

DE1-SOC FPGA: Independent study



Submitted by:
Ahmed Kamel

Project Advisor:
Dr. Bruce Land

Introduction	2
FPGA	2
Purpose	3
Hard Processor vs Soft Processor	4
Hardware / Software Integration	5
Hardware / Software Connection	5
Applications	7
Virtual Address	7
Software Control of Lights	8
Hardware	8
Programming the FPGA	13
Software	14
VGA Control From HPS	15
University Programs	15
Classical Approach	16
Evaluation	18
Synthesis - Area	18
Synthesis report	20
Conclusion	21
References	23
Data Sheets	23
Appendix	23

Introduction

The DE1-SOC field programmable gate array (FPGA) is the focus of this independent study. The overall goal is to see if this DE1-SOC will work as a sufficient replacement for the DE2 and DE2-115 FPGA's currently in use for the ECE 5760 course. The reason the DE1-SOC was chosen is due to the embedded processor it has available. This project aims at redesigning the course labs and observing the difficulty level / learning opportunities available.

FPGA

The FPGA that was utilized for the course of this project was the DE1-SoC. which is equipped with a Hard Processor System (HPS) that is a Dual-core ARM Cortex-A9. The HPS made this FPGA an ideal choice for this project to allow students to explore the ever growing field of embedded systems. Each core runs at 800MHz and is equipped with 1GB of DDR3 SDRAM. The FPGA is part of the Cyclone V SoC family from the Altera family. The FPGA is equipped for 85K programmable logic elements and 4,450 Kbits of embedded memory, 64 MB off-chip SDRAM, 6 fractional PLL's, and 2 hard memory controllers. Another key reason the DE1-SoC was chosen was due to the video support that it contains. It has a VGA DAC with a VGA-out connector, as well as a TV decoder with a TV-in connector. All of the connections of the DE1-SOC can be seen in figure 1. This figure also shows which hardware components are connected to the ARM core and the FPGA fabric.

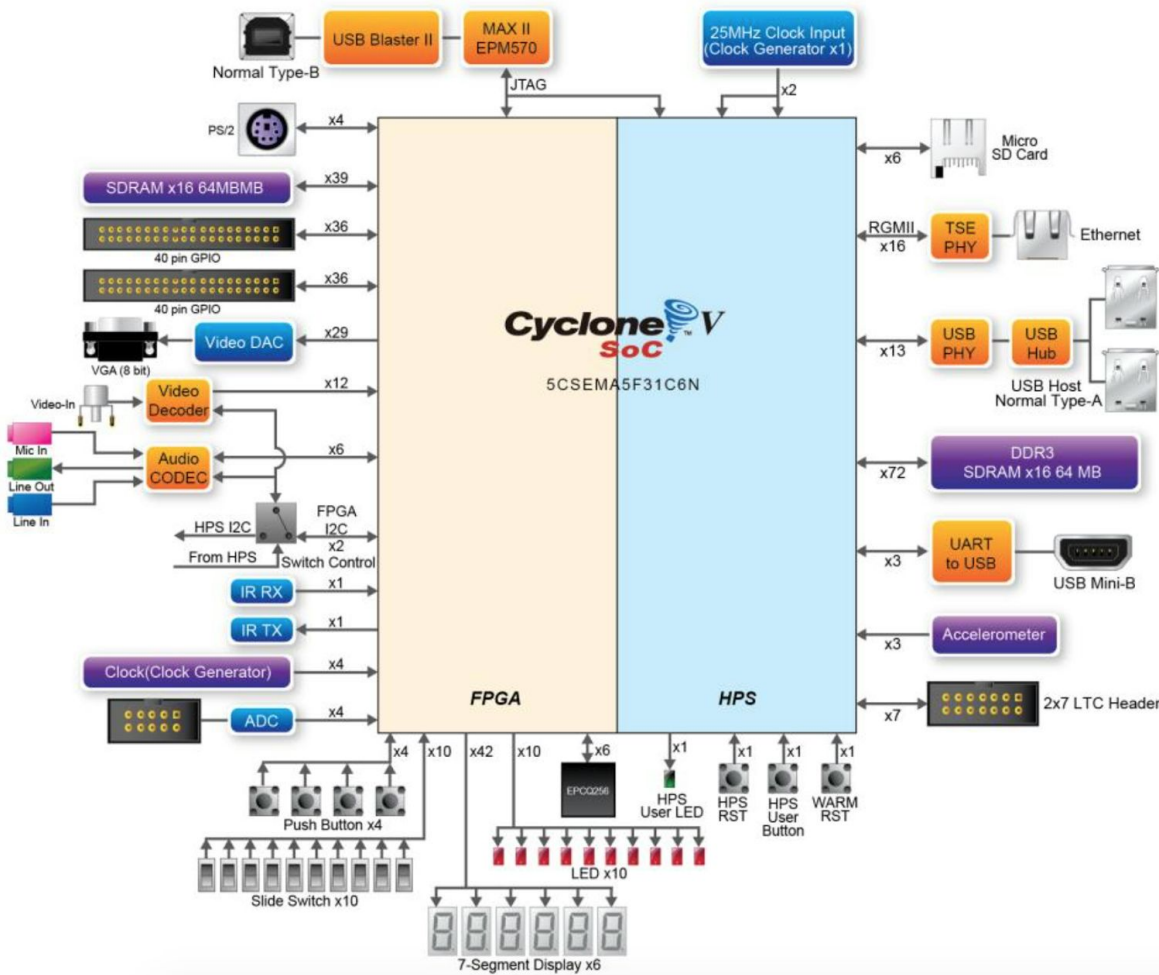


Figure 1: DE1-SoC layout
User Manual [5]

Purpose

One of the reasons the DE1-SOC was chosen over other FPGA's is due to the ARM core that it has available. The embedded systems domain is quickly growing and the ability to have one device that utilizes both an embedded processor and an FPGA opens a significant amount of opportunities for any programmer. The flexibility of an FPGA with the ability to specialize a processor for specific tasks arms the developer with a powerful tool as well. For example the developer has the ability to pick and choose peripherals that get attached to the hard processor to maximize performance as much as possible. Using

the processor to control the peripherals attached the FPGA allows for ease of development of projects (excluding the overhead associated with setting up the initial framework).

