Northwestern University

EECS 495 - KERNEL AND OTHER LOW-LEVEL SOFTWARE DEVELOPMENT

Final Project

Embedded FPGA-CPU Communication via an External Memory Interface

March 18, 2017

Patrick Abiney
Department of Mechanical Engineering
patrickabiney2017@u.northwestern.edu

Josh Fixelle
Department of Physics and Astronomy
joshuafixelle2014@u.northwestern.edu

Abstract

Many embedded devices are designed around the concept of heterogeneous computing for both speed and power consumption reasons. Often this will lead to systems containing a primary compute unit (a CPU), and a secondary compute unit that acts as a co-processor, where the two are connected through some variant of a system bus. The most notable example of this would be the standard CPU-GPU model seen in laptop and desktop computers. In more specialized cases, such as test instrumentation, the model will often take the form of CPU-FPGA, where the primary compute unit is a low power embedded processor. In the context of test instrumentation, this model allows for real-time data streaming from sensors to be processed by the FPGA, and then presented to the user by way of the CPU. The design and implementation of such a system, however, is non-trivial, requiring custom software solutions to properly bridge the CPU-FPGA gap. For example, signal acquisition and display, as seen in bench oscilloscopes and spectrum analyzers, is an application of a CPU-FPGA. To implement a FPGA-MPU bridge design, we will be considering communication between a TI AM1808 ARM CPU and an Altera FPGA . The communication will be done over the exposed EMIFA bus on the ARM CPU, which provides an interface of synchronous and asynchronous RAM, as well as DMA channel support. In the end, we were able to verify successful write operations to the FPGA, however, we were unsuccessful at read operations.

Contents

1 Introduction **5**
1.1 Motivation 5
1.2 Link Methods 6
1.2.1 Low Speed Serial Link 6
1.2.2 Low Speed Parallel Link 6
1.2.3 Memory Interface Link 7
1.2.4 High Speed Parallel Link 8
1.2.5 High Speed Serial Link 9
1.3 Related Work 10

2 Hardware Selection **11**

3 Workflow **12**

4 Functional Requirement and Design Challenges **14**
4.1 Baseline Functional Requirements 14
4.2 Challenges in Link Method 14
4.3 Challenges Due to Chosen Hardware 14

5 System Design **17**
5.1 Hardware Setup 18
5.2 FPGA Side 18
5.3 ARM CPU Side 19

6 User Interface **21**

7 Evaluation **23**
7.1 Write Test 23
7.2 Read Test 24

8 Conclusion **25**

References **26**

A Cross-Compiler Tool-chain **27**
A.1 Prerequisites 27
A.2 Compilation 27

B AM1808 Configuration **30**
B.1 AM1808 Memory Map 30
B.2 Pin Multiplexing 30
B.3 EMIF Timing and Mode Selection 30

List of Figures

1.1 Low-speed FPGA-CPU Serial Link 6
1.2 Low-speed FPGA-CPU Parallel Link 7
1.3 Memory Interface Based FPGA-CPU Link 7
1.4 High-speed FPGA-CPU Parallel Link 8
1.5 High-speed FPGA-CPU Serial Link 9

3.1 Development Process 12

4.1 SBC8018 EMIFA Pin Header 15
4.2 EMIFA Byte Mapping 16

5.1 FPGA-ARM Block Diagram 17
5.2 Hardware Setup 18
5.3 FPGA block diagram 19

6.1 Driver Interface 22

7.1 Write Result 23

B.1 AM1808 Memory Map 30
B.2 IO Selection/Configuration 31
B.3 Timing and Mode Selection 31

List of Tables

1.1 FPGA-CPU Examples 5

Chapter 1

Introduction

Many embedded devices are designed around the concept of heterogeneous computing for both speed and power consumption reasons. Often this will lead to systems containing a primary compute unit (a CPU), and a secondary compute unit that acts as a co-processor, where the two are connected through some variant of a system bus. The most notable example of this would be the standard CPU-GPU model seen in laptop and desktop computers. In more specialized cases, such as test instrumentation, the model will often take the form of CPU-FPGA, where the primary compute unit is a low power embedded processor. In the context of test instrumentation, this model allows for real-time data streaming from sensors to be processed by the FPGA, and then presented to the user by way of the CPU. The design and implementation of such a system, however, is non-trivial, requiring custom software solutions to properly bridge the CPU-FPGA gap.

For this application, we will be considering a streaming application where the FPGA will act as a producer of data and the CPU will act as a consumer. This is in contrast to the typical example of a CPU-GPU pair, where the CPU is the producer and the GPU is the consumer. For this reason, we will be considering the typical link model of an external memory interface.

1.1 Motivation

The application of a producer-consumer model for an FPGA-CPU, respectively, is often seen in test instrument applications. One can often find such configurations when inspecting the design of, or reverse engineering such products. A few notable examples can be found in Tab.1.1. Each of the examples has a high bandwidth signal input (via and Analog to Digital Converter) attached to an FPGA which is then linked to an embedded ARM based CPU via a memory interface link.

Device	CPU	FPGA
VDS1022 USB Oscilloscope	SiM3U156 (ARM Cortex-M3)	Spartan-3
SSA3021X Spectrum Analyzer	AM3352 (ARM Cortex-A8)	Spartan-6
DS1054Z Oscilloscope	iMX283 (ARM CoreSight ETM9)	Spartan-6

Table 1.1: A Few Examples of FPGA-CPU Architecture in Test Instrumentation. Each of the examples listed happen to use a memory interface to stream data from the FPGA to the CPU. Additionally, all of the FPGAs are made by Xilinx.

1.2 Link Methods

There exist several link methods for FPGA-CPU communication, each having their own strengths and weaknesses. Additionally, each link method has a range of applications and hardware requirements (both at the Device and PCB level). All of the diagrams in this section are from [1].

1.2.1 Low Speed Serial Link

Low speed serial links using standard protocols such as I2C, SPI, UART, and CAN are often used for low bandwidth communication requirements. These protocols have the benefit that they are relatively low frequency, most of which having a 400 KHz maximum frequency[1]. A typical link configuration can be seen in figure 1.1.

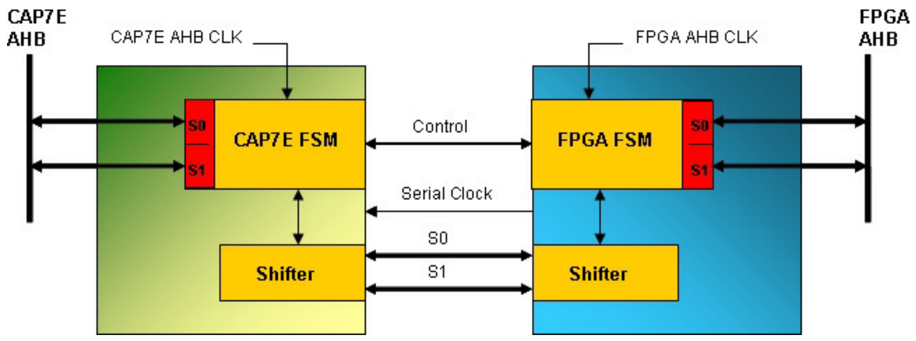


Figure 1.1: Low-speed FPGA-CPU Serial Link

There are several benefits of low-speed serial communications. In terms of the PCB level, low-speed serial communication do not often require proper impedance matching, and thus are very easy / low cost to implement on the PCB level. In terms of devices, many embedded CPUs (MCUs and MPUs) contain hardware support for common serial communication protocols (i.e. I2C, SPI, UART, and less common, CAN). From the FPGA side, implementation of the aforementioned serial communication protocols can often be done on the order of 100 Logic Elements, which leaves more room for the remaining logic IP Cores.

The downside to using low-speed serial communication, is that it is low bandwidth, leaving it inappropriate for high bandwidth applications. It should be noted, however, that many designs using high-bandwidth link methods will often also employ an additional low-speed serial link for low-bandwidth communication. A typical example of this would be a serial link that acts as a system management bus.

