Lab 3: Instruction Memory, Register File, and Data Memory

Course: COEN 122L
TA: Brandon Quant (bquant@scu.edu); Andrei Negulescu (anegulescu@scu.edu)
TA Office Hours: Brandon - Tuesday, Wednesday 1:15pm - 2:15pm; Andrei - Thursday 1:15pm - 2:15pm

Description

The final pipeline that you will design will require three main memory components. The instruction memory will hold the actual binary instructions that the pipeline will run. The program counter (PC) will address the instruction memory, meaning that the PC will determine the current instruction to be loaded into the pipeline. The register file holds the register memory for the pipeline. It contains 64 available registers each with a width of 32 bits. The final memory component is the data memory, which tracks the data in the load/store instructions for the pipeline.

Assignment

In this lab, you will design three memory modules using the SystemVerilog language. You may use dataflow and behavioral syntax for this assignment. You will write both the module description and a testbench to verify that your circuit functions correctly.

To receive full credit you will need to demo your working code and turn in copies of your source code and testbench.

Approach

In this lab we introduce a new style of circuit called the sequential circuit. The primary difference between combinational and sequential circuits is the concept of timing. Combinational circuits assume that the circuit's functionality happens "instantly", meaning that a logical change in the inputs immediately becomes reflected in the outputs. In sequential circuits, this is not always the case. Sequential circuits have a component commonly referred to as the clock, a signal shared across all the components of a module. The behavior of the clock signal dictates the updating/processing of data throughout the given circuit. Usually the clock signal has a constant behavior, switching back and forth between high and low at regular intervals. One sequence of the clock from high to low back to high again is called a cycle.

In this lab we will implement each of the memory modules as a sequential circuit, meaning that the circuits' functions (in this case reads and writes) will only occur once every clock cycle. In order to do this we need to know a few more things about SystemVerilog. First is the concept of registers. In addition to the wire type that we have been using, SystemVerilog allows us to declare and use register types with the reg keyword. Declaring registers is the same as declaring wires (e.g. reg [3:0] myReg). While wires in SystemVerilog represent the physical wire interconnects in a circuit, registers represent flip-flops, which is why we use them in sequential circuit design. The purpose of the register (flip- flop) is to maintain or hold a certain signal (high or low) until the next clock cycle when it will update.

The second part of designing sequential circuits is understanding how to set up the clock driver. In the module description, we do this by using an always block. The always block allows us to define an action or event that will trigger some action by the circuit. The syntax is:

$$always\ @(mySignal)\ begin$$
.
.
.
$$end$$

The mySignal object may be preceded by either the posedge or negedge keywords that define which type of signal transition (low to high or high to low) triggers the action. If neither specifier is given then any value change of the signal will drive the block action. For this project I recommend using the positive edge of the clock signal.

The last part of the design is determining what goes inside the always block. The behavioral syntax in SystemVerilog allows us to assign values to registers using simple assignment (e.g. reg1 = reg2;). You may also make use of if statements to check the state of certain signals before taking an action. For example, some of these memory modules will have a write signal that controls whether or not the input data is written to the input address. To check the value of write and perform a write operation if it is high you can use an if statement.

Instruction Memory

The instruction memory should hold 32-bit values for the pipeline instructions. You may choose to make the memory byte-addressable or word addressable. You may choose the number of instructions that your module holds; I recommend using a depth of 256.

Register File

The register file should hold 32-bit values for the registers used in the pipeline. It should have a depth of 64. It will take three 6 bit addresses as inputs and output the values stored in two of them (the rs and rt addresses) as well as a 32-bit data input and a write signal. If the write signal is high then the data input should be written to the third address input (which is rd).

Data Memory

Looking at the behavior of the load/store instructions, the data memory should support 32 bit addressing. However, that's a lot of addresses for the simulator to handle. To avoid crashing the simulator we need to restrict the range of addresses the data memory will use. The way I recommend doing this is to use 16 bit addressing instead. Leave the input address signal as 32 bits, but when you use it to read/write only use the lower 16 bits which you can access by writing address[15:0]. This will generate a reasonable amount of memory that the simulator can handle just keep in mind that when you generate and test assembly code on the final pipeline the memory addresses stay in the range 0 to 65536. Similar to the register file the data memory will also have a write signal that triggers a write of the input data to the input address.