Another important factor of employing an the DE1-SOC is the evaluation of the results. The students will be able to observe the power consumption, performance tradeoffs, and area when using a hardware accelerator in contrast to a purely software design. This is possible by using the synthesis tools available in Quartus.

Hard Processor vs Soft Processor

When an HDL developer comes to the point where a processor would be beneficial, two options are available; a hardware processor or a software processor. A software processor is typically what has been used in the past in the ECE 5760 course (the NIOS II processor). In some cases however, the processor needs to be created from scratch and synthesized onto the FPGA. The FPGA fabric would be molded to provide this “soft” processor with memory, memory controllers, busses, peripherals, etc... all of which use up the FPGA’s logic elements. The benefit of this is that the developer is able to create a processor as powerful as needed for the design. However, there is still the disadvantage that goes alongside the soft processor, and that is performance. The NIOS II processor’s clock rate is slow when compared to that of the ARM cores, in the past ECE 5760 has run this processor at speeds below 100 MHZ. There is also the factor of bulkiness associated with the NIOS II processor, it has a large amount of overhead to provide general purpose support.

On the other end of the spectrum is the hardware processor. This is a physical processor that is embedded into the FPGA fabric for use by the developer; for example the ARM core attached to the DE1-SOC. The hard processor does not need to be synthesized into logic elements thus freeing up some resource, and can run at a clock rate much faster then the processor. Similar to the “soft” processor the “hard” processor can be connected with as many peripherals from the FPGA side as needed.

Hardware / Software Integration

One of the primary goals of the independent study is to explore the hardware / software integration of using the DE1-SOC FPGA. When the term hardware is used in this sense, it is referring to any code that is developed in verilog and will run strictly on the FPGA side. The term software refers to any code that is developed in C that targets the ARM core. The connection between these two components is a delicate one with multiple factors that must be taken into account, with timing being one of the most important.

Hardware / Software Connection

The FPGA fabric and the ARM core are connected through two Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) bridge. Although these two components can function completely independently, communication between the ARM core and FPGA fabric can be a bottleneck for the overall system. That is why they are connected with two high speed 128-bit AMBA AXI bus bridge called HPS-to-FPGA and FPGA-to-HPS. The datapath width for both bridges is not fixed to 128 bits, it can be configured via QSYS to 32, 64, and 128. By having this variable data width, the bridge can be tuned to for maximum performance when communication between the FPGA fabric and the HPS L3 occurs. Figure 2 shows the interconnections between the FPGA and the HPS system. It also shows how these two components are connected the global shared memory.

Block Diagram: SoC Hardware Architecture

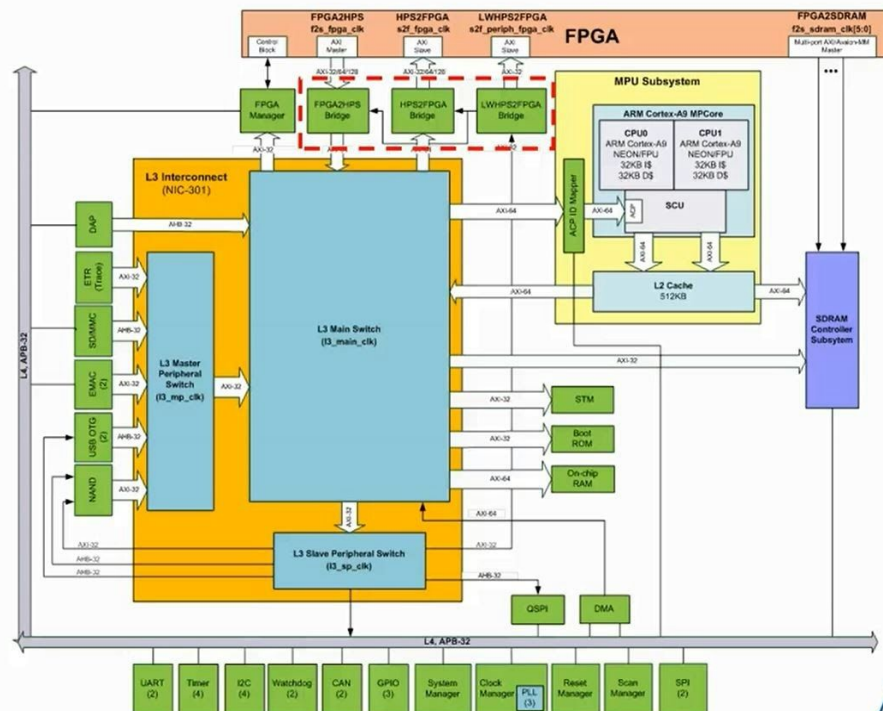


Figure 2: The connection between the FPGA, HPS, L3 Cache
[\[youtube video link here\]](#)

In addition to the two high speed bridges, a lightweight HPS-to-FPGA bridge is also available. This bridge provides access to the control and status register (CSR) slave ports in the FPGA fabric. The lightweight HPS-to-FPGA bridge is 32-bits wide because usually CSR slaves are only 32 bits. By using the lightweight port for control of slave ports such as PIO ports, the HPS-to-FPGA bridge can handle the high speed burst transactions required for large data transfers. Figure 3 shows a demonstration of how the lightweight AXI bus can be utilized. [\[cortex 9 pdf\]](#)

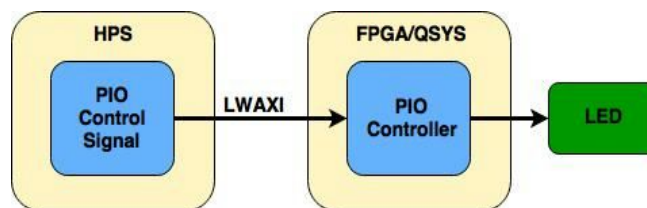


Figure 3: Demonstration of LWAXI bus

Applications

Virtual Address

The software that is ran on Linux (the ARM side) does not have access to the physical memory of any peripherals running on the HPS; such as the Lightweight HPS-to-FPGA bridge. So in order to work around this restriction, the physical address is mapped in the user space to give the developer access to the peripherals connected. These peripherals can vary in their application, but for the purpose of this independent study these peripherals were simply PIO ports. The entire span of the CSR can be mapped to allow access to any peripheral slaves within the span. The following formula found in the DE1-SOC user manual can be used to access the “Virtual Address” of these peripherals, this example is used for LED access.

```
h2p_lw_led_addr = virtual_base + ( ( unsigned long )( ALT_LWFPGASLVS_OFST + LED_PIO_BASE ) & ( unsigned long )( HW_REGS_MASK ) )
```

This formula references macro's that are automatically generated using a scripted provided on the DE1-SOC REV. E cd. The only term/macro's not created by the script is the “virtual_base” and the HW_REGS_SPAN/MASK term.