1.2.2 Low Speed Parallel Link

Low-speed parallel links are often only used when the devices being considered do not have the required peripheral hardware interfaces for a high speed link, and a low-speed serial link is insufficient or not applicable (i.e. devices that do not have dedicated serial communication hardware). A typical link configuration can be seen in figure 1.2.

[1]There exist SPI standards that allow it to operate upwards of 8 MHz, however, this is still considered low frequency serial compare to the other link methods listed in this section.

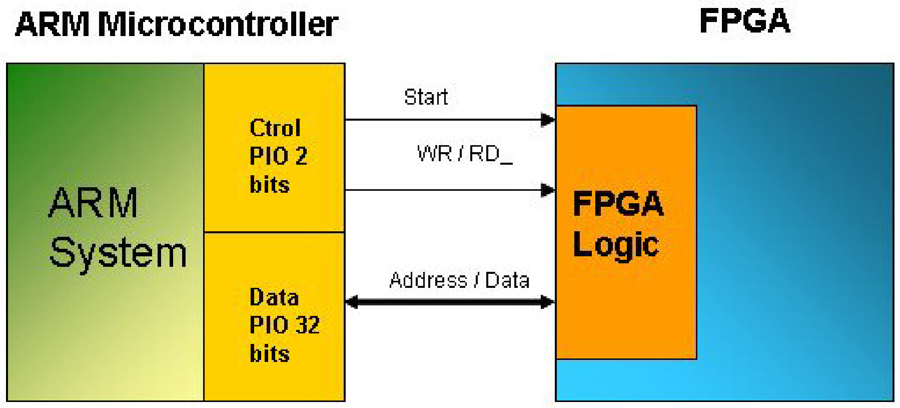


Figure 1.2: Low-speed FPGA-CPU Parallel Link

The main hallmark of a low-speed parallel link, is that it uses GPIO pins on the embedded CPU. This means that the link must be implemented in software on the CPU side, and will thus be limited in speed, and often will not be access able to perform other work while the link is being used.

1.2.3 Memory Interface Link

The primary benefit to using a memory interface link (or External Memory Interface - EMIF) link, is that the interface is relatively simple, and the memory interface operates at a speed which most low end FPGAs can handle. Typical frequencies here are somewhere between 50-200 MHz. A typical link configuration can be seen in figure 1.3.

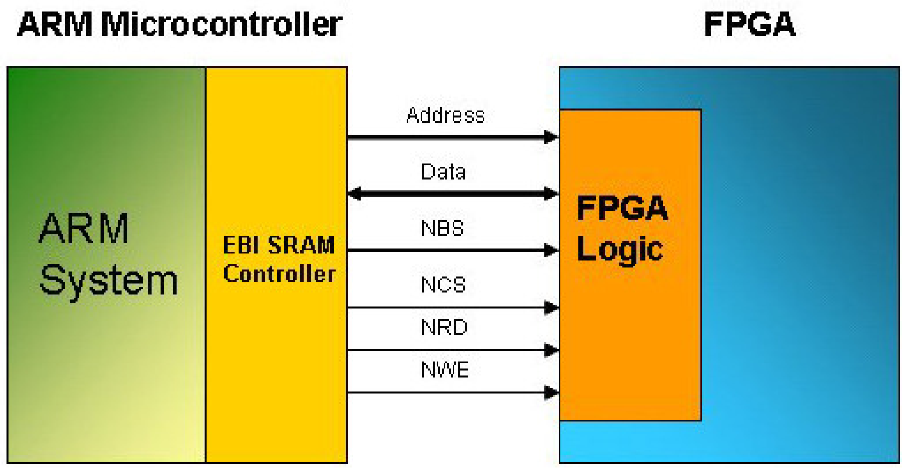


Figure 1.3: Memory Interface Based FPGA-CPU Link

This method, however, requires that the embedded CPU exposes a memory interface. If the CPU relies on external memory and only exposes a single interface, then it will end up sharing the interface with the FPGA, which will reduce the potential link bandwidth. Additionally, due to the relatively high speed communication, impedance matching becomes important leading to restrictions concerning PCB implementations, thus increasing potential costs.

It should be noted that the majority of memory interface links are based on asynchronous memory timing diagrams. This allows for the FPGA core design to be simpler (as opposed to implementing a DDR slave interface). It is possible to rely on a synchronous memory timing diagram (for SDRAM), however, this will lead to “dead” cycles where the hard SDRAM controller on the CPU will be attempting to refresh / precharge the FPGA.

1.2.4 High Speed Parallel Link

High-speed parallel links can come in several different forms, and often allow for very high bandwidths. High-speed parallel links will often operated somewhere between 50-200 MHz (similar to the memory interface link), however, they will often allow for a 32+ bit data bus. Additionally, due to these speeds, impedance matching becomes important.

Two main parallel link methods will easily come to mind: 1) PCI, 2) HPI. The PCI interface was common in the mid 1990s-2000s for desktop computer peripheral devices. This interface exists in both a 32-bit and 64-bit version, operating between 33-133 MHz, which provided decent bandwidths (upwards of 500 MB/s). It is rare to find an embedded CPU which contains a PCI peripheral interface, however, some DSP co-processor devices (such as the Sitara family of Texas Instrument processors) allow for pins to be connected directly to the co-processor. This means that the co-processor can implement a mock PCI interface in software, however, since the main CPU and co-processor in these devices are independent, this will effectively lead to the appearance of a hard PCI peripheral interface. Another option here would be to implement the PCI periphal interface using software emulation and GPIO pins. It is possible that on devices operating at 1+ GHz, that such an emulator could achieve the minimum frequency of the PCI specification (33 MHz).

The second high-speed parallel link is the HPI or Host-Port Interface. This interface is meant to interface with other devices, specifically FPGAs. This interface can operate on order of several hundred MHz, and can have a varying width, depending on the device. It should be noted, however, that this requires a specific hard peripheral interface on the embedded CPU to work. As in the previous case of a PCI peripheral interface, a processor with dedicated co-processor GPIO pins can emulate a HPI using co-processor specific software. An example of a HPI link can be seen in figure 1.4.

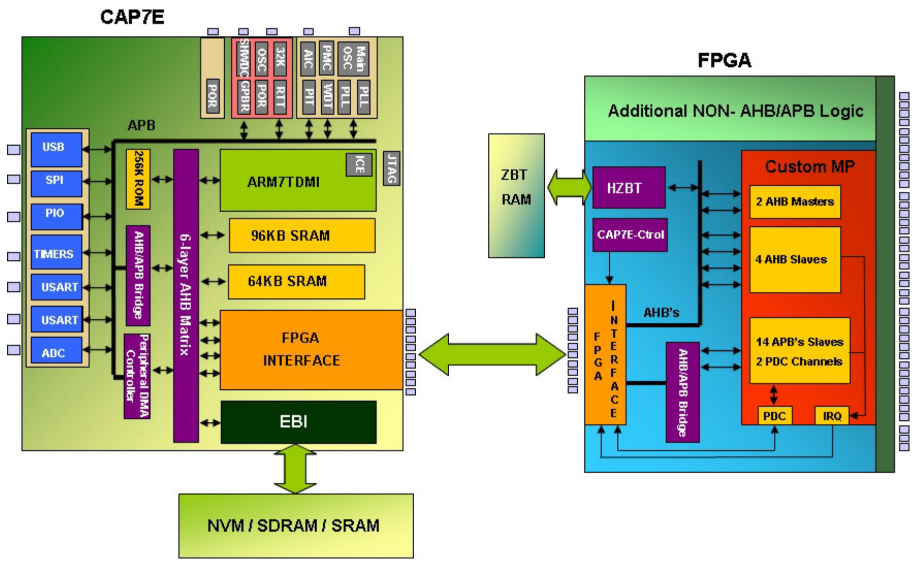


Figure 1.4: High-speed FPGA-CPU Parallel Link

1.2.5 High Speed Serial Link

High-speed serial links are the option that allow for the highest possible bandwidth between devices. The links will often operated at several GHz, and as such, require highly controlled differential impedance traces on the PCB, as well as FPGAs containing hard high-speed differential transceivers. As such, these links are often very expensive (compared to the other methods) to implement. It should be noted that the idea of a high-speed serial link can mean that there are multiple links operating in parallel. An example link can be seen in figure 1.5.

