

CSE 240A Homework One

April 5, 2011

*Only **Problem Set Two** will be graded. Turn in only **Problem Set Two** which will be due on April 25, 2011 (Monday) at 3:00pm.*

1 Problem Set One

- Hennessy & Patterson (4th Ed) 1.2
- Hennessy & Patterson (4th Ed) 1.5
- Hennessy & Patterson (4th Ed) 1.7
- Hennessy & Patterson (4th Ed) 1.9
- Hennessy & Patterson (4th Ed) 1.10
- Hennessy & Patterson (4th Ed) 1.13
- Hennessy & Patterson (4th Ed) 1.14
- And Attached 5 Appendix A problems

2 Problem Set Two

1 (Amdahl's law)

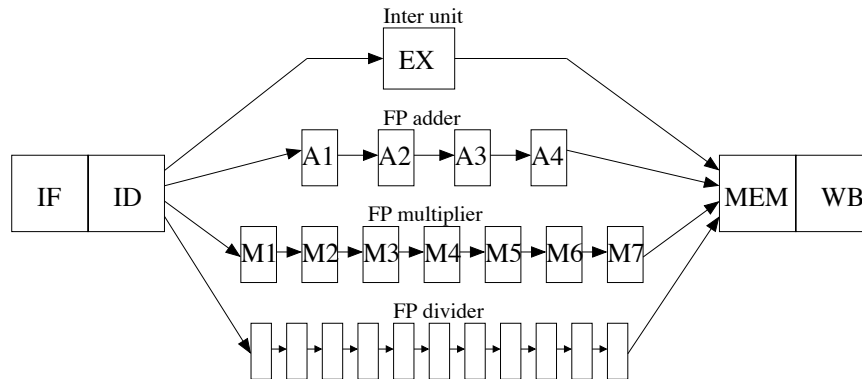
A company is considering modifications to its processor architecture. The modifications that are being considered, unlike the simple formulation of Amdahl's law, have multiple ramifications to the three major components of the processor. The ramifications are outlined in the following table, in which each entry denotes the factor by which the execution time of that component in the column header will be affected.

Modifications	Component X	Component Y	Component Z
A	0.2	1.2	1.0
B	1.0	0.5	1.2
C	1.3	1.0	0.8
D	0.7	1.1	1.2

(Part A) If the fractions of the total execution time for the three components, X, Y, and Z are 30%, 45%, and 25%, respectively, please identify the speedups (or possibly performance degradations) that are to be expected from each of these 4 modifications. Please rank the four modifications in terms of speedup.

(Part B) Suppose it is NOT known what percent of the overall execution time is derived from each of the three pieces. What is known is that the 4 modifications result in a speedup ranking of A, B, C, D, with A being the fastest. Please identify the functional relationships in the component execution time percentages (the sum of which should of course add to 1 as in Part A), for which this particular ranking holds. For partial credit, please provide a possible component execution time percentage, for which the aforementioned ranking holds.

2 (Structural Hazards in Floating-Point Pipeline)



The figure above shows a floating point pipeline that can execute three types of floating point instructions, *add*, *mult*, and *div*. Addition takes 4 cycles to finish, while multiply takes 7 and divide takes 11. More interestingly, during execution all three instructions need to use **two** pieces of common hardware, namely, *X* and *Y*, thus giving rise to possible structural hazards. The execution stages during which these two common hardware components are utilized for each instruction is given below, with “x” denoting the use of component *X*, “y” the use of component *Y*, and “0” *no use* of either *X* or *Y* in that stage.

Add: 0x0y
 Mult: x0y0y0x
 Div: xy00xy00x0y

In this pipeline design, the conflict resolution for the possible structural hazard is performed once and for all at the **ID** stage. In other words, if the instruction at the ID stage conflicts with an instruction in execution, it is blocked until the cycle at which it can be issued with **no** conflict. Once the execution starts, the instruction does not stall.