Virtual base is defined as:

```
virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, HW_REGS_BASE );
```

The HW_REGS_BASE is defined as:

```
#define HW_REGS_SPAN (ALT_STM_OFST)
```

The HW_REGS_SPAN is defined as:

```
#define HW_REGS_SPAN 0x0400000
```

The HW_REGS_MASK that is used when creating the h2p_lw_led_addr in the example is defined as:

```
#define HW_REGS_MASK (HW_REGS_SPAN - 1)
```

All peripherals that sit within the span address can be accessed in the same fashion as the LED address above.

Software Control of Lights

Hardware

One of first examples that was created was used to test how the HPS can control the hardware on the FPGA side such as LED's and SSEG display. This example explored various paths to allow this type of control, however the end example just utilized a basic PIO port in QSYS. The overall goal of the example was to control the LED's via the Dip-Switches and the HPS.

From the very beginning the first step is to create the QSYS IP component for the HPS. This is set up with specific parameters that will allow for complete customization of the processor. Figures 4 - 7 show all of the different settings required for the HPS IP.

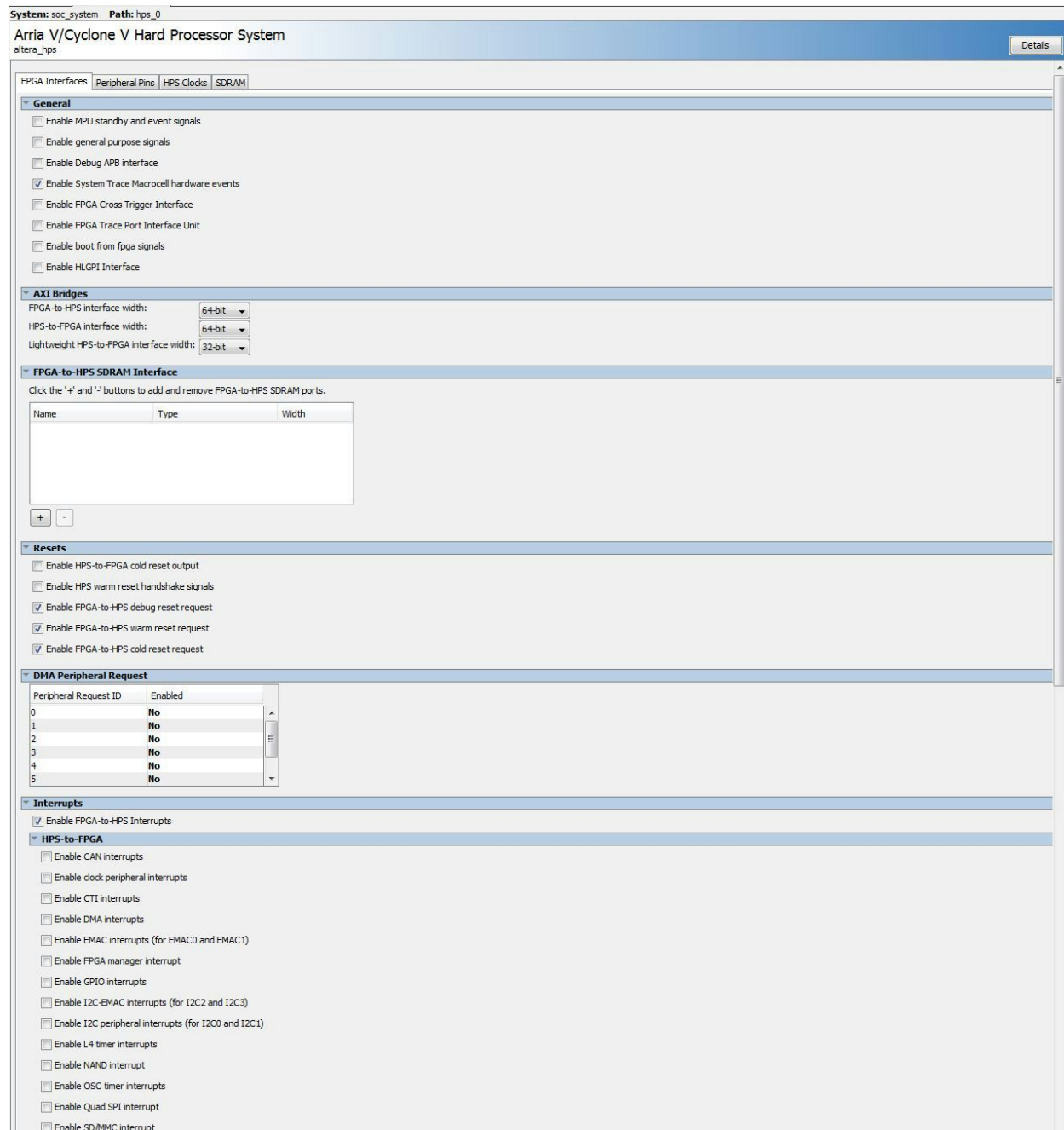


Figure 4: HPS screenshot - FPGA Interface

Details
Parameters

system: soc_system
Path: hps_0

Arria V/Cyclone V Hard Processor System
altera_hps
Details

FPGA Interfaces
Peripheral Pins
HPS Clocks
SDRAM

Ethernet Media Access Controller
EMAC0 pin: Unused
EMAC0 mode: N/A
EMAC1 pin: HPS I/O Set 0
EMAC1 mode: RGMII

NAND Flash Controller
NAND pin: Unused
NAND mode: N/A

Quad SPI Flash Controller
QSPI pin: HPS I/O Set 0
QSPI mode: 1 SS

SD/MMC Controller
SDIO pin: HPS I/O Set 0
SDIO mode: 4-bit Data

USB Controllers
USB0 pin: Unused
USB0 PHY interface mode: N/A
USB1 pin: HPS I/O Set 0
USB1 PHY interface mode: SDR with PHY clock output mode

