
ECE 4999: INDEPENDENT STUDY ON
PORTING AMBER CPU TO DE1-SOC AND
ALTERA BUS

Mohammad Saifee Dohadwala

Under the supervision of: Professor Bruce R. Land
Fall 2016

Table of Contents

Table of Figures	2
1. Introduction.....	3
2. FPGA	3
3. Amber Core.....	5
3.1 Amber 23 Pipeline Architecture	6
3.2 Registers.....	6
3.3 Cache.....	7
4. Amber FPGA system	7
4.1 Clocks and Reset.....	8
4.2 BOOT Memory	10
4.3 Main Memory	12
4.4 Unified Instruction and Data Cache.....	13
4.5 Debug.....	17
4.6 UART.....	17
4.7 Pin Mapping.....	17
5. System Addresses	18
5.1 Address Map	18
6. Software Support	18
6.1 Installing the compiler	18
6.4 Standalone Application.....	22
6.5 Linux.....	22
7. Major Obstacles	24
8. Acknowledgement	25
9. Conclusion	25
A. References.....	25

Table of Figures

Figure 1: DE1-SoC development board.....	4
Figure 2: Block diagram of DE1-SoC.....	4
Figure 3: Amber 23 Core pipeline stages.....	5
Figure 4: Amber 23 Core code Structure	7
Figure 5: Amber FPGA system.....	8
Figure 6: ALTPLL input clock settings	9
Figure 7: ALTPLL control signals.....	9
Figure 8: ALTPLL output clock settings	10
Figure 9: Boot Memory configuration wizard	11
Figure 10: Boot Memory port configuration.....	11
Figure 11: Boot Memory Initialization	12
Figure 12: Main Memory Initialization	12
Figure 13: Data Array size configuration	13
Figure 14: Data Array port configuration	14
Figure 15: Data Array Initialization.....	15
Figure 16: Tag Array size configuration.....	15
Figure 17: Tag Array port configuration.....	16
Figure 18: Tag Array Initialization	16
Figure 19: Pin Mapping of Amber FPGA System on DE2-115 board	17
Figure 20: Steps to compile bootloader serial.....	19
Figure 21: bootloader options	20
Figure 22: .mem file to .mif file conversion	21
Figure 23: Hello-world application code	22
Figure 24: Steps to compile hello-world application	22
Figure 25: Linux boot print on Amber FPGA system.....	24
Table 1: Register Sets	6
Table 2: Status Bits - part of the PC.....	6
Table 3: System Address Map	18
Table 4: Files required for booting Linux	22

1. Introduction

The Amber processor core is an ARM-compatible 32-bit RISC processor. The Amber core is fully compatible with the ARM® v2a instruction set architecture (ISA) and is therefore supported by the GNU toolset. The Amber project is a complete embedded system implemented on the Xilinx Spartan-6 SP605 FPGA development board. The project is hosted on opencores.org. The project provides a complete hardware and software development system based around the Amber processor core. Several applications, with C source code, are provided as examples of what the system can be used for. The embedded system includes the Amber core and several peripherals, including a UART, a timer and an Ethernet MAC.

There are two versions of the core provided in the Amber project. The Amber 23 has a 3-stage pipeline, a unified instruction & data cache, a 32-bit Wishbone interface, and is capable of 0.75 DMIPS per MHz. The Amber 25 has a 5-stage pipeline, separate data and instruction caches, a 128-bit Wishbone interface, and is capable of 1.05 DMIPS per Mhz. Both cores implement the same ISA and are 100% software compatible. The cores do not contain a memory management unit (MMU) so they can only run the non-virtual memory variant of Linux. The cores have been verified by booting a 2.4 Linux kernel.

The cores were developed in Verilog 2001, and are optimized for FPGA synthesis. For example, there is no reset logic, all registers are reset as part of FPGA initialization. The complete system has been tested extensively on the Xilinx SP605 Spartan-6 FPGA board. The full Amber system with the A23 core uses 32% of the Spartan-6 XC6SLX45T-3 FPGA Look Up Tables (LUTs), with the core itself occupying less than 20% of the device using the default configuration, and running at 40MHz. It has also been synthesized to a Virtex-6 device at 80MHz, but not yet tested on a real Virtex-6 device. The maximum frequency is limited by the execution stage of the pipeline which includes a 32-bit barrel shifter, 32-bit ALU and address incrementing logic.

The goal of this project is to port the design to Cyclone V FPGA, and interface it to Avalon bus to establish communication with ARM Hard Processing System. The idea is to be able to have a microcontroller on the FPGA with real-time response characteristics, but running the same tool chain as the ARM processors which are running Linux. However, given the struggles with development tools, the project goals were scaled down to getting a more general familiarity with the Amber system, GNU tools and Quartus development tools. This project made an attempt to port Amber system on DE2-115 board Cyclone IV FPGA board. This was a reasonable milestone since it would allow development towards the original goal with continued effort.

2. FPGA

The FPGA board used for the course of this project was DE1-SoC, which is equipped with Altera Cyclone® V SE 5CSEMA5F31C6N device. The FPGA is part of the Cyclone V SoC family from the Altera family. The photograph for DE1-SoC board is shown in Figure 1. The DE1-SoC board has many features that allow users to implement a wide range of designed circuits, from simple circuits to various multimedia projects.

The FPGA consists of a Hard Processor System (HPS) composed of Dual-core ARM Cortex-A9 MPCore processor. 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 has 1GB of DDR3 SDRAM connected to HPS on board. The FPGA fabric is equipped with 85K programmable logic elements, 4450 Kbits of embedded memory, 6 fractional PLL's, and 2 hard memory controllers. All the connections on the DE1-SoC board is shown in Figure 2. The figure shows which hardware components are connected to the ARM HPS and the FPGA fabric.

Cyclone V FPGA consists of large logic elements and hence is a good candidate to port a processor system. Also, the processor in HPS belong to ARM family (ARM® v7a), therefore running the same toolchain as that of the Amber core (ARM® v2a).

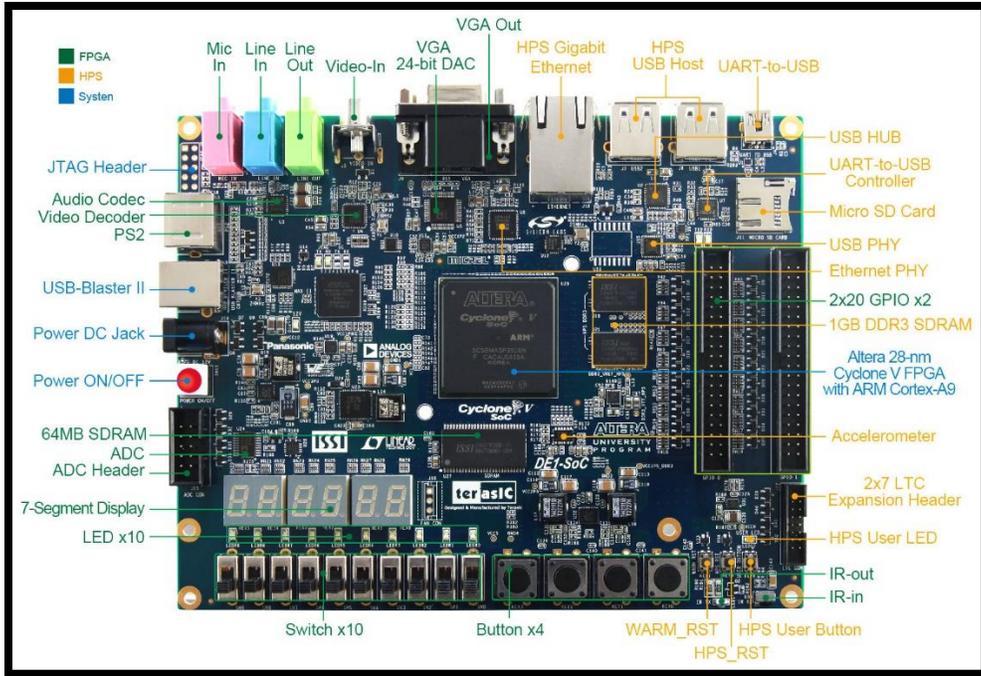


Figure 1: DE1-SoC development board

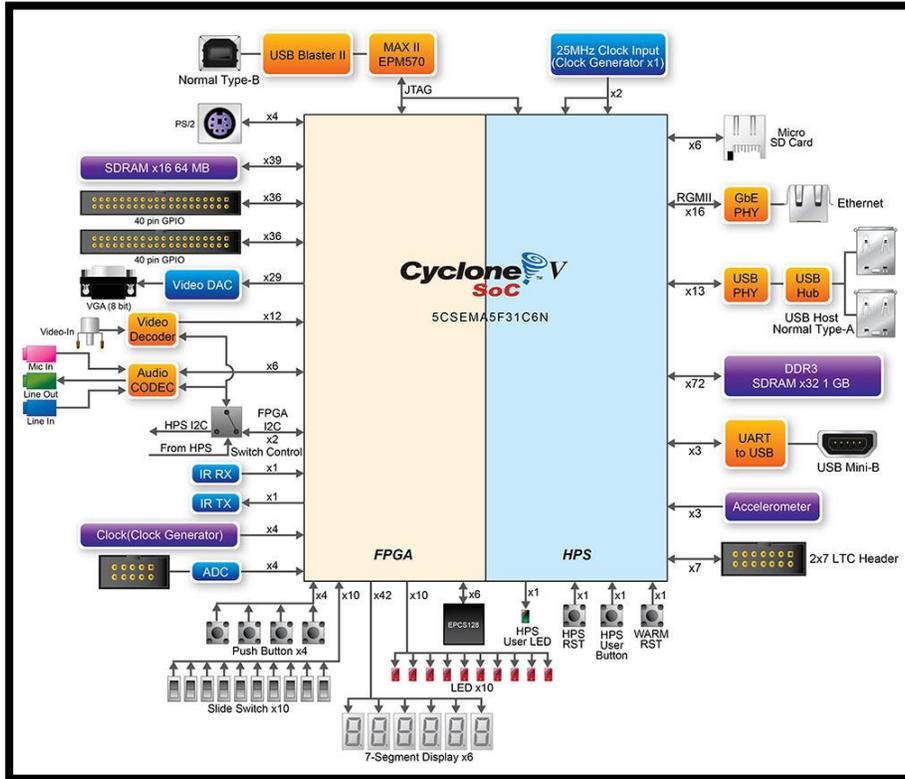


Figure 2: Block diagram of DE1-SoC

3. Amber Core

There are two versions of the core provided in the Amber project. The project focuses on implementation of Amber 23 system. Amber 23 has following features:

- 3-stage pipeline.
- 32-bit Wishbone system bus.
- Unified instruction and data cache, with write through and a read-miss replacement policy. The cache can have 2, 3, 4 or 8 ways and each way is 4kB.
- Multiply and multiply-accumulate operations with 32-bit inputs and 32-bit output in 34 clock cycles using the Booth algorithm. This is a small and slow multiplier implementation.
- Little endian only, i.e. Byte 0 is stored in bits 7:0 and byte 3 in bits 31:24.

