CS232 Final Exam
May 10, 2002

Name: Magneto

- This exam has 15 action-filled pages, including this cover.
- There are six questions, worth a total of 150 points.
- Pages 14-15 contain a summary of the MIPS instruction set and a diagram of a pipelined datapath. You may tear these sheets out.
- No other written references or calculators are allowed.
- Write clearly and show your work. State any assumptions you make.
- You have 3 hours. Budget your time!

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

Question 1: MIPS programming

We'll start off by writing some MIPS functions to solve a common technical interview question: how can you reverse the order of the words in a string? For example, reversing the words in the string *I Love Lucy* would result in *Lucy Love I.*

A string is just an array of bytes, with each byte representing one character. The last byte of the array is always 0, to indicate the end of the string. Also, a space is represented by the value 32. For instance, the string above might be stored in memory from addresses 500–511 as follows.

Address	500	501	502	503	504	505	506	507	508	509	510	511
Byte	73	32	76	111	118	101	32	76	117	99	121	0
Character	I		L	o	v	e		L	u	c	y	\0

Part (a)
First write a function *rev*, which simply reverses bytes in an array. The arguments passed in \$a0 and \$a1 are the starting address and the number of bytes to reverse. For example, invoking *rev* for the string above with \$a0 and \$a1 set to 502 and 4 should change the string to *I evoL Lucy*. There is no return value. Make sure to follow all MIPS calling and register save conventions. (10 points)

Our sample solution below swaps bytes starting from the ends of the range, and working towards the middle.

```
rev:    addu  $a1, $a0, $a1    # $a1 points to last byte
loop:   bgeu  $a0, $a1, exit   # Stop when the pointers cross
        lb    $t0, 0($a0)      # Swap two bytes
        lb    $t1, 0($a1)
        sb    $t0, 0($a1)
        sb    $t1, 0($a0)
        addiu $a0, $a0, 1      # Work towards middle of array
        subu  $a1, $a1, 1
        j     loop
exit:   jr    $ra
```

Question 1 continued

Part (b)
Using the *rev* function, you should now be able to write *revwords* to reverse the order of the words within a string. The only argument is the starting address of the string, passed in $a0. Again, there is no return value. You may write additional "helper" functions if you like, but be sure you follow all MIPS calling and register save conventions. (15 points)

You can implement *revwords* according to the following basic algorithm.

1. Call *rev* to completely reverse the original string. For example, *I Love Lucy* becomes *ycuL evoL I*. Remember that strings are terminated by a 0 byte.
2. Then use *rev* to reverse each word in the new string—*ycuL evoL I* becomes *Lucy Love I*. You can assume that "words" are separated by a space character, with the byte value 32.

```
revwords:
         subu   $sp, $sp, 12   # $sp is usually word-aligned
         sw     $ra, 0($sp)
         sw     $a0, 4($sp)    # Save starting address of string

         move   $a1, $a0       # First find the end of the string
count:   lb     $t0, 0($a1)
         beq    $t0, $0, revstr
         addiu  $a1, $a1, 1
         j      count
revstr:  subu   $a1, $a1, $a0  # Compute the string length in $a1
         subu   $a1, $a1, 1
         jal    rev            # Reverse the whole string

next:    lw     $a0, 4($sp)    # Start of word to reverse
         move   $a1, $a0       # The end of the word
scan:    lb     $t0, 0($a1)    # ...is marked by
         beq    $t0, 32, word  # ...either a space
         beq    $t0, $0, word  # ...or the end of the string
         addiu  $a1, $a1, 1
         j      scan
word:    addiu  $a2, $a1, 1
         sw     $a2, 4($sp)    # Remember the start of the next word
         sb     $t0, 8($sp)    # ...and the terminator of this word
         subu   $a1, $a1, $a0
         subu   $a1, $a1, 1    # Find length of current word
         jal    rev            # Reverse the current word
         lb     $t0, 8($sp)    # If terminator was a space
         bne    $t0, $0, next  # ...continue to next word

         lw     $ra, 0($sp)
         addiu  $sp, $sp, 12
         jr     $ra
```

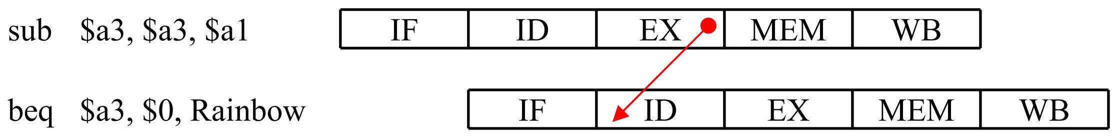
Question 2: Forwarding

Consider the pipelined datapath shown on page 15.

