arm

Cortex-M-based SoC Design and Prototyping using Arm DesignStart

Ashkan Tousi* and Xabier Iturbe
2018

*Thanks to our postgraduate interns, Mirko Gagliardi and Grigorios Kalogiannis who made substantial contribution to this work

Copyright © 2018 Arm Limited or its affiliates.

License

© Arm Limited 2018, all rights reserved. This document may only be used by a person authorized under and to the extent permitted by the applicable End-User License Agreement (EULA) issued by Arm Limited.

arm

Abstract

Current trends in autonomous driving, cyber-physical systems, Internet of Things (IoT) and Virtual Reality (VR) suggest a future where an enormous number of objects will be connected together and provided with context awareness to cooperatively anticipate and fulfill the needs and wishes of users.

Arm is devoted to accelerating the arrival of this bright future by developing necessary technology and addressing the most important challenges in power-efficiency, reliability, security, cost and form factor. Namely, Arm Cortex-M series CPUs are designed to meet the needs of tomorrow's smart and connected embedded applications and are part of the most widely adopted embedded ecosystem. Cortex-M0 and M0+ are intended for applications requiring minimal cost, power, and area while Cortex-M3, M4, and M7 target applications requiring higher performance.

This document is part of the Arm Research Enablement Kit on SoC Design and Prototyping. It is intended to help you design and prototype a System-on-Chip (SoC) based on the Arm Cortex-M0 CPU and AMBA standard using Arm DesignStart [1]. We first focus on the SoC design flow and how to integrate custom Intellectual Property (IP) cores into an Arm-based SoC. We then introduce Arm DesignStart, the simplest and fastest route to a custom SoC.

We show how to integrate two IP cores, namely Floating Point Unit and AES IP cores into a Cortex-M0-based SoC and peripherals provided by Arm DesignStart, and evaluate their performance.

Copyright © 2018 Arm Limited or its affiliates.

arm

Contents

License 2

Contents 4

1 SoC Design and FPGA Prototyping 5
- 1.1 Introduction 5
- 1.2 SoC Design 6
- 1.3 IP Integration Flow 7
- 1.4 Arm DesignStart 8

2 Custom IP Integration into a Cortex-M0 based SoC 13
- 2.1 Getting Started 13
- 2.2 APB and AHB Bus Wrappers 13
- 2.3 Software Drivers 14
- 2.4 FPU IP Integration 14
- 2.5 AES IP Integration 19

Summary 25

Acknowledgements 25

Bibliography 26

Copyright © 2018 Arm Limited or its affiliates.

1 SoC Design and FPGA Prototyping

1.1 Introduction

In the current post-PC era shaped by smartphones, tablets, wearables and cyber-physical systems, the primary metrics of silicon chips have changed from clock-frequency to cost, form-factor, and power consumption. On-chip integration of functional hardware is now more important than ever.

A System-on-Chip (SoC) is an integrated circuit that packages most of the necessary computing components into a single chip. These include: (1) a system master, such as a CPU or DMA controller; (2) system peripherals, such as memory blocks, timers and external digital/analog interfaces; and (3) a system bus that connects master and peripherals together using a specific bus protocol. More sophisticated components are also typically integrated in modern SoCs, such as hardware accelerators to offload heavy processing tasks from the CPU and speed up computation in specific applications. Such accelerators as well as other application-dependent components are known as Intellectual Property (IP) cores.

In addition to the obvious reduction in device size and weight, the advantages of SoCs are as follows:

- Higher performance as a result of less propagation delay introduced by shorter internal wires, and less gate delay provoked by lower impedance of on-chip transistors.
- Power efficiency as a result of lower capacitance of on-chip components, and lower on-chip voltage compared to the external chip voltage.
- Increased reliability as all components are contained within a single chip package and are thus less exposed to interference from the external world.
- Lower per-unit cost as a single chip design can be fabricated in large volumes.

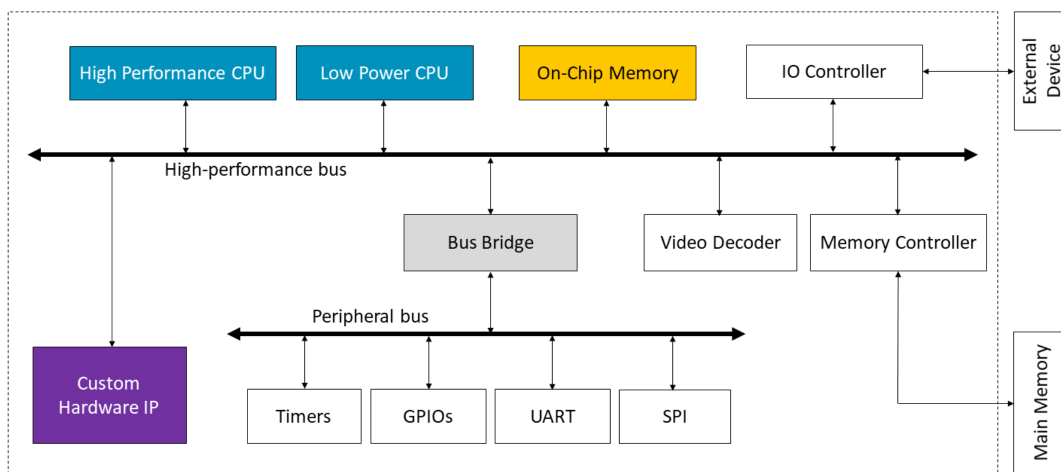


Figure 1: Typical System-on-Chip architecture

Nowadays, the architecture of SoCs is dictated by applications. A typical SoC, shown in Figure 1, consists of one or multiple CPUs and special purpose IPs with different data throughput, which are typically connected using a bus hierarchy. High data rate IPs and local memory use a high-performance bus, which usually supports data bursts and DMA transactions, while low data rate IPs are connected to a peripheral bus.

The CPUs may have a custom or a standard Instruction Set Architecture (ISA), such as Arm. Arm-based CPUs deliver high performance at low power consumption, and are of increasing interest as the building blocks of modern SoCs that are widely used in every kind of electronic devices, including state-of-the-art technologies for smart living and Internet of Things (IoT) applications. In this context, custom IP cores are emerging as a way to increase system performance and/or

Copyright © 2018 Arm Limited or its affiliates.

energy efficiency in SoCs. Therefore, more and more engineers and researchers are focused on designing IPs to target a wide range of applications, including digital signal processing, data streaming, graphics and cryptography.

This document comes along with a number of hardware and software resources, which together make up a Research Enablement Kit (REK). The objective of this REK is to ease the development of Arm Cortex-M-based efficient SoCs that include custom IPs and software components. This REK could be also of interest for designers willing to test their IPs in a complete SoC solution.

1.2 SoC Design

SoC design is mainly an integration process where designers connect a set of IPs together using standard buses to build their systems. IPs are validated, verified, documented and reusable building blocks, which allow designers to deal with the growing complexity of modern applications. Moreover, IP-based design improves productivity compared to a situation where systems have to be designed from scratch. As shown in Figure 2, the SoC design flow starts with identifying the target application needs, and ends with verifying that such specifications are fulfilled in a prototyped SoC, which is typically implemented on an FPGA prior to silicon tape-out. In order to meet the specifications in the most efficient way (i.e., energy-area-performance trade-off), SoCs are partitioned into hardware and software components which are developed by dedicated teams at the same time. The hardware team focus on system integration and interconnection, while the software team use a programmer's view model of the system before the SoC prototype is available [2].

This REK covers the SoC prototyping flow, as shown in Figure 2, but the silicon tape-out flow is beyond its scope.

