CS232 Midterm Exam 2 Solutions
April 14, 2003

Name: Pooh

- This exam has 7 pages including the pipelined datapath diagram on the last page, which you are free to tear off.
- 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	35	
2	30	
3	35	
Total	100	

Question 1: Multicycle CPU implementation (35 points)

Consider extending the MIPS architecture with the instruction below, which loads two consecutive words of data from memory and stores them into two destination registers.

```
ld  rt, rd, rs       # rt = Mem[rs]; rd = Mem[rs + 4]
```

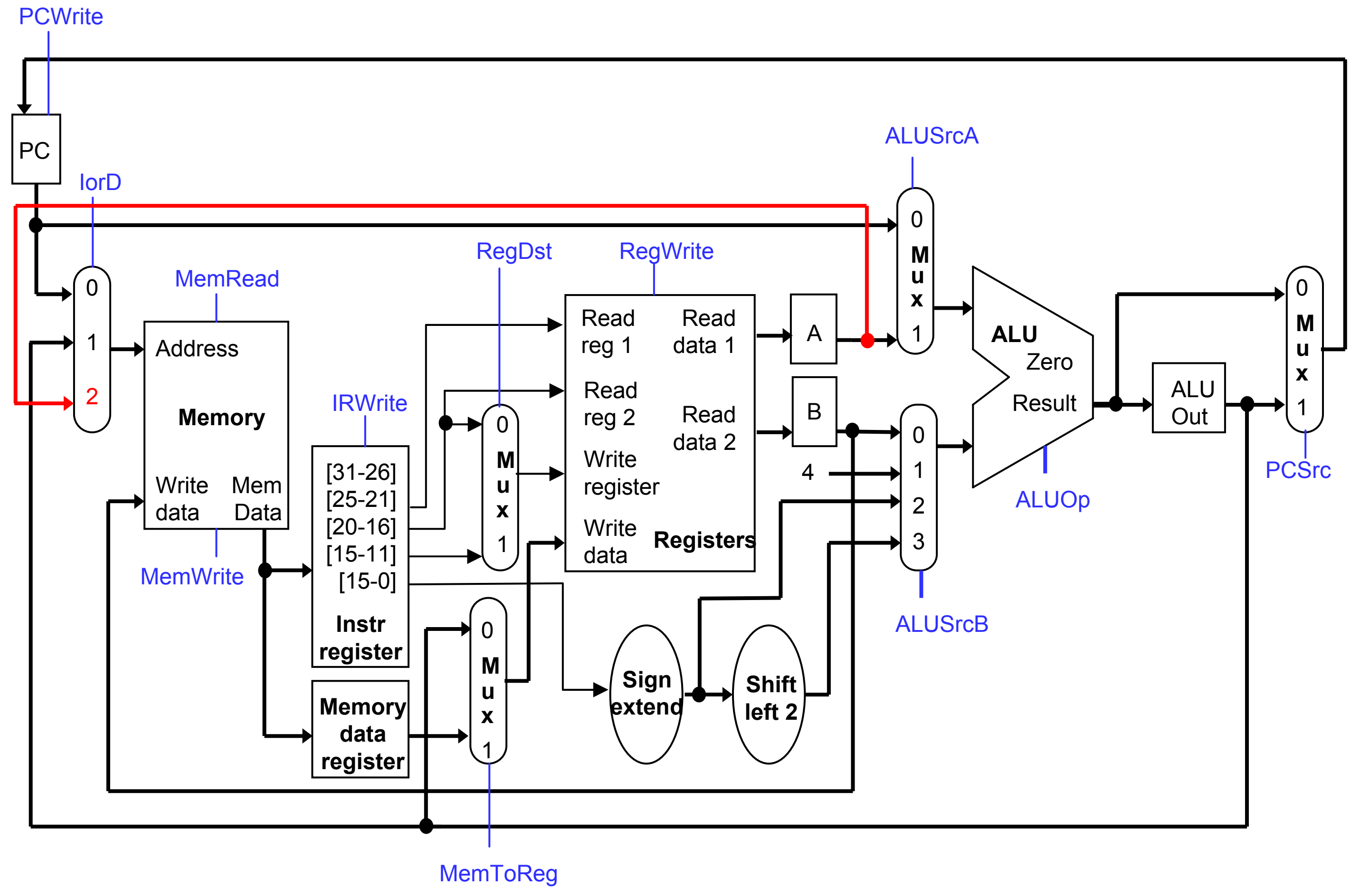
This will use the same format as R-type instructions, shown here for reference (shamt and func are not used).