The following diagram shows the data flow through the 3-stage core.

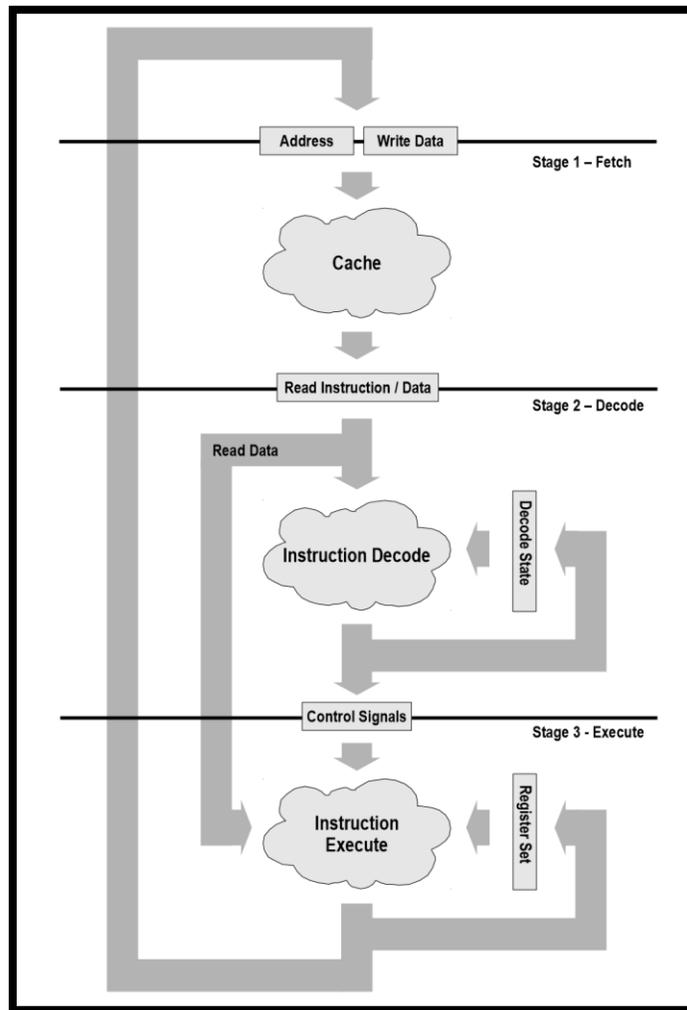


Figure 3: Amber 23 Core pipeline stages

3.1 Amber 23 Pipeline Architecture

The Amber 2 core has a 3-stage pipeline architecture. The best way to think of the pipeline structure is of a circle. There is no start-point or end-point. The output from each stage is registered and fed into the next stage. The three stages are;

- **Fetch** – The cache tag and data RAMs receive an unregistered version of the address output by the execution stage. The registered version of the address is compared to the tag RAM outputs one cycle later to decide if the cache hits or misses. If the cache misses, then the pipeline is stalled while the instruction is fetched from either boot memory or main memory via the Wishbone bus. The cache always does 4-word reads so a complete cache line gets filled. In the case of a cache hit, the output from the cache data RAM goes to the decode stage. This can either be an instruction or data word.
- **Decode** - The instruction is received from the fetch stage and registered. One cycle later it is decoded and the datapath/control signals prepared for the next cycle. This stage contains a state machine that handles multi-cycle instructions and interrupts.
- **Execute** – The control signals from the decode stage are registered and passed into the execute stage, along with any read data from the fetch stage. The operands are read from the register bank, shifted, combined in the ALU and the result written back. The next address for the fetch stage is generated.

3.2 Registers

Table 1: Register Sets

User (USR)	Supervisor (SVC)	Interrupt (IRQ)	Fast Interrupt (FIRQ)
		r0	
		r1	
		r2	
		r3	
		r4	
		r5	
		r6	
		r6	
		r7	
	r8		r8_firq
	r9		r9_firq
	r10		r10_firq
	r11 (fp)		r11_firq
	r12 (ip)		r12_firq
r13 (sp)	r13_svc	r13_irq	r13_firq
r14 (lp)	r14_svc	r14_irq	r14_firq
		r15 (pc)	

Table 2: Status Bits - part of the PC

Field	Position	Type	Description
flags	[31:28]	User Writable	{ Negative, Zero, Carry, overflow }
I	27	Privileged	IRQ mask, disables IRQs when high
F	26	Privileged	FIRQ Mask, disables FIRQs when high
mode	[1:0]	Privileged	Processor mode 3 - Supervisor 2 - Interrupt 1 - Fast Interrupt 0 - User

3.3 Cache

The Amber cache size is optimized to use FPGA Block RAMs. Each way has 256 lines of 16 bytes. 256 lines x 16 bytes x 2 ways = 8k bytes. The address tag is 20 bits. Each cache can be configured with either 2, 3, 4 or 8 ways.

More details about Amber core can be obtained from the Amber 2 core specification. Figure 4 shows how the amber core is divided into modules located in different files.

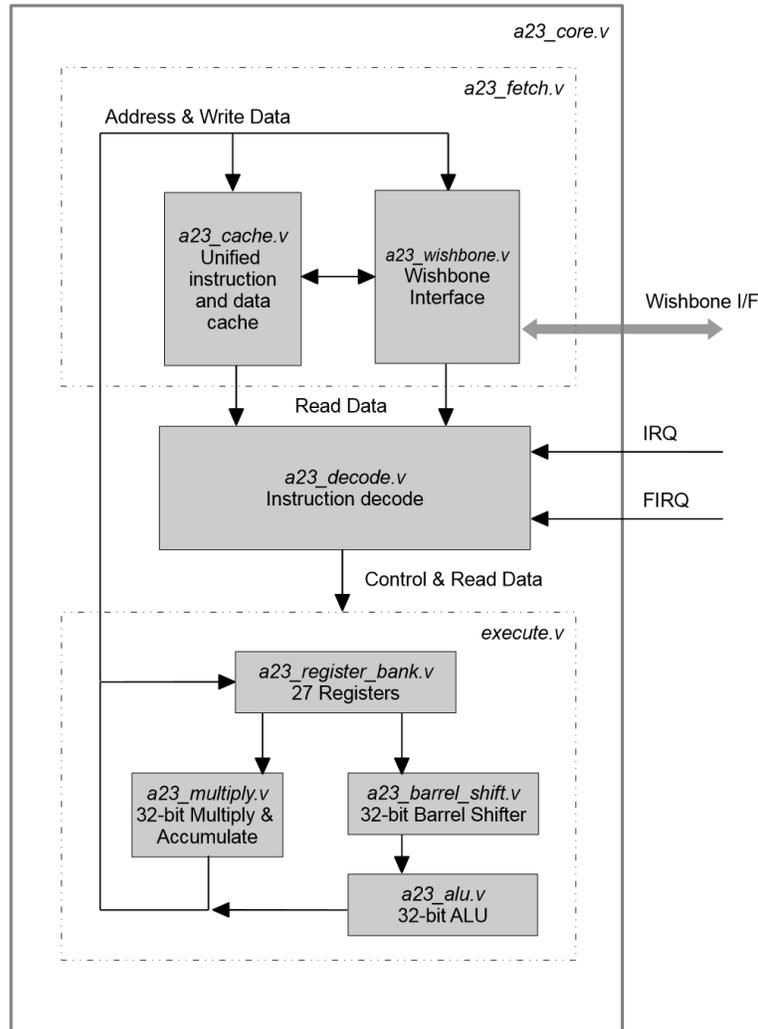


Figure 4: Amber 23 Core code Structure

4. Amber FPGA system

The FPGA system included with the Amber project is a complete embedded processor system which included all peripherals needed to run Linux, including UART, timers and an Ethernet (MII) port. The following diagram shows the entire system.

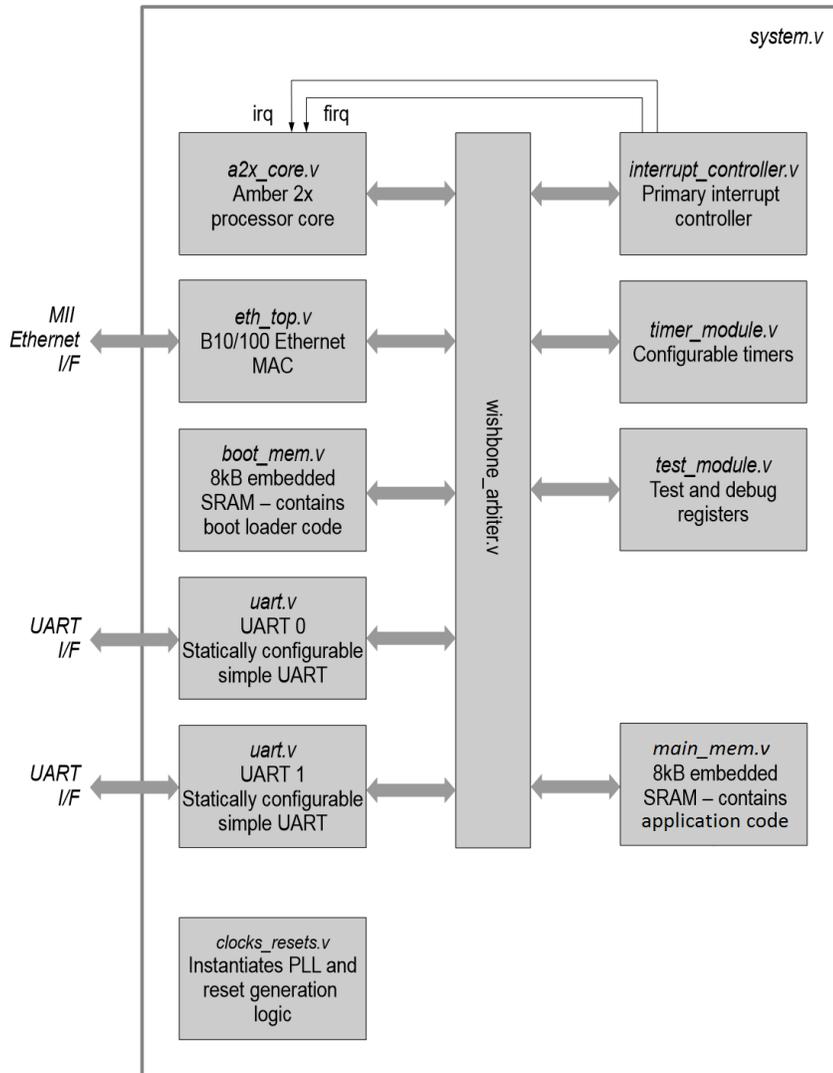


Figure 5: Amber FPGA system

Since the original project was developed for Xilinx FPGA, there were some mandatory changes required to port it to Altera FPGA. Below sub-sections walk through the necessary changes made to port it to DE1-SoC board.

