CpE 313 Fall 2004
Microprocessor Systems Design
One Solution to Exam 1

September 30, 2004

Instructions

Read each individual problem appearing on this exam carefully and do only what is specifically stated.

There are two blank grids at the very end if you would like to use them for drawing pipeline timings.

This exam is designed to be completed by a well-prepared student in approximately **75 minutes**; most students are expected to finish it within the allotted time. On your initial pass through this exam, skip any problems that appear to be overly difficult.

IMPORTANT: Put down your initials at the TOP of EACH page. Also, be sure to read and sign the Academic Honesty Statement that follows:
"In signing this statement, I hereby certify that the work on this exam is my own and that I have not copied the work of any other student while completing this exam. I understand that, if I fail to honor this agreement, I will receive a score of ZERO for this exam and will be subject to possible disciplinary action."

Printed Name: .... **Signature:** ....

Date: ....

DO NOT BEGIN UNTIL INSTRUCTED TO DO SO.

Problem 1 – Performance Measurement .... 15 Points

RISCy Computers Corporation is designing a new microprocessor. The benchmark on which they have chosen to evaluate their processor has 60% integer instructions. The CPI for integer instructions on their existing design, called "bRISCy," is 5.

The CPI for floating point instructions on bRISCy is 16. A certain percentage, X, of the total instructions are floating point multiply instructions that take 40 cycles each on bRISCy.

In the new design, two proposals are being considered. Proposal A suggests putting a new floating point multiply unit to reduce the floating point multiply CPI to 20. Proposal B suggests re-hauling the entire floating point unit to decrease the floating point CPI to 10.

1a) Calculate the CPI for the benchmark on a bRISCy computer.

The formula needed here is:

$$CPI = \sum_{instruction\ type, op} CPI_{op} \times freq_{op}$$

The problem divides instructions into two broad types, integer instructions and floating point instructions. $CPI_{integer} = 5$, $CPI_{floating\ point} = 16$. *The respective frequencies are 60% and 40%.*

$$CPI = CPI_{integer} \times freq_{integer} + CPI_{floating} \times freq_{floating}$$

$$CPI = 5 \times 0.6 + 16 \times 0.4 = 9.4$$

1b) Calculate the CPI for the benchmark if bRISCy is redesigned using proposal B.

With proposal B, $CPI_{floating}$ *becomes 10.*

$$CPI_{proposal\ B} = 5 \times 0.6 + 10 \times 0.4 = 7$$

1c) What value of X makes proposal A better than proposal B?

Proposal A will be better than proposal B if:

$$\begin{aligned} CPI_{proposal\ A} &< CPI_{proposal\ B} \\ CPI_{proposal\ A} &< 7 \end{aligned}$$

To calculate $CPI_{proposal\ A}$, we subtract the change in CPI from the CPI value in part 1a. Let us denote by "fpm" the floating point multiply instructions. Then,

$$\begin{aligned} CPI_{proposal\ A} &= CPI_{from\ part\ 1a} - freq_{fpm}(CPI_{fpm,\ original} - CPI_{fpm,\ proposal\ A}) \\ &= 9.4 - X \times (40 - 20) \\ &< 7 \\ 9.4 - 7 &< X \times (40 - 20) \\ \frac{2.4}{20} &< X \\ 0.12 &< X \end{aligned}$$

Problem 2 – Instruction Set Design I .... 15 Points

2a) The MIPS instruction set provides two instructions for loading a byte into a given register? These instructions are:
load byte unsigned (e.g., `LBU r1, 60(r8)`) and load byte (e.g., `LB, r1, 60(r8)`). How do these instructions differ in their execution? Assume the memory location at address 60+r8 contains the byte 0x1F. What are the contents of r1 after `LBU r1, 60(r8)`? What are the contents of r1 after `LB r1, 60(r8)`?

LBU pads the would-be empty bit positions in a register with zeros. However, LB pads such bit positions with the sign bit of the data being stored in the register.

Contents after `LBU r1, 60(r8)` *are: 0x 0000 001F*
Contents after `LB r1, 60(r8)` *are: 0x 0000 001F.*

Note that 0x1F equals $(0001\ 1111)_2$*. From this hex to binary conversion, we can see that the sign bit, i.e., the leftmost bit, is 0. Therefore the LB instruction will pad the would-be empty bit positions with 0.*

2b) In the MIPS pipeline, the actions taken in the IF and ID stages do not depend on the actual instruction. What two features of the MIPS ISA enable this? How?

