EECS 361 Homework 3
Fall 2006
Due: 11/09/06

1. [5] Describe the effect that a single stuck-at-0 fault (i.e., regardless of what it should be, the signal is always 0) would have on the multiplexors in the single-cycle datapath in Figure 1. Which instructions, if any, would still work? Consider each of the following faults separately: RegDst = 0, ALUSrc = 0, MemtoReg = 0, Zero = 0. Fill in the provided table below.

Working Instructions for stuck-at-0 faults

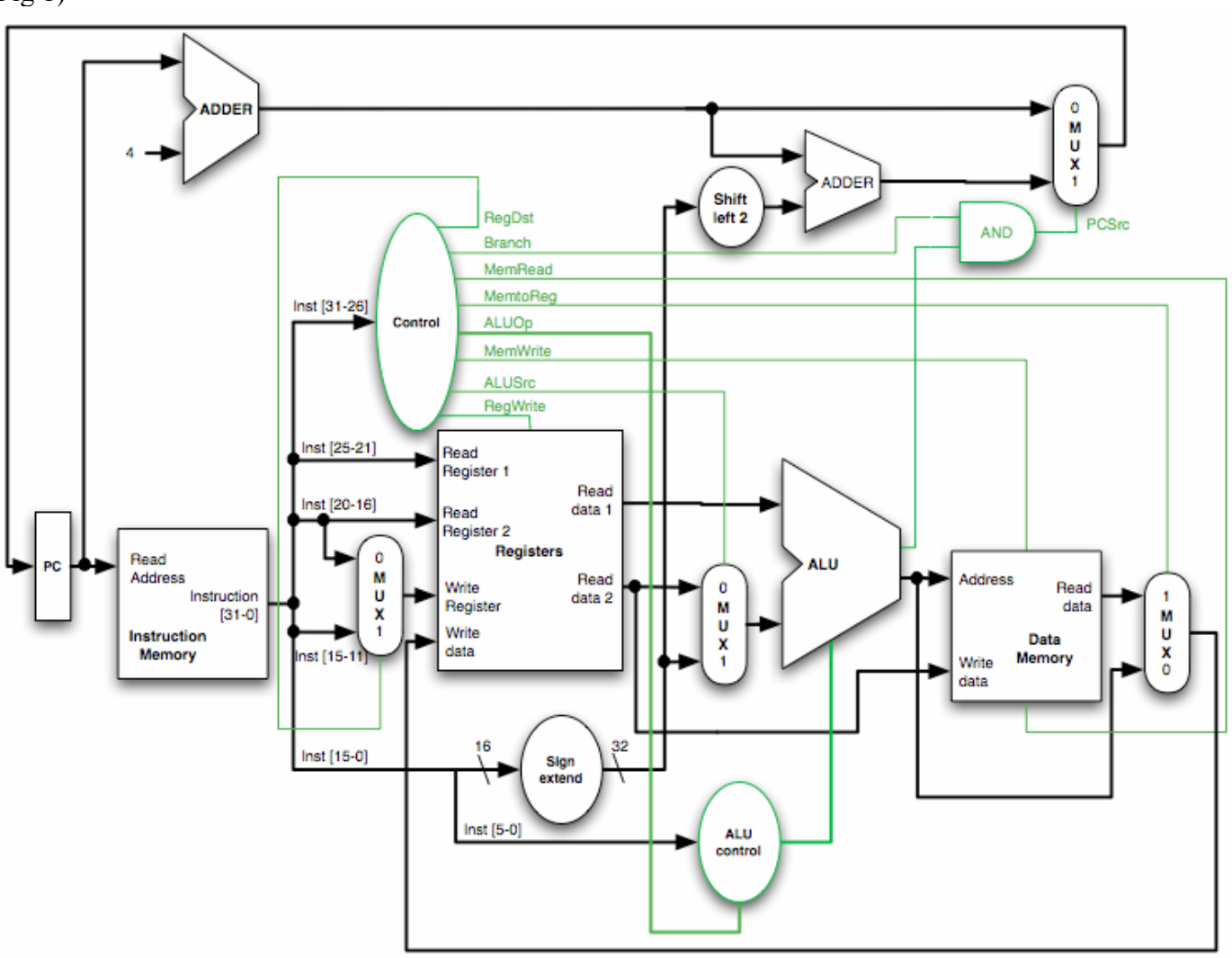
	RegDst = 0	ALUSrc = 0	