ECE3375B Lab 1: Hardware Familiarization

Objective

The main purpose of this lab exercise is to gain familiarity with the equipment and development environment we will be using in the lab portion of ECE3375.

Version History: This lab manual was originally written by Prof. Ken McIsaac and used in ECE3375 from 2019 to 2022. It was rewritten by Prof. John McLeod in 2023.

This lab exercise has a mixture of activities: writing code, programming hardware, and using debugging tools. Of course, *reading this lab manual* is also part of the exercise, so why don't you go ahead and do that. There is no *explicit* pre-lab, but your laboratory experience will be immensely improved if you are familiar with this manual and have completed a few simple tasks *before* showing up to the lab.

Remember that all the features of hardware and the debugging interface are available in the online simulator: https://cpulator.01xz.net/?sys=arm-de1soc.

Deliverables

This lab manual contains extensive background information and step-by-step instructions designed to help familiarize you with programming the ARM Cortex A9 and using the monitoring and debugging features in the Intel FPGA Monitor program. All these features are also present in the simulator, however, so it is easy to gain familiarity without direct access to the lab.

Here is what you will be graded on in this lab:

- Write the code (in Assembly or C, your choice) to read the 4-bit binary value set by the first four slide switches and display that value as a single hex digit on the first seven-segment display panel. Your code should run in a continuous loop, reading the switches and displaying the value. Your code should use **subroutines** to handle reading from switches and writing a digit to the seven-segment display panel. I know it seems like extra work, but this will allow you to easily reuse this code in future lab exercises. Template files (`Lab1Part2.s` for Assembly and `Lab1Part2.c` for C) are available.
- Program the DE10-Standard with this code, demonstrate the working device to the TA, and answer their questions about the code, the DE10-Standard, and the Intel FPGA Monitor program.

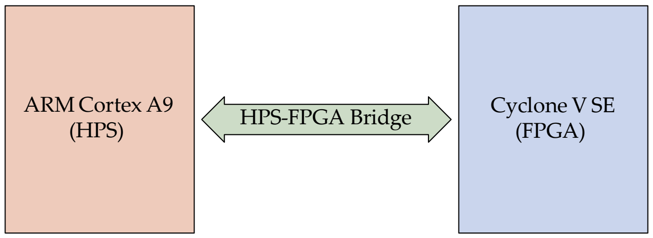
You do not need to submit any report, simply demonstrate your working device, show your source code for the hex display driver, and answer questions to receive full marks.

Most of this lab manual is a tutorial for working with the Intel FPGA Monitor program. You won't be graded on this, so I suppose you can skip it. At any point in the future, however, if you ask questions about the Intel FPGA Monitor program, you will receive the polite suggestion to go back and finish this tutorial.

Background Information

For the labs in this course, we will be working on the “bare metal”. We are going to be writing and debugging programs that run directly on the ARM hardware with very little software in between: no operating system, no device drivers, etc. As such, the best place to start is to explain what exactly it is that we are working with.

The hardware we will use in the lab is the DE-10 standard platform that you probably remember from digital logic class (ECE2277). The DE-10 is basically an Altera Cyclone V system-on-chip (SoC) surrounded by some input/output devices. The Cyclone V is a fascinating piece of electronics. A schematic representation of the Cyclone V’s structure is shown below.



Basically, it has two interconnected halves.

- One half is the hard processor system (HPS), which is an ARM Cortex A9 microprocessor. That’s all it is, and all it can ever be — the hardware is immutable.
- The other half is a field-programmable gate array (FPGA), and it is as reconfigurable as any other FPGA — if it can be constructed from logic arrays, you can implement any hardware you want.
- Then there’s the HPS-FPGA bridge. This is a programmable set of interconnects that allows you to map the FPGA into the ARM’s memory space. This means you can add hardware functionality to the microcontroller almost at will, limited only by the resources in the FPGA. If there is a peripheral device the ARM Cortex A9 doesn’t have, you can make it, put it into the FPGA and map it into the ARM using the bridge, then control it using software written for ARM.

As succinctly expressed by a previous professor in this course; “this is *really, really* cool”. It is, however, “also *really, really* complicated” (same professor). To program the processor, you must first program the FPGA. In essence, you have to first tell the device how to configure itself, then you can download a “preloader”, then you can download application software.

Here’s the good news: You *can* take advantage of these features for the purposes of the design project later in the course, if you want to do that. However, you don’t actually *have* to do any it, because Intel has made available a piece of software called the Intel FPGA Monitor. The Monitor makes all of the above simple — and mostly invisible — in practice.

Intel FPGA Monitor

When working with a new microcontroller, once you can light up a single LED, you can do anything. In this section, we will create a simple project in the Monitor to make one of the LEDs on the DE-10 board to blink.

Directory Preparation

The Intel FPGA Monitor seems to be a program that was written a while ago and hasn't been updated much. In particular, the Monitor does not seem to like long complicated path names. As such, it is recommended to create a folder on your `H:\` drive with a short, simple name for use with the labs in the course (such as `ECE3375` perhaps?). Within that folder create a subfolder for this lab (perhaps call it `Lab1`?)

Once your directories are prepared, go to the *Laboratory* section on OWL for this course and obtain the sample code files for this lab.

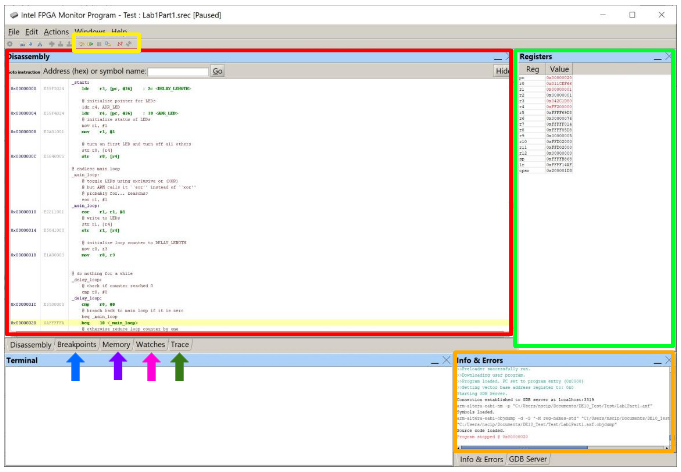
- Use `Lab1Part1.s` if you will complete the lab in assembly.
- Use `Lab1Part1.c` if you will complete the lab in C.