SPI Controllers
SPIM0 pin: Unused
SPIM0 mode: N/A
SPIM1 pin: HPS I/O Set 0
SPIM1 mode: Single Slave Select
SPIS0 pin: Unused
SPIS0 mode: N/A
SPIS1 pin: Unused
SPIS1 mode: N/A

UART Controllers
UART0 pin: HPS I/O Set 0
UART0 mode: No Flow Control
UART1 pin: Unused
UART1 mode: N/A

I2C Controllers
I2C0 pin: HPS I/O Set 0
I2C0 mode: I2C
I2C1 pin: HPS I/O Set 0
I2C1 mode: I2C
I2C2 pin: Unused
I2C2 mode: N/A
I2C3 pin: Unused
I2C3 mode: N/A

CAN Controllers
CAN0 pin: Unused
CAN0 mode: N/A
CAN1 pin: Unused
CAN1 mode: N/A

Trace Port Interface Unit
TRACE pin: Unused
TRACE mode: N/A

Figure 5: HPS Screenshot - Peripheral Pins

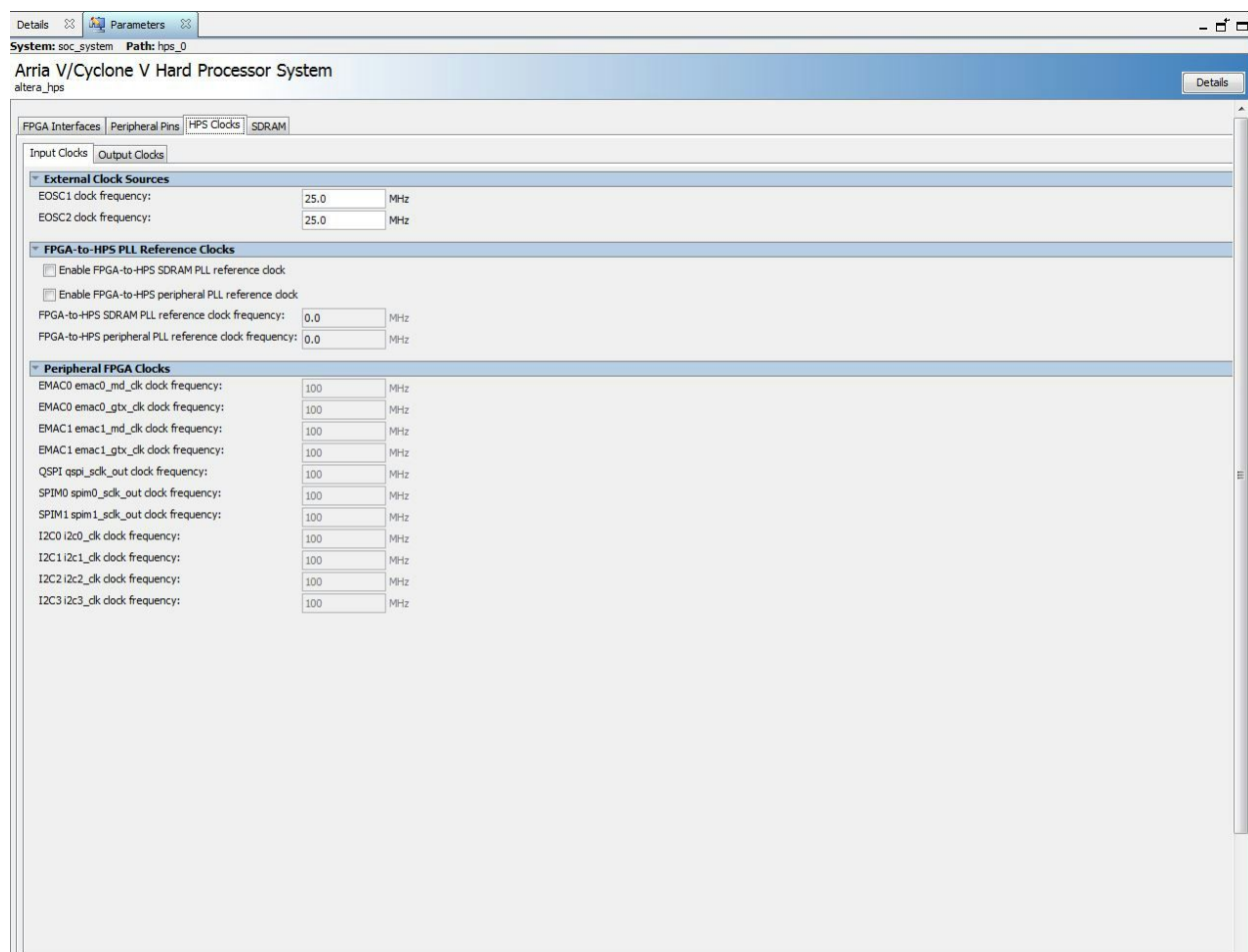


Figure 6: HPS Screenshot - HPS Clock

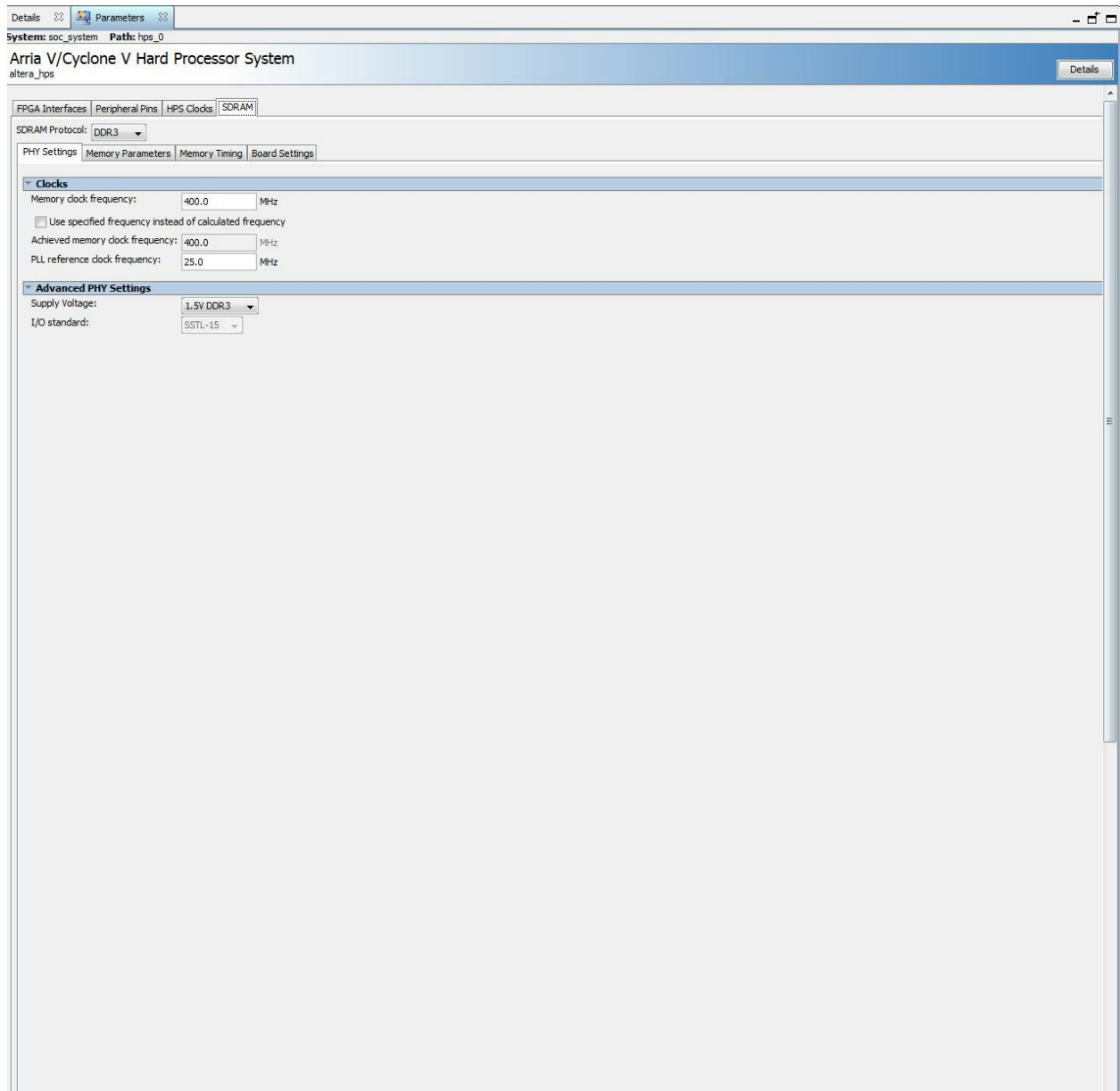


Figure 7: HPS Screenshot - SDRAM

The next step was to add the PIO ports required for controlling the LED's, these were declared as an output 10 bit port. Next was the PIO ports for the switches, this was declared as an input 10 bit port. The lightweight AXI master port of the HPS IP was connected to the peripherals. The reason the lightweight port was used is due to the fact that their will only be control signals for these ports. From there, the hardware was pretty much completed. All that remained was the instantiation of the SOC_System in the

System Contents										
System: soc_system Path: clk_0										
Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
<input checked="" type="checkbox"/>		clk_0	Clock Source							
		clk_in	Clock Input	clk	exported					
		clk_in_reset	Reset Input	reset						
		clk	Clock Output		clk_0					
		clk_reset	Reset Output							