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

Part (a)

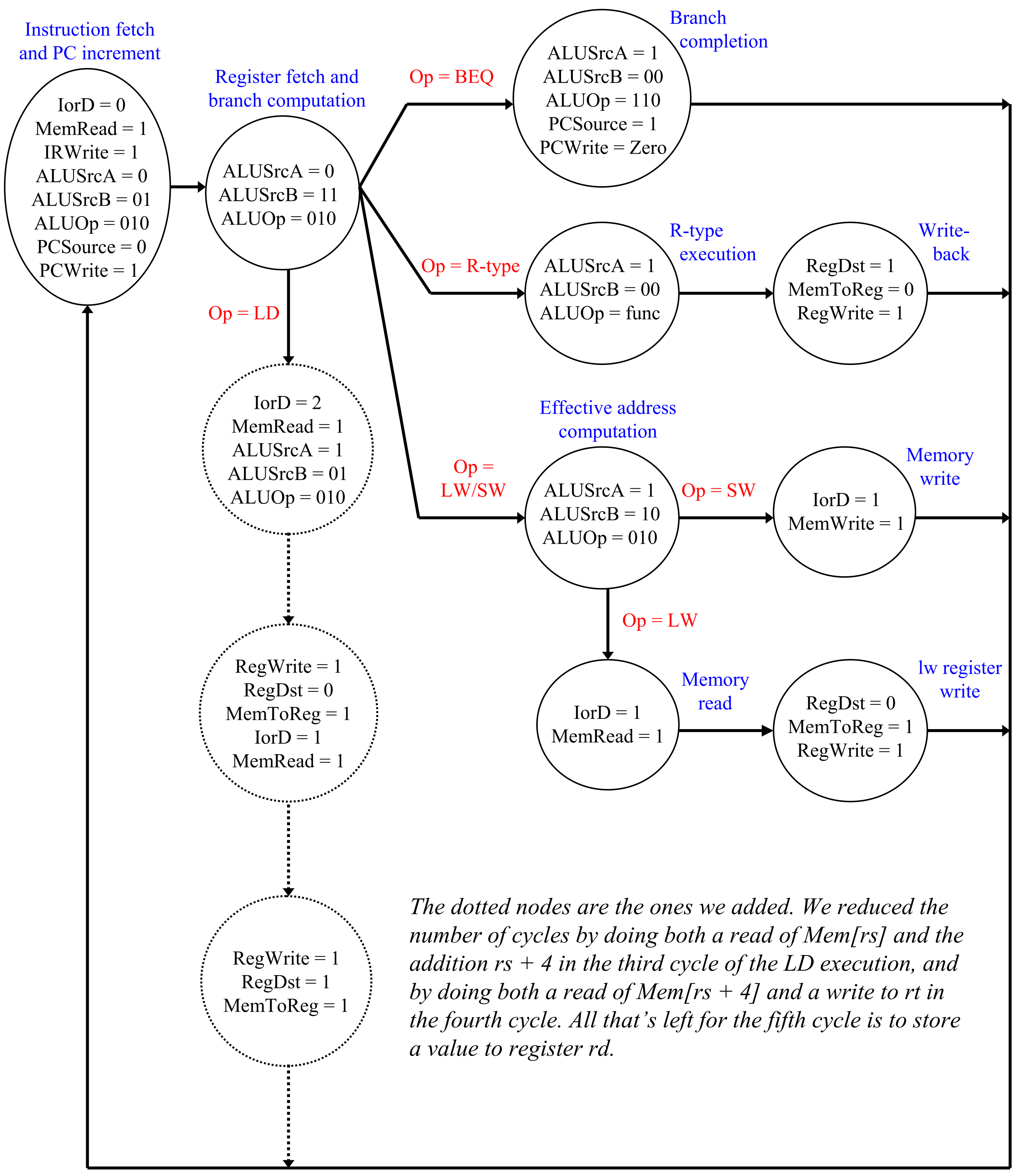
The multicycle datapath from lecture appears below. Show what changes are needed to support *ld*. You should not need to modify the main functional units (the memory, register file and ALU), but you can make any other changes or additions necessary. Try to keep your diagram neat! (10 points)

The ld instruction can be split into several smaller single-cycle operations: fetch and decode the instruction (as usual), read Mem[rs] and store it into register rt, compute rs + 4, and read Mem[rs + 4] and store that into rd. There are a few possible solutions, so we'll just show one of them. The sole datapath change we'll make is to connect A, which contains the value of register rs, to the memory's address input to let us read Mem[rs]. The IorD mux is expanded appropriately.