Part (a)
Describe the difficulty in executing the following instructions in this datapath. How many stall cycles must be inserted to correctly execute this code? (5 points)

```
sub   $a3, $a3, $a1
beq   $a3, $0, SomewhereOverTheRainbow
```

In the datapath from class shown on page 15, branches are decided in the ID stage, so this is the stage in which the source operands must be available.But if one of the branch sources is updated by the previous instruction, there will be a dependency that the datapath doesn't handle.



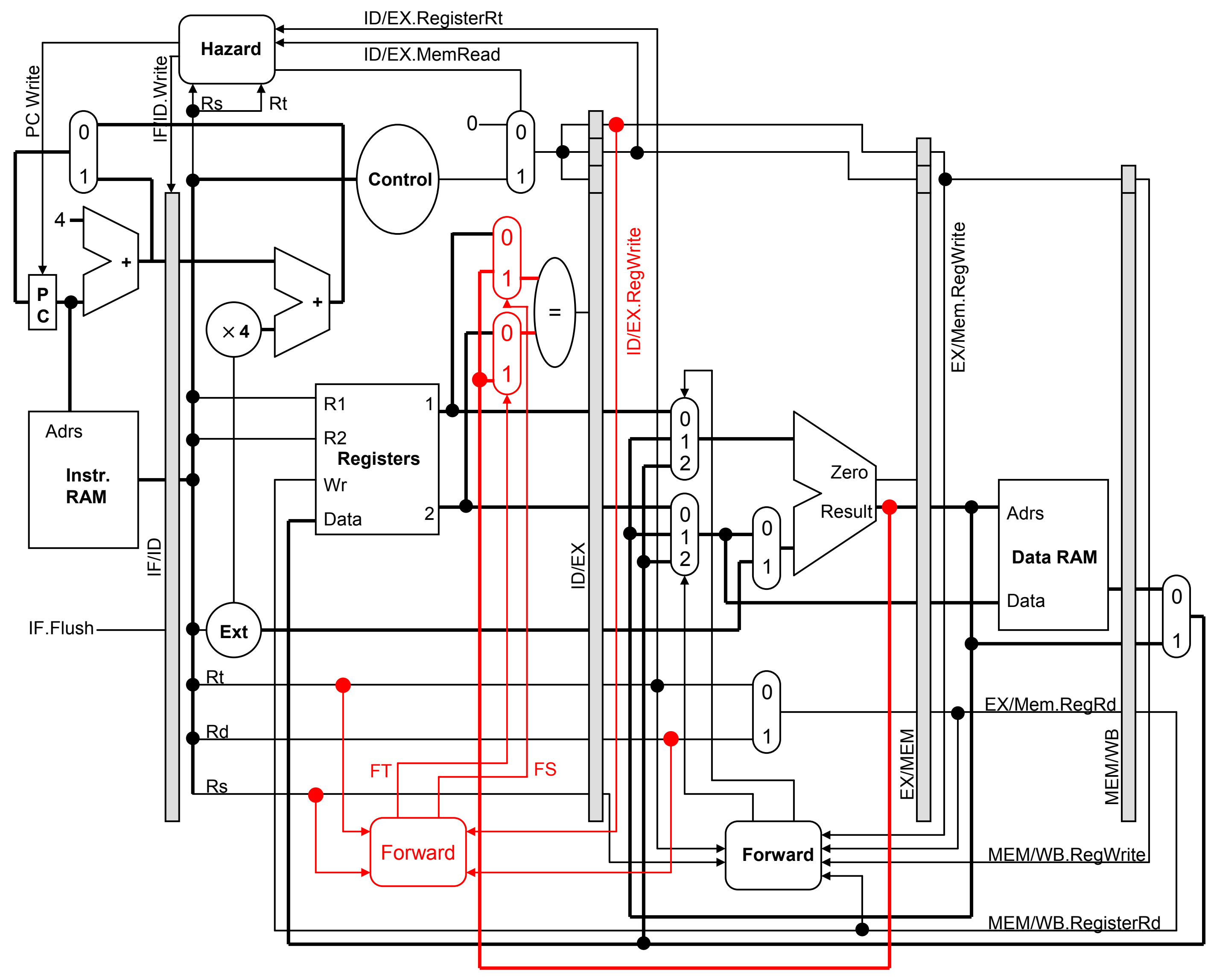
In a pipeline diagram for the code above, you can see that the sub doesn't produce a result until the end of its EX stage, but that's the same cycle in which the branch needs to read register $a3. Since this datapath doesn't support forwarding to the ID stage, the branch would have to stall for two cycles, until sub writes the new value of $a3 to the register file.

