CS232 Midterm Exam 2
April 8, 2002

Name: Frodo Baggins

- This exam has 7 pages, including this cover.
- There are three questions worth a total of 100 points.
- You have 50 minutes. Budget your time!
- No written references or calculators are allowed.
- To make sure you receive full credit, write clearly and show your work.
- We will not answer questions regarding course material.

Question	Maximum	Your Score
1	30	
2	35	
3	35	
Total	100	

Question 1: Single-cycle datapath (30 points)

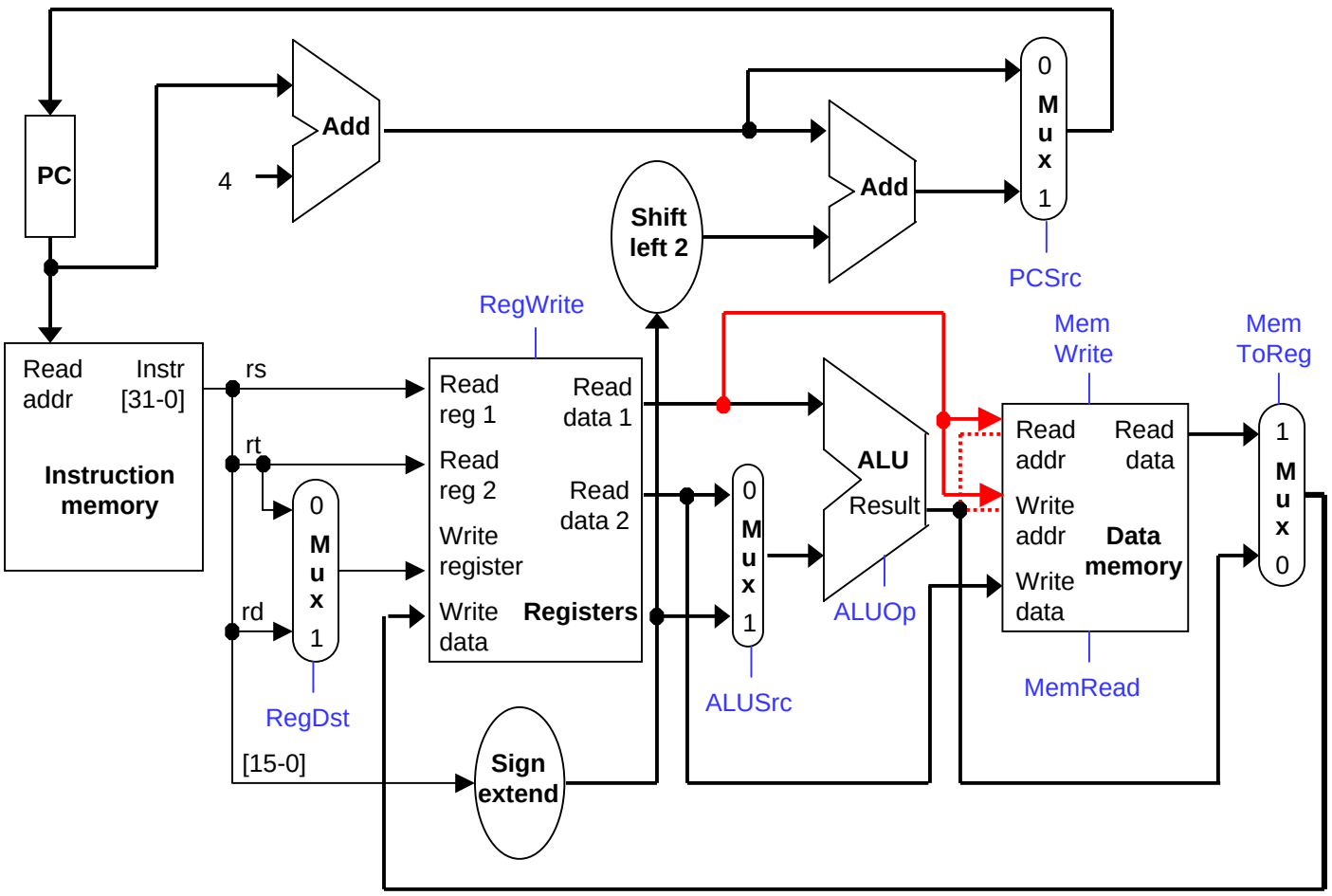
Let's simplify the MIPS instruction set architecture a little by *removing* the original lw and sw instructions and replacing them with ones that do not contain a constant offset. Our new loads and stores will have the following general forms.

```
lw  rt, rs      # rt = Mem[rs]
sw  rt, rs      # Mem[rs] = rt
```

