ECSE 425: Computer Organization and Architecture
Pipelined Processor

Due March 17, 2017, 11:59 PM

This deliverable consists of a pipelined implementation, in VHDL, of a simplified MIPS processor.

Change Log

- 17-Feb Initial revision.
- 20-Feb Requirement for GitHub repository added.
- 21-Feb Clarified that input and output files should be in *ascending order*.
- 22-Feb Specified that data memory must have size 32768 bytes.
- 22-Feb Specified that instruction memory must have size 1024 bytes, and that LB and SB are not required to be implemented.

Pipelined Implementation of a 32-bit MIPS Processor in VHDL

You will implement a standard five-stage pipelined 32-bit MIPS processor in VHDL. A pipelined processor stores control signals and intermediate results associated with each executing instruction in pipeline registers between stages. Refer to the the text for further information. For full credit, implement:

1. Hazard detection
2. Forwarding

Instruction Set Architecture

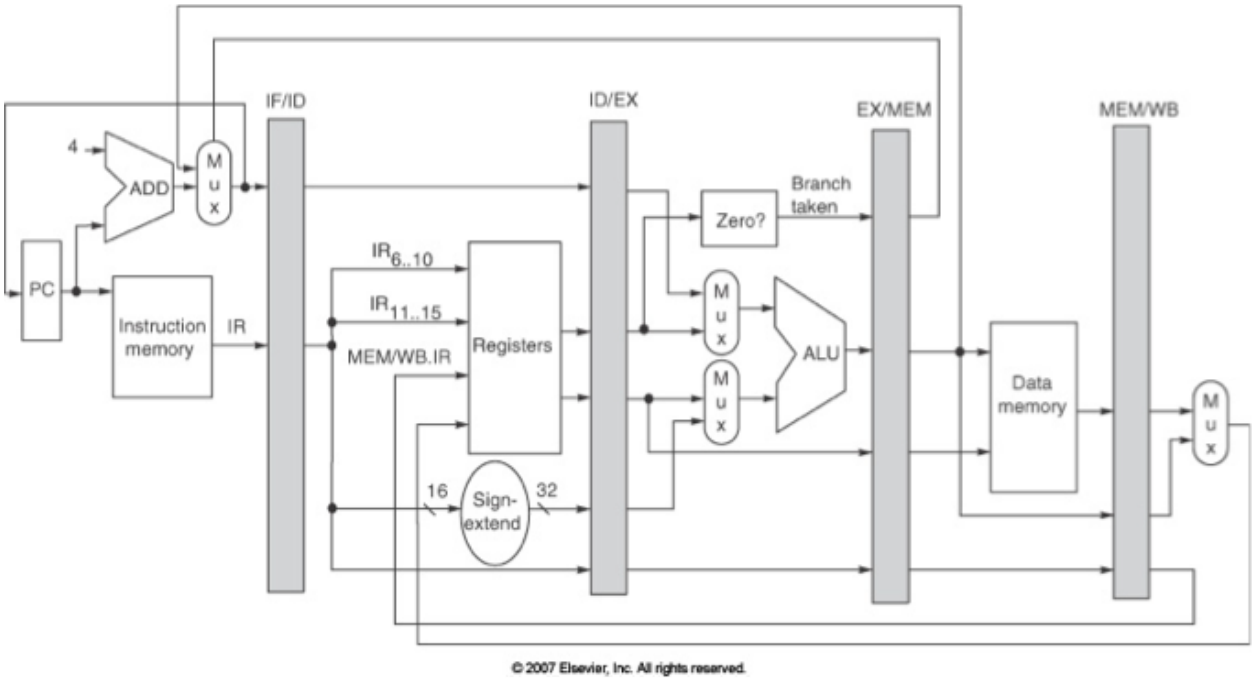
You will implement a subset of the MIPS instruction set architecture. Refer to the MIPS Reference Data card:

http://www-inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf

and the text, for instruction formats, descriptions of operations, and other information. To ensure compatibility with the provided assembly language programs, your instruction formats must conform to those in the MIPS Reference Data card.

You may assume that all instructions—including `mult`*—require a single cycle in the EX stage.*

Your processor must be able to execute each of the required assembly language instructions (see *Appendix*). Assume that execution begins at address `0x0` in memory.



Beyond the required instructions, your processor must implement the following functionality:

- Register `$0` must be wired to `0x0000`, and
- The PC must be initialized to `0x0`.

Your processor *need not* implement the following functionality:

- Floating point arithmetic, and
- Interrupts and Exceptions.

Hazard Detection

In order to function properly, your pipelined processor must implement *hazard detection*. Hazard detection stalls instructions in ID when a required operand is not ready yet (e.g., due to a pending ALU operation or load instruction). A stall (or bubble) can be inserted in the pipeline by inserting an `add $r0, $r0, $r0` instruction into the EX stage rather than the waiting instruction.

