WP473 (v1.0.1) April 11, 2016

Software Migration to 64-bit ARM Heterogeneous Platforms

Zynq UltraScale+ MPSoCs provide a robust platform upon which system architects can innovate without concern about loss of investment in their existing software infrastructure.

ABSTRACT

The ARM®v8 architecture at the heart Zynq® UltraScale+™ MPSoC allows system designers to get their existing ARMv7 code up and running quickly with minimal rewrites. This architectural compatibility allows designers to increase productivity and accelerate their time to market while also reducing development costs and engineering investment.

Introduction

Xilinx Zynq UltraScale+ MPSoCs, based on next-generation ARMv8 architecture, allow software designers to harness cutting-edge performance while also preserving their existing software investment.

In a recent column in the Association for Computing Machinery journal *Communications of the ACM*, Steve Furber notes that 32-bit ARMv7 SoCs, such as Zynq UltraScale+ MPSoCs, are the most prolific and popular of all time.[Ref 1] Implicitly, this large installed base of ARMv7 SoCs represents an even larger installed base of software from simple libraries and tools all the way up to complete operating systems. As the recent migration to 64-bit platforms continues to accelerate, system designers are faced with the question of whether to undertake the monumental effort of porting their existing software base to a new architecture or simply starting over on the new platform.[Ref 2]

A software project of any reasonable size represents a significant investment in both manpower and software infrastructure. Losing these investments can be catastrophic for both the short-term and long-term future of a project. Thus, porting software between SoC platforms can be the source of great uncertainty. Errors during the process can easily result in less-than-optimal capabilities and performance. In this era of shrinking design budgets and accelerated time-to-market requirements, there is little margin for uncertainty.

Xilinx's latest addition to the Zynq Portfolio, the Zynq UltraScale+ MPSoC, is based on next-generation ARMv8 architecture, which addresses these concerns by design. By leveraging a familiar CPU architecture and software development process, ARMv8 allows designers to start their porting project—with minimal loss of their existing software infrastructure investment.

ARMv8 Eases Software Migration by Design

The ARMv8 architecture found in Zynq UltraScale+ MPSoCs fundamentally eases the software porting effort by being compatible with the ARMv7 architecture that preceded it, while also extending it to support the latest features and capabilities. This creates a familiar development environment that allows software developers to focus on their code from the beginning of the porting effort. The ARMv8 architecture accomplishes this deep compatibility in two fundamental ways:

- The ARMv8 architecture supports two distinct execution states:
  - AArch32, a native 32-bit state
  - AArch64, a native 64-bit state
- The 64-bit ARMv8 instruction set found in the AArch64 state is a natural extension of that found in ARMv7, while the 32-bit instruction set found in the AArch32 state is directly compatible with that found in ARMv7.

The AArch64 execution state is the native ARMv8 execution environment. It supports the full feature set and capability of the ARMv8 architecture. In contrast, the AArch32 execution state provides native backwards compatibility with the environment found in ARMv7. From a software execution perspective, this state is a native ARMv7 environment. Developers choose which execution state the code should run in at compilation time.

During normal run time, the CPU switches between AArch64 execution and AArch32 execution on-the-fly to accommodate the execution environment required by the software. These context changes happen on exception boundaries with no input required from either the user or software designer; 32-bit code and 64-bit code execute effectively side-by-side.

When operating in AArch64 state, the full 64-bit ARMv8 instruction set is exposed. To a large extent, the 64-bit ARMv8 instruction set is an extension of that found in ARMv7. With slight changes in syntax and behavior for the instruction set, it supports 32-bit instructions and a mixture of 32-bit and 64-bit registers. When operating in AArch32 state, the processor uses standard 32-bit instructions as well as 32-bit registers; software designers can treat the instruction set just as they would any other ARMv7 processor.

Utilizing the Codebase

Developers of low-level bare metal and RTOS code are among those who are most frequently affected by differences in the underlying processor architecture. When starting to port the software, a best practice is to enable all warnings and error messages in the compiler and simply try recompiling it without modification using the ARMv8 compiler. Any warnings or errors generated by this exercise should be analyzed so that the software development team can determine which can be safely ignored vs. those that will require greater attention during the porting process.

ARMv8 compilers support standard high-level code such as C and C++ natively; this code will compile and run after an appropriate ARMv8 Board Support Package (BSP) is in place. Assembly code, by contrast, requires that careful attention be paid to how the code is used. While many assembly instructions from ARMv7 still exist in ARMv8, their syntax or behavior can differ in very subtle ways. Some coding constructs that do not compile or behave as expected relative to ARMv7 include hard-coded memory locations (true of any software porting project), access to the ARMv7 coprocessors (such as CP15) and register names, and data alignment. The *ARM® Cortex™-A Programmer's Guide for ARMv8-A* (DEN0024A), published by ARM, presents a detailed analysis of porting concerns.

When conceptualizing the differences between the architectures, it is often helpful to look at code examples that are easy to port as well as code that might present more of a challenge. The ADD instruction is a good example of an instruction that needs minimal effort to convert from ARMv7 to ARMv8 syntax. Registers in ARMv7 have a naming scheme of **R*n*** where ***n*** is the number of the register. This is straightforward because all registers are 32-bit in ARMv7. In ARMv8, though, registers can be either 32-bit or 64-bit. 32-bit registers follow a naming scheme of **W*n***, while 64-bit registers follow a naming scheme of **X*n***. Thus, software designers must evaluate the existing code and make appropriate changes, depending on the sizes of the operands in question. See Table 1.