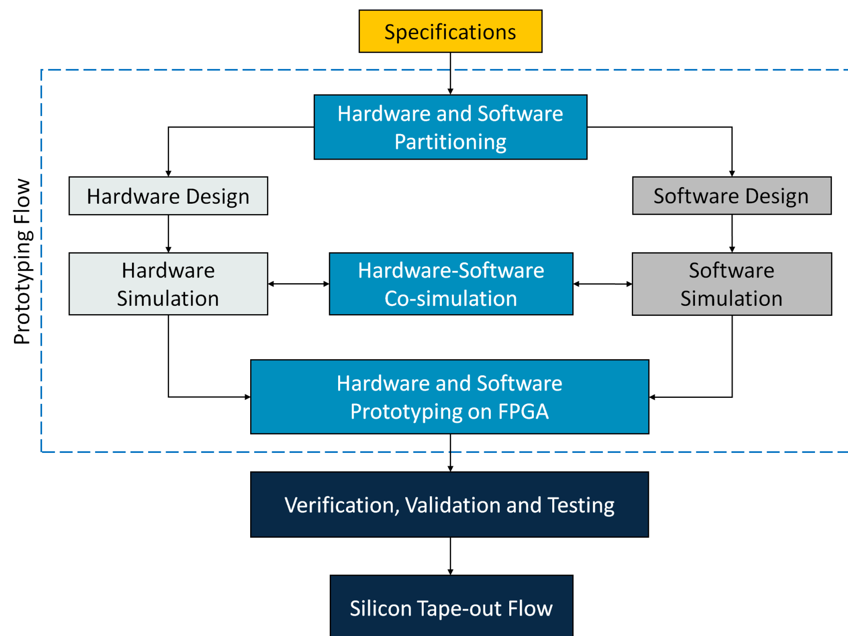


Figure 2: System-on-Chip design flow

An important aspect of SoCs is the memory map, which provides consistent physical addresses to all shared resources and system IPs. A uniform memory map allows the CPU to access the system components using standard memory instructions, and plays a central role in the programmer's view model used by software developers. Figure 3 shows a generic memory map for an Arm Cortex-M-based SoC, where there are dedicated memory regions for off-chip and on-chip resources. Application-dependent custom IPs are expected to be mapped to the on-chip peripheral region.

Note that the high-level memory map shown in Figure 3 requires buses to provide actual access to peripherals, custom IPs, devices and memories. Arm DesignStart [1] provides a ready-to-use SoC subsystem that implements this memory map and includes standard on-chip AMBA buses [3, 4] along with memory and device controllers, mastered by Arm's most power efficient Cortex-M CPUs. The Arm DesignStart is thus a good starting point to get up to speed in SoC development.

Copyright © 2018 Arm Limited or its affiliates.

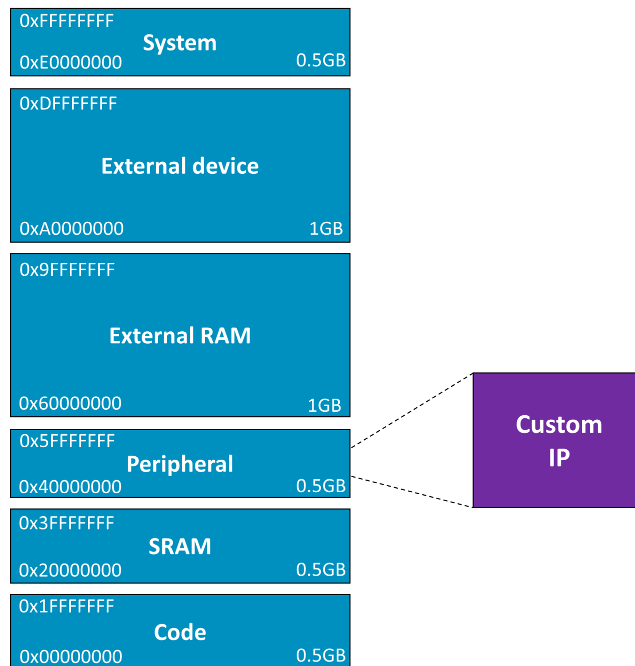


Figure 3: Memory map for an Arm Cortex-M-based SoC

1.3 IP Integration Flow

In this section, we introduce the proposed integration flow of a custom slave IP into an Arm-based SoC, which involves actions at the IP, system and software levels. The first decision to be made is to choose the system bus to which the IP should be connected. For example, computation-intensive accelerator IPs are usually connected to high-performance buses, whereas low-speed communication controller IPs are typically connected to peripheral buses.

1.3.1 IP Level

If the custom IP does not natively support the system bus interface, a wrapper needs to be created to make the IP's input and output ports accessible for the CPU through memory-mapped registers. As shown in Figure 4, these wrappers typically consist of four parts: bus and IP interfacing logic, storage and a control unit.

The bus interfacing logic deals with the system bus protocol and manages data transactions between the IP and the system bus. This is typically implemented using write and read multiplexers. When the bus data width does not match the width of IP data ports, storage needs to be added to keep data temporarily buffered (e.g., FIFOs). Storage is also used to hide high IP latencies, especially when used in conjunction with burst data transactions through the bus. In addition to this, storage is useful in IPs that need to access multiple times the same input datasets, or produce a significant amount of intermediate data while performing their computation (e.g. scratch-pad memories). The IP interfacing logic manages data movements between storage and the IP, dealing with different data widths and rates. The control unit ensures coordination among all parts in the wrapper, and typically implements status/control registers. As previously mentioned, these registers together with internal storage (or part of it) are made accessible by the IP wrapper through an addressable interface, to be mapped to the system memory space.

Copyright © 2018 Arm Limited or its affiliates.

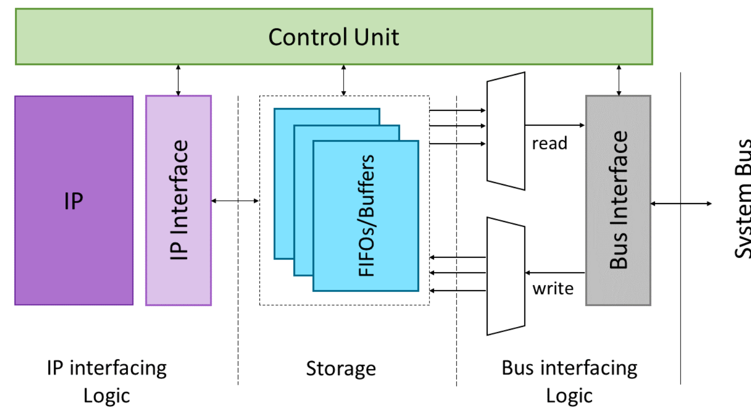


Figure 4: Custom accelerator wrapping

1.3.2 System Level

The IP wrapper signals are physically connected to other components in the SoC, and the addressable interface presented by the wrapper is mapped to the actual SoC memory space, usually within the peripheral region. That is, the IP is assigned with a base address in the SoC memory map. Signal connections are typically done with the bus arbiter, other master/slaves in the bus, and the interrupt controller (if needed). It is important to make sure that these connections do not violate any timing constraints in the system. Another important aspect to consider is that the IP and wrapper are provided with a clock signal that satisfies their individual timing constraints.

1.3.3 Software Level

The IP must be accompanied by a software driver that allows easy access to it without having to know its exact implementation details. The driver typically automates the IP initialization, configuration, monitoring, data transferring and fault handling. Interactions with the IP usually involves polling (i.e., periodic checks of the status register) or interrupts.

1.4 Arm DesignStart

Arm DesignStart packages provide free and quick access to industry-proven Arm processor IPs and subsystems, easing the process of designing and prototyping Arm-centric SoCs [5]. Namely, this REK deals with Arm Cortex-M0- and M3-based SoCs. The Cortex-M0 is the smallest and most-efficient 32-bit Arm CPU with only 12k gates at its minimum configuration, which allows the design of extremely small, low-cost SoCs. Likewise, the Cortex-M3 is an extensively optimized and power-efficient 32-bit CPU that provides high-performance processing capabilities, making it an ideal choice to power SoCs in demanding embedded applications.

At time of publication of this whitepaper, two versions of each DesignStart package are available. Arm DesignStart Eval provides instant access to the obfuscated Verilog RTL code of the Cortex-M0 and M3 CPUs with a click-through EULA. Free access to the full Verilog code is also available with the DesignStart Pro under a very flexible license that allows most research use cases.

As shown in Figure 5, both Cortex-M0 and M3 CPUs come integrated into a reference SoC to allow quick and easy prototyping and evaluation of an Arm-based solution on an FPGA. The reference SoC includes a number of essential peripherals, such as GPIO, timers, watchdog and UART. These are interconnected using standard AMBA on-chip buses, which ease the integration and test of custom IPs. High-performance IPs are expected to be connected to the AHB bus, whereas IPs that require low bandwidth data exchanges are expected to be connected to the APB bus. Note that the Arm CPUs can also be used alone and instantiated in custom SoCs to create ad-hoc multi-core systems and experiment with new inter-core connectivity and hardware acceleration solutions.

Copyright © 2018 Arm Limited or its affiliates.