Part (b)
Explain how the number of stalls could be minimized with forwarding. Show how muxes might be added to the pipelined datapath on the *next* page, and discuss how they can be controlled by a forwarding unit. Giving equations is the most concise approach, but in any case make sure your description is clear and precise.

You can assume the register file and ALU have the same latency, while muxes have negligible delays. You only need to consider dependencies between branches and R-type arithmetic instructions with the format shown below—you don't have to worry about loads or other I-type instructions. (20 points)

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

Question 2 continued



This problem is just asking you to apply the ideas of forwarding to a different stage of the pipeline. Our muxes and forwarding unit are shown in red here; they are very similar to the existing forwarding hardware in the EX stage. The branch comparator can then access values from the register file, or from the ALU output of the previous instruction when there is a dependency. The new muxes should be set to 1 only if the ID/EX destination register Rd is the same as one of the ID source registers Rs/Rt.

if *(ID/EX.RegWrite = 1* and *IF/ID.RegisterRs = ID/EX.RegisterRd)*
then *FS = 1*

if *(ID/EX.RegWrite = 1* and *IF/ID.RegisterRt = ID/EX.RegisterRd)*
then *FT = 1*

Question 3: Pipelining performance

Here is a short MIPS assembly language loop. (This is a simpler version of a very common operation in scientific applications.) Assume that we run this code on the pipelined datapath shown on page 15.

```
Frodo:
    lw     $t0, 0($a1)
    mul    $t1, $t0, $a2
    lw     $t0, 0($a0)
    add    $t0, $t0, $t1
    sw     $t0, 0($a0)
    sub    $a3, $a3, 1
    addi   $a0, $a0, 4
    addi   $a1, $a1, 4
    bne    $a3, $0, Frodo
```

Part (a)
Find the number of clock cycles needed to execute this code, accounting for all possible stalls and flushes. Assume that $a3 is initially set to 100. (5 points)

To figure out the number of cycles needed, we'll need to find the total number of instructions executed and adjust it by the number of stalls and flushes. There will be a load stall between the first two instructions, and another one between the third and fourth instructions. None of the other dependencies are hazards. Because branches are determined in the ID stage and we predict that they are not taken, there will be a flush after the bne *on every loop iteration except the last one.*

If we consider the load stalls and branch flushes as extra nop *instructions, each loop iteration except the last one will execute 12 instructions. For 100 iterations, this works out to (100 × 12) – 1 = 1199 instructions. In a pipelined datapath with five stages, we'll need four cycles to fill the pipeline and then 1199 more cycles to complete the 1199 instructions. The total number of clock cycles works out to 4 + 1199 = 1203 cycles.*

Part (b)
Rewrite the code to eliminate as many stalls as possible. Show your modified assembly language program below. (10 points)

As you should have discovered in Part (a), there are only two load stalls to worry about. We can insert one or more instructions between each of the two loads and the subsequent uses of $t0. In the sample solution at the bottom, we just moved two of the later, independent instructions after the two loads.

```
Frodo:
    lw     $t0, 0($a1)
    addi   $a1, $a1, 4
    mul    $t1, $t0, $a2
    lw     $t0, 0($a0)
    addi   $a0, $a0, 4
    add    $t0, $t0, $t1
    sw     $t0, 0($a0)
    sub    $a3, $a3, 1
    bne    $a3, $0, Frodo
```

Question 3 continued

Part (c)
What is the performance improvement of your new function as compared to the original one? Again, assume that $a3 is initially set to 100. (5 points)

We still need one flush for the first 99 loop iterations, but we've removed both of the load stalls. Thus, each loop iteration now executes the equivalent of 10 instructions, for a total of (100 × 10) – 1 = 999 instructions. This will require 4 + 999 = 1003 clock cycles in our five-stage pipeline.

