CS232 Final Exam
May 5, 2001

Name: Spiderman

- This exam has 14 pages, including this cover.
- There are six questions, worth a total of 150 points.
- You have 3 hours. Budget your time!
- *Write clearly and show your work.* State any assumptions you make.
- No written references or calculators are allowed.

Question	Maximum	Your Score
1	25	
2	25	
3	25	
4	25	
5	30	
6	20	
Total	150	

MIPS Instructions

These are some of the most common MIPS instructions and pseudo-instructions, and should be all you need. However, you are free to use *any* valid MIPS instructions or pseudo-instruction in your programs.

Category	Example	Meaning
Arithmetic	add rd, rs, rt	rd = rs + rt
	sub rd, rs, rt	rd = rs - rt
	addi rd, rs, const	rd = rs + const
	mul rd, rs, rt	rd = rs * rt
	move rd, rs	rd = rs
	li rd, const	rd = const
Data Transfer	lw rt, const(rs)	rt = Mem[const + rs] (one word)
	sw rt, const (rs)	Mem[const + rs] = rt (one word)
	lb rt, const (rs)	rt = Mem[const + rs] (one byte)
	sb rt, const (rs)	Mem[const + rs] = rt (one byte)
Branch	beq rs, rt, Label	if (rs == rt) go to Label
	bne rs, rt, Label	if (rs != rt) go to Label
	bge rs, rt, Label	if (rs >= rt) go to Label
	bgt rs, rt, Label	if (rs > rt) go to Label
	ble rs, rt, Label	if (rs <= rt) go to Label
	blt rs, rt, Label	if (rs < rt) go to Label
Set	slt rd, rs, rt	if (rs < rt) then rd = 1; else rd = 0
	slti rd, rs, const	if (rs < const) then rd = 1; else rd = 0
Jump	j Label	go to Label
	jr $ra	go to address in $ra
	jal Label	$ra = PC + 4; go to Label

The second source operand of *sub*, *mul*, and all the branch instructions may be a constant.

Register Conventions

The *caller* is responsible for saving any of the following registers that it needs, before invoking a function:

$t0-$t9 $a0-$a3 $v0-$v1

The *callee* is responsible for saving and restoring any of the following registers that it uses:

$s0-$s7 $ra

Performance

Formula for computing the CPU time of a program P running on a machine X:

$$\text{CPU time}_{X,P} = \text{Number of instructions executed}_P \times \text{CPI}_{X,P} \times \text{Clock cycle time}_X$$

CPI is the average number of clock cycles per instruction, or:

CPI = Number of cycles needed / Number of instructions executed

Question 1: MIPS programming

The goal of this problem is to write a MIPS function *flipimage* which flips an image horizontally. For example, a simple image is shown on the left, and its flip is shown on the right.

A picture is composed of individual dots, or pixels, each of which will be represented by a single byte. The entire two-dimensional image is then stored in memory row by row. For example, we can store a 4 x 6 picture in memory starting at address 1000 as follows:

- The first row (consisting of 6 pixels) is stored at addresses 1000-1005.
- The second row is stored at addresses 1006-1011.
- The third row is stored at 1012-1017
- The last row is stored at addresses 1018-1023.

Part (a)
Write a MIPS function *fliprow* to flip a *single* row of pixels. The function has two arguments, passed in $a0 and $a1: the address of the row and the number of pixels in that row. There is no return value. Be sure to follow all MIPS calling conventions. (10 points)

There are oodles of ways to write fliprow. The example solution here uses $a0 and $a2 as "pointers" to the beginning and end of the array. It repeatedly swaps the data at those addresses, and increments $a0 and decrements $a2 until they cross. Since fliprow doesn't invoke any other functions, we can use caller-saved registers like $a0-$a3, $t0-$t9 and $v0-$v1 without having to save them. Also note that you need to load and store bytes, *not words.*

```
fliprow:
        sub     $a1, $a1, 1             # Array indices start at 0
        add     $a2, $a0, $a1           # $a2 is address of last element

        bge     $a0, $a2, rowexit
        lb      $t0, 0($a0)
        lb      $t1, 0($a2)
        sb      $t0, 0($a2)
        sb      $t1, 0($a0)             # Swap two elements

        addi    $a0, $a0, 1
        sub     $a2, $a2, 1             # Go on to next pair of elements

rowexit:
        jr      $ra
```

Question 1 continued

Part (b)

Using the *fliprow* function, you should now be able to write *flipimage*. The arguments will be:

- The memory address where the image is stored
- The number of rows in the image
- The number of columns in the image

Again, there is no return value, and you should follow normal MIPS calling conventions. (15 points)

Unlike fliprow, flipimage is a nested function that acts as both a caller and a callee. You'll basically have to preserve every register that flipimage uses; caller-saved registers will have to be saved and restored before and after calling fliprow, while callee-saved registers must be restored before flipimage returns.