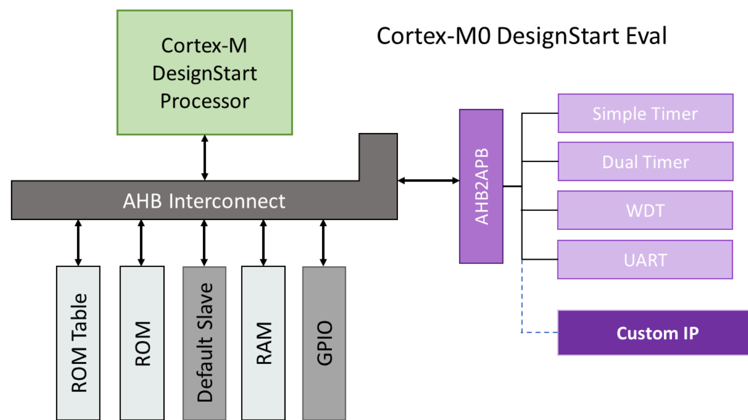


Figure 5: Arm Cortex-M DesignStart Eval

Although the reference SoC can be implemented in any prototyping board populated with an FPGA, this REK uses the Arm Versatile Express V2M-MPS2+ prototyping platform [6]. The latter platform is equipped with an Altera Cyclone FPGA and a Motherboard Configuration Controller (MCC) that hides low-level operations in the board. The MCC handles the bitstream loading on the FPGA and the software image on SRAM memories. The latter FPGA bitstream and software images are kept on a microSD card from where they can be easily accessed by the user.

1.4.1 Cortex-M0 DesignStart Eval

Table 1 shows the structure of the Cortex-M0 DesignStart Eval package. The `Recovery` directory includes a pre-built image for the FPGA (i.e., bitstream), the BIOS and several software test code images. The `RevC` directory contains a Quartus project including top-hierarchy Verilog files. The `cores` directory contains the obfuscated Cortex-M0 integration Verilog code, while the `logical` directory includes Verilog-based Cortex-M System Design Kit (CMSDK) components (such as AHB and APB buses, GPIOs). The `smm_common` contains Verilog files that describe common FPGA components (e.g. PLLs) and peripherals (such as memories) and the `software` directory contains the Cortex Micro-controller Software Interface Standard (CMSIS)[7] header and source files required to build software images for the Cortex-M0 along with some software examples. The `systems` directory includes design and testbench files for the FPGA system. More detailed information is available in the DesignStart Eval user guide [8].

Table 1: Cortex-M0 DesignStart Eval package file structure.

`Directory name`	**Directory contents**
`Recovery/`	This directory includes a pre-built image for the FPGA, the BIOS and software test code images.
`RevC/`	Quartus project including top-hierarchy Verilog files
`cores/`	Obfuscated Cortex-M0 Integration Verilog files.
`logical/`	Cortex-M System Design Kit (CMSDK) Verilog files.
`smm_common/`	Verilog files that describe common FPGA components and peripherals
`software/`	Software examples and demos along with the CMSIS header files.
`systems/`	Testbench files, and simulation setup files for the CMSDK and FPGA example system.
`documentation/`	Documentation files.

Copyright © 2018 Arm Limited or its affiliates.

1.4.2 Cortex-M0 DesignStart FPGA Design Overview

Figure 7 shows the Cortex-M0 DesignStart system, which runs at 25MHz on the MPS2+ board, and Figure 6 shows its memory map. The top module is the `fpga_top`, which wraps and interconnects PLLs and other supporting logic and allocates a pass-through level, named `fpga_system`, where the main User Partition is located. This contains the AHB and APB subsystems, instantiates the CMSDK subsystem and connects the ports required to interface the devices present on the MPS2+ board. The latter `cmsdk_mcu_system` module is a simple micro-controller design using an obfuscated version of the Cortex-M0 CPU. Finally, the `fpga_apb_subsystem` module implements the APB subsystem and includes some additional peripherals which might be useful in certain applications.

Purple boxes in Figure 7 show two placeholders for user IPs. AHB slave IPs are to be added in the CMSDK subsystem while APB slave IPs can be added in the FPGA APB subsystem.

Start address	End address	Region	Size
0x00000000	0x0000FFFF	64K flash / boot firmware	64KB
0x00001000	0x00FFFFFF	unused	
0x01000000	0x0100FFFF	boot firmware : only 4k is used	64KB
0x01010000	0x1FFFFFFF	unused	
0x20000000	0x207FFFFF	SRAM	8MB
0x20800000	0x3FFFFFFF	unused	
0x40000000	0x4000FFFF	APB subsystem	64KB
0x40010000	0x4001FFFF	AHB peripherals	64KB
0x40020000	0xEFFFFFFF	unused	
0xF0000000	0xF0000FFF	System ROM Table	4KB

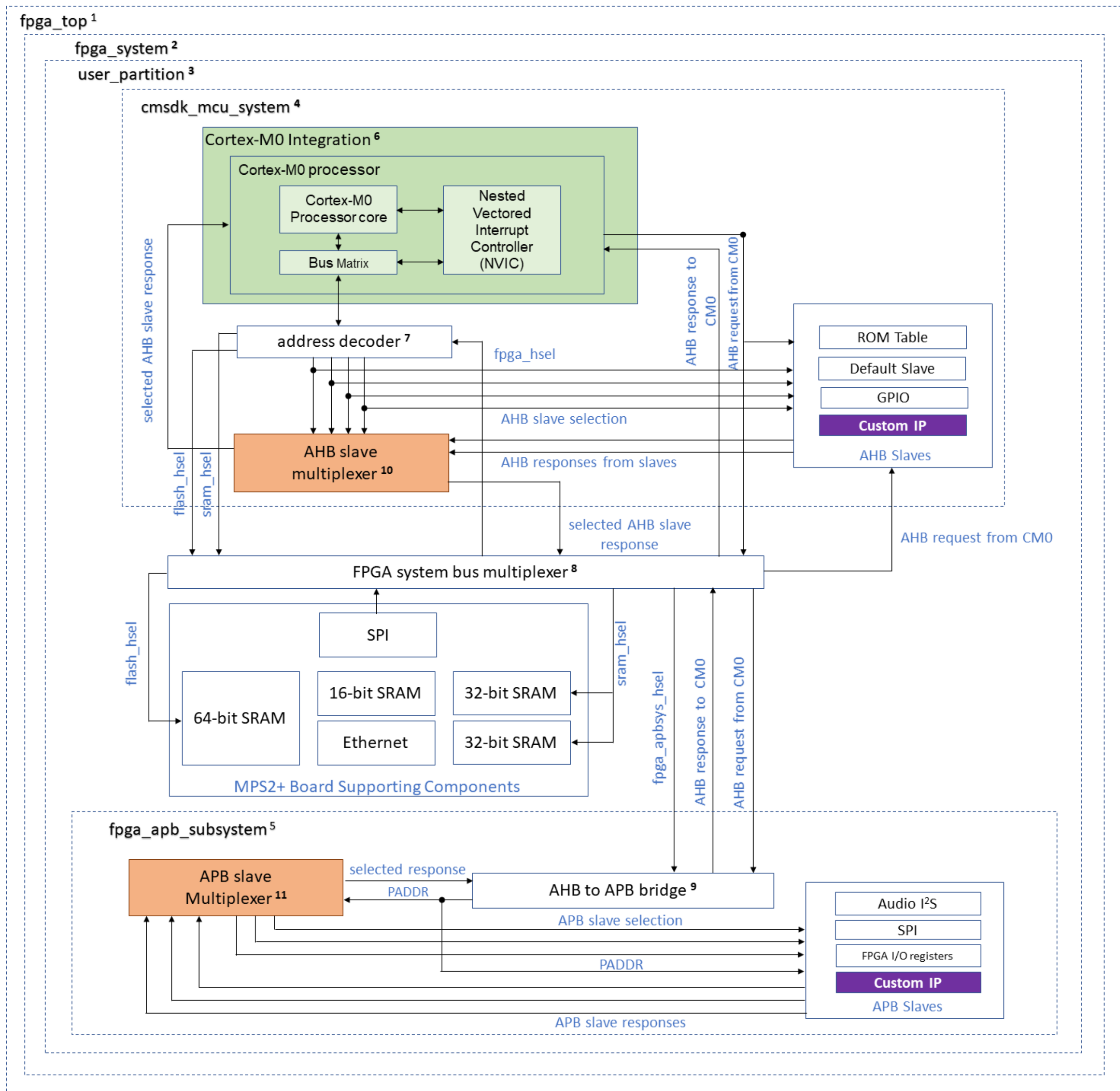
Figure 6: Memory map of the Cortex-M0-based DesignStart system

CMSDK Subsystem

The `cmsdk_mcu_system` instantiates an address decoder, an AHB slave multiplexer, AHB slaves, and the Cortex-M0 CPU, together with an interrupt controller and a bus matrix, as shown in Figure 7. The latter bus matrix arbitrates the processor core memory accesses to both the SRAM and to the internal interrupt controller. The address decoder activates the appropriate selection signal in the target AHB slave based on the address issued by the Cortex-M CPU. The AHB slave multiplexer reads the selection signal and forwards the selected AHB slave response back to Cortex-M0. If the requested address is in the FLASH memory space, then the decoder enables the `flash_hsel` signal and the request is directly forwarded to the corresponding SRAM controller in the user partition. Alternatively, if the requested address falls in the APB subsystem memory space, then the `fpga_apbsys_hsel` signal is activated and the AHB to APB bus bridge is enabled. These accesses are handled as described in the FPGA APB subsystem section below.