Of course you can grab all the files if you want, but you only need to work with one language. Since your `H:\` drive is (or *should be*) common on every computer platform at Western Engineering, you should be able to do this step before coming to the lab.

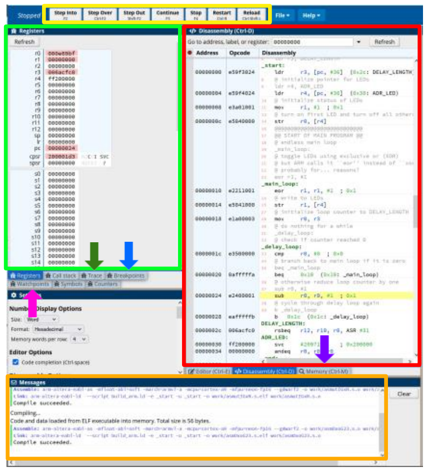
It might also be a good idea to take a look at the source code *before* coming to the lab, just so you know what to expect.

Creating the Project

Once you get to the lab, open the Intel FPGA Monitor. There should be an icon on the desktop called "Intel FPGA Monitor Program" or something similar. When you open it, you should see the main screen of the Monitor, which has mostly the same windows as the on-line simulator but with a slightly different layout.


Intel FPGA Monitor Program - Test : Lab1Part1.srec [Paused]
File Edit Actions Windows Help
Disassembly
Registers
Terminal
Info & Errors
Disassembly Breakpoints Memory Watches Trace
Info & Errors GDB Server

Intel FPGA Monitor

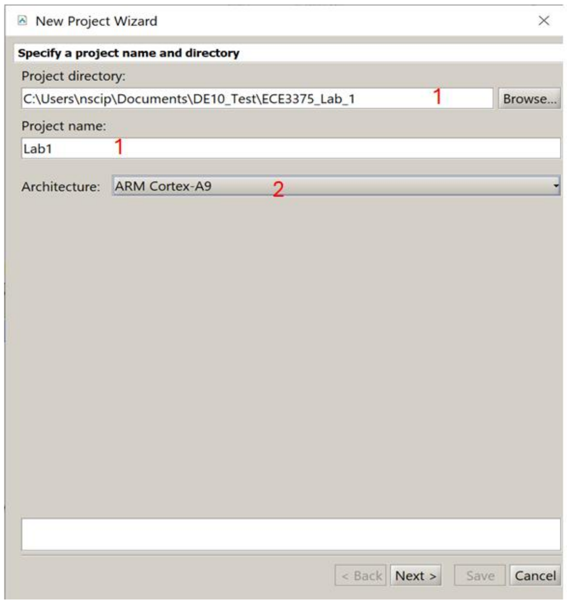


CPUlator

The Monitor *might* include a text editor for modifying code, but we've heard mixed reports on whether or not this works. It might be easiest to use another editor (probably Notepad, unless the lab computers have something else installed) to actually write or modify the code. As long as all code is saved to the correct directory, the Monitor program should be able to find it.

To begin, we need to create a project. Go to **File→New Project** in the Monitor Program and follow these steps:

1. Select the directory you created previously on your `H:\` as the project directory and give your project a simple name.
2. Select `ARM Cortex-A9` in the drop-down menu for *Architecture*.

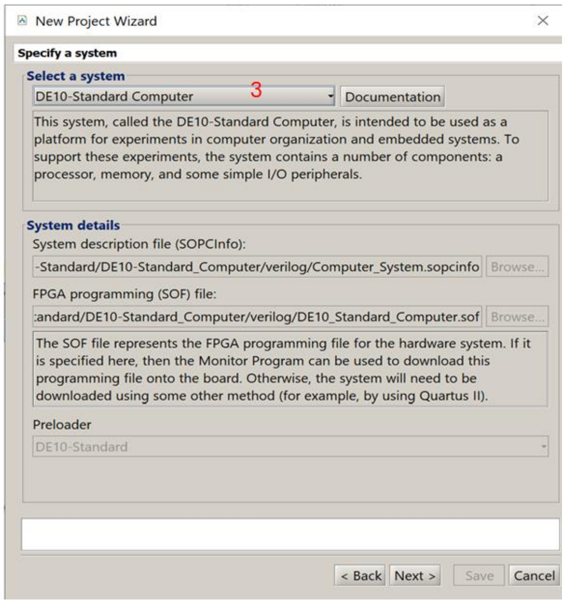


Specify a project name and directory

Project directory: C:\Users\nscip\Documents\DE10_Test\ECE3375_Lab_1

Project name: Lab1

Architecture: ARM Cortex-A9



Specify a system

Select a system: DE10-Standard Computer

This system, called the DE10-Standard Computer, is intended to be used as a platform for experiments in computer organization and embedded systems. To support these experiments, the system contains a number of components: a processor, memory, and some simple I/O peripherals.

System details

System description file (SOPCInfo): -Standard/DE10-Standard_Computer/verilog/Computer_System.sopcinfo

FPGA programming (SOF) file: :andard/DE10-Standard_Computer/verilog/DE10_Standard_Computer.sof

The SOF file represents the FPGA programming file for the hardware system. If it is specified here, then the Monitor Program can be used to download this programming file onto the board. Otherwise, the system will need to be downloaded using some other method (for example, by using Quartus II).