Question 1 continued

Part (b)
Complete this finite state machine diagram for the *ld* 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)



The dotted nodes are the ones we added. We reduced the number of cycles by doing both a read of Mem[rs] and the addition rs + 4 in the third cycle of the LD execution, and by doing both a read of Mem[rs + 4] and a write to rt in the fourth cycle. All that's left for the fifth cycle is to store a value to register rd.

Question 2: Pipelining performance (30 points)

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 7.

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. (20 points)

```
     Elvis:
1        lw    $t0, 0($a1)
2        mul   $t0, $t0, $a2
3        lw    $t1, 0($a0)
4        add   $t0, $t0, $t1
5        sw    $t0, 0($a0)
6        sub   $a3, $a3, 1
7        addi  $a0, $a0, 4
8        addi  $a1, $a1, 4
9        bne   $a3, $0, Elvis
```

This is very similar to one of the homework problems. First, you should see that the dependencies which may cause a problem are the ones between instructions 1 & 2, 3 & 4 and 4 & 5. Other dependencies, such as the ones between lines 2 & 4 or lines 6 & 9, are too far apart to be potential hazards. The two "lw" dependencies will force a stall, but the dependency in lines 4 & 5 will be resolved by forwarding from the EX/MEM register of the "add" to the EX stage of the "sw." Also, if we predict that branches are not taken as specified on page 7, each execution of "bne" except for the last one will result in a misprediction and one cycle lost to a flush (branches are determined in the ID stage).

If you think about stalls and flushes in terms of "nop" instructions like in our Homework 5 solutions, then 12 instructions are executed for every loop iteration—there are nine instructions originally, plus two stalls and one flush. So running the loop 100 times involves executing 1199 instructions. (We subtract one since the last "bne" will be predicted correctly and does not result in a flush.) With a five-stage pipeline, 1199 instructions would execute in 1199 + 4 = 1203 clock cycles.

Part (b)
Show how you can rearrange the instructions in the code above to eliminate as many stall cycles as possible. You do *not* need to reduce the number of flushes. (10 points)

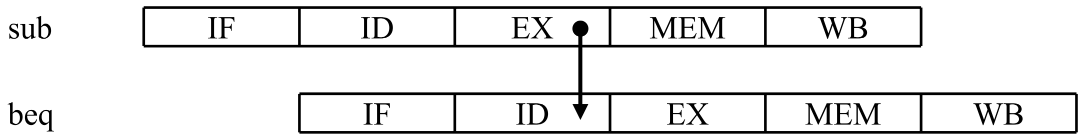
Again, there are only two stalls in each loop iteration due to the "lw" instructions. The simplest solution is to just swap instructions 2 and 3 of the loop. This will separate the load and use of registers $t0 and $t1 by one extra cycle, which is enough to prevent the stalls. This idea is also very similar to one illustrated in Problem 3 of Homework 5, which had two versions of a loop that differed in the number of stalls needed.

Question 3: Forwarding (35 points)

Here is a sequence of two instructions, with a dependency between them.

```
sub   $s1, $s2, $s3
beq   $s1, $0, Pooh
```

If we execute this code using the datapath on page 7, where branches are determined in the ID stage, then the new value of $s1 will be needed in the *same* cycle in which it is produced, as shown in the diagram below.

