2.1) library abstracts the low-level details of the hardware and provides a simple yet complete programming interface that can be used for advanced projects.

The library comprises a high-level application programming interface and a set of back ends, as shown in Figure 3.

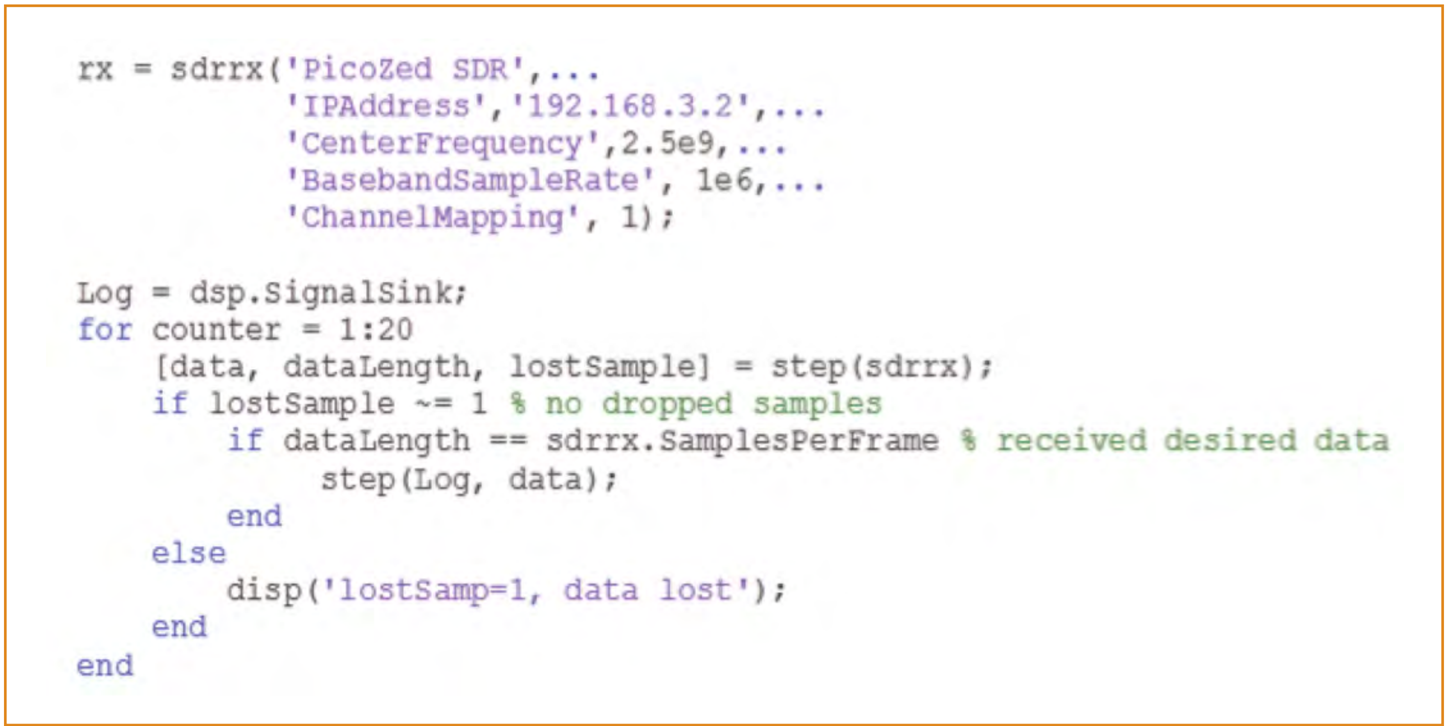
You would use Libiio to interface to a PicoZED SDR during a project's prototyping phase in order to stream samples to and from models in tools such as MATLAB, Simulink or GNURadio.

- The local back end interfaces the Linux kernel through the kernel's sysfs virtual file system. This back end has bindings for C, C++ and Python to support remotely deployed applications running on the PicoZED SDR.
- The network back end interfaces the IIO Daemon (iiod) server through a network link. The network back end supports multiple operating systems (Linux, OS X, Windows) to enable remote GUI-based debug on more-powerful host platforms, running applications such as MATLAB and Simulink [5], GNURadio [6] or the IIO Oscilloscope [7].

You would use Libiio to interface to a PicoZED SDR during a project's prototyping phase in order to stream samples to and from models in tools such as MATLAB, Simulink or GNURadio, which can model either physical layers (QPSK, QAM, OFDM, etc.) or entire media access controllers (MACs). Libiio supports both streaming (losing no samples) at medium data rates (approximately 8 Msps) and burst mode (capturing bursts of samples (up to ~1Msample, losing data between bursts) at the maximum data rate (61.44 Msps). Typically, you would use lower-data-rate streaming when working on PHY development and then use burst mode to verify the design at speed before HDL/C-code generation.

SYSTEM INTEGRATION WITH HW/SW CO-DESIGN FOR PICOZED SDR

Once you have fully verified an algorithmic model with PicoZed SDR radio-in-the-loop, the next phase would be to generate the HDL/C code and package an intellectual-property core for integration into a larger



```
rx = sdrrx('PicoZed SDR',...
           'IPAddress','192.168.3.2',...
           'CenterFrequency',2.5e9,...
           'BasebandSampleRate', 1e6,...
           'ChannelMapping', 1);

Log = dsp.SignalSink;
for counter = 1:20
    [data, dataLength, lostSample] = step(sdrrx);
    if lostSample ~= 1 % no dropped samples
        if dataLength == sdrrx.SamplesPerFrame % received desired data
             step(Log, data);
        end
    else
        disp('lostSamp=1, data lost');
    end
end
```

Figure 2 — PicoZed SDR receiver MATLAB System object

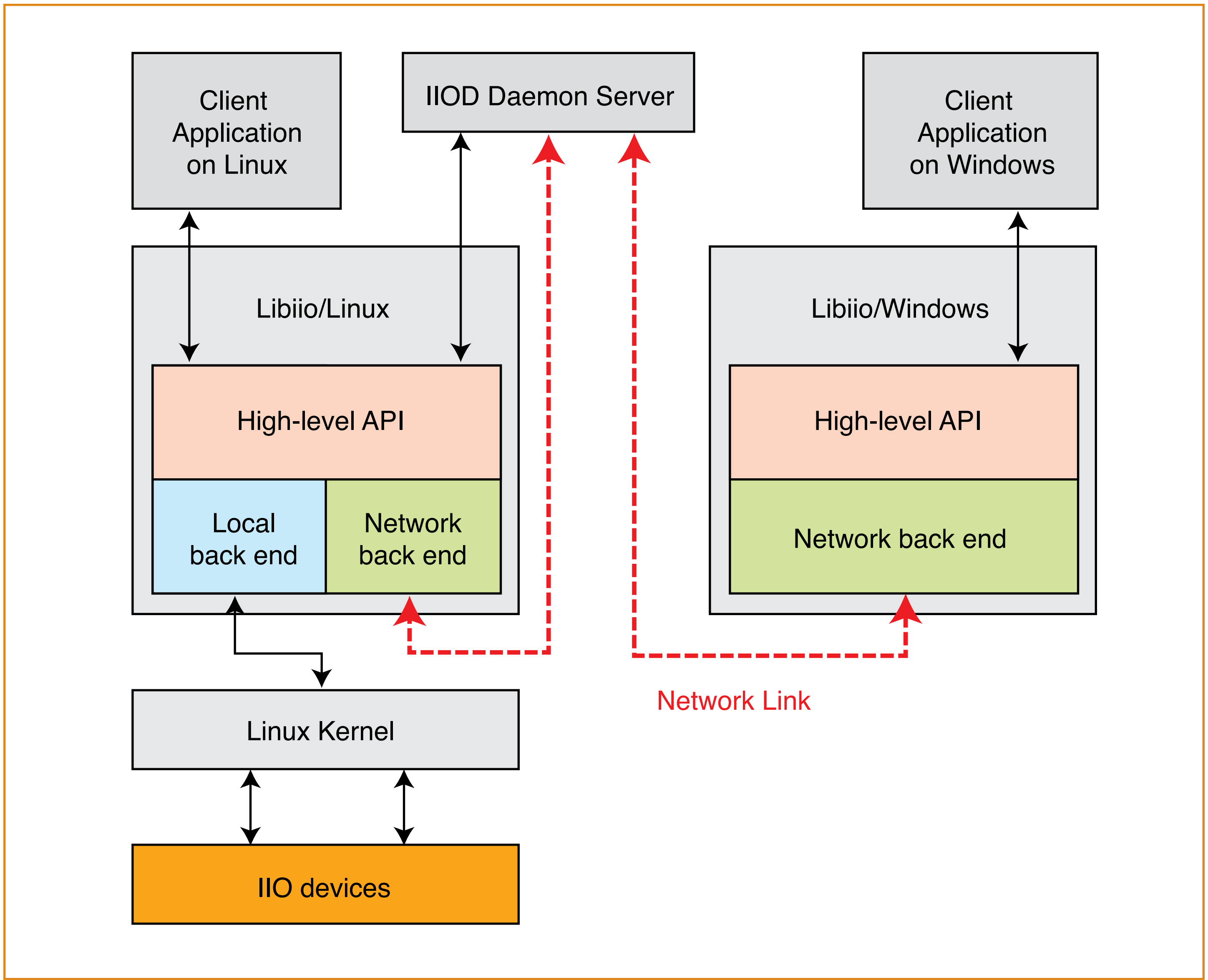


Figure 3 — Libiio API and back ends

system. For example, a wireless receiver subsystem modeled in MATLAB and Simulink might be destined for a point-to-point radio link streaming real-time video from an Avnet camera module mounted on the PicoZed SDR carrier card.

The hardware-software co-design workflow in HDL Coder™ from MathWorks lets you explore the optimal partition of your design between software and hardware targeting the Zynq SoC (Figure 4). The part destined for programmable logic can be automatically packaged as an IP core, including hardware interface components such as ARM AMBA® AXI4 or AXI4-Lite interface-accessible registers, AXI4 or AXI4-Lite interfaces, AXI4-Stream video interfaces, and external ports. The MathWorks HDL Workflow Advisor IP core generation workflow lets you insert your generated IP core into a predefined embedded system project in the Xilinx Vivado® integrated design environment. [8] HDL Workflow Advisor contains all the elements Vivado IDE needs to deploy your design to the SoC platform, except for the custom IP core and embedded software that you generate.

If you have a MathWorks Embedded Coder® license, you can automatically generate the software interface model, generate embedded C/C++ code from it, and build and run the executable on the Linux kernel

The osc application supports Linux, Windows and OS X. It can run on a remotely connected host PC or on the PicoZed SDR FMC Carrier, since it supports HDMI video display.