In the event of adding a custom high-performance AHB slave IP, the user should modify input connections in the AHB slave multiplexer and update the corresponding Verilog parameters accordingly. This is explained in detail in Chapter 2.

Copyright © 2018 Arm Limited or its affiliates.



	Module name	Path and File Name
1	fpga_top	\RevC\SMM_M0DS\fpga_top\Verilog\fpga_top.v
2	fpga_system	\RevC\SMM_M0DS\fpga_top\Verilog\fpga_system.v
3	user_partititon	\RevC\SMM_M0DS\fpga_top\Verilog\user_partition.v
4	cmsdk_mcu_system	\RevC\SMM_M0DS\fpga_top\Verilog\cmsdk_mcu_system.v
5	fpga_apb_subsystem	\smm_common\verilog\fpga\fpga_apb_subsystem.v
6	Cortex-M0 Integration	\cores\cortexm0_designstart_r2p0\logical\cortexm0_integration\verilog\CORTEXM0INTEGRATION.v
7	cmsdk_mcu_addr_decode	\RevC\SMM_M0DS\fpga_top\Verilog\cmsdk_mcu_addr_decode.v
8	fpga_sys_bus_mux	\smm_common\verilog\fpga\fpga_sys_bus_mux.v
9	cmsdk_ahb_to_apb	\logical\cmsdk_ahb_to_apb\Verilog\cmsdk_ahb_to_apb.v

Obfuscated file

Modules that need to be modified when integrating an IP

User added Custom IP

	Module name	Instantiated in file
10	cmsdk_ahb_slave_mux	user_partition.v
11	cmsdk_apb_slave_mux	fpga_apb_subsystem.v

Figure 7: Cortex-M0 DesignStart FPGA System simplified block scheme

Copyright © 2018 Arm Limited or its affiliates.

FPGA APB Subsystem

The `fpga_apb_subsystem` is composed of an APB slave multiplexer, APB slaves and an AHB to APB bridge, as shown in Figure 7. When enabled, the bridge dispatches the `PADDR` signal to the APB slave multiplexer with the requested address by the Cortex-M0 CPU. The latter multiplexer enables the selected slave and waits for its response, which is forwarded by the bridge as an AHB transaction to the FPGA System bus multiplexer, and from there all the way back to the Cortex-M0 CPU.

In the event of adding a custom low-speed APB slave IP, the user should modify input connections in the APB slave multiplexer and update the corresponding Verilog parameters accordingly. This is explained in detail in Chapter 2.

1.4.3 Education and Research using Arm DesignStart

The Arm University Program provides free education materials around Arm DesignStart to help faculty members worldwide teach the latest concepts and trends in SoC design (developer.arm.com/academia/arm-university-program/for-educators/system-on-chip-design). These materials include lecture slides/notes, exercises and labs that can be freely edited/adapted to satisfy any particular University curricular agenda. The Arm University Program also donates licences of DS-5 and Keil MDK professional versions for use in education (www.arm.com/research-education/aup-donation-request-form). Educational support is also available in related areas such as Embedded System Design, Digital Signal Processing, Real-Time Operating Systems (RTOS) and IoT.

This section has discussed the conventional structure and use of the DesignStart reference SoC, which allows for quick and easy integration of custom IPs into an Arm-powered SoC. However, the Arm DesignStart comes with a very flexible license that allows and empowers researchers to explore and experiment almost any novel idea they might have. The fact that the Cortex-M0 is extremely small enables the integration of many CPUs into a single FPGA chip, which is very useful in the context of many research areas, including novel and unconventional SoC architectures, fault-tolerance, security, heterogeneous and parallel programming, IoT and edge computing, as well as hardware-software co-design and high-level programming.

Copyright © 2018 Arm Limited or its affiliates.

2 Custom IP Integration into a Cortex-M0 based SoC

In this chapter, we explore how to integrate custom IPs into an Arm-based SoC following the integration flow previously described in section 1.3. In order to illustrate this process, we use completely arbitrary open-source and easy-to-access IP cores, namely FPU and AES cypher IPs. This document uses Cortex-M0 DesignStart Eval, which provides a synthesizable processor pre-integrated into a customizable SoC subsystem.

2.1 Getting Started

This Research Enablement Kit (REK) uses a number of resources which are available as follows. The Arm Cortex-M0 DesignStart Eval package can be downloaded following the instructions described in [1].The target prototyping platform of this document is the Arm MPS2+ board [9], which is populated with an Altera Cyclone V FPGA. Hence, the Altera Quartus suite is needed in order to synthesize a SoC design for this FPGA [10]. In fact, the hardware and software resources provided in this REK are intended to be used in a Quartus project. For the software flow, the Arm Compiler 5 toolchain can be downloaded from [11]. Note that all software tools used in this REK as well as Arm DesignStart Eval are freely available from Arm and Altera.

2.1.1 File Structure of this Research Enablement Kit

This section describes the file structure of this REK, which is shown in Figure 8. The root folder includes the script that integrates the required hardware modules in the Quartus project. The `cm0_aes` and `cm0_fpu` folders contain `hw` (hardware) and `sw` (software) subfolders for AES cypher and FPU IPs, respectively. The `hw` subfolder includes the required hardware modules (e.g., IP bus wrappers), while the `sw` subfolder includes the demo application, drivers and additional useful files.

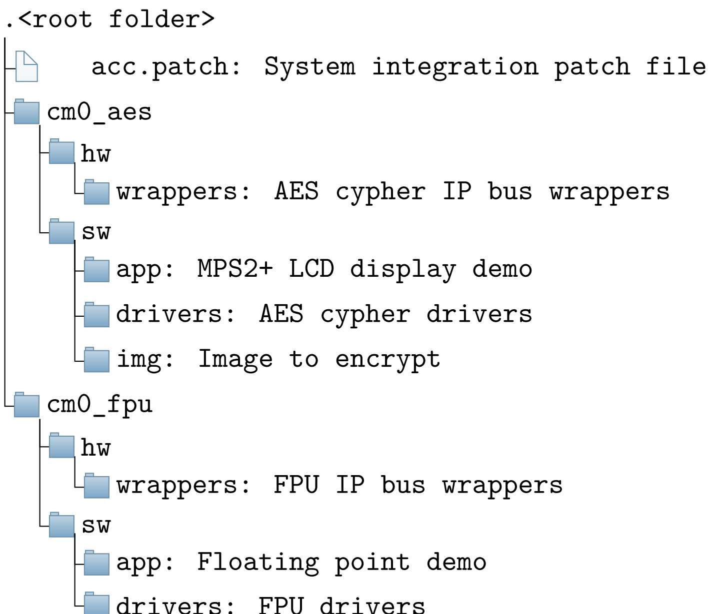


Figure 8: Research Enablement Kit file structure

2.2 APB and AHB Bus Wrappers

The REK provides two wrappers for IPs that implement both APB and AHB bus interfaces. The wrappers tie PSLVERR (APB) and HRESP (AHB) to '0' (i.e., OK), which means that instruction and data aborts are not supported. In this chapter

Copyright © 2018 Arm Limited or its affiliates.

the wrappers do not include any storage, and thus the bus interfacing logic is directly connected to the IP interfacing logic. A simple control unit coordinates the two interfaces and implements a status register. This provides a set of IP-specific flags as well as other flags that are common for most IPs, including availability to start a new computation (`ready`), validation of input data (`valid`), and completion of computation (`done`). In the particular case of AHB, the IP wrapper allows byte-access.

The wrappers handle bus transactions forwarded to the IP address space, which is shown in Figure 9. Note that in this figure, the status register is always mapped to local address 0, while IP's inputs and outputs are mapped to dedicated local memory addresses.