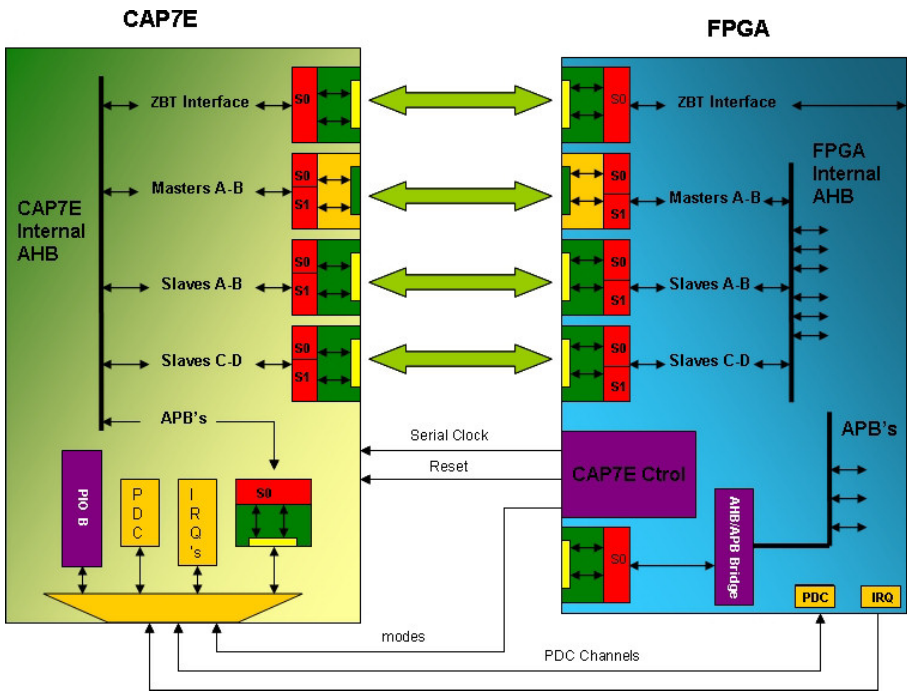


Figure 1.5: High-speed FPGA-CPU Serial Link

Typical high-speed serial links are: 1) PCI-Express (successor to PCI), 2) USB, 3) SATA, 4) LVDS, 5) STL. Both LVDS and STL specifications are in terms of impedance and electrical characteristics, however, neither has a specific packet specification. These link methods are not typically used in CPU-FPGA links, since most embedded CPUs do not provide an LVDS / STL transceiver interface[2]. USB and SATA are two well known high-speed serial communication protocols, however, both are rarely (if ever) used for an FPGA-CPU link. They can, however, be used in principle to form a high-speed link.

The most common high-speed serial link is PCI-Express, which allows for high bandwidth upwards of 5 GB/s depending on the version of PCI-Express implemented[3]. PCI-Express defines a specification for both the electrical characteristics, and packet structure for transactions, which often leads to the instantiation of complicated and resource heavy IP cores in the FPGA. The main exception to this limitation, is when an FPGA implements a hard PCI-Express IP Core (meaning dedicated hardware), where no logic is required to implement the interface.

[2]Many embedded CPUs provide LVDS interfaces for LCD screens, however, these are transmitters on the CPU side, and will not receive data from the FPGA.

[3]Most FPGAs only implement PCI-Express 1.0 or 2.0, which are not capable of 5 GB/s bandwidths with a single lane.

1.3 Related Work

There exist several open-source example implementations [2, 3] (hardware and software) for a embedded CPU-FPGA interfaces via an external memory interface, however, none adequately address high throughput applications. Additionally, as is the case for many hardware applications, the details of the implementation are often the most complicated aspect. For this reason, this project serves as an additional guide for embedded CPU-FPGA communication, specifically focusing on high throughput applications.

Chapter 2

Hardware Selection

For the simplicity of the project, our goal is to find an embedded processor development platform that exposes a parallel Asynchonous SRAM interface. This will allow us to easily write an interface IP core for the FPGA. This requirement, however, severely limits the number of suitable platforms, since an external memory interface will typically require many GPIO connections. We were able to find and procure several copies of the Embest SBC8018 development board for the Texas Instruments Sitara AM1808 processor, which implements the ARM Cortex-A9 specification. This specific processor contains two external memory interface (one of DRAM and one for SRAM), making it an ideal fit for our application. Additionally, the SBC8018 development board does expose this interface via a GPIO connector.

In addition to the exposed external memory interface on the SBC8018 development board, the board also contains an LCD screen, Ethernet connector, and a RS232 serial connection for debug and console instantiation. All of these features are enabled with the pre-configured Linux kernel on the device. These additional features will further facilitate the development of our intended application.

The choice of FPGA is not crucial, considering that the only requirement is that it have a sufficient number of GPIO pins accessible to facilitate the external memory interface connection. Due to the abundance of University owned Terasic DE2-115 development boards in the department, we decided that those would be the simplest solution. The DE2-115 development boards contain a Cyclone IV E FPGA, with an exposed 35 pin GPIO header (as well as a high-speed mezzanine connector if needed). In addition to the GPIO header, the DE2-115 development board also contains a 16x2 character LCD screen, which can be used to visualize the internal state of the memory interface for debug purposes.

Chapter 3

Workflow

The SBC8018 development board is equipped with an RS232 port which the processor uses for debug output, as well as the primary console. This allows for a virtual console into the linux kernel on the device to be established via a COM port (either direct, or from a USB-RS232 serial cable). Any work done on the device should be done via this port as opposed to using external Human Interface Devices (e.g. Keyboard and Mouse) in conjunction with the LCD screen.

In order to develop low-level software for the ARM CPU development board, a compilation workflow was needed. Due to the size constraints on the development board (8 MB Flash), the compilations tools needed to exist on a separate host machine, where the executables are copied over to the development board. To do this, we chose to use a linux based x86 host machine to develop all software. The procedure for setting up the cross-compilation tool-chain (ARM binaries from an x86 host) is described in Appendix A.

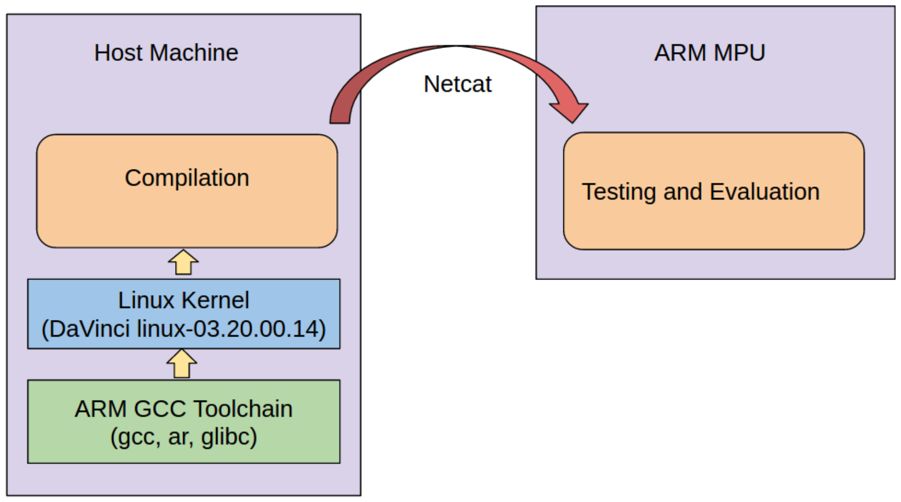


Figure 3.1: Development Process

In the Development Process diagram, figure 3.1, this ARM GCC tool-chain is represented in green. After the tool-chain builds the kernel module for the Texas Instrument DaVinci kernel, shown in blue, the kernel module can be sent over to the ARM CPU development board, shown as the right, purple block, via a Netcat connection, shown as the red arrow. From there, the kernel module can be tested on the ARM development board.