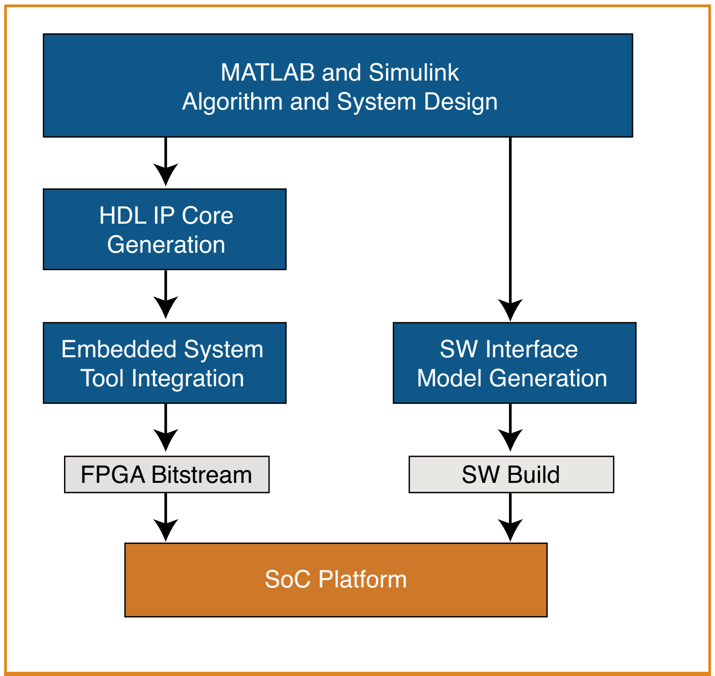
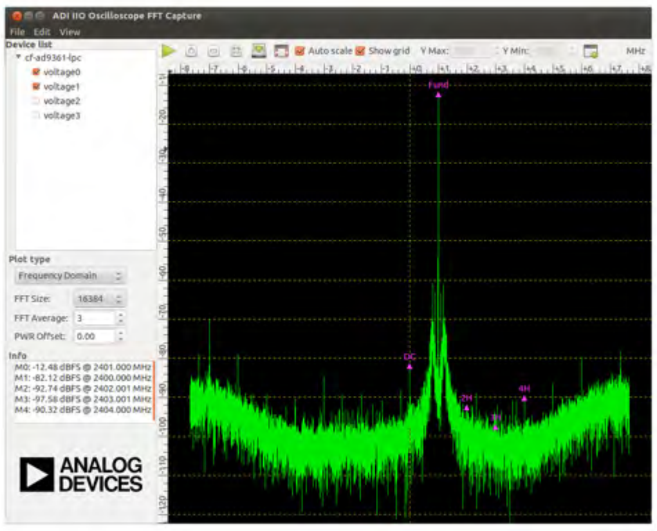

MATLAB and Simulink
Algorithm and System Design
HDL IP Core Generation
Embedded System Tool Integration
SW Interface Model Generation
FPGA Bitstream
SW Build
SoC Platform

Figure 4 – MathWorks hardware-software co-design workflow



Plot type: Frequency Domain
FFT Size: 16384
FFT Average: 3
PWR Offset: 0.00

Info
M0: -12.48 dBFS @ 2401.000 MHz
M1: -82.12 dBFS @ 2400.000 MHz
M2: -92.74 dBFS @ 2402.001 MHz
M3: -97.58 dBFS @ 2403.001 MHz
M4: -90.32 dBFS @ 2404.000 MHz

Figure 5 — ADI IIO Oscilloscope

on the ARM processor within the Zynq SoC. The generated embedded software includes AXI driver code, generated from the AXI driver blocks, that controls the HDL IP core. Alternatively, you can write the embedded software and manually build it for the ARM processor.

IIO OSCILLOSCOPE

The ADI IIO Oscilloscope (osc) is an example application that demonstrates how to interface different Linux IIO devices within a Linux system. The application allows you to plot the captured data in four modes (time domain, frequency domain, constellation and cross-correlation) and to view and modify several IIO device settings.

The osc application supports Linux, Windows and OS X. It can run on a remotely connected host PC or on the PicoZed SDR FMC Carrier because it supports HDMI video display and, therefore, a graphical environment. Instructions are available for building the osc app on Linux, and prebuilt Windows binaries exist.

Figure 5 shows the FFT of two channels (I/Q) of the PicoZED SDR, with markers set up to look at single tones and measure harmonics.

Building directly on the PicoZED SDR is a simple matter of (1) downloading the source:

```
> git clone https://github.com/analogdevic-
esinc/iio-oscilloscope.git

> cd iio-oscilloscope

> git checkout origin/master
```

and (2) building and installing it:

```
rgetz@pinky:~/iio-oscilloscope$ make

rgetz@pinky:~/iio-oscilloscope$ sudo make install
```

With the powerful processing system (dual ARM Cortex-A9 processors running at 1 GHz, plus 1-Gbyte DDR3 SDRAM) at your disposal, compiling natively on the Zynq SDR SOM is a quick process.

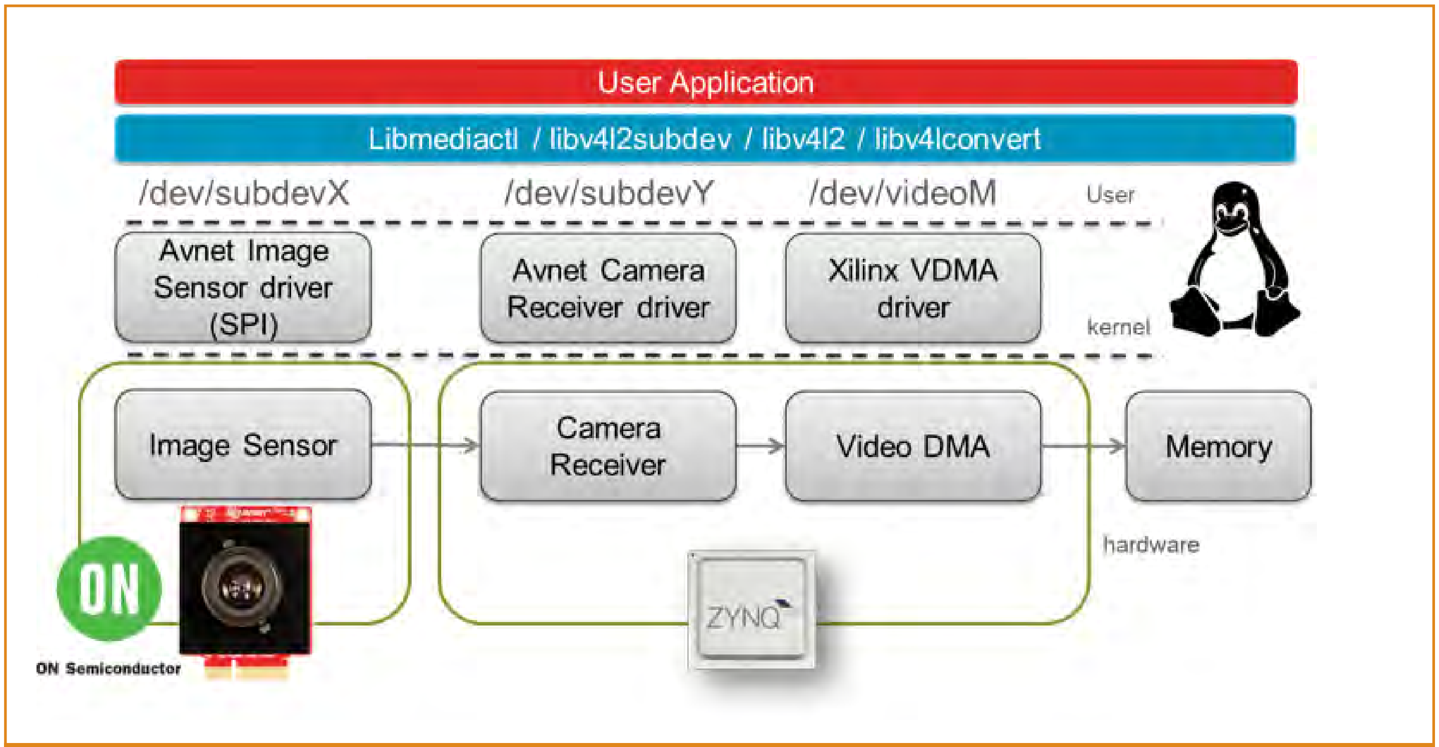


Figure 6 — V4L2 capture pipeline with Avnet ON PYTHON-1300 camera module

REAL-TIME VIDEO CAPTURE WITH PICOZED SDR

High-performance video has become a key component of intelligent systems in wireless applications such as autonomous vehicles, military vision systems, surveillance systems and drones. These applications combine high-pixel-rate video capture with real-time analytics that can exceed the performance limitations of purely software-based implementations. With the Zynq SoC and Xilinx's SDSoC™ development environment, embedded vision system designers have access to the best of both worlds, leveraging the rich heritage of existing software-based image processing algorithms through hardware acceleration for real-time processing of high-definition video at high frame rates.

The Avnet ON PYTHON-1300 camera module features ON Semiconductor's PYTHON-1300 color image sensor, which is capable of SXGA resolution (1,280 x 1,024 pixels) at 210 frames per second. [9]. Avnet supports the module on several of its Zynq SoC-based development platforms, including PicoZed SDR (for transmission of video analytics over the air). System designers can integrate the camera module into a complete Linux system using Avnet software drivers that adhere to the Video4Linux2 API specification (V4L2; Figure 6). The V4L2 framework can implement complete video data paths known as pipelines. A typical capture pipeline will capture video from the camera receiver, optionally process the video and then send the content to an external frame buffer using a video DMA engine. Avnet provides a Vivado IP Integrator-compatible "camera receiver" IP core with HDL source code, provided without a fee or royalty, and V4L2 subdevice Linux drivers available as a Linux patch.