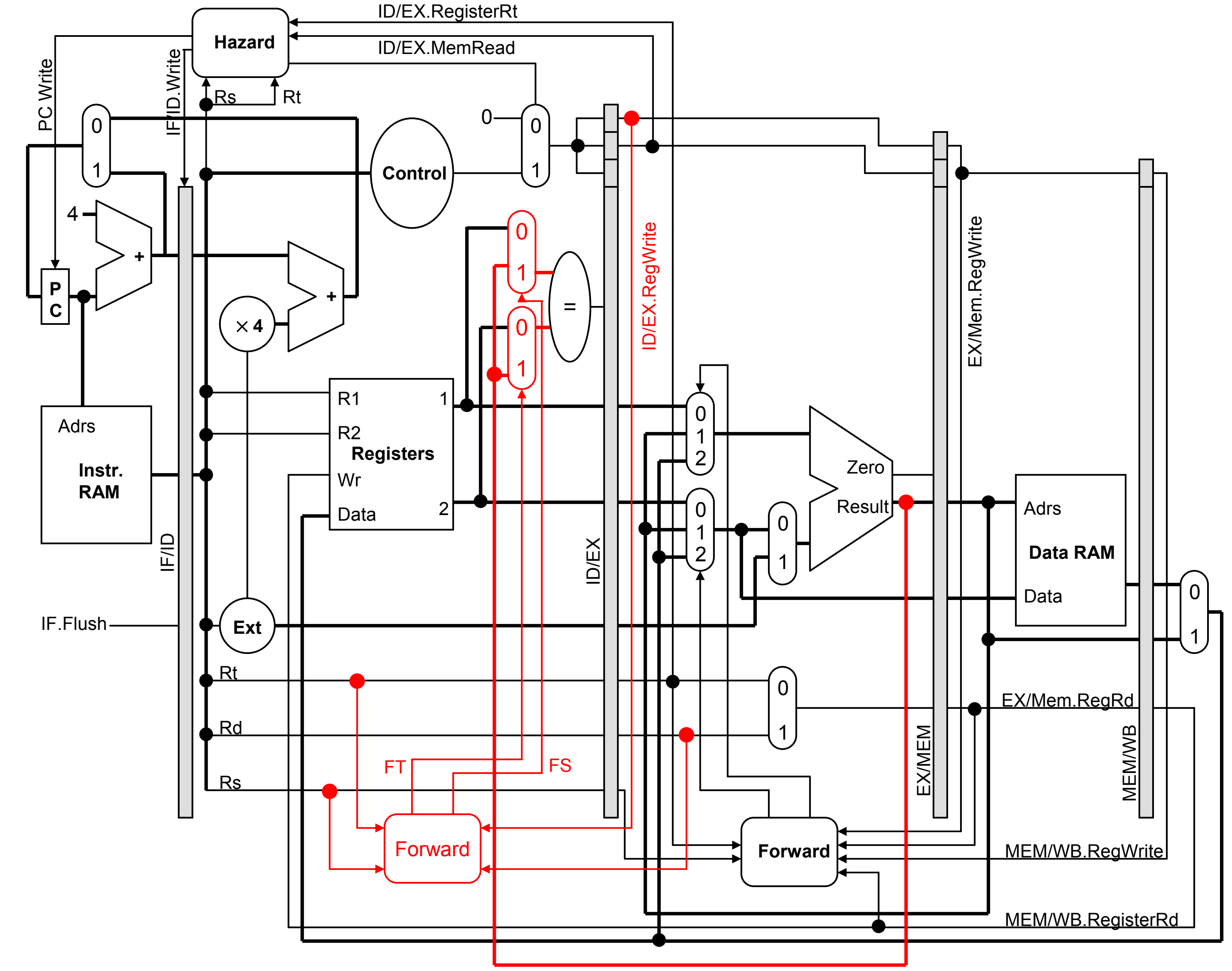


Our datapath doesn't handle this situation properly because it doesn't permit forwarding to the ID stage, so let's try to fix that.

1. Add multiplexers and a forwarding unit to the ID stage of the datapath on the next page (it's the same as the one on page 7). Be sure to show the mux and forwarding unit inputs and outputs clearly.
2. Give forwarding equations to explain how all your mux selections are made.

You can assume the register file and ALU have the same latency while muxes have negligible delays, so you can forward the ALU output directly to the branch comparator. Also, you only need to consider dependencies between R-type arithmetic instructions and branches as in the example above.

See the next page!



