EE 457 Lab 3 Pipelined Processor

1 Introduction

You will apply your knowledge of the basic 5-stage pipeline by converting a single-cycle CPU datapath and control to a 5-stage pipelined implementation and add various additional units, including a Hazard Detection Unit and Flushing Unit.

2 What you will learn

This lab will help you:

- Understand the detailed implementation of a 5-stage pipelined processor
- Understand internal forwarding in the register file
- Understand the inputs and output control associated with the Hazard Detection Unit
- Further develop your Verilog description skills

3 Background Information and Notes

Refer to the Single-Cycle CPU Lecture Notes and related Pipelining Lecture Notes. We will provide you with a working single-cycle CPU datapath and control unit. Because we are simulating memory, Be sure to fully understand

Lab Organization: You will be implementing your CPU using Verilog to describe it and Modelsim to simulate it.

System Inputs/Outputs: The inputs to your CPU are 'clk', 'rst' (active-high), and the memory interface signals. We will also bring out the interface signals to the register file so that they can be easily viewed. Any other signals can be viewed in the Modelsim Simulator by drilling down in the Objects Pane to find the internal signals in the UUT (Unit-under-Test = CPU). The provided testbench will instantiate your CPU design and a memory model and connect them together. This is depicted in the Figure 1 below.

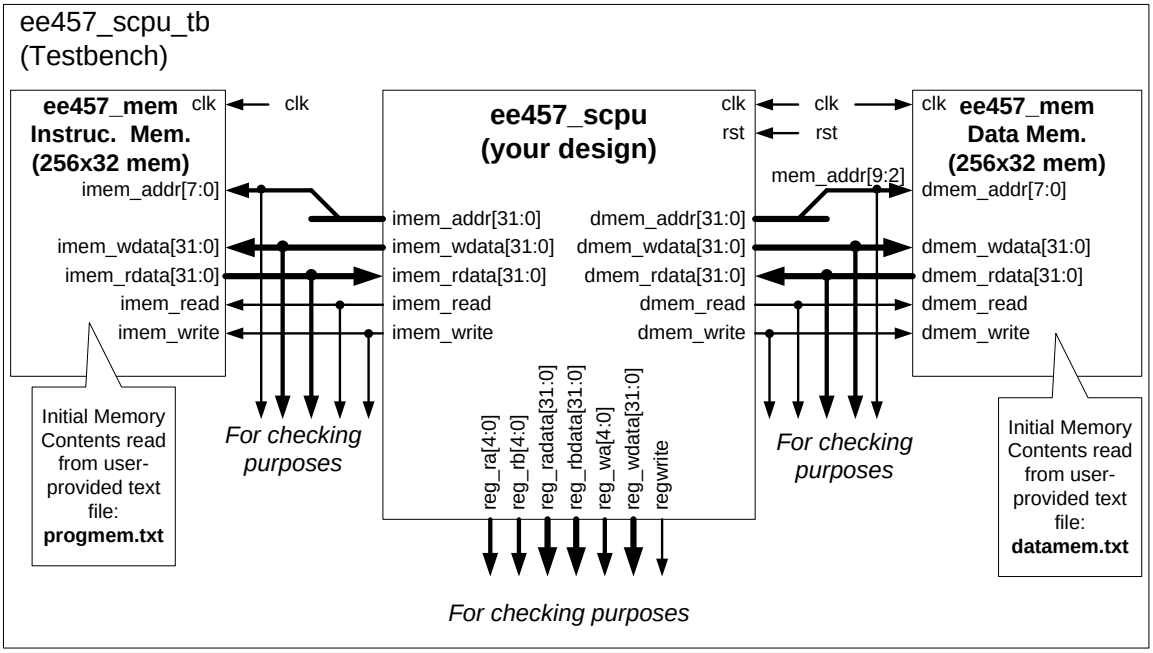

Figure 1 - System Block Diagram

Provided Components: We have implemented (either fully or partially) several components for you to make the task more manageable. These have been instantiated in the provided CPU skeleton design.

ee457_regfile_2r1w.v: Implements a 2 read-port, 1 write-port, 32x32 register file. The initial register file DOES NOT IMPLEMENT INTERNAL FORWARDING. The I/O is defined below. It is complete and need not be modified by you.