Netcat was chosen as the method to copy new binaries over to the development board, since the SBC8018 board supports and Ethernet connection, and the stock Linux image comes equipped with Ethernet drivers. Additionally, Netcat is significantly faster than copying via a serial connection, or via an external memory device.

Chapter 4

Functional Requirement and Design Challenges

4.1 Baseline Functional Requirements

The baseline functional requirements for such an application is the ability for the CPU to perform read and write operations on memory locations exposed by the FPGA's interface IP Core. This can be demonstrated, via a simple register or block-RAM interface on the FPGA side, where each address (or a subset of addresses) on the EMIF bus corresponds to either a block-RAM or a register address. If successful high-speed read/write operations can be demonstrated, then the interface has achieved minimum functionality. Any further generalization will extend and build upon read/write operations.

Here the notion of "high-speed" is a relative term, which we will define as for the given hardware setup. This means that if the connection between the two devices can only deal with timing constrains at a maximum of 50 MHz bus speed, then high-speed is 50 MHz, even though the bus may theoretically be capable of operating at higher frequencies.

4.2 Challenges in Link Method

The link method that we are choosing relies upon a board-wire-board connection via individual wires. These individual wires do not have a controlled impedance, and also serve as a parasitic inductor in series with the bus. As a result, the interface could become unstable for high frequencies due to reflections and distortions.

To minimize the effects of improper impedance matching, steps can be taken to ensure that the board-wire-board cables are as short as possible, thus minimizing the inductance. To verify that impedance will not be an issue, however, requires that the hardware be connected and probed in real-time via an oscilloscope.

4.3 Challenges Due to Chosen Hardware

We encountered several design challenges caused by the chosen hardware, all of which due to the SBC8018 development board. The design challenges were as follows: 1) Incorrect schematic diagram, 2) poor header specification choice, and 3) poor bus export design for pin headers. The challenge of the incorrect schematic diagram was caused by the pin-header being flipped on the board (i.e. a bottom connector when populated on the topside of the board). This lead to the pin connections to be mirror from what was expected, causing confusion with signals not showing up correctly. The second challenge was caused by the company's choice to use 2 mm headers instead of 2.54 mm (0.1") headers. The reduction of the header size saves space, however, it leads to brittle and easily broken crimp connections with connected wires. This problem occurred several times, requiring that the wires be re-crimped to fix the connection issues.

The most significant challenge was that of the poor bus export design by the company that manufactured the development board. The header schematic symbol can be seen in figure 4.1. It can very easily be noted when comparing the EMIFA signals on the pin header to those in the AM1808 manual, that the exported signals are incomplete for all possible memory implementation. That means that there exists no complete memory implementation choice for our FPGA design.

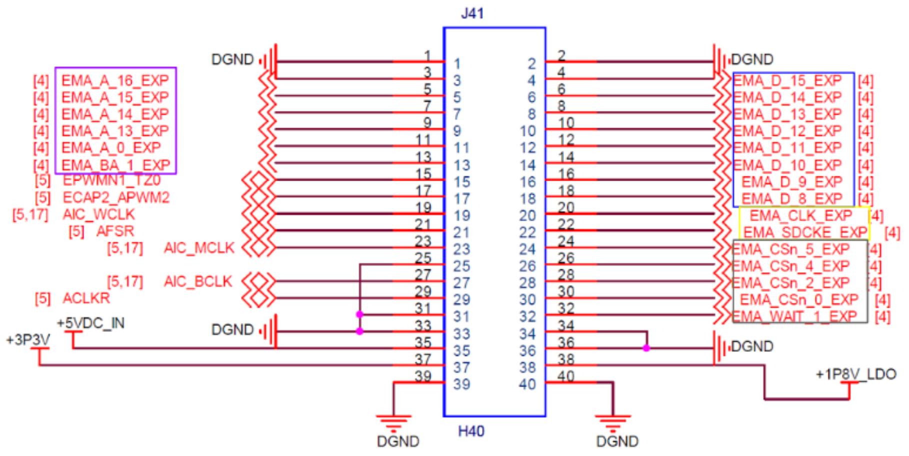
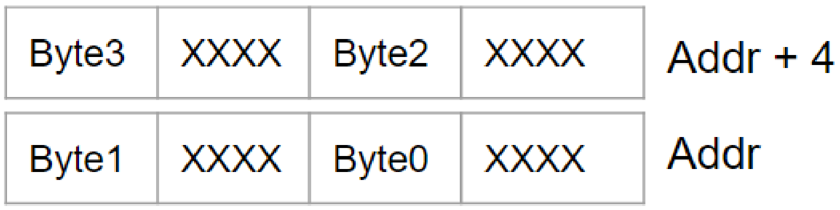


Figure 4.1: The schematic symbol for the pin header on the SBC8018 development board which contains the EMIFA signals. The different signal groups are color-coded: data (blue), address (purple), clock (yellow), and control (gray). It should be noted that the following problems with the header exist: 1) the symbol is mirrored on the x-axis, 2) the header is 2mm and not 2.54mm, and 3) the exported signals are incomplete.

To overcome this challenge, we decided to use the ASRAM interface from the AM1808. Unfortunately, the output enable, and read/write signals are not exported on the pin header. These, however, can be emulated using the chip select signals. We can write the interface as follows:

$$\begin{aligned} \overline{\mathrm{OE}} &= \neg\left(\overline{\mathrm{cs4}} \wedge \neg\overline{\mathrm{cs5}}\right) \\ \overline{\mathrm{W/R}} &= \overline{\mathrm{cs4}} \\ \overline{\mathrm{cs}} &= \neg\left(\overline{\mathrm{cs4}} \oplus \overline{\mathrm{cs5}}\right) \end{aligned}$$

This allows us to effectively use writes to the CS4 section on the AM1808 to act as writes to the FPGA, and reads from the CS5 section on the AM1808 to act as reads from the FPGA. The next issue to deal with is the export of only the upper 8-bits of the data bus. For data to appear on the bus at all, we must configure it to be 16 bits, however, the lower 8 bits of any write will be discarded and of any read will be garbage. To deal with this, we can pretend the bus is only 8-bits on the FPGA side, and the address is byte addressed. This allows us to configure the address bus as follows: `Addr[6:0] = {Addr[16:13], Addr[0], BA1}`. This means that we will need to configure and reads / writes on the AM1808 side as seen in figure 4.2.



Byte3	XXXX	Byte2	XXXX	Addr + 4
Byte1	XXXX	Byte0	XXXX	Addr

Figure 4.2: The byte mapping for the EMIFA interface that only uses bits 15:8. The X values represent don't-care values.

Chapter 5

System Design

The intended system design of the FPGA-ARM CPU bridge can be seen represented in the FPGA-ARM block diagram, figure 5.1. The goal implementation included an end user application utilizing a driver which exposes the read and write capabilities to the FPGA's EMIFA bus. The driver and user application will be running on top of a linux kernel, in this case the DaVinci linux-03.20.00.14 kernel. The kernel would be modified to use mmap to give a layer of abstraction to the memory on the ARM development board for the driver to use. From the hardware of the ARM development board to the FPGA development board, a custom pin-header cable was used to provide a connection. The FPGA provides an interface which is used to allow the ARM CPU to access the IPCore(s) of the FPGA. The entire implementation of the hardware can be seen in the Hardware Setup photograph, figure 5.2.