The two features are ***fixed width instructions*** *and* ***fixed locations for source registers and the immediate.***

A fixed width instruction ensures that the IF unit knows exactly how many accesses to instruction cache are required to get one instruction into the IR. If instructions are variable width, as in Intel x86, then an instruction has to be partially decoded to determine how long the instruction is. Such partial decoding determines how many memory accesses the IF unit needs to perform to get the full instruction. ***So for the variable-width instruction case, the action of the IF unit depends on the instruction itself.***

Having fixed locations for source registers and the immediate means that the

ID unit can decode the instruction ***in parallel*** *with reading the source registers and the immediate value. If the instruction turns out to be an R-type, for example, the work done in reading the immediate value could be deemed* ***unnecessary*** *but* ***will not cause incorrect execution*** *of the instruction.*

The following explains what happens when the locations of source registers, for example, are not fixed. Assume that there is an architecture where the location of one source register is different for different instructions. That means that an instruction has to be decoded ***before*** *that particular source register can be read.* ***So for the non-fixed source register case, the action of the ID unit depends on the instruction itself.*** *The above argument also applies when the immediate location varies with instruction.*

Problem 3 – Instruction Set Design II .... 20 Points

3a) The MIPS ISA allows only 16-bit long immediate values. Does this mean that C programs that use constants that need more than 16 (but still less than 32) bits in binary representation cannot be compiled for the MIPS ISA? Explain briefly. Details of any instruction are not needed.

No.
The constants mentioned above can be loaded into registers using LUI instruction.

3b) Enforcing object alignment means that sometimes memory may be used inefficiently. Explain why? Why is it still common to enforce object alignment?

*The First Why: Enforcing object alignment means that a given object cannot be stored **starting** at any arbitrary address; the **starting** address must be a multiple of object size. This means that some memory locations will be intentionally left empty. This is inefficient. For example, assume that mem[0] is the only location currently occupied in a memory, and that you now want to store a word. Object alignment says that the earliest address where this word can be stored is mem[4]. This means that locations mem[1] to mem[3] will be left empty. This is an inefficient use of memory.*

The Second Why: Aligned objects can be retrieved with fewer memory accesses. Because the improvements in memory speeds over the past many years have been much slower than the improvements in memory size, it is sensible to try to minimize the number of memory accesses for retrieving a required object.

Problem 4 – Pipelining I .... 10 Points

This question examines your understanding of how, if at all, a compiler can avoid RAW hazards. Assume a 5-stage MIPS pipeline.

4a) Write down a sequence of two or three assembly language instructions to demonstrate a RAW hazard that a compiler can avoid. How does the compiler avoid the hazard?

A compiler can avoid a RAW hazard if it knows how many independent instructions to put ***before*** *the data dependent instruction so as to give the producer instruction enough time to produce the required data.* ***A very important point*** *to remember is that if the compiler cannot find sufficient independent instructions from within the given code,* ***it will use enough NOP instructions*** *to act as independent instructions.* ***The option to use a NOP instruction is always available to the compiler.***

Given the above discussion, the next page shows two tables. The first table shows two example sequences for a pipeline that does not use forwarding. The next table shows three example sequences for a pipeline that uses forwarding.

In the tables, a three-dot sequence has been used to denote a register whose value is of no importance to this discussion.

Table 1: Two example sequences for a pipeline that does not use forwarding

original sequence	*compiler corrected sequence*	*explanation*
add r1, ···, ··· *sub ···, r1, ···*	*add r1, ···, ···* *NOP* *NOP* *sub ···, r1, ···*	*two NOPs will cause enough delay for add to produce* ***r1*** *by the time sub instruction needs it*
lw r1, ···, ··· *sub ···, r1, ···*	*lw r1, ···, ···* *NOP* *NOP* *sub ···, r1, ···*	*two NOPs will cause enough delay for lw to produce* ***r1*** *by the time sub instruction needs it*

Table 2: Three example sequences for a pipeline that uses forwarding

original sequence	*compiler corrected sequence*	*explanation*
add r1, ···, ··· *sub ···, r1, ···*	*add r1, ···, ···* *sub ···, r1, ···*	*no delay needed; the add* ***result*** *will be* ***forwarded*** *by the time sub instruction needs it*
lw r1, ···, ··· *sub ···, r1, ···*	*lw r1, ···, ···* *NOP* *sub ···, r1, ···*	***perfect cache****: one NOP will cause enough delay for lw to* ***forward its result*** *by the time sub instruction needs it*
lw r1, ···, ··· *sub ···, r1, ···*	*lw r1, ···, ···* *NOP* ⋮ *NOP* *sub ···, r1, ···*	***imperfect cache****: twenty NOPs will cause enough delay for lw to* ***forward its result*** *by the time sub instruction needs it*

4b) Write down a sequence of two or three assembly language instructions to demonstrate a RAW hazard that a compiler *cannot* avoid. Why not?