Preloader: DE10-Standard

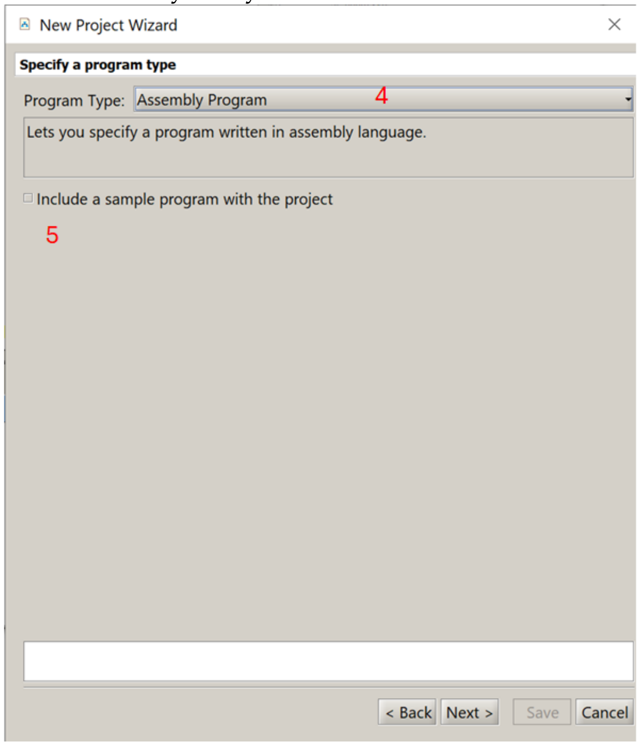
Press **Next** when you have completed the above steps.

3. Select `DE10-Standard Computer` as the system.

Press **Next** when you have completed the above step. In the subsequent window you need to specify your programming language. You can program the lab in C if you are familiar with the language, but we won't be teaching C in the lessons.

4. Select Assembly or C.

5. Leave the box about using a sample program unchecked — you already have sample code in the project directory that you downloaded from OWL.

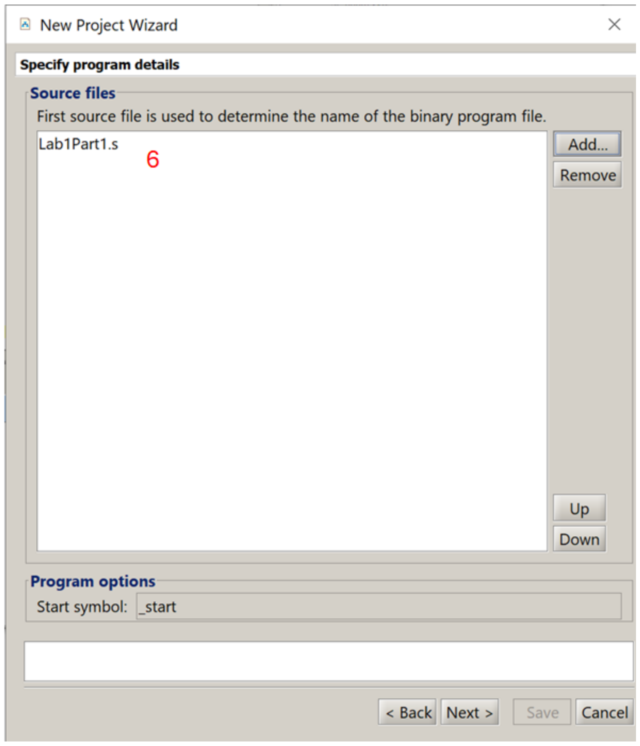


Specify a program type

Program Type: Assembly Program

Lets you specify a program written in assembly language.

☐ Include a sample program with the project



Specify program details

Source files

First source file is used to determine the name of the binary program file.

Lab1Part1.s

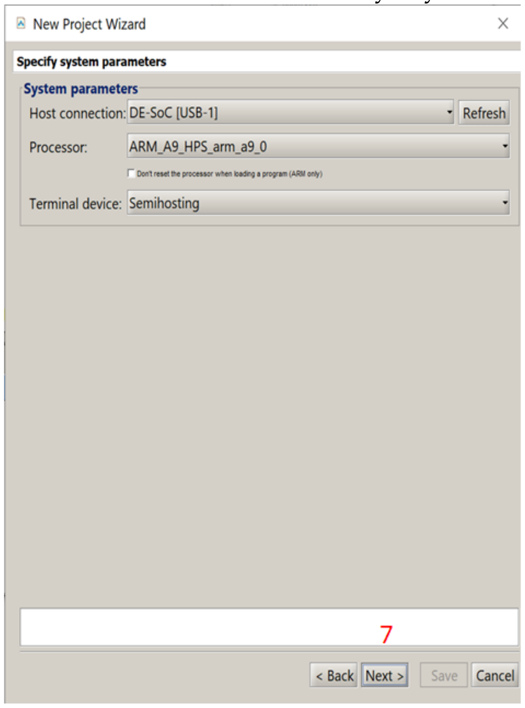
Program options

Start symbol: _start

Press **Next**. This window allows you to add files to your project.

6. Add `Lab1Part1.s` if you are programming in Assembly, or `Lab1Part1.c` if you are programming in C.

7. You do not need to modify any of the other options.



Specify system parameters

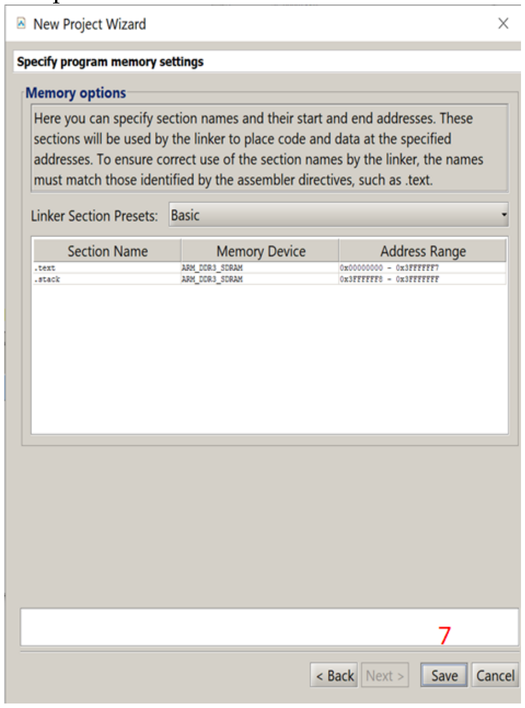
System parameters

Host connection: DE-SoC [USB-1] Refresh

Processor: ARM_A9_HPS_arm_a9_0

☐ Don't reset the processor when loading a program (ARM only)

Terminal device: Semihosting



Specify program memory settings

Memory options

Here you can specify section names and their start and end addresses. These sections will be used by the linker to place code and data at the specified addresses. To ensure correct use of the section names by the linker, the names must match those identified by the assembler directives, such as .text.

