CS/ECE 552: Introduction to Computer Architecture
Spring 2019

Professor Matthew D. Sinclair

Midterm Exam SOLUTION

Thursday, March 7th, 2019

Weight: 20%

CLOSED BOOK

ONE 8.5"x11" SHEET OF NOTES (TWO-SIDED) ALLOWED

NAME: ____________________________

DO NOT OPEN EXAM UNTIL TOLD TO DO SO!

Read over the entire exam before beginning. You should verify that your exam includes all of the problems listed in the table below as well as two spare pages for scratch work. Budget your time according to the weight of each question and your ability to answer them. I have provided a large amount of space for you to write your answers, but if that space is not sufficient, please write on the BACK of the SAME sheet. Otherwise, you should use the back of the sheet or the provided extra pages for scratch work. WRITE YOUR NAME ON EACH SHEET. You will turn in your cheat sheet with your exam, so please make sure your name is written on it.

Upon announcement of the end of the exam, stop writing on the exam paper immediately. Pass the exam to the head of the tables to be picked up by the proctor (the TAs and me). The instructor will announce when to leave the room. Failure to follow instructions may result in forfeiture of your exam and will be handled according to the UW Academic Misconduct policy.

Problem	Possible Points	Points
Problem 1	5	
Problem 2	4	
Problem 3	10	
Problem 4	10	
Problem 5	10	
Problem 6	21	
Total	60	

Name: ______________________________

Problem 1 [5 points]

Part A [1 point]

Assume 90% of a program can be executed in parallel. What speedup is required on the parallel section to achieve an overall speedup of 4X for the full program?

Solution:

Let S be the speedup on the parallel section and T be the original total time.

.9T/S + .1T = T/4

.9/S + .1 = .25

.9/S = .15

S = .9 / .15 = 6X speedup is required on the parallel section.

Part B [2 points]

Again assume 90% of a program can be executed in parallel. What is the maximum possible speedup achievable on the program through parallelization?

Solution:

In the limit, the parallel section reduces to 0, so speedup = T / 0.1T = 10X.

Part C [2 points]

Evaluate the potential performance of two processors, each implementing a different ISA. The evaluation is based on its performance on a particular benchmark. On the processor implementing ISA *A*, the best compiled code for this benchmark performs at the rate of 8 IPC. That processor has a 512 MHz clock. On the processor implementing ISA *B*, the best compiled code for this benchmark performs at the rate of 3 IPC. That processor has a 1024 MHz clock. What is the performance in Millions of Instructions per Second (MIPS) of the processor implementing ISA *A* and the processor implementing ISA *B*?

Solution:

ISA *A*: 8 instructions/cycle ∗ 512,000,000 cycle/second = 4096 MIPS

ISA *B*: 3 instructions/cycle ∗ 1,024,000,000 cycle/second = 3072 MIPS

Problem 2 [4 points]

Given the following MIPS code:

```
SUB $1, $4, $3
SLL $1, $1, 7
ADD $1, $6, $1
LW $2, 8192($1)
```

```
SW $2, 4($30)
```

And given that the machine code for the subtract is 0x00830822, what is the correct machine language code for the remaining MIPS assembly in either binary or hexadecimal?

Solution:

Hex	Op	Rs	Rt	Rd	Address/shamt	Funct
0x00830822	00 0000	0 0100	0 0011	0 0001	0 0000	10 0010
0x000109C0	00 0000	0 0000	0 0001	0 0001	0 0111	00 0000
0x00C10820	00 0000	0 0110	0 0001	0 0001	0 0000	10 0000
0x8C222000	10 0011	0 0001	0 0010	0010 0000 0000 0000		
0xAFC20004	10 1011	1 1110	0 0010	0000 0000 0000 0100		

Problem 3 [10 points]

Assume a 5-stage pipeline, as discussed in class, where branches are resolved in ID, and all needed forwarding paths are provided. Consider the following code on the above machine:

```
loop:

     1.  LW        $t4,     0($s1)
     2.  LW        $t5, 4096($s1)
     3.  LW         $t6,     0($t5)
     4.  SUB        $t7, $t4, $t6
     5.  SUB       $t5, $t1, $t2
     6.  ADD        $t3, $t3, $t7
     7.  ADDI       $s1, $s1, 8
     8.  BNE        $s1, $s3, loop   // branch if $s1 does not equal $s3
```

Part A [4 points]

Identify all RAW hazards (regardless of whether they require forwarding or not) in the above code.

