Homework Assignment 3

Fall 2017

Department of Computer Science

George Mason University

Due Nov. 6th, 23:59pm

No Late Submissions Accepted

Task:

There are 2 parts to this assignment: part 1: written exercises for Ch4; part 2: MIPS programming. Basic rules regarding assignment implementation and submission:

- Total points of this assignment: 100.
- **Team Allowed. Maximum 2 per Team**. State clearly names and G#'s in all files. If you have a group of two, only one needs to submit the required zip file as described below. But **names and G#'s of group members must be on ALL submitted files.** The other member must just submit a one-page pdf stating team member names and IDs. **Members of a group must attend the same section of CS465.**
- **What to submit**: follow these steps to prepare and submit a ZIP file
    a. Prepare a PDF file for Part 1 with answers to questions named as "cs465_hw3_LastName1_Lastname2.pdf" (for 2-member teams), or "cs465_hw3_LastName1.pdf" (for 1-member teams);
    b. For Part 2, name your MIPS source code as "cs465_hw3_LastName1_LastName2.asm"(for 2-member teams), or "cs465_hw3_LastName1.asm" (for 1-member teams). Do NOT submit it as a .pdf. As in all programming assignments, your code must be very well commented. A description of your algorithm must be included as part of your header comment.
    c. Place both files in a folder and rename the folder to be "cs465_hw3_LastName1_LastName2"(for 2-member teams), or "cs465_hw3_LastName1" (for 1-member teams). Then compress it to a **ZIP** file (instead of compressing and then renaming).
- **How to submit**: Submission will be made via a blackboard link available to you.
- **Plagiarism is not permitted in any form. We enforce the university policy and honor code.**

Part 1. Written Exercise (40%)

- **A large portion of (or all) points will be taken off if you only answer with a number.** You must show steps to justify your answer.
- Answers must be legible, especially if you scan to generate your .pdf.

1. (15 pts) Assume that we only consider the following latencies for elements in a datapath as in Figure 4.17 of the textbook.

I-Mem	Add	Regs	ALU	D-Mem	Sign-extend	Shift-2
380ps	100ps	150ps	160ps	450ps	40ps	40ps

a. How long does it take to complete the execution of a BEQ instruction?
b. How long does it take to complete the execution of an ADD instruction?
c. How long does it take to complete the execution of a LW instruction?
d. If we can reduce the latency of only one datapath element by 25%, which component should it be? Why?

2. (18 pts) Given the following sequence of instructions to be executed on a 5-stage pipelined datapath as descripted in our textbook:

```
I1:   add   $8, $9,$10
I2:   add   $11,$11,$8
I3:   lw    $8,0($9)
I4:   or    $8,$8,$10
I5:   sw    $11,0($8)
```

a. List true dependencies in the given sequence in the format of (register_involved. producer_instruction, consumer_instruction). Use labels to indicate instructions. For example: `($1, I10, I11)` means a true dependence between instruction `I10` and `I11`: value of register `$1` is generated by `I10` and used by `I11`. Do **NOT** list output or anti-dependencies.

b. If there is no forwarding or hazard detection, no reordering allowed, insert nops to ensure correct execution.

c. If there is full forwarding support, draw multiple-cycled pipeline diagram (like Figure 4.44) to show the execution of the sequence. Use arrows to mark forwardings clearly in your diagram. Each arrow should point from instruction/stage that generates the data to instruction/stage that consumes the data. Also mark the necessary pipeline stalls.

3. (7 pts) This exercise examines the accuracy of various branch predictors for the following repeating pattern (e.g., in a loop) of branch outcomes: **NT, T, T, T, NT, T, T.** Note: include a brief explanation for each number in your answer.
a. What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

b. What is the accuracy of the two-bit predictor if the pattern is repeated forever? You can assume the initial state is strong "Predict not taken" (bottom left one of Figure 4.63).

Part 2. MIPS Programming (60%)

MIPS Interpreter. For this assignment, you will write a program that accepts a MIPS program in machine code (as hex numbers) from the user, decodes and executes the program. You will consider a limited instruction set: **`add`**`,` **`addi`**`,` **`sub`**`,` **`slt`**`,` **`lw`**`,` **`sw`**`,` **`beq`**`,` `and` **`j`** `plus some additional instructions listed below.`

You have to simulate the instruction memory, the register file, the ALU, and the data memory. The instruction memory should be 100 words long and should be declared as IM in the data area, the register file is 32 words long and should be declared as REGS in the data area, and the data memory should be 200 words long and should be declared as DM in the data area. Note that you have to initialize REGS[0] as zero because register zero always has the value zero.

