Chapter 15
Systems-on-Chip on FPGAs

Jeffrey Goeders, Graham M. Holland, Lesley Shannon, and Steven J.E. Wilton

This chapter provides an overview of SoCs on reconfigurable technology. SoCs are customized processing systems, typically consisting of one or more processors, memory interfaces, and I/O peripherals. FPGA vendors provide SoC design tools, which allow for rapid development of such systems by combining together different IP cores into a customized hardware system, capable of executing user-provided software. Using FPGAs to implement an SoC provides software designers a fabless methodology to create and tailor hardware systems for their specific software workloads. The vendor tools support a vast range of system architectures that can span from small embedded microcontroller-like systems to multiprocessor/Network-on-Chip architectures.

This chapter describes the advantages and limitations of designing SoCs on reconfigurable technology, what is possible with modern FPGA vendor SoC design tools, and the main steps in creating such systems. The SoC development tools for FPGAs are rapidly changing. As such, this chapter intentionally does not contain step-by-step instructions; instead, it focuses on the overarching concepts and techniques used in the latest SoC tools.

J. Goeders (✉) • S.J.E. Wilton
Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC, Canada
e-mail: jgoeders@ece.ubc.ca

G.M. Holland • L. Shannon
School of Engineering Science, Simon Fraser University, Burnaby, BC, Canada

D. Koch et al. (eds.), *FPGAs for Software Programmers*,
DOI 10.1007/978-3-319-26408-0_15

15.1 Challenges and Opportunities for SoC on Reconfigurable Technology

A SoC is a complete hardware processing system, often including one or more processors, memory controllers, and I/O interfaces. Each processor executes user-provided software, which can perform computational tasks and manipulate the operation of many different I/O devices. By mapping these architectures to an FPGA, the software designer has the opportunity to create a *custom* system platform while remaining *fabless*. Furthermore, designing and deploying their system on an FPGA enables them to repeatedly modify the system architecture as needed. This flexibility empowers software designers to create hardware tailored specifically to the software workload that it will execute.

SoCs on FPGAs are typically designed as a set of hardware blocks, connected together, as shown in Fig. 15.1. The hardware blocks, commonly referred to as IP (intellectual property) cores, each provide a piece of functionality to the system, and the interconnect represents wires and buses that allow the blocks to communicate. The vendor design tools provide a library of IP cores, including processors, memory controllers, communication ports, multimedia interfaces, and more. Users are able to select and connect these components through a graphical interface, allowing them to quickly create a system with the key elements they require. This design abstraction allows users to rapidly build processing systems, ranging from simple designs with a light-weight processor and UART, to large systems with multiple processors, fast I/O, and multimedia capabilities.

FPGA-based SoCs offer many advantages over both custom digital hardware circuits, and traditional x86-based processing systems. They key strengths of these platforms are:

Rapid Fabless Design Vendor provided IP libraries allow users to create an SoC very quickly; blocks can be added with a few mouse-clicks, and the communication interconnect is automatically handled by the design tools. In a matter of

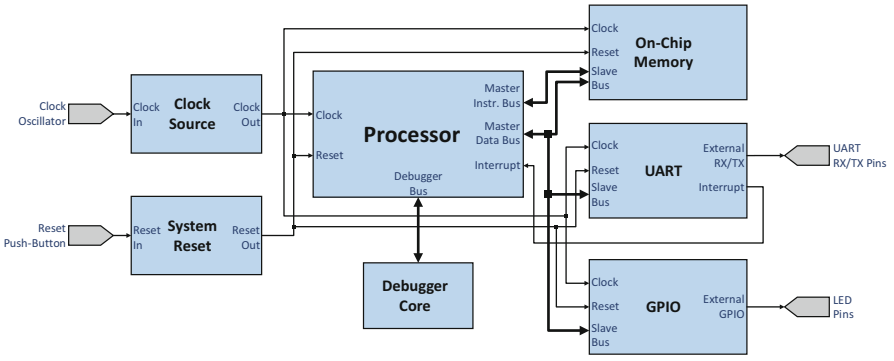

Fig. 15.1 Basic FPGA-based SoC

minutes, a complete hardware system, with processor, memory, and I/O can be created. This reduces design time over full custom hardware designs, lowering costs, and providing a fast time-to-market.

Flexibility FPGA-based SoCs can be made as small and simple or as large and complex as needed. Simple designs for embedded solutions can be developed and implemented on small, inexpensive FPGAs; or large systems with many processors can be created, targeting high-capacity FPGAs. This flexibility means the user only pays to deploy and power the minimum features required. This can reduce both production and operating costs over x86-based processing systems, while limiting non-recurring engineering costs if a redesign is needed.

Reconfigurability The reconfigurable nature of FPGAs means that the processor systems can be altered as needed, even after deployment. If software workloads change, the entire hardware system can be reconfigured. Designers can also incorporate hardware accelerators—custom compute units—into the SoC on the reconfigurable fabric of the same die, facilitating their integration. In fact, there is also the potential to create a Reconfigurable system-on-chip (RSoC), where the system's underlying architecture (e.g. the hardware accelerators) are reconfigured during the system's operation to support additional services.

These numerous advantages are offset by a set of limitations and challenges. For example, compared to high performance x86-based processor systems, the operating frequency, and thus performance, may be much lower. The flexibility that FPGAs offer comes at the cost of much lower clock speeds, and SoC designs implemented only using an FPGA's reconfigurable fabric generally operate in the range of 100 s of MHz. Although some modern FPGAs offer embedded ARM processors, running at 1–1.5 GHz, the processing performance will still be lower than modern x86-based systems. Despite this difference in clock speeds, it is still possible to obtain high performance SoCs on FPGAs, even beating the performance of x86-based systems, by leveraging the opportunities for a custom platform that better leverages spatial and temporal parallelism, through multiple processor systems and/or using hardware acceleration [TPD15].