4.1 Clocks and Reset

The Amber FPGA system has a clock and reset module to take care of clocking and reset of the entire system. The module takes in a 50MHz single ended clock from board, passes it to Altera PLL module that converts it to 33.33MHz clock. Pushbutton key is used to reset the entire system. To make the system more robust, the clock and reset module takes in the reset from board, passes it to DDR controller as well as PLL. Then using PLL lock and calibration done signal, system ready signal is generated. There is also a synchronous reset generation module to synchronize external asynchronous reset.

Figure 6, 7 and 8 illustrates how the ALTPLL is configured to generate the required clocking.

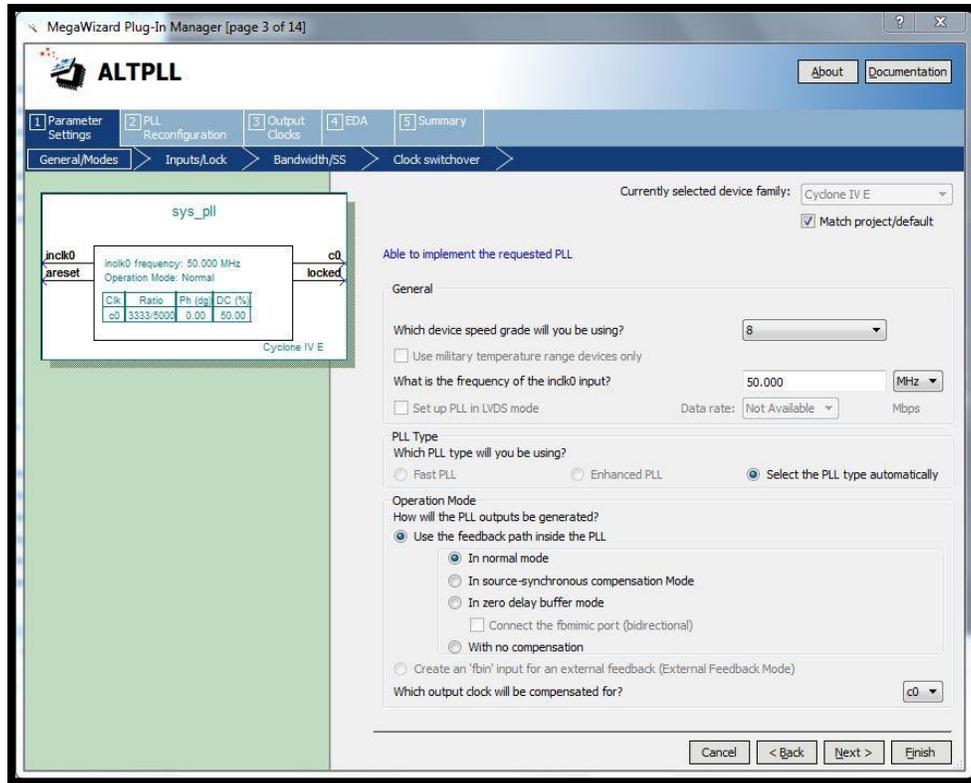


Figure 6: ALTPLL input clock settings

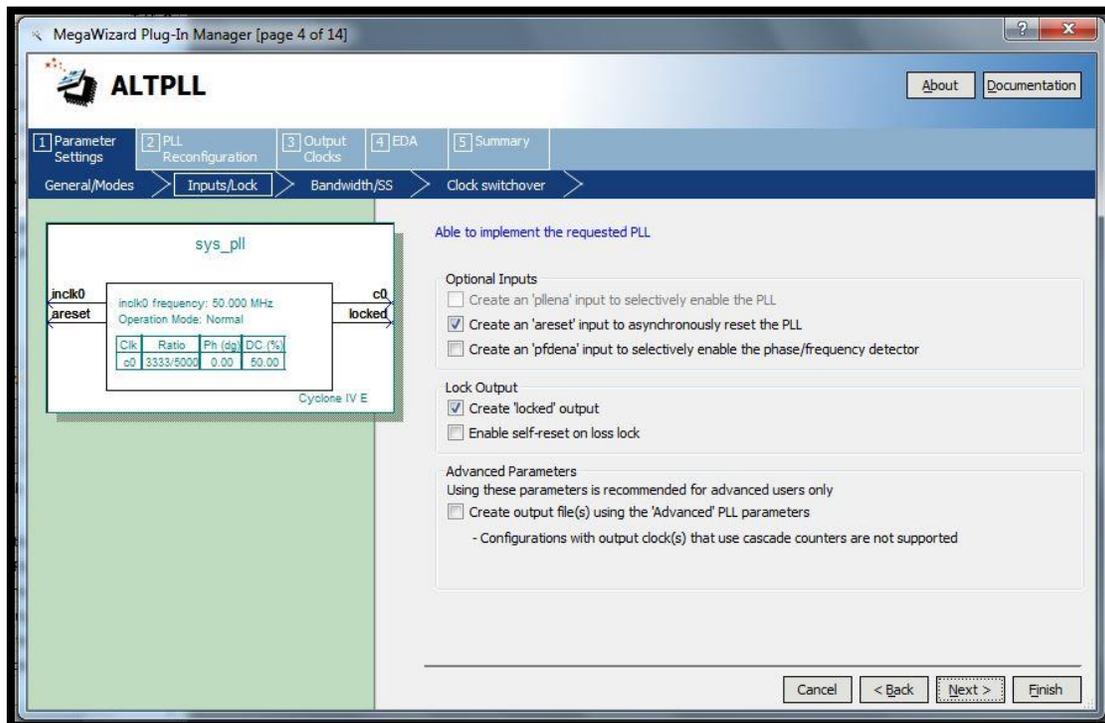


Figure 7: ALTPLL control signals

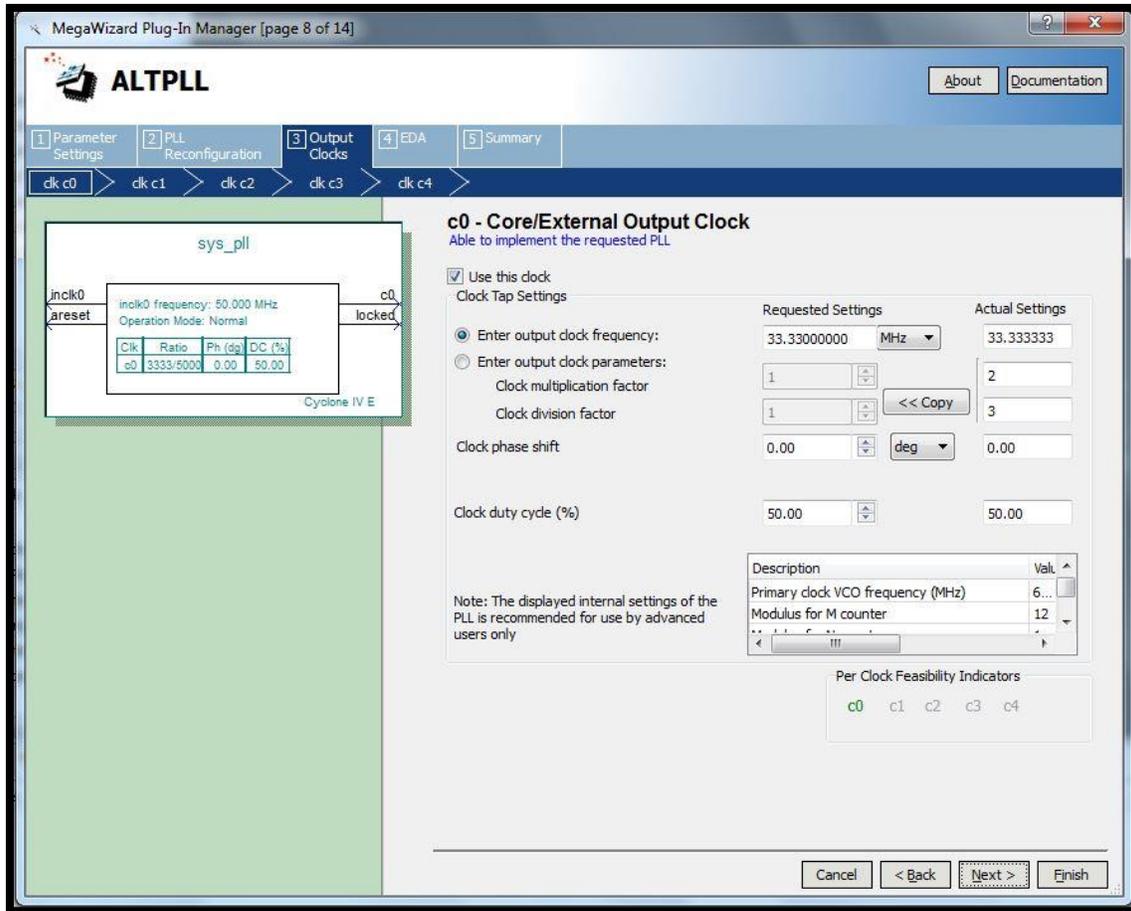


Figure 8: ALTPLL output clock settings

4.2 BOOT Memory

The Amber FPGA system has a separate memory (different from main memory) that contains the boot code. Boot code is the sequence of code that is executed once the processor is out of reset. Next section will cover the configurations done by bootloader code.

Boot memory is 32 bits wide, and has a depth of 4096 making it 16kBytes. This memory is inferred using 16 M9K blocks. Figure 9, 10 and 11 shows how to configure the RAM 1-Port to generate 16kBytes boot memory. Also, note that the boot memory needs to be initialized with boot code. Altera FPGA supports only “.mif” or “.hex” file for memory initialization. Next section discusses how to convert “.elf” to “.mif” file. While configuring RAM 1-Port, a memory initialization file can be provided as shown in Figure 11.