2.3 Software Drivers

This REK provides software drivers to the user for accessing the interface of both AES and FPU IPs. Access to the IP functionality is accomplished using a C structure that abstracts the IP interface and status register as shown in Figure 9. The volatile fields in this structure are arranged in such a way that its layout in memory reflects the local memory map of the IP wrapper thus the C structure must be pointed to the base address `IP_BASE_ADDR` assigned to the IP in the SoC. Note that the most likely type of structure variables are `uint32` since as smaller inputs, such as individual bits are typically mapped into a 32-bit word.

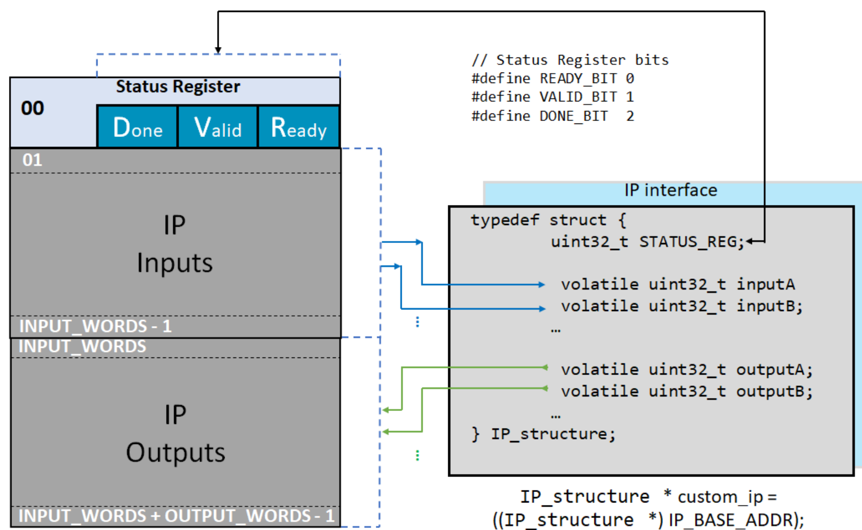


Figure 9: IP address space

2.3.1 IP Hardware Access

By polling the `READY` flag in the status register, the user waits until the IP component is ready. When that occurs, input data can be loaded to the IP by accessing the respectively input fields of the structure. Thus, the user can enable the IP and start the computation. The final result can be retrieved by reading the output fields of the structure, without checking the `READY` flag in the status register.

Note that since Cortex-M0 processor supports interrupts using the Nested Vectored Interrupt Controller (NVIC) [12], the IP component can be enabled by using hardware interrupts without the need to continually check the `READY` flag in the status register .

2.4 FPU IP Integration

This section deals with the integration of an open-source FPU IP that can be downloaded from [13]. As previously described , in 2.1 the Aletra Quartus suite is needed in order to synthesize a Soc design for Altera cyclone V FPGA. The SoC integration

Copyright © 2018 Arm Limited or its affiliates.

flow described in this section can also be used with the Altera FPU IP [14] available in the Quartus suite.

2.4.1 FPU IP Overview

The FPU IP implements IEEE-754 standard single-precision arithmetic operations (i.e., addition, subtraction, multiplication, division, and square root). It enables a significant performance enhancement and energy efficiency improvement when used with the Arm Cortex-M0 CPU, which lacks of hardware support for floating-point operations (i.e., floating-point operations are software emulated).

Figure 10 depicts the FPU IP interface. On the data side, `opa_i`, `opb_i` and `output_o` are single-precision floating-point 32-bit input and output ports, respectively. On the control side, `fpu_op_i` 3-bit input is used to specify the floating point operation to perform on the input data, as shown in Table 2, and `rmode_i` 2-bit input specifies the rounding mode. The FPU IP also provides exception outputs which are not described in this document. The `start_i` and `ready_o` ports are used to signal the beginning and completion of a floating point computation. That is, `start_i` is asserted when the inputs are ready in the `opa_i` and `opb_i` ports, and `ready_o` is asserted when the result is ready in the `output_o` port. This process is shown in the timing diagram in Figure 11. Table 2 shows the IP latencies for the different floating-point operations.

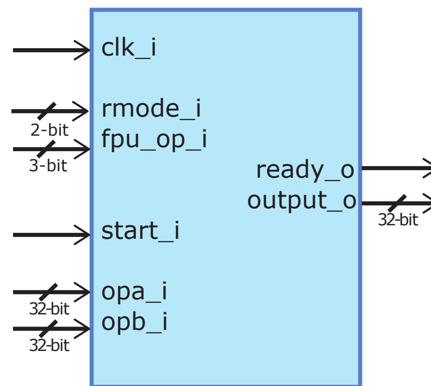


Figure 10: FPU IP interface

Operation	fpu_op_i	Latency (Clock cycles)
Add	0	7
Sub	1	7
Multiply	2	12
Division	3	35
Sqrt	4	35

Table 2: FPU IP operation codes and latencies

The FPU IP documentation states that the hardware design is pipelined and can run at 100MHz clock frequency, which is considerably higher than the frequency used in DesignStart Eval.

2.4.2 Customizing the APB Wrapper for the FPU IP

The FPU IP is connected to the APB bus, which is an unusual case for a hardware accelerator, but still serves the objective of showing how to integrate a simple IP in the peripheral bus.

As shown in Figure 12, the `start_i` and `ready_o` IP control ports are directly connected to `valid` and `ready` flags in the status register, respectively. The `done` flag is also connected to the `ready_o` port in the IP. The FPU IP-specific flags in the status register are connected to the floating-point exceptions ports. For the sake of simplicity, each individual input and

Copyright © 2018 Arm Limited or its affiliates.

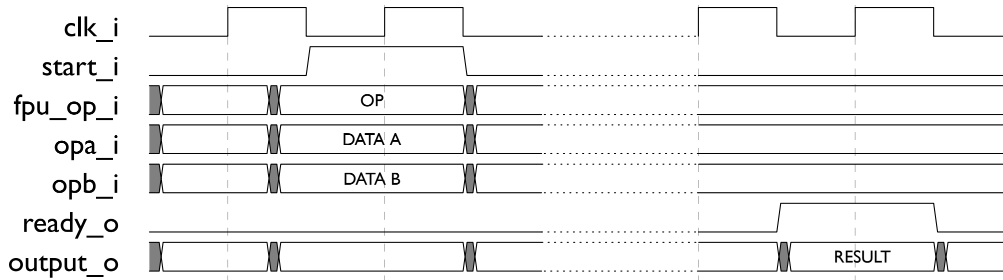


Figure 11: FPU IP timing diagram

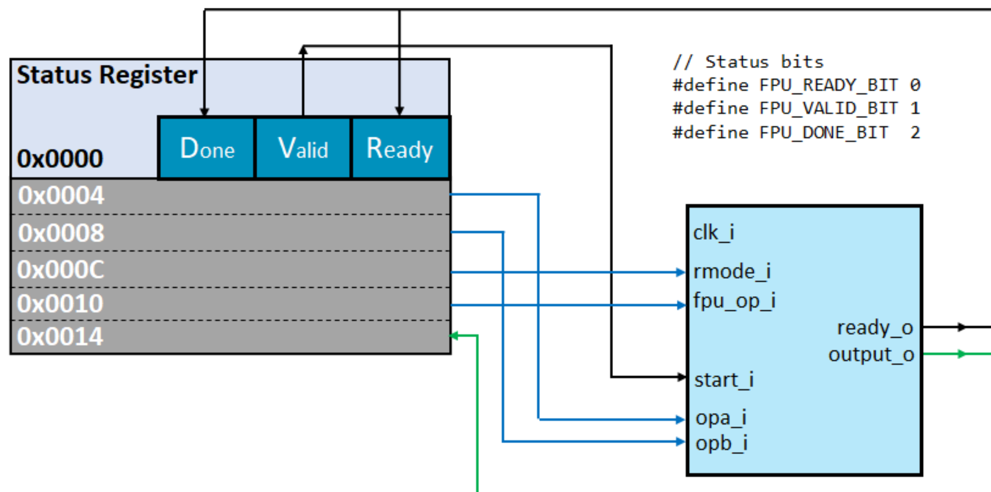


Figure 12: FPU IP wrapper memory map

output port in the IP are mapped to a separate address in the local memory space. This REK also provides a wrapper for the connection of this FPU IP to the AHB bus. Both APB and AHB wrappers are located inside the `/cm0_fpu/hw/wrappers` folder.

