Lab 4: Implementation of a Memory-Mapped Slave Peripheral

Introduction

This lab introduces a lot of new material, and begins the re-integration of assembly language programming experience from ECE243 or a similar course. First, you will learn a brand new tool: the Intel Platform Designer (formerly Qsys) Tool. You will use it to generate a hardware system containing a Nios II processor and other peripherals. Systems created by Qsys are Verilog code that can be synthesized for the FPGA on the DE1-SoC board. The Monitor Program, which communicates with the Nios II processor in this system via the USB cable, can then be used to compile, load, run, and debug assembly and C-language programs. The final goal of this lab will be to be able to control your Line Drawer peripheral from Lab 3 from a software program running on a Nios II processor that lives within a Qsys-generated system. Figure 1 shows a block diagram of the final system.

NOTE 1: The "Qsys System Integration" Tool has been renamed to "Intel Platform Designer". Mentions of Qsys in this lab document refer to the new "Platform Designer" tool. Similarly, any references to "Altera" are synonymous with "Intel FPGA".

NOTE 2: Some of you may have used the ARM CPU in ECE243. The NIOS II CPU used in this lab works in a very similar fashion. We will be using this CPU in the lab to facilitate integration using Qsys. If you are unfamiliar with NIOS II assembly, you can code using C and compile and load the program using the Monitor program in a similar way to ARM.

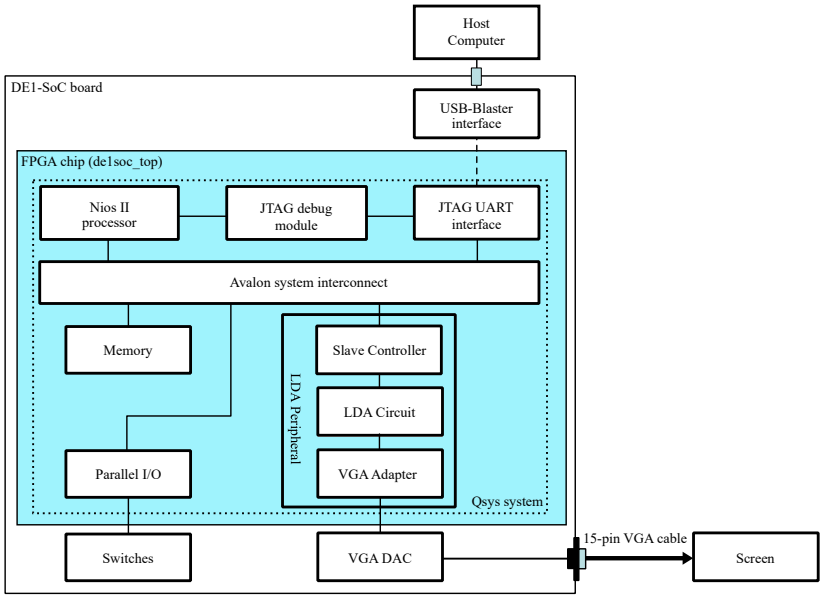

Figure 1: Top-level hardware block diagram for this lab

Part 0: Tutorials

Before doing anything involving the Line Drawer, you will first acquaint yourself with Qsys, and re-acquaint yourself with the Monitor Program and how to write and run programs using it. First, you will need to download the Intel University Program installer from this link. Click on *University Program Installer* and download the latest version.

Then, go to the Tutorials folder in the Course Materials on Blackboard and read and complete the tutorials named *Introduction to the Altera Qsys System Integration Tool* and *Making Qsys Components*.

Part 1: Basic Qsys System

Open the Quartus project given to you in the lab starter kit, launch Qsys, and create a system similar to the one in the *Introduction to the Altera Qsys System Integration Tool* tutorial. It should have the same peripherals as in the tutorial, notably the following:

- A Nios II/e Classic Processor
- On-Chip Memory to hold program code and data (4KB should be sufficient)
- A Parallel IO Peripheral, configured as inputs, to read 8 switches `SW[7:0]`.
- Optionally: another Parallel IO, configured as output, to drive the red LEDs. This is not required, but can be useful for debugging.