These are I-type instructions, but you may *not* make any assumptions about the instructions' constant fields.

Part (a)
Show what changes must be made to the single-cycle datapath below so the new lw and sw instructions can avoid going through the ALU. Please try to keep your modifications as neat as possible! (10 points)

Our modified diagram is shown below. Some people used multiplexers to select the memory addresses from either the ALU output or the register file, which is all right. But since the problem says that we are removing the original lw/sw instructions, we could completely eliminate the connections between the ALU and data memory. Instead, we just use Read Data 1 from the register file (representing register rs) as the memory's Read and Write address inputs.



Question 1 continued

Part (b)
Fill in the table below to show the correct control signals for the new lw and sw instructions. You must write 'X' to indicate any don't-care conditions. (5 points)

Since we're not using the ALU anymore, the related signals should be don't cares.

	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg	PCSrc
lw	*0*	*1*	*X*	*X*	*0*	*1*	*1*	*0*
sw	*X*	*0*	*X*	*X*	*1*	*0*	*X*	*0*

Part (c)
Assume that memories and the ALU have 2ns delays, and the registers have a 1ns delay. Find the minimum clock cycle times for *both* the original single-cycle datapath and your modified one. (5 points)

These are the same delays we used in some lecture examples. In the original ISA, lw would be the slowest instruction, requiring 8ns total (2ns for instruction fetch, 1ns for reading registers, 2ns for effective address computation, 2ns to read memory, and 1ns for writeback).

For our new ISA, lw can avoid the ALU step and complete in just 6ns instead. If you work through the other delays, you'll see that R-type instructions also require 6ns, while beq and sw need less time.

The delay of the longest instruction will determine the minimum clock cycle time, so that would be 8ns in the original system and 6ns with the modified lw/sw instructions.

Part (d)
What general conclusions can you draw, if any, about the performance of the original processor as compared to your modified one? (10 points)

In a single-cycle machine, each instruction takes one cycle to execute. By lowering the cycle time from 8ns to 6ns, it seems like performance will always improve. However, we shouldn't forget to think about the other factors that affect performance, such as the number of instructions executed.

A single lw instruction such as

```
lw    $t0, 4($sp)
```

in the original system would require 8ns to execute. In the new ISA though, this would have to be done with two *instructions, in a total of 12ns:*

```
addi  $sp, $sp, 4
lw    $t0, $sp
```

So if the original code contains many memory accesses with constant offsets, its equivalent code with the simplified instruction set may actually take more time!

Question 2: Multicycle CPU implementation (35 points)

MIPS is a register-register architecture, where arithmetic source and destinations must both be registers. But let's think about including a register-memory addition instruction.

addm rd, rs, rt # rd = rs + Mem[rt]

In other words, register rt contains a memory address which is read to produce the ALU's operand. The instruction format is given here for your reference (shamt and func are not used).