(Part A) A technique was outlined in your book for dealing with the structural hazards of register write in the MIPS floating point pipeline. Please first briefly *describe* that technique, then furthermore *generalize* it so as to resolve the aforementioned resource conflicts. More specifically, you need to address the type and the number of hardware component(s) that need to be employed, as well as the approach for handling structural hazards using such component(s).

(Part B) The instructions listed in the following cases are assumed to be *data independent*. For each case, please identify the minimum latency (in terms of clock cycles) induced by the aforementioned *structural hazards* between any two adjacent instructions.

1. *add* followed by *multiply*

2. *multiply* followed by *divide*

3. *add* followed by *divide*

4. *divide* followed by *add*

5. *multiply* followed by *add* followed by *multiply*

6. *divide* followed by *add* followed by *multiply*

(Part C) Noticing that structural hazards have caused a large number of stalls between adjacent floating-point instructions, one engineer in the design team has proposed to insert a *no-op* stage (which obviously uses neither the *X* nor the *Y* component), during the execution of *mult* so as to reduce the impact of structural hazards. The engineer, however, is not quite confident of being able to persuade the manager to adopt this idea, since this *no-op* will increase the execution time of *multiply* instructions by 1 cycle. Please help the engineer to evaluate her idea, by determining whether inserting a *no-op* is helpful in reducing the number of stalls caused by structural hazard. If you think it is **not** helpful, please state the reasons; if you think it is helpful, please show the position where the *no-op* should be inserted, and furthermore show instruction sequences that can benefit from this *no-op*.

3 (Scoreboard Hazards Handling)

This question examines the basic concepts of the scoreboard technique. Recall that in a scoreboard processor, instructions are issued *in order* yet executed *out of order*. This execution strategy requires three types of information to be recorded: *instruction status*, *functional unit status*, and *register result status*.

(Part A) This part of the question considers the *register result status*, used to record the functional unit that will write each register, such as the table shown below.

Table 1: **Register result status**

	F0	F2	F4	F6	... F30
FU	<i>Mult1</i>	<i>Int2</i>		<i>Add1</i>	

As you may have noticed, in scoreboard at any time there exists at most **one** issued instruction that can write to a particular register file. Motivated by this observation, one engineer in the design team has suggested that instead of recording the functional unit names, the *register result status* table only use, for each register, a **single** bit to indicate whether there exists an issued instruction that has the register as its destination. Such an instruction, once it finishes execution, obviously knows where to write anyway (through the **Fi** field in the *functional unit status* table). The company's attempts to use this idea resulted eventually in this poor engineer getting fired. What is the reason? Why do we need to record functional unit names in the *register result status* table?

(Part B) This part of the question considers the *functional unit status* table in a scoreboard processor. Recall that this table uses two fields **Qj** and **Qk** to keep record as to which functional units will produce the source registers **Fj** and **Fk**, as well as another two fields, **Rj** and **Rk**, to indicate whether **Fj** and **Fk** are ready or not. However, since in a scoreboard the operands are fetched from the register file directly, it would seem that only using the **Rj** and **Rk** fields would suffice.

You have been provided three instructions, *add*, *mult*, *div*, as well as four possible instruction sequences, listed in the following table.

Sequence A	Sequence B	Sequence C	Sequence D	Sequence E	Sequence F
<i>mult</i>	<i>mult</i>	<i>div</i>	<i>add</i>	<i>div</i>	<i>add</i>
<i>add</i>	<i>div</i>	<i>mult</i>	<i>div</i>	<i>add</i>	<i>mult</i>
<i>div</i>	<i>add</i>	<i>add</i>	<i>mult</i>	<i>mult</i>	<i>div</i>

The following tables correspond to a state wherein the *div* operation is somewhere in progress, while both the *add* and the *mult* operation are known to have just finished execution and are about to enter the WB stage to write results. As you may have noticed, none of the *functional unit status* tables listed below has any **Qj** or **Qk** fields. You are asked to identify for each case, **all** the possible instruction sequence(s) that make the table *valid*. Please give a brief explanation for each case.