One more thing you'll have to be careful of is how the flipimage and fliprow arguments match up. Argument $a2 of flipimage is the number of columns, which must be passed to flipimage in register $a1.

```
flipimage:
        sub     $sp, $sp, 16
        sw      $ra, 0($sp)
        sw      $a0, 4($sp)         # Starting address of image
        sw      $a1, 8($sp)         # Number of rows
        sw      $a2, 12($sp)        # Number of columns

fliploop:
        lw      $a0, 4($sp)         # Get address of current row
        lw      $a1, 12($sp)        # Length of each row
        jal     fliprow

        lw      $a1, 8($sp)
        sub     $a1, $a1, 1
        sw      $a1, 8($sp)         # Update number of rows left
        beq     $a1, $0, flipexit   # Stop if no more rows

        lw      $a0, 4($sp)
        lw      $a2, 12($sp)
        add     $a0, $a0, $a2
        sw      $a0, 4($sp)         # Address of next row

        j       fliploop

flipexit:
        lw      $ra, 0($sp)         # $ra is the only register we need
        addi    $sp, $sp, 16
        jr      $ra
```

Question 2: Multicycle CPU implementation

MIPS is a register-register architecture, where arithmetic source and destinations must be registers. But let's say we wanted to add a register-memory instruction:

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

Here is the instruction format, for your reference (shamt and func are not used):

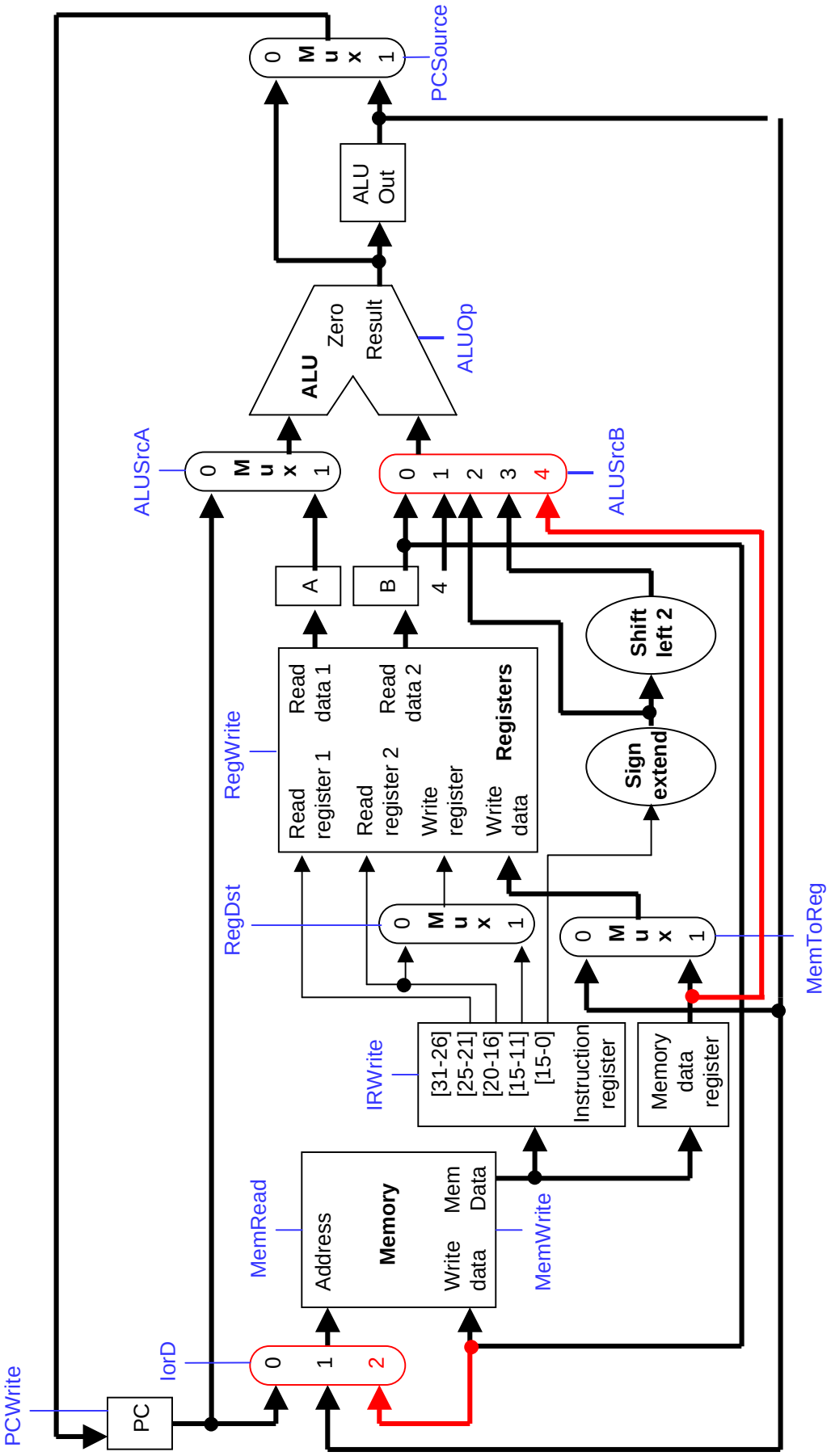
Field	op	rs	rt	rd	shamt	func
Bits	31-26	25-21	20-16	15-11	10-6	5-0