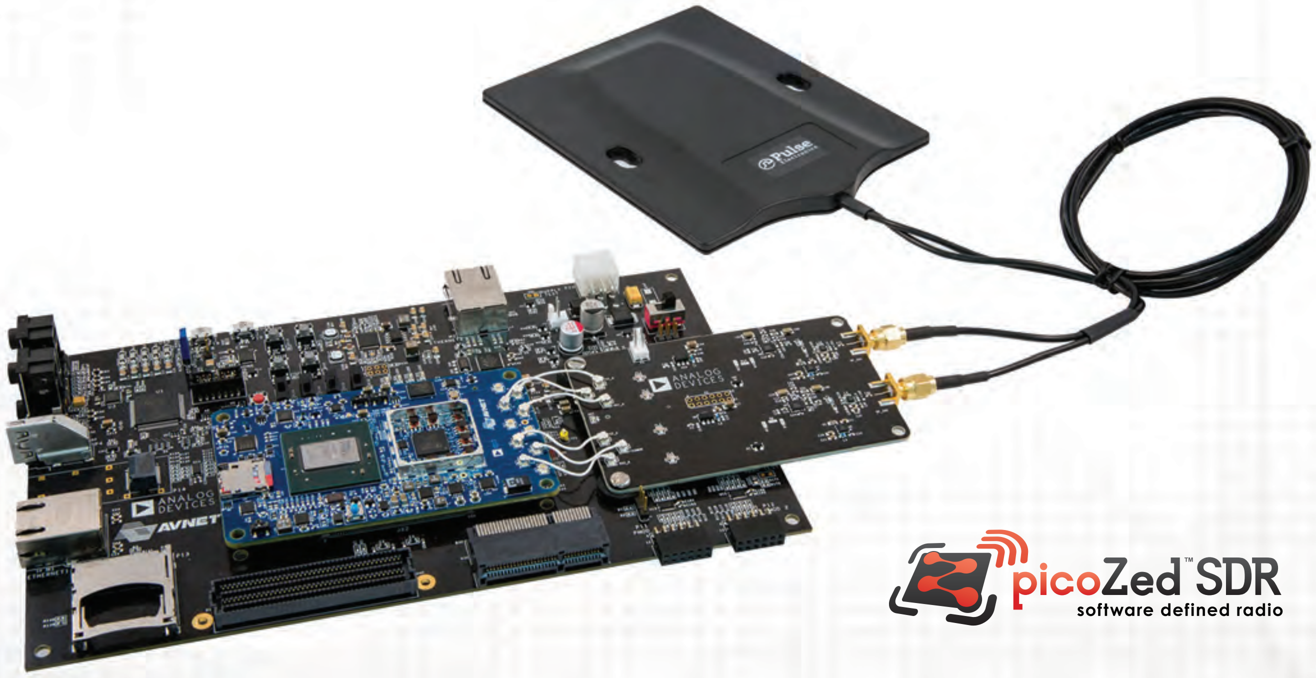
As we have shown here, through the automated workflows provided in Avnet's PicoZed SDR, you can substantially reduce your development times from concept to deployment, while focusing on your SDR products' differentiating features. ■

REFERENCES

1. PicoZed SDR Development Kit
2. Wireless Communications Design with MATLAB
3. What is libiio?
4. analogdevicesinc/libiio GitHub
5. IIO System Object
6. GNU Radio
7. IIO Oscilloscope
8. ADI Reference Designs HDL User Guide
9. Avnet ON PYTHON-1300-C Camera Module

** SEE THE FEATURED ARTICLE IN THIS ISSUE **

AVNET®



Lifecycle

Technology

Software-Defined Radio from Concept to Production

Product Benefits/Features:

- Combines the Analog Devices AD9361 integrated RF Agile Transceiver™ with the Xilinx Z7035 Zynq®-7000 All Programmable SoC
- Running Linux on the dual core ARM A9, the PicoZed SDR provides a full software environment which can be used from prototyping to production
- Userspace I/O (UIO), Industrial I/O (IIO) subsystems and drivers
- Supports Avnet camera modules with V4L2 drivers for realtime video capture
- Frequency-agile 2x2 receive and transmit paths with independent LO from 70 MHz to 6.0 GHz
- Handheld form-factor, ideal for a broad range of fixed and mobile SDR applications
- Full support in MATLAB and Simulink

For more information, visit picozed.org/sdr

facebook.com/avnet twitter.com/avnet youtube.com/avnet

Accelerating Your Success™

AMP up Your Next SoC Project

by Scott McNutt
Senior Software Engineer
DesignLinx Hardware Solutions, LLC
smcnutt@designlinxhs.com

Harness real-time performance and the rich features of Linux.

Embedded systems usually fall into one of two categories: those that require hard real-time performance and those that don't. In the past, we had to pick our poison—the performance of our "go to" real-time operating system or the rich feature set of our favorite Linux distribution—and then struggle with its shortcomings.

Today, embedded developers no longer need to choose between the two. Asymmetric multiprocessing (AMP) offers the best of both worlds.

Several modern system-on-chip (SoC) product offerings integrate multiple CPUs, a broad variety of standard I/O peripherals and programmable logic. The Xilinx® Zynq-7000® All Programmable SoC family, for example, includes a dual-core ARM® Cortex™-A9, standard peripherals (such as Gigabit Ethernet MACs, USB, DMA, SD/MMC, SPI and CAN) and a large programmable logic array. We can use these SoC products as the basis of a Linux/RTOS AMP system that provides considerable flexibility.

In many ways, the typical AMP configuration is similar to a PCI-based system, with the Linux domain functioning as the host, the RTOS domain functioning as an adapter, and one or more shared memory regions used for communication between the two domains. Unlike PCI, however, an AMP configuration can more conveniently—and dynamically—assign resources (both the standard peripherals and custom logic) to one domain or the other. In addition, a Linux/RTOS AMP system can dynamically reconfigure programmable logic based on runtime requirements, such as the presence or absence of various external devices.

This level of flexibility is often coupled with concerns about complexity and the degree of difficulty involved in bringing up an AMP system. Rest assured that the Linux development community has introduced many features into the kernel that greatly simplify AMP configuration and use.

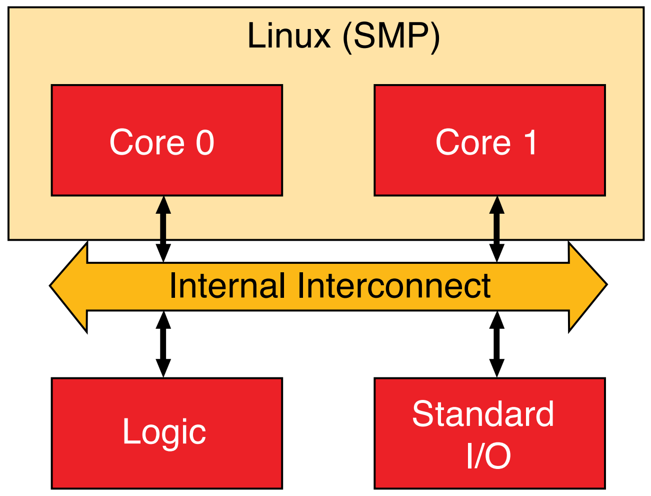


Figure 1 — Symmetric multiprocessing. The SMP kernel can run simultaneously on multiple cores.

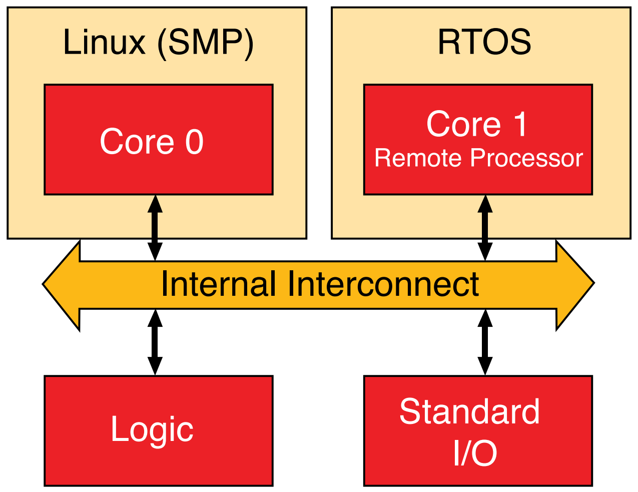


Figure 2 — AMP with the Linux SMP kernel

LINUX MULTIPROCESSING IN A NUTSHELL

With respect to multiprocessing, the Linux kernel comes in two flavors: the uniprocessor (UP) kernel and the symmetric multiprocessor (SMP) kernel. The UP kernel can only run on a single core, regardless of the number of available cores. AMP systems can incorporate two or more instances of the uniprocessor kernel.

The SMP kernel, however, can run on one core or simultaneously on multiple cores (Figure 1). An optional kernel command line parameter controls the number of cores that the SMP kernel uses following system initialization. Once the kernel is running, various command line utilities control the number of cores assigned to the kernel. The ability to dynamically control the number of cores used by the kernel is a primary reason AMP developers prefer the SMP kernel over the UP kernel.

The Remote Processor Framework

The Remote Processor (remoteproc) Framework is the Linux component that is responsible for starting and stopping individual cores (remote processors), as well as for loading a core's software in an AMP system. For example, we can dynamically reconfigure the SMP system shown in Figure 1 into the AMP system shown in Figure 2, and then back again to SMP, using the capabilities of remoteproc.

We can fully control reconfiguration via a user-space application or system initialization script. Reconfiguration control allows user applications to stop, reload and run a variety of RTOS applications based on the dynamic needs of the system.

The core's software (in our example, the RTOS and user application) is loaded from a standard Executable and Linkable Format (ELF) file that contains a special section known as the resource table. The resource table is analogous to the PCI configuration space in that it describes the resources that the RTOS requires. Among those resources is the memory needed for the RTOS code and data.

Trace buffers

Trace buffers are regions of memory that automatically appear as files in a Linux file system. As their name suggests, trace buffers provide basic tracing capabilities to the remote processor. A remote processor writes trace, debug and status messages to the buffers, where the messages are available for inspection via the Linux command line or by custom applications.

The ability to dynamically control the number of cores used by the kernel is a primary reason AMP developers prefer the SMP kernel over the UP kernel.

One or more trace buffers may be requested via entries in the resource table. Although they typically contain plain text, trace buffers may also contain binary data such as application state information or alarm indications.

Virtual I/O devices

We can also use the resource table to define virtual input/output devices (VDEVs), which are basically pairs of shared memory queues that support message transfer between the Linux kernel and the remote processor. The VDEV definition includes fields that negotiate the size of the queues as well as the interrupts used to signal between the processors.

The Linux kernel handles initialization of the virtual I/O queues. The software running on a remote processor need only include a VDEV description in its resource table and then use the queues once it begins execution; the kernel handles the rest.

Remote Processor Messaging Framework

The Remote Processor Messaging (rpmsg) Framework is a software messaging bus based on the Linux kernel's virtual I/O system. The messaging bus is similar to a local area subnetwork in which individual processors can create addressable endpoints and exchange messages, all via shared memory.

The kernel's rpmsg framework acts as a switch, routing messages to the appropriate endpoint based on the destination address contained in the message. Because the message header includes a source address, ad hoc connections can be established between various processors.

Naming service

Processors can dynamically announce a particular service by sending a message to the rpmsg framework's naming service. By itself, the naming service feature is only marginally useful. The rpmsg framework, however, allows service names to be bound to device drivers to support the automatic loading and initialization of specific drivers. For example, if a remote processor announces the service dlinx-h323-v1.0, the kernel can search for, load and initialize the driver bound to that name. This greatly simplifies driver management in systems where services are dynamically installed on remote processors.

Managing interrupts

Interrupt management can be a little tricky, especially when starting and stopping cores. Ultimately, the system needs to redirect specific interrupts dynamically to the remote processor domain when the remote processor is started, then reclaim those interrupts when the remote processor is stopped. In addition, the system must protect the interrupts from inadvertent allocation by potentially misconfigured drivers. In short, interrupts must be managed systemwide.

For the Linux SMP kernel, this is a routine matter—and a further reason that the SMP kernel is preferred in AMP configurations. The remote processor framework conveniently manages interrupts with only minimal support from the device driver.

Device drivers

Device driver development is always a concern because it requires a skill set that may not be readily available. Fortunately, the Linux kernel's remoteproc and rpmsg frameworks do most of the heavy lifting; drivers need only implement a handful of standard driver routines. A fully functional driver may only require a few hundred lines of code. The kernel source tree includes sample drivers that embedded developers can adapt to their requirements.

Generic open-source device drivers are also available from vendors. DesignLinx Hardware Solutions provides generic rpmsg drivers for both Linux and FreeRTOS. Since the generic driver makes no assumptions about the format of the messages that are exchanged, embedded developers can use it for a variety of AMP applications without any modifications.