The performance improvement turns out to be 1203 cycles / 1003 cycles, or approximately 1.2 times. That's a 20% performance gain just for moving two instructions in the loop body!

Part (d)
Suggest a way to rewrite this code to reduce the number of cycles lost to flushes. You do *not* have to show any actual code. (5 points)

We looked at two possible ways of reducing the number of flushes. First, you could "unroll" the loop as we discussed in one homework assignment, to reduce the number of branches. For example, each loop iteration might process two *words from the arrays at $a0 and $a1, resulting in half as many branches and half as many flushes.*

Second, there are many flushes currently because the code has a branch instruction which is taken 99 times out of 100, defying our simple prediction algorithm. You might rewrite the loop using a branch which is not *taken 99 times out of 100. For example, you could rework the loop with a* beq *at the beginning.*

```
Frodo:  beq  $a3, 100, Exit
        lw   $t0, 0($a1)
        ...
        j    Frodo
Exit:   ...
```

For our datapath, the added jump instruction negates any performance advantage of removing the flush. But a trick like this could be useful in processors which determined branches later. If branches were tested in the third pipeline stage so each mispredicted branch resulted in two flushed instructions, for instance, then we could save one cycle per iteration with this code reorganization.

Question 4: Cache performance

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

The Corleone 2000 processor has two levels of data caches, with the characteristics shown below. You can also assume that it takes 50 clock cycles to request and complete a 32-byte transfer between main memory and the L2 cache.

	L1	L2
Data size	32 KB	256 KB
Block size	8 bytes	32 bytes
Associativity	Direct-mapped	4-way
Hit time	1 cycle	19 cycles
Miss rate	5%	2%

Part (a)
How many bits of a 32-bit main memory address would be used as the tag, index and block offset fields, for each of the two caches above? Assume that 1 KB = 2^{10} bytes. Also, 32 = 2^5 and 256 = 2^8. (5 points)

The number of block offset bits is determined by the cache block size; three bits of the memory address are needed to select one of the eight bytes in each L1 cache block, for example. Also, eight-byte blocks implies that the L1 cache has 4,096 total blocks (2^{15} total bytes / 2^3 bytes per block = 2^{12} blocks), so the index is 12 bits long. The remaining 32 – 12 – 3 = 17 bits form the tag.

	L1	L2
Tag size	*17*	*16*
Index size	*12*	*11*
Block offset	*3*	*5*

The numbers for the L2 cache can be computed in a similar fashion. There are 2^{18} / 2^5 = 2^{13} blocks in the L2 cache. The main difference is that with 4-way associativity, the blocks are split into 2^{11} sets, so the index will be just 11 bits long.

Part (b)
Find the average memory access time for both the L2 and L1 caches. (5 points)

It takes 50 cycles to retrieve 32 bytes of data from main memory, so this would be the miss penalty for the L2 cache. The L2 AMAT is thus 20 cycles.

$$AMAT_{L1} = 19 \text{ cycles} + (0.02 \times 50 \text{ cycles}) = 20 \text{ cycles}$$

This also serves as the miss penalty for the L1 cache; when there is a miss in the primary cache, the average time to get data from the L2 is 20 cycles. So the AMAT of the L1 cache works out to 2 cycles.

$$AMAT_{L2} = 1 \text{ cycle} + (0.05 \times 20) \text{ cycles} = 2 \text{ cycles}$$

Question 4 continued

Part (c)

While you're still in the mood, let's look at the `Frodo` code again. Assume that $a0 and $a1 are initially 3000 and 4000, and the loop iterates 100 times. Both level 1 and level 2 data caches are empty at first. How many memory stall cycles would be required for the execution of this code, according to the formula and table given on the previous page? (5 points)

```
Frodo:
     lw    $t0, 0($a1)
     mul   $t1, $t0, $a2
     lw    $t0, 0($a0)
     add   $t0, $t0, $t1
     sw    $t0, 0($a0)
     sub   $a3, $a3, 1
     addi  $a0, $a0, 4
     addi  $a1, $a1, 4
     bne   $a3, $0, Frodo
```

Each loop iteration includes two loads and one store, so the entire program execution makes 300 memory accesses. According to the previous page, the L1 miss rate is 5% and the L1 miss penalty is 20 cycles. So the number of memory stall cycles should be 300 accesses × 0.05 misses/access × 20 cycles/miss = 300 cycles.

