CS232 Midterm Exam 3
April 27, 2005

Name: _____

Section: _____

- This exam has 6 pages (nothing to tear off this time).
- You have 50 minutes, so budget your time carefully!
- No written references or calculators are allowed.
- To make sure you receive credit, please write clearly and show your work.
- We will not answer questions regarding course material.

Question	Maximum	Your Score
1	50	
2	50	
3	+6	
Total	100	

Question 1: Pipelined Datapath (50 points)

Here is the final datapath that we discussed in class, much like the one we built in Verilog. Branches are resolved in the decode stage.

Part (a)

If branches are not predicted, how many stall/flush cycles would be required for each branch? Assume no data hazards. (5 points)

1 cycle would be lost for each branch, because the branch target would be resolved at the end of the decode stage. The funny thing is it is actually easier to predict not taken, than stall, because stalling requires that an instruction must be recognized as a branch needs in the IF stage.

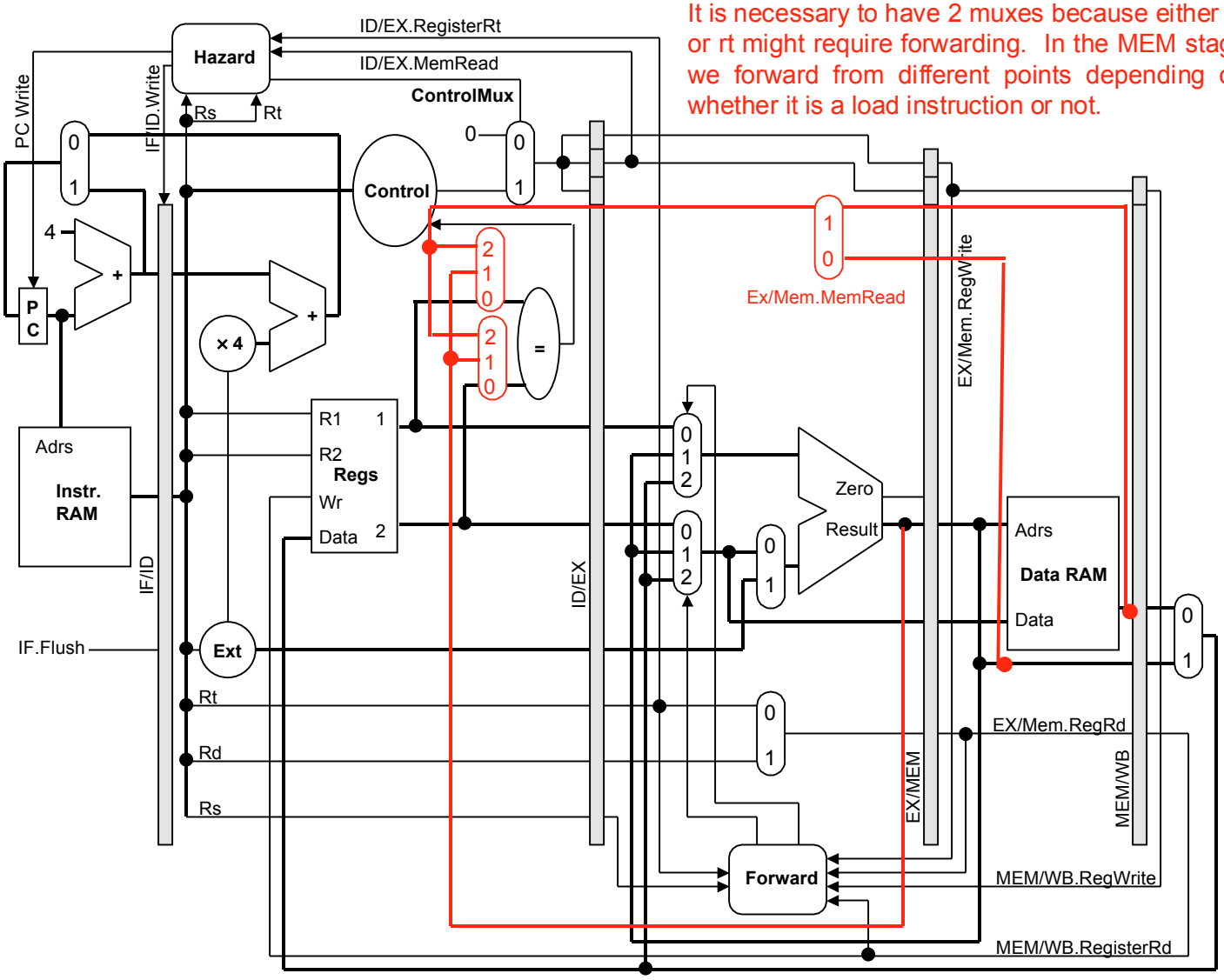
As drawn, there is an unsupported data hazard when branches use register values computed by earlier instructions, *e.g.,*

```
add     $3, $1, $2
bne     $3, $0, branch target
```

Part (b)

Add forwarding data paths to correctly support **all** data hazards for the `bne` instruction to the data path below. *Hint: by forwarding from the ALU output (marked with a dot), you can avoid adding new stalls.* (15 points)

The solution below requires no additional stalls. It is necessary to have 2 muxes because either rs or rt might require forwarding. In the MEM stage we forward from different points depending on whether it is a load instruction or not.