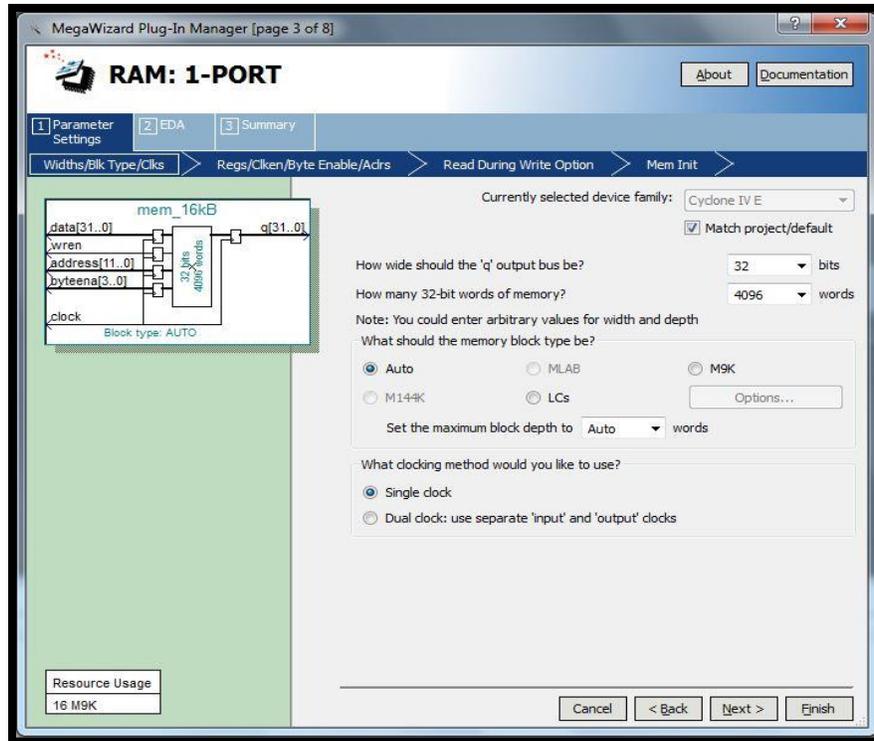


Figure 9: Boot Memory configuration wizard

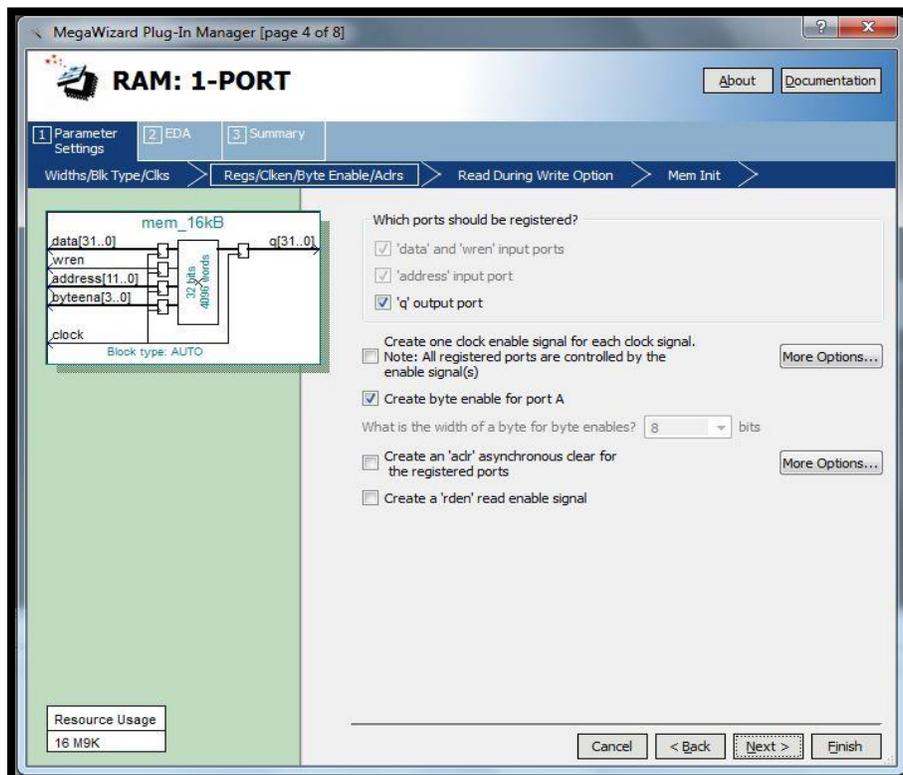


Figure 10: Boot Memory port configuration

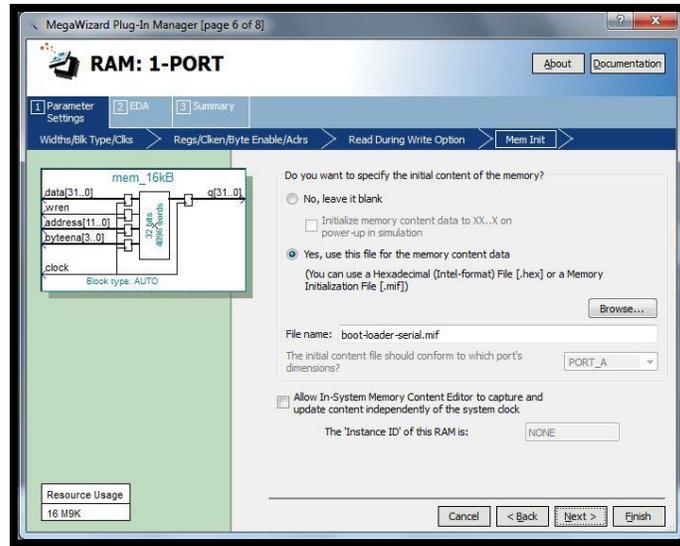


Figure 11: Boot Memory Initialization

4.3 Main Memory

In original Amber FPGA system, external DDR3 memory is used as main memory. That system consists of Wishbone to DDR controller bridge, Xilinx DDR3 controller and external DDR3 on board. For simplicity, the new Amber FPGA system uses on-chip M9K RAM as main memory. Due to limitation on number of M9K RAMs, main memory of 16kByte is appropriate.

The steps to configure RAM 1-Port for main memory is same as that of Boot Memory. Only difference is that instead of giving bootloader initialization file, either the main memory can be kept uninitialized or application initialization file can be used. This is illustrated in Figure 12.

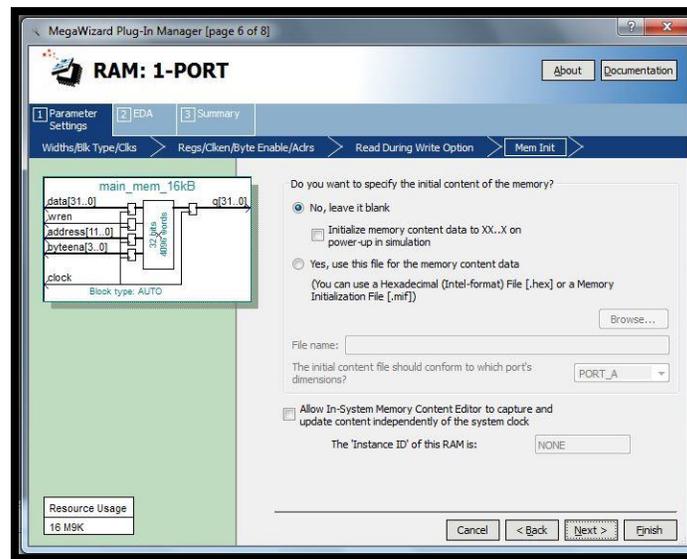


Figure 12: Main Memory Initialization

4.4 Unified Instruction and Data Cache

Amber 2 core has a unified instruction and data cache. The cache has 2 ways and each way has 256 lines of 16 bytes. 256 lines x 16 bytes x 2 ways = 8k bytes. The address tag is 20 bits.

To implement cache, two important data structures are required – tag array and data array. Since tag is 20 bits plus 1 bit for valid, tag array is 256x21. Since cache line size is 16 bytes or 128 bits, data array is 256x128.

Both Tag array and data array is implemented by using RAM 1-Port Megawizard IP. Since write and read to data array could be byte-wise instead of word-wise, hence while configuring the IP byte enable port needs to be selected.

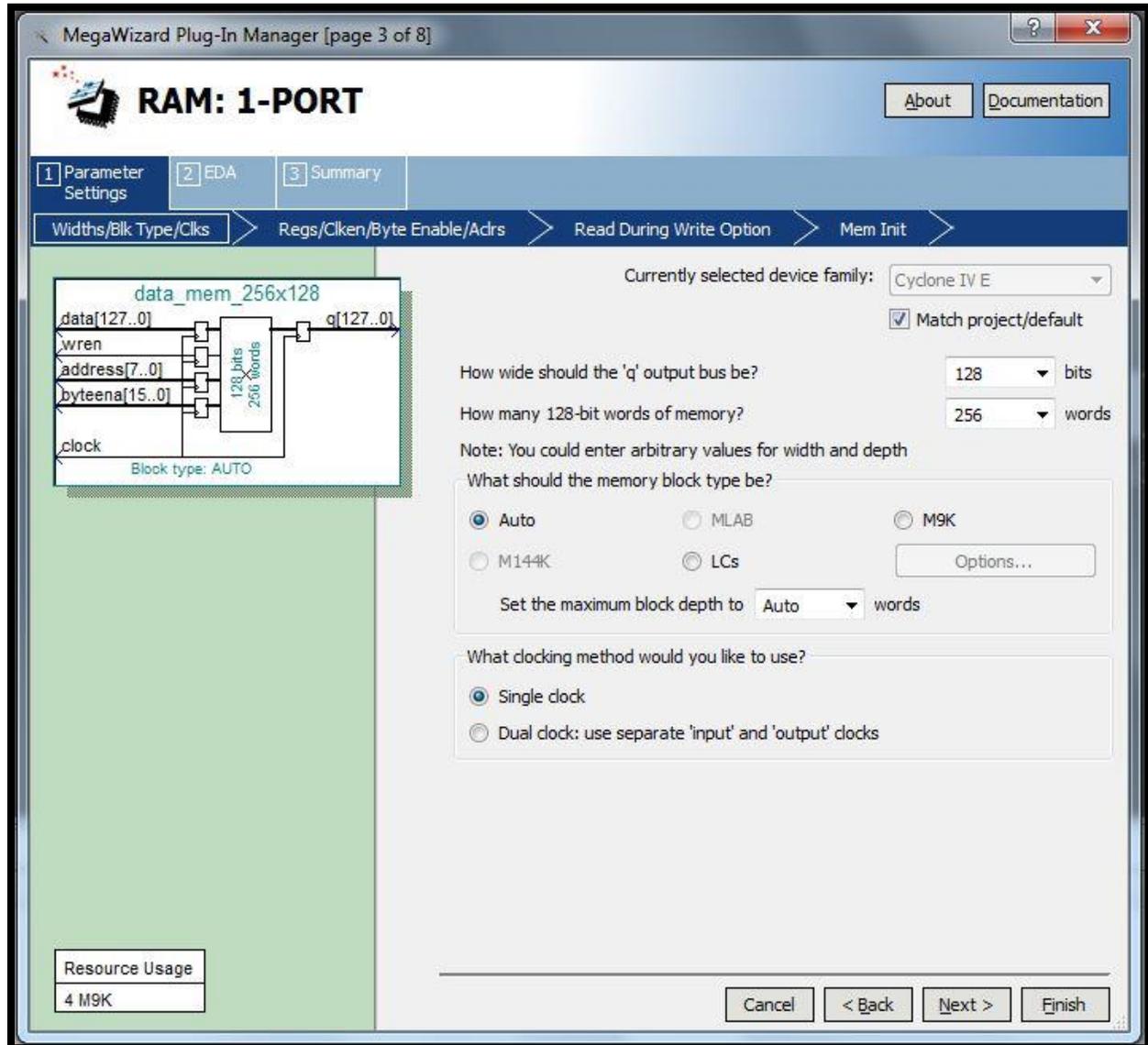


Figure 13: Data Array size configuration

Figure 13, 14 and 15 shows IP configuration to enable data array of the cache. Note that it requires 4 M9K memory blocks.