<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Rj</i>	<i>Rk</i>
<i>Add</i>	yes	ADD	F6	F8	F2	no	no
<i>Mult</i>	yes	MUL	F0	F2	F4	no	no
<i>Div</i>	yes	DIV	F10	F0	F6	no	yes

The valid instruction sequence(s) are:

<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Rj</i>	<i>Rk</i>
<i>Add</i>	yes	ADD	F6	F8	F2	no	no
<i>Mult</i>	yes	MUL	F0	F2	F4	no	no
<i>Div</i>	yes	DIV	F10	F0	F6	no	no

The valid instruction sequence(s) are:

(Part C) In hardware, either of the **Rj** and **Rk** fields in the *functional unit status* table is represented by a single bit. However, one engineer has proposed to extend each field to **two** bits that can represent the following three states:

- *wait* – the instruction is waiting for an operand to be produced;
- *ready* – the operand is ready in the register file, but has not been read yet.
- *done* – the instruction has already read its operand.

The engineer claims that by extending the **Rj** and **Rk** fields, the corresponding instruction sequence can be solely determined without knowing the **Qj** and **Qk** fields.

Again, the following tables correspond to an execution cycle at which *div* is still in process, while both *add* and *mult* have just finished execution and are about to enter the WB stage to write results. You are asked to identify for each case, whether the status shown in the table is a valid status in the scoreboard or not. If you think it is *invalid*, please provide a brief reasoning. On the other hand if you think the status is *valid*, please provide **all** the possible instruction sequence(s) (among the group shown in Part (B)) that make the table *valid*.

<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Rj</i>	<i>Rk</i>
<i>Add</i>	yes	ADD	F6	F8	F2	<i>done</i>	<i>done</i>
<i>Mult</i>	yes	MUL	F0	F2	F4	<i>done</i>	<i>done</i>
<i>Div</i>	yes	DIV	F10	F0	F6	ready	wait

<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Rj</i>	<i>Rk</i>
<i>Add</i>	yes	ADD	F6	F8	F2	<i>done</i>	<i>done</i>
<i>Mult</i>	yes	MUL	F0	F2	F4	<i>done</i>	<i>done</i>
<i>Div</i>	yes	DIV	F10	F0	F6	ready	done

<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Rj</i>	<i>Rk</i>
<i>Add</i>	yes	ADD	F6	F8	F2	<i>done</i>	<i>done</i>
<i>Mult</i>	yes	MUL	F0	F2	F4	<i>done</i>	<i>done</i>
<i>Div</i>	yes	DIV	F10	F0	F6	done	done

A.1 [15/15/15] <A.2> Use the following code fragment:

```
Loop: LD      R1,0(R2)      ;load R1 from address 0+R2
      DADDI   R1,R1,#1      ;R1=R1+1
      SD      0(R2),R1      ;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
```

Assume that the initial value of R3 is R2 + 396.

Throughout this exercise use the classic RISC five-stage integer pipeline (see Figure A.1) and assume all memory accesses take 1 clock cycle.

- [15] <A.2> Show the timing of this instruction sequence for the RISC pipeline *without* any forwarding or bypassing hardware but assuming a register read and a write in the same clock cycle “forwards” through the register file, as in Figure A.6. Use a pipeline timing chart like Figure A.6. Assume that the branch is handled by flushing the pipeline. If all memory references take 1 cycle, how many cycles does this loop take to execute?
- [15] <A.2> Show the timing of this instruction sequence for the RISC pipeline with normal forwarding and bypassing hardware. Use a pipeline timing chart like Figure A.6. Assume that the branch is handled by predicting it as not taken. If all memory references take 1 cycle, how many cycles does this loop take to execute?
- [15] <A.2> Assume the RISC pipeline with a single-cycle delayed branch and normal forwarding and bypassing hardware. Schedule the instructions in the loop including the branch delay slot. You may reorder instructions and modify the individual instruction operands, but do not undertake other loop transformations that change the number or opcode of the instructions in the loop (that’s for Chapter 4!). Show a pipeline timing diagram and compute the number of cycles needed to execute the entire loop.