Question 1, continued

Here is the control logic for the multiplexor that handles the **rs** input of the ALU:

```
if ((EX/MEM.RegWrite == 1) and
    (EX/MEM.RegisterRd == ID/EX.RegisterRs))
        ForwardA = 1
else if ((MEM/WB.RegWrite == 1) and
        (MEM/WB.RegisterRd == ID/EX.RegisterRs))
        ForwardA = 2
else
        ForwardA = 0
```

Part (c)
Write the control logic for one of the forwarding muxes you implemented in part (a). (15 points)

The following equation is for the top mux (the one that deals with *rs*).

```
if  ((ID/EX.RegWrite == 1) and (ID/EX.RegisterRd == IF/ID.RegisterRs))
                        ForwardA = 1
else if  ((EX/MEM.RegWrite == 1) and  (EX/MEM.RegisterRd == IF/ID.RegisterRs))
                        ForwardA = 2
else
                        ForwardA = 0
```

Part (d)
Identify the (true) data dependences in the following piece of code: (10 points)

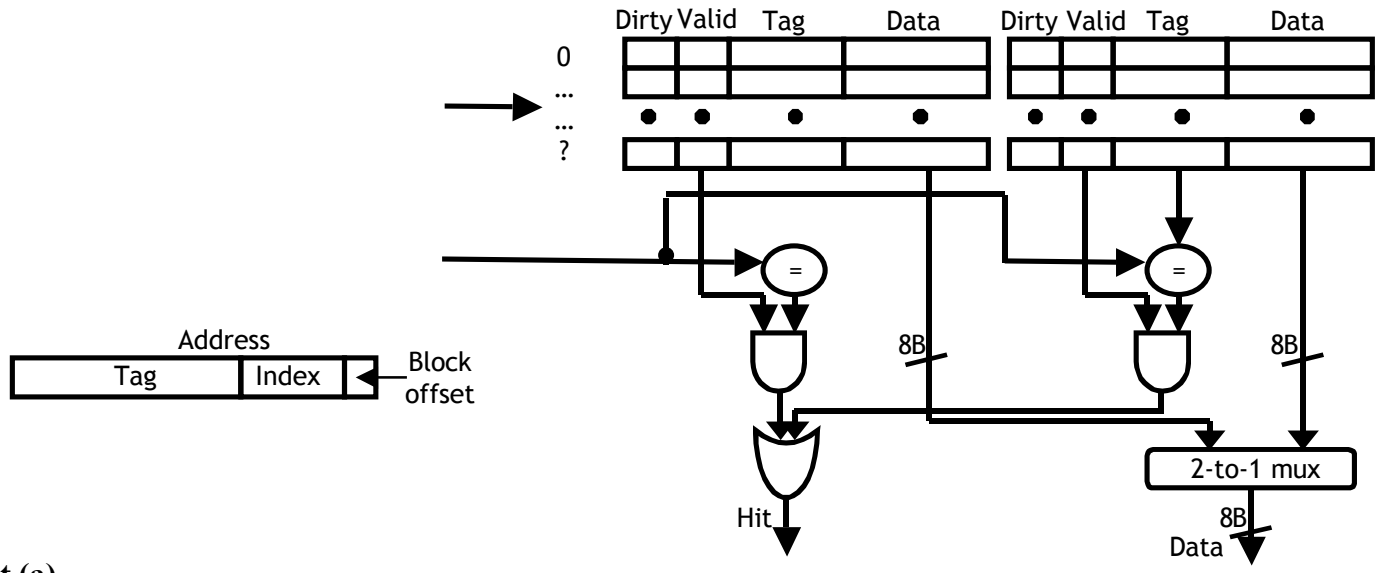
```
addi    $7,  $7,  1
sll     $8,  $7,  2
add     $8,  $8,  $14
lw      $8,  0($8)
lw      $8,  0($8)
bne     $7,  $22, target
```

True data dependences are between an instruction that writes a value and one that reads it. The set of dependences is independent of implementation (I.e., it is a data dependences whether it is a hazard or not).

