Lab 6: Systems on a Chip

Introduction

In the previous lab, you designed your own processor and verified its functionality in a simulator. In this lab, you will implement this processor on the DE1-SoC board.

A processor is only useful if it can interact with the outside world. In addition to memory (which contains the program code and working RAM), you will be connecting your processor to various I/O devices. In the final part of this lab, you will be able to control your Line Drawer peripheral using your custom processor.

Part I: Simple I/O

Your first hardware implementation of your processor will be a system containing four components:

Component	**Address Range (bytes)**
The processor itself	
4KB memory	`0x0000 - 0x0FFF`
Slider switches x8	`0x2000 - 0x2FFF`
Red LEDs x8	`0x3000 - 0x3FFF`

Table 1: System Address Map

The processor will then run a program that forever reads the switches and copies their values to the LEDs. In the past, you have built such a system using a system-building tool like Qsys by simply selecting the components from a library, defining the connections, and specifying which addresses each peripheral was mapped to. In this part of the lab, you will have a chance to better understand and appreciate what happens 'under the hood' by connecting all the above components yourself and creating all the necessary address decoding hardware.

Figure 1 shows a block diagram of the system and its connections. The processor can write to two devices (memory, and a register feeding the 8 red LEDs) and can read from two devices (memory, and a register capturing the 8 switches). This is decided based on the address emitted by the processor, and is performed by the blocks marked "*decoding logic*" in the figure, which you will need to create.

All clock inputs should be connected to the DE1-SoC's `CLOCK_50` signal. You may use `KEY[0]` as a reset. Note that the processor's read enable signal `o_mem_rd` is unused in this part of the lab. However, the processor still expects read data to arrive **one cycle after** the address is issued (just like in the previous lab). This means that the decoding logic that feeds the processor's `i_mem_rddata` input will need an internal one-cycle delay (as hinted in the figure by a clock input).

We provide a memory block in the lab starter kit in a file called `mem4k.sv`. Its contents are initialized from a file called `mem.hex` containing the processor's program code and data. Be aware that the address generated by the processor is in units of **bytes** (with the lowest bit always 0) but the memory block accepts an address whose units are **16-bit words**. For example, address `0x800` sent by the processor accesses the word with address `0x400` in the RAM.

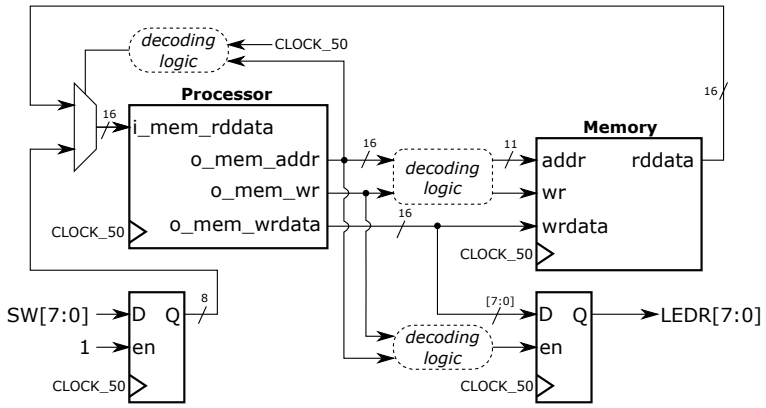

Figure 1: Part I Block Diagram

Instructions

1. Use the `part1` folder in the starter kit and the Quartus project inside. Copy all the source files for your processor from Lab 5 into this folder.
2. Implement the rest of the system shown in Figure 1, and using the memory map given in Table 1. There are a lot of addresses assigned to the LED and switch peripherals, to simplify address decoding. The easiest and recommended approach is to, for example, treat all addresses `0x2000-0x2FFF` as mapping to the same 16-bit word, and have each switch be one bit inside this word. Same for the LEDs.
3. Write a program for your processor that continuously reads the eight switches and displays their state on the eight LEDs. Compile your program with the assembler provided in Lab 5 and rename the generated file to `mem.hex`. Place it with the rest of your Quartus project files.
4. Compile your project with Quartus Prime and demonstrate it to your TA on the DE1-SoC board in the lab. Note that changes to your `mem.hex` file require a recompilation of the Quartus project.

Part II: Avalon Master

In this part of the lab, you will return to using Qsys to build your system, by making your processor into a component. In Lab 4, you gained experience creating a custom Qsys component for your Line Drawer peripheral. The difference here is that the Line Drawer had a *slave* interface which only *responds* to read and write requests, while your processor will need to have a *master* interface which *generates* those read/write requests. The end goal of this part of the lab will be to create a Qsys system containing your new processor component and have it run a program to interact with I/O devices.

Project Setup

You will work with the starter project located in the `part2` folder in the starter kit. Copy your processor source files into here, or into a descriptively-named subfolder like "`cpu`". There is no need to add these files via *Add/Remove Files in Project* within Quartus.