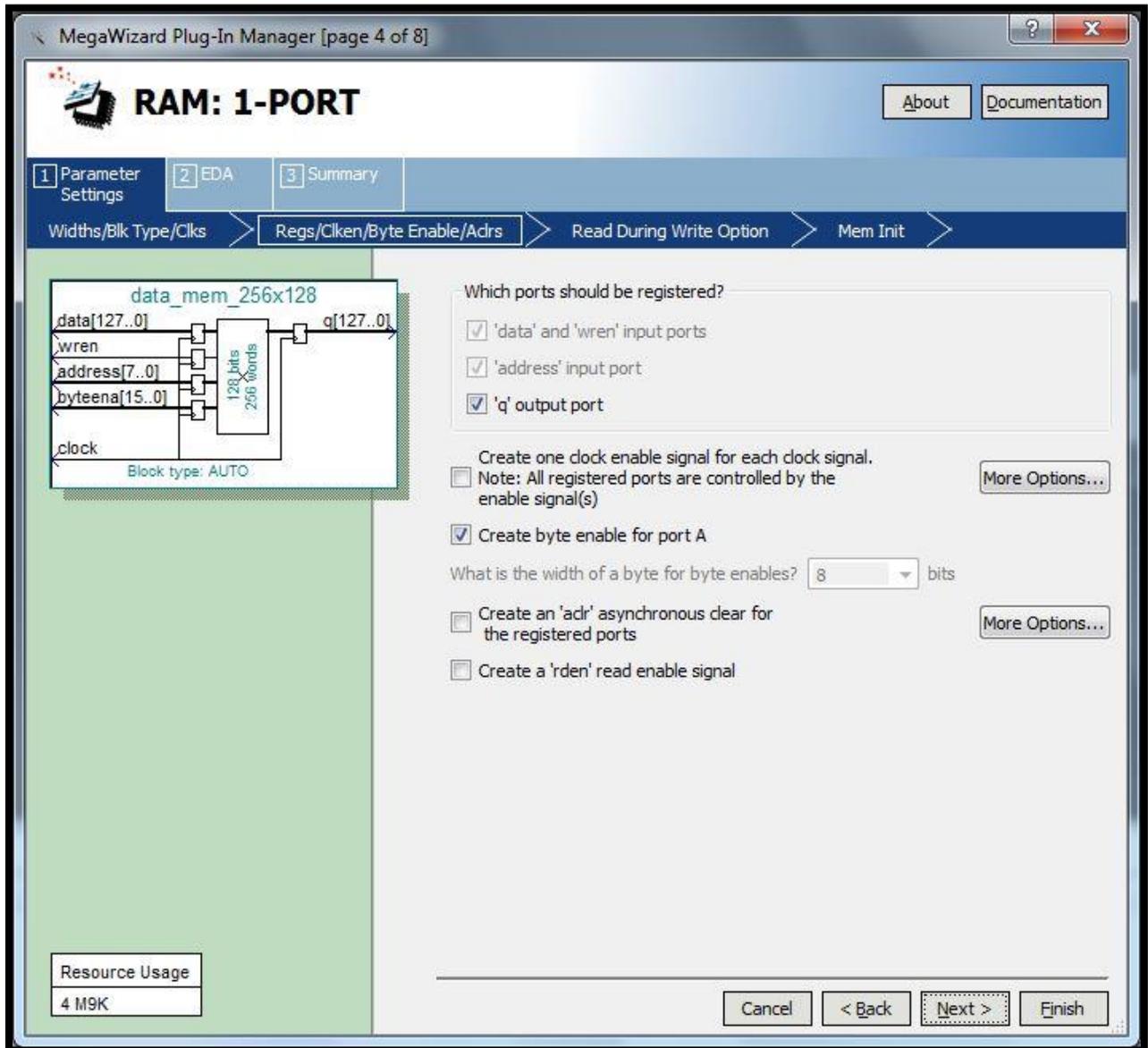


Figure 14: Data Array port configuration

Since a valid entry in cache is determined by valid bit in tag array, it is not necessary to initialize data array with all 0s.

Figure 16, 17 and 18 shows RAM 1-Port IP configuration to enable tag array for unified cache. As shown in Figure 16, tag array of size 256x21 is chosen. Also since tag array is always accessed word-wise hence byte enable is not required for tag array. Note that tag array requires only 1 M9K memory block. Since it is necessary that cache is clean before use, tag array is initialized to 0. This is shown in Figure 18. “tag_init.hex” file contains memory initialization file (all 0s) for tag array.