Signal	I/O	Description
ra[4:0], rb[4:0]	Input	Read register select port a and port b
radata[31:0], rbdata[31:0]	Output	Read data from selected registers (from ra and rb)
wa[4:0]	Input	Write register select
wdata	Input	Data to write to selected register
regwrite	Input	Write enable for register selected by wa.

ee457_alu.v: This file is complete and takes in a 6-bit function code to indicate the operation. The function code is shown below and matches the function codes used by R-Type instructions in the MIPS ISA.

Operation	FUNC[5:0]	RES[31:0]	UOV (unsigned overflow)	SOV (signed overflow)
ADD	010000	OPA + OPB	*	*
SUB	010010	OPA – OPB	*	*
AND	010100	OPA & OPB	0	0
OR	010101	OPA | OPB	0	0
XOR	010110	OPA ^ OPB	0	0
NOR	010111	~(OPA | OPB)	0	0
SLT	011010	1 if OPA < OPB (signed), 0 otherwise	*	*

'*' = output as traditionally defined

One important aspect of this project is the size of the memory. While your processor will implement a byte-addressable 32-bit address, we will only interface it to much smaller 256x32 instruction and data memories (i.e. 256 words) to limit the amount of initialization and make simulation faster. To this end, while your processor will generate mem_addr[31:0], we will only connect 8 bits of the address to the memory (since it has 256 locations). Those 8-bits will be mem_addr[9:2]. mem_addr[1:0] are unneeded since we will always do word accesses. [In an actual design these bits would be converted to "byte enable" signals that would select the appropriate bytes from the word.]

Testbench and Initial Memory Contents: A Verilog testbench has been provided for you that will generate the rst signal and hold it active for the first few clock cycles. It also generates the clock signal. You will use Modelsim simulator to simulate your CPU fetching and executing instructions from instruction memory. This requires that the instruction memory be initialized with the machine code of some instructions before simulation begins. The memory component provided to you will initialize itself with the values in 'progmem.txt' in your project folder. The format of this file assumes a single 32-bit word per line, specified in hex. The provided 'progmem.txt' file has a few instructions. The value of the PC at reset will be address 0, thus your instructions should start at address 0 in memory. To test certain features of your CPU you will need to convert some instruction sequences to machine code and fill in this progmem.txt file by hand. Initial data for access by LW's can be placed in the 'datamem.txt' file. The testbench will initialize the data memory with the contents of this file.

Program 1 in progmem.txt

```
        addi $9,$0,0x0084
        xor $8,$0,$0
        nor $8,$0,$0
L1:     lw   $4,-4($9)
# data at addr 0x80 = 0x12345678
        lw   $5,0($9)
# data at addr 0x84 = 0xfffffffe
        add  $17,$4,$5
        sub  $18,$4,$5
        and  $16,$4,$5
        or   $16,$4,$5
        slt  $16,$17,$18
        slt  $19,$8,$0
        beq  $19,$0,L2
        sw   $16,0($9)
        addi $8,$8,1
        beq  $0,$0,L1
L2:     beq  $0,$0,L2      # inf. loop
```

When hand-assemblying instructions, use the machine code format below:

R-Type:	31:26	25:21	20:16	15:11	10:6	5:0
ADD $rd,$rs,$rt	000000	rs	rt	rd	00000	100000
SUB $rd,$rs,$rt	000000	rs	rt	rd	00000	100010
AND $rd,$rs,$rt	000000	rs	rt	rd	00000	100100
OR $rd,$rs,$rt	000000	rs	rt	rd	00000	100101
XOR $rd,$rs,$rt	000000	rs	rt	rd	00000	100110
NOR $rd,$rs,$rt	000000	rs	rt	rd	00000	100111
SLT $rd,$rs,$rt	000000	rs	rt	rd	00000	101010

I-Type:	31:26	25:21	20:16	15:0
LW $rt,$disp_{16}$($rs)	100011	rs	rt	$disp_{16}$
SW $rt,$disp_{16}$($rs)	101011	rs	rt	$disp_{16}$
ADDI $rt,$rs,imm_{16}	001000	rs	rt	imm_{16}
BEQ $rs,$rt,$disp_{16}$	000100	rs	rt	$disp_{16}$

*Note: BEQ adds two 0's to the LSB's of the displacement and adds that value to the already incremented PC (i.e. target addr. = addr. of BEQ + 4 + (disp.*4)). Make sure you remove the two 0's when you calculate the stored disp. value.