Part (e)
For the pipeline on the previous page, do any of these data dependences cause stalls? If so, which ones and for how many cycles. (5 points)

Only the dependence between the two loads causes a stall, because it's value isn't available until the end of the MEM stage and needed at the beginning of the EX stage of the following instruction. This causes a 1 cycle stall and then uses the bypass path from the WB stage. All other dependences between neighboring instructions can be satisfied by the bypass path from the MEM stage. Register **$**7 has already been written back by the time the **bne** instruction is decoded.

Question 2: Cache Organization and Performance (50 points)



Part (a)
Consider the cache drawn above in a processor that uses 32-bit addresses for a byte-addressable memory. If the cache uses 10-bit indices, how big (in bits) are the cache tags? (5 points)

From the picture, we can see that it caches 8 byte blocks; therefore the block offset has to be 3 bits. Since the tag must contain all address bits that aren't index or block offset:

tag_size = address_size - index_size - block_offset_size = 32b - 10b - 3b = 19 bits

Part (b)
How much data can the cache hold? (10 points)

With 10-bit indices, each set holds 2^{10} (or 1024) blocks. From the picture, the cache is two-way set associative, so there are two such sets, or a total of 2048 blocks.

Since each block holds 8 bytes, the cache holds 2048 * 8B = 16384 bytes = 16KB.

Part (c)
How many bits of storage are required to implement the cache (include all data, tag, and control state)? (5 points)

We previously found that the cache holds 2048 blocks. In addition to the 8 bytes of data, we previously found the tags to be 19 bits, plus the picture shows a valid bit and a dirty bit. Thus, the full cache size is:

2048 blocks * (8 bytes + 19 bits + 1 bit + 1 bit) = 2048 * (8B + 19b) = 2048 * (64b + 19b) =
= 2048 * (83b) = 169,984b = 21,248B

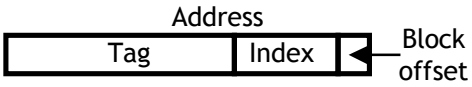
Part (d)
Explain how dirty bits get used (*i.e.,* when are they set, and when are they read). (5 points)

Dirty bits track which cache blocks contain values that are potentially inconsistent with the values stored in memory for those addresses. They are set when the block is written to (*e.g.,* by a store) and are read when the block is replaced to determine whether a write-back is required.

Question 2, continued

Part (e)

Given a direct-mapped cache with 4 blocks of 8 bytes, which of the following byte accesses hit? For those accesses that hit indicate whether the hit is because of spatial locality, temporal locality, or neither in the **reason** column. (20 points)



Address (binary)	Hit/Miss	Reason
000000	Miss	---
010101	Miss	---
111111	Miss	---
010000	Hit	spatial
011111	Miss	---
000100	Hit	spatial
111111	Miss	---
110111	Miss	---
000111	Hit	spatial
111111	Hit	temporal
011100	Miss	---

Part (f)

If you were given a processor's base CPI (*i.e.,* the CPI that a processor would have if it never missed in the data cache), what information would you need in order to compute its CPI with a real cache? (5 points)

To compute the complete CPI we need to compute the memory stall contribution to the CPI. This can be done by computing the number of stall cycles and dividing by the number of instructions (CPI = cycles per instruction).

The number of stall cycles can be computed as: (I is the number of instructions)

$$\#\ \text{stall cycles} = I * \text{fraction\_of\_loads\_and\_stores} * \text{data\_cache\_miss\_rate} * \text{miss\_penalty}$$