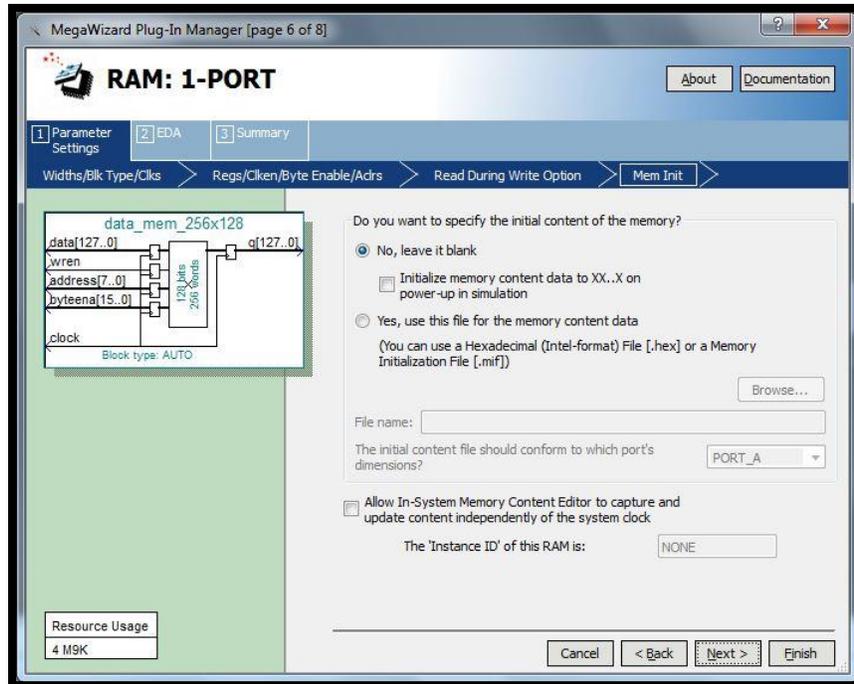


Figure 15: Data Array Initialization

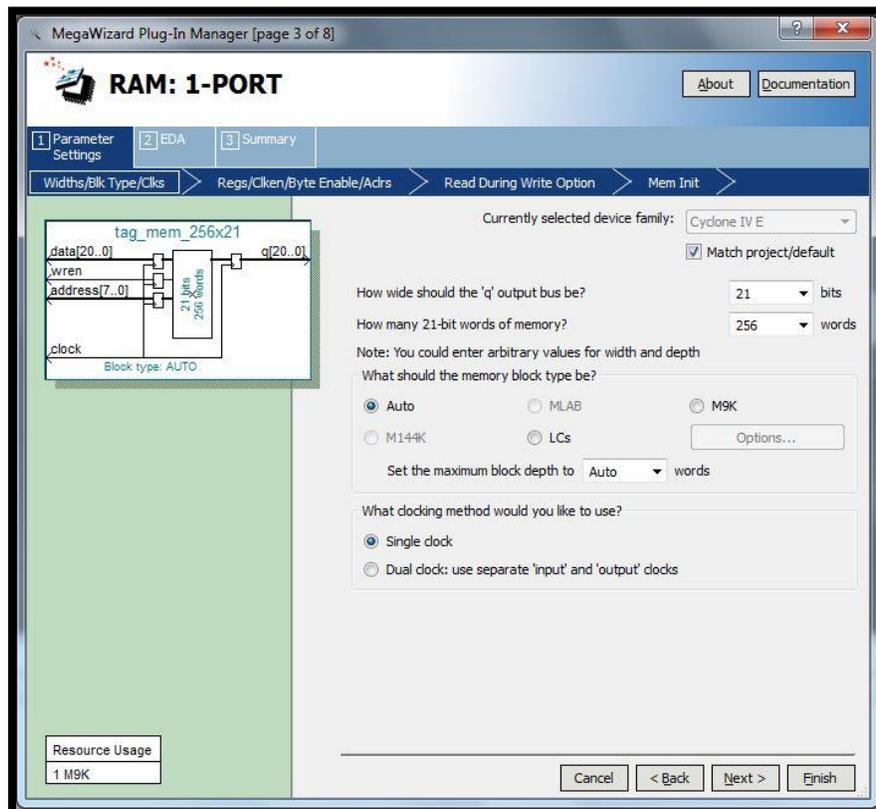


Figure 16: Tag Array size configuration

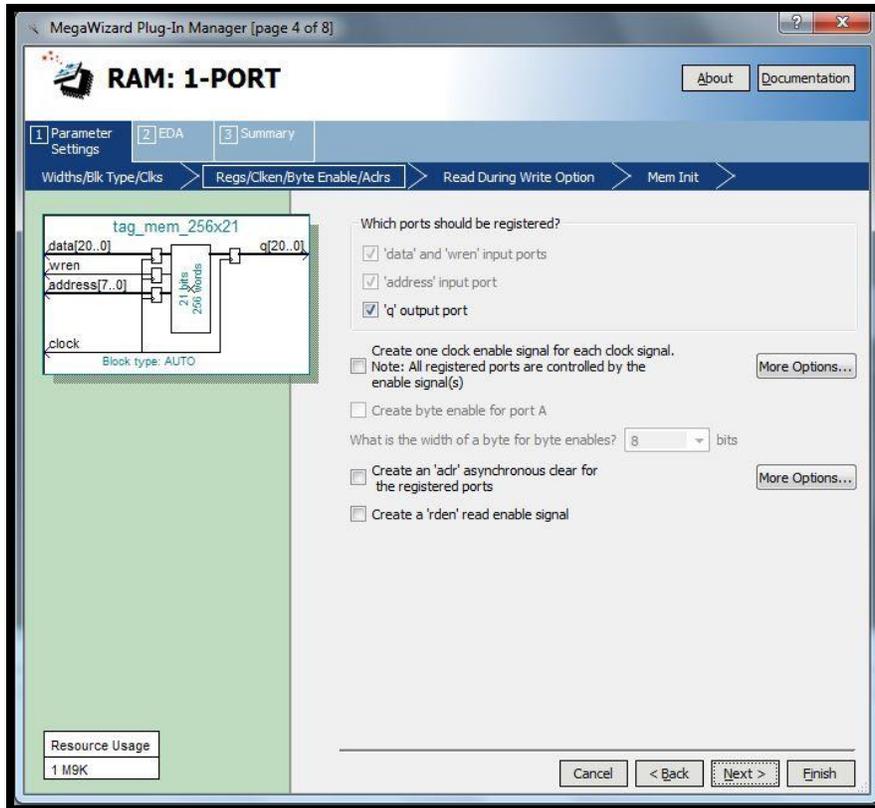


Figure 17: Tag Array port configuration

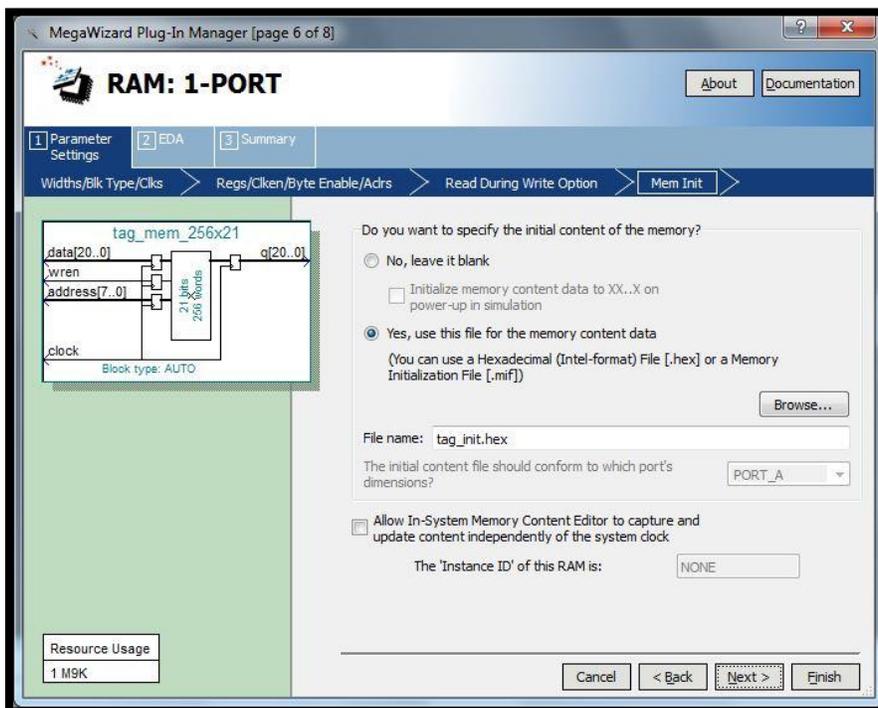


Figure 18: Tag Array Initialization

4.5 Debug

The Amber FPGA system has multiple blocks. Hence, it becomes difficult to debug in case of failure. To ease system level debug, few extra ports are added.

1. System Ready: This port is connected to LED on board. The LED shall glow indicating that the system is out of reset, DDR (if present) calibration is done and Ethernet PHY (if present) configuration is done.

```
// Halt core until system is ready
assign system_rdy = phy_init_done && !sys_rst;
```

2. PLL Lock: This port is also connected to LED on board. The LED shall glow indicating that ALTPLL is locked and that 33.33 MHz system clock is ready.

4.6 UART

The Amber FPGA system also has UART for external communication. By default, the baud rate of UART is set to 9600 Hz. Since DE1-SoC board doesn't have a UART port on fabric side, so I decided to use DE2-115 Cyclone IV board for porting Amber system on FPGA.

4.7 Pin Mapping

Figure 19 shows pin mapping of Amber FPGA system for DE2-115 board.

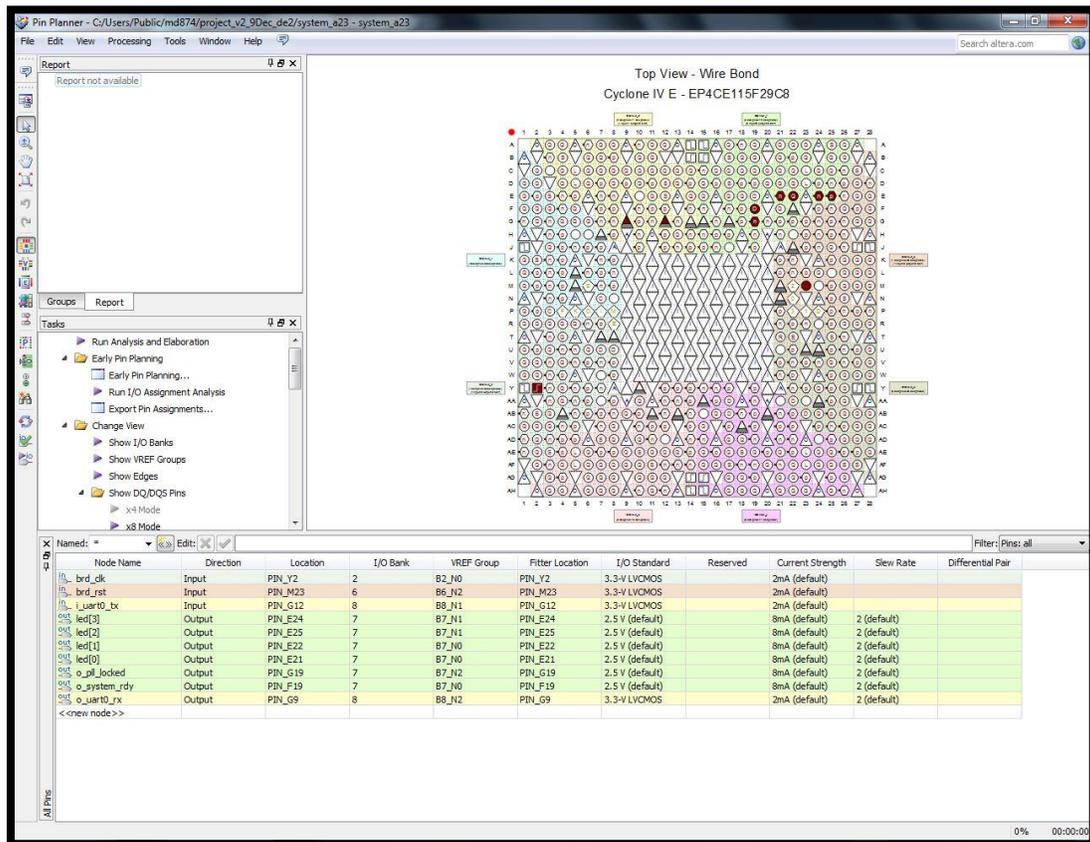


Figure 19: Pin Mapping of Amber FPGA System on DE2-115 board

5. System Addresses

5.1 Address Map

The comprehensive system level address map is shown in Table 3. Note that Amber system on Xilinx FPGA has an external DDR memory and hence has 128Mbytes of address space for main memory. The Amber system ported on Altera FPGA in this project doesn't have external main memory but on-chip M9K RAM. Hence, the address space for main memory is reduced to 16kBytes.

Table 3: System Address Map

Peripherals	Address Space	Size
Boot Memory (cached)	0x0000_0000 – 0x0000_3FFF	16kBytes
Boot Memory (Uncached)	0x2800_0000 – 0x2800_3FFF	16kBytes
Main Memory	0x0000_0000 – 0x07FF_FFFF	128MBytes
UART0	0x1600_0000 – 0x1600_0FFF	4kBytes
UART1	0x1700_0000 – 0x1700_0FFF	4kBytes
Timer	0x1300_0000 – 0x1300_0FFF	4kBytes
Interrupt Controller	0x1400_0000 – 0x01400_00FF	256Bytes

6. Software Support

Till this point, all discussion Amber FPGA system was made from hardware point of view. This section discusses all the software tools and support necessary to run bootloader, standalone application or Linux.

6.1 Installing the compiler

Tests need to be compiled before running on board. To do this, it is necessary to install a GNU cross-compiler. The easiest way to install the GNU tool chain is to download a readymade package. Code Sourcery provides a free one. To download the Code Sourcery package, go to this page <http://www.codesourcery.com/sgpp/lite/arm>

Select the GNU/Linux version and then the IA32 GNU/Linux Installer. Once the package is installed, add the following to `<.bashrc>` file, where the PATH is set to where you install the Code Sourcery GNU package.

```
# Change /opt/Sourcery to where the package is installed on your system
PATH=/<your code sourcery install path>/bin:${PATH}

# AMBER_CROSSTOOL is the name added to the start of each GNU tool in
# the Code Sourcery bin directory. This variable is used in various makefiles to set
# the correct tool to compile code for the Amber core
export AMBER_CROSSTOOL=arm-none-linux-gnueabi
```

6.2 GNU Tools usage

It's important to remember to use the correct switches with the GNU tools to restrict the ISA to the set of instructions supported by the Amber 2 core. The switches are already set in the makefiles included with the Amber 2 core. Here are the switches to use with gcc (arm-none-linux-gnueabi-gcc).

```
-march=armv2a -mno-thumb-interwork
```

These switches specify the correct version of the ISA, and tell the compiler not to create bx instructions. Here is the switch to use with the GNU linker, arm-nonelinux-gnueabi-ld;

```
--fix-v4bx
```