After opening the starter project, you can skip everything related to project creation and start from page 6. After that, the only differences from the tutorial are related to Section 5.1 and Section 6. Rather than creating your own top-level module (which they named *lights.v*), use the top-level module from this lab's starter kit. You may then instantiate the Qsys system module there, as described in Section 5.1.1. You can also copy and paste a module instantiation template from within the Qsys GUI (*Generate* → *Show instantiation template*).

You do not have to create a new Quartus project nor import pin assignments as in Section 6 (these are already provided in the starter kit). All you need to do from that section is to add the Qsys-generated `.qip` file to your project. This will add all the auto-generated Verilog code for the system into your Quartus project.

It is recommended at this point to write a quick test program that makes sure you can read the board's switches. Either display them on the LEDs (if you instantiated the second PIO peripheral), or just read them into one of the CPU registers and observe its value by single-stepping in the Monitor Program. The system you have created will be a stepping stone to completing the rest of the lab.

Part 2: LDA Qsys Component

In this part of the lab, you will turn your Line Drawer from Lab 3 into a Qsys Component, in preparation for adding it to your Qsys system from Part 1.

Overview

The LDA component will consist of three modules: Slave controller, LDA circuit and VGA Adapter. A block diagram of the peripheral is provided in Figure 2. The LDA Circuit is the same one as from Lab 3. However, instead of having a UI module, the LDA circuit will be controlled by the Avalon Slave Controller (ASC) that you will design. Your LDA Circuit is supposed to take the following inputs from the ASC: start signal, the starting point of the line (X0, Y0), the end point (X1, Y1) and also the line colour C. Once it completes drawing a line, it is supposed to assert its only output: the done signal, for one clock cycle.