Implement hazard detection logic and the corresponding control logic such that instructions dependent operands that are not yet available stall in ID. It is recommended that you implement and test hazard detection *first*.

Forwarding

Forwarding takes results from the EX/ME and ME/WB pipeline registers and makes them available as ALU inputs in order to eliminate stalls. Without forwarding, an instruction

stalled in ID cannot proceed until the cycle during which the required operand is written back to the register file.

Implement forwarding from the EX/ME and ME/WB pipeline registers and the corresponding control logic such that stalls are eliminated when possible. It is recommended that you implement forwarding *second*, after hazard detection has been tested, and that you update hazard detection accordingly.

Memory

To implement the memory for instructions and data, use the memory model provided for PD3. Instantiate two memories, one memory for instructions and one memory for data. You may alter the memory model as you see fit (e.g., set the memory delay to 1 clock cycle, if it makes your life easier). However, you must keep the data memory sized at 32768 bytes. You must also ensure that your processor can run a program of at most 1024 instructions.

Testbench

To allow you to try different MIPS programs on your processor, we are providing you with an assembler which can convert *MIPS assembly* into *machine code*. For our purposes, *MIPS assembly* is restricted to the subset of functionality supported by the MiSaSiM instruction set simulator. For example:

```
# This program computes the factorial of the input in $1. The
# result is returned in $2.
# $1: n
# $2: running product (output n!)

Fact:   addi  $1, $0, 5         # input: n = 5
        addi  $2, $0, 1         # initialize output to 1
Loop:   slti  $3, $1, 2         # if input is less than 2
        bne   $3, $0, Skip      # then skip to return
        mult  $1, $2            # else multiply input and running product
        mflo  $2                # assume small numbers
        addi  $1, $1, -1        # decrement input
        j Loop                  # and loop
Skip:   jr    $31               # return to caller
```

The provided assembler supports the following functionality:

- Comments, indicated with "`#`",
- Empty lines,
- Arbitrary whitespace (spaces, tabs) within lines,
- Labels, if present at the beginning of a line and ending with "`:`",
- Instruction arguments using numbered registers, such as "`$0`" and "`$31`" and
- Branches and jumps to labels.

The output of the assembler is an ASCII text file, with one 32-bit word on each line, encoded in binary (i.e., 32 '1's or '0's), *in ascending order*; this is the expected input format for your processor testbench. Your output files should also have this format.

Your testbench should read a program in machine code from an ASCII text file called "program.txt", put the program into instruction memory, run the program on your processor, write the final contents of data memory to a text file called "memory.txt", and write the final contents of the register file to a text file called "register_file.txt". Since the data memory has 32768 bytes, "memory.txt" should have 32768/4 = 8192 lines, one for each 32-bit word. Likewise, since there are 32 registers, "register_file.txt" should have 32 lines. Name your testbench `testbench.tcl`.

You will need to wait until the program is completely finished to write the output files. To determine when the program is finished, your testbench should monitor the value of the PC and, when the processor has completed the final instruction of the program, stop the simulation and write the output.

Grading

To test and grade your processor, the instructional staff will run your `testbench.tcl` and check whether the contents of the output files are correct. We are providing you with a subset of the suite of assembly programs which we will use to test your processor.

In the event that your results do not match the correct results, your processor will be inspected *to the extent possible* and partial credit awarded accordingly. Note: we will rely heavily on comments provided in your code to find the location of errors. Obtuse, uncommented code will receive substantially less partial credit.

Hand In Procedure

Hand in, via MyCourses, in a single ZIP file:

- `testbench.tcl`
- All VHDL files required to simulate your processor.

Also, please include the link to a public Github repository hosting your code in the submission box comments.

Appendix

Your processor must implement each of the following 27 instructions. These instructions are all supported by MiSaSiM. Refer to the MIPS Reference Data card for instruction format, opcode, and functionality.

Class	**Instruction**	**Mnemonic**
Arithmetic	Add	add
	Subtract	sub
	Add Immediate	addi
	Multiply	mult
	Divide	div
	Set Less Than	slt
	Set Less Than Immediate	slti
Logical	And	and
	Or	or
	Nor	nor
	Xor	xor
	And Immediate	andi
	Or Immediate	ori
	Xor Immediate	xori
Transfer	Move From HI	mfhi
	Move From LO	mflo
	Load Upper Immediate	lui
Shift	Shift Left Logical	sll
	Shift Right Logical	srl
	Shift Right Arithmetic	sra
Memory	Load Word	lw
	Store Word	sw
Control-flow	Branch On Equal	beq
	Branch On Not Equal	bne
	Jump	j
	Jump Register	jr
	Jump and Link	jal