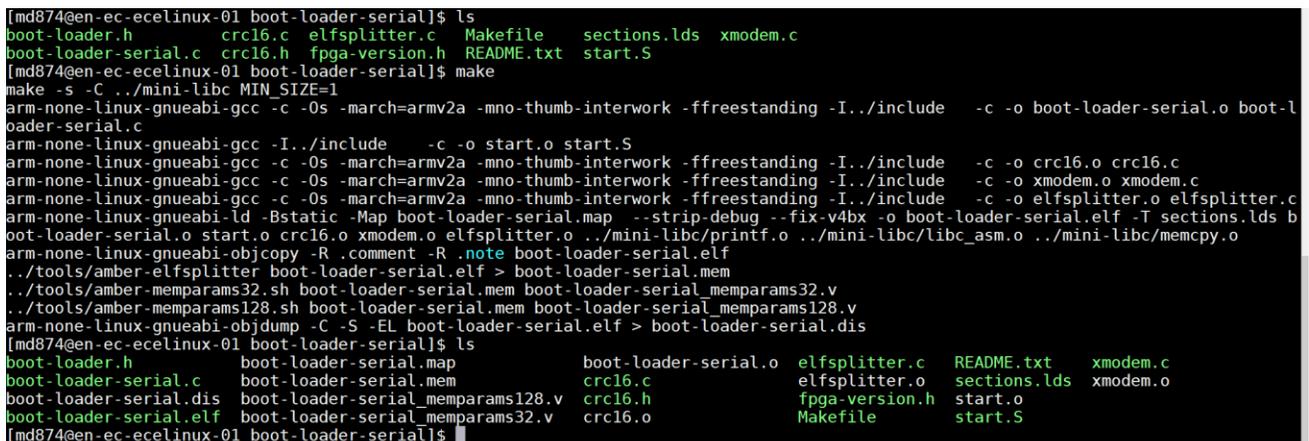
This switch converts any bx instructions (which are not supported) to 'mov pc, lr'. Here is an example usage from the boot-loader make process;

```
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a -mno-thumb-interwork -ffreestanding -I./include -c -o boot-loader.o boot-loader.c
arm-none-linux-gnueabi-gcc -I./include -c -o start.o start.S
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a -mno-thumb-interwork -ffreestanding -I./include -c -o crc16.o crc16.c
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a -mno-thumb-interwork -ffreestanding -I./include -c -o xmodem.o xmodem.c
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a -mno-thumb-interwork -ffreestanding -I./include -c -o elfsplitter.o elfsplitter.c
arm-none-linux-gnueabi-ld -Bstatic -Map boot-loader.map --strip-debug --fix-v4bx -o boot-loader.elf -T sections.lds boot-loader.o start.o crc16.o xmodem.o elfsplitter.o ../mini-libc/printf.o ../mini-libc/libc_asm.o ../mini-libc/memcpy.o
arm-none-linux-gnueabi-objcopy -R .comment -R .note boot-loader.elf
../tools/amber-elfsplitter boot-loader.elf > boot-loader.mem
../tools/amber-memparams.sh boot-loader.mem boot-loader_memparams.v
arm-none-linux-gnueabi-objdump -C -S -EL boot-loader.elf > boot-loader.dis
```

6.3 Bootloader

The boot loader is used to download longer applications onto the FPGA development board via the UART port and using Putty on a host Windows PC. As shown in Figure 21, bootloader gives many options.

- Load an executable file
- Load binary file to a particular address
- Execute an already loaded application by jumping to that address
- Read/Write to main memory
- View core status



```
[md874@en-ec-ecelinux-01 boot-loader-serial]$ ls
boot-loader.h      crc16.c  elfsplitter.c  Makefile  sections.lds  xmodem.c
boot-loader-serial.c  crc16.h  fpga-version.h  README.txt  start.S
[md874@en-ec-ecelinux-01 boot-loader-serial]$ make
make -s -C ../mini-libc MIN_SIZE=1
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a -mno-thumb-interwork -ffreestanding -I./include -c -o boot-loader-serial.o boot-loader-serial.c
arm-none-linux-gnueabi-gcc -I./include -c -o start.o start.S
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a -mno-thumb-interwork -ffreestanding -I./include -c -o crc16.o crc16.c
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a -mno-thumb-interwork -ffreestanding -I./include -c -o xmodem.o xmodem.c
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a -mno-thumb-interwork -ffreestanding -I./include -c -o elfsplitter.o elfsplitter.c
arm-none-linux-gnueabi-ld -Bstatic -Map boot-loader-serial.map --strip-debug --fix-v4bx -o boot-loader-serial.elf -T sections.lds boot-loader-serial.o start.o crc16.o xmodem.o elfsplitter.o ../mini-libc/printf.o ../mini-libc/libc_asm.o ../mini-libc/memcpy.o
arm-none-linux-gnueabi-objcopy -R .comment -R .note boot-loader-serial.elf
../tools/amber-elfsplitter boot-loader-serial.elf > boot-loader-serial.mem
../tools/amber-memparams32.sh boot-loader-serial.mem boot-loader-serial_memparams32.v
../tools/amber-memparams128.sh boot-loader-serial.mem boot-loader-serial_memparams128.v
arm-none-linux-gnueabi-objdump -C -S -EL boot-loader-serial.elf > boot-loader-serial.dis
[md874@en-ec-ecelinux-01 boot-loader-serial]$ ls
boot-loader.h      boot-loader-serial.map      boot-loader-serial.o  elfsplitter.c  README.txt  xmodem.c
boot-loader-serial.c  boot-loader-serial.mem      crc16.c               elfsplitter.o  sections.lds  xmodem.o
boot-loader-serial.dis  boot-loader-serial_memparams128.v  crc16.h               fpga-version.h  start.o
boot-loader-serial.elf  boot-loader-serial_memparams32.v  crc16.o               Makefile        start.S
[md874@en-ec-ecelinux-01 boot-loader-serial]$
```

Figure 20: Steps to compile boot-loader serial

Figure 20 shows the steps to compile bootloader. The Makefile has necessary steps to generate executable file as well as .mem file for Xilinx FPGA.

```
void print_help ( void )
{
    printf("Commands\n");
    printf("l");
    print_spaces(29);
    printf(": Load elf file\n");
    printf("b <address>");
    print_spaces(19);
    printf(": Load binary file to <address>\n");
    printf("d <start address> <num bytes> : Dump mem\n");
    printf("h");
    print_spaces(29);
    printf(": Print help message\n");
    printf("j <address>");
    print_spaces(19);
    printf(": Execute loaded elf, jumping to <address>\n");
    printf("p <address>");
    print_spaces(19);
    printf(": Print ascii mem until first 0\n");
    printf("r <address>");
    print_spaces(19);
    printf(": Read mem\n");
    printf("s");
    print_spaces(29);
    printf(": Core status\n");
    printf("w <address> <value>");
    print_spaces(11);
    printf(": Write mem\n");
}
```

Figure 21: bootloader options

Since Altera FPGA support memory initialization in .mif or .hex file format. Since both memory initialization format are readable, with slight difference in address and data format. Figure 22 shows equivalent .mif file for the .mem file generated for bootloader code.

Memory Initialization file is an ASCII text file (with the extension **.mif**) that specifies the initial content of a memory block (CAM, RAM, or ROM), that is, the initial values for each address. This file is used during project compilation and/or simulation. A Memory Initialization File serves as an input file for memory initialization in the Compiler and Simulator. A Hexadecimal (Intel-Format) File (**.hex**) to provide memory initialization data.

A Memory Initialization File contains the initial values for each address in the memory. A separate file is required for each memory block. In a Memory Initialization File, the memory depth and width values must be specified. In addition, data radixes can be specified as binary (BIN), hexadecimal (HEX), octal (OCT), signed decimal (DEC), or

unsigned decimal (UNS) to display and interpret addresses and data values. Data values must match the specified data radix.

Tab "\t" and Space " " characters are used as separators, and multiple lines of comments can be inserted with the percent "%" character, or a single comment with double dash "--" characters. Address : data pairs represent data contained inside certain memory addresses and must place them between the CONTENT BEGIN and END keywords, as shown in the following examples.

```
% multiple-line comment
multiple-line comment %
-- single-line comment

DEPTH = 8;                -- The size of memory in words
WIDTH = 8;                -- The size of data in bits
ADDRESS_RADIX = HEX;     -- The radix for address values
DATA_RADIX = BIN;        -- The radix for data values
CONTENT                   -- start of (address : data pairs)
BEGIN

00 : 00000000;           -- memory address : data
01 : 00000001;
02 : 00000010;
03 : 00000011;
04 : 00000100;
05 : 00000101;
06 : 00000110;
07 : 00000111;
END;
```

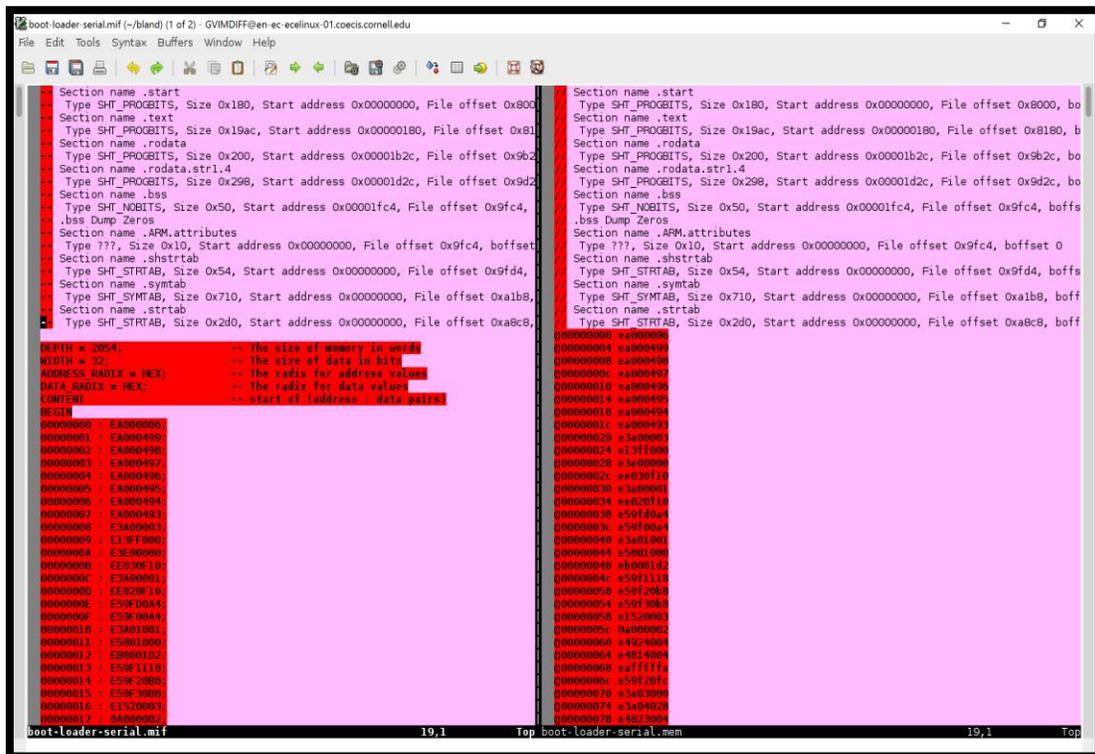


Figure 22: .mem file to .mif file conversion

6.4 Standalone Application

Once bootloader is run, then the processor would do the initialization and jump to main memory depending on the option selected. Then any standalone application can be run. Figure 23 shows a hello-world application.

```
/* Note that the stdio.h referred to here is the one in
   mini-libc. This applications compiles in mini-libc
   so it can run stand-alone.
*/
#include "stdio.h"

main ()
{
    printf ("Hello, world!\n");
    /* Flush out UART FIFO */
    printf (" ");
    _testpass();
}
```