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; note that the setup is very similar to that of the existing forwarding hardware in the EX stage. The branch comparator unit can then compare 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. Following the style of the equations shown on page 7:

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

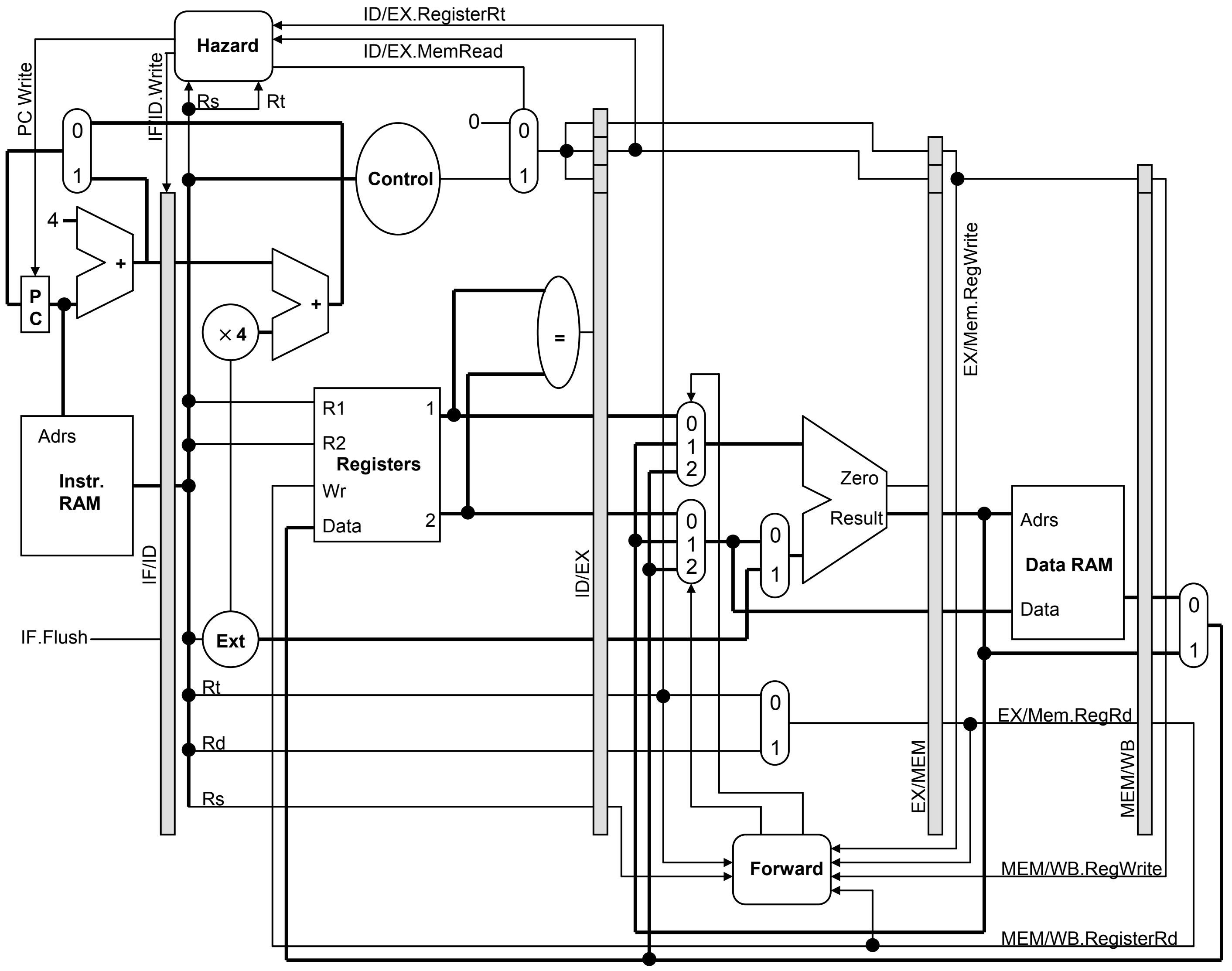
if *(ID/EX.RegWrite = 1* and *IF/ID.RegisterRt = ID/EX.RegisterRd)*
then *FT = 1*

Some people included an additional condition to check if the instruction in ID was "beq," but this shouldn't be necessary since the Control unit only uses the comparator result for "beq" anyway.

Pipelined Datapath

Here is the final datapath that we discussed in class.

- Forwarding for arithmetic operations 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, and branch determination is done in the ID stage.
- Equations for the ForwardA mux are given below; the ForwardB mux is similar.



```
if  (MEM/WB.RegWrite = 1
     and MEM/WB.RegisterRd = ID/EX.RegisterRs
     and EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
then ForwardA = 1

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