Solution:

NOTE: This solution assumed you only considered a single loop iteration. If you assumed there were multiple loop iterations, there are additional answers and you were graded accordingly.

- 1 → 4
- 2 → 3
- 3 → 4
- 4 → 6
- 7 → 8

Part B [4 points]

Give one pair of instructions with a RAW dependence that requires the most separation between them (in the above program) to avoid stalling. What is the minimum number of other instructions that would need to be between this pair in program order to avoid the stall? If multiple pairs require the same separation, pick one. Explain your answer.

Solution:

A load and an instruction that uses the load's result require the most separation. Any such pair is acceptable. Each of these pairs requires 1 instruction of separation. The load produces a result after its MEM stage, and in this code all the instructions that need the result of the load will use it in or have it forwarded to EX.

Alternative: The $s1 calculation in the ADDI needs to be forwarded to the BNE, which does the decision logic in ID. This also requires a stall in a similar manner.

Part C [2 points]

What pair of instructions with a WAR dependence, if any, requires the most separation between them (in the above program) to avoid stalling? What is the minimum number of other instructions that would need to be between this pair in program order to avoid the stall? If multiple pairs require the same separation, pick one. Explain your answer.

Solution:

There is only 1 WAR dependence (3 → 5). However, for an in-order pipeline WAR hazards never cause stalls, so the minimum number of instructions needed is 0.

Problem 4 [10 points]

You are an engineer at a company that produces 5-stage RISC processors, like the ones discussed in class. Due to a tight product deadline, you only have enough time to implement some of the forwarding optimizations (you may assume your processor has already implemented register file bypassing). Assume that branches are resolved in the ID stage. To prioritize how you spend your time, you profile the applications expected to run on your processor, and find that they have the following mix of instructions:

Type of Instruction	**% Instructions**
Store	25
Load	25
Branch (including all jumps)	20
ALU	30

Part A [5 points]

If you only have enough time to implement one forwarding optimization, which one would you choose and why? To receive credit, you must justify your answer.

Solution:

The options to consider are EX → ID, MEM → ID, EX → EX, MEM → EX, and MEM → MEM.

Given the above breakdown, forwarding for stores is likely going to be the least useful, in particular MEM → MEM forwarding (which is only useful for stores). EX → EX forwarding or MEM → EX forwarding may be more useful for stores, but again the percentage of stores is small. Forwarding for branches is going to be useful at most 20% of the time (if every branch needs to be forwarded). Given that some branches may not use forwarding, and that some may need to be forwarded from EX and others from MEM, this is unlikely to be the most useful (i.e., by reducing potential stalls).

However, given that loads and ALU operations make up the majority of instructions, and both need the ALU, it is likely that EX → EX forwarding will be the mostly useful and impact CPI the most. MEM → EX may also be very useful, depending on the flow of instructions. But, with the above breakdown, it is likely that EX → EX will be more useful since instructions often have dependencies on the instruction directly before them and ALU instructions (and Loads/Stores) will all benefit from this.

(MEM → EX is also fine, if you justify why it is more useful than EX → EX)

Part B [5 points]

If you have enough time to implement a second forwarding optimization, which one would you choose and why? To receive credit, you must justify your answer.

Solution:

MEM → EX is the next most useful, for all of the reasons mentioned above.

Problem 5 [10 points]

Part A [5 points]

In terms of gate delays, what is the performance of a 64-bit ripple carry adder and a 64-bit adder using carry-lookahead logic?