Another potential drawback of using an FPGA-based SoC, versus a traditional x86 system, is the software environment. Although FPGA vendors make it easy to generate the underlying hardware for an SoC, the native software environment is generally "bare metal". In other words, there is no operating system, no filesystem, limited driver support, no TCP/IP support, and designers are often limited to programming in C or C++. Although all of these limitations can be overcome, by expanding the SoC, this may require greater expertise, and take significant time to design/integrate, implement and debug within the system. Conversely, x86 systems offer these features out of the box.

The remainder of this chapter describes the process for creating an FPGA-based SoC, comprising both the hardware system and the accompanying software. The hardware design is described in Sects. 15.2 and 15.3, with the former describing the process of creating a bare-bones, processor-based SoC using both Altera and Xilinx design tools, and the latter outlining how this basic system can be expanded

to interface with the many peripherals that are often offered on FPGA boards. Section 15.4 describes how to program these systems, detailing the software development process from a basic bare-metal approach, to a full operating system. Section 15.5 outlines the process for expanding a system to contain multiple processors, and outlines the considerations and limitations of such an approach.

15.2 Basic Hardware System

This section describes the setup of a very basic FPGA-based SoC design, consisting of a processor running software, a UART to provide stdin/stdout communication with the system, and a GPIO to control LEDs.

15.2.1 Basic SoC Design

Figure 15.1 provides a conceptual diagram of a basic SoC design. The blue boxes represent instances of IP cores, and provide the main functionality of the design. The thin lines show single wire connections and are used to propagate signals between blocks, such as clock and reset signals. The wider lines represent buses, which provide communication between cores using standard bus protocols. The grey pentagons at the far left and right of the design represent external connections off the FPGA. They are used to connect to specific FPGA pins, which are wired to different devices on the printed circuit board on which the FPGA is mounted. The following describes the purpose of each IP core for the basic system shown in Fig. 15.1:

Processor At the heart of the system is a processor, which executes the software and is (usually) responsible for managing the flow of data throughout the system. This may be a soft processor that is implemented using reconfigurable logic on the FPGA; or a hard processor that is embedded into the FPGA's silicon (see Sect. 15.2.3).

Local Memory Connected directly to the processor is a local memory. This is an on-chip memory, meaning it uses memory resources within the FPGA, versus using a separate memory chip on the circuit board. This memory can be used for storing both the program instructions and data assuming they do not require too much memory. This option is preferential when possible as it has extremely low memory latency, often a single clock cycle. As shown in the figure, the instruction and data buses from the processor are connected to the memory.

Debug Module The debug module is connected to the processor and is necessary to enable interactive debugging of the software from a connected workstation.

Clock Source This core contains a clock input, which is driven from a pin on the FPGA, usually connected to a clock oscillator on the circuit board. The core is responsible for propagating the clock signals to the rest of the system, and

ensuring that they are properly synchronized throughout the system, handling issues of clock skew and jitter.

Reset This core takes a reset input, often from a push-button on the circuit board, and forwards it to all components in the system.

UART This core provides logic to handle a serial UART connection, which can be used to provide stdin and stdout for the software running on the processor.

GPIO The General Purpose Input/Output (GPIO) core allows the processor to get and set the value of external pins. In Fig. 15.1, the core is connected to LEDs on the circuit board, allowing the processor to control the LEDs.

Bus Interconnect The processor communicates with IP cores through bus transactions. In the system shown, the bus connects the master device (the processor) to multiple slave devices (the UART, GPIO, and on-chip memory). This allows the processor to provide data to the other IP cores, such as sending UART data, or request information back, such as reading memory values in the on-chip memory. This is described in greater detail in Sect. 15.2.2.

Altera and Xilinx both offer SoC design tools to target their respective FPGAs. Although the tools differ in their visual style, they both offer similar core functionality. The design tools provide Graphical User Interfaces (GUIs) that allow a user to build a system that looks very similar to Fig. 15.1; IP blocks can chosen from vendor supplied IP libraries, and connections can be made with a few mouse clicks.

Creating the Basic System with Xilinx Vivado

This section describes creating the basic system from Fig. 15.1 using the Xilinx Vivado tool flow (version v2015.2). This system can be quickly set up by creating a new project, and selecting the *Base MicroBlaze* example project.

The created system will appear similar to Fig. 15.1, and should contain the same IP cores. There should also be one extra core, the *AXI Interconnect*; this core provides the logic for the bus transactions between the processor and slave devices (UART and GPIO). This functionality is implied in Fig. 15.1; however, the Vivado tools show this logic explicitly.

During project creation, the user must choose from a set of preconfigured FPGA boards. By doing so, the external connections are automatically bound to the appropriate pins on the FPGA, and the IP cores are configured as needed. For example, if the Virtex-7 VC707 board is chosen then the following will automatically be set: (1) the clock input will be connected to a 200 MHz oscillator on the circuit board, and the clock source IP core will be configured to scale this down to 100 MHz, (2) the reset input will be connected to a push-button on the circuit board, and (3) the I/O pins for the UART and LEDs will automatically be connected to appropriate pins. To use an FPGA board not in the preconfigured list, first choose a supported board to create the system, then alter the configuration options as appropriate.

Once the system is created and configured, it must be synthesized before it can be placed on the FPGA. The *Generate Bitstream* command in Vivado will perform the process of translating the SoC design to an FPGA bitstream. This process can take several hours for large, complex, SoC designs; however, for this simple example it should take only a few minutes. Once this is complete, the *Hardware Manager* tool can be used to connect to the FPGA and configure it to implement the SoC circuit. After this is complete, the FPGA will contain a full processor system; however, it will sit idle unless it is given software to execute, which is described later in Sect. 15.4. If at any point the SoC design is modified in Vivado, the synthesis and programming process will have to be repeated.