If you actually work out the number of hits and misses in this code, you'll see that there are many more stalls than accounted for by the simple formula. In this particular program, every other loop iteration would result in two L1 load misses (L1 blocks are eight bytes wide), for a total of 100 misses and 2000 stall cycles. There are so many misses because this code doesn't really exhibit a lot of locality—each element of the arrays $a0 and $a1 is loaded just once.

This illustrates again that the actual performance of a system depends on not just the hardware, but also the particular program being executed.

Part (d)

Briefly discuss how each of the different write policies (write-back or write-through for hits, and allocate-on-write or write-around for misses) might affect the performance of the `Frodo` function. (10 points)

In this code, each store will be a hit because the same memory address 0($a0) would have been loaded just previously. Thus, the cache's write miss policy will not affect this loop at all.

If we used a write-through cache, then every store instruction would result in a memory access. We can do a little better using a write-back cache instead, because the L1 cache blocks are eight bytes long and the loop accesses memory elements in sequence. All 100 elements of the $a0 array would fit into 50 cache blocks, so the write-back policy would result in at most 50 eight-byte stores to main memory.

Question 5: Cache computations

Here is a short C program that we can use to determine some information about a processor's cache.

```
char a[LENGTH];          // array of bytes
int i, j, temp;

for (i = 0; i < 10000; i = i + 1)
   for (j = 0; j < LENGTH; j = j + STRIDE)
      temp = temp + a[j];
```

The inner loop reads data from an array of *bytes* in a pattern that depends on `STRIDE`; different values of `LENGTH` and `STRIDE` may cause the inner loop body to execute different numbers of times. The entire inner loop is repeated 10,000 times. You can assume that the variables `i`, `j`, `LENGTH`, `STRIDE`, `temp` and `a` are all stored in registers, and only the array contents are kept in main memory.

We run this program on a processor with several different values of `LENGTH` and `STRIDE`, and compute the average amount of time to execute "`temp = temp + a[j]`" once. The results are summarized in the table below, with `LENGTH` and `STRIDE` given in bytes and average times in nanoseconds.

	STRIDE							
LENGTH	1	2	4	8	16	32	64	128
8	7 ns	7 ns	7 ns	7 ns	6 ns	7 ns	7 ns	7 ns
16	8 ns	7 ns	7 ns	6 ns	6 ns	7 ns	7 ns	7 ns
32	7 ns	8 ns	7 ns	7 ns	7 ns	7 ns	6 ns	7 ns
64	16 ns	21 ns	45 ns	45 ns	7 ns	8 ns	8 ns	9 ns
128	14 ns	25 ns	47 ns	45 ns	45 ns	7 ns	7 ns	7 ns
256	17 ns	25 ns	43 ns	45 ns	45 ns	46 ns	8 ns	7 ns

Use the data in this table to answer the questions on the next page. Explain your reasoning in each part.

You'll need to understand the program's memory access patterns in order to analyze the cache behavior. For example, when `LENGTH` *= 64 and* `STRIDE` *= 1, the inner loop will access all 64 elements of the array* `a[]`*, from element 0 to 63. But with* `LENGTH` *= 64 and* `STRIDE` *= 2, only half of the array is accessed— just elements 0, 2, 4, 6, ..., 60, 62.*

Question 5 continued

Part (a)
What is the size of the cache, not including valid and tag bits? (5 points)

The average times are approximately the same until the array length is greater than 32, which suggests the cache size is 32 bytes. You can verify this by looking at the first column of the table. A 32-byte cache with any *associativity and* any *block size would be able to hold array elements 0, 1, 2, 3, ..., 31. (Why?) However, the average times are higher for a 64-element array with a stride of one, indicating that there are misses and that the cache isn't able to hold that many elements at once.*

Part (b)
What is the block size? (5 points)

To determine any further information about the cache, we'll have to study the miss rates and patterns; in other words, we'll have to look at array lengths of 64 or more. We would expect larger times to indicate more cache misses, and you can see the times for a stride of four are larger than the times for a stride of two, which in turn are larger than the times for a stride of one. This can be explained by a block size of four. For a stride of one the program would read array elements 0, 1, 2, 3, ..., resulting in good spatial locality and one cache miss followed by three hits in the same four-byte cache block. For a stride of two reading elements 0, 2, 4, 6, ..., every other read would be a hit. But for a stride of four involving array elements 0, 4, 8, 12, ..., each read would be a miss. The increasing miss rates from a stride of one to four account for the increasing times shown in the table.