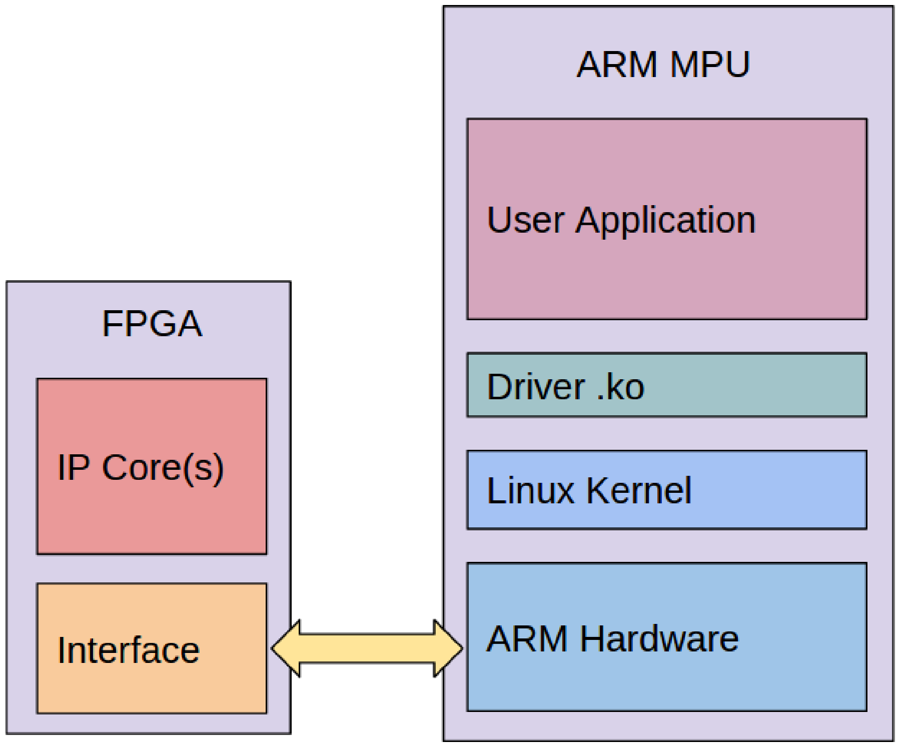


Figure 5.1: FPGA-ARM Block Diagram

5.1 Hardware Setup

In the image of the hardware setup, figure 5.2, there are two development boards. The FPGA development board is on the left side of the photograph while the ARM CPU development board is on the right side of the photograph. The FPGA development board is connected via a custom pin connection to the ARM CPU development board. The ARM photograph clearly displays the ARM CPU running Linux. Notice that an image of TUX has been rendered to the ARM CPU development board's display. The FPGA's display is in fact enabled as well as displaying a lit row of 0's. However the photograph does not do the FPGA's display justice because the lighting in the room at the time of the photograph forces the display to appear to be disabled or turned off when the display is in fact working.

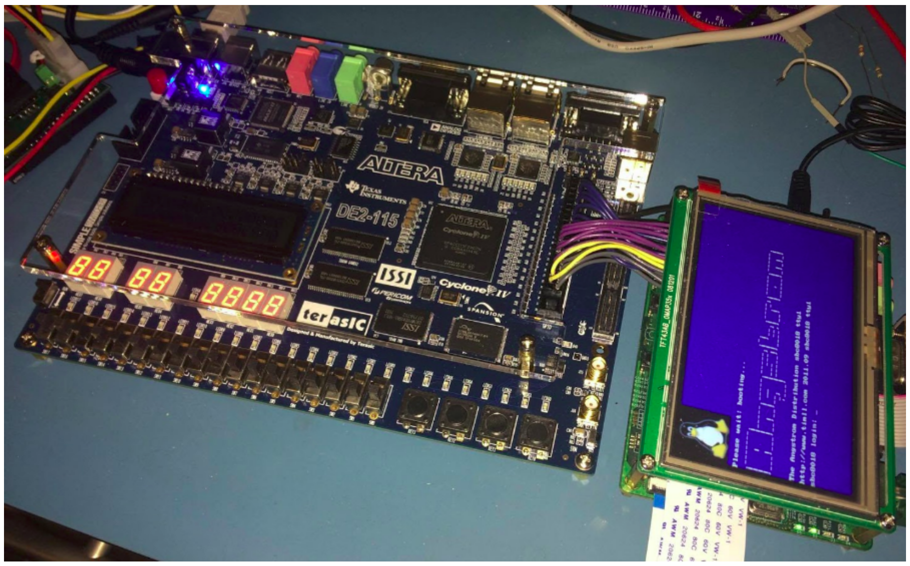


Figure 5.2: Hardware Setup

5.2 FPGA Side

On the FPGA side of the FPGA-ARM block diagram, figure 5.1, the programmable interface will be communicating with the IP Core(s) of the FPGA and, through the custom pin header cable, the ARM hardware. The FPGA's interface allows the FPGA to play nicely with the ARM CPU. The interface's communication to the IP Core(s) is what allows the FPGA to be reconfigurable and do what an FPGA was intended to do.

The FPGA interface with the EMIFA bus is represented in figure 5.3. Here, we implement a simple register interface, which allows the ARM CPU to perform 32 bit read / write operations on registers within the FPGA. The contents of the first 4 registers are output to the LCD screen in hex format (since the screen is 16x2=32 characters, and 8 are needed to represent a 32 bit word). Further extensions can be made to change the value of the FPGA registers on the FPGA side, however, a simple read / write storage interface is sufficient for a proof of concept impelmentation.

One thing that can be noted in figure 5.3, is that we make use of the EMIFA clock. This is due to timing issues on the FPGA. Since most applications on an FPGA will rely on an internal clock for synchronization, a purely asynchronous interface would not be viable for an

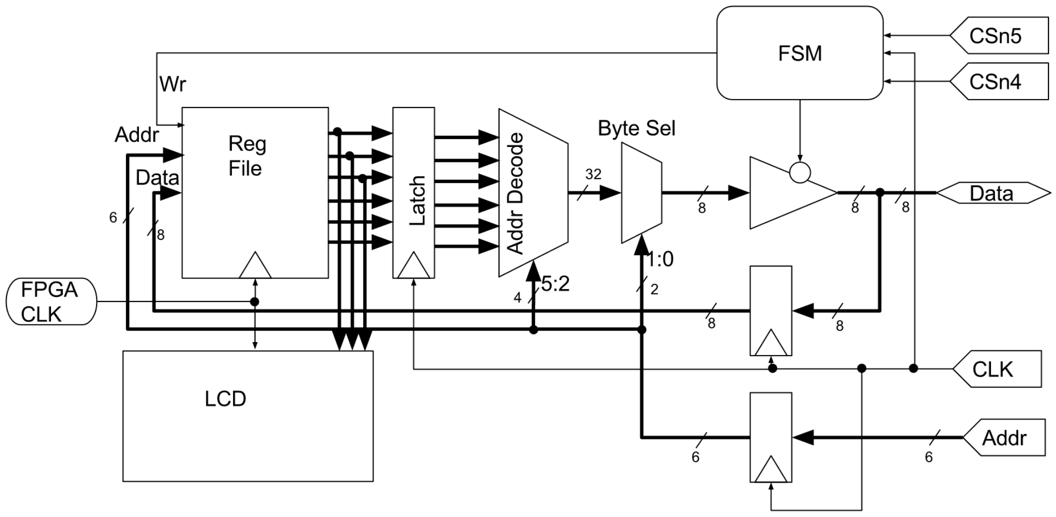


Figure 5.3: FPGA block diagram

FPGA design (i.e. the data needs to be latched by something). We investigated the possibility of latching on the chip-select, however, this would not work due to the data / address becoming invalid when the chip select is raised (i.e. the hold-time on the FPGA side would be violated). Another option was to have a high speed sampling clock on the FPGA side, which would latch the data when the the chip-select was low, however, this could lead to glitching. The simplest solution was to use the EMIFA clock that was meant for SDRAM, which is what the peripheral is clocked by. Here we were able to notice that the data is valid for half a cycle before and after the rising-edge of the EMIFA clock (determined via oscilloscope probing). For that reason, we decided to use the EMIFA clock to synchronize the interface on the FPGA side.