Creating the Basic System with Altera Qsys

This section describes creating the basic system using the Altera Qsys tool (version 15.0). The visual format differs slightly from Fig. 15.1; rather than a block diagram, the IP blocks are shown in a list format, each with a sub-list of available connections. Connections between blocks are shown in a matrix format, and connections can be made by clicking the intersections of the wires.

Although the tool does not include a pre-configured example system, the system shown in Fig. 15.1 can be created with a few mouse clicks. This is done by locating the *Clock Source*, *NIOS II Processor*, *On-Chip Memory*, *UART*, and *Parallel I/O* cores from the *IP Catalog* and adding them to a new (empty) design. After the cores are added, the connections should be made as shown in the figure, with the following exceptions: (1) in Qsys, the *Clock Source* core handles the connections for both the clock and reset signals, and (2) the debug module is automatically contained within the processor, and is not displayed as a separate block.

External connections are added using the *Export* column. In our example design, this includes the clock input, reset input, UART interface, and LED connections. In Qsys, these external connections are not automatically connected to the appropriate FPGA pins. Instead, the Qsys design must be imported into the Quartus II software in order to specify how these external connections are connected to the physical pins on the FPGA. This requires creating a hardware description language (HDL) wrapper around the Qsys system, and using the Quartus II software to add pin constraints. This process does require some familiarity with the hardware design tools, and further documentation can be found in the Quartus II handbook [Alt15c].

Performing synthesis using the Altera tools is a two-step process. First the *Generate HDL* command must be issued in Qsys, which will translate the SoC high-level description to a digital circuit description. Next, Quartus II is used to generate an FPGA bitstream. Again, complex designs may take hours to synthesize; however, this basic system should only take a few minutes. Once complete, the *Programmer* tool is used to program the FPGA with the hardware bitstream. If the user changes the design in Qsys, the entire synthesis process must be repeated.

Table 15.1 Address space configurator provided by FPGA Vendor SoC tools

Device	Address
Processor instruction bus	
↪ On-chip memory	`0x4000-0x7FFF`
Processor data bus	
↪ On-chip memory	`0x4000-0x7FFF`
↪ UART	`0x0000-0x001F`
↪ GPIO	`0x8000-0x8003`

15.2.2 Address Space Layout

In SoCs, the primary method for IP cores to communicate is through memory-mapped bus transactions. An IP core that initiates these transactions is referred to as a *master* on the bus, and a core that waits and responds to requests is referred to as a *slave*. A master can provide data to a slave via write operations, and can retrieve data via read operations. To facilitate this memory-mapped system, each device on the bus must be assigned its own unique region in the memory space.

Table 15.1 provides an example of an address space configuration, similar to the editors provided in the Vivado and Qsys tools, for the system in Fig. 15.1. The following provides a description of this example address space, assuming Altera IP cores are used:

On-Chip Memory This memory is a 16k on-chip memory, and thus requires a 16k address space. As shown in the Address Map, it is assigned the range 0x4000–0x7FFF. If the processor performs a write to address 0x4008, the bus logic will recognize that this request falls within the address space of the on-chip memory and propagate the request to the on-chip memory controller, which will write the specified value to address 0x0008 in the memory.

UART The communication interface with the UART consists of six 4-byte registers, so it is assigned a 32 byte address space (rounded up to the nearest power of 2), located at 0x0000–0x001F. The processor accesses data received by the UART by reading from the first register address 0x0000, and can send messages by writing to address 0x0004. The other four registers are used for status and configuration.

For most IP cores, this low-level communication is handled by the included software drivers; however, for some IP cores it may be necessary to interact at the register level (see Sect. 15.4). It does not matter where in the overall system address space each core is located; however, the designer must make sure that (1) no IP cores overlap in the address space, and (2) each IP core is assigned a large enough address space to handle its functionality. Both Vivado and Qsys provide menu options to automatically assign the addresses to the IP cores.

15.2.3 Hard vs. Soft Processors

The previous examples of SoCs on Xilinx and Altera FPGAs both used soft processors. A soft processor is implemented using the standard logic resources on an FPGA. The other option is to use hard processors, which are embedded into the silicon during fabrication of certain models of FPGAs. For example, Altera offers an *SoC* variant of their FPGAs, such as the *Cyclone V SoC*, *Arria V SoC*, *Arria 10 SoC*, and *Stratix 10 SoC*, which includes a hardened ARM multicore processor. Likewise, Xilinx offers the *Zynq* and *Zynq Ultrascale* line of FPGAs, also with a multicore ARM processor. In both Vivado and Qsys, the hard processor is added to the system using the IP catalog, in the same manner as a soft processor or other IP blocks. When using a hard processor, the user must provide a boot ROM that is used to initialize the hard processor and start up the system. This is typically connected to the system using a flash medium, such as an SD card.

There are several trade-offs between using hard and soft processors, primarily regarding performance vs. configurability, that is now discussed.

Performance

Soft processors are implemented using the highly reconfigurable fabric of FPGAs, which introduces overheads versus a hardened implementation of a processor. The operating frequency of a softcore processor depends on many factors, including FPGA generation, model, processor configuration, and utilization of the FPGA fabric. For the latest generation of FPGAs, the *maximum* operating frequency of the soft processors provided by the Qsys and Vivado tools is in the range of 165–469 MHz [Alt15b, Xil15a]. In contrast, the ARM cores provided on the latest FPGAs operate in the 1.0–1.5 GHz range [Xil15g, Xil15f, Alt14b]. In addition to operating at a higher frequency, the hard processors offer architectural advantages, providing greater performance, even when scaled to the same frequency. The soft processors offer performance in the 0.15–1.44 DMIPs/MHz range [Xil15b, Alt15b], while the hard ARM processors provide about 2.5 DMIPs/MHz per core [Xil15f].