The multicycle datapath from lecture is shown below. You may assume that ALUOp = 010 performs an addition.

Part (a)
On the next page, show what changes are needed to support *addm* in the multicycle datapath. (10 points)

On the next page, 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.

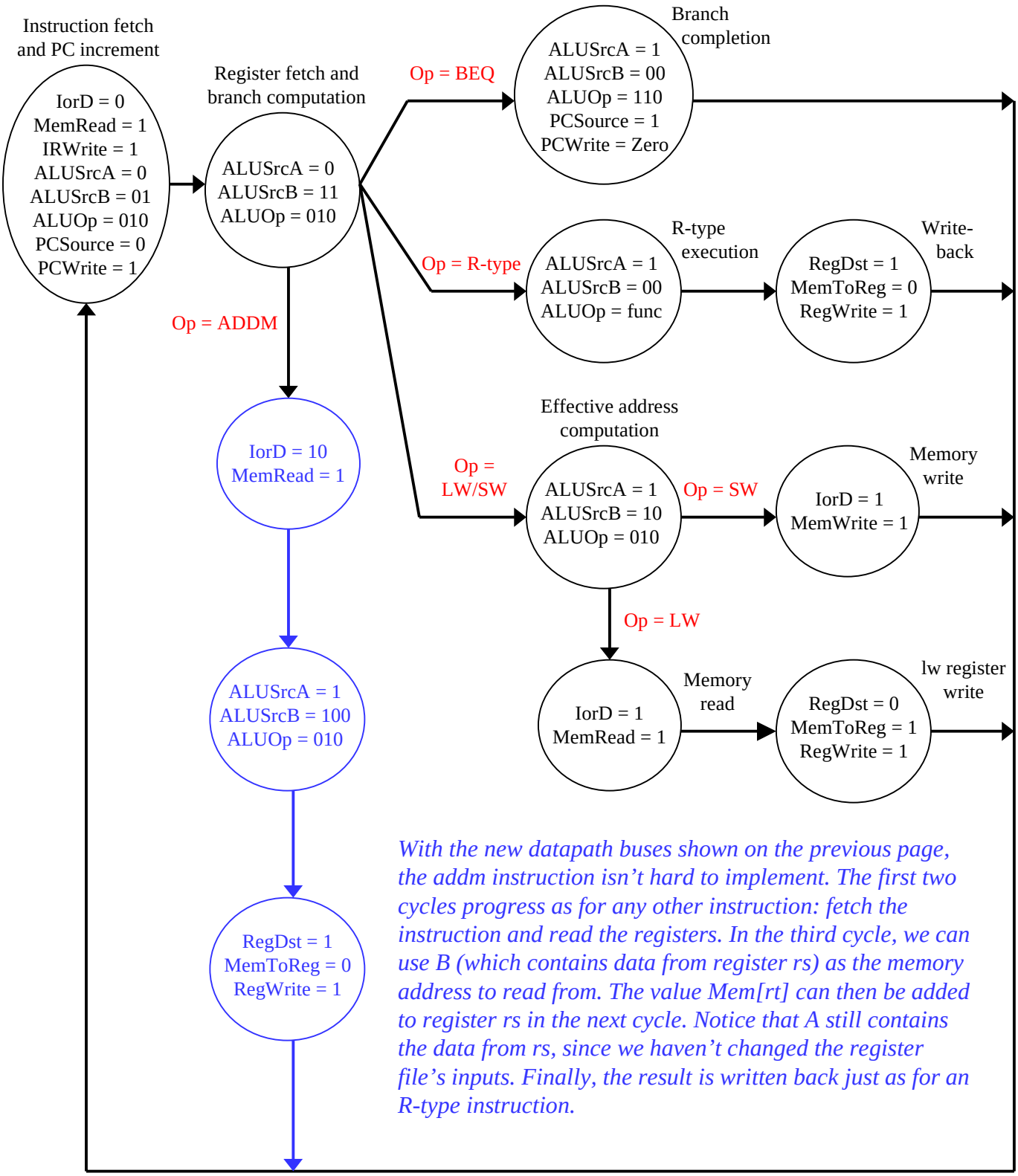
Question 2 continued



Question 2 continued

Part (b)

Complete this finite state machine diagram for the *addm* instruction. Be sure to include any new control signals you may have added. (15 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: Pipelining and forwarding

The next page shows a diagram of the pipelined datapath with forwarding, but no hazard detection unit.

Part (a)
Both of the code fragments below have dependencies, but only one of them will execute correctly with the forwarding datapath. Tell us which one, and why. If you like, you can draw a pipeline diagram to aid in your explanation. (5 points)

```
add $t0, $a0, $a1
add $v0, $t0, $t0
```

```
lw $t0, 0($a0)
add $v0, $t0, $t0
```

The code on the right fails. The data that gets loaded into $t0 won't be available until the end of the lw instruction's MEM stage, but that's the same stage in which the add needs to use $t0. The simplest solution, as we saw in class, is to stall the add instruction for one cycle until the data from 0($a0) is available.

Part (b)
Here is one more code fragment. How is this one different from the previous ones? (5 points)

```
lw $t0, 0($a0)
sw $t0, 0($a1)
```

The sw can't write $t0 to memory until it's first loaded by the lw. The given datapath doesn't forward data properly for this case; an alternative solution would be to stall the sw instruction for two cycles, as shown in the pipeline diagram below (assuming that data written to the register file can be read in the same cycle).

	1	2	3	4	5	6	7	8
lw $t0, 0($a0)	IF	ID	EX	MEM	WB			
sw $t0, 0($a1)		IF	—	—	ID	EX	MEM	WB