Modifying the Processor

Your processor's external signals must conform to the Avalon-MM interface specification so that it can participate in a Qsys system as a Master. All of the *existing* signals already conform to this specification, but two more signals will need to be added. Previously, your processor has always assumed two things:

- When it sends a read or write request to memory or an I/O device, that request will always be accepted in the same cycle it was issued.
- Read data will always be returned one cycle after the read request is accepted by the target.

These assumptions were easily satisfied because the devices, as well as the interconnect between the processor and devices, were designed by you. However, in general, a processor may need to interact with devices made by third parties and connected using interconnect whose latency is not known ahead of time. The two signals you need to add to your processor will allow it to relax these assumptions:

Signal	Direction	Width	Description
`i_mem_wait`	input	1	Avalon *waitrequest* signal
`i_mem_rddatavalid`	input	1	Tells your processor when read data has returned

When issuing a read or write request, your processor must now continue to retry the request until `i_mem_wait` is 0. This could happen in the very same cycle the read or write is issued, or it could immediately go to 1 and only become 0 many cycles later. This was the same signal that your LDA peripheral used in Stall Mode in Lab 4, except there, it was an output. Now, it is an input.

After successfully issuing a read request, your processor must now wait until `i_mem_rddatavalid` is 1 to indicate that the data has actually returned. You can assume that this will happen at least 1 cycle after the request was made. Your CPU needs to be modified to use this signal to complete instruction fetches and `ld` instructions. Click here if you'd like more details about the new Avalon signals.

Creating the Qsys Component

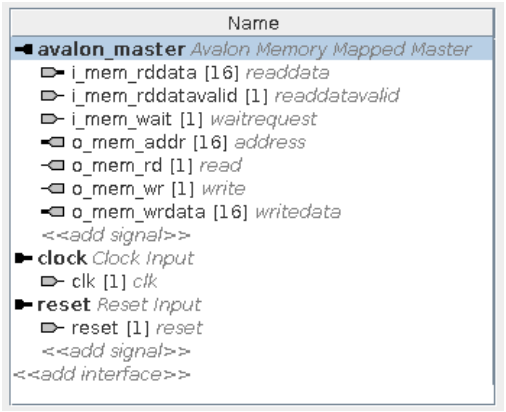

Figure 2: Signal Roles for the Processor Component

This process will be similar to what you did with the Line Drawer peripheral:

1. Launch Qsys and start the *New Component...* wizard.
2. In the first tab, name your component something sensible like `cpu`.
3. In the Files tab, browse and add your processor's source files. Set the top-level *file* by clicking in in the Attributes column. Click 'Analyze Synthesis Files'. Then set the top-level *module*.
4. In the Signals & Interfaces tab, configure the interfaces according to Figure 2. Make sure to associate the clock and reset interfaces with the Avalon Master interface. Note that unlike Lab 4, you need to set the signal roles explicitly, because the signal names do not conform to the special naming scheme.

5. Still in the Signals & Interfaces tab, observe the example waveforms for the Avalon Master interface to make sure that you understood and implemented the timing of the new processor signals correctly.

6. Click Finish to generate the component.

Building the System

With Qsys still open, add your newly-defined processor component to the system. Also add the following components from the library. You can use the search box to help you find them:

- **On-Chip Memory (RAM or ROM):** Set the slave data width to 16, and the total size to 4096 bytes. Check "Enable non-default initialization file" and change the filename to `onchip_mem.mif`. You will create this file later.
- **PIO (Parallel I/O):** This will be connected to the board switches. Set the width to 10 bits and the direction to Input.
- **PIO (Parallel I/O) #2:** This will be connected to the red LEDs. Set the width to 10 bits and the direction to Output.
- **Quad HEX Decoder:** Will drive four of the HEX displays with the word that's written to it. Comes with the starter kit.

Connect each component's clock and reset inputs to the initial "clock source" component that existed when you created the system. All Avalon slave ports get connected to the processor's Avalon master port. The two parallel ports and the Quad HEX Decoder have conduit interfaces that need to be exported (*Double-click to export* in the system editor). Assign base addresses to all components, using whatever values you wish. However, the memory must start at `0x0000` because the processor starts reading instructions from that address upon reset.

Click *Generate HDL* to make Qsys create the system. Copy and paste the code from *Generate* → *Show Instantiation Template* into the `de1soc_top.sv` file in the Quartus project. This instantiates the Qsys system module. Connect its signals to the appropriate FPGA pins and the `reset_n` signal. Add the `.qip` file that Qsys generated into your Quartus project through the *Project* → *Add/Remove Files in Project* menu. The Quartus project is almost ready for compilation – it's just missing the `.mif` file that defines your software code.

Assembly Program

Write a program for your processor that increments a 16-bit counter and displays its contents on four of the hex displays via the Quad HEX Decoder peripheral. The counter will need to increment at a slow enough speed such that you can actually see it change on the board. You can write an empty delay loop in software, or add an Interval Timer to your Qsys system if you are feeling adventurous.

The speed of the counter will be controlled by the switches. Whether more switches means more speed, or less speed, is up to you. The only requirements are that there exist at least two non-zero counting speeds, that they are visibly distinct, and that the switches are somehow used to select between them. Use the red LEDs to represent the current speed. The simplest solution is to just mirror the value of the switches, as in Part I.