Although hard processors offer superior performance, they are limited to the quantity that are fabricated onto the chip; whereas soft processors can be continually added to the design until there is no more room on the FPGA. This can provide performance advantages for a soft processor system, provided that the software workload can be sufficiently parallelized.

Configurability

The main advantage that soft processors offer is configurability. Options can be added or removed to trade off between performance and FPGA resource requirements. Removing options from the processor will likely lower performance.

For example, if the processor is configured without a floating point unit, but the user's software includes floating point instructions, the software versions of floating point operations will incur a significant penalty. However, it has added benefits, such as fitting on a smaller and cheaper FPGA, requiring less power, or being able to fit more processor instances on the same chip. Some of the configuration options available with the Altera Nios II and Xilinx MicroBlaze processors include:

- Adding instruction and/or data caches, and changing cache sizes.
- Adding specialized computation units, including hardware multipliers and dividers, and floating-point units.
- Adding a memory management unit (MMU), memory protection unit (MPU), or exception handler.
- Configuring the resources allocated for debugging, including the number of hardware breakpoints or memory watchpoints.
- Adding branch prediction, and configuring the branch prediction cache size.

The benefit of configurability is not just in the initial design phase; the reconfigurable nature of FPGAs allow these options to be changed even after production roll-out. All that is required is that the FPGA be reprogrammed with the new bitstream.

15.3 Expanding the Hardware System

This section describes how to expand the basic system from the previous section to include additional functionality, such as memory systems, I/O interfaces, and hardware accelerators.

15.3.1 Expanding Using IP Cores

SoC hardware designs are expanded by adding IP cores from the vendor supplied libraries. Behind the scenes, IP cores typically consist of two parts: a hardware description language (HDL) circuit, which implements the functionality of the core, and a software driver that facilitates the user's software communications with the hardware (described in Sect. 15.4). When the hardware system is synthesized, the HDL circuits from all IP cores are bundled together into a single large design, which is implemented on the FPGA using the programmable fabric.

Some FPGAs may contain permanent hard logic for a processor, DDR memory controller, PCIe interface, or other I/O interfaces; this is done to provide higher performance for commonly-used IP cores, and to meet timing requirements that

would be difficult to obtain using soft logic. In these cases, the IP cores are still added to the design in the same fashion, but instead of using the FPGA fabric, the SoC circuit will incorporate the existing hard cores on the FPGA. In many cases, the same version of the IP core from the library will be used for both hard and soft logic implementations; the soft logic version will automatically be used if the FPGA lacks a specialized hard core.

The vendor provided IP library is sufficient for most designs; however, if needed, IP cores can be obtained from other sources, such as:

- 3rd party IP cores. These can be obtained from online open-source repositories, or purchased as licenses from other companies.
- User-created hardware accelerator IP cores generated using the HLS tools or traditional HDL flow.
- Packaging the user-created SoC design into an IP core, for use in a hierarchical SoC design.

Some cores provided in the vendor libraries are not available in all editions of the design tools. For example, the lowest performance configuration of the soft processors are available with the most basic editions of the design tools, while the higher performance configurations require upgraded editions of the design tools.

To add an IP core to the design, first select it from the IP library, and then add it to the layout. Next, the connections need to be made to the rest of the system. Figure 15.2 shows the common connections that are available; note that not all cores will have all connection types. The connections should be made as follows:

Clock and Reset Most cores contain clock and reset signals, and usually these should all be connected to the system wide clock and reset signals. In some cases IP cores will require a specific clock rate, which may be different from the system clock (see *Clocks* in Sect. 15.3.3).

Buses Most IP cores have one or more bus connections to communicate with and transfer data to other IP cores in the design. Most commonly, the core will contain a memory-mapped slave bus connection that should be connected to the master port of the processor (for Xilinx Vivado, rather than connecting directly to the processor, the connection is made to the AXI Interconnect core, which is in turn connected to the processor). This allows the processor to communicate

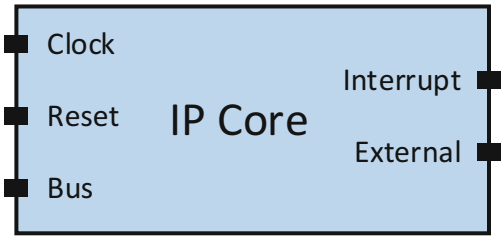

Fig. 15.2 IP core connections

with and control the IP core (usually through the provided software drivers). Although memory-mapped buses are common, there are also other bus types, such as data streaming, or first-in first-out (FIFO) style buses. For example, when processing video content, it is common to use streaming connections to feed video frame data from one processing IP core to the next.

Interrupt Some IP cores contain interrupt connections, which should be connected to the processor (or interrupt IP core, if separate from the processor). These connections allow the core to generate an interrupt request, which triggers the processor to halt execution and run the interrupt handler routine.

External External connections are used to connected to signals outside of the SoC. Typically these are connected to FPGA pins that are physically connected to various devices on the board (clock oscillator, Ethernet PHY, multimedia codec chips, etc.)

The connection types listed above, and their descriptions, are not exhaustive, nor without exception. The user should consult the documentation for each IP core when adding it to the system. The vendor tools make this easy, and the documentation can usually be accessed by simply right-clicking on the IP core and selecting the appropriate menu option.

IP Core Configuration

Most IP cores contain several configuration options. As examples, a UART core will usually allow you to change the baud rate or error detection schemes, and on-chip memory will allow you to configure the memory width and depth, number of ports, latency, and more. The configuration pages for each IP core are usually opened within the GUI when you add the IP to the design, but can also be accessed later if modifications are desired. Again, it is best to consult the documentation for each IP core for full details on the configuration options.