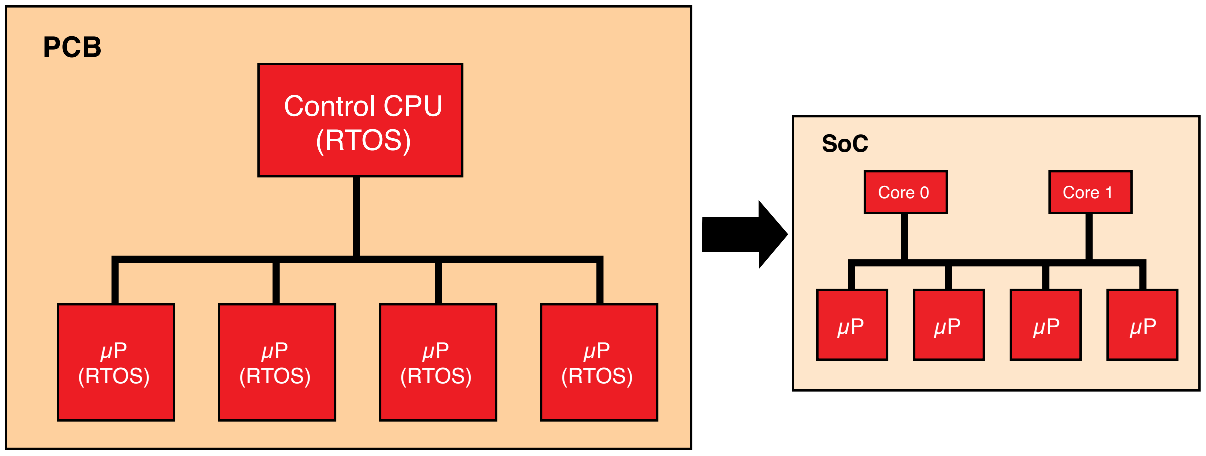


Figure 3 — Moving discrete PCB elements 'inside the pins' of an SoC

MOVING INSIDE THE PINS

The kernel's multiprocessing support is not limited to homogeneous multiprocessing systems (systems using only the same kind of processor). All of the features described above can also be used in heterogeneous systems (systems with different kinds of processors). These multiprocessing features are especially useful when migrating existing designs "inside the pins."

Modern SoC products let designers conveniently move various hardware designs from a printed-circuit board to a system-on-chip (Figure 3). What was once implemented as a collection of discrete processors and components on a PCB can be implemented entirely inside the pins of an SoC.

For example, we can implement the original PCB hardware architecture of Figure 3 with a Xilinx Zynq-7000 family SoC using one of the ARM processors as the control CPU and soft processors (such as Xilinx MicroBlaze™ processors) in the programmable logic to replace the discrete microprocessors. We can use the remaining ARM processor to run the Linux SMP kernel (Figure 4).

The addition of Linux to the original design provides all of the standard multiprocessing features described above for both the ARM cores and the soft core processors (such as start, stop, reload, trace buffers and remote messaging). But it also brings the broad Linux feature set, which supports a variety of network interfaces (Ethernet, Wi-Fi, Bluetooth), networking services (Web servers, FTP, SSH, SNMP), file systems

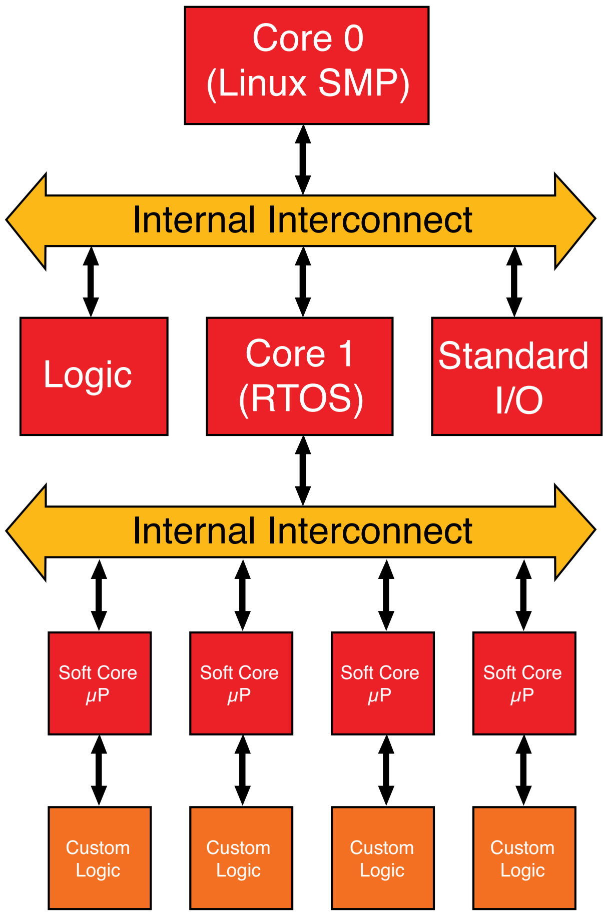


Figure 4 — Multiprocessing inside the pins

Modern SoC products let designers conveniently move various hardware designs from a printed-circuit board to a system-on-chip.

(DOS, NFS, cramfs, flash memory) and other interfaces (PCIe, SPI, USB, MMC, video), to name just a few. These features offer a convenient pathway to new capabilities without significantly altering tried-and-true architectures.

THE CORES KEEP COMING

The past several years have seen an increase in multicore SoC offerings that target the embedded market and are well suited for AMP configurations.

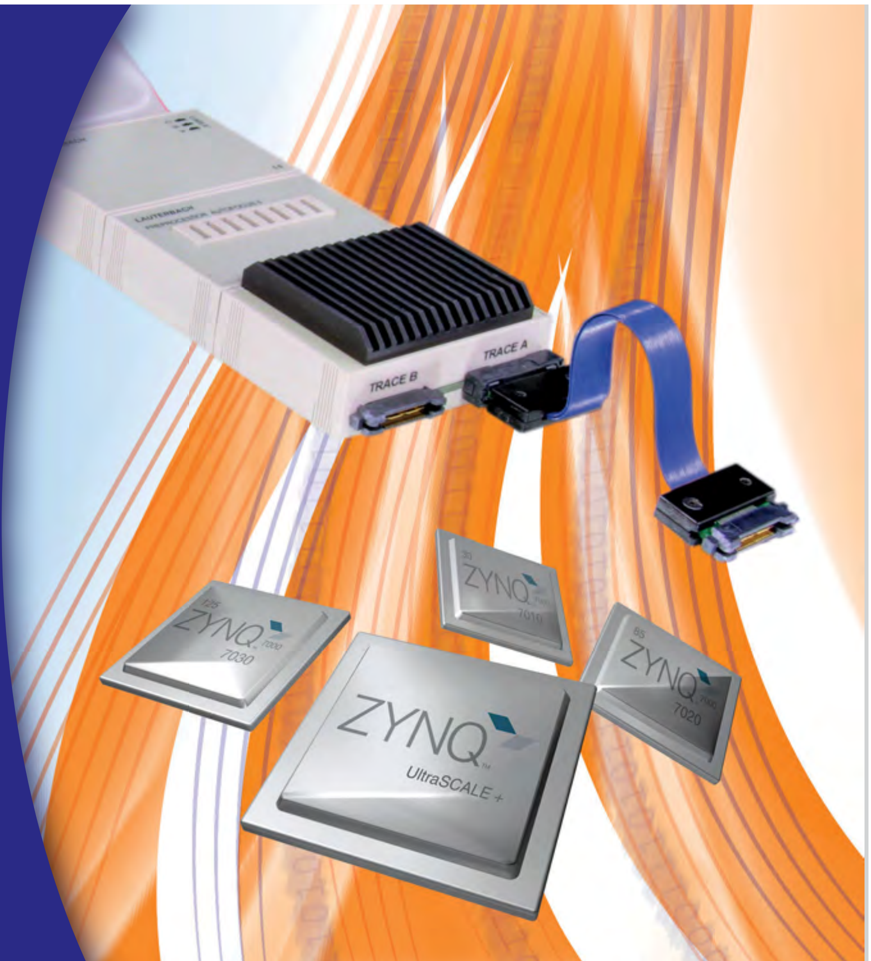
The Xilinx UltraScale+™ MPSoC architecture, for example, includes a 64-bit quad-core ARM Cortex-A53, a 32-bit dual-core ARM Cortex-R5, a graphics processing unit (GPU) and a host of other peripherals—and, of course, a healthy helping of programmable logic. This is fertile ground for designers who understand how to harness the performance of real-time operating systems coupled with the rich feature set of the Linux kernel.

For more information on designing a Linux/RTOS AMP system, contact DesignLinx Hardware Solutions. A premier member of the Xilinx Alliance Program, DesignLinx specializes in FPGA design and support, including systems design, schematic capture and electronic packaging/mechanical engineering design, and signal integrity. ■

TRACE32®

Support for
Xilinx Zynq® Ultrascale+™
and Zynq-7000 families

- ► Concurrent debugging and tracing of ARM Cortex-A53/-R5, -A9 and MicroBlaze™ soft processor core
- ► RTOS support, including Linux kernel and process debugging
- ► Run time analysis of functions and tasks, code coverage, system trace

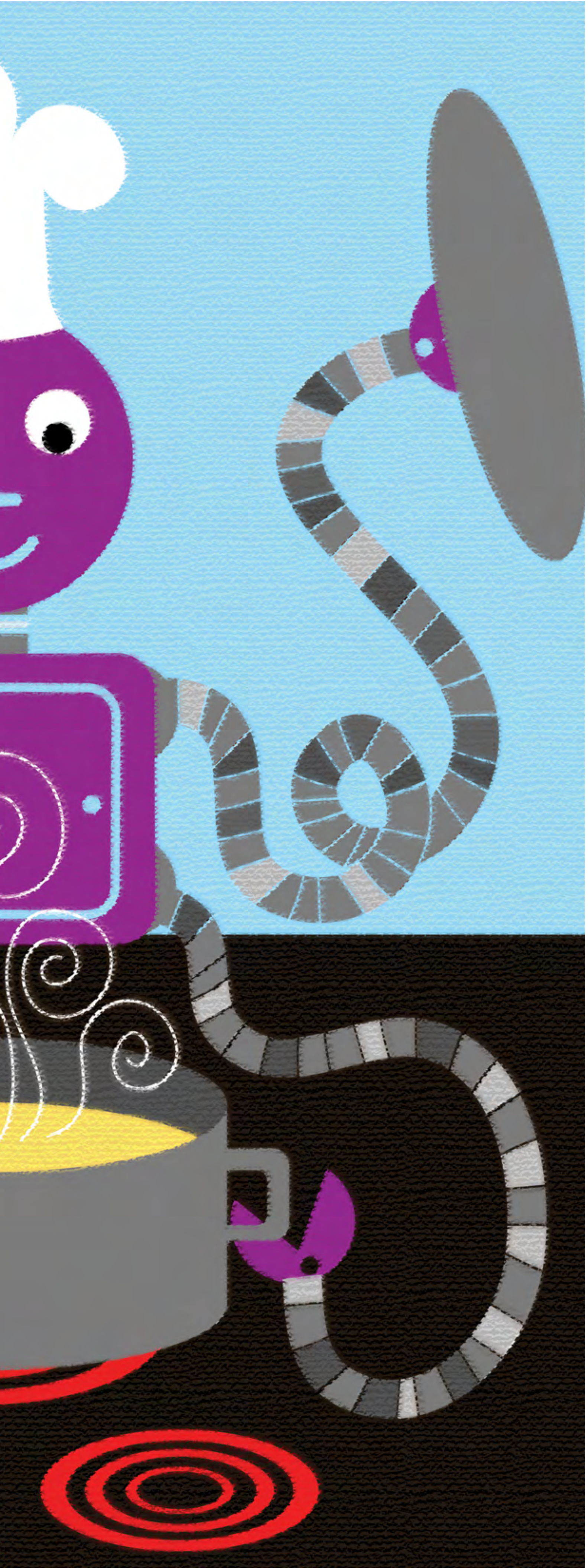


LAUTERBACH
DEVELOPMENT TOOLS

www.lauterbach.com

A Recipe for Embedded Systems

As embedded systems have become ubiquitous, we can benefit from accumulated development knowledge to build better systems.



by Adam Taylor
Chief Engineer
e2v
aptaylor@theiet.org

Engineers never lose sight of the need to deliver projects that hit the quality, schedule and budget targets. You can apply the lessons learned by the community of embedded system developers over the years to ensure that your next embedded system project achieves those goals. Let's explore some important lessons that have led to best practices for embedded development.