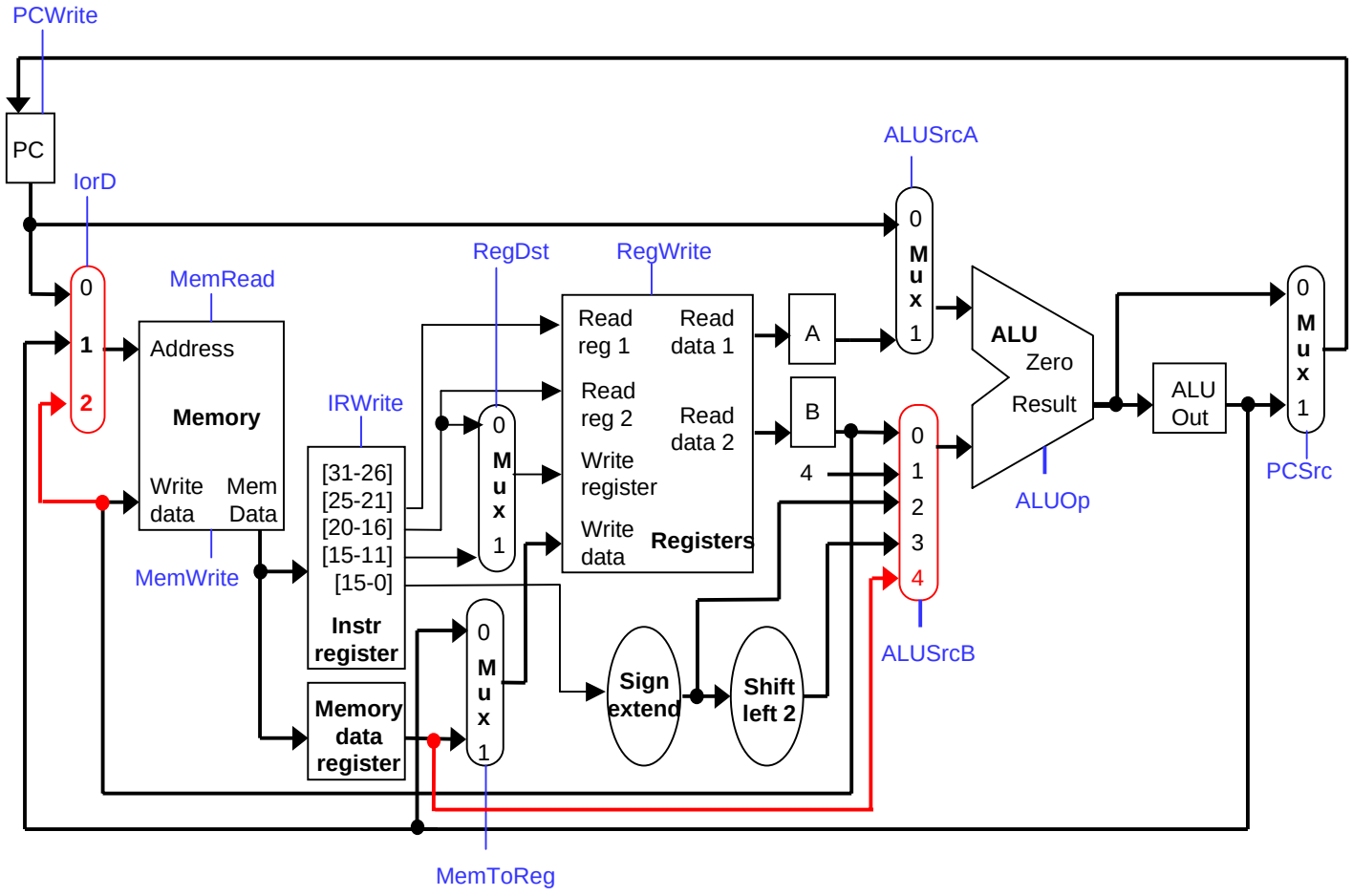
Field	op	rs	rt	rd	shamt	func
Bits	31-26	25-21	20-16	15-11	10-6	5-0

It is possible to include *addm* in a multicycle processor by modifying the datapath and control unit presented in class. Your implementation of *addm* should not need more than five stages for execution, and you should be careful not to modify any registers other than rd.