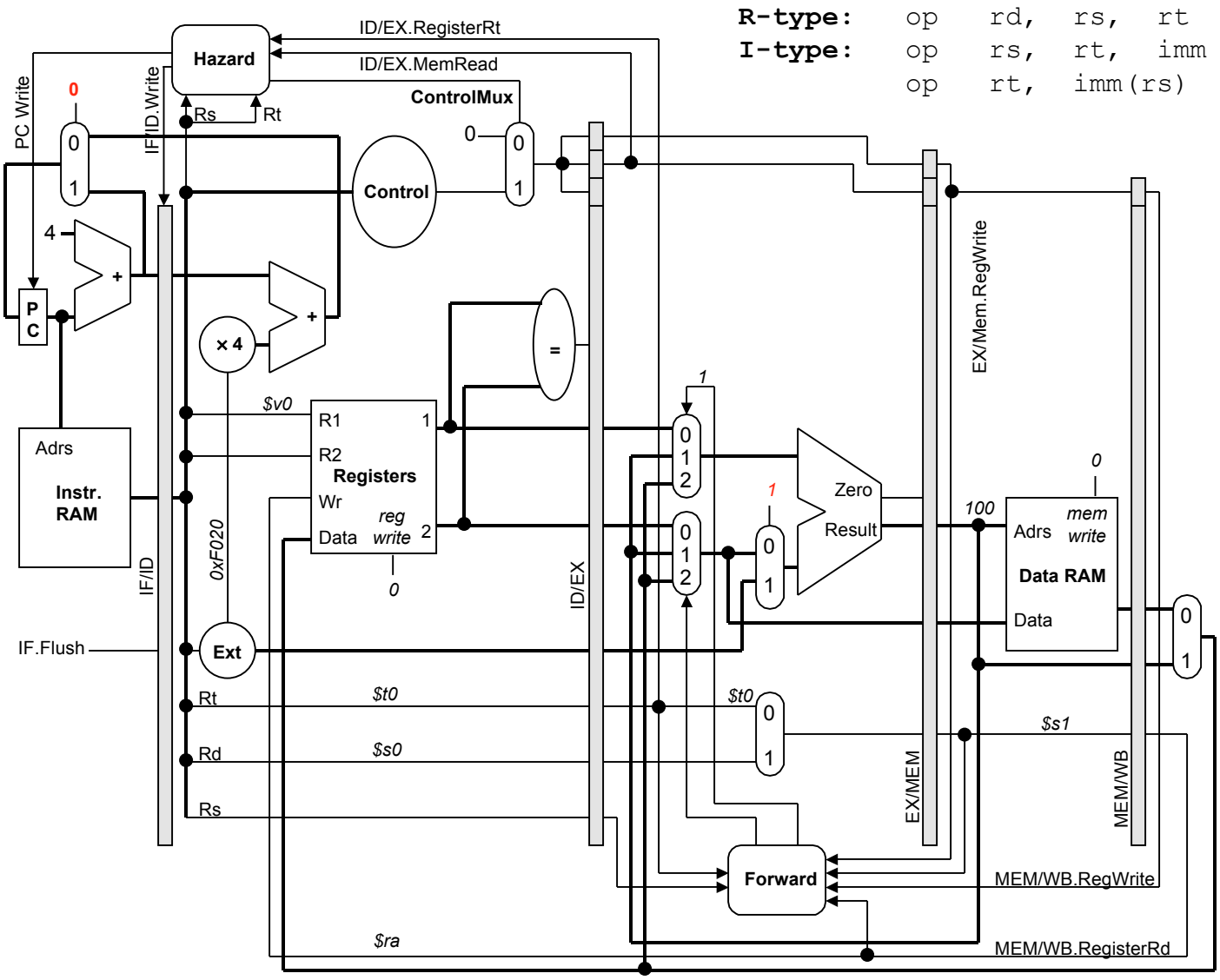
The total CPI can be computed as:

$$CPI_{total} = CPI_{base} + (\#\ \text{stall cycles} / I)$$

Thus, to compute this we need to know: 1) **the fraction of loads & stores**, 2) **the data cache miss rate**, and 3) **the miss penalty**. Alternatively, **the # of misses per instruction** can be substituted for #1 and #2.

Problem 3, More Pipelined Datapath (+6 extra credit points)

Below is a MIPS pipelined datapath, annotated with some signal values. Select the instructions in each pipe stage that are consistent with the annotated signal values. Be careful, **these are tricky** because we've included some values that are correct, but will be ignored (*e.g.,* the memory address is ignored by the instruction in the MEM stage). A good approach to this problem is to eliminate the answers that are inconsistent with the data path. (+2 points each)



IF stage	ID stage	EX stage	MEM stage	WB stage
	a) add $t0, $s0, $v0	a) sub $t0, $s8, $t0		**a) sw $t0, -4($v0)**
	b) bne $t0, $s0, label	b) addi $t0, $s0, 100		b) addi, $ra, $v0, 100
	c) bne $s0, $v0, label	**c) subi $s1, $t0, 100**		c) lw $a1, 16($s1)
	d) add $s0, $v0, $t0	**d) addi $a2, $s1, 100**		d) sll $ra, $s1, $t0
	e) bne $v0, $t0, label	e) sub $t0, $t0, $s1		e) sub $s1, $ra, $a3

ID stage: rs is $v0, rt is $t0, and $s0 would be $s0, but since the PC select mux is set to 0 (highlighted) this instruction must be a branch.
EX stage: this question was a little broken, we know that the instruction operates on an immediate, and that the first register is $s1 since ForwardA is set to 1. According to the description above, this would indicate answer (c), but the true format for I-type instructions is **op rt, rs, imm**, (e.g., answer (d)), but rt is $t0, not $a2.
WB stage: the reg_write signal is 0; the only instruction that doesn't write the register file is the store.