A compiler CANNOT avoid a RAW hazard if it does NOT know how many independent instructions (which can be NOPs) to put before the data dependent instruction so as to give the producer instruction enough time to produce the required data.

One such example is of a RAW hazard ***on a memory location****.*

sw 20(r3), ···
lw ···, 400(r7)

In the sequence above, ***if*** *the address (20+r3) equals address (400+r7),* ***then*** *we have a RAW hazard on the memory location mem[20+r3]. This is because mem[20+r3] is being written by the store instruction and then being* ***later*** *read by the load instruction, thereby creating a Read-After-Write haz-*

ard. Because it is not known whether (20+r3) equals (400+r7) ***until the EXE stages of these two instructions have been completed****, the compiler has no way of knowing if a RAW hazard exists. Please remember that a compiler's role in computer architecture finishes once it compiles a C program into assembly language, which is well before the HW starts executing the assembled instructions.*

Problem 5 – Pipelining II.................................20 Points

Examine the code sequence below. Assume a 5-stage MIPS pipeline. Use the blank columns in the table below as you wish.

#	***instruction***	no forwarding	forwarding	
1	lw r12, 4(r8)	5	5	
2	lw r13, 2(r8)	6	6	
3	add r1, r12, r13	**2**+7	**1**+7	
4	sw 4(r27), r1	**2**+10	9	
5	lw r15, 20(r8)	13	10	
6	lw r6, 4(r29)	14	11	
7	add r4, r15, r6	**2**+15	**1**+12	
8	sw 12(r27), r4	**2**+18	14	

5a) Assuming that each instruction takes 5 clock cycles to execute, determine the total number of clock cycles to execute this program for the *unpipelined case.*

40 clock cycles. 5 cycles for each of the 8 instructions.

5b) Repeat (a) for the pipelined case where there is no forwarding between the pipeline stages. However the register file can be read and written in the same cycle.

20 clock cycles. See the column labeled "no forwarding" in the table above. A two cycle stall is inserted for each data dependent instruction. Note that we are assuming a perfect data cache here. If the cache were not perfect, we will need more than 2 stalls for a load-to-ALU RAW hazard.

5c) Repeat (b) for the case of forwarding-enabled pipeline. Assume that the pipeline employs hazard detection logic that stalls it for one cycle after a load if the following instruction is dependent on load.

14 clock cycles. See the column labeled "forwarding" in the table above. All ALU-to-ALU RAW hazards are resolved by forwarding of results between the *pipeline registers. However, a single cycle stall is* ***still needed*** *for each load-to-ALU RAW hazard. Note that we are assuming a perfect data cache here. If the cache were not perfect, we will need more than 1 stall for a load-to-ALU RAW hazard.*

5d) The compiler has been charged with the task of producing an optimized version of the above code on a machine with a *forwarding-enabled* pipeline. Assume that the compiler does not know the value of r8, but that it knows that r29 = 200_{10} and r27 = 400_{10}. Write the best re-ordered version the compiler can produce. Determine the total number of clock cycles to execute this re-ordered code.

We can see from the "forwarding" column in the table above that there are two stalls that we can ***potentially*** *remove by finding two independent instructions from within the code. One such stall is for instruction number 3, and the other is for instruction number 7.*

To remove the stall for instruction number 3, we can move instruction number 6, "lw r6, 4(r29)," right before "add r1, r12, r13."

It seems like we have a choice of just as easily moving the instruction "lw r15, 20(r8)" (instead of "lw r6, 4(r29)") before "add r1, r12, r13." However, it will be a mistake to do that for the following reason. It is given that the compiler does not know the value of r8. Therefore, there is a possibility that the address 20+r8 equals the address 4+r27 (which is the address from the first sw instruction). That means there is a potential RAW hazard on mem[4+r27] between instructions:

sw 4(r27), r1
lw r15, 20(r8)

This, in turn, means that "lw r15, 20(r8)" ***cannot be moved to any slot before*** *"sw 4(r27), r1."*

It is ***not*** *a mistake to move "lw r6, 4(r29)" to* ***some slot before*** *"sw 4(r27), r1" because the compiler knows that 4+r29 $\neq$ 4+r27. Note that the question does not explain how the compiler knows that 4+r29 $\neq$ 4+r27, but that obviously does not concern us.*