Figure 23: Hello-world application code

Figure 24 below shows the steps to compile standalone application. Note that .elf and .mem file are generated and must be converted in .hex or .mif file.

```
[md874@en-ec-ecelinux-01 hello-world]$ ls
hello-world.c Makefile sections.lds start.S
[md874@en-ec-ecelinux-01 hello-world]$ make
make -s -C ../mini-libc MIN_SIZE=1
arm-none-linux-gnueabi-gcc -c -O3 -march=armv2a -mno-thumb-interwork -ffreestanding -I../include -c -o hello-world.o hello-world.c
arm-none-linux-gnueabi-gcc -I../include -c -o start.o start.S
arm-none-linux-gnueabi-ld -Bstatic -Map hello-world.map --strip-debug --fix-v4bx -elf2flt=-v -elf2flt=-k -o hello-world.flt -T sections.lds hello-world.o start.o ../mini-libc/printf.o ../mini-libc/libc_asm.o ../mini-libc/memcpy.o
arm-none-linux-gnueabi-ld: warning: cannot find entry symbol lf2flt=-k; defaulting to 00008000
arm-none-linux-gnueabi-ld -Bstatic -Map hello-world.map --strip-debug --fix-v4bx -o hello-world.elf -T sections.lds hello-world.o start.o ../mini-libc/printf.o ../mini-libc/libc_asm.o ../mini-libc/memcpy.o
arm-none-linux-gnueabi-objcopy -R .comment -R .note hello-world.elf
../tools/amber-elfsplitter hello-world.elf > hello-world.mem
../tools/amber-memparams32.sh hello-world.mem hello-world_memparams32.v
../tools/amber-memparams128.sh hello-world.mem hello-world_memparams128.v
arm-none-linux-gnueabi-objdump -C -S -EL hello-world.elf > hello-world.dis
[md874@en-ec-ecelinux-01 hello-world]$ ls
hello-world.c hello-world.elf hello-world.map hello-world_memparams128.v hello-world.o sections.lds start.S
hello-world.dis hello-world.flt hello-world.mem hello-world_memparams32.v Makefile start.o
[md874@en-ec-ecelinux-01 hello-world]$
```

Figure 24: Steps to compile hello-world application

6.5 Linux

List of necessary files required for booting Linux on Amber FPGA system is shown in Table 4 below: -

Table 4: Files required for booting Linux

File	Description
initrd	A disk image needed to build the Amber Linux kernel from sources
patch-2.4.27-amber2.bz2	Amber Linux patch file
patch-2.4.27-vrs1.bz2	ARM Linux patch file
vmlinux	Kernel executable file
vmlinux.dis.bz2	Kernel disassembly file, bzip2 compressed
vmlinux.mem.bz2	Kernel <.mem file> for Verilog simulations, bzip2 compressed

There are different steps to be followed to run Amber Linux kernel on a development board. These are listed below:

1. Download the bitfile (.sof) to configure the FPGA using JTAG programmer
 2. Connect Putty to the serial port on the FPGA to get bootloader prints
 3. Download the disk image
- ```
> b 800000
```
4. Then select one of the provided disk image files to transfer, e.g.  
`$AMBER_BASE/sw/vmlinux/initrd-200k-hello-world`
  5. Download the kernel image
- ```
> l
```
6. Then select the file `$AMBER_BASE/sw/vmlinux/vmlinux` to transfer
 7. Execute the kernel
- ```
> j 800000
```

Below are the steps to build Amber Linux kernel from source:

1. Set the location on the system where the Amber project is located
- ```
> export AMBER_BASE=/proj/opencores-svn/trunk
```
2. Pick a directory on the system where to build Linux
- ```
> export LINUX_WORK_DIR=/proj/amber2-linux
```
3. Create the Linux build directory
- ```
> test -e ${LINUX_WORK_DIR} || mkdir ${LINUX_WORK_DIR}
> cd ${LINUX_WORK_DIR}
```
4. Download the kernel source
- ```
> wget http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.27.tar.gz
> tar zxf linux-2.4.27.tar.gz
> ln -s linux-2.4.27 linux
> cd ${LINUX_WORK_DIR}/linux
```
5. Apply 2 patch files
- ```
> cp ${AMBER_BASE}/sw/vmlinux/patch-2.4.27-vrs1.bz2 .
> cp ${AMBER_BASE}/sw/vmlinux/patch-2.4.27-amber2.bz2 .
> bzip2 -d patch-2.4.27-vrs1.bz2
```

```
> bzip2 -d patch-2.4.27-amber2.bz2
> patch -p1 < patch-2.4.27-vrs1
> patch -p1 < patch-2.4.27-amber2

6. Build the kernel

> make dep
> make vmlinux
> cp vmlinux vmlinux_unstripped
> ${AMBER_CROSSTOOL}-objcopy -R .comment -R .note vmlinux
> ${AMBER_CROSSTOOL}-objcopy --change-addresses -0x02000000 vmlinux
```

```
# Amber Boot Loader v20110117211518
# j 0x2080000
#
# Linux version 2.4.27-vrs1 (conor@server) (gcc version 4.5.1 (Sourcery G++ Lite 2010.09-50) ) #354 Tue Feb 1 17:56:00 GMT 2011
# CPU: Amber 2 revision 0
# Machine: Amber-FPGA-System
# On node 0 totalpages: 1024
# zone(0): 1024 pages.
# zone(1): 0 pages.
# zone(2): 0 pages.
# Kernel command line: console=ttyAM0 mem=32M root=/dev/ram
# Calibrating delay loop... 19.91 BogoMIPS
# Memory: 32MB = 32MB total
# Memory: 31136KB available (493K code, 195K data, 32K init)
# Dentry cache hash table entries: 4096 (order: 0, 32768 bytes)
# Inode cache hash table entries: 4096 (order: 0, 32768 bytes)
# Mount cache hash table entries: 4096 (order: 0, 32768 bytes)
# Buffer cache hash table entries: 8192 (order: 0, 32768 bytes)
# Page-cache hash table entries: 8192 (order: 0, 32768 bytes)
# POSIX conformance testing by UNIFIX
# Linux NET4.0 for Linux 2.4
# Based upon Swansea University Computer Society NET3.039
# Starting kswapd
# ttyAM0 at MMIO 0x16000000 (irq = 1) is a WSBN
# pty: 256 Unix98 ptys configured
# RAMDISK driver initialized: 16 RAM disks of 208K size 1024 blocksize
# NetWinder Floating Point Emulator V0.97 (double precision)
# RAMDISK: ext2 filesystem found at block 8388608
# RAMDISK: Loading 200 blocks [1 disk] into ram disk... done.
# Freeing initrd memory: 200K
```

Figure 25: Linux boot print on Amber FPGA system

7. Major Obstacles

Getting all the required resources took us a while since I could not find an organized set of documentation describing what was necessary. I eventually found that there was a lot of documentation and references available from different sources. However, they were a bit all over the place and finding them was not straightforward. Also, I had trouble getting the Amber source code from opencores.

It took some time for me to get familiar with Altera tools since I didn't have any prior experience on them. Once I started exploring the system I figured few major obstacles. I used an incremental approach and decided to first port the Amber system on FPGA and then later focus on interfacing it with ARM HPS. First obstacle was that there are no UART port on DE1-SoC fabric. There were different ways to resolve the issue – make custom GPIO to RS-232

daughter board or use separate FPGA board. Since the idea was to implement the system first on FPGA fabric, so I chose to use DE2-115 Cyclone IV FPGA board. Once the porting is successful then the next step would be to build Wishbone Master to Avalon slave and then use ARM HPS UART controller for serial communication.

Another major obstacle was that the software tools provided to compile bootloader and standalone applications generates “.elf” and “.mem” files. No tool support for “.mif” file generation. Since M9K RAM in Altera FPGA only supports “.mif” or “.hex” file format for memory initialization, I had to convert from “.mem” file format to “.mif” file format.

One of the biggest obstacle was that Amber FPGA system had a bridge to support interface with Xilinx DDR controller from the Wishbone Bus interface. However, since Altera has different controller the same bridge wouldn't work. So one task was to write a bridge to interface Altera SDRAM/DDR controller with Wishbone and test it independently before integrating it into the main system. For simplicity, I used on-chip M9K RAM as main memory but with small size of 16kBytes as compared to 128Mbytes in original system.

After integrating the entire system, when I download the bitfile on FPGA, PLL lock LED and system ready LED glow. This indicates that system is completely out of reset and PLL is completely locked. But I didn't see any print on UART port. I built a small project to test the UART port independently. On looking at the bootloader startup code, I realized the cause of system hang. In startup code, after configuring caches and interrupts, the processor would then set the stack pointer to 0x0200_0000 location. Since this points to location in main memory which doesn't exist, the system would raise exception and get stuck. One probable solution is to change the startup and bootloader code not to use any address space beyond 0x0000_3FFF. Another solution is to integrate SDRAM/DDR controller into the system to get the exact address space from 0x0000_0000 to 0x07FF_FFFF for main memory.

8. Acknowledgement

I would like to thank Professor Bruce Land for giving me the opportunity to be involved with this project, as well as for his guidance and inspiration.

9. Conclusion

I have made progress in both sections of the independent study – the hardware implementation of the Amber System and the software programming section. I have created an Amber FPGA system for Altera that also contains memory initialization file for bootloader code. I have familiarized myself with the Quartus tools and the software tools for cross-compiling application for Amber. I am currently stuck with an obstacle, which I believe I would be able to solve soon and accelerate progress. I hope to continue this next semester

A. References

1. <http://opencores.org/project.amber>
2. <http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=836&PartNo=4>
3. ftp://ftp.altera.com/up/pub/Altera_Material/Boards/DE2-115/DE2_115_User_Manual.pdf
4. <http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/ARM-Options.html#ARM-Options>
5. <http://sourceware.org/binutils/docs-2.21/ld/ARM.html#ARM>
6. http://quartushelp.altera.com/15.0/mergedProjects/reference/glossary/def_mif.htm
7. https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/HPS_peripherals/index.html
8. <https://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/>
9. https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/External_Bus_to_Avalon_Bridge.pdf
10. http://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/index.html