Due to the expected stability and drive strength of the EMIFA clock, we did not want to drive anything other than the interface part of the FPGA with the external clock. Instead, we decided to drive the actual registers with an internal FPGA clock which is substantially more stable (more drive strength, and less jitter). This design choice, however, could lead to issues with write updates from the FPGA side (if we decided to implement the FPGA writing to registers). To solve this problem, we implemented a simple latch which is clocked on the EMIFA clock. This ends up significantly expanding the amount of resources required by the design, and would be better implemented via a pipelined read in a real application. However, due to the simplicity of our implementation, the extra resource consumption was acceptable.

Finally, we implemented a very simple finite state machine which encompasses the read-/write logic described in section 4.3.

5.3 ARM CPU Side

In the FPGA-ARM block diagram, figure 5.1, the ARM hardware has a Linux kernel on it. The Linux kernel used for this project was one that was provided for the ARM CPU development board from Texas Instruments, specifically DaVinci linux-03.20.00.14 kernel. A driver was intended to go on top of the DaVinci kernel. However in the current stage of development for the FPGA-ARM CPU bridge, a kernel module is being used to confirm communication between devices rather than a driver. Finally, on top of the driver was intended to go an end user application in which the user would be able to request a specific address on the FPGA to read from an user specified address as well as write a user specified value to a user specified address.

Chapter 6

User Interface

To properly configure and interact with the external memory interface, the user interface needs to be two fold - a kernel level driver and a library. Upon driver loading, the pinmux registers need to be configured such that the pins corresponding to the EMIFA interface are set to that function. Additionally, the EMIFA interface needs to be configured to meet the expected timing requirements for the FPGA. Both of these tasks require writing to memory addresses which are only accessible in supervisor mode (i.e. kernel mode). To do this, the configuration can be added to the initialization function of a kernel level driver (a kernel module).

To actually interact with the FPGA device one the EMIFA interface is initialized, can be done via several methods. In the end, all of the methods effectively boil down to memory mapped access to physical addresses which correspond to the EMIFA interface. The memory mapping can either be done on the kernel side via the `iomapnocache()` function, or the user side via the `mmap()` function. In order to enable interrupt like behavior and more specific driver configuration, we chose to go the route of mapping the memory on the kernel side, and interfacing with a user library via a device file.

The proposed configuration is shown in figure 6.1. This interface will allow us to perform read / write operations on specific registers within the FPGA address space, as well as perform streaming read / write operations on the device in general. The application model that we had in mind is what one might expect from a piece of streaming test instrumentation, where we have some device registers (in the FPGA) which set the FPGA state. The user application is responsible for setting those registers, and will then perform regular memory transfers from the FPGA into the CPU memory. These transfers will most likely be streamed (with a FIFO like interface) and not be random accessed. To implement such a behavior with the EMIFA interface, given the setup and hardware limitations we have with the chosen development board, we can implement several "devices" via the chip select domains of the EMIFA interface.

We have effectively already created two devices, a read only device (CS5) and a write only device (CS4) due to the missing R/W and OE signals. We can use another exported chip select (CS2) to represent another read only device, which does not care about the address (i.e. a FIFO). This means that we could configure the FPGA logic to respond to the chip selects differently, with separate register block (CS4 and CS5) and FIFO block (CS2).

All of the aforementioned functionality can be encapsulated in the kernel driver module, with the exposed file descriptor acting as an interface to the user library. The user library can then abstract away most of the interface details with the kernel module and file descriptor using user level API calls.

The above was not implemented due to time constraints for this project, where the current

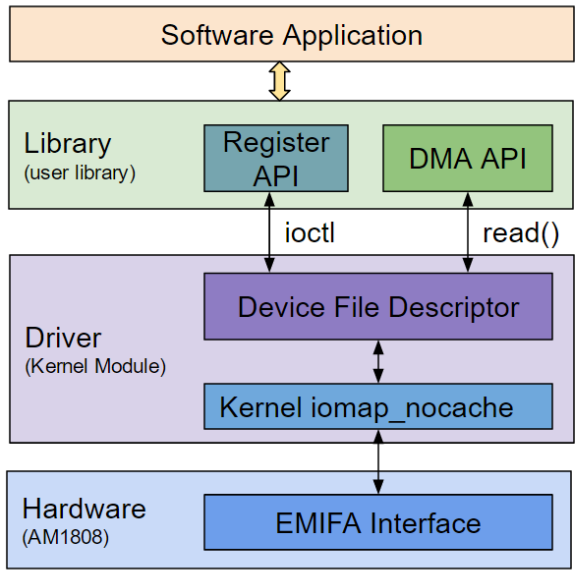


Figure 6.1: Driver Interface

state is a rough kernel module which performs EMIFA initialization, and hard coded R/W access to the EMIFA interface. Further work is required to fully implement a functional driver.

Chapter 7

Evaluation

Due to time constrains, we performed a simple test evaluation for the interface. This was done via hard coded values in our test kernel module. The values chosen were to help determine the byte order.

7.1 Write Test

We tested the write operation on the EMIFA bus via writing the value `0x8EAB33F` to the address corresponding to register 0. The LCD screen on the FPGA was used to verify the write. After some fiddling with the timing parameters, the value of `0xDAEAF33B` was consistently written to register 0 (See figure 7.1). To verify that there was no error with the LCD screen, the hex digits for the first byte were also written to the hex 7-segment displays.

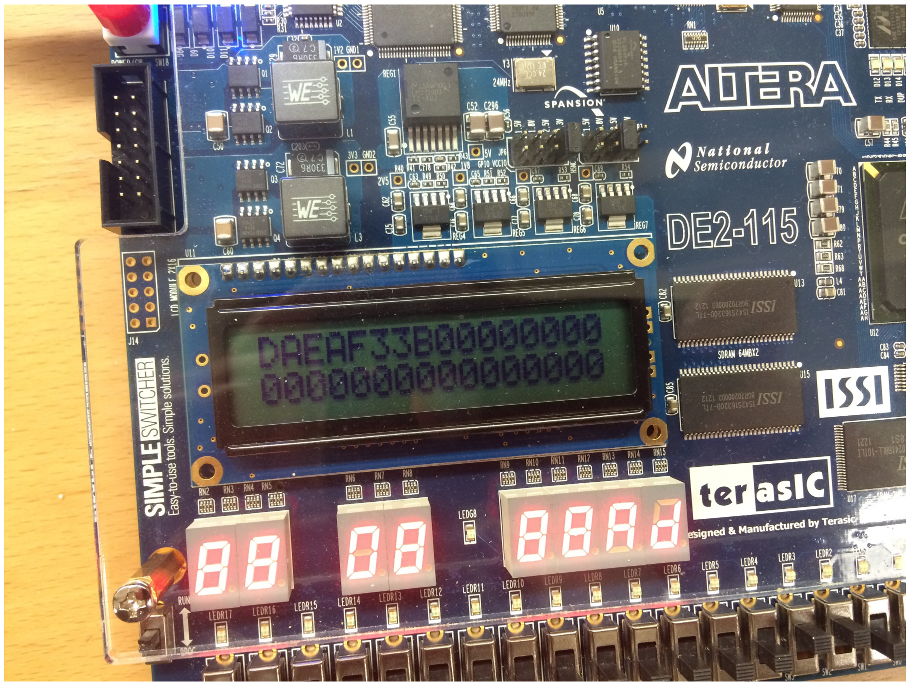


Figure 7.1: Write Result

Additionally, a value of `0x88888888` was written to test the signals, resulting in a value of `0x8A8A8A8A` showing up on the LCD screen. This indicates a stuck at 1 bug. Most likely the bug is being caused by a floating wire, which was caused by a broken connection. This has been a problem in the past, and was discussed in a previous section. Due to time constraints, further investigation was not possible.