Part (c)
Is the cache direct mapped, fully associative, or *n*-way associative (and what is *n*)? (10 points)

If the cache holds 32 bytes and each block is four bytes long, then the cache must have eight blocks. In the bottom right side of the table, you can see for example that there is a very high access time for a length of 64 and a stride of 8, accessing array elements 0, 8, 16, 24, 32, 40, 48 and 56. This makes some sense, as each of these elements would map to the same index in an eight-block cache. However, when the stride increases to 16, we can access elements 0, 16, 32 and 48 with a very low average time, even though these elements all map to the same index as well. These elements must all be going in a single set that can hold four blocks. So this is a 32-byte cache, with 4-way associativity and four-byte blocks.

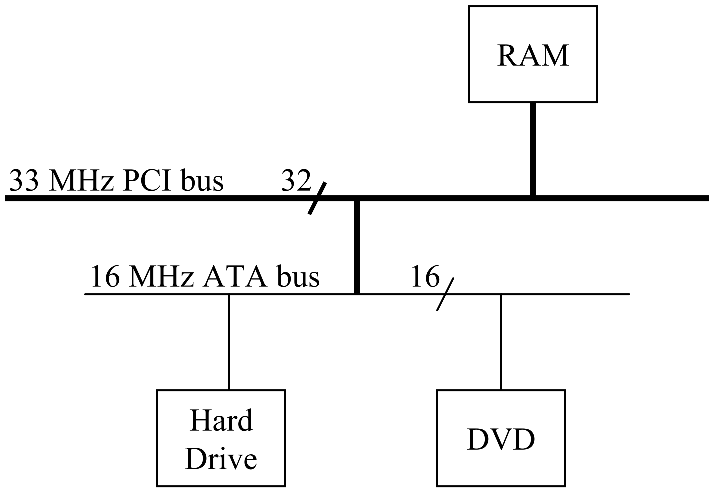
Part (d)
What are the approximate hit and miss times? (5 points)

Any array of length 32 or less should completely fit in the cache as mentioned in Part (a), so the 7ns average times should represent the hit time. But from Part (b), we know that array lengths of 64 or more and a stride of four should result in all misses, so the 45 ns average times there should be the miss time. (The hit and miss times will be a little less since these times include an address computation for `a[j]` *and an addition to* `temp`*, but you weren't expected to account for that.)*

Question 6: Input/Output

Big Howie is upgrading his computer system!

- Howie's new DVD-ROM drive advertises a seek time of 85ms, a rotational speed of 7500 rpm (or 8 ms per rotation), and a transfer rate of 25,000,000 bytes per second.
- He also bought a new 40 GB hard drive that has 4 double-sided platters and 20,000 tracks on each surface. Each track has 500 sectors, and each sector contains 500 bytes. The drive's rotational speed is 9600 rpm, which works out to 6.25 ms per revolution.
- Finally, he has shiny new SDRAM memory that transfers data at a 133 MHz clock rate.



All of this precious hardware is connected as shown above. The main PCI system bus is 32-bits wide and runs at 33 MHz. The drives are connected to a slower, narrower "ATA" bus attached to the PCI bus.

Compute exact answers for the questions below, instead of giving unsimplified arithmetic expressions. Also be sure to specify the units for your solutions.

Part (a)
How much time would it take for the DVD-ROM drive to read a random 500,000-byte sector? (5 points)

The response time will be the sum of the seek time, rotational delay, transfer time and controller overhead. The seek time is already given as 85ms. A single DVD rotation takes 8ms, so turning the disk halfway on average would require 4ms. The transfer time is just the transfer size divided by the transfer rate:

500,000 bytes / 25,000,000 bytes per second = 1/50 second, or 20ms

The problem doesn't mention any controller overhead, so the response time for the read is 109ms.

85ms + 4ms + 20ms = 109ms

Part (b)
Compute the hard drive's maximum data transfer rate per second, assuming that it can read and write data as fast as it spins. (10 points)

Each track on the hard drive contains 250,000 bytes of data.