Solution:

See page B-46, which shows the logic for a 16-bit carry ripple adder, which has 16x2 + 1 = 32 gate delays (the extra 1 gate is for the initial carry in). Similarly, a 64-bit carry ripple adder has 64x2 + 1 = 129 gate delays. A 64-bit carry lookahead adder's worst case delay is similar to the 16-bit one mentioned on B-46, except we need to add an extra level (think of the 16-bit carry lookahead adder being built out of 4 of the 16-bit carry lookahead adders), this extra level has 2 gate delays in the worst-case – so overall the 64-bit adder has 5 + 2 = 7 gate delays (5 comes from the 16-bit carry lookahead adder delay on B-46). Thus, the 64-bit carry lookahead adder is ~18X faster than the ripple carry adder.

Part B [5 points]

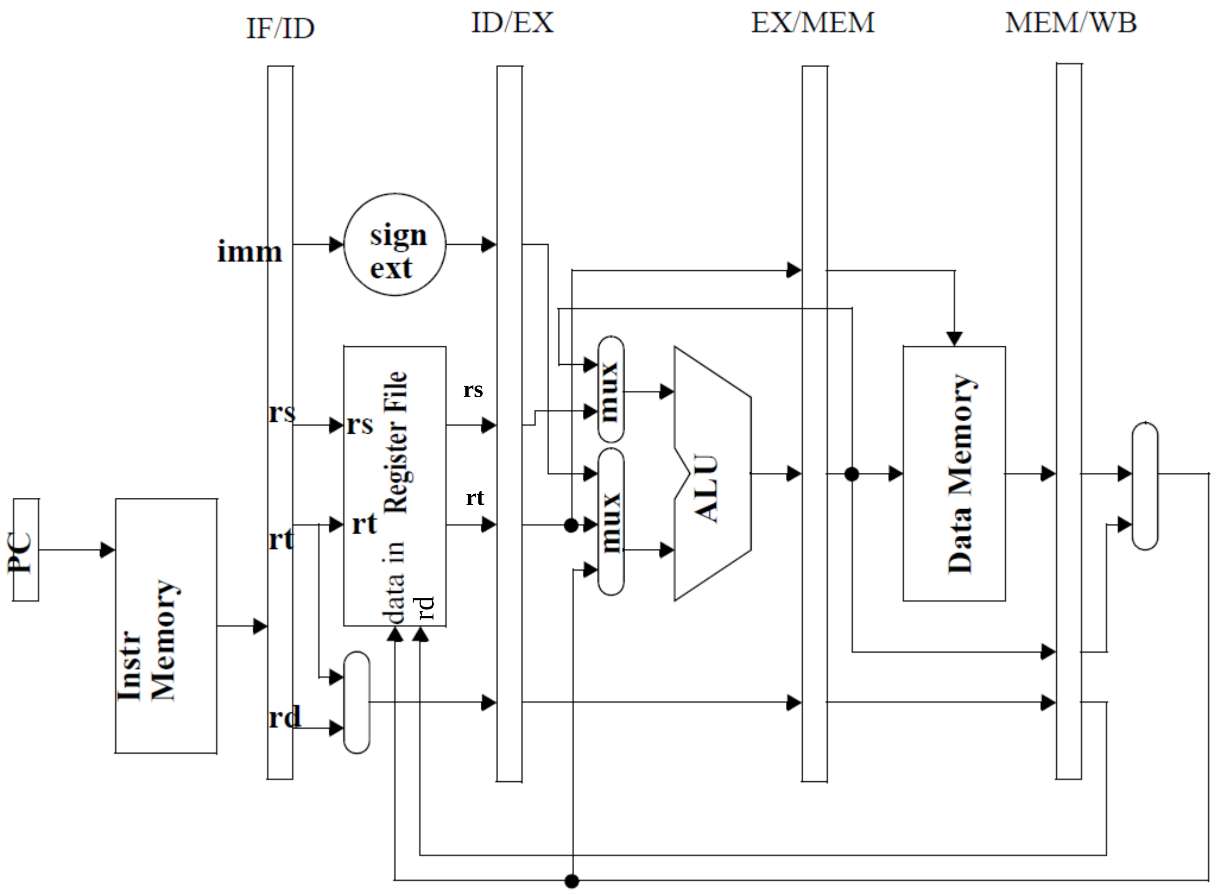
As discussed in class, the MIPS ISA utilizes a branch delay slot. Does the branch delay slot work better with a predict-taken or a predict not-taken policy for branches? To receive credit, you must explain your answer.