Use the assembler provided in Lab 5 to compile your program. It generates both a `.mif` and a `.hex` file. We only need the `.mif` file, and it must be renamed to match the filename entered in the On-Chip Memory component in Qsys, which was suggested to be called `onchip_mem.mif`. Place the MIF file into the same folder as your Quartus project. You can now compile the Quartus project and test your system.

Unlike Part I, there is a way you can avoid having to recompile the entire Quartus project when changing your source code. After re-generating the MIF file from the assembler, use *Processing* → *Update Memory Initialization File*, and then *Processing* → *Start* → *Start Assembler*. You can then re-program the FPGA.

Part III: Adding the Line Drawer

In the final part of this lab, you will add your LDA component from Lab 4 to your Qsys system from Part II of this lab, and write a simpler version of the animated line program for your processor.

The Avalon Slave Controller within your LDA component will need modification in order to be able to be accessed from your 16-bit processor. The start/end registers that hold the X0/Y0 and X1/Y1 coordinates are larger than 16 bits. This was not a problem when they were accessed from a Nios II processor, since it can write up to 32 bits at once. However, your processor would need two 16-bit writes to access each register, and this is currently not supported by your component.

To see why, let's use an example: to set the start coordinates to $(256, 192)$, we would need to store the 17-bit value `0x18100` to the Line Start register. With a Nios II, we could do this by directly writing the 32-bit value `(0x00018100)` to offset `0xC` of the LDA. In order to accomplish this as two 16-bit writes using your processor instead, it would have to write `0x8100` to offset `0xC` and `0x0001` to offset `0xE`. However, recall that within your ASC, the incoming `avs_s1_address` signal is only 3 bits wide and can't differentiate between two halves of a 32-bit register – the processor's attempts to access `0xC` and `0xE` **both** appear as address `011` within the ASC.

To solve this, your LDA component and ASC will accept a new signal from the Avalon interconnect: a *byte enable* signal named `avs_s1_byteenable` that is 4 bits wide and is an input. Each of the 4 bits in this new signal represents one of the 4 bytes in the existing 32-bit `avs_s1_writedata` signal. If a bit is 1, then the processor wishes to write to that corresponding byte.

Now, when the processor writes the value `0x8100` to offset `0xC` of your LDA, the ASC will see `avs_s1_address = 011`, `avs_s1_byteenable = 0011` and `avs_s1_writedata = 0x00008100`. When the processor writes `0x0001` to offset `0xE`, the ASC will see `avs_s1_address = 011`, `avs_s1_byteenable = 1100` and `avs_s1_writedata = 0x00010000`. Notice how the Qsys interconnect automatically positions the 16-bit value within the 32-bit writedata signal for you when accessing offset `0xE`. This means you do not need to change any existing data connections from the Avalon bus to your registers – bit 16 of the incoming writedata still connects to bit 16 of your start coordinates register and end coordinates register.

Instructions

1. Make a copy of your completed Part II as a starting point for Part III. Copy all the files for your LDA component from Lab 4, including the `_hw.tcl` file that the Qsys component editor generated.

2. Modify the Avalon Slave Controller within your Line Drawer peripheral, as well as the component's top-level module, to accept a 4-bit `byteenable` signal as an input from the Avalon bus, and make use of it as described above.

3. In Qsys, right click and Edit your LDA component definition to updated it to include the new byteenable signal. You can click *Analyze Files* to do this. Make sure your new signal appears in Signals & Interfaces and has type "byteenable".

4. Insert the updated LDA component into your Qsys system. Assign it a base address, and export the VGA signals as you did in Lab 4.

5. Regenerate the Qsys system, and update the instantiation in `de1soc_top.sv` to include the exported VGA signals.

6. Write a program for your processor that animates a vertical line that moves horizontally from left to right. The height of the line should be the full height of the screen (Y=0 to Y=209) and its X coordinate traverses the full width of the screen (X=0 to X=335). After reaching X=335, the line wraps around to X=0 again. Use the same technique as Lab 4 for animating the line, by drawing a black line at the previous location to erase it. Lines only need to be drawn in Stall Mode. Your program should read the switches to choose the colour of the line.

Marking Scheme

This is a two-week lab. Have the following completed for the first lab session:

- Preparation: Verilog/SystemVerilog code for Part I (2 marks)
- Preparation: Assembly source code for Part I (1 mark)
- In-lab: Demonstrate the working Part I system on the DE1-SoC board (3 marks)

Have the rest of the lab completed for the second week:

- Preparation: Verilog/SystemVerilog code for modified processor, Part II (1 mark)
- Preparation: Qsys component for processor (1 mark)
- Preparation: Assembly source code for Part II (3 marks)
- Preparation: Verilog/SystemVerilog code for modified ASC for Part III (2 marks)
- Preparation: Assembly source code for Part III (3 marks)
- In-lab: Demonstrate Part II (5 marks)
- In-lab: Demonstrate Part III (5 marks)