				Double-click to export						
				Double-click to export						
<input checked="" type="checkbox"/>		hps_0	Arria V/Cyclone V Hard Processor System							
		f2h_cold_reset_req	Reset Input	hps_0_f2h_cold_reset_req						
		f2h_debug_reset_req	Reset Input	hps_0_f2h_debug_reset_req						
		f2h_warm_reset_req	Reset Input	hps_0_f2h_warm_reset_req						
		f2h_stm_hw_events	Conduit	hps_0_f2h_stm_hw_events						
		memory	Conduit	memory						
		hps_io	Conduit	hps_0_hps_io						
		h2f_reset	Reset Output	hps_0_h2f_reset						
		h2f_axi_dclk	Clock Input		clk_0					
		h2f_axi_master	AXI Master		[h2f_axi_dclk]					
		h2f_axi_slave	AXI Slave		clk_0					
		h2f_hw_axi_dclk	Clock Input		[f2h_axi_dclk]					
		h2f_hw_axi_master	AXI Master		clk_0					
		f2h_irq0	Interrupt Receiver		[h2f_hw_axi_dclk]					
		f2h_irq1	Interrupt Receiver							
				Double-click to export						
<input checked="" type="checkbox"/>		pio_led	PIO (Parallel I/O)							
		clk	Clock Input		clk_0					
		reset	Reset Input		[clk]					
		s1	Avalon Memory Mapped Slave							
		external_connection	Conduit	pio_led_external_connection		0x0000_0020	0x0000_002f			
<input checked="" type="checkbox"/>		pio_sw	PIO (Parallel I/O)							
		clk	Clock Input		clk_0					
		reset	Reset Input		[clk]					
		s1	Avalon Memory Mapped Slave							
		external_connection	Conduit	pio_sw_external_connection		0x0000_0010	0x0000_001f			

Programming the FPGA

1. click Auto Detect in the programmer window
2. Choose the device family for the FPGA (for the board used it was 5CSEMA5F31)

At this point there should be two devices in the tool chain, one called SOCVHPS, and another with the device family name.