Linker Section Presets: Basic

Section Name	Memory Device	Address Range
.text	ARM_DDR3_SDRAM	0x00000000 - 0x3FFFFFF7
.stack	ARM_DDR3_SDRAM	0x3FFFFFF8 - 0x3FFFFFFF

After this point, continue to press **Next** and accept all default options until the last window (the **Next** button becomes greyed out). You can then click **Save** to create the project and start the debugging session.

8. A prompt should appear asking if you want to download the system to the board. Double check that the DE10-Standard is plugged into the computer and powered on, then click **Yes**.

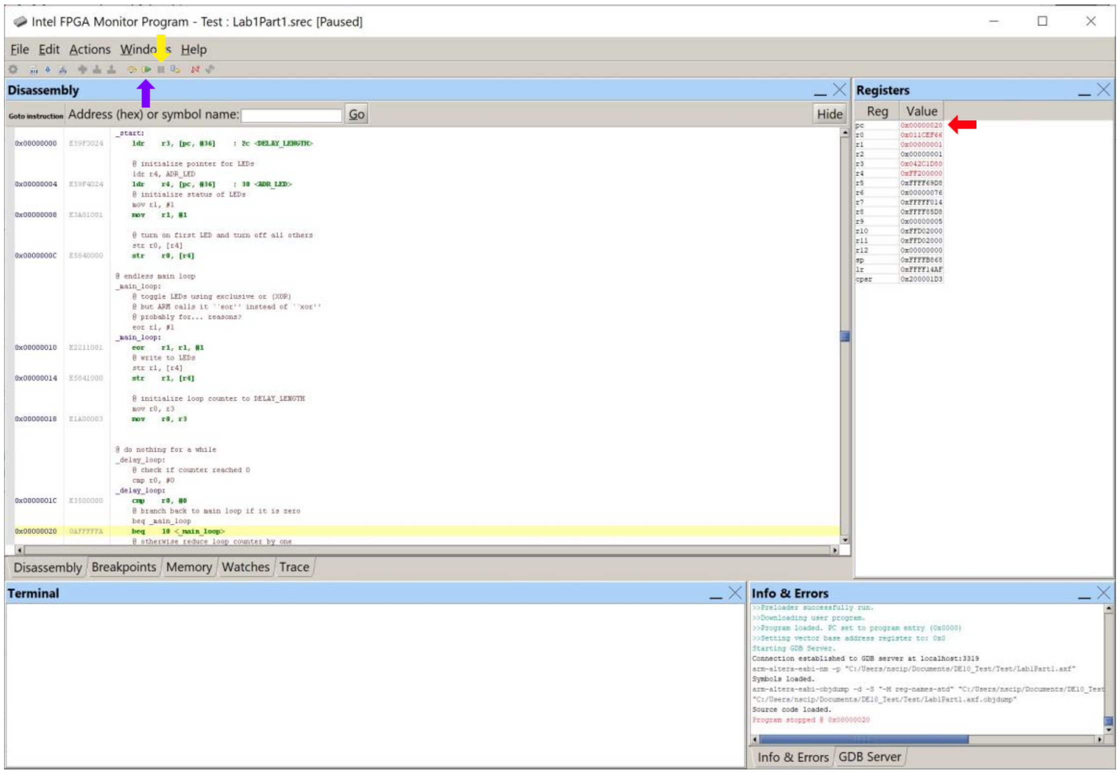
9. Things will happen, LEDs may turn on an off. Eventually it should tell you that the system has been successfully downloaded onto the board.

Once you are successful in downloading the system to the board, you can start studying how the code runs in hardware using the debugging tools.

Programming a Blinking LED

Now we want to try out some of the things the Monitor can do, and experiment with making simple program changes. Thus far, we have created a project and configured the board. However, we have not yet done anything — our program hasn't been compiled and programmed in hardware. In fact, the **Disassembly** window in the Monitor is probably still blank. To fix that, we need to Load a program. Choose **Actions→Compile & Load**.

You can watch the **Info & Errors** window to see the progress of this activity. In this case, the program should compile without errors, but if compilation errors were to occur, that's where you'd see them. The **Load** process might take a bit of time. Once it is done, the monitor will be much more interesting. It should look something like what is shown below.


Intel FPGA Monitor Program - Test : Lab1Part1.srec [Paused]
File Edit Actions Windows Help
Disassembly
Goto instruction Address (hex) or symbol name: Go
Hide
Registers
Disassembly Breakpoints Memory Watches Trace
Terminal
Info & Errors
Info & Errors GDB Server

```
                    _start:
0x00000000  E59F3024    ldr    r3, [pc, #36]    ; 2c <DELAY_LENGTH>

                        @ initialize pointer for LEDs
                        ldr r4, ADR_LED
0x00000004  E59F4024    ldr    r4, [pc, #36]    ; 30 <ADR_LED>
                        @ initialize status of LEDs
                        mov r1, #1
0x00000008  E3A01001    mov    r1, #1

                        @ turn on first LED and turn off all others
                        str r0, [r4]
0x0000000C  E5840000    str    r0, [r4]

                    @ endless main loop
                    _main_loop:
                        @ toggle LEDs using exclusive or (XOR)
                        @ but ARM calls it ``eor'' instead of ``xor''
                        @ probably for... reasons?
                        eor r1, #1
                    _main_loop:
0x00000010  E2211001    eor    r1, r1, #1
                        @ write to LEDs
                        str r1, [r4]
0x00000014  E5841000    str    r1, [r4]

                        @ initialize loop counter to DELAY_LENGTH
                        mov r0, r3
0x00000018  E1A00003    mov    r0, r3


                    @ do nothing for a while
                    _delay_loop:
                        @ check if counter reached 0
                        cmp r0, #0
                    _delay_loop:
0x0000001C  E3500000    cmp    r0, #0
                        @ branch back to main loop if it is zero
                        beq _main_loop
0x00000020  0AFFFFFA    beq    10 <_main_loop>
                        @ otherwise reduce loop counter by one
```

