Lab 6: Systems on a Chip

Introduction

In lab 4, you built a system connecting the NIOS II CPU with a floating point core. At the time, you used QSys to provide you with the needed connection and logic to allow for communication between primary and secondary elements of your system. Then in lab 5, you built your own custom CPU. In this lab, without the use of QSys, you will build on what you did in labs 4 and 5 by connecting your own CPU with the floating point core. This is to say that you will design the interconnect and logic to put your system together. To start with, you will connect your own CPU with some simple peripherals such as RAM, switches and LEDs. Then you will turn your CPU into an Avalon Master (similar to what you did in lab 4 to convert the floating point multiplier into an Avalon slave). And finally, you will control the Floating Point Multiplier you made in lab4 using your custom processor.

Part I: Simple I/O

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

Component	Address Range (bytes)
The processor itself	
A dual port 32KB memory	`0x0000 - 0x7FFF`
Red LEDs x8	`0xA000 - 0xA00F`
Slider switches x8	`0xA010 - 0xA01F`

Table 1: System Address Map

The processor will utilize the two ports of the RAM as separate instruction and data interfaces. It will also need to be connected to the switches and LEDs. The processor must run a program that forever reads the switches and copies their values to the LEDs. In Lab 4, you 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, omitting all the clock signals and the processor instruction interface. 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*" in the figure, which you will need to create.

Note that 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.

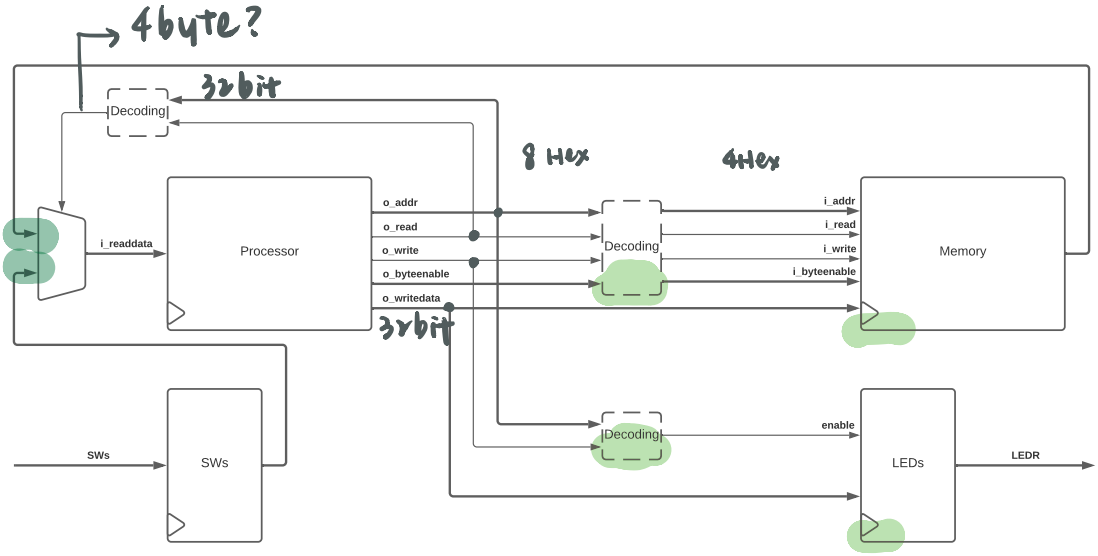

Figure 1: Part I Block Diagram

We provide a memory block in the lab starter kit in a file called `mem.sv`. Its contents are initialized from a file called `part1.hex` which should contain the processor's program code and data. Note that the address generated by the processor is in units of **bytes** while the memory block expects the address in units of **32-bit words**.

Instructions

1. Use the starter kit for `part1` and copy the source file 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. Note that while there are address ranges assigned to the LEDs and switches, only the first address of each is actually used to read from or write to the switches and LEDs.
3. Write a program for your processor that continuously reads the switches and display their state on the LEDs. Compile your program with the assembler provided in Lab 5 and rename the generated file to `part1.hex`.
4. Test your system rigorously using Modelsim, make sure you debug and fix all issues and pass the provided tester. Note that any changes to your `part1.hex` file requires a recompilation in Modelsim.
5. Combine all your verilog in one file called `part1.sv` that you can then submit alongside your `part1.hex`

Part II: Avalon Master

In this part of the lab, you will need to modify your processor to make it compliant to the Avalon bus specifications as an Avalon master, which you will then use to control the FP multiplier you made in Lab 4.