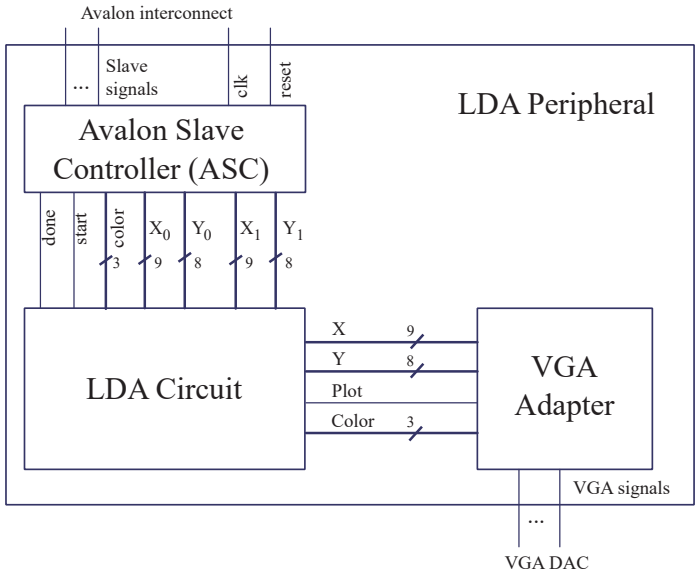

Figure 2: The LDA peripheral diagram

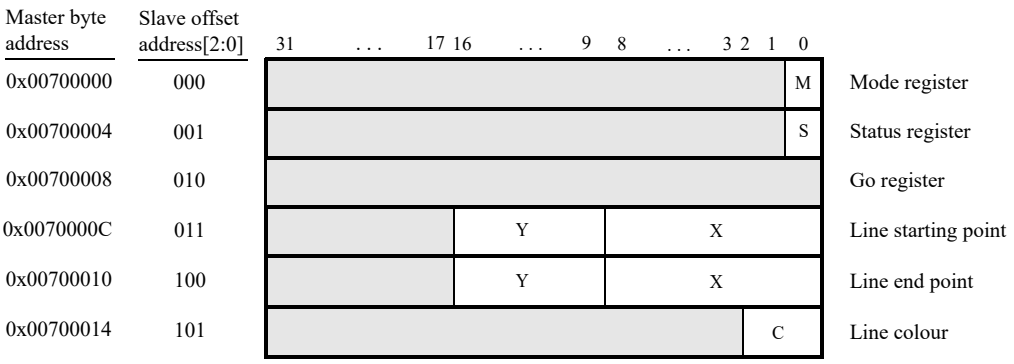

Figure 3: The LDA peripheral memory-mapped registers

The "Avalon interconnect" signals will connect to the rest of the Qsys-generated system using the Avalon-MM protocol, which your ASC will need to understand. Via these signals, an Avalon Master (such as the Nios II) can communicate with your peripheral. These signals include a clock and synchronous reset. The VGA signals from your peripheral do not connect to the Avalon interconnect, rather they connect directly to the FPGA I/O pins and thus to the VGA DAC.

Register Map

Within a Qsys system, all communications between the processor and peripherals look like reads and writes to memory, to addresses well outside the actual range of system RAM. By reading from or writing to a range of addresses associated with a peripheral, code running on the Nios II processor can access different functions of that peripheral via its set of exposed memory-mapped registers.

The memory-mapped register interface of the LDA peripheral is shown in Figure 3, using an example base address of `0x00700000`. Through this interface the processor can instruct the peripheral to draw a single line on the screen. The Line starting point register and the Line end point register store the X and Y coordinates

for the starting point and the end point of the line, respectively. The X coordinate is placed in bits [8:0] and the Y coordinate in bits [16:9]. The colour is stored in the Line colour register, in bits [2:0].

By writing the Go register, the processor initiates the drawing algorithm implemented by the peripheral (the value written is arbitrary, the write transfer itself initiates the peripheral). Note that the processor must set the start/endpoint registers and the color register before writing to the Go register. The mode register determines the operating mode of the peripheral. The two modes define two ways of synchronization between the processor and the peripheral. You will implement two following two modes:

- Stall mode (default) (M=0): In this mode, the processor will stall during the write instruction (stwio) to the Go register. It will be stalled until the LDA peripheral finishes drawing the line. Only then the processor can proceed to the next instruction in the code. In other words, the operation of the LDA peripheral will appear as a long-executing stwio instruction. You will achieve this by using the waitrequest signal (described later). The ASC must keep it asserted (thereby stalling the processor) until the LDA circuit has finished drawing the line. Afterwards, the processor can instruct the peripheral to draw a new line. In this mode the Status register is not used.

- Poll mode (M=1): On the contrary, in this mode, the processor does not stall but is able to immediately proceeds to the next instruction in the code (waitrequest should not be asserted by the ASC). In order for processor to know when the LDA peripheral finishes drawing the line, the processor needs to check bit 0 of the Status register. Bit 0 will have the value 1 if the LDA is currently busy drawing a line, and have the value 0 if it is ready to start drawing a new line. Any writes to the Go register while Status is 1 should be ignored. When later writing your code to draw a line in Poll mode, a good technique is to *first* wait until the Status register contains 0, and *then* to draw the line.

Upon reset of the system the peripheral should be in the stall mode by default. The processor can later change the mode by writing to the M register.

Figure 3 also shows the master byte address and the slave offset address for each register. The master byte address is from the point of view of the Nios II processor and any code running on it. This is because to Avalon masters, 1 unit of address always represents 1 byte, regardless of the actual physical size of the register. The slave offset address is the address that the peripheral sees when it is being accessed. It is a relative, rather than absolute address, from the peripheral's assigned base address within the system. Its units are defined by the peripheral, and can be set to a size most convenient for it. In our case, that means 1 address is 1 entire 32-bit register. Only 3 bits are needed, since there are only 6 such registers.

So, for example, if Nios II wants to write to the Go register, it will present an address of `0x00700008`[1] to the Avalon interconnect. Since this address is within the range of our peripheral, the interconnect will know to direct the write to us and not some other device. The interconnect then strips off most of the bits, and translates the rest into the slave offset address, `010` in binary. The ASC module within the peripheral will then see its `write` signal go high along with the slave address being the value `010`. The exact details of how a read or write arrives at the ASC, and what signals it sees, is covered in the next section.