Solution:

The branch delay slot works better with the predict not-taken policy, because in this policy we already know what the next PC (PC + 4) is. With a predict taken policy, we won't know what the next PC to use (the one immediately after the branch label) is until the ID stage, which defeats the purpose of the branch delay slot.

Problem 6 [21 points]

High performance datapaths use bypass paths (also known as data forwarding logic) to reduce pipeline stalls. However, bypass paths are relatively expensive, especially in some wire constrained technologies. To reduce the cost (and potential cycle time impact), some architects have explored omitting some of the possible bypass paths. Consider the datapath illustrated below (note that the PC update logic and all control logic is intentionally omitted). This pipelined datapath is similar to the ones in the book and discussed in class, *but has several differences including limited bypass paths*. **BE SURE TO STUDY THE DATAPATH CAREFULLY**! In each stage, you should assume the upper wire is for 'rs', and the lower one is for 'rt' (as highlighted in the Decode stage below). Assume that the register file supports bypassing, so that if register `$i` is read and written in the same cycle, then the read returns the new value. Assume that the control logic bypasses the data as soon as possible using the given forwarding data paths, and stalls in Decode otherwise. **You may NOT add additional data paths**.



In this problem, you will look at how a program snippet performs on this pipeline. Recall that R-format instructions have the form:

```
  opcode rd, rs, rt
```

and I-format instructions have the form:

```
opcode rt, imm(rs)
```

or:

```
opcode rt, rs, imm
```

Use the table below to show how the given instruction sequence flows through the pipeline and where stalls are necessary to resolve hazards (*as discussed in class, use "*" to indicate a pipeline stall at the clock cycle of the corresponding instruction; for example F* signifies a stall in the Fetch stage*).

Solution:

Instr / Cycle	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
SUBI $1, $1, 1	**F**	**D**	**X**	**M**	**W**															
LW $3, 0($1)		F	D	X	M	W														
LW $4, 4($1)			F	D*	D	X	M	W												
OR $5, $3, $4				F*	F	D*	D	X	M	W										
ADD $3, $5, $6						F*	F	D	X	M	W									
SW $3, 8($2)								F	D*	D*	D	X	M	W						
ADDI $2, $2, 8									F*	F*	F	D	X	M	W					

For each cycle where a stall occurs, *explain why below*.

Note that 'rs' has EX → EX forwarding, but not MEM → EX forwarding, and 'rt' has MEM → EX forwarding, but not EX → EX forwarding (except that SW can't use the 'rt' forwarding).

Cycle 3: There is a stall for the LW because $1 ('rs' for the LW) is read from the SUBI, which is in the MEM stage in cycle 4, but there is no forwarding for MEM → EX forwarding for 'rs'. This also causes a stall in the subsequent LW $4 (in fetch).

Cycle 5: The OR is stalled in decode. Although the LW $3 has loaded it's value from memory and we could use register file bypassing in this cycle, LW $4 has a load-use stall. This also causes a stall in the subsequent ADD (in fetch).

Cycle 8 & 9: $3 is the 'rt' register for the SW, and it needs the value written by the ADD on the previous line, but there is no support no bypass path on the path to data memory (i.e., SW can't use the 'rt' bypassing in EX). Thus, the SW must stall until it can read $3 from the RF bypassing. This also stalls the subsequent ADDI (in fetch).