A.5 [12/13/20/20/15/15] <A.2, A.3> For these problems, we will explore a pipeline for a register-memory architecture. The architecture has two instruction formats: a register-register format and a register-memory format. There is a single-memory addressing mode (offset + base register).

There is a set of ALU operations with format

ALUop Rdest, Rsrc₁, Rsrc₂

or

ALUop Rdest, Rsrc₁, MEM

where the ALUop is one of the following: Add, Subtract, And, Or, Load (Rsrc₁ ignored), Store. Rsrc or Rdest are registers. MEM is a base register and offset pair.

Branches use a full compare of two registers and are PC-relative. Assume that this machine is pipelined so that a new instruction is started every clock cycle. The following pipeline structure—similar to that used in the VAX 8700 micropipeline [Clark 1987]—is

IF	RF	ALU1	MEM	ALU2	WB					
IF		RF	ALU1	MEM	ALU2	WB				
		IF	RF	ALU1	MEM	ALU2	WB			
			IF	RF	ALU1	MEM	ALU2	WB		
				IF	RF	ALU1	MEM	ALU2	WB	
					IF	RF	ALU1	MEM	ALU2	WB

The first ALU stage is used for effective address calculation for memory references and branches. The second ALU cycle is used for operations and branch comparison. RF is both a decode and register-fetch cycle. Assume that when a register read and a register write of the same register occur in the same clock the write data is forwarded.

- [12] <A.2> Find the number of adders needed, counting any adder or incrementer; show a combination of instructions and pipe stages that justify this answer. You need only give one combination that maximizes the adder count.
- [13] <A.2> Find the number of register read and write ports and memory read and write ports required. Show that your answer is correct by showing a combination of instructions and pipeline stage indicating the instruction and the number of read ports and write ports required for that instruction.
- [20] <A.4> Determine any *data forwarding* for any ALUs that will be needed. Assume that there are separate ALUs for the ALU1 and ALU2 pipe stages. Put in all forwarding among ALUs needed to avoid or reduce stalls. Show the relationship between the two instructions involved in forwarding using the format of the table in Figure A.22 but ignoring the last two columns. Be careful to consider forwarding across an intervening instruction, for example,

ADD R1, ...

any instruction

ADD ..., R1, ...

- [20] <A.3> Show all data forwarding requirements needed to avoid or reduce stalls when either the source or destination unit is not an ALU. Use the same format as Figure A.22, again ignoring the last two columns. Remember to forward to and from memory references.
- [15] <A.3> Show all the remaining hazards that involve at least one unit other than an ALU as the source or destination unit. Use a table like that in Figure A.21, but listing the length of hazard in place of the last column.

- f. [15] <A.2> Show all control hazard types by example and state the length of the stall. Use a format like Figure A.11, labeling each example.

A.6 [12/13/13/15/15] <A.1, A.2, A.3> We will now add support for register-memory ALU operations to the classic five-stage RISC pipeline. To offset this increase in complexity, *all* memory addressing will be restricted to register indirect (i.e., all addresses are simply a value held in a register; no offset or displacement may be added to the register value). For example, the register-memory instruction ADD R4, R5, (R1) means add the contents of register R5 to the contents of the memory location with address equal to the value in register R1 and put the sum in register R4. Register-register ALU operations are unchanged. Answer the following for the integer RISC pipeline.

- [12] <A.1> List a rearranged order of the five traditional stages of the RISC pipeline that will support register-memory operations implemented exclusively by register indirect addressing.
- [13] <A.2, A.3> Describe what new forwarding paths are needed for the rearranged pipeline by stating the source, destination, and information transferred on each needed new path.
- [13] <A.2, A.3> For the reordered stages of the RISC pipeline, what new data hazards are created by this addressing mode? Give an instruction sequence illustrating each new hazard.
- [15] <A.3> List all ways that the RISC pipeline with register-memory ALU operations can have a different instruction count for a given program than the original RISC pipeline. Give a pair of specific instruction sequences, one for the original pipeline and one for the rearranged pipeline, to illustrate each way.
- [15] <A.3> Assume all instructions take 1 clock cycle per stage. List all ways that the register-memory RISC can have a different CPI for a given program as compared to the original RISC pipeline.