Avalon Slave Read/Write Transfers

This section describes the "Slave signals" portion of Figure 2, and their purpose and timing. This will allow you to design your ASC.

The shortest duration of Avalon read/write transfers is one cycle. The address and control signals are held constant for the duration of only one cycle, so the slave must respond within that cycle by either presenting valid data (read transfer) or capturing data (write transfer). In this lab all of your transfers will be one cycle, with a single exception: a write to the Go register in Stall Mode will be of variable duration.

Figure 4 shows the examples of one-cycle transfers with no waitrequest signal (CC[n] = n-th clock cycle)

- CC [1] The master asserts address and read on the rising edge of the clock. In the same cycle the slave decodes the signals from the master and presents valid readdata.

[1]The base address `0x00700000` is arbitrary and you can choose a different one in Qsys.

- CC [2] The master captures readdata on the rising edge of the clock and deasserts the address and control signals marking the end of the transfer.
- CC [4] The master asserts address, write, and writedata on the rising edge of the clock. The master signals are held constant and the slave decodes them.
- CC [5] The slave captures writedata on the rising edge of the clock. The master deasserts the address, writedata and control signals marking the end of the transfer.

Figure 5 shows the examples of variable duration transfers. Below is the explanation for the write transfer that is 3 cycles in duration. The slave prolongs the transfer for 2 extra cycles by inserting 2 wait-states using the waitrequest signal.

- CC [1] The master asserts address, write and writedata on the rising edge of the clock. In the same cycle the slave decodes the signals from the master and asserts waitrequest and thereby stalls the transfer.
- CC [2] The master samples waitrequest on the rising edge of the clock. Thus, this cycle becomes a wait-state (or wait cycle), hence the signals address, write and writedata remain constant. In this cycle, CC (2), the slave keeps the waitrequest high, thereby inserting the second wait-state.
- CC [3] The slave deasserts waitrequest on the rising edge of the clock.
- CC [4] The slave captures writedata on the rising edge of the clock. The master samples deasserted waitrequest and ends the transfer by deasserting all signals.

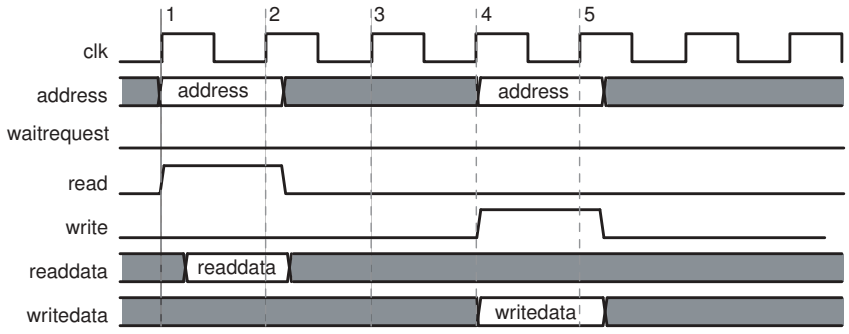

Figure 4: Avalon slave timing diagrams for read/write transfers (without `waitrequest`) - 1 cycle duration

Creating the Component

In the starter kit, there is a folder named `lda/` for containing all the hardware shown in Figure 2. It already has the files for the VGA Adapter, and you should first copy and paste the files you wrote in Lab 3 for the LDA Circuit (do not copy anything related to the UI module). You need to write two more modules in the `lda` folder: the Avalon Slave Controller, and a top-level component module that instantiates and connects everything in Figure 2.