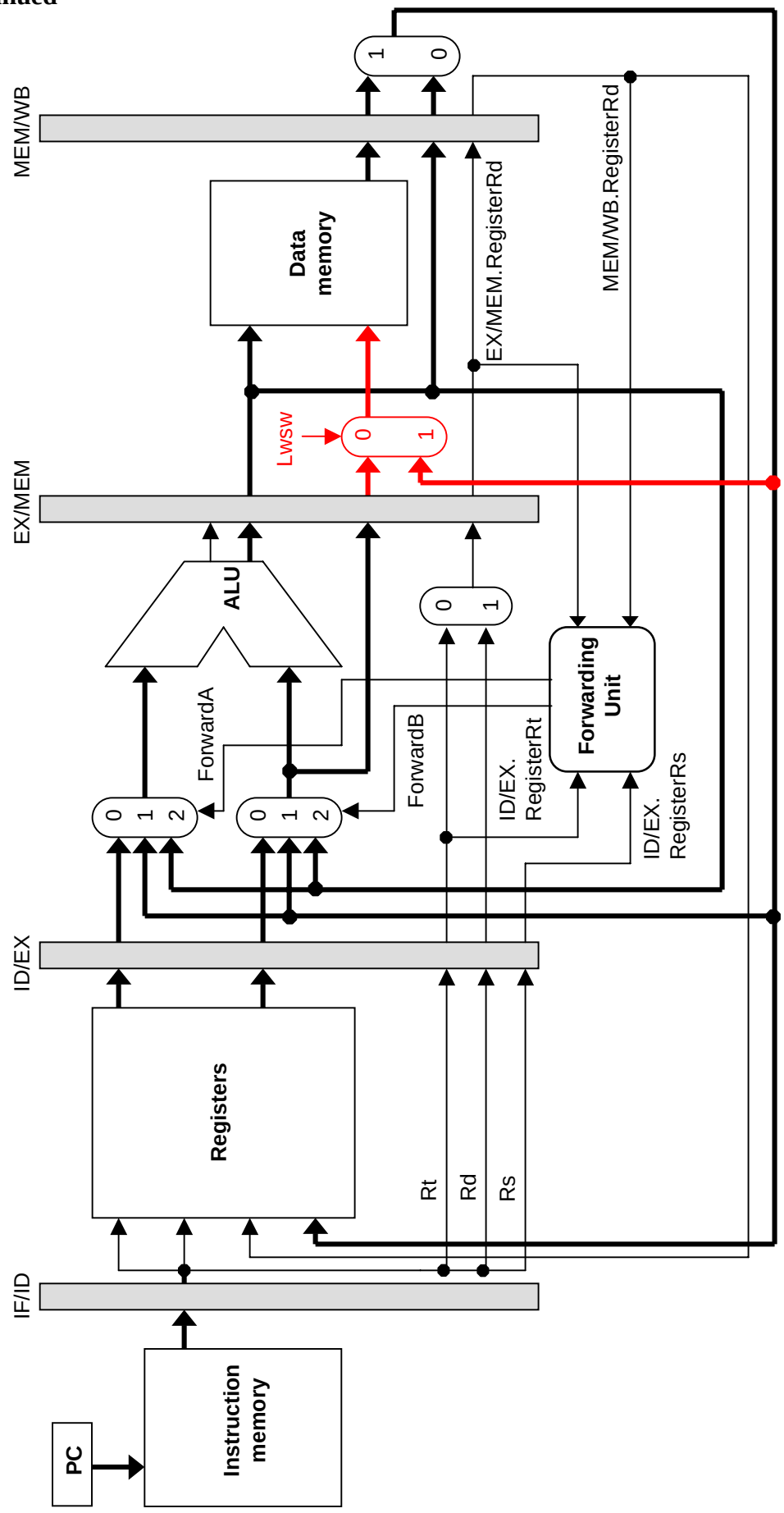
Part (c)
It is possible to modify the datapath so the code in Part (c) executes without *any* stalls. Explain how this could be done, and show your changes on the next page. (15 points)

	1	2	3	4	5	6
lw $t0, 0($a0)	IF	ID	EX	MEM	WB	
sw $t0, 0($a1)		IF	ID	EX	MEM	WB

Here is a pipeline diagram assuming no stalls. The new value of $t0 would be available in the fifth cycle, which is the same time that "sw" needs to write $t0 to memory. We can forward this value using a mux as shown on the next page. For "ordinary" instructions, the mux selection would be set to 0. But if we need to forward data from a lw to a sw instruction, the mux selection would be 1 instead.

This situation occurs when there is a lw instruction in the writeback stage which writes to the same register that is read by a sw instruction in its memory stage: MEM/WB.MemRead = 1, EX/MEM.MemWrite = 1, and MEM/WB.RegisterRt = EX/MEM.RegisterRt. (With the datapath we showed in class, most of these control signals are not *available anymore in the EX/MEM or MEM/WB stages, but it's not hard to put them back in the appropriate pipeline registers.)*

Question 3 continued



Question 4: Pipelining and performance

Here is the *toupper* example function from lecture. It converts any lowercase characters (with ASCII codes between 97 and 122) in the null-terminated argument string to uppercase.

```
toupper:
    lb   $t2, 0($a0)
    beq  $t2, $0, exit      # Stop at end of string
    blt  $t2, 97, next      # Not lowercase
    bgt  $t2, 122, next     # Not lowercase
    sub  $t2, $t2, 32       # Convert to uppercase
    sb   $t2, 0($a0)        # Store back in string
next:
    addi $a0, $a0, 1
    j    toupper
exit:
    jr   $ra
```

Assume that this function is called with a string that contains exactly 100 lowercase letters, followed by the null terminator.