The IP core configuration options are not just on/off switches in a static circuit; rather, changing the options will actually alter the circuit that is implemented on the FPGA. If a feature is disabled, it will usually be removed entirely from the circuit; thus, changes to the configuration options will often affect the number of FPGA resources required by the IP core. For example, enabling large data and instruction caches on the processor will consume large amounts of on-chip memory, leaving less available for the main program memory. If trying to fit many processors on the same FPGA, one will have to decide between having *fewer* processors with large caches and specialized multiply/divide or floating-point units, versus having *many* simple processors.

15.3.2 The Memory System

In the bare-bones system from Sect. 15.2, on-chip memory resources are used for storing the program instruction and data. On-chip memory has very low latency, typically providing read and write operations with only one or two cycle delay. However, FPGA on-chip memory is relatively scarce, ranging from 100 s of KBs in the smallest FPGAs, to 10 s of MBs in the largest. In some cases, the memory may not even be large enough to store the program executable, especially when large software libraries are linked in. Even if the executable does fit, there may not be enough remaining memory to provide a sufficiently large stack and heap for the application.

DDR Memory

To provide a larger memory for both the program and its data, most SoCs contain an SDRAM memory controller that can access the DDR SDRAM memory module(s) on the FPGA board. When adding a memory controller, the user will be prompted to provide timing information for the SDRAM chip; this information can be found in the datasheet for the memory module. The FPGA vendor SoC tools already contain preconfigured timing information for certain FPGA boards and memory modules, so it is not always necessary to manually look up the information.

The SDRAM controller will also require a clock signal, running at a specific frequency range for the type of memory module used. Some tools offer a wizard when adding an SDRAM controller, which will also automatically add a clock generator IP core to your design, while other tools require you to do this manually. Refer to the documentation for the memory controller IP core for more detailed information.

Once the SDRAM controller is added to the system, the system needs to be modified so that the software executable is run from the newly added memory module. This is done by configuring the software linker, as described in Sect. 15.4. Finally, although SDRAM is the most commonly used memory type, the vendor libraries also contain IP cores for interfacing with other types, such as SRAM and Flash.

15.3.3 Commonly Used IP Cores

The vendor libraries offer IP cores for interfacing with a wide variety of I/O peripherals. These vary greatly in their complexity; some can be added to the system with a few clicks and include simple software drivers as their interface; others are

much more complex and may require expert configuration. The IP cores also vary greatly in software driver support, ranging from no software driver to large, complex libraries.

Basic I/O General purpose I/O (GPIO) cores can be used to get and set values of FPGA pins. In the basic example (Sect. 15.2), this was used to control LEDs; however, there are many other uses, such as reading the value of switches and push-buttons, or interacting with simple devices in an embedded environment.

Timers Timer IP cores can be added to the system to provide very high resolution (cycle-accurate) timers, which are useful for profiling and system scheduling.

Clocks Certain IP cores may require a clock input with a specific frequency; this is common for DDR memory controllers, audio and video interfaces, and other I/O devices. In such a case, a Digital clock manager (DCM) or Phase-locked loop (PLL) can be added to the system. These IP cores are able to derive a new clock signal with a specified frequency, using the system clock as input.

Ethernet Ethernet is a commonly-used method for providing high-speed communication between the SoC and other devices. The Ethernet IP cores are designed to support a very wide range of FPGA boards and system types and, as such, can be quite complex with many configuration options. However, since it is a widely used feature, one can often find online step-by-step tutorials for a specific FPGA board. Included with these cores is software to support the full TCP/IP stack; allowing the SoC to communicate with standard IP networks as both client and server.

DMA DMA cores perform memory transfer operations, either within the same memory, or between two different memories. The processor can request a DMA operation by providing source and destination memory addresses and transfer size, and the DMA core will carry out the memory copy operation while the processor continues on with other tasks. This allows the processor to execute other code in parallel with the memory operation, increasing performance.

Persistent Storage Many FPGA boards offer SATA, USB or Flash connections that can be used to interface with persistent storage devices. It may be straightforward to add IP cores to provide a hardware interface with such systems; however, often the complexity lies in the software. The software drivers may only provide primitive interfaces with the devices, leaving the complex task of file systems and communication protocols up to the user.

Multimedia FPGA boards offer a wide range of multimedia, including audio and video input and output through a variety of formats and connections. IP cores are provided to interface with many of these protocols, and in many cases basic processing (video scaling, cropping, etc.) can be performed using existing hardware cores without any intervention from the processor. This allows for high-performance media streaming and processing, while the processor performs other tasks in parallel.

Hardware Accelerators Most IP cores are designed to provide interaction with external I/O; however, another class of IP cores, hardware accelerators, use the FPGA fabric to implement hardware circuits that can perform a task faster than

the processor (or in parallel with it). Creating such IP cores usually requires hardware expertise; a HDL circuit that performs the acceleration is wrapped in logic that provides bus protocols for communication with the rest of the SoC. However, many modern high-level synthesis (HLS) tools (see Chap. 3), provide the ability to automatically create a hardware accelerator IP core from a software description of an algorithm, which could then be imported into the users SoC design.

PCIe Some FPGA boards offer PCI-Express (PCIe) interfaces so that the board can be inserted into a server system or workstation. PCIe allows a high-speed connection between the host (commonly an x86 processor) and the SoC; however, such systems are very complex compared to the simple SoC designs presented here.