Reg	Value
pc	0x00000020
r0	0x011CEF66
r1	0x00000001
r2	0x00000001
r3	0x042C1D80
r4	0xFF200000
r5	0xFFFF69D8
r6	0x00000076
r7	0xFFFFF014
r8	0xFFFF85D8
r9	0x00000005
r10	0xFFD02000
r11	0xFFD02000
r12	0x00000000
sp	0xFFFFB868
lr	0xFFFF14AF
cpsr	0x200001D3

```
>>Preloader successfully run.
>>Downloading user program.
>>Program loaded. PC set to program entry (0x0000)
>>Setting vector base address register to: 0x0
Starting GDB Server.
Connection established to GDB server at localhost:3319
arm-altera-eabi-nm -p "C:/Users/nscip/Documents/DE10_Test/Test/Lab1Part1.axf"
Symbols loaded.
arm-altera-eabi-objdump -d -S "-M reg-names-std" "C:/Users/nscip/Documents/DE10_Test
"C:/Users/nscip/Documents/DE10_Test/Test/Lab1Part1.axf.objdump"
Source code loaded.
Program stopped @ 0x00000020
```

Here I ran the program for a bit and then stopped it, so the contents of my registers and the line highlighted in the Disassembly window are not the same as you will see after just loading your program.

What you are seeing in the **Disassembly** window is an assembly language program, *reconstructed* from the contents of the ARM's memory. The Monitor knows your "symbol table" from your source code file (all the names of your functions, subroutines, variables and other labels), so it can turn the disassembly into something very nearly readable.

On the right, you will see the **Registers** window. That is the *actual register contents* of the ARM processor at this moment: it looks and behaves exactly like the **Registers** window in the simulator, but in this case it is a true representation of what is actually occurring in the machine in front of you.

- If you look at the top of the **Registers** window, for example, you will see that the first register listed is the program counter (pc).
- At this point, the contents of pc should be the address of the first instruction the compiler put into your program. It is also the address of the highlighted line of disassembled code in the **Disassembly** window. If you are using Assembly in your project, probably pc is 0x00000000. If you are using C in your project, probably pc is 0x00000128, because the C compiler creates a bunch of extra (and unnecessary in this simple project) initialization instructions.

OK! Let's see some blinking lights. Press **Continue** (it looks like ▶).

- An LED should blink. It is probably blinking approximately twice a second, i.e. a frequency of roughly 2 Hz.
- If you get bored of looking at it, you can push **Break** (it looks like ⏸) and it will stop.

Also, when you press **Break**, you will find yourself now in the program. The Monitor is smart enough to put the disassembled code reconstructed from the memory contents *in line with the source*. What this means is that you can see the lines of the original program code interspersed with the assembly code the compiler has generated.

- If you are coding in Assembly the result is probably pretty close to the original, although some surprising differences can occur based on compiler optimization settings.
- If you are coding in C the result will obviously be less similar.

You are almost certainly somewhere in the delay loop, which causes the system to pause for about half a second between toggling the LED on and off.

Now, let's do some stuff. A very useful function is *single step debugging*. **Step** looks like this: ↷, and it will cause the program counter to advance by one *assembly* instruction (if you chose to program in C, it will *not* advance by one line of C code, as most single lines of C translate to multiple lines of Assembly).

What we want to do next is step out of the time delay loop. Press single step a few times and you can watch the program counter chase its way around the loop.

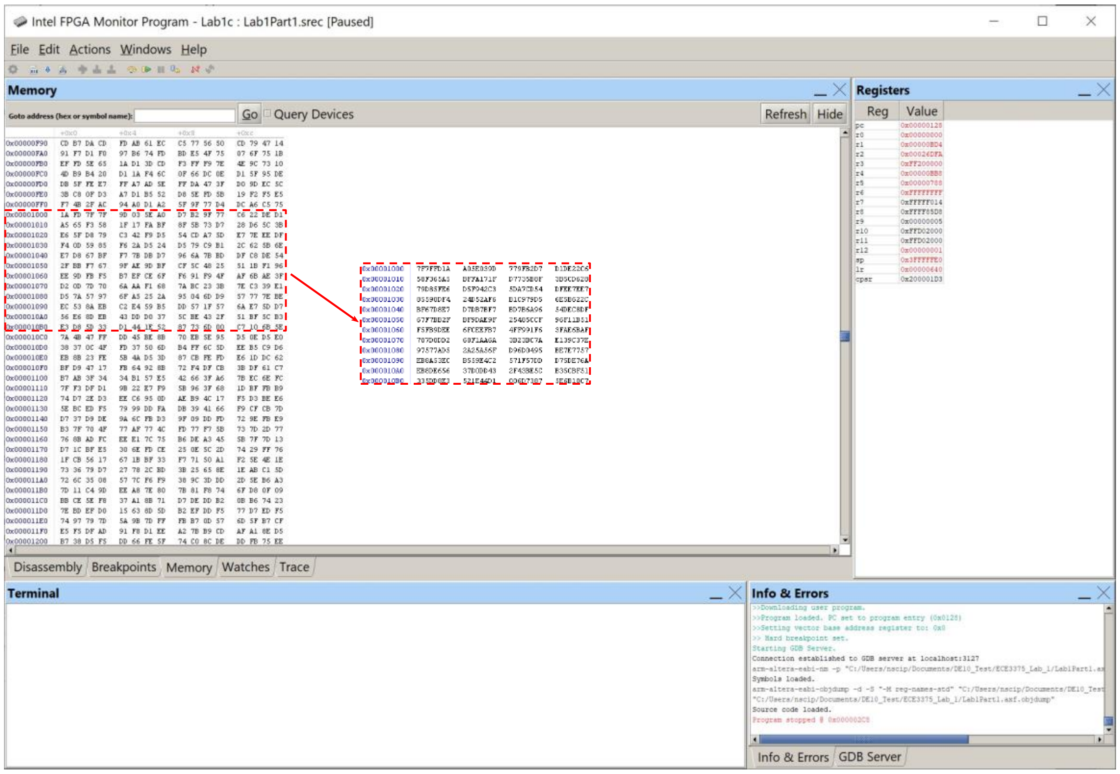
- Wow. That's (probably) boring.
- The delay loop was set to a staggeringly large number of cycles. Unless you were amazingly lucky when you hit **Break**, you probably have tens of thousands of loop iterations left to go.