Additionally, your data segment should have the following words to support the execution of an instruction:

- a word labeled **PC** to simulate the program counter
- a word labeled **INSTR** to store the instruction fetched
- a word labeled **RS** to store the value of the rs register field if the instruction in INSTR has a rs field. This is the register number stored as a word not its value.
- a word labeled **RT** to store the value of the rt register field if the instruction in INSTR has a rt field. This is the register number stored as a word not its value.
- a word labeled **RD** to store the value of the rd register field if the instruction in INSTR has a rd field. This is the register number stored as a word not its value.
- a word labeled **REG1** to store the value of the top register read from the register file (this is the register output that goes directly to an ALU input without going to a multiplexer).
- a word labeled **REG2** to store the value of the bottom register read from the register file.
- a word labeled **OPCODE** to store the value of the opcode of the instruction in INSTR
- a word labeled **FUNC** to store the value of the function field of the instruction in INSTR
- a word labeled **CONSTANT** to store the value of the constant/address field of the instruction in INSTR
- a word labeled **ALUOUT** to store the value of the output of the ALU.
- a word labeled **ALUZERO** to store the value of ALU's zero bit (as a word).
- a word labeled **DMOUT** that stores a word read by DM.

Required subroutines: You need to write the following subroutines that correspond to phases of a datapath:

- **Fetch ():** fetches the instruction from IM pointed by PC and stores it into INSTR and increments the PC by 4.
- **Decode():** extracts the opcode from INSTR and stores it into OPCODE. Depending on the opcode, extracts the relevant fields (i.e., RT, RS, RD, FUNC, CONSTANT/ADDRESS) of the instruction. If any of the fields does not exist in the

instruction they should be set to -1. The extract function that you programmed for homework 2 will be useful here.

- **ReadRegister (r1, r2):** read registers numbered r1 and r2 from REGS and store their values in the appropriate words REG1 and REG2. Arguments are one word each.
- **ALU (input1, input2, alu-control):** performs the operation indicated by alu-control according to the table below

Alu-control	Operation
0010	ALUOUT = Input1 + Input2
0110	ALUOUT = Input1 – Input2 and (ALUZERO = 1 if ALUOUT == 0 and 0 otherwise)
0111	1 if Input 1 < Input 2 and 0 otherwise

- **ReadMem (address):** reads the word in the data memory DM [address] and stores it into DMOUT.
- **WriteMem (data, address):** writes the word in "data" into DM [address].
- **RegWrite (r, data):** writes the word into "data" into register r (i.e., REGS[r]).
- **ExecuteInstr ():** executes the instruction pointed by PC. This subroutine uses the subroutines described above and further updates the PC in case of branch and jump instructions. Note that the fetch subroutine already incremented PC by 4.

Consider the following **additional instructions**:

- **halt:** opcode= 111111 stops the execution of the program
- **preg rs:** opcode= 111110. Prints the value of register rs (bits 25-21) in the following format "Register [rs] = value of register rs"

Main program: You should read the instructions of the program to be simulated from the input, one instruction per line, and store them in the instruction memory starting at IM[0].

Set PC = 0 and invoke ExecuteInstr in a loop until a halt instruction is found.

Use the following program (given below in MIPS assembly language; you have to convert it to hexadecimal to use as input):

Address	Instruction	Assembly language
0x00000000	0x200D0000	addi $t5,$zero,0
0x00000004	0x20090022	addi $t1,$zero,34
0x00000008	0x200A0000	addi $t2,$zero,0
0x0000000C	0xF9400000	preg $t2
0x00000010	0xADAA0000	sw $t2, 0($t5)
0x00000014	0x200B0001	addi $t3,$zero,1
0x00000018	0x21AD0004	addi $t5,$t5,4
0x0000001C	0xF9600000	preg $t3
0x00000020	0xADAB0000	sw $t3,0($t5)

0x00000024	0x21AD0004		addi	$t5,$t5,4
0x00000028	0x01606020	loop:	add	$t4,$t3,$zero
0x0000002C	0x014B5820		add	$t3,$t2,$t3
0x00000030	0x01805020		add	$t2,$t4,$zero
0x00000034	0xF9600000		preg	$t3
0x00000038	0xADAB0000		sw	$t3,0($t5)
0x0000003C	0x21AD0004		addi	$t5,$t5,4
0x00000040	0x11690001		beq	$t3,$t1,load
0x00000044	0x08000007		j	loop
0x00000048	0x21ADFFFC	load:	addi	$t5,$t5,-4
0x0000004C	0x8DAE0000		lw	$t6,0($t5)
0x00000050	0xF8C00000		preg	$t6
0x00000054	0x11A00001		beq	$t5,$zero,stop
0x00000058	0x08000012		j	load
0x0000005C	0xFC000000	stop:	halt	

Grading rubric:

- Code submitted, well commented, assembled 10/60
- Support initialization, halt and preg 10/60
- Support add, addi, sub,slt 15/60
- Support lw, sw 15/60
- Support beq, j 10/60