15.4 Programming Environment

This section gives an overview of the tools available to software developers that allow them to write applications targeting SoC platforms, both in the case of bare metal and with the use of an operating system. Both Altera and Xilinx provide Integrated Development Environments (IDEs) that integrate with their SoC design tools to aid the developer in configuring a cross compilation toolchain, using libraries and vendor supplied IP core drivers, as well as downloading their code to the target platform and debugging.

15.4.1 Bare Metal Software

Depending on the requirements of an application, the performance overhead and memory requirements of running an entire operating system kernel may not justify the benefits gained. This is especially true of soft-processor-based SoCs, which trade performance per core for configurability. In these cases, a bare metal application development workflow may be more suitable.

In a bare metal environment, the abstractions commonly provided by an operating system, including process management, virtual address spaces, and file systems are unavailable. As a result, the user's programs have complete access to the underlying physical memory of the system, without any protection mechanisms. Since SoC platforms use memory mapped I/O, writing software to control different I/O hardware in the system is simply a matter of reading and writing appropriate memory addresses that get mapped to device registers. For this reason, the languages of choice for bare metal programming are those that support direct access to memory, such as C and C++.

To aid software developers, both Altera and Xilinx provide software IDEs (Altera EDS and Xilinx SDK) that integrate with their respective hardware design

tools. While there are differences between the IDEs, both are Eclipse-based and provide developers with tools for application development including: source code editing, build tools based on GCC, cross-compiler configuration, organization of applications into projects, board support package generation and customization, device programming and debugging. Since the soft processor targets within an SoC are highly configurable, compiler options must be correctly set to match the hardware configurations in order to build binaries that are compatible with a given configuration. For example, a soft processor may be configured to use a floating point unit, in which case it will be able to execute floating point instructions. The compiler must be passed an option to insert these floating point instructions into the output binary. This setting of compiler options is just one example of the automation that these vendor IDEs provide.

Board Support Package

After completing the hardware design for an FPGA-based SoC, the first step toward developing a bare metal application is to export a hardware description file from the hardware design tool. In the Altera flow, this is a *.sopc file*, and in the Xilinx flow this is a *.hdf file*. While the exact semantics vary between vendors, this file generally contains information about the hardware design relevant for software, including a list of IP cores in the design, their configurations and address space mappings. This hardware description file is read by the IDE and used to generate and customize a *board support package*.

A board support package contains a collection of libraries and device drivers for peripheral IP, and can be thought of as a minimal low level software layer. A board support package also contains a header file (system.h in Altera flow, xparameters.h in Xilinx flow), that defines constants related to the hardware system, for example interrupt constants, IP core base addresses, etc. These values may be required as parameters to IP driver Application Programming Interface (API) calls. Since these values vary dependent on the system configuration, the header file is automatically generated from the hardware description file for each system.

Both Altera and Xilinx allow a board support package to be customized after it is first created. Examples of the customizations this enables the user incorporate include: adding libraries, selecting which software drivers and which respective versions are included, selecting a device to use for stdin and stdout (e.g. UART or JTAG), and setting compiler flags. We now discuss the included libraries and drivers in more detail.

Libraries

The board support package contains at minimum an implementation of the C standard library, but other libraries can be included as needed. By default, the board support package will also contain some library code for accessing basic processor

features including caches, exceptions, and interrupts. The libraries that are available vary between vendors, but some examples include, bare metal TCP/IP stacks and file system libraries. More information about the available libraries for bare metal platforms can be found in [Xil15c, Alt15a].

Drivers

The task of writing software that interacts with various IP cores in the SoC design is simplified by the availability of drivers that exist for the majority of vendor library IP. The API level of these drivers varies for each IP block and between vendors. Lower level drivers typically export functions to access individual device registers and the programmer must handle manipulating register bits to enable the desired device functions, via masking and other means. Higher level drivers will usually provide a more programmer-friendly API that hides the hardware register interface from the programmer. As an example, a higher level UART driver may provide functions to set the baud rate and modify serial settings, while a low level driver would merely export functions to read and write the baud rate bits within a device register.

Documentation for the Altera and Xilinx IP driver APIs can be accessed by clicking on the IP block in Qsys and Xilinx SDK respectively.

Application Development

In order to create a new application, a new project can be created in the IDE. This encapsulation within a project allows the IDE to handle Makefile generation and setup paths to the various parts of the compiler toolchain. Upon application project creation, the programmer must select a board support package that the application will reference, as this allows application code to call into the libraries and drivers included therein. Multiple applications may share a common board support package. The Altera and Xilinx IDEs provide a number of sample applications varying from simple “hello world” programs, to more complex memory and peripheral test programs.

SoCs often contain multiple different memory regions, including local on-chip memory, external flash memory and DDR, for example. A programmer can decide which parts of different code sections (stack, heap, text and data, etc.) map to which memory regions by configuring the linker with a script. This linker script also defines the maximum sizes of the stack and heap memories.

Debugging

The Altera and Xilinx IDEs contain device programming features that allow a user to both configure an FPGA with a bitstream and download executable code to the

SoC conveniently from within the same interface. Additionally, these IDEs provide debugging capability, that will likely be familiar to users of Eclipse or other software IDEs. Software breakpoints, stepping through both C and assembly code, stack tracing, inspecting variables and viewing memory are all fully supported.

15.4.2 Operating Systems

While developing programs that run on bare metal is sufficient for many application domains, sometimes it is desirable to make use of the abstractions afforded by an operating system. This is particularly true if significant code is being reused that already targets an OS such as Linux, or makes use of standard libraries (POSIX threads, sockets, etc.). Building an operating system to run on a specific FPGA board/SoC platform is a more complex task than setting up a bare metal environment. However, a number of vendor supplied and third party tools exist to make the process of building a working kernel image for a specific FPGA board easier and less error prone. While a complete tutorial on configuring and building an OS image is beyond the scope of this book, we provide an overview of the steps involved and provide references to additional resources.