Figure 6 provides a detailed list of all signals for this top-level component module, and you should name your signals as indicated. The 3 sets of signals correspond to the external interfaces at the edges of Figure 2. The names of the signals were chosen so that the Qsys Component Editor can automatically recognize their types, saving you from manually selecting them yourself later in the editor. The names are in the format `<interface type>_<interface name>_<signaltype>`. For example, with the `avs_s1_address` signal, the `avs` prefix represents the Avalon slave interface type, and `s1` is the name of the slave (it will appear in the

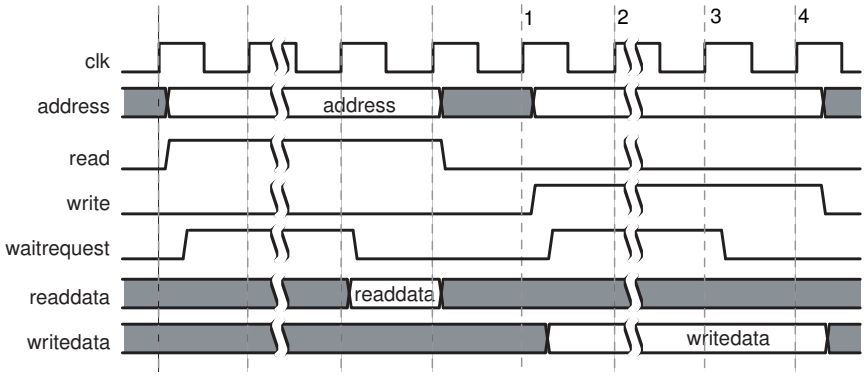

Figure 5: Avalon slave timing diagrams for read/write transfers (with `waitrequest`) - variable duration

Qsys tool), and `address` determines the signal type. The Avalon Slave (`avs`) signals are the ones described in Figure 5 and should be forwarded to your Avalon Slave Controller module and handled appropriately. The VGA signals are to be connected directly to the FPGA I/O pins. We achieve this by assigning them the `coe` interface type (`coe` stands for "conduit end").

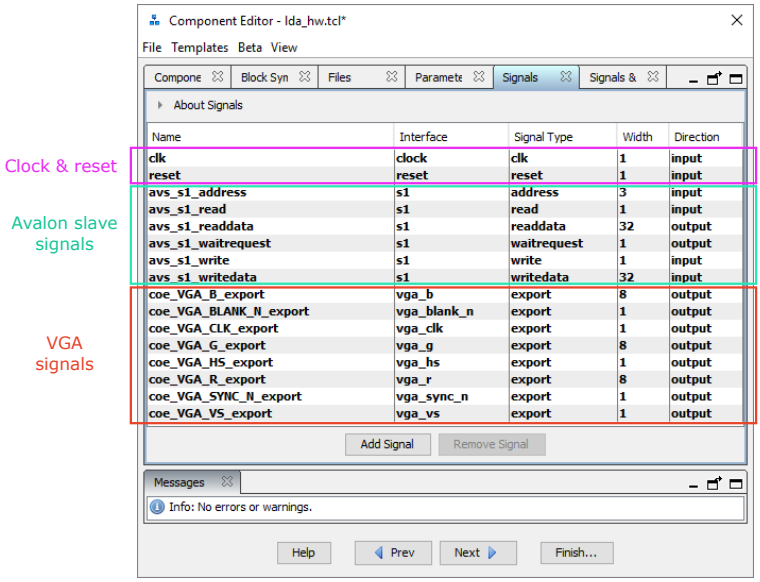

Figure 6: Qsys component for LDA peripheral with a list of input/output signals

After you have finished writing the ASC module and top-level LDA Component module, open your Qsys system from Part 1 and click *New component*. Name your component something reasonable. In the Files tab, click *Add File...* and add all the files in the `lda` folder. Find your top-level component file from the list and click its row in the Attributes column to set it as the Top Level File. Then click *Analyze Synthesis Files*. The editor is quite unforgiving of syntax errors and won't tell you if/where they happened. The easiest way to fix your files is to run the `vlog` command in ModelSim on each file.