Modifying the Processor

Your processor's external signals must conform to the Avalon interface specification so that it can access the bus as a Master. All of the *existing* signals in your custom CPU already conform to this specification, but one more signal will need to be added. Previously, your processor has always assumed that 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. This assumptions was easily satisfied because the devices, as well as the bus 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 a bus whose latency is not known ahead of time. You will need to add a `i_ldst_waitrequest` signal to your processor to allow it to relax this assumption.

When issuing a read or write request, your processor must now continue to retry the request until `i_ldst_waitrequest` 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 FP multiplier peripheral used in Lab 4, except there, it was an output, where you signaled to the avalon master (the NIOS II CPU in lab 4) to wait till the floating point operation was completed. In this lab, the waitrequest is an input that you CPU must wait for before continuing. Note that this change needs to **ONLY** happen to the data interface of the processor.

Instructions

1. Use the starter kit for `part2`. Copy all the source files for your processor from Lab 5 into this folder.
2. Modify your processor to utilize the `waitrequest` signal and be able to stall until memory requests are finished.
3. Test your processor rigorously using Modelsim, make sure you debug and fix all issues and pass the provided tester. Note that merely passing this tester does not guarantee your processor can wait correctly.
4. Combine all the CPU files in one file, and give it the name `part2.sv` before submitting it.

Part III: Building the System

With your processor now modified and ready, you will build a new system containing three components:

Component	Address Range (bytes)
The processor itself	
A dual port 32KB memory	`0x0000 - 0x7FFF`
FP Multiplier	`0xA000 - 0xA020`

Table 2: System Address Map

Similar to the `part I`, the processor will utilize the two ports of the RAM as separate instruction and data interfaces. It will also need to be connected to the floating point multiplier. The processor will then run a program that forever reads the memory words **8189** and **8190**, use them as multiplication operands, and then writes the result back into a third register **8191**. Note that only the multiplier uses the `waitrequest` signal while the RAM does not.

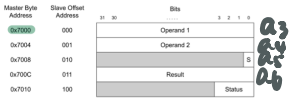

Figure 15: The FP multiplier memory-mapped registers

Instructions

1. Use the same folder and files you used for `Part II`. Copy all the source files for the FP multiplier from Lab 4 into this folder.
2. Build your system using the processor, the RAM, and the FP multiplier, and using the memory map given in Table 2.
3. Make sure your ram is instantiated using the name `mem_inst`. So your `part3.sv` should include the line `mem mem_inst`. This is necessary for the tester and marker to read and write test values in your RAM.

4. Write a program for your processor that continuously reads the memory words `8189` and `8190`, uses their value for multiplication, and writes the result back in memory word `8191`. Compile your program with the assembler provided in Lab 5 and rename the generated file to `part3.hex`. Place it with the rest of your project files.
5. Test your system rigorously using Modelsim, make sure you debug and fix all issues and pass the provided tester. Note that any changes to your `part3.hex` file requires a recompilation in Modelsim.
6. Combine all your designs, including the modified processor of `Part II` in one file called `part3.sv` that you can then submit alongside your `part3.hex`.

QSys vs Handmade

You have likely noticed that in this lab what you were required to do is the same of what QSys did for you in a previous lab; all the decoding and glue logic you used for parts I and II is equivalent to what QSys would build if it were used to implement the same systems.

When generating QSys based systems using the Avalon bus, QSys implicitly creates the bus interconnects for you. These bus interconnects are not just plain wires, they also include some decoding logic, allowing each slave to get its own `avs_read` and `avs_write` signals, to get slave specific addresses (like the word addresses you used for the FP multiplier in lab 4) even when the system uses byte addresses, etc. This is exactly the same as what you were required to do for this lab.

Submission

For Part 1, you need to combine all your modules into one file named `part1.sv` which you will then submit alongside your `part1.hex`. For part 2 you just need to submit the `part2.sv` file. Finally, for part 3 you will need to combine all modules (including the files you used for part 2) into file `part3.sv` which you will submit in addition to `part3.hex`. For example:

submitece342s 6 part1.sv part1.hex part2.sv part3.sv part3.hex