THINK SYSTEMATICALLY

Systems engineering is a broad discipline covering development of everything from aircraft carriers and satellites, for example, to the embedded systems that enable their performance. We can apply a systems engineering approach to manage the embedded systems engineering life cycle from concept to end-of-life disposal.

The first stage in a systems engineering approach is not, as one might think, to establish the system requirements, but to create a systems engineering management plan. This plan defines the engineering life cycle for the system and the design reviews that the development team will perform, along with expected inputs and outputs from those reviews. The plan sets a clear definition for the project management, engineering and customer communities as to the sequence of engineering events and the prerequisites at each stage. In short, it lays out the expectations and deliverables.

With a clear understanding of the engineering life cycle, the next step of thinking systematically is to establish the requirements for the embedded system under development. A good requirement set will address three areas. Functional requirements define how the embedded system performs. Nonfunctional requirements define such aspects as regulatory compliance and reliability. Environmental requirements define such aspects as the operational temperature and shock and vibration requirements, along with the electrical environment (for example, EMI and EMC).

Within a larger development effort, those requirements will be flowed down and traceable from a

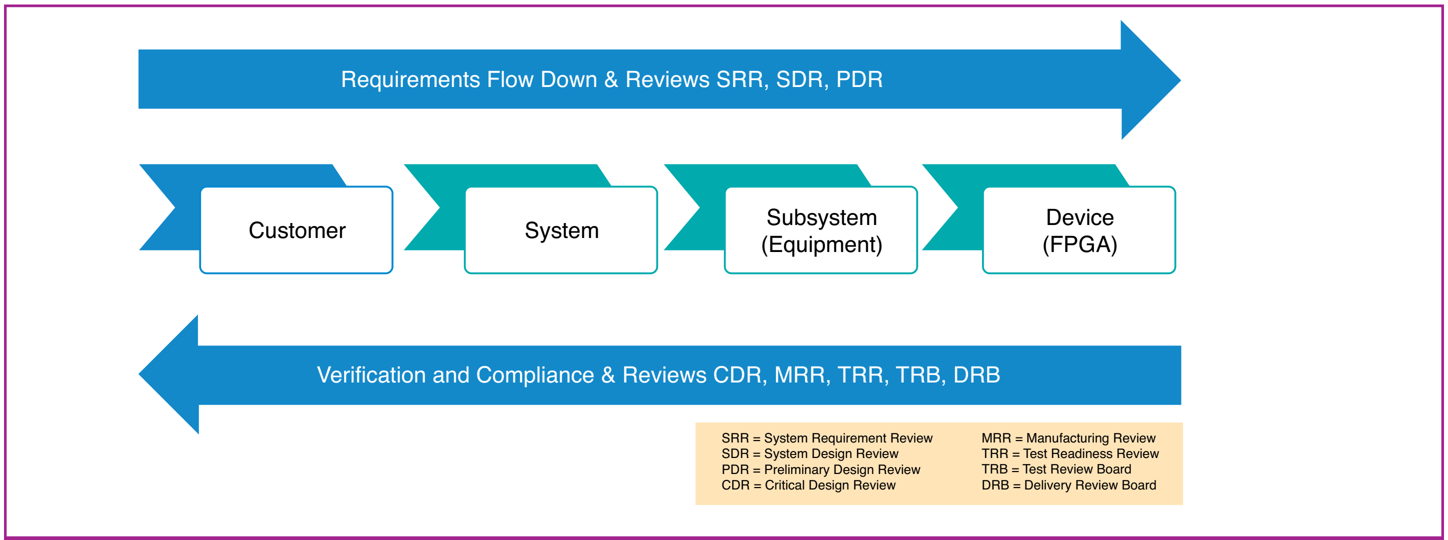

Requirements Flow Down & Reviews SRR, SDR, PDR
Customer
System
Subsystem (Equipment)
Device (FPGA)
Verification and Compliance & Reviews CDR, MRR, TRR, TRB, DRB
SRR = System Requirement Review
SDR = System Design Review
PDR = Preliminary Design Review
CDR = Critical Design Review
MRR = Manufacturing Review
TRR = Test Readiness Review
TRB = Test Review Board
DRB = Delivery Review Board

Figure 1 — Within a development effort, requirements flow down and are traceable from a higher-lever specification.

higher-level specification, such as a system or subsystem specification (Figure 1). If there is no higher-level specification, we must engage with stakeholders in the development to establish a clear set of stakeholder requirements and then use those to establish the embedded system requirements.

Generating a good requirement set requires that we put considerable thought into each requirement to ensure that it meets these standards:

1. **It is necessary.** Our project cannot achieve success without the requirement.
2. **It is verifiable.** We must ensure that the requirement can be implemented via inspection, test, analysis or demonstration.
3. **It is achievable.** The requirement is technically possible, given the constraints.
4. **It is traceable.** The requirement can be traced from lower-level requirements and can trace to higher-level requirements.
5. **It is unique.** This standard prevents contraction between requirements.
6. **It is simple and clear.** Each requirement specifies one function.

It is also common to use specific language when defining requirements to demonstrate intention. Typically, we use SHALL for a mandatory requirement and SHOULD for a nonmandatory requirement. Nonmandatory requirements let us express desired system attributes.

After we have established our requirements baseline, best practice is to create a compliance matrix, stating compliance for each requirement. We can also start establishing our verification strategy by assigning a verification method for each requirement. These methods are generally Test, Analysis, Inspection, Demonstration and Read Across. Creating the requirements along with the compliance and verification matrices enables us to:

- Clearly understand the system behavior.
- Demonstrate the verification methods to both internal test teams and external customers. This identifies any difficult test methods early on in the development and allows us to determine the resources required.
- Identify technical performance metrics. These spring from the compliance matrix and comprise requirements that are at risk of not achieving compliance.

ASSIGN ENGINEERING BUDGETS

Every engineering project encompasses a number of budgets, which we should allocate to solutions identified within the architecture. Budget allocation ensures that the project achieves the overall requirement and that the design lead for each module understands the module's allocation in order to create an appropriate solution. Typical areas for which we

Assigning a technology readiness level to each technology used in our architecture, in conjunction with the compliance matrix, enables us to determine where the technical risks reside.

allocate budgets are the total mass for the function; the total power consumption for the function; reliability, defined as either mean time between failures or probability of success; and the allowable cross-talk between signal types within a design (generally a common set of rules applicable across a number of functions). One of the most important aspects of establishing the engineering budgets is to ensure that we have a sufficient contingency allocation. We must defeat the desire to pile contingency upon contingency, however, as this becomes a significant technical driver that will affect schedule and cost.

MANAGE TECHNICAL RISK

From the generation of the compliance matrix and the engineering budgets, we should be able to identify the technically challenging requirements. Each of these at-risk requirements should have a clear mitigation plan that demonstrates how we will achieve the requirement. One of the best ways to demonstrate this is to use technology readiness levels (TRLs). There are nine TRL levels, describing the progression of the maturity of the design from its basic principles observed (TRL 1) to full function and field deployment (TRL 9).

Assigning a TRL to each of the technologies used in our architecture, in conjunction with the compliance matrix, lets us determine where the technical risks reside. We can then effect a TRL development plan to ensure that as the project proceeds, the low TRL areas increase to the desired TRL. The plan could involve ensuring that we implement and test the correct functionality as the project progresses, or performing functional or environmental/dynamic testing during the project's progression.

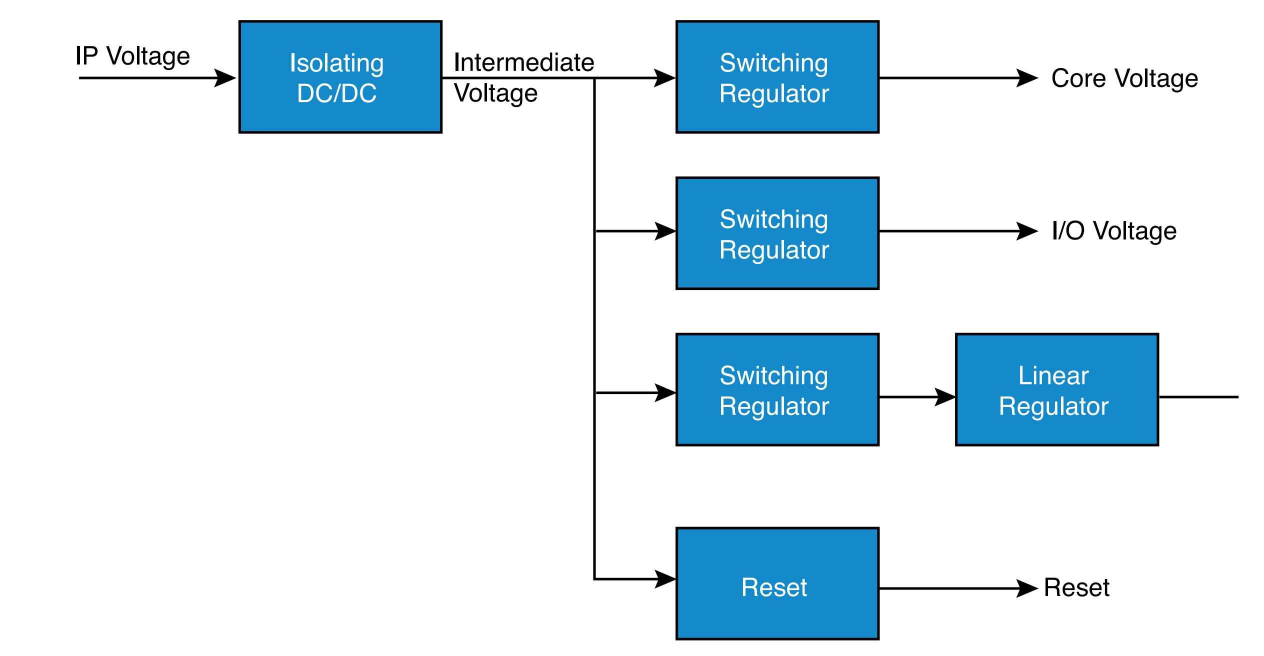


Figure 2 — In this power architecture example, the output rails from the module will require subregulation.