3. Right click the device and select Edit → Change File (show in the figure 9)

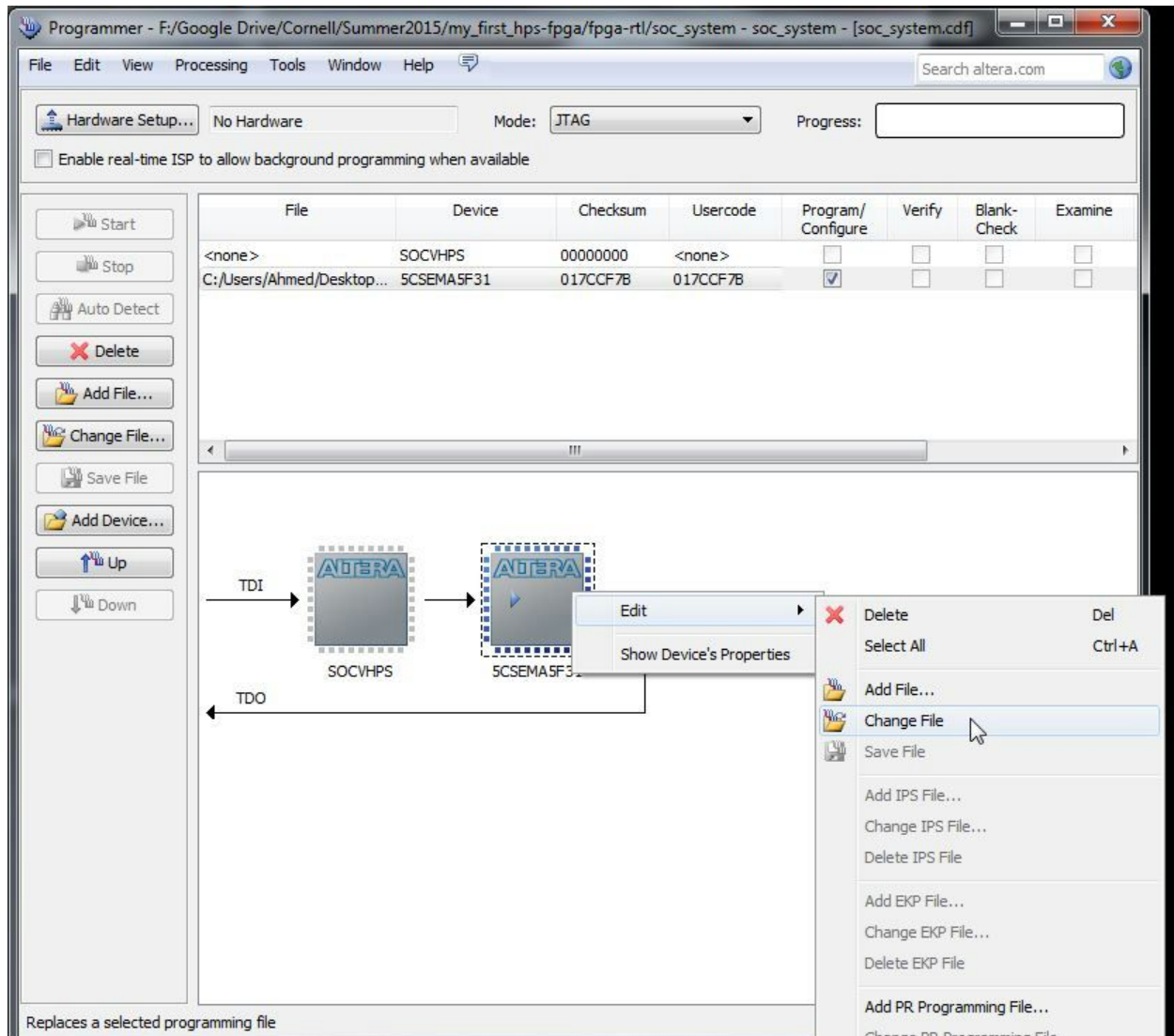


Figure 9: Step 3 in programming the FPGA

4. Select the appropriate SOF file in the pop up window
5. Select Start to program the board.

Software

The next aspect of this was to write the C code that will run on the ARM core. This code was developed in notepad++ on a windows platform and cross-compiled to run on the ARM core using a makefile found on the DE1-SOC REV. E cd. The makefile from the CD was modified to adapt a more organized directory where the source code and header files are located in separate folders; as well as the binaries

produced. The main source code used a header file that was generated by a script called “generate_hps_qsys_header.sh”. This header file contained macros for all of the memory addresses that were assigned in the QSYS project. Using these macro’s the virtual address space was mapped and ready to be utilized. The next step was to create pointers that targeted the PIO ports memory locations using the equation described in the previous section. At this point the code contains two pointers, one for the LED’s and one for the Dip-Switches. The next step was to simply assign the value of Dip-Switches pointer to the value of the LED’s pointers in an infinite loop. This allowed the flip of the switch to control the LED’s.

VGA Control From HPS

Another important feature of the DE1-SOC that needs to be explored is the VGA DAC. The ability to control the VGA through the ARM core would allow for new and interesting designed to be developed. In the past the course as typically used the NIOS II processor to allow students to develop C code to run on the FPGA. As stated before however, the NIOS II processor causes a hit to performance because of the large amount of overhead required. Also the there are FPGA resources such as logic blocks that are utilized for the synthesis of the NIOS II processor, which increase the overall space of the design. That is why if the same projects that were created to run on the NIOS II processor can be ported over to the ARM core; the overall design would perform better, while giving students a great learning opportunity.

University Programs

One of the ideal aspects of using the NIOS II processor was the library supports that came with it. The University Programs IP include a series of QSYS components and C functions that allow for the NIOS II to control the VGA screen seamlessly. The QSYS components involve the Pixel Buffer and the Character Buffer that were previously used in the past. These components work by utilizing the double buffering technique. This techniques allows the developer to write data to one of the buffers, the back buffer, and swapping it with the other at the desired refresh rate. One of the greatest challenges faced was attempting

to port these functions and components to utilize the ARM core instead of the NIOS II processor. The hardware side of the conversion was not too daunting, however the software side is where the major issues arose. The functions that were utilized to handle the buffer read/writes and buffer swaps required the Board Support Package (BSP) that gets created with the NIOS II processor. Since the sole purpose of this was to replace the NIOS II processors creation, this was not acceptable. After several modifications to the makefile and attempting to work around the requirements, it became clear that it is not worth the hassle; another approach was required.

Classical Approach

The second approach was to use the same techniques as in the early labs of ECE 5760 to control the monitor, where HDL code is used to instantiate a chunk of memory and control the signals going directly to the the vga monitor.

The most efficient manner in controlling the VGA DAC through the ARM core was by utilizing basic PIO ports to communicate with the ARM core. The hardware side of the design consists of a Reset Delay module, a Phase Lock Loop (PLL), M10k blocks (the Buffer), a VGA Controller imported from Lab 1 of ECE 5670, and the last component of the hardware was an FSM.