Problem 6 – Pipelining III .... 20 Points

Examine the code sequence shown below. Assume a 5-stage MIPS pipeline. Note that LD, SD, DADDI and DSUB are just the “double-word” versions of LW, SW, ADDI and SUB, respectively.

```
Loop: LD      R1,0(R2)     ; load R1 from address 0+R2
      DADDI   RI,R1,#1     ; R1=R1+1
      SD      0(R2),RI     ; store R1 at address 0+R2
      DADDI   R2,R2,#4     ; R2=R2+4
      DSUB    R4,R3,R2     ; R4=R3-R2
      BNEZ    R4,Loop      ; branch to Loop if R4!=0
```

6a) Assuming that the pipeline employs forwarding and the branches are resolved at the end of ID, *draw the pipeline timing diagram for this code.* Use Figure 1 on the next page.

Please see Figure 1. Note that correct sequence of the first two stalls is:

IF	*stall*	*ID*	*EXE*	*MEM*	*WB*

and not

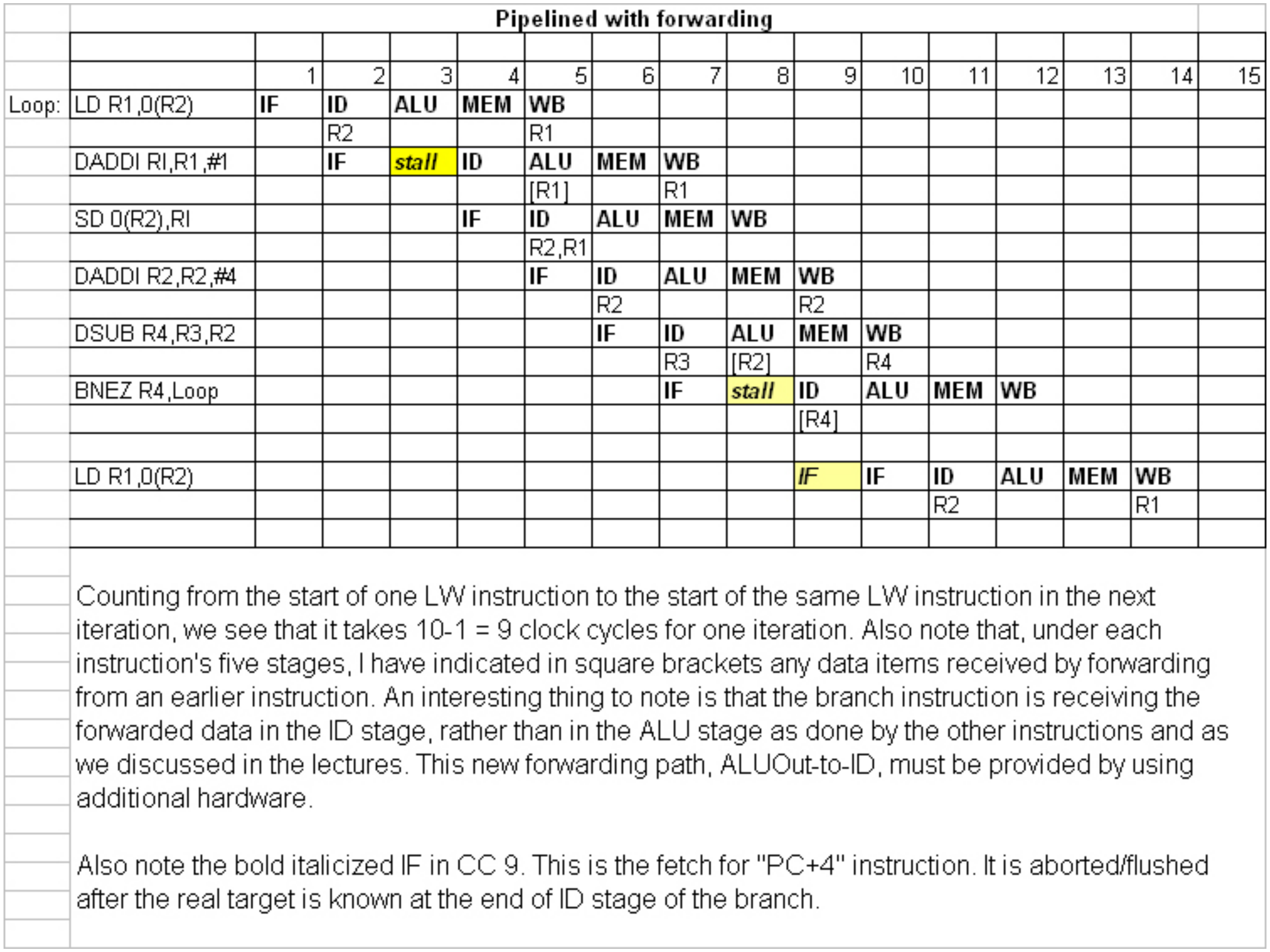
stall	*IF*	*ID*	*EXE*	*MEM*	*WB*

6b) With the same assumptions as in part (a), reorder the instructions to minimize the number of clock cycles it will take to execute one iteration of this loop. Assume a delayed branch architecture. You can modify the offset values if that helps in re-ordering. *Draw the pipeline timing diagram for the re-ordered code.* Use Figure 2.

Please see Figure 2.

6c) It is already understood that ***two forwarding paths*** exist from ***MEM / WB to ID/EXE*** and from ***EXE/MEM to ID/EXE***. If you used any additional forwarding paths for part (b), please state them here.

Yes, one additional forwarding path is needed from EXE/MEM to IF/ID to forward the R4 value to the branch instruction's ID stage.



Pipelined with forwarding

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Loop:	LD R1,0(R2)	**IF**	**ID**	**ALU**	**MEM**	**WB**										
			R2			R1										
	DADDI RI,R1,#1		**IF**	***stall***	**ID**	**ALU**	**MEM**	**WB**								
						[R1]		R1								
	SD 0(R2),RI				**IF**	**ID**	**ALU**	**MEM**	**WB**							
						R2,R1										
	DADDI R2,R2,#4					**IF**	**ID**	**ALU**	**MEM**	**WB**						
							R2			R2						
	DSUB R4,R3,R2						**IF**	**ID**	**ALU**	**MEM**	**WB**					
								R3	[R2]		R4					
	BNEZ R4,Loop							**IF**	***stall***	**ID**	**ALU**	**MEM**	**WB**			
										[R4]						
	LD R1,0(R2)									***IF***	**IF**	**ID**	**ALU**	**MEM**	**WB**	
												R2			R1	