The architecture should not be limited to the hardware (electrical) solution, but should include the architecture of the FPGA/SoC and associated software.

CREATE THE ARCHITECTURES

Once we understand the required behavior of the embedded system, we need to create an architecture for the solution. The architecture will comprise the requirements grouped into functional blocks. For instance, if the embedded system must process an analog input or output, then the architecture would contain an analog I/O block. Other blocks may be more obvious, such as power conditioning, clocks and reset generation.

The architecture should not be limited to the hardware (electrical) solution, but should include the architecture of the FPGA/SoC and associated software. Of course, the key to modular design is good documentation of the interfaces to the module and the functional behavior.

One key aspect of the architecture is to show how the system is to be created at a high level so that the engineering teams can easily understand how it will be implemented. This step is also key for supporting the system during its operational lifetime.

When determining our architecture, we need to consider a modular approach that not only allows reuse on the current project but also enables reuse in future projects. Modularity requires that we consider potential reuse from day one and that we document each module as a standalone unit. In the case of internal FPGA/SoC modules, a common interface standard such as the ARM® AMBA® Advanced Extensible Interface (AXI) facilitates reuse.

An important benefit of modular design is the potential ability to use commercial off-the-shelf modules for some requirements. COTS modules let us develop systems faster, as we can focus our efforts on those aspects of the project that can best benefit from the added value of our expertise.

The system power architecture is one area that can require considerable thought. Many embedded systems will require an isolating AC/DC or DC/DC converter to ensure that failure of the embedded system cannot propagate. Figure 2 provides an example of a power architecture. The output rails from this module will require subregulation to provide voltages for the processing core and conversion devices. We must take care to guard against significant degradation of switching losses and efficiency in these stages. As we decrease efficiency, we increase the system thermal dissipation, which can affect the unit reliability if not correctly addressed.

We must also take care to understand the behavior of the linear regulators used and the requirements for further filtering on the power lines. This need arises as devices such as FPGAs and processors switch at far higher frequencies than a linear regulator's control loop can address. As the noise increases in frequency, the noise rejection of the linear regulator decreases, resulting in the need for additional filtering and decoupling. Failure to understand this relationship has caused issues in mixed-signal equipment.

Another important consideration is the clock and reset architecture, especially if there are several boards that require synchronization. At the architectural level, we must consider the clock distribution network: Are we fanning out a single oscillator across multiple boards or using multiple oscillators of the same frequency? To ensure the clock distribution is robust, we must consider:

- **Oscillator startup time.** We must ensure that the reset is asserted throughout that period if required.
- **Oscillator skew.** If we are fanning out the oscillator across several boards, is timing critical? If so, we need to consider skew both on the circuit cards (introduced by the connectors) and skew introduced by the buffer devices themselves.
- **Oscillator jitter.** If we are developing a mixed-signal design, we need to ensure a low-jitter clock source because increases in jitter reduce the mixed-signal converter's signal-to-noise ratio. This is also the case when we use multigigabit serial links, as we require a low-jitter source to obtain a good bit error rate over the link.

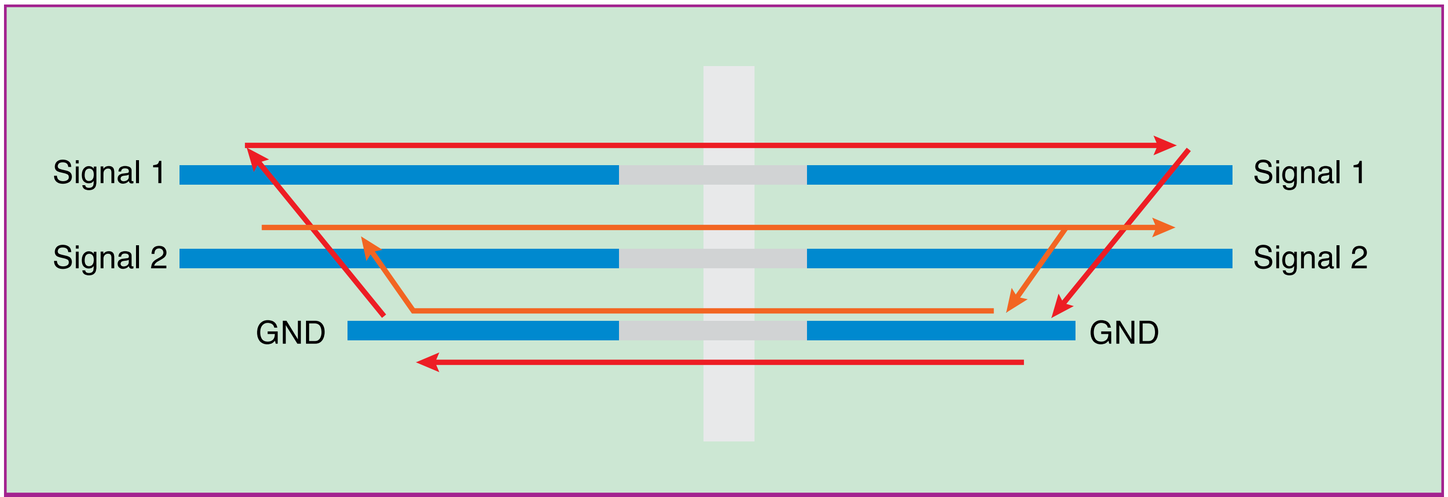


Figure 3 — Connectorization is one of the most important aspects of interface definition.

We must also pay attention to the reset architecture, ensuring that we only apply the reset where it is actually required. SRAM-based FPGAs, for example, typically do not need a reset.

If we are using an asynchronous assertion of the reset, we need to ensure that its removal cannot result in a metastability issue.

CLEARLY DEFINE INTERFACES

Formal documentation of both internal and external interfaces provides clear definition of the interfaces at the mechanical, physical and electrical levels, along with protocol and control flows. These formal documents are often called interface control documents (ICDs). Of course, it is best practice to use standard communication interfaces wherever possible.

One of the most important areas of interface definition is the "connectorization" of the external interfaces. This process takes into account the pinout of the required connector, the power rating of the connector pins and the number of mating cycles required, along with any requirements for shielding.

As we consider connector types for our system, we should ensure that there cannot be inadvertent cross connection due to the use of the same connector type within the subsystem. We can avoid the possibility of cross connection by using different connector types or by employing different connector keying, if supported.

Connectorization is one of the first areas in which we begin to use aspects of the previously developed budgets. In particular, we can use the crosstalk budget to guide us in defining the pinout.

The example in Figure 3 illustrates the importance of this process. Rearranging the pinout to place the ground reference voltage (GND) pin between Signal 1 and Signal 2 would reduce the mutual inductance and hence the crosstalk.

The ICD must also define the grounding of the system, particularly when the project requires external EMC. In this case, we must take care not to radiate the noisy signal ground.

Engineers and project managers have a number of strategies at their disposal to ensure they deliver embedded systems that meet the quality, cost and schedule requirements. When a project encounters difficulties, however, we can be assured that its past performance will be a good indicator of its future performance, without significant change on the project. ■

FURTHER READING

1. Nuts and Bolts of Designing an FPGA into Your Hardware. Xcell Journal, 82, 42-49.

2. A Pain-Free Way to Bring Up Your Hardware Design. Xcell Journal, 85, 48-51.

3. Design Reliability: MTBF Is Just the Beginning. Xcell Journal, 88, 38-43.

Next Disruptive Digital Business Model Innovation: Neuroplastic Clouds

by Francesc Fons
Dr.-Ing, MBA
ESADE Business School, Barcelona, Spain
francesc.fons@alumni.esade.edu

FPGA-based infrastructure is poised to bring neuroplasticity to IaaS, enabling high-performance, customized, secure-by-design cloud computing services.

N

Neuroscientists once believed the human brain's structure to be immutable and its neurons fixed after early childhood. Research has since proved that the brain is in fact plastic; it dynamically alters its structure, functions and neural connections—even growing new neurons via neurogenesis—in response to environmental stimuli and through the very act of thought. Further, this neuroplasticity persists into old age.

The proven plasticity of the brain is inspiring transformative digital business models founded on flexible, powerful and made-to-measure cloud computing infrastructures as a service (IaaS). Customizable cloud-based IaaS promises to enable innovative product differentiation by synthesizing optimized computations relative to the service requested, bringing added value to the end user by delivering a specific quality of service (QoS). The paradigm requires special attention to such design parameters as the performance, customization and security of the computer architecture behind the compute-intensive functionality or application. These technical features, already present in the DNA of reconfigurable hardware technology and deployable today in FPGA-driven cloud computing infrastructures, are poised to change the rules of the game in digital business.

NEUROPLASTICITY AND CLOUD COMPUTING

Recent advances in neuroscience have shown that we can rewire our brains simply by adopting new patterns of thinking and acting. The neural pathways that the brain and nervous system use to navigate activities connect relatively distant areas of the brain, and each