Linux is the operating system of choice for use on many embedded computing platforms. Both Altera and Xilinx have added architectural support for their soft processors (Nios II and MicroBlaze respectively) to the mainline Linux kernel within the last few years. However, this support is currently limited to single processor configurations only. More details on multiprocessor SoCs are given in Sect. 15.5.

Hardware Requirements

In order for an SoC platform to run a Linux operating system, certain IP cores must be present in the hardware system. These required IP cores, along with some commonly included optional cores, are described below:

Processor with MMU Since Linux provides a virtual memory system, the processor must be configured to use a memory management unit (MMU) to perform translation of virtual memory address to physical addresses.

Timer A hardware timer is required for the OS to manage software timers.

UART A UART is needed to provide a serial console to the Linux kernel for printing of messages and to provide the user with a command line shell.

External Memory An external memory such as DDR is required to store the OS binary and optionally the root filesystem. As a result, this memory must be large enough to fit these components.

Ethernet (Optional) While not required, including Ethernet in the hardware system enables the use of the network stack within Linux. If Ethernet is present,

the bootloader running on the SoC can be configured to download the OS image from a development host over a local network. Since Ethernet allows for high-speed data transfer this can save significant time during development.

Non-volatile Memory (Optional) Optionally, a non-volatile memory such as NAND flash, or an SD card controller may be included in the hardware system. The root filesystem can then be mounted to this memory to provide persistent storage. If this hardware is not included, the filesystem may be mounted in external memory.

Building and Booting an Operating System Image

The process of generating a bootable Linux operating system image for an SoC target is similar in many aspects to generating an operating system image for any other embedded CPU target. The main steps are outlined below:

- Setting up a toolchain
- Configuring and building a bootloader
- Configuring and building the Linux kernel
- Configuring and building a root filesystem
- Specifying a device tree

The first step is to obtain a cross-compiler toolchain for the target CPU that will be used to compile the various components of the operating system. If the target is a soft-processor based SoC, these can be obtained from Altera and Xilinx for the Nios II and MicroBlaze, respectively. Usually the easiest way to setup these toolchains is by installing the appropriate vendor IDEs, which are also used for bare metal application development.

U-boot is the de-facto standard bootloader for embedded targets, including FPGA-based SoC platforms. U-boot must be configured for the appropriate target and compiled as a bare metal application. In order to boot Linux, u-boot requires the kernel binary, a device tree blob (i.e. a binary description of the device tree), and a root filesystem.

While Nios II and MicroBlaze support is present in the mainline kernel repository, both Altera and Xilinx maintain their own kernel source repositories, which typically also include Linux device drivers for their library IP. After downloading an appropriate kernel tree, the kernel must be then configured and compiled.

As with other embedded targets, FPGA-based SoC platforms use a *device tree* data structure to specify the hardware configuration, including information about the CPU, physical address space, interrupts, etc. This device tree must be converted from its text format to a binary format (blob) using the device tree compiler from the toolchain. Lastly, one must build and populate a root filesystem.

In order to boot the system, the u-boot binary must be downloaded to the target device, and the kernel binary, device tree and root filesystem should be placed in the

board's external memory. This can be achieved via commands at the u-boot prompt. Once a Linux image has been built and booted successfully, users may wish to write the u-boot and kernel binaries, root filesystem and device tree blob to non-volatile memory, such as a NAND Flash or an SD card, if available on the platform. This way the system can be booted in the future without being connected to a development host PC.

Additional Resources

Building and booting an operating system for a particular FPGA board can be a complicated task. Fortunately there is a large amount of information available online, usually in the form of wikis and user guides, that provide step by step instructions on how to obtain, configure and build a Linux image for Altera and Xilinx development boards. Additionally, there are some vendor-supplied and third party toolkits that aim to make entire embedded Linux development process easier. These tools include Xilinx's PetaLinux Tools [Xil15d], Altera's SoC EDS suite [Alt14a], and Wind River Linux [Win15].

15.5 Multiprocessor Systems

In this section, we discuss multiprocessor systems for reconfigurable hardware. We describe how to construct an SoC with multiple processors, including the hardware required for interprocessor communication. We also give an overview of some of the unique aspects of writing software for multiprocessor systems including cache coherency and mutual exclusion.

As stated previously, the Nios II and MicroBlaze soft processors are simple RISC processors, with limited instructions per cycle (IPC) due to their relatively low operating frequencies because they are implemented using the logic resources of the FPGA. To overcome this limitation, a designer can often improve overall system performance by adding more processors to the system and dividing software tasks among them. Using a soft processor based system has an advantage in that the number of processors and each of their configurations can be tuned exactly to the application task requirements they will execute. For these types of systems, designers are limited in their choices only by the available logic resources of the target FPGA device.

15.5.1 Building a Multiprocessor System

Creating a hardware system with multiple processors is a fairly straightforward task in both Qsys and Vivado. Beginning with a basic single processor system, as

described in Sect. 15.2, additional processors may be added to the design from the IP core library in the same manner as any other IP core. At a minimum, connections must be made for each processor to valid clock and reset signals and at least one memory. In order to save resources, common peripherals, as well as connections to external memory are often shared between processors in a multiprocessor system. To implement this sharing in hardware, multiple processors can connect to the same bus interconnect, which in turn is connected to the shared peripheral/memory. When connecting multiple processors to the same bus, the design tools are generally capable of automatically adding the required bus arbitration logic, but this can be dependent on the type of bus interface being used.

Users can create symmetric multiprocessor systems by configuring each processor in the system identically. Alternatively, asymmetric topologies can be created by modifying the configurations for different processors in the system. For example, if a user knows that only one software task will require floating point operations, only one of the processors needs to be configured to include a hardware floating point unit, while the remaining processors will only support integer arithmetic.