Fortunately, you are all powerful, and there is a better way to exit the loop than by just cycling through it until the counter exhausts itself.

- First, press **Step** a few times until you reach to the instruction where the loop test occurs.
- This loop test should be a comparison (`cmp` mnemonic in Assembly) between the register used as a loop counter and the register used to hold the number of loops. (Look for the register whose value changes each pass through the loop — that is the one used as the loop counter).

In the **Registers** window, you can simply change the register value so the loop will terminate: just double click on the register, type a new value, and hit enter. This *actually changes the state of the hardware* in the DE10-Standard board.

So that's a useful trick. What else can we do? Well, suppose you decide you don't like the length of the delay loop and you'd like it to be slower (or faster, whatever you prefer). Of course, we can re-compile and download again, but we can also change the behaviour at run time. Click the **Memory** tab under the main window. You will see something like the example shown below.


Intel FPGA Monitor Program - Lab1c : Lab1Part1.srec [Paused]
File Edit Actions Windows Help
Memory
Goto address (hex or symbol name):
Go
Query Devices
Refresh
Hide
Registers
Disassembly
Breakpoints
Memory
Watches
Trace
Terminal
Info & Errors
GDB Server

```
              +0x0          +0x4          +0x8          +0xc
0x00000F90   CD B7 DA CD   FD AB 61 EC   C5 77 56 50   CD 79 47 14
0x00000FA0   91 F7 D1 F0   97 B6 74 FD   BD E5 4F 75   07 6F 75 1B
0x00000FB0   EF FD 5E 65   1A D1 3D CD   F3 FF F9 7E   4E 9C 73 10
0x00000FC0   4D B9 B4 20   D1 1A F4 6C   0F 66 DC 0E   D1 5F 95 DE
0x00000FD0   DB 5F FE E7   FF A7 AD 5E   FF DA 47 3F   D0 9D EC 5C
0x00000FE0   3B C8 0F D3   A7 D1 B5 52   D8 5E FD 5B   19 F2 F5 E5
0x00000FF0   F7 4B 2F AC   94 A0 D1 A2   5F 9F 77 D4   DC A6 C5 75
0x00001000   1A FD 7F 7F   9D 03 5E A0   D7 B2 9F 77   C6 22 DE D1
0x00001010   A5 65 F3 58   1F 17 FA BF   8F 5B 73 D7   28 D6 5C 3B
0x00001020   E6 5F D8 79   C3 42 F9 D5   54 CD A7 5D   E7 7E EE DF
0x00001030   F4 0D 59 85   F6 2A D5 24   D5 79 C9 B1   2C 62 5B 6E
0x00001040   E7 D8 67 BF   F7 7B DB D7   96 6A 7B BD   DF C8 DE 54
0x00001050   2F BB F7 67   9F AE 9D BF   CF 5C 48 25   51 1B F1 96
0x00001060   EE 9D FB F5   B7 EF CE 6F   F6 91 F9 4F   AF 6B AE 3F
0x00001070   D2 0D 7D 70   6A AA F1 68   7A BC 23 3B   7E C3 39 E1
0x00001080   D5 7A 57 97   6F A5 25 2A   95 04 6D D9   57 77 7E BE
0x00001090   EC 53 8A EB   C2 E4 59 B5   DD 57 1F 57   6A E7 5D D7
0x000010A0   56 E6 8D EB   43 DD D0 37   5C BE 43 2F   51 BF 5C B3
0x000010B0   E3 D8 5D 33   D1 44 1E 52   87 73 6D 00   C7 10 6B 5E
0x000010C0   7A 4B 47 FF   DD 45 BE 8B   70 EB 5E 95   D5 0E D5 E0
0x000010D0   38 37 0C 4F   FD 37 50 6D   B4 FF 6C 5D   EE B5 C9 D6
0x000010E0   EB 8B 23 FE   5B 4A D5 3D   87 CB FE FD   E6 1D DC 62
0x000010F0   BF D9 47 17   FB 64 92 8B   72 F4 DF CB   3B DF 61 C7
0x00001100   B7 AB 3F 34   34 B1 57 E5   42 66 3F A6   7B EC 6E FC
0x00001110   7F F3 DF D1   9B 22 E7 F9   5B 96 3F 68   1D BF FB B9
0x00001120   74 D7 2E D3   EE C6 95 0D   AE B9 4C 17   F5 D3 BE E6
0x00001130   5E BC ED F5   79 99 DD FA   DB 39 41 66   F9 CF CB 7D
0x00001140   D7 37 D9 DE   9A 6C FB D3   9F 09 DD FD   72 9E FB E9
0x00001150   B3 7F 70 4F   77 AF 77 4C   FD 77 F7 5B   73 7D 2D 77
0x00001160   76 8B AD FC   EE E1 7C 75   B6 DE A3 45   5B 7F 7D 13
0x00001170   D7 1C BF E5   30 6E FD CE   25 0E 5C 2D   74 29 FF 76
0x00001180   1F CB 56 17   67 1B BF 33   F7 71 50 A1   F2 5E 4E 1E
0x00001190   73 36 79 D7   27 78 2C BD   3B 25 65 8E   1E AB C1 5D
0x000011A0   72 6C 35 08   57 7C F6 F9   38 9C 3D DD   2D 5E B6 A3
0x000011B0   7D 11 C4 9D   EE A8 7E 80   7B 81 F8 74   6F D8 0F 09
0x000011C0   BB CE 5E F8   37 A1 8B 71   D7 DE DD B2   0B B6 74 23
0x000011D0   7E BD EF D0   15 63 8D 5D   B2 EF DD F5   77 D7 ED F5
0x000011E0   74 97 79 7D   5A 9B 7D FF   FB B7 0D 57   6D 5F B7 CF
0x000011F0   E5 F5 DF AD   91 F8 D1 EE   A2 7B B9 CD   AF A1 8E D5
0x00001200   B7 38 D5 F5   DD 66 FE 5F   74 C0 8C DE   DD FB 75 EE
```

```
0x00001000   7F7FFD1A   A05E039D   779FB2D7   D1DE22C6
0x00001010   58F365A5   BFFA171F   D7735B8F   3B5CD628
0x00001020   79D85FE6   D5F942C3   5DA7CD54   DFEE7EE7
0x00001030   85590DF4   24D52AF6   B1C979D5   6E5B622C
0x00001040   BF67D8E7   D7DB7BF7   BD7B6A96   54DEC8DF
0x00001050   67F7BB2F   BF9DAE9F   25485CCF   96F11B51
0x00001060   F5FB9DEE   6FCEEFB7   4FF991F6   3FAE6BAF
0x00001070   707D0DD2   68F1AA6A   3B23BC7A   E139C37E
0x00001080   97577AD5   2A25A56F   D96D0495   BE7E7757
0x00001090   EB8A53EC   B559E4C2   571F57DD   D75DE76A
0x000010A0   EB8DE656   37D0DD43   2F43BE5C   B35CBF51
0x000010B0   335DD8E3   521E44D1   006D7387   5E6B10C7
```