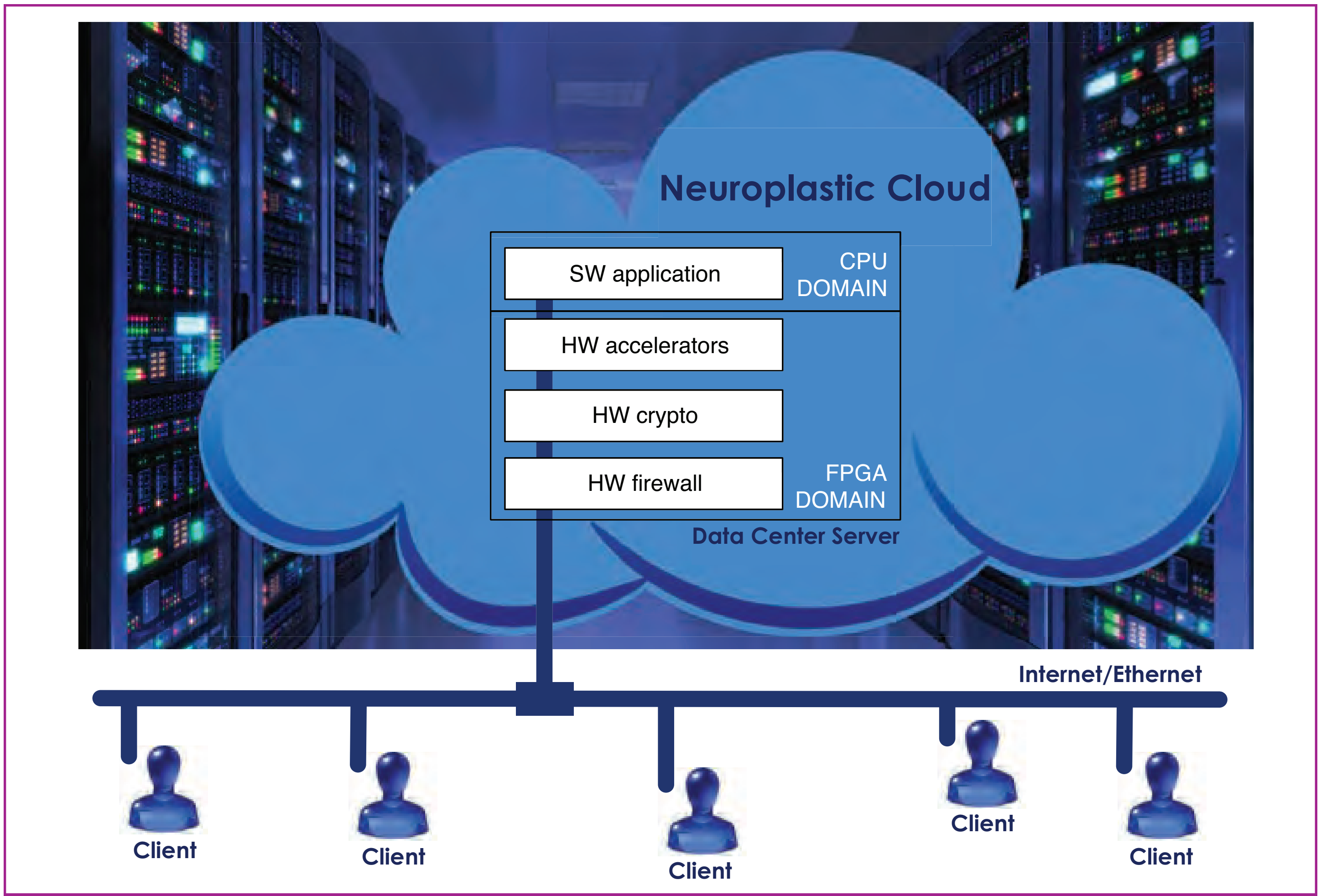


Figure 1 — Neuroplastic cloud system architecture

pathway is associated with a particular domain or behavior. New thoughts and skills carve out and pave new pathways. Every time we think, feel or do something, we strengthen the associated pathways through repetition and practice, until actions become habits.

As neuroscience has advanced our understanding of the brain, cloud computing technology has evolved over the past decade to deliver computing, networking and storage services over the Internet to billions of users around the globe. Individuals, companies and other institutions are embracing cloud services such as storage, video, messaging, social networking, online gaming and Web searching. Cloud computing is not merely an IT phenomenon; increasingly, it is an engine for enterprises to grow their businesses while decreasing their costs, enabling innovation and business transformation fueled by ever-growing computational power. Industry watchers have predicted that the bulk of future IT infrastructure spending in the coming years will be on cloud platforms and applications.

That said, can we find a relationship between the discovery of neuroplasticity and the explosion of cloud computing that might inspire a new digital business model?

Imagine abstracting the natural neuroplasticity of the human nervous system and porting the concept directly to the computing infrastructure of the cloud. With the support of the reconfigurable hardware technology available in FPGA and SoC devices, disruptive IaaS models can adapt and optimize the computer architecture of the servers allocated in the data centers of any cloud computing infrastructure to the consumer's specific computational needs at any given time, transforming the resultant computation into value added. Reinforcing the neuroscience analogy, we use

A number of research groups are convinced that FPGAs are the future enablers of application-specific accelerators in cloud computing servers.

the term neuroplastic cloud to describe a physical cloud computing infrastructure that merges processor soft cores with programmable logic (FPGAs) to yield a heterogeneous hardware-software processing ecosystem through which data center servers customize and adapt their computational power to the specific application in use in the cloud (Figure 1).

This concept encourages a cloud computing paradigm shift in that the adaptability of any computational "brain"—once synthesized in reconfigurable hardware as a made-to-measure computing unit ported to the cloud—can deliver a clear competitive advantage to digital businesses that perform cloud services under stringent and specific computational requirements often not achievable through today's standard cloud solutions. Fields of application that could benefit from this approach include financial trends analysis; real-time medical image processing; bioinformatics; computational biology; genome sequencing; real-time control of energy, oil and gas distribution; big-data analytics; and deep learning.

The notion of introducing reconfigurable hardware technology into the cloud computing arena originated with academic research. The reconfigurable-computing community has been exploring opportunities for coupling FPGAs with general-purpose processors for some time, and a number of research groups are convinced that FPGAs are the future enablers of application-specific accelerators in cloud computing servers. This understanding has not yet translated into widely available commercial solutions; but cloud technology leaders such as Microsoft, IBM, Intel, Qualcomm and Baidu, together with FPGA vendors, are strongly advocating for FPGA optimization of data center workloads.

At a time when the digital services economy is fueling cloud computing growth, cloud computing stakeholders increasingly believe that FPGA adoption in the data center will provide better performance/power efficiency than the alternatives in place today. That means cloud service providers could soon have new strategies at their disposal for delivering computing infrastructures tailored to the specific needs of their customers, quantified through valuable QoS parameters such as response time, cybersecurity and, of course, performance. The cloud computing infrastructure of any digital enterprise could drastically change with the deployment of reconfigurable hardware technology, to the extent that an enterprise would cease treating infrastructure as a commodity and would instead leverage it as a key link in the value chain of the business.

Three intrinsic properties sustain the proposed neuroplastic cloud computing infrastructure based on reconfigurable hardware: high performance (faster time to result), flexibility (optimized computation, with an architecture fully adapted to the specific app that will run) and security (data privacy, encryption and protection by design against cybersecurity threats). Joint exploitation of these characteristics promises to empower disruptive innovation in the digital business era.

HPC MOVES TO THE CLOUD

High-performance computing (HPC) is basically the use of parallel processing to run advanced application programs efficiently, reliably and quickly. Historically, access to HPC was restricted to academics, engineers and scientists; indeed, HPC became virtually synonymous with the supercomputing centers that perform the complex computations used in simulations of theoretical physical models. Typical fields of application for HPC have included climate modeling, crash simulations and bioinformatics—domains that are compute-intensive by nature.

The cloud computing phenomenon is putting HPC within everyone's reach. Day by day, the demand for this computational power is extending further beyond traditional supercomputing centers into public, private, hybrid, community and even personal clouds that users access through portable—or even wearable—embedded devices.

For many services and applications, the ever-increasing demand for computational power translates to stringent response time mandates, requiring providers to update their computing platforms with the latest technology. Real-time image processing, video streaming and big-data analytics are among the applications in which lower latencies and faster time to results are major contributors to the QoS that end users value. Big corporations as well as startups and small and midsized enterprises (SMEs) increasingly believe that advanced computational infrastructures will bring competitive advantages. As HPC goes mainstream, cloud computing is becoming critical to HPC delivery—particularly for engineering and scientific applications that already use cluster and grid computing—by enabling shared, elastic access to unlimited compute resources.

As cloud infrastructure workloads demand higher computing capabilities, greater flexibility and more power efficiency, pioneering FPGA-based alternatives can deliver high-performance solutions that exceed industry needs. Computing platforms powered by parallel processing through reconfigurable hardware technology promise an immediate and balanced solution, especially in applications that require extremely fast processing of real-time data. Through collaborations with key stakeholders, FPGA vendors are working to enable higher-performance, more energy-efficient data centers by attaching accelerators synthesized in FPGA logic to the existing processors in order to achieve dramatic latency reductions.

FLEXIBLE COMPUTING IN THE DATA CENTER

As end users become increasingly comfortable with and reliant upon hosting their personal data in the cloud, the expectation is that cloud computing will become the default method of IT delivery. An associated trend is the transformation of products into services, broken down into units that can be recombined to suit customers' precise requirements in a pay-per-use business model. IaaS is a way of providing cloud computing hardware infrastructure (servers, storage, network and operating systems) as an on-demand service over the Internet. Many enterprises acknowledge that cloud hardware infrastructure is indispensable to their business, but they treat it as a commodity that does not bring them any competitive advantage, as their competitors use essentially the same technology to offer essentially the same QoS.

Reconfigurable hardware technology can disrupt this model. The technology brings compelling performance/watt gains, lowers total cost and serves as a scalable reconfigurable acceleration platform that can be optimized on demand to any workload, thereby enabling clear differentiation. Data center servers equipped with FPGA devices can propel hardware-software computing platforms that are optimizable for specific computations. Flexible hardware will be key to bringing end users access to a wealth of features strongly tied to the cloud infrastructure, such as application-specific high-performance computing by design.

Several factors argue in favor of a shift to flexible computing in the cloud. FGPAs' parallel processing capabilities are one clear advantage. Moreover, the integration of heterogeneous hardware resources in the cloud through FPGAs offers an opportunity to improve computational efficiency without relying on continued CPU performance scaling.

Additionally, to increase the throughput of any specific computation task, it is often possible to pipeline its implementation in FPGA resources. Meticulously pipelining the end-user application through flexible hardware can deliver a customized solution with optimized performance.

It is further possible to leverage partial reconfiguration of the hardware resources in the FPGA, swapping different custom coprocessors in and out on specific resources at runtime in order to compute—on demand and dynamically, in a multiplexed way—the most compute-intensive stages of the demanded application algorithm. The approach incurs nearly no degradation in execution time while delivering a cost-effective solution that balances area and performance.

Adoption of reconfigurable hardware for cloud-based IaaS not only would make cloud computing more accessible and customizable but also would enhance cybersecurity.

To sum up, made-to-measure cloud computing can bring key differentiation to providers and practitioners in a number of fields. For example, finance computing applications powered by specific computer infrastructures, instead of standard ones, can speed brokers' analysis of financial trends so that they can move to buy or sell shares more quickly than their competitors. Medical staff can improve imaging quality and response time in surgeries performed remotely if the computer that handles the requisite real-time processing algorithms is specifically optimized to perform them. Big-data analytics on convolutional neural networks and online gaming are other computational fields in which high QoS, measured in terms of latency and overhead, can bring a clear competitive advantage.

TRUSTED CLOUD COMPUTING

Cybersecurity and data privacy challenges are the main barriers to universal cloud computing adoption. Because the cloud infrastructure, to varying degrees, is always an open and shared resource, it is a target for malicious attacks from both insiders and outsiders, and today the security implications of executing certain computations or storing sensitive data on shared resources make cloud computing inadvisable for critical applications. Side-channel attacks, identity hijacking and distribution of malicious code have all been observed in current cloud platforms and infrastructure. Trusted cloud computing solutions that eliminate such breaches are therefore critical to the efforts to enable customizable cloud computing anytime, anywhere, by anyone.

Adoption of reconfigurable hardware for cloud-based IaaS not only would make cloud computing more accessible and customizable but also would enhance security compared with software-based solutions. By design, FPGA devices offer a substantially smaller and more well-defined attack surface than the software-based solutions traditionally used in the cloud. Designers of FPGA-based cloud computing infrastructure can push security to the top of the design criteria for the cloud system architecture. The following factors argue in favor of hardware-strengthened trusted cloud computing.