Note that these multiprocessor systems are symmetric in that they have the same processor configuration. They would also share any off-chip memory system. However, their on chip local memories and caches have no coherency mechanisms that ensure that if one processor writes data to off-chip memory, the other processors will evict the same data from their caches as invalid.

Interprocessor Communication

Processors within a multiprocessor system need to communicate for the purposes of data sharing, synchronization and coordination of tasks. One method for achieving this interprocessor communication is through the use of a shared memory, as described above.

Another technique is to provide direct communication between processors with the use of a FIFO buffer memory. Both Qsys and Vivado, include FIFO IP cores in their libraries that may be added to a user design in the same manner as any other IP. FIFOs are directional, and have ports for connecting to a master and slave, for writing and reading to/from the FIFO respectively. When connecting two processors with a FIFO, one processor will connect as a master, allowing it to write data into the FIFO, and the other processor will connect as a slave, allowing it to read data from the FIFO. To enable bidirectional communication between processors, two FIFOs must be used.

Coherency and Mutual Exclusion

As suggested previously, two problems that arise when dealing with a shared memory multiprocessor environment are memory coherency and exclusive access

to memory. While creating a shared memory multiprocessor system for reconfigurable hardware is generally straightforward by following the steps in Sect. 15.5.1, ensuring cache coherency and exclusive access to memory is not easily achieved.

Implementing hardware to enable cache coherency protocols is a complex task that is beyond the scope of our discussion. Software developers must take this lack of coherency into consideration when writing software, as failing to do so so can lead to bugs that are difficult to diagnose and correct. As such, while an individual processor's cache maintains coherency with itself, software designers are better off viewing these caches as scratchpads that support hardware prefetching and writebacks to memory and use other mechanisms to maintain coherency system-wide when there are multiple processors.

For ensuring mutual exclusion, both Qsys and Vivado contain mutex IP cores in their libraries, that provide implementations of hardware mutexes. Such IP can be used to ensure exclusive access to memory and other peripherals if required.

Hard Multiprocessors

Although this discussion of multiprocessors focuses largely on soft processor based systems, as discussed in Sect. 15.2.3, both vendors also ship FPGA devices with hardened multicore ARM processors. Specifically, Xilinx's Zynq family of parts and the Altera *SoC* FPGA part variants all include embedded ARM cores. Generally speaking these hardened multiprocessors do have cache coherency protocols implemented in hardware as well as exclusive memory instructions.

15.5.2 Software for Multiprocessors

In the bare metal software environment, each processor in a multiprocessor system requires its own application binary that is specifically compiled to account for its specific processor configuration. As mentioned in Sect. 15.4.1, an application's referenced board support package encapsulates this processor configuration and sets up the compiler toolchain as appropriate. For this reason, the board support package must target a particular processor in the system. In the case of a single core system, this defaults to the only available processor, but in a multiprocessor system it may be necessary to use multiple board support packages, with each one targeting a different processor. Note that if the targeted processors have identical configurations, as would likely be the case in a symmetric multiprocessor system, then applications can share board support packages and all compiled program binaries should be portable between processors.

If applications targeting different processors within a system are to be located in a shared memory, developers must ensure that the code sections do not overlap unless this is specifically intended. This can be achieved by modifying the linker script which specifies address ranges for each of the code sections within an application binary. For example, each processor can use their local memory to store their

individual instruction code and then store any shared data in the shared system memory.

In a bare metal environment, programmers can write "multi-threaded" style code, where applications targeting different CPUs run in parallel and synchronize and share data through global variables. It is often necessary for a single processor to setup these shared variables and pass references to the other processors in the system. For such a task, interprocessor communication only through shared memory may be insufficient and FIFOs may be required to pass memory references between processors.

Operating Systems

Due to the lack of hardware to allow cache coherency and exclusive memory accesses, vendor Linux support is currently not implemented for multiprocessor systems built using either Nios II or MicroBlaze. Multiprocessor support may be added in future versions of the Linux kernel, however, this would also necessitate a change in the hardware IP used as well [MSF12].

Conversely, the hardened multicore ARM processors available in certain devices, are supported in Linux, since they have exclusive memory instructions and coherency protocols implemented in hardware.

15.6 Conclusions

This chapter provides an overview of how reconfigurable technology can be used to create custom SoC designs. It describes how the vendor tool flows support the quick generation of a basic hardware system through a graphical user interface. We have outlined the varied levels of software support available, ranging from bare metal systems to those with an operating system as well as third party support.

Currently, the main benefit of creating an FPGA-based SoC is the ability to create a custom hardware system for a specific embedded application. Software designers can start with a very basic system and then update and customize it as needed to include hardware accelerators and/or multiple processors with minimal non-recurring engineering costs. The FPGA provides a configurable substrate that allows software designers to remain fabless, while selecting between low cost, low power, and low performance soft processor based systems to ARM based systems, to systems that combine both hard and soft processors.

Looking forward, there are some significant opportunities for designing SoCs on reconfigurable technology. Although there currently is not vendor O/S support for SMP multicore systems, with the inclusion of multicore ARM processors in commercial products, there is an increased possibility for this extension. Even now, it is possible to update the open source Linux kernel for these platforms for those comfortable with editing kernel code. Another unique opportunity for FPGA-based SoCs is to leverage the dynamic partial reconfiguration support available from both

Altera and Xilinx. Dynamic partial reconfiguration is the ability to dynamically reconfigure the hardware substrate *while* the application is running. As such, there exists the possibility to incorporate this functionality into SoC designs on FPGAs to create *Reconfigurable* SoCs (RSoCs). While the hardware infrastructure needed to create RSoCs exists, the necessary vendor tool support still needs to be developed to make this accessible to non-FPGA experts.