A.7 [10/15/15] <A.2> Scheduling branch delay slots (see the three ways in Figure A.14) can improve performance. Assume a single branch delay slot and an instruction execution pipeline that determines branch outcome in the second stage.

- [10] <A.2> For a delayed branch instruction, what is the penalty for each branch delay slot scheduling scheme if the branch is taken and if it is not taken, and what condition, if any, must be satisfied to ensure correct execution?
- [15] <A.2> A cancel-if-not-taken branch instruction (also called *branch likely* and implemented in MIPS) does not execute the instruction in the delay slot if the branch is not taken. Thus, a compiler need not be as conservative when filling the delay slot. For each branch delay slot scheduling scheme, what is the penalty if the branch is taken and if it is not taken, and what condition, if any, must be satisfied to ensure correct execution?

- [15] <A.2> Assume that an instruction set has a delayed branch and a cancel-if-not-taken branch. When should a compiler use each branch instruction and from where should the slot be filled?

A.13 [25] <A.8> It is critical that the scoreboard be able to distinguish RAW and WAR hazards, because a WAR hazard requires stalling the instruction doing the writing until the instruction reading an operand initiates execution, but a RAW hazard requires delaying the reading instruction until the writing instruction finishes—just the opposite. For example, consider the sequence

```
MUL.D    F0,F6,F4
DSUB.D   F8,F0,F2
ADD.D    F2,F10,F2
```

The DSUB.D depends on the MUL.D (a RAW hazard) and thus the MUL.D must be allowed to complete before the DSUB.D; if the MUL.D were stalled for the DSUB.D due to the inability to distinguish between RAW and WAR hazards, the processor will deadlock. This sequence contains a WAR hazard between the ADD.D and the DSUB.D, and the ADD.D cannot be allowed to complete until the DSUB.D begins execution. The difficulty lies in distinguishing the RAW hazard between MUL.D and DSUB.D, and the WAR hazard between the DSUB.D and ADD.D.

To see just why the three-instruction scenario is important, trace the handling of each instruction stage by stage through Issue, Read Operands, Execute, and Write

Instruction producing result	Instruction using result	Latency in clock cycles
FP multiply	FP ALU op	6
FP add	FP ALU op	4
FP multiply	FP store	5
FP add	FP store	3
Integer operation (including load)	Any	0

Figure A.56 Pipeline latencies where latency is number of cycles between producing and consuming instruction.

Result. Assume that each scoreboard stage other than Execute takes 1 clock cycle. Assume that the MUL.D instruction requires 3 clock cycles to execute and that the DSUB.D and ADD.D instructions each take 1 cycle to execute. Finally, assume the processor has two multiply function units and two add function units. Present the trace as follows.

Step 1. Make a table with the column headings Instruction, Issue, Read Operands, Execute, Write Result, and Comment. In the first column, list the instructions in program order (be generous with space between instructions; larger table cells will better hold the results of your analysis). Start the table by writing a 1 in the Issue column of the MUL.D instruction row to show that MUL.D completes the Issue stage in clock cycle 1. Now, fill in the stage columns of the table through the cycle at which the scoreboard first stalls an instruction.

Step 2. For a stalled instruction write the words “waiting at clock cycle X,” where X is the number of the current clock cycle, in the appropriate table column to show that the scoreboard is resolving an RAW or WAR hazard by stalling that stage. In the Comment column, state what type of hazard and what dependent instruction is causing the wait.

Step 3. Adding the words “completes with clock cycle Y” to a “waiting” table entry, fill in the rest of the table through the time when all instructions are complete. For an instruction that stalled, add a description in the Comments column telling why the wait ended when it did and how deadlock was avoided. (*Hint:* Think about how WAW hazards are prevented and what this implies about active instruction sequences.) Note the completion order of the three instructions as compared to their program order.