Part (a)

The multicycle datapath from lecture appears below. Show the changes needed to support *addm*. Try to keep your diagram neat! (10 points)

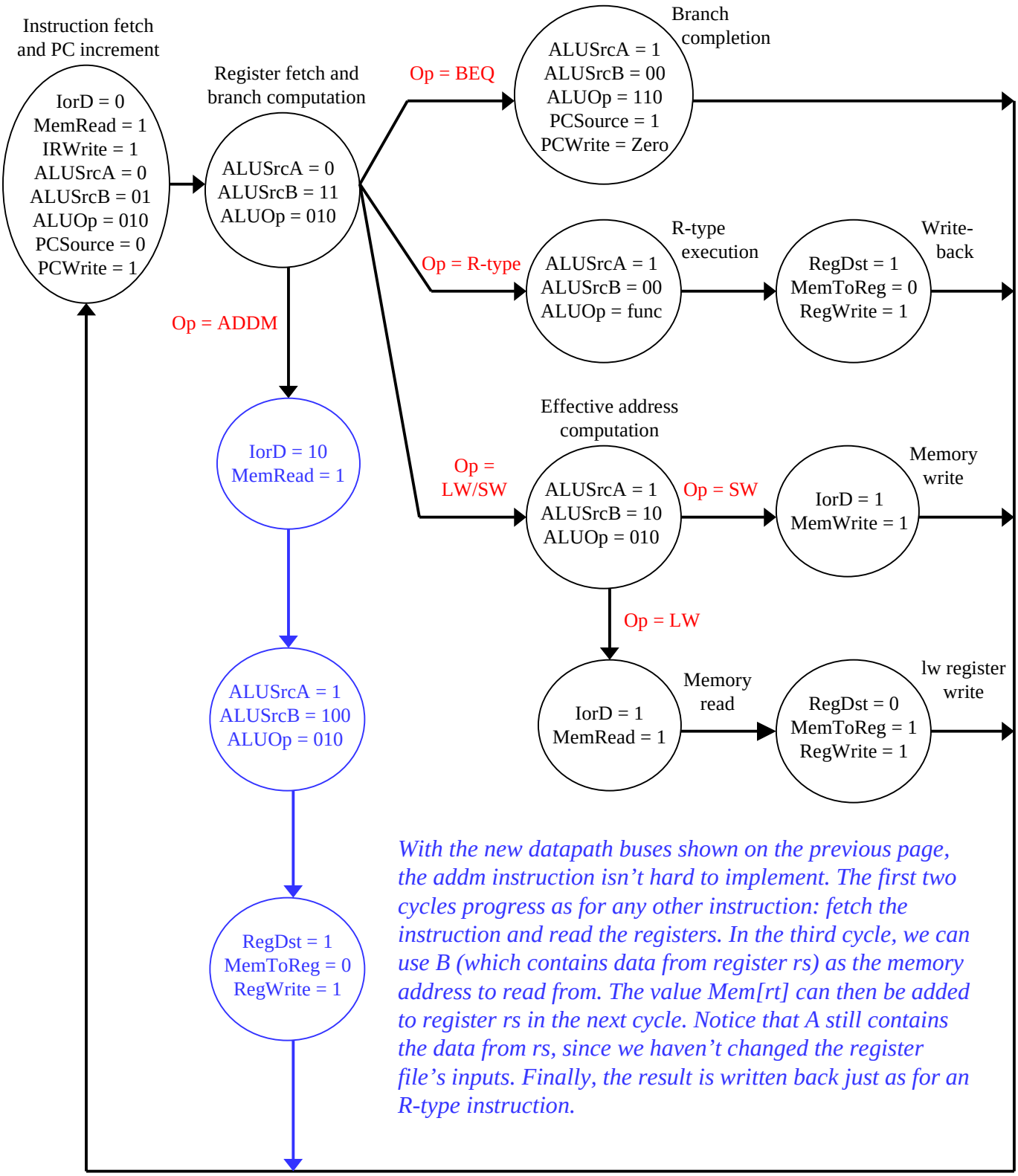
Here we've connected the intermediate register B to the memory unit so we can read from address rt. We also feed MDR (which will contain Mem[rt]) into the ALU, for addition with register rs. These are probably the simplest set of changes; the control unit on the next page shows the details of how to get this to work.