7.2 Read Test

A read operation was then tested on register address 0, which proceeded the write to register address 0. If all went well, we would have expected to read back `0x8EAB33F`, or more specifically, or some variant of that which takes into consideration any internal pullups / floating behavior of the disconnected pin corresponding to `Data[1]`. Instead, however, we received a value of `0x3F0000BD`, which makes little sense. It is likely that this value is showing up due to timing violations, however, we could not investigate this further due to time constraints. It should be noted that timing violations here refers to expectations from the EMIFA timing diagram. The read valid data time segment may not be properly lining up with the FPGA is producing (i.e. the issue is most likely due to our logic implementation). Furthermore, this can be adjusted using the timing configuration parameters for the EMIFA interface.

Chapter 8

Conclusion

Overall, we were able to successfully configure the EMIFA interface on the AM1808 processor using the embedded linux kernel via a custom kernel module. This was verified via oscilloscope probing. Additionally, we were able to successfully write to the FPGA (we will consider it a success, given that the only problem was the stuck at 1 bug). We were unsuccessful at reading from the FPGA, however, this is most likely due to a timing issue which should be relatively easily fixed after checking the timing with an oscilloscope. Additional discussion on this topic is presented in the Evaluation section.

Future work would involve verifying that our stuck at 1 bug is caused by a loose connection, and then correcting that issue. We would then proceed to work on performing successful reads on the FPGA via the processor, which should be fairly easily to do with proper timing analysis via oscilloscope. The next step would be to tighten the read / write operations, to enable faster transactions (if possible). We would then follow that up by implementing a FIFO interface on a second chip-select via the interface (possibly the SDRAM interface, since it allows for burst reads). Finally, the remaining step is to encapsulate all of the functionality hard coded in the kernel module into the driver interface described in the Interface section. At that point, we would have a working proof of concept design for a high bandwidth streaming architecture similar to what is seen in test instrumentation.

References

[1] R. B. D. Toro, “How to interface FPGAs to microcontrollers,” 2008. [Online; accessed 23-January-2017].

[2] M. Williamson, “EMIFA Interface,” 2011. [Online; accessed 23-January-2017].

[3] W. C. ’Deffe’, “Connect a ARM Microcontroller to a FPGA using its Extended Memory Interface (EMI),” 2013. [Online; accessed 23-January-2017].

[4] “GSG: Building Software Components for OMAP-L1/AM1x,” 2013. [Online; accessed 23-January-2017].

[5] J. Preshing, “How to Build a GCC Cross-Compiler,” 2014. [Online; accessed 23-January-2017].

[6] T.I., *AM1808 ARM Microprocessor*. Texas Instruments.

[7] T.I., *AM1808/AM1810 Sitara ARM Microprocessor Technical Reference Manual*. Texas Instruments.

Appendix A

Cross-Compiler Tool-chain

In this chapter, we will describe a procedure for compiling a cross-compilation tool chain for ARM development on the AM1808 processor via an x86 host machine. The following procedure should work for any form of unix, as well as under the Cygwin environment for Windows. It will be assumed that `GCC`, `GCC-C++`, `make`, and `awk` are already installed on the host machine. This process will be based off of the one presented in [5].

A.1 Prerequisites

The following packages are required to complete the compilation process since this is done from source.

```
$ wget http://ftpmirror.gnu.org/binutils/binutils-2.24.tar.gz
$ wget http://ftpmirror.gnu.org/gcc/gcc-4.9.2/gcc-4.9.2.tar.gz
$ wget http://ftpmirror.gnu.org/glibc/glibc-2.20.tar.xz
$ wget http://ftpmirror.gnu.org/mpfr/mpfr-3.1.2.tar.xz
$ wget http://ftpmirror.gnu.org/gmp/gmp-6.0.0a.tar.xz
$ wget http://ftpmirror.gnu.org/mpc/mpc-1.0.2.tar.gz
$ wget ftp://gcc.gnu.org/pub/gcc/infrastructure/isl-0.12.2.tar.bz2
$ wget ftp://gcc.gnu.org/pub/gcc/infrastructure/cloog-0.18.1.tar.gz
```

A copy of the TI DaVinci Linux kernel will also be needed to complete this compilation. The reason for this, is that the DaVinici Linux kernel has been modified to run on the AM1808 (and similar) devices.

All of the above, with the exception of `glibc`, `gcc`, `Linux Source`, and `binutils` should be compiled on the host machine using standard compilation procedures found in their respective README or INSTALL files. They should be installed to the PATH so that they can be used by the tools in the next section.

A.2 Compilation

The following will be done assuming that the directory structure exists as below:

```
        cwd/.
                binutils-2.24/.
                build-binutils/.
                gcc-4.9.2/.
                build-gcc/.
                glibc-2.20/.
                build-glibc/.
                linux-DaVinci/.
```

With that said, the compilation process is to create the `arm-linux-gnueabi` compilers. This is done via the following sequence of commands, while the user is set to root:

```
cd build-binutils
../binutils-2.24/configure --prefix=/opt/cross --target=arm-linux-gnueabi --disable-multi
make -j4
make install

cd ../linux-DaVinci
make ARCH=arm INSTALL_HDR_PATH=/opt/cross/arm-linux-gnueabi headers_install

cd ../build-gcc
../gcc-4.9.2/configure --prefix=/opt/cross --target=arm-linux-gnueabi --enable-languages=
make -j4 all-gcc
make install-gcc

cd ../build-glibc/
../glibc-2.20/configure --prefix=/opt/cross/arm-linux-gnueabi --build=$MACHTYPE --host=ar
make install-bootstrap-headers=yes install-headers
make -j4 csu/subdir_lib
install csu/crt1.o csu/crti.o csu/crtn.o /opt/cross/arm-linux-gnueabi/lib
arm-linux-gnueabi-gcc -nostdlib -nostartfiles -shared -x c /dev/null -o /opt/cross/arm-li
touch /opt/cross/arm-linux-gnueabi/include/gnu/stubs.h

cd ../build-gcc
make -j4 all-target-libgcc
make install-target-libgcc

cd ../build-glibc/
make -j4
make install

cd ../build-gcc
make -j4
make install
```

Upon completion, all of the relevant tools should be placed in `/opt/cross/bin`, and can be added to the PATH via `export PATH=$PATH:/opt/cross/bin` in either the working terminal, or in the user's `.bashrc` file. The tools can be used by a non-root user, however, they must be compiled as root.

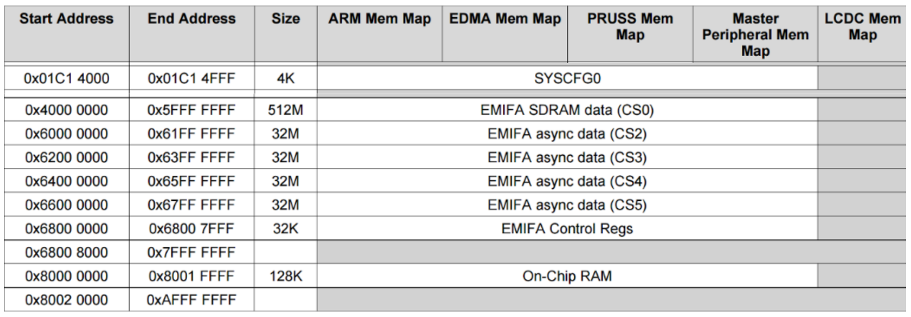
Finally, the kernel image, as well as example kernel modules and user applications can be found in [4]. The only modification is that `arm-none-linux-gnueabi-` should be replaced with `arm-linux-gnueabi-`.

Appendix B

AM1808 Configuration

B.1 AM1808 Memory Map

Figure B.1 depicts the memory map of the TI AM1808 development board. The system configuration register, SYSCFG0 contains 4K of memory. The memory map also contains four EMIFA asynchronous data registers, each containing 32M of memory . CS0 is used strictly as an EMIFA asynchronous SDRAM data and contains 512M of memory. There are 32K of memory reserved for the EMIFA control registers. There is also 128K of on-chip RAM.