Table 1: **ADD Instruction Naming Schemes in Various Environments**

ARMv7	ARMv8 A32	ARMv8 A64 (32-bit)	ARMv8 A64 (64-bit)
`ADD Rd, Rn`	`ADD Rd, Rn`	`ADD Wd, Wn`	`ADD Xd, Xn`

Porting instructions such as ADD require little more than evaluating the intended goal of the code and then altering register names accordingly. This migration process can even be automated, depending on how complex the overall codebase is.

Other scenarios, such as accessing ARMv7 coprocessors, might require a more thorough code analysis. The analysis is considered a best practice, because these coprocessors control many system-level attributes and functions for the entire processor, and manipulating them is a common operation for ARMv7.

When operating in AArch64 state, the CP15 registers are not directly accessible because they do not exist in the ARMv8 architecture. When operating in AArch32 state, though, they are accessible via a conceptual coprocessor. This is a construct exposed by the AArch32 execution state to present a consistent execution environment for legacy 32-bit code.

For example, suppose a program needs the contents of the Main ID Register, traditionally part of the `c0` register found in coprocessor CP15. This value is accessed by using the specialized MRC instruction for ARMv7 and ARMv8's AArch32 execution state:

```
MRC p15, 0, r1, c0, c0, 0
```

Legacy code running directly in AArch32 state requires no modification, but code ported to AArch64 state needs to be adjusted for the requirements of the new architecture.

In the native AArch64 execution state the Main ID Register value is found in a new dedicated register called MIDR_EL*n*. This register is easily accessed with the system register access instruction MRS:

```
MRS x1, MIDR_EL1
```

Thus, developers should be aware of specialized operations and calls like this so that the assembly instructions can be changed accordingly. Still, this is a single-line change that does not impact the overall structure of the surrounding software code. In addition, note that the source and target registers are both natively 64-bit registers in line with the remainder of the 64-bit architecture.

Keeping Existing Software Intact

For software based on the Linux operating system, the porting scenario is often even more straightforward than for developers of RTOS or bare metal code. The underlying Linux operating system abstracts away many of the differences in the underlying hardware. In most cases, recompilation with a 64-bit Linux compiler is all that is needed to get the code running on 64-bit Linux. The Xilinx PetaLinux tools can automate this process by natively incorporating and recompiling such code for the 64-bit Linux environment.

In other cases, developers might need to maintain the code as 32-bit binaries. The software might be linked against external libraries that are only available in 32-bit form; or, it might include 32-bit assembly code that will take time to convert. Porting this kind of software can be accelerated greatly by utilizing a feature found in many modern Linux distributions called *multiarch* or *multib*. While not strictly equivalent, these terms are often used interchangeably.

Multiarch is a way of configuring the system and the layout of its file system to provide the capability to run both 32- and 64-bit dynamically linked programs. Legacy 32-bit ARMv7 Linux programs can be dropped in unmodified and coexist with new 64-bit programs. This capability can be utilized to quickly get the system up and running while the specific 32-bit applications are ported and tuned for the underlying 64-bit architecture.

Multiarch-capable distributions accomplish this compatibility by incorporating both 32-bit and 64-bit libraries into the same Linux file system. When a 64-bit application is executed, it utilizes the native 64-bit libraries; a 32-bit application uses 32-bit libraries instead.

Creating and maintaining a multiarch operating system can require a non-trivial investment of time to maintain independently. Fortunately, open source offerings such as those from the Yocto Project, Canonical, and Linaro provide easy-to-integrate options. While the Yocto Project provides a complete end-to-end build system, both Canonical and Linaro provide ready-to-use drop-in reference file systems that allow developers to quickly make progress. Xilinx Answer 66636 provides details about integrating the Ubuntu Core 14.04 file system on a ZCU102 evaluation platform.

By default, these multiarch file systems are configured to use only 64-bit applications. Since they are based on the `dpkg` management tool, additional architectures can be added with a simple command line interface:

```
# dpkg -add-architecture armhf
```

After configuring the `dpkg` tool to support the 32-bit architecture (e.g., `armhf`), higher-level package management tools such as `apt-get` can be used just as they would be on an Intel x86-compatible version of Linux.

For some systems, running existing 32-bit applications on a 64-bit host via multiarch is sufficient and no additional effort is needed. For others, multiarch can serve as a time-saving enablement effort, while other components of the software stack are tuned and optimized for the underlying 64-bit architecture and operating system.

Putting It All Together

ARMv8 provides software developers a plethora of new capabilities in terms of performance, security, and architecture—but those new capabilities will go untapped if software developers are reluctant to adopt it over concerns about losing their existing software infrastructure. ARMv8 allays these fears by providing a natural evolution of the processor architecture from both the hardware and software interfaces. Porting low-level code is streamlined by providing a natural evolution of the instruction set that developers are already familiar with. In addition, the ability to fall back to AArch32 state provides developers with the ability to get existing 32-bit code running quickly so that they can evaluate it on the new hardware with minimal investment of time and effort. Finally, multiarch Linux allows some code to run completely unmodified so that holistic system integration can be dramatically reduced. ARMv8 processors such as Zynq UltraScale+ MPSoCs allow developers to harness next-generation capabilities and preserve their existing software investment. For more information about Zynq UltraScale+ MPSoCs, go to xilinx.com. Also review ARM's official porting guide located here: https://community.arm.com/docs/DOC-8453

References

1. Jason Fitzpatrick, "An Interview with Steve Furber" *Communications of the ACM*, Vol. 54 No. 5: http://cacm.acm.org/magazines/2011/5/107684-an-interview-with-steve-furber/fulltext
2. David Brash, "ARMv8-A Architecture Evolution" *ARM Connected Community*, 5 Jan 2016: https://community.arm.com/groups/processors/blog/2016/01/05/armv8-a-architecture-evolution

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
04/11/2016	1.0.1	Updated title.
03/31/2016	1.0	Initial Xilinx release.

Disclaimer

Automotive Applications Disclaimer