Reg	Value
pc	0x00000128
r0	0x00000000
r1	0x00000BD4
r2	0x00026DFA
r3	0xFF200000
r4	0x00000BB8
r5	0x00000788
r6	0xFFFFFFFF
r7	0xFFFFF014
r8	0xFFFF85D8
r9	0x00000005
r10	0xFFD02000
r11	0xFFD02000
r12	0x00000001
sp	0x3FFFFFE0
lr	0x00000640
cpsr	0x200001D3

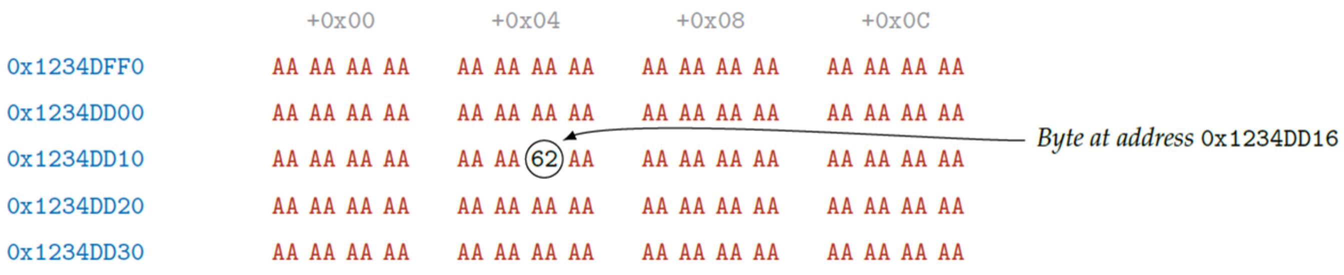
```
>>Downloading user program.
>>Program loaded. PC set to program entry (0x0128)
>>Setting vector base address register to: 0x0
>> Hard breakpoint set.
Starting GDB Server.
Connection established to GDB server at localhost:3127
arm-altera-eabi-nm -p "C:/Users/nscip/Documents/DE10_Test/ECE3375_Lab_1/Lab1Part1.ax
Symbols loaded.
arm-altera-eabi-objdump -d -S "-M reg-names-std" "C:/Users/nscip/Documents/DE10_Test
"C:/Users/nscip/Documents/DE10_Test/ECE3375_Lab_1/Lab1Part1.axf.objdump"
Source code loaded.
Program stopped @ 0x000002C8
```

You may see quite different contents in memory for your project. Here the inset shows the memory viewed in words, the original image shows memory viewed in bytes. (It is possible that your Monitor program will show the memory as words by default.)

That is the *actual memory contents* of the ARM's RAM, organized in words (four bytes, or 8 hex digits) and visually tabulated with four words per row. This makes it fairly convenient to find a specific address in memory; as each word is four bytes and the memory is tabulated with four words per row, that provides 4 × 4 = 16 bytes of memory on each row. Since the memory is addressed in hexadecimal, the memory addresses on each row differ only by the least significant hex digit. To find the byte of memory at address 0x1234DD16, for example:

1. Scroll to find the row that starts with the base address 0x1234DD10.
2. The LSb in the address 0x1234DD16 is 6, so count 6 + 1 = 7 bytes (indexing starts at zero) over from the left.

The following graphic hopefully makes the situation clearer.



```
              +0x00         +0x04         +0x08         +0x0C
0x1234DFF0   AA AA AA AA   AA AA AA AA   AA AA AA AA   AA AA AA AA
0x1234DD00   AA AA AA AA   AA AA AA AA   AA AA AA AA   AA AA AA AA
0x1234DD10   AA AA AA AA   AA AA 62 AA   AA AA AA AA   AA AA AA AA
0x1234DD20   AA AA AA AA   AA AA AA AA   AA AA AA AA   AA AA AA AA
0x1234DD30   AA AA AA AA   AA AA AA AA   AA AA AA AA   AA AA AA AA
```

Byte at address 0x1234DD16

For our purposes we want to change the number of loops, which is a large value and is stored as an entire word. Go back to the disassembly and figure out the address that has been assigned to the variable DELAY_LENGTH. Type the address into the **Goto Address** bar of the **Memory** window.