Counting from the start of one LW instruction to the start of the same LW instruction in the next iteration, we see that it takes 10-1 = 9 clock cycles for one iteration. Also note that, under each instruction's five stages, I have indicated in square brackets any data items received by forwarding from an earlier instruction. An interesting thing to note is that the branch instruction is receiving the forwarded data in the ID stage, rather than in the ALU stage as done by the other instructions and as we discussed in the lectures. This new forwarding path, ALUOut-to-ID, must be provided by using additional hardware.

Also note the bold italicized IF in CC 9. This is the fetch for "PC+4" instruction. It is aborted/flushed after the real target is known at the end of ID stage of the branch.

Figure 1: Use this figure to answer question 6(a).

Re-orderd, Pipelined with forwarding

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Loop:	LD R1,0(R2)	**IF**	**ID**	**ALU**	**MEM**	**WB**										
			R2			R1										
	DADDI **R2**,R2,#4		**IF**	**ID**	**ALU**	**MEM**	**WB**									
				R2			R2									
	DSUB ***R4***,R3,**R2**			**IF**	**ID**	**ALU**	**MEM**	**WB**								
					R3	[R2]		R4								
	DADDI **R1**,R1,#1				**IF**	**ID**	**ALU**	**MEM**	**WB**							
						R1			R1							
	BNEZ ***R4***,Loop					**IF**	**ID**	**ALU**	**MEM**	**WB**						
							[R4]									
	SD -4(R2),**R1**						**IF**	**ID**	**ALU**	**MEM**	**WB**					
								R2	[R1]							
	LD R1,0(R2)							**IF**	**ID**	**ALU**	**MEM**	**WB**				
									R2			R1				

All stalls are eliminated using the re-ordered code shown above. Counting from the start of one LW instruction to the start of the same LW instruction in the next iteration, we see that it takes 7-1 = 6 clock cycles for one iteration. Also note that, under each instruction's five stages, I have indicated in square brackets any data items received by forwarding from an earlier instruction. An interesting thing to note is that the branch instruction is receiving the forwarded data in the ID stage, rather than in the ALU stage as done by the other instructions and as we discussed in the lectures. This new forwarding path, ALUOut-to-ID, must be provided by using additional hardware.

Figure 2: Use this figure to answer question 6(b).