Next, there will likely be an error in the Messages window. You will need to go to the *Signals & Interfaces* tab to set the `s1` interface's associated reset interface to `reset`. Also, make sure that the signals have been automatically recognized and match Figure 6. Note that no other settings in the component editor have to

be changed, so leave default values. Once you have completed the component, save it and exit the editor.

Part 3: Qsys System with LDA

Finally, you will incorporate your LDA Component into your system, and instantiate the Qsys-generated files for the systems in your FPGA top-level `de1soc_top` module, resulting in the system shown in Figure 1.

Your new component will appear on top of the component list in Qsys. Add it to the system from Part 1, assign its base address and connect it to the data master of the Nios II processor. Connect the clock and reset signals similarly to the other existing peripherals. You will need to *Double click to export* all the VGA signals from the component instance.

Re-generate the Verilog files for the Qsys system. Before you compile the your Quartus project, you have to edit `de1soc_top.sv` file to connect the exported VGA signals from the system module to the inputs and outputs of the top module. The instantiation template in Qsys (*Generate* → *Show instantiation template*) will help you with this.

After you compile the Quartus project, you are ready to test the operation of the entire system on the DE1-SoC board. Try drawing a single line in Poll or Stall mode by writing and running a small assembly language program with the Monitor Program. Note that if you need to change/fix any of your LDA component's Verilog/SystemVerilog files, you will need to re-generate the Qsys system each time.

Part 4: Animation

With the hardware now complete, you will write some interesting software to run on your system's Nios II processor and exercise your LDA component. To do this, you will write a C (or assembly) language program that animates a horizontal line that moves vertically across the screen. The direction of motion (up or down) will be selected with one of the switches, which your code can read from the associated Parallel I/O peripheral. The line colour, and drawing mode (Poll Mode vs. Stall Mode) should also be selectable using more of the switches. Once the line has reached the top (or bottom) of the screen, it should wrap around to the other end and continue moving. Animation using your LDA Peripheral is achieved by repeatedly:

1. Drawing the line at its current position.
2. Erasing the line at its previous position, by drawing a black line with `colour=000`.
3. Waiting for one frame's worth of time (eg. 1/30th of a second), which can be done with a large empty delay loop in your code.
4. Updating the line's current/old positions and going back to Step 1

The LDA is accessed in your C/assembly code by reading and writing to its memory-mapped register set starting at the base address you chose for the component in Qsys. The registers will be laid out according to Figure 3.

Part 5: Better Animation (BONUS)

Instead of a horizontal line that moves vertically, animate a *rotating* line of length 20 pixels, with one endpoint anchored at the center of the screen $(168, 105)$. The other endpoint rotates clockwise or counterclockwise around the center depending on the position of a switch on the board. Colour and LDA Mode (Stall vs. Poll) are also switch-selectable as in Part 3. The position of the moving endpoint of the line, as a function of angle θ, is:

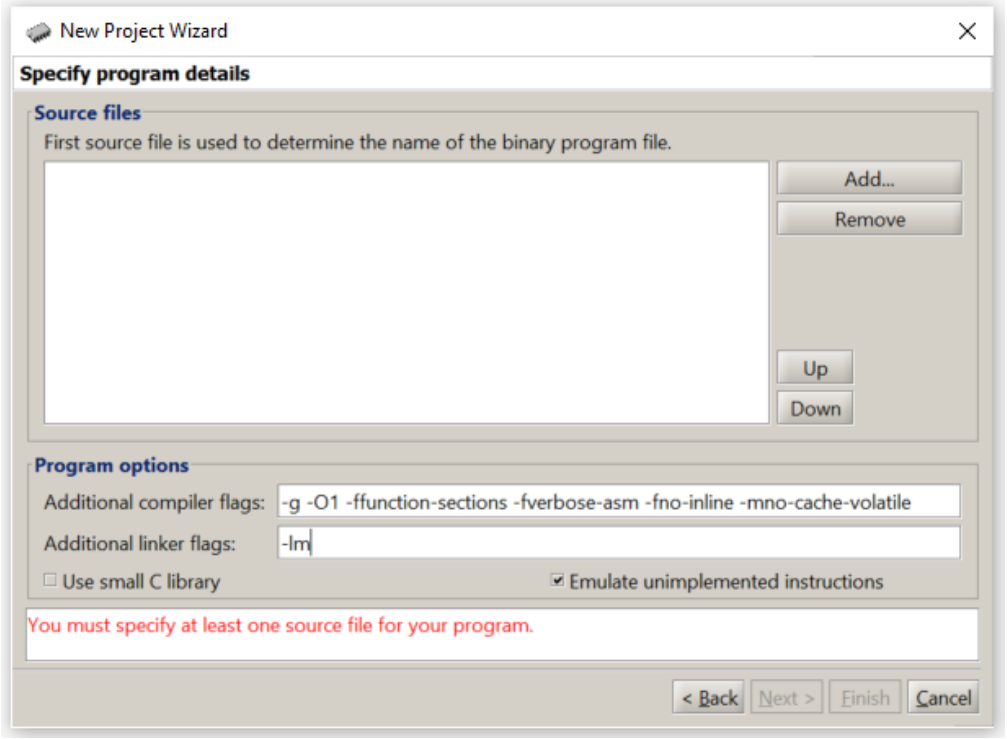

Figure 7: Adding `-lm` flag to instruct the linker to include the math library in your program

$$x = 20\cos(\theta) + 168$$
$$y = 20\sin(\theta) + 105$$

You will need to write your program in C to be able to use the *sin*/*cos* trigonometric functions in `<math.h>`. You will also need to add `-lm` to the Additional Linker Flags field in the Monitor Program, as shown in Figure 7. This extra library code will require you to increase the size of your on-chip RAM to more than 4KB, and to upgrade your processor to a Nios II/s in order to have native `mul` instructions.

NOTE: When upgrading your processor in Qsys to one that is better than a Nios II/e, the lab computers (which run 18.0 Standard Edition) will behave slightly differently when programming the FPGA. This is because the better processors require a license to use in an unrestricted way. First, Quartus will generate a de1soc_time_limited.sof file, instead of just de1soc.soc. This needs to be reflected in your Monitor Program project settings. Next, when you program the FPGA (using either the Programmer or Download System from the Monitor Program), a window will pop up with the title "Open cores plus status". **Do not close it or press Disconnect**. You can still use the Monitor Program with it open. If you get an error doing Download System, close the Monitor Program, power-cycle your board, and try again.

Marking Scheme

Have the following prepared and ready for inspection by your TA at the beginning of the lab:

- Verilog/SystemVerilog code for your Avalon Slave Controller, top-level LDA Component module, and top-level `de1soc_top` module (2 marks)

- Qsys system containing your LDA Component (1 mark)
- C or assembly source code for your animated vertically-moving horizontal line program (2 marks)
- **Bonus**: Source code for rotating line program (+2 marks, all or nothing)

Demonstrate the following to your TA in the lab. You will receive 5 marks for a fully-working Part 4, as it implies the ability to perform the first two milestones as well. Partial marks will be awarded for demonstrating only some of the milestones (eg. 3 marks for being able to draw non-animated lines under the control of switches).

- Ability to read the switches (1 mark)
- Drawing a single, or multiple, static non-animated lines (2 marks)
- A fully-working animating line as described in Part 4, in both Stall and Poll modes (2 marks)
- **Bonus:** A rotating line as described in Part 5 (+2 marks, all or nothing)

Frequently Asked Questions

In this section, questions from previous years have been collected and answered. Please check here first before posting a question on piazza.

Q1: Where do the byteenable and chipselect registers get created, since they are not part of the ASC we create?

A1: Qsys generates these for you when you generate the top-level design file. Later in Lab 6, you will add in a byteenable to your ASC to allow the processor to write 8, 16 or 32-bit values to your LDA.

Q2: When recompiling the top-level file containing the Qsys project, do we need to recompile the C code as well?

A2: No. But remember to reprogram the FPGA using either the Quartus Programmer or the Download System menu item in the Monitor Program.