- **Hardware security primitives and protection against tampering.** FPGAs provide certain security features, such as physical unclonable functions (PUFs), an alternative mechanism for key storage that provides a unique identifier for each integrated circuit. FPGAs are also suitable for implementing true random-number generators (TRNGs), required for creating cryptographic keys.

- **Hardware-based implementation of cryptographic algorithms.** The Advanced Encryption Standard (AES) and Elliptic Curve Cryptography (ECC) are examples of cryptographic algorithms performed in hardware. The cryptographic primitives (rotations, XOR operations, etc.) of such algorithms are better suited to deployment in FPGA hardware than to sequential software execution on CPUs. AES, for instance, is decomposed in a set of stages or steps, executed sequentially, with each stage decomposed internally as a loop of basic operations. These loops can be performed faster by unrolling them in hardware, making use of parallel execution. Moreover, the steps can be pipelined to increase performance. These techniques optimize the synthesis of cryptographic algorithms in hardware.

- **Data privacy at any time (data-in-motion, data-in-use and data-at-rest).** Performance degradation is a key concern when developing security solutions. Hardware-based solutions offer advantages over software approaches because the hardware can perform low-latency data encryption and decryption with virtually no overhead. In this way, all of the information that the application manages through the cloud can be sent, received and stored encrypted—i.e., as ciphertext instead of plaintext—guarding it against cyberattack.

Realizing the full potential of the cloud to serve today's tech-savvy users will require new business models that leverage a scalable and customizable set of computing, networking and storage resources.

- **Hardware-based firewalls.** Hardware security modules filter all data going through the system communication bus, strengthening the data's resilience against specific kinds of attacks.
- **Digital signature.** Hardware approaches also support user authentication as well as verifiable attestation and certificate management, thereby empowering Root-of-Trust (RoT).

ADDITIONAL BENEFITS

Industry use cases illustrate how hardware-based cloud solutions support the key attributes of security, high performance and flexibility. They include the Google Project Vault, which embeds cryptographic computing synthesized in hardware into a microSD device; the FPGA-based Microsoft Azure SmartNIC, allocated in the servers to offload software-defined networking functions from the CPU; the Catapult Project, a push by Microsoft to speed Bing's search engine through FPGA technology; and the Bitfusion Cloud Adaptor initiative, which lets developers use FPGAs in the cloud. FPGA technology is already available to prove the neuroplastic cloud-based computing concept in data center servers; the Xilinx® Kintex® UltraScale™ FPGA and Xilinx Zynq® Ultrascale+™ MPSOC device series are valid examples.

Further considerations that argue for FPGA-based cloud computing infrastructures include:

- **Scalability.** Cloud computing is bringing great cost reductions by leveraging economies of scale in data centers. FPGA-based cloud computing solutions scale far more easily than solutions based on CPUs and GPUs.
- **Low power.** More important than raw performance in the context of data centers is performance per watt. Data centers require high performance but at a power profile within the limits of data center server requirements. FPGA-based solutions can deliver much higher performance/watt than other alternatives in the market. Clearly, maximizing performance/watt is essential to improving data center reliability and managing operating costs.
- **Environmental friendliness.** Extending the argument of reduced power consumption, FPGA-based cloud computing is emerging as an unsurpassed way to reduce computing's carbon footprint.
- **Redundancy.** FPGA technology, managing the heterogeneity of hardware resources, lets developers synthesize customized solutions with the specific redundancies required.

Consumers of IT services want it all: mobility, connectivity, instant access to information, immediate computation results and security by design. Cloud computing offers enterprises the means to shift tasks from their local IT infrastructure to remote, optimized computing clusters. But realizing the full potential of the cloud to serve today's tech-savvy users will require new business models that leverage a scalable and customizable set of computing, networking and storage resources to create value for all customers.

Neuroplastic cloud computing overlays high-performance and trusted computing architectures on top of fine-grained FPGA devices equipped with heterogeneous resources to improve computational capability, flexibility and security. Hardware neuroplasticity will revolutionize the way people do business by adding personalized, individually tailored computing capabilities to the cloud's established connectivity and mobility features.

Gauging by the frenetic stakeholder activity already under way, the rapid assimilation of reconfigurable hardware technology into the cloud is not far off. The combination of software code running on the data center server CPUs with critical sections of the application processed directly in hardware will enable technological differentiation that delivers a clear competitive advantage to the end user. ■

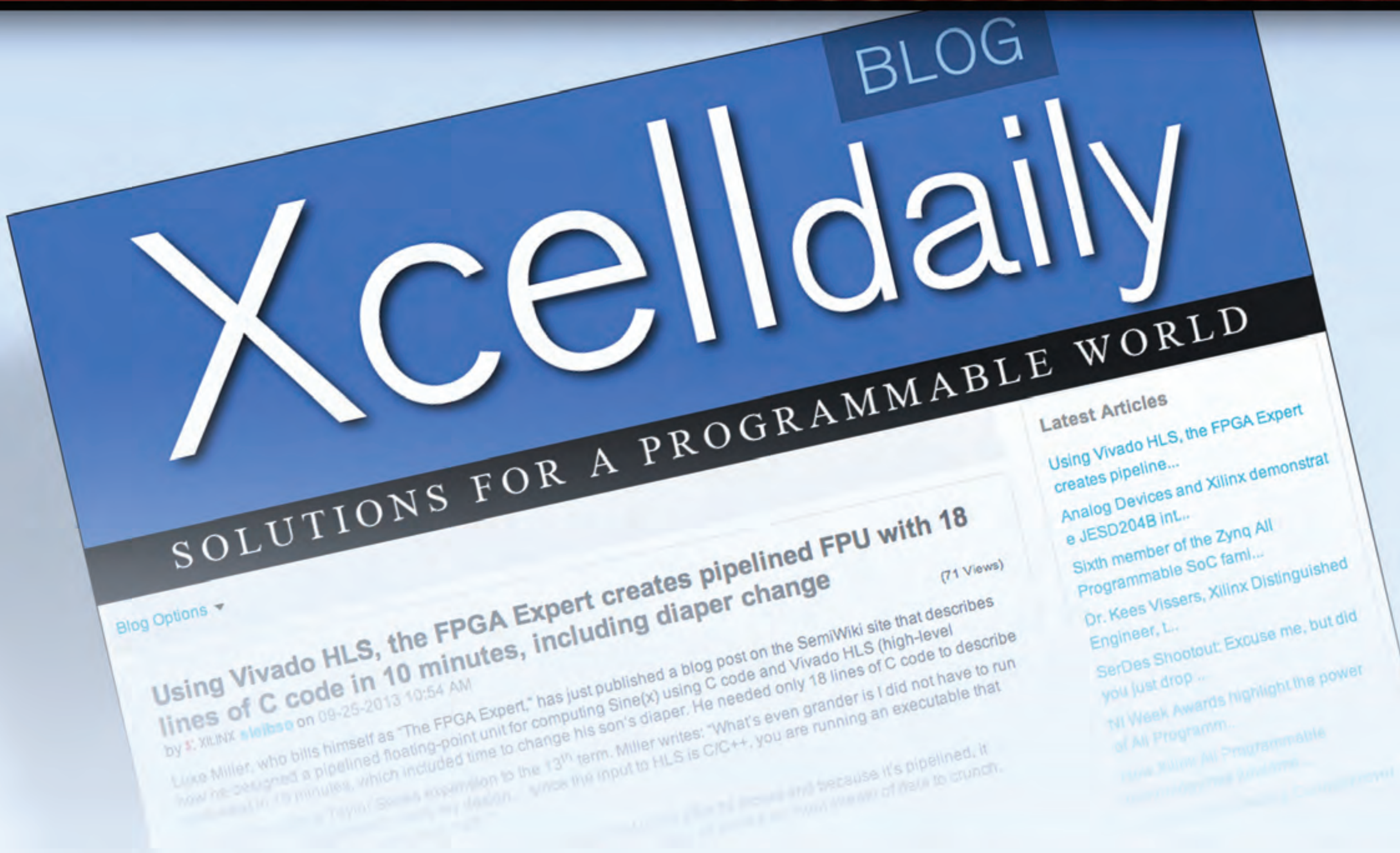
HAPS-80 FPGA-Based Prototyping Solution Delivers Up to 100 MHz System Performance

- New automated high-speed pin-multiplexing delivers the highest system performance
- Integrated ProtoCompiler software automates partitioning to reduce time to first prototype to less than two weeks
- Built-in debug enables the capture of thousands of RTL signals
- Scalable architecture supports up to 1.6 billion ASIC gates based on the Xilinx Virtex UltraScale FPGA

To learn more visit: **www.synopsys.com/haps-80**

SYNOPSYS®
Silicon to Software™

Xcell Journal Adds New Daily Blog


BLOG

Xcelldaily

SOLUTIONS FOR A PROGRAMMABLE WORLD

Blog Options ▾

Using Vivado HLS, the FPGA Expert creates pipelined FPU with 18 lines of C code in 10 minutes, including diaper change

(71 Views)

by XILINX sleibso on 09-25-2013 10:54 AM

Luke Miller, who bills himself as "The FPGA Expert," has just published a blog post on the SemiWiki site that describes how he designed a pipelined floating-point unit for computing Sine(x) using C code and Vivado HLS (high-level synthesis) in 10 minutes, which included time to change his son's diaper. He needed only 18 lines of C code to describe a Taylor Series expansion to the 13th term. Miller writes: "What's even grander is I did not have to run [illegible] since the input to HLS is C/C++, you are running an executable that [illegible] because it's pipelined, it [illegible] of data to crunch.

Latest Articles

Using Vivado HLS, the FPGA Expert creates pipeline...

Analog Devices and Xilinx demonstrate JESD204B int...

Sixth member of the Zynq All Programmable SoC fami...

Dr. Kees Vissers, Xilinx Distinguished Engineer, t...

SerDes Shootout: Excuse me, but did you just drop ...

NI Week Awards highlight the power of All Programm...

[illegible]

Xilinx has extended the Award Winning Journal and added an exciting new *Xcell Daily Blog*. The new site provides dedicated readers with a frequent flow of content to help engineers leverage the flexibility and extensive capabilities of Xilinx products, ecosystem, and customers to create All Programmable and Smarter Systems.

Recent

- ***4 Minutes to Error-Free 100G Ethernet Operation using Xilinx UltraScale+ Integrated 100G IP***
- ***Avnet introduces $699 Zynq-based, Multiprotocol, Industry 4.0/IIoT MicroZed Kit***
- ***3 Eyes are Better than One for 56Gbps PAM4 Communications: Xilinx silicon goes 56Gbps for future Ethernet***
- ***Tiny Zynq Board "does absolutely nothing, and that's a good thing" explains Servaes Joordens***
- ***How I got lost and beer-buzzed in Prague just so I could see Sphericam's Zynq-based, 360-degree, 4K video VR camera***

Visit Blog: www.forums.xilinx.com/t5/Xcell-Daily/bg-p/Xcell

XILINX
ALL PROGRAMMABLE.