Assuming you didn't modify the original source code that was provided on OWL, the value DELAY_LENGTH is defined as 70 000 000 in the Assembly code, which is 0x42C1D80. If you are using the source code in C, then DELAY_LENGTH is only 700 000, which is 0xAAE60 (this should tell you something about how much more efficient Assembly is).

- Unfortunately, the ARM's *endian-ness* means that it is kind of hard to read: bytes of lesser significance are to the *left* of bytes of greater significance, so the number 0x042C1D80 is actually stored in bytes as 0x80 1D 2C 04 (or for C code, the value is 0x000AAE60 which is actually stored in bytes as 0x60 AE 0A 00).

- Fortunately, if you right click the word in memory, you have the choice to view the data as **Byte** (1-byte), **Half-word** (2-bytes) or **Word** (4-bytes). Put it into **Word** (4-bytes) mode, and the number is much easier to read.
- The right-click popup menu also gives you the option to choose **Number format**. If you choose **Decimal**, then the value (70 000 000 or 700 000 for Assembly or C) becomes very easy to see.

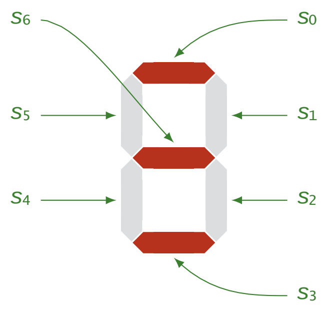
Now, change it to something else and watch the blink rate change as you vary the number (the value should still be fairly large or the LED will blink too fast to see).

Here's one more useful debugger feature before we move on to writing our own program. Go back to the **Disassembly** window and click in the gray margin along the left hand side. A red dot should appear. This is a **Break- point**. Execution will immediately halt if the program counter gets to that location in the program memory.

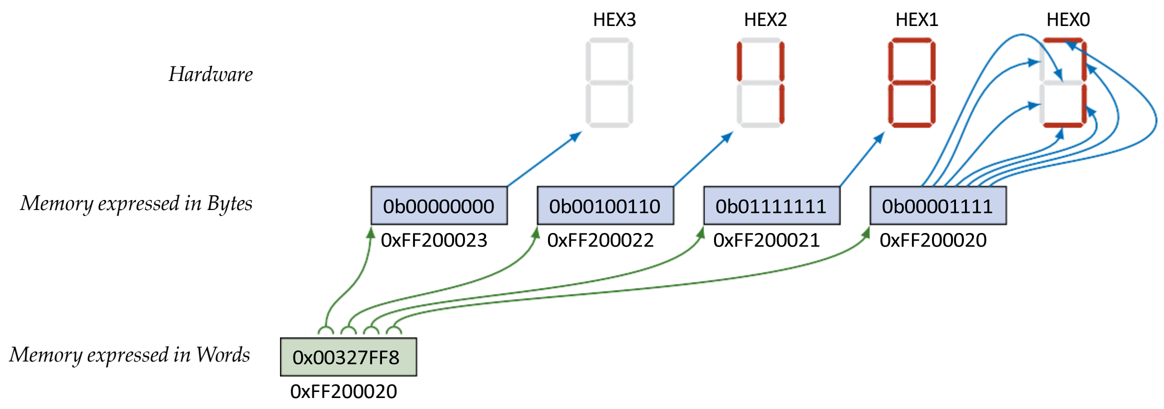
Programming a Hex Display Driver

Now it is time for you to do something. We want to use the lowest four slide switches (of the set of 10) to control the hex character displayed on the seven segment display. For this we need some information.

- The slide switches are memory mapped to address 0xFF200040 (called SW_BASE in the sample code files we provided on OWL).
- The lower order four seven segment displays are mapped to address 0xFF200020 (called HEX3_HEX0_BASE) in the sample code files we provided on OWL).
- The wiring of port I/O bits to segments in the seven segment display is as in the following diagram.



- These seven bits correspond to the *least-significant seven bits* in the byte for the appropriate display panel at memory address 0xFF200020 (the most significant bit in the byte is ignored).



The above graphic demonstrates what happens when the value 0x00327FF8 is written to address 0xFF200020. In this lab you should only set the value of HEX0, so either write only a byte to address 0xFF200020 or write the word 0x000000?? where only the last byte is changed to reflect the digit given on the slide switches.

- In Assembly, use `ldrb` and `strb` to access bytes in memory. In C, use variables of type `char`.
- In Assembly, use `ldr` and `str` to access the entire word in memory. In C, use variables of type `int`.

It is strongly recommended that you develop the basic program implementing the hex driver as described above, so you can then devote time in lab to testing and debugging. Your program must have the following features:

1. The code to read the slide switch bank and obtain the current value, encoded as a binary number. This should be implemented as a subroutine.
2. The code to display *any* single hex digit on the lowest-order seven segment display (i.e., `HEX0`). For example, if you want to display the digit "1", you need to turn on only the segments $\mathbf{s}_1$ and $\mathbf{s}_2$ in the picture. So your code needs to write the value `0b0000110` = `0x06` to `HEX0` if a value `1` is given. This code should be implemented as a subroutine.
3. The main program loop should blink the `HEX0` seven segment display on and off. When it is on, it should be displaying the hex digit encoded by the lowest for slide switches. When off, the whole display should be blank. You can use the same delay loop from the previous part of this lab to control the time of the blinking — it doesn't really matter the exact speed as long as it blinks in manner easily observable to humans.

The TAs will test your code by toggling the switches and seeing if the correct number is displayed. They may also ask you some questions about how everything works.

Evaluation

You will be evaluated based on demonstrating working hardware, completing the seven segment display driver code according to specifications, and answering questions from the TA.

Category	Grade
Demonstrating working Seven Segment Display Driver in Hardware	20
Seven Segment Display Driver Code Completed to Specifications	20
Answering Questions	10

Partial marks may be awarded in each category for partial completion.