Start Address	End Address	Size	ARM Mem Map	EDMA Mem Map	PRUSS Mem Map	Master Peripheral Mem Map	LCDC Mem Map
0x01C1 4000	0x01C1 4FFF	4K	SYSCFG0				
0x4000 0000	0x5FFF FFFF	512M	EMIFA SDRAM data (CS0)				
0x6000 0000	0x61FF FFFF	32M	EMIFA async data (CS2)				
0x6200 0000	0x63FF FFFF	32M	EMIFA async data (CS3)				
0x6400 0000	0x65FF FFFF	32M	EMIFA async data (CS4)				
0x6600 0000	0x67FF FFFF	32M	EMIFA async data (CS5)				
0x6800 0000	0x6800 7FFF	32K	EMIFA Control Regs				
0x6800 8000	0x7FFF FFFF						
0x8000 0000	0x8001 FFFF	128K	On-Chip RAM				
0x8002 0000	0xAFFF FFFF						

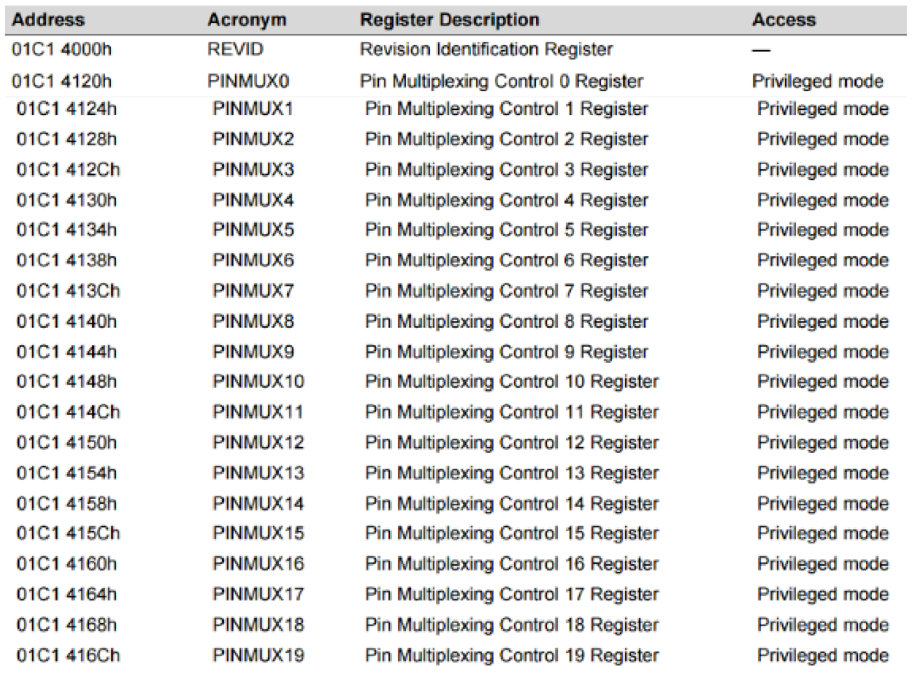
Figure B.1: AM1808 Memory Map

B.2 Pin Multiplexing

The TI AM1808 ARM CPU development board has twenty pin multiplexing control register as well as a register for the revision ID. These twenty-one registers can be seen depicted in the IO Selection/Configuration diagram, figure B.2. As the title suggest, these twenty-one registers are used to select and configure the input as well as ouput.

B.3 EMIF Timing and Mode Selection

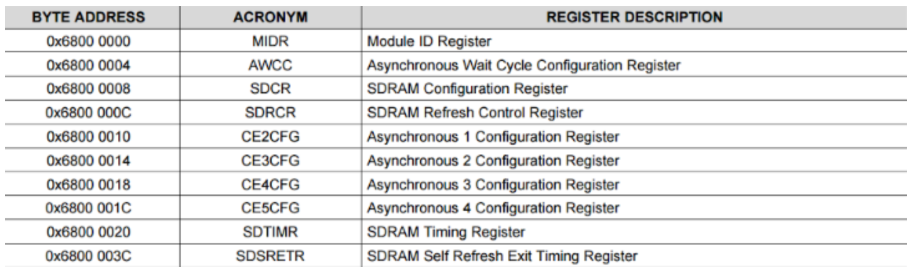
The TI AM1808 has ten timing and mode selection registers that will be used for the FPGA-CPU bridge. Figure B.3, the Timing and Mode Selection Registers diagram, depicts the byte address, acronym, and description of each of these ten registers. There is one asynchronous wait cycle configuration register referred to as AWCC, one module ID register referred to as



Address	Acronym	Register Description	Access
01C1 4000h	REVID	Revision Identification Register	—
01C1 4120h	PINMUX0	Pin Multiplexing Control 0 Register	Privileged mode
01C1 4124h	PINMUX1	Pin Multiplexing Control 1 Register	Privileged mode
01C1 4128h	PINMUX2	Pin Multiplexing Control 2 Register	Privileged mode
01C1 412Ch	PINMUX3	Pin Multiplexing Control 3 Register	Privileged mode
01C1 4130h	PINMUX4	Pin Multiplexing Control 4 Register	Privileged mode
01C1 4134h	PINMUX5	Pin Multiplexing Control 5 Register	Privileged mode
01C1 4138h	PINMUX6	Pin Multiplexing Control 6 Register	Privileged mode
01C1 413Ch	PINMUX7	Pin Multiplexing Control 7 Register	Privileged mode
01C1 4140h	PINMUX8	Pin Multiplexing Control 8 Register	Privileged mode
01C1 4144h	PINMUX9	Pin Multiplexing Control 9 Register	Privileged mode
01C1 4148h	PINMUX10	Pin Multiplexing Control 10 Register	Privileged mode
01C1 414Ch	PINMUX11	Pin Multiplexing Control 11 Register	Privileged mode
01C1 4150h	PINMUX12	Pin Multiplexing Control 12 Register	Privileged mode
01C1 4154h	PINMUX13	Pin Multiplexing Control 13 Register	Privileged mode
01C1 4158h	PINMUX14	Pin Multiplexing Control 14 Register	Privileged mode
01C1 415Ch	PINMUX15	Pin Multiplexing Control 15 Register	Privileged mode
01C1 4160h	PINMUX16	Pin Multiplexing Control 16 Register	Privileged mode
01C1 4164h	PINMUX17	Pin Multiplexing Control 17 Register	Privileged mode
01C1 4168h	PINMUX18	Pin Multiplexing Control 18 Register	Privileged mode
01C1 416Ch	PINMUX19	Pin Multiplexing Control 19 Register	Privileged mode

Figure B.2: IO Selection/Configuration

MIDR, one SDRAM configuration register referred to as SDCR, one SDRAM refresh register referred to as SDRCR, one SDRAM timing register referred to as SDTIMR, and one SDRAM self refresh external timing register referred to as SDSRETR. There are also four asynchronous configuration registers referred to as CE2CFG through CE5CFG.



BYTE ADDRESS	ACRONYM	REGISTER DESCRIPTION
0x6800 0000	MIDR	Module ID Register
0x6800 0004	AWCC	Asynchronous Wait Cycle Configuration Register
0x6800 0008	SDCR	SDRAM Configuration Register
0x6800 000C	SDRCR	SDRAM Refresh Control Register
0x6800 0010	CE2CFG	Asynchronous 1 Configuration Register
0x6800 0014	CE3CFG	Asynchronous 2 Configuration Register
0x6800 0018	CE4CFG	Asynchronous 3 Configuration Register
0x6800 001C	CE5CFG	Asynchronous 4 Configuration Register
0x6800 0020	SDTIMR	SDRAM Timing Register
0x6800 003C	SDSRETR	SDRAM Self Refresh Exit Timing Register

Figure B.3: Timing and Mode Selection