2.4.3 System Integration for the FPU IP

This section discusses how to use the `fpu_apb_wrapper` to connect the FPU IP to one of the available ports in the APB slave multiplexer `cmsdk_apb_slave_mux` instantiated in `fpga_apb_subsystem.v`. In this example we use port `11`, which has to be enabled by setting the `PORT11_ENABLE` parameter port to `1` in the multiplexer interface (see instantiation code below).

```
wire        i_psel_fpu;
wire        i_pready_fpu;
wire [31:0] i_prdata_fpu;
wire        i_pslverr_fpu;

// FPU Co-Processor IP Core
fpu_apb_wrapper u_fpu_apb_wrapper (
  .PCLK    (hclk),
  .PRESETn (hresetn),
  .PSEL    (i_psel_fpu),
  .PADDR   (i_paddr[11:2]),
  .PENABLE (i_penable),
  .PWRITE  (i_pwrite),
  .PWDATA  (i_pwdata[31:0]),
  .PRDATA  (i_prdata_fpu[31:0]),
```

Copyright © 2018 Arm Limited or its affiliates.

```
  .PREADY  (i_pready_fpu),
  .PSLVERR (i_pslverr_fpu)
);

cmsdk_apb_slave_mux
  #( // Parameter to determine which ports are used
  .PORT0_ENABLE  (1), // SPI
  ...
  .PORT11_ENABLE (1), // FPU
  ...
  )
u_apb_slave_mux (
  // Inputs
  ...
  .PSEL11    (i_psel_fpu),
  .PREADY11  (i_pready_fpu),
  .PRDATA11  (i_prdata_fpu),
  .PSLVERR11 (i_pslverr_fpu),
  ...
  // Output
  ...
  );
```

The FPU IP is mapped to base address 0x4002B000, which belongs to the peripheral memory section in the DesignStart SoC.

2.4.4 Extending Software Support for the FPU IP

The FPU software driver uses a C structure that abstracts the IP's interface and the status register. The fields in this structure are arranged in such a way that its layout in memory reflects the local memory map of the IP wrapper. This structure is defined in `soc.h` file, which is in the `cm0_fpu/sw/drivers` folder.

```
typedef struct {
__IO uint32_t STATUS_REG;      // Offset: 0x000 (R/W)  Status Register
__IO float DATA_A;             // Offset: 0x004 (R/W)  Operand A
__IO float DATA_B;             // Offset: 0x008 (R/W)  Operand B
__IO uint32_t RMODE;           // Offest: 0x00C (R/W)  Round Mode
__IO uint32_t OP;              // Offset: 0x010 (R/W)  Operation
__I  float RESULT;             // Offset: 0x014 (R)    Result
} FPU_TypeDef;
```

User can access the FPU IP by reading and writing to the structure. In the following usage example, we are using two static floating values as case scenario. User first waits until the component is ready polling the `READY` bit in the status register. Then, user loads the operands and selects the addition as operation. Finally, the IP module is enabled and the computation starts.

```
// Access the FPU IP on its base address
FPU_TypeDef * fpu_ip = ((FPU_TypeDef *) FPU_ADDR);
float fpu_op_res;

// Wait ready bit
while ((fpu_ip->STATUS_REG & (1ul << FPU_READY_BIT)) == 0);

//Set the input data writing to the C structure
fpu_ip->DATA_A  = (float) 7.125; // Static Value A
fpu_ip->DATA_B  = (float) 8.125; // Static Value B
fpu_ip->RMODE   = RM_NEAR;  // Selecting the rounding mode
```

Copyright © 2018 Arm Limited or its affiliates.

```
fpu_ip->OP     = FPU_ADD; // Selecting floating point operation, addiction

// Enable and start the module
fpu_ip->STATUS_REG = (1ul << FPU_VALID_BIT);

// Fetch the result without checking the ready bit
fpu_op_res     = fpu_ip->RESULT; // Reading the final result

//Print the result
printf("\nFPU IP result: %f", fpu_op_res);
```

Note that we fetch the final result without checking the READY bit in the status register. In AMBA 3, a slave can insert wait states into any transfer to enable additional time for completion [15]. The processor request is stalled until the slave result is ready. It is recommended that a slave must not generate more than 16 wait states.

2.4.5 Performance Evaluation

This section discusses the performance delivered by the FPU IP when it is integrated in APB bus, and compares this with the performance achieved by the Cortex-M0 when dealing with floating point operations. Figure 13 shows the number of clock cycles needed in each case to perform addition, multiplication and division. These results are collected using Timer0 in the DesignStart SoC. Note that in this particular case data transfer takes a long time, thus reducing the potential speed-up that can be achieved by hardware acceleration.

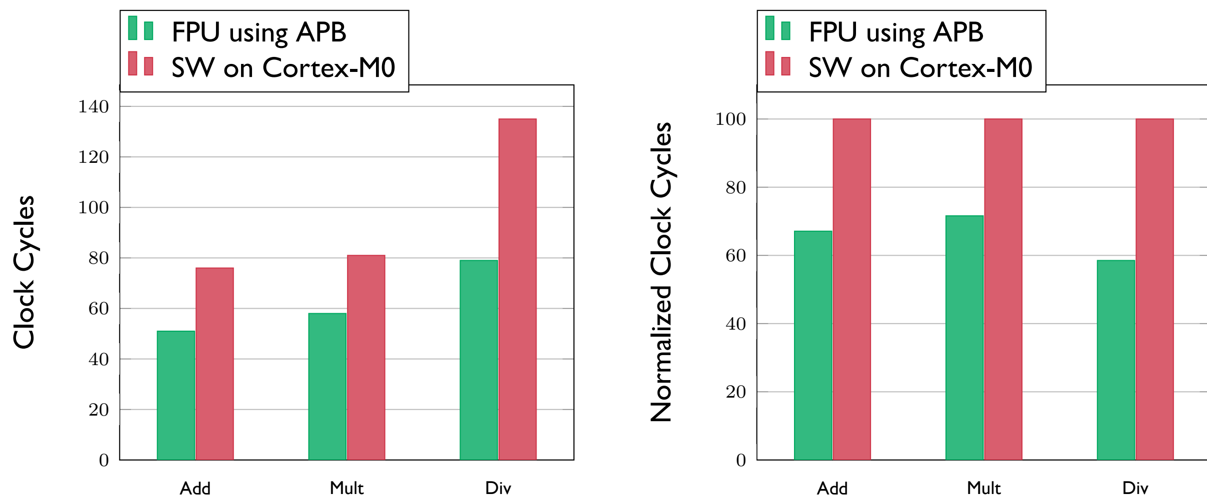


Figure 13: Single-precision floating-point performance

Copyright © 2018 Arm Limited or its affiliates.

2.5 AES IP Integration

This sections deals with the integration of an open source AES IP available on Github [16].

2.5.1 AES IP Overview

The AES IP implements 128- and 256-bit AES encryption and decryption. Since security is of utmost importance in modern applications, especially in the context of IoT, this IP enables secure SoCs with adequate performance and power consumption to be built.

It is worth mentioning, without going into details, that Arm has its own security IP solution. The Arm TrustZone CryptoCell-300 family of embedded security solutions is aimed at high efficiency systems with emphasis on small footprint and low power consumption [17]. However, in the section we do not aim to focus on the security IP itself. The objective of this section is to show how to integrate an open source IP into the DesignStart SoC. Figure 17 depicts the AES IP interface. On the data side, `block` and `result` are 128-bit data input and output ports, respectively. The encryption key is provided in the 256-bit `key` input port while the `keylen` input bit is used for the selection of key length. On the control side, `encdec` input bit is used to specify the operation type (encryption or decryption) while `reset_n` and `clk` are used to reset and enable the IP respectively. The `init` input bit starts the key initialization and `next` input is used to validate the `block` data and start a cryptography operation. The AES IP asserts `result_valid` output bit when the result is ready and `ready` indicates the availability of the IP to accept new operation. The process of key initialization is shown in the timing diagram in Fig. 15. Table 3 shows the IP latencies for different length keys in both encryption and decryption operation.