500 sectors per track × 500 bytes per track = 250,000 bytes per track

At 9600 revolutions per minute, the disk would spin 160 times per second.

9600 revolutions per minute / 60 seconds per minute = 160 revolutions per second

So the disk can transfer 40,000,000 bytes per second.

250,000 bytes per track × 160 tracks per second = 40,000,000 bytes per second.

Question 6 continued

Part (c)
How long is needed to copy an entire 5 GB (5×10^9 bytes) DVD to the system's main memory? (5 points)

The DVD drive's maximum transfer rate is 25,000,000 bytes per second. To get data from the DVD-ROM to main memory, it will have to also go through the ATA bus and the PCI bus.

- *The ATA bus can transfer 16,000,000 cycles/second × 16 bits/cycle = 32,000,000 bytes/second.*
- *The PCI bus can transfer 33,000,000 cycles/second × 32 bits/cycle = 132,000,000 bytes/second.*
- *The memory accesses data at 133,000,000 cycles/second × 32 bits/cycle = 533,000,000 bytes/second.*

Thus, the DVD drive itself is the bottleneck along this path. Transferring 5 GB at a rate of 25,000,000 bytes per second would then require 200 seconds.

$$5 \times 10^9 \text{ bytes} / 2.5 \times 10^7 \text{ bytes/second} = 2 \times 10^2 \text{ seconds}$$

Part (d)
How long does it take to copy 5 GB (5×10^9 bytes) of data from RAM to the hard drive? (5 points)

The memory can access data at a rate of 533MB per second. This data has to go along the PCI and ATA buses, before it can be written by the hard drive. From the previous parts of this problem, we know the PCI bus can handle 133MB/s, and the hard disk can transfer 40MB/s. However, the ATA bus is limited to just 32,000,000 bytes/second, so this will limit the overall speed of the transfer from hard drive to main memory. In this case copying 5 GB of data will end up taking 156 seconds—faster than copying from the DVD, but not 40/25 times faster.

$$5 \times 10^9 \text{ bytes} / 3.2 \times 10^7 \text{ bytes/second} = 1.5625 \times 10^2 \text{ seconds}$$

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 $t0, $t1, $t2	$t0 = $t1 + $t2
	sub $t0, $t1, $t2	$t0 = $t1 – $t2
	addi $t0, $t1, 100	$t0 = $t1 + 100
	mul $t0, $t1, $t2	$t0 = $t1 × $t2
	move $t0, $t1	$t0 = $t1
	li $t0, 100	$t0 = 100
Data Transfer	lw $t0, 100($t1)	$t0 = Mem[100 + $t1] (one word)
	sw $t0, 100($t1)	Mem[100 + $t1] = $t0 (one word)
	lb $t0, 100($t1)	$t0 = Mem[100 + $t1] (one byte)
	sb $t0, 100($t1)	Mem[100 + $t1] = $t0 (one byte)
Branch	beq $t0, $t1, Label	if ($t0 == $t1) go to Label
	bne $t0, $t1, Label	if ($t0 != $t1) go to Label
	bge $t0, $t1, Label	if ($t0 >= $t1) go to Label
	bgt $t0, $t1, Label	if ($t0 > $t1) go to Label
	ble $t0, $t1, Label	if ($t0 <= $t1) go to Label
	blt $t0, $t1, Label	if ($t0 < $t1) go to Label
Set	slt $t0, $t1, $t2	if ($t1 < $t2) then $t0 = 1; else $t0 = 0
	slti $t0, $t1, 100	if ($t1 < 100) then $t0 = 1; else $t0 = 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:

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

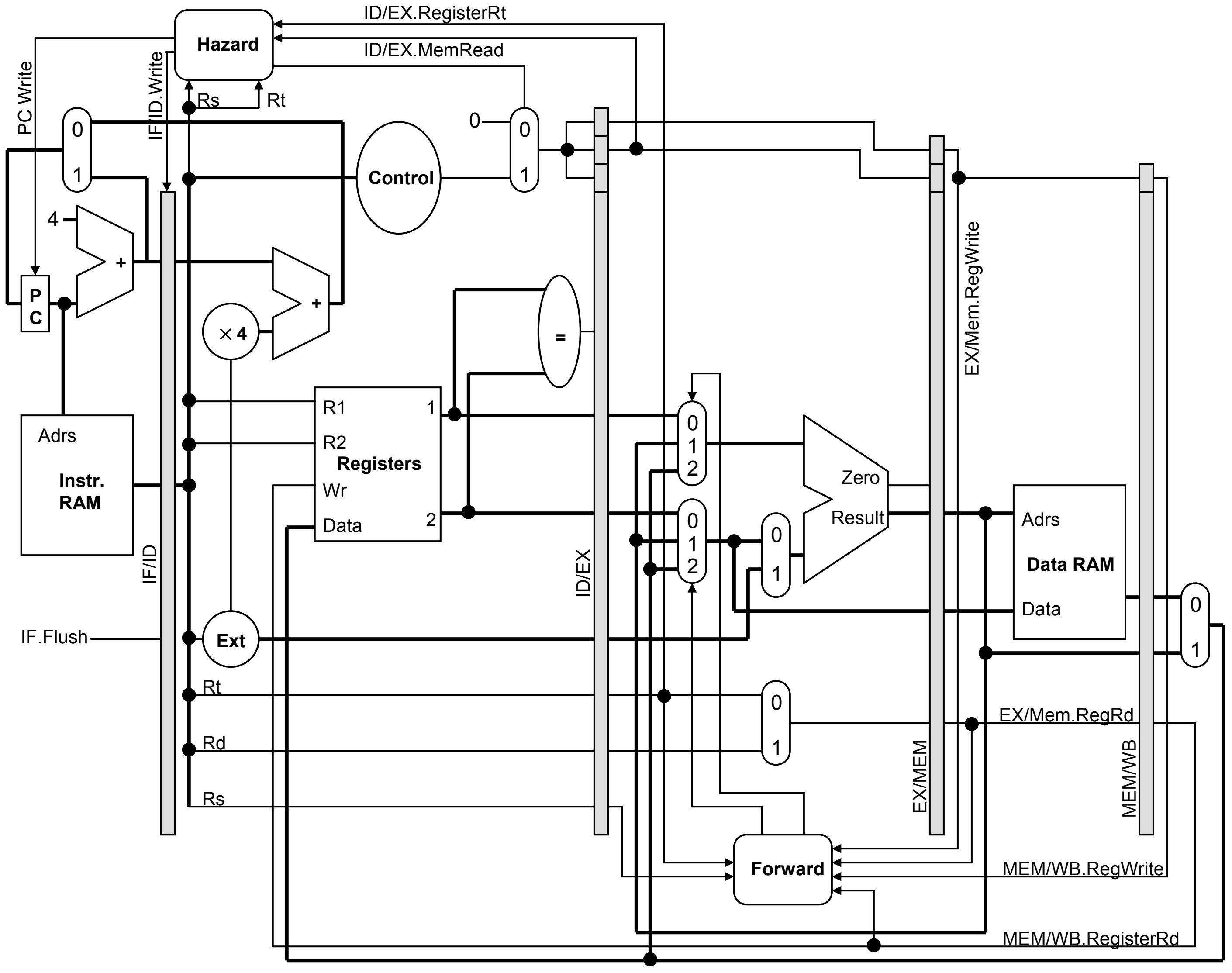
CPI is the average number of clock cycles per instruction, or:

$$\text{CPI} = \text{Number of cycles needed} / \text{Number of instructions executed}$$

Pipelined Datapath

Here is the final datapath that we discussed in class.

- Forwarding for arithmetic instructions is done from the EX/MEM and MEM/WB stages to the ALU.
- A hazard detection unit can insert stalls for lw instructions.
- Branches are assumed to be not taken.
- To minimize the number of flushes on a misprediction, branch determination is done in the ID stage.



ID/EX.RegisterRt
Hazard
ID/EX.MemRead
PC Write
IF/ID.Write
Rs
Rt
0
1
0
1
Control
4
+
P C
+
× 4
=
Adrs
Instr. RAM
R1
R2
Registers
Wr
Data
1
2
IF/ID
ID/EX
EX/MEM
MEM/WB
EX/Mem.RegWrite
Zero
Result
Adrs
Data RAM
Data
IF.Flush
Ext
Rt
Rd
Rs
EX/Mem.RegRd
Forward
MEM/WB.RegWrite
MEM/WB.RegisterRd