The Reset Delay module was used to automatically reset the PLL on boot. The PLL module was created from the IP Catalog to provide the system with clocks that would control the VGA controller, Memory, and the FSM. The PLL also provided an out of phase clock that ran the VGA Monitor. The VGA controller is used to assign all of the appropriate signals on the VGA_RGB lines as well as the sync signals. The last component of the hardware was the FSM, which is used to control the timing. The FSM creates the necessary 1 cycle delay for memory read and writes to go through.

There are few PIO ports connected to the hardware. These ports are the address, data, and write enable for memory. Initially the system was created using 1 bit color (black and white) by allocating $640 \times 480 = 307200$ bits of memory of M10K blocks. Once the bugs in the FSM were corrected and the system was

functioning with properly with 1 bit color, memory was upgraded to 8-bits. So now $640 \times 480 \times 8 = 2457600$ bits of memory were allocated (exhausting 60% M10K blocks available). The entire QSYS system can be seen in figure 10.

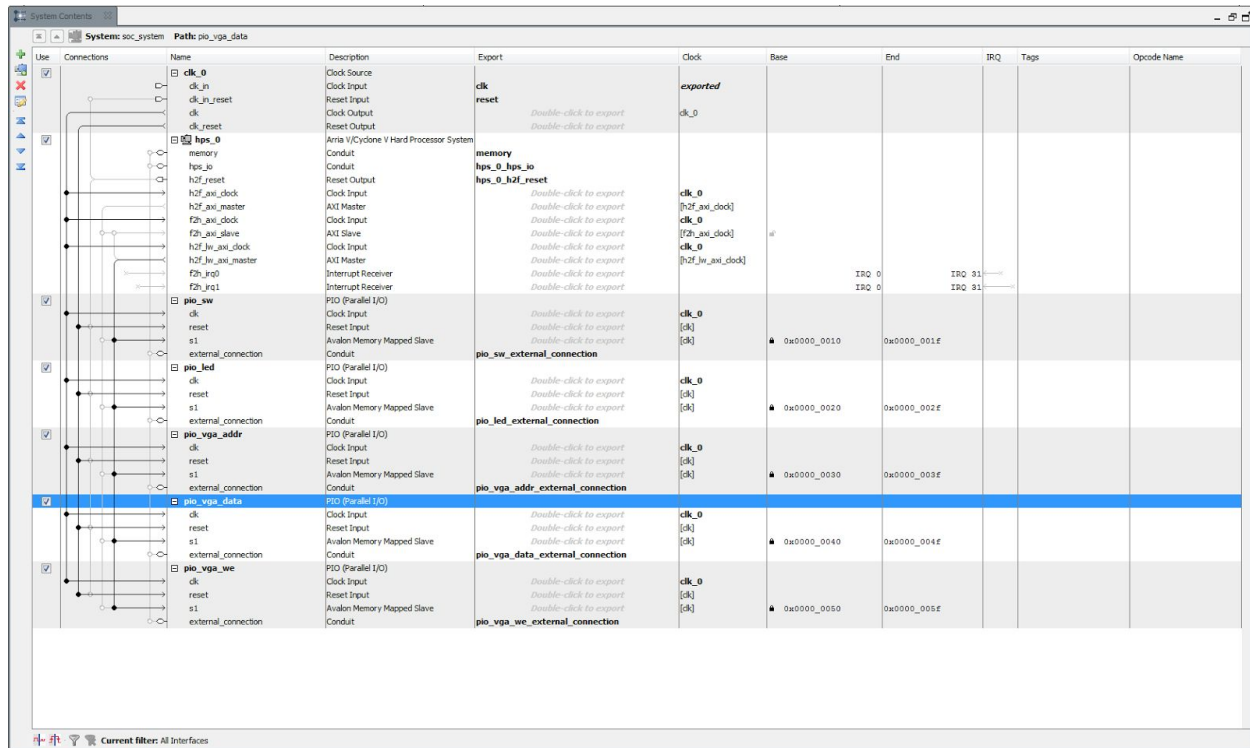


Figure 10: Screenshot of VGA QSYS system

On the software side of the design the only thing software was responsible for was setting up the virtual memory map (as explained in the LED - SW example), and setting the address, data, and write enable for the VGA buffer. Which is summed up into one function that gets called with a set of coordinates and color. Similar to Lab 1 from ECE 5760, the VGA memory system was set up in an X,Y coordinate system where 0,0 is the top left of the monitor.