Part (a)
How many instructions would be executed for this function call? (5 points)

With 100 lowercase letters, every one of the eight instructions in the loop will be executed 100 times. In addition, three more instructions are executed to process the final null character. That's 803 instructions!

Part (b)
Assume that we implement a single-cycle processor, with a cycle time of 8ns. How much time would be needed to execute the function call? What is the CPI? (5 points)

A single-cycle processor by definition always executes each instruction in one cycle, for a CPI of 1. You also know that 803 instructions are executed in all, so plugging these values into the equation for CPU execution time gives you:

$$\begin{aligned} \text{CPU time} &= \text{Number of instructions executed} \times \text{CPI} \times \text{Clock cycle time} \\ &= 803 \text{ instructions} \times 1 \text{ cycle/instruction} \times 8 \text{ ns/cycle} \\ &= 6424 \text{ ns} \end{aligned}$$

Question 4 continued

Part (c)
Now assume that our processor uses a 5-stage pipeline, with the following characteristics:

- Each stage takes one clock cycle.
- The register file can be read and written in the same cycle.
- Assume forwarding is done *whenever* possible, and stalls are inserted otherwise.
- Branches are resolved in the ID stage and are predicted correctly 90% of the time.
- Jump instructions are fully pipelined, so no stalls or flushes are needed.

How many total cycles are needed for the call to *toupper* with these assumptions? (10 points)

The "base" number of cycles needed, assuming no stalls or flushes, would be four cycles to fill the pipeline and 803 more cycles to complete the function, for a total of 807 cycles.

However, the given code also includes 301 branches (three for each of the first 100 loop iterations, and one for the end of the string). If 10% of these are mispredicted and require one instruction to be flushed, there will be a total of 0.10 x 301 x 1 = 30 wasted cycles for flushes.

There is also a hazard between the "lb" instruction and the "beq" right after it. If we assume forwarding is done whenever possible and that registers can be written and read in the same cycle, we'd still need to insert a two-cycle stall between these instructions, since branches are determined in the ID stage.

		1	2	3	4	5	6	7	8
lb	$t2, 0($a0)	IF	ID	EX	MEM	WB			
beq	$t2, $0, exit		IF	—	—	ID	EX	MEM	WB

The "lb/beq" sequence is executed 101 times, so we need a total of 202 stall cycles. The total number of cycles would then be 807 + 30 + 202 = 1039.

Part (d)
If the cycle time of the pipelined machine is 2ns, how would its performance compare to that of the single-cycle processor from Part (b)? (5 points)

At 2 ns per cycle, the pipelined system will require 1039 x 2 = 2078 ns. This is a performance improvement of 6424/2078, or roughly three times!

Question 5: Cache computations

$$\text{AMAT} = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty})$$
$$\text{Memory stall cycles} = \text{Memory accesses} \times \text{miss rate} \times \text{miss penalty}$$

The Junkium processor has a 16KB, 4-way set-associative (i.e., each set consists of 4 blocks) data cache with 32-byte blocks. Here, a "KB" is 2^{10} bytes.

Part (a)
How many total blocks are in the Level 1 cache? How many sets are there? (5 points)

A 16KB cache would have $16(2^{10}) = 2^{14}$ bytes of total data. With 32 bytes per block, this means the cache must have $2^{14}/2^5 = 2^9 = 512$ blocks. And with four blocks in each set, you would have 128 sets.

Part (b)
Assuming that memory is byte addressable and addresses are 35-bits long, give the number of bits required for each of the following fields: (5 points)

Tag	*23*
Set index	*7*
Block offset	*5*

Part (c)
What is the total size of the cache, including the valid, tag and data fields? Give an exact answer, in either bits or bytes. (5 points)

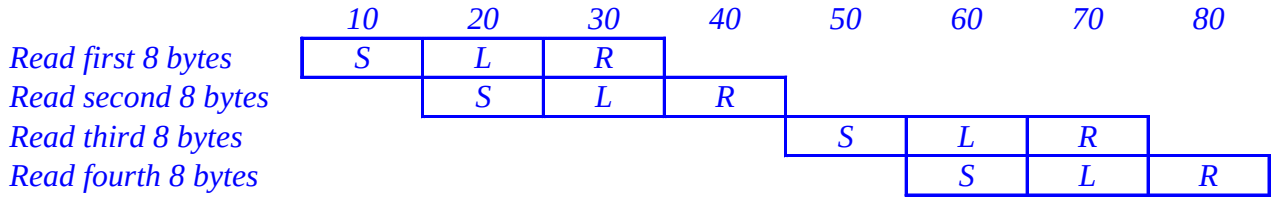
Each cache block must contain one valid bit and a 23-bit tag field, in addition to 32 bytes of data. This works out to 35 bytes per block, and 512 ×35 = 17,920 bytes. In other words, this 16,384-byte data cache needs about 1.5KB of extra "overhead" storage.