Question 2 continued

Part (b)

Complete this finite state machine diagram for the *addm* instruction. Control values not shown in each stage are assumed to be 0. Remember to account for any control signals that you added or modified in the previous part of the question. (25 points)



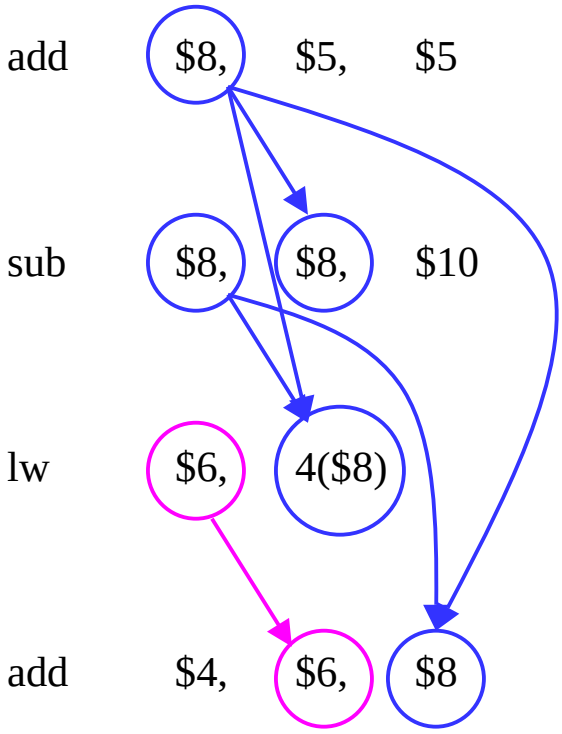
With the new datapath buses shown on the previous page, the addm instruction isn't hard to implement. The first two cycles progress as for any other instruction: fetch the instruction and read the registers. In the third cycle, we can use B (which contains data from register rs) as the memory address to read from. The value Mem[rt] can then be added to register rs in the next cycle. Notice that A still contains the data from rs, since we haven't changed the register file's inputs. Finally, the result is written back just as for an R-type instruction.

Question 3: Forwarding and stalling (35 points)

Part (a)
Show or list all of the dependencies in the following code fragment. Make sure that you clearly indicate which instructions *and* registers are involved in each dependency. (5 points)

Lots of dependencies here! Register $8 is read and written several times, as shown with the blue circles and arrows. There is also a dependency between the third and fourth instructions involving $6.

Note that $8 is written to in both the first and second instructions. We didn't talk about this kind of "write after write" dependency in class, so it's all write if you did or didn't show it.



Part (b)
The pipelined datapath on the next page shows the fifth cycle of executing this program. Fill in the correct datapath values for the *fifteen* question mark symbols **?** in the diagram. There is one **?** in the IF stage, three in the ID stage, eight in EX, two in MEM, and one in WB. (30 points; 2 points each)

- Assume that registers initially contain their number plus 100: $6 contains 106, $8 contains 108, etc.
- Write your values directly on the diagram, but please write clearly.
- Use decimal notation. You may write 'X' for any values that cannot be determined.

Since the datapath on the next page shows forwarding and hazard units as discussed in class, we can assume that dependencies between the first three instructions will be resolved correctly. For example, your answer should show the correct value of $8 being forwarded from the sub instruction in the MEM stage to the lw in the EX stage.

A load stall is required between the third and fourth instructions; the hazard unit should detect this situation and output the appropriate control signals to force a stall. Notice that the register file outputs in the ID stage are an incorrect, old value for $6 and an incorrect, new value for $8. (The first add instruction is writing 210 to $8 in this cycle, but the sub result isn't available yet.)

Question 3 continued