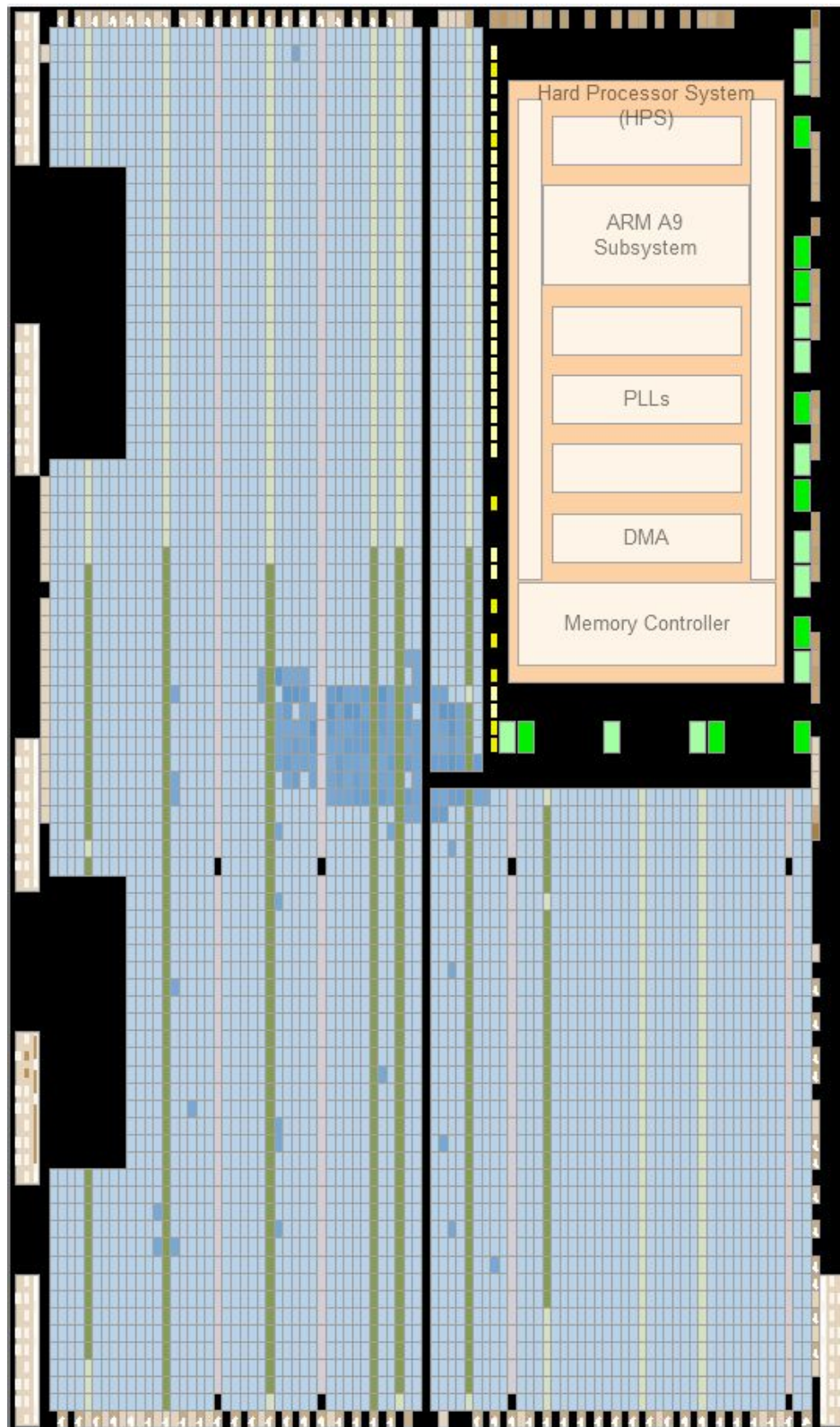
The function in software that sets the address, data, and write enable concatenates the X,Y coordinates into one address before sending it off to the hardware. This is done by shifting the X coordinate over by 9 and adding the Y coordinate; this provides the address in 18 bit notation.

Evaluation

Below are the statistics from compiling the VGA color example. Figure 11 shows that the logic utilization of the design is very small, which makes sense because the design centered around using the ARM for all logical operations. The hardware was simply responsible for making the appropriate connections. The other important factor in observed in table 1 is the the memory allocation. The design uses M10K Blocks to hold memory instead of SDRAM; which is not the most efficient of designs but there is still a significant amount of M10K blocks remaining for future additions.

Synthesis - Area

Figure 11: Area usage of VGA project



Synthesis report

Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	935 / 32,070 (3 %)
Total registers	1298
Total pins	368 / 457 (81 %)
Total virtual pins	0
Total block memory bits	2,457,600 / 4,065,280 (60 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1 / 6 (17 %)
Total DLLs	1 / 4 (25 %)

Table 1: Synthesis Report

In order to see if this design can be used for any practical application a quick test was written. This tested involved writing all 307,200 pixels of the monitor several times over and observing how much time it would take. The easiest method to implement this design was two loops, one for the rows and one for the columns. The same operation was performed 150 times to get an accurate average on how long it would take. For the 150 iterations, it took a total time of 75 seconds; which comes out to roughly 500 milliseconds per iteration (500 milliseconds per frame). This operation takes much less time if the items being drawn are smaller; so the test was modified to observe different box sizes. Table 2 shows how different sizes affect the frame draw time:

Size	Time per Iteration (s)	Total Time (s)
50x50	0.004067	0.610000
100x100	0.016267	2.440000
200x200	0.065200	9.780000
400x300	0.195667	29.350000
640x480	0.500667	75.100000

Table 2: Timing of various sized box

From these results one can see that animated projects, such as the lander game from Lab 3 of ECE 5760, can be created using this foundation.

Conclusion

The few experiments that were created and ran on the FPGA prove that it is capable of being utilized for the ECE 5760 course. The independent study has proved that the DE1-SOC would make a fine substitute for the DE2-115 FPGA due to its adaptability for the previous labs, as well as offering new interesting challenges for students to overcome. There are however downfalls that come with the DE1-SOC that need to be worked around. Using the ARM core as a replacement for the NIOS II soft processor has added certain difficulties to the labs, but these difficulties are a great learning opportunity. The ARM core will cause the students to be more concerned with the timing of the entire system because now there are two completely independent components working at the same time. There is always room for improvement with every experiment, with the DE1-SOC one of the biggest issues was compile time. On average even the simplest designs would take more than 5 minutes to compile. This is largely due to the HPS system that gets instantiated; however, there should be ways to reduce this. Another possible opportunity to explore with the DE1-SOC is to construct a computationally heavy experiment and explore

the tradeoff of using the ARM core in parallel with the FPGA. These tradeoffs would include performance, power consumption, memory allocation, and bottleneck points that are slowing the computation. One of the reasons this would be an interesting experiment is because developing complex algorithms in C versus HDL is much simpler. If the hit to performance and power are not severe, then this type of technology will be very important to learn as new engineers.

References

1. Altera Corp. "Configuring HPS to FPGA and FPGA to HPS Bridges in Altera SoCs." *YouTube*. YouTube, 13 Oct. 2013. Web. 04 Aug. 2015.
2. Altera Corp. "SoC FPGA ARM Cortex-A9 MPCore Processor Advance Information Brief." (n.d.): n. pag. *Altera*. 2012. Web. 15 July 2015.
3. Fletcher, Bryan H. "FPGA-based Embedded Processor." *Introduction to Embedded System Design Using Field Programmable Gate Arrays*(2009): 81-101. *Xilinx*. 2005. Web. 15 July 2015.

Data Sheets

1. Cyclone V SoC Development Board - Reference Manual:
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/rm_cv_soc_dev_board.pdf
2. DE1-SoC Manual. Rev E
<http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836&PartNo=4>

Appendix

Source code for hardware and software can be found at:
https://github.com/ayk33/my_first_hps-fpga