Question 5 continued

Assume that the Junkium cache communicates with main memory via a 64-bit bus that can perform one transfer every 10 cycles. Main memory itself is 64-bits wide and has a 10-cycle access time. Memory accesses and bus transfers may be overlapped.

Part (d)
What is the miss penalty for the cache? In other words, how long does it take to send a request to main memory and to receive an entire cache block? (5 points)

64 bits is 8 bytes, so we would need four memory transfers to fill a 32-byte cache block. Assuming that you can "pipeline" memory transfers, it'll take a total of 80 clock cycles as shown below. The labels S, L and R indicate sending an address, waiting for the memory latency, and receiving the data.



Note that with a single bus between the cache and RAM, you wouldn't be able to send an address to the memory and *receive data from the memory at the same time; that's why there is a "stall" between the second and third transfers.*

Part (e)
If the cache has a 95% hit rate and a one-cycle hit time, what is the average memory access time? (5 points)

You can just use the formula given on the previous page.

$$AMAT = 1 + (0.05 \times 80) = 5 \text{ cycles}$$

Part (f)
If we run a program which consists of 30% load/store instructions, what is the average number of memory stall cycles per instruction? (5 points)

Again, this is just plug and play. Assuming the program has I instructions, we would find the following.

$$\begin{aligned} \text{Stall cycles} &= \text{Memory accesses} \times \text{miss rate} \times \text{miss penalty} \\ &= 0.30I \text{ instructions} \times 0.05 \text{ misses/instruction} \times 80 \text{ cycles/miss} \\ &= 1.2I \text{ cycles} \end{aligned}$$

So the average number of stall cycles per instruction is 1.2.

Question 6: Input/Output

Little Howie is setting up a web site. He bought a fancy new hard disk which advertises:

- an 8ms average seek time
- 10,000 RPM, or roughly 6ms per rotation
- a 2ms overhead for each disk operation
- a transfer speed of 10,000,000 bytes per second

Howie had enough money left for a 10,000,000 byte per second network connection. His system bus has a maximum bandwidth of 133 megabytes per second, and his HTML files have an average size of 8,000 bytes.

Part (a)
How much time will it take, on average, to read a random HTML file from the disk? Include the seek time, rotational delay, transfer time and controller overhead. (5 points)

It will take 8,000/10,000,000 = 0.8ms to transfer one HTML file. The average rotational delay would be the time for a disk platter to spin 1/2 way, or 3ms. Adding these numbers together with the 8ms seek time and 2ms overhead, it'll take 0.8ms + 3ms + 8ms + 2ms = 13.8ms on average to read an HTML file.

Part (b)
With Howie's disk and network configuration, how many pages can be served per second? Round your answer to the nearest integer. (5 points)

This is basically asking you for the throughput of Little Howie's computer. If it takes 13.8ms to read one HTML file, the machine will be able to read 1000/13.8, or roughly 72, files per second. (That's 72 x 8KB = 576KB/second, which is easily handled by the 133MB/s system bus and 10MB/s network.)

Part (c)
Times are good, and Little Howie upgrades his web server with three additional hard disks, each with the specifications listed above. Now what is the maximum number of pages that can be served per second? Again, round to the nearest integer. (5 points)

With four drives, Howie can theoretically serve 4 x 72 = 288 pages per second. In terms of the transfer rate, this would be 4 x 576KB = 2304KB/second...still not enough to overload the system bus or network.

Part (e)
Times are bad, and Howie has to downgrade his network to one with a 1,000,000 byte per second bandwidth. How will this affect the number of pages per second that can be served? (5 points)

Poor Little Howie won't be able to serve 288 pages per second anymore. Even though his computer can serve 2304KB/s of data, the slower 1MB/s network can't handle that much traffic. The network basically becomes the bottleneck in this system, and limits Howie to just 1,000,000/8,000 = 125 pages per second.