J-Type:	31:26	25:0
J $addr_{26}$	000010	$addr_{26}$

*Note: J adds two 0's to the LSB's of the jump address. Thus, you should store the desired address with the two 0's removed.

When you simulate your design you will be able to see the signals that have been brought out as outputs. To determine if your design is working or not, it is likely easiest to look at the memory address, read and write values as well as register read and write values to see if they match expectation [obviously you will need to calculate the expected values from each instruction and ensure the actual values match your expectation]. Once an error is found, you can drill down into the design hierarchy in the Workspace pane to find the component and internal signals that you'd like to view and drag the desired signal name to the waveform window. Restart the simulation [restart -f] and re-run the simulation [run XXXns].

4 Prelab

None.

5 Procedure

5.1 Part 0:

Download the "Single Cycle Reference Code". This file is complete. Run the simulation of the test bench "ee457_scpu_tb.v" for about 1000 ns and observe the behavior of the single cycle CPU. It is essential that you thoroughly understand the code "ee457_scpu.v" for the following parts. The TA will run through this code during the discussion.

Add support for the ADDI instruction: Modify the Control Unit ("ee457_scpu_cu.v") and add/modify the datapath in "ee457_scpu.v" as necessary.

5.2 Part 1:

You will now take the single cycle CPU design and modify it to create a 5 stage pipelined CPU WITHOUT forwarding or hazard detection. To do this:

Modify the single cycle CPU into a 5 stage pipeline processor by adding pipeline registers (modeled as always blocks that take one set of input signals, register them and output them to the next stage. Reminder you do not need to consider data forwarding, flushing and hazards.

The 5 stages are:

- Instruction Fetch: Consisting of the PC (a register itself!) and the instruction memory
- Instruction Decode: Consisting of the register file and all control logic that outputs control signals. Unconditional jumps commits at this stage

- Execution: Consisting of the ALU, branch target is computed in this stage, branch decisions are computed, but not made here
- Memory: Consisting of the data memory, branch decisions are propagated to this stage from the execution stage, and decisions are being carried out at this stage
- Write-Back: Writes back to the register file, does not consist of any big modules.

Update the register file without forwarding to now allow for internal forwarding (i.e. when one of the read register IDs matches the write register ID and there is intent to write, then pass the new write data, not the old value of the read register).

Test your design by simulating it. Note that if you use the contents of the instruction memory (i.e. the earlier given assembly program) it will not function properly because of the lack of forwarding, but the first couple instructions will. You are strongly advised to hand-assemble your own instruction sequence where there are no dependencies (i.e. no need for forwarding), type their hexadecimal values into the instruction memory text file, and verify your Verilog is working.

Submit your part 1 tested code by following the instructions on the website.

5.3 Part 2

Forwarding Logic

Implement the forward logic described in class and add forwarding muxes to the EX stage logic. You do not have to create a separate Verilog module for the forwarding unit but can describe it in the same Verilog file as your CPU. Similarly, implement the HDU. You can make this a separate module or implement it in the top-level design file. Test your design with an appropriate sequence of instructions (you may hand assemble some instructions that have data dependencies that will exercise your forwarding logic and HDU).

Branch hazard detection and resolution.

We will implement "late" (MEM-Stage) Branch outcome determination. Implement the flushing logic. Do you flush all stages or particular stages? How to flush the IF stage? Add logic to flush appropriate instructions in the pipeline when a successful branch is executed. You can zero the control signals or simply reset an entire pipeline registers. Your new pipelined CPU should be able to handle the original progmem.txt program that exercises stalling and flushing.

5.4 What to turn in

You will turn in your part 1 and part 2 at separate due dates. See website for more info and be sure to submit all of your working Verilog code files and memory files (see instructions on the website) in a .ZIP file.