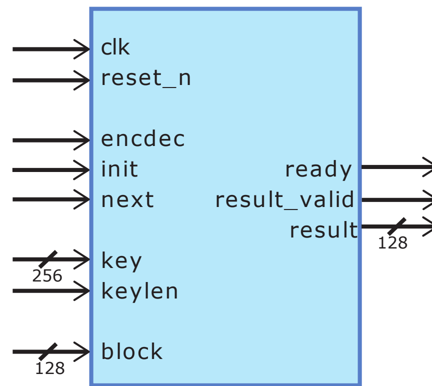


Figure 14: AES Core interface

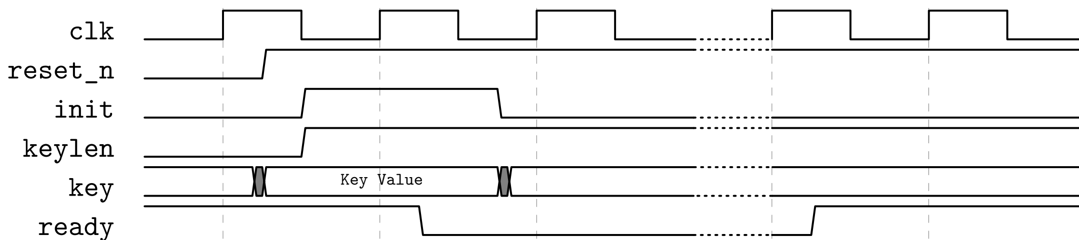


Figure 15: Key initialization

Copyright © 2018 Arm Limited or its affiliates.

	Key Init	Enc.	Dec.
Key 128	14	53	53
Key 256	18	73	73

Table 3: AES IP operation codes and clock cycles

Figure 16 shows the process of encryption. Note that the AES IP documentation states that the hardware design is pipelined and can run at 96MHz clock frequency, which is considerably higher than that used in DesignStart Eval.

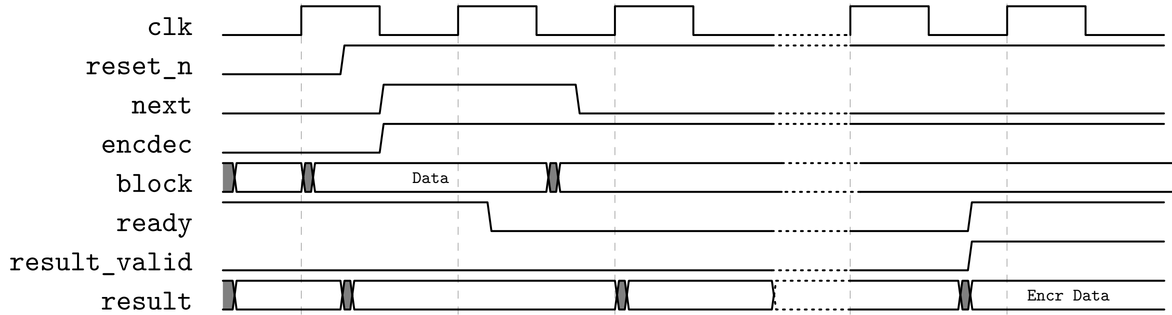


Figure 16: Encryption operation timing diagram

2.5.2 Customizing the AHB and APB Wrapper for the AES IP

This REK provides two wrappers for both APB and AHB buses. The wrappers are located inside `/cm0_aes/hw/wrappers` folder. Following the customization previously described in 2.4.2, status register contains several flags that are directly connected to IP control ports, as shown if Figure 17. Each individual input and output port in the AES IP are mapped to a separate address in the local memory space due to simplicity.

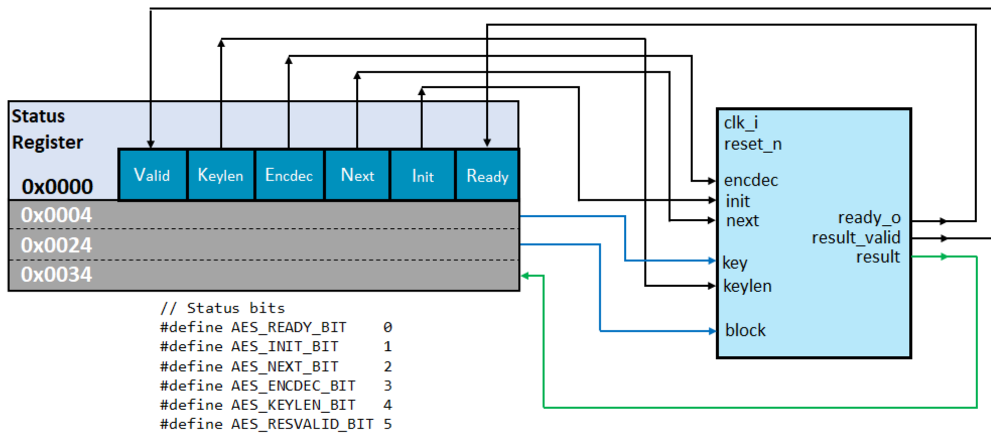


Figure 17: AES IP wrapper memory map

2.5.3 System Integration for the AES IP

This section discusses how to use the `aes_ahb_wrapper` to connect the AES IP to one of the available ports in the AHB slave multiplexer `cmsdk_ahb_slave_mux` instantiated in `user_partition.v`. In this example we use port 8, which has to be enabled by setting the `PORT8_ENABLE` parameter port to 1 in the multiplexer interface (see instantiation code below).

Copyright © 2018 Arm Limited or its affiliates.

```
wire            aes_hsel;     // AHB AES bus interface signals
wire            aes_hreadyout;
wire     [31:0] aes_hrdata;
wire            aes_hresp;

aes_ahb_wrapper u_aes_ahb_wrapper (
  //AHB Inputs
  .HCLK      (hclk_sys),            //system bus clock
  .HRESETn   (hresetn),             //system bus reset
  .HSEL      (aes_hsel),            //AHB peripheral select
  .HREADY    (fpgasys_hready),      //AHB ready input
  .HTRANS    (fpgasys_htrans),      //AHB transfer type
  .HSIZE     (fpgasys_hsize),       //AHB hsize
  .HWRITE    (fpgasys_hwrite),      //AHB hwrite
  .HADDR     (fpgasys_haddr[11:0]), //AHB address bus
  .HWDATA    (fpgasys_hwdata),      //AHB write data bus
  //AHB Outputs
  .HREADYOUT(aes_hreadyout),        //AHB ready output to S->M mux
  .HRESP     (aes_hresp),           //AHB response
  .HRDATA    (aes_hrdata)
);

...

cmsdk_ahb_slave_mux #(
  .PORT0_ENABLE (1), // SRAM - lower word
  ...
  .PORT8_ENABLE (1), // AES
  ...
  .PORT9_ENABLE (1))
  u_ahb_slave_mux_sys_bus (
  ...
  .HSEL8      (aes_hsel),       // Input Port 8
  .HREADYOUT8 (aes_hreadyout),
  .HRESP8     (aes_hresp),
  .HRDATA8    (aes_hrdata),
  ...
);
```

Unlike in the APB, the AHB address decoding is implemented in the `user_partition.v` (i.e., not in the AHB slave multiplexer), hence we need to design the logic to drive the `aes_hsel` selection signal. The code below shows how to generate this signal when the AES IP is mapped to 0x40016000 base address. Note that the `aes_hsel` selection signal needs to be considered in the generation of the slave select signal `defslv_hsel` (see the OR chain in the code).

```
assign aes_hsel    = fpgasys_hsel & ( fpgasys_haddr[31:12]==20'h40016 );
assign defslv_hsel = fpgasys_hsel &~( sram2_hsel|sram3_hsel|vga_hsel|
   fpga_apbsys_hsel|smi_hsel|gpio2_hsel|gpio3_hsel|mtimer_hsel|fpu_hsel|aes_hsel);
```

2.5.4 Extending Software Support for the AES IP

The AES software driver uses a C structure that abstracts the IP's interface and the status register. The fields in this structure are arranged in such a way that its layout in memory reflects the local memory of the IP wrapper. This structure is defined in `soc.h`, which is in the `cm0_aes/sw/drivers` folder.

Copyright © 2018 Arm Limited or its affiliates.

```
typedef struct {
    __IO uint32_t AES_STATUS;     // Offset: 0x000 (R/W)  Status Register
    __IO uint32_t AES_KEY[8];     // Offset: 0x004 (R/W)  Key 256 bits
    __IO uint32_t AES_BLOCK[4];   // Offset: 0x024 (R/W)  Data Block 128 bits
    __I  uint32_t AES_RESULT[4];  // Offset: 0x034 (R)    Result 128 bits
} AES_TypeDef;
```

Users can access the AES IP by reading and writing to the structure. Basic functions such as initialization of the AES IP using a symmetric key, data encryption and decryption are provided inside the file `aes.h` located on `cm0_aes/sw/drivers` folder of the REK. In the following simple usage example, the AES IP is initialized with a defined 256-bit key. The example then encrypts a message and finally decrypts it back.

```
// 128 bit block data
char data[16] = "Secret Message!!";
uint8_t crypt_data[16];
uint8_t decrypt_data[16];

uint32_t i;
uint32_t * pkey_32;
// 256bit key
uint32_t key[8] = {
(uint32_t) 0x2b7e15f4, (uint32_t) 0x1628aed2, (uint32_t) 0xa6abf715, (uint32_t) 0x92b6105e,
(uint32_t) 0x8809cf4f, (uint32_t) 0x3c1917a0, (uint32_t) 0x6e927afe, (uint32_t) 0xc120d25f
};
pkey_32 = (uint32_t *) key;

printf("\nPlain text: ");
for (i = 0; i < 16; i++)
printf("%c", data[i]);

// Init Key
aes_init_key(AHB_AES_ADDR, KEY_LENGTH_256, pkey_32);

// Start Ecryption
aes_op(AHB_AES_ADDR, KEY_LENGTH_256, AES_ENC, (uint32_t*) data, (uint32_t*) crypt_data);

printf("\nHW AES encrypt result: ");
for (i = 0; i < 16; i++)
printf("%x", crypt_data[i]);

// Start Decryption
aes_op(AHB_AES_ADDR, KEY_LENGTH_256, AES_DEC, (uint32_t*) crypt_data, (uint32_t*) decrypt_data);

printf("\nHW AES encrypt result: ");
for (i = 0; i < 16; i++)
printf("%c", decrypt_data[i]);
```

If the integration is successful, the processor will be able to communicate with the AES IP and the previous example code will produce the following output:

```
Plain text: Secret Message!!
HW AES encrypt result: c0868993116321d39585ba33bd12
HW AES decrypt result: Secret Message!!
```

Figure 18: AES IP successful encryption and decryption results

A more advanced example is provided in `app_display.h` located inside the `cm0_aes/sw/app` folder of the REK. It uses the on-board LCD display of the MPS2+ board. First an image is displayed on the LCD, then according to users input,

Copyright © 2018 Arm Limited or its affiliates.

it performs encryption to the image. If Button0 on MPS2+ board is pressed, the AES hardware module starts the image encryption flow and the output is displayed on the LCD screen. If Button1 is pressed, Cortex-M0 performs the same flow in software.

2.5.5 Performance Evaluation

This section discusses the performance delivered by the AES IP when it is integrated in AHB and APB buses, and compares this with the performance achieved by the Cortex-M0 when running the AES 256 algorithm available at [18]. Figure 19 shows the number of clock cycles needed in each case to initialize the encryption key as well as to encrypt and decrypt 38400 32-bit words. These results are collected using `Timer0` in the DesignStart SoC.

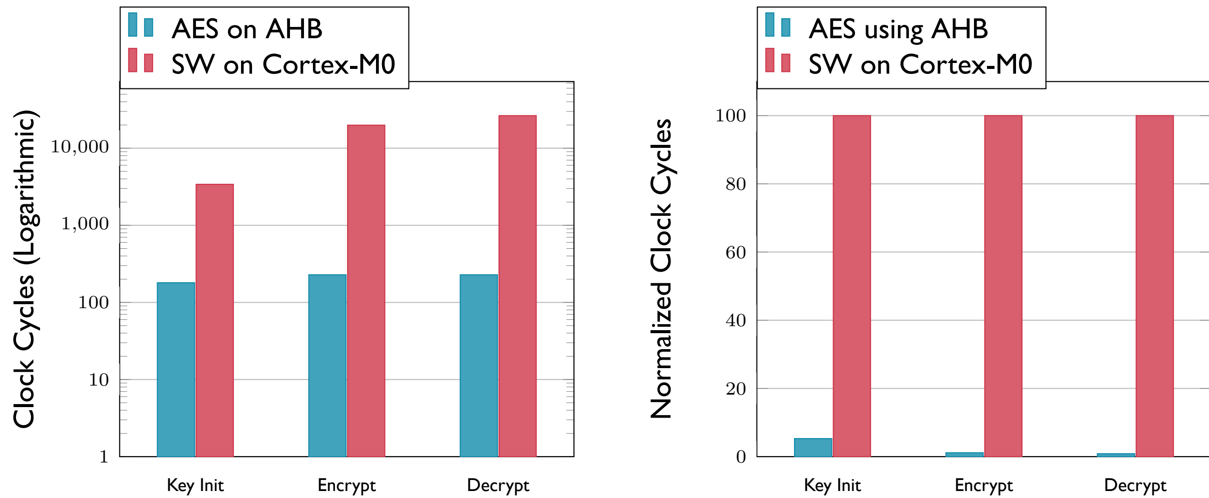


Figure 19: AES performance evaluation

Copyright © 2018 Arm Limited or its affiliates.

arm

Summary

This document is part of the Arm Research Enablement Kit (REK) on SoC Design and Prototyping. The purpose of this document is to help designers with prototyping a SoC based on the Arm Cortex-M0 processor and AMBA standards using Arm DesignStart.

In the first chapter, we focused on the SoC design flow and how to integrate custom IP cores to an Arm-based SoC. We also introduced Arm DesignStart, the ideal starting point for a custom SoC, which provides easy and instant access to Arm processor IP and subsystems.

In the second chapter, we described the integration of two IP cores, namely Floating Point Unit (FPU) and Advanced Encryption Standard (AES) IP cores into the Cortex-M0-based subsystem and peripheral components provided by Arm DesignStart. We also compared the performance results of the IPs integrated in AMBA Advanced High-Performance Bus (AHB) and Advanced Peripheral Bus (APB) with those of the software implementations of the algorithms running on the Cortex-M0 processor itself.

Copyright © 2018 Arm Limited or its affiliates.

Acknowledgements

We would like to thank the following people for their constructive feedback:

- Sean Houlihane, Arm
- Charlotte Christopherson, Arm
- Prof. Loukas Petrou, AUTH
- Prof. George Hassapis, AUTH
- Dr. Basel Halak, University of Southampton

Copyright © 2018 Arm Limited or its affiliates.

arm

Bibliography

[1] Arm Developer, "DesignStart." https://developer.arm.com/products/designstart.

[2] Arm Developer, "Fast Models." https://developer.arm.com/products/system-design/fast-models.

[3] Arm, "Amba 3 ahb-lite protocol v1.0 specification." http://www.eecs.umich.edu/courses/eecs373/readings/ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf.

[4] Arm, "AMBA 3 apb protocol v1.0 specification." http://web.eecs.umich.edu/~prabal/teaching/eecs373-f12/readings/ARM_AMBA3_APB.pdf.

[5] P. Burr and T. Menasveta, "An introduction to Arm Cortex-M0 DesignStart," tech. rep., Arm, March 2017.

[6] Arm, "Arm Versatile Express Cortex-M Prototyping System (V2M-MPS2) - Technical Reference manual." http://infocenter.arm.com/help/topic/com.arm.doc.100112_0100_03_en/arm_versatile_express_cortex_m_prototyping_system_(v2m_mps2)_technical_reference_manual_100112_0100_03_en.pdf.

[7] Arm Developer, "CMSIS cortex microcontroller software interface standard." https://developer.arm.com/embedded/cmsis.

[8] Arm, "Arm Cortex-M0 DesignStart Eval." http://infocenter.arm.com/help/topic/com.arm.doc.dui0926b/DUI0926B_cortex_m0_designstart_eval_guide.pdf.

[9] Arm Developer, "Arm Cortex-M Prototyping Systems." https://developer.arm.com/products/system-design/development-boards/cortex-m-prototyping-system.

[10] Altera, "Intel Quartus Prime Design Software Overview." https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html.

[11] Arm Developer, "Arm Compiler 5 Downloads." https://developer.arm.com/products/software-development-tools/compilers/arm-compiler/downloads/version-5.

[12] Arm, "Cortex-M0, Technical Reference Manual ." http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/DDI0432C_cortex_m0_r0p0_trm.pdf, 2009.

[13] OpenCores, "FPU Overview OpenCores." https://opencores.org/project,fpu100.

[14] "Nios II Floating Point Hardware 2 Componsent User Guide," tech. rep., Altera, 2016.

[15] Arm, "AMBA Design Kit." http://infocenter.arm.com/help/topic/com.arm.doc.ddi0243c/DDI0243C_adk_r3p0_trm.pdf.

[16] J. Strombergson, "Aes." https://github.com/secworks/aes, 2013-2014.

[17] Arm Developer, "TrustZone Security ip, CryptoCell-300." https://developer.arm.com/products/system-ip/trustzone-security-ip/cryptocell-300-family.

[18] Literatecode, "A byte-oriented aes-256 implementation." http://www.literatecode.com/aes256.

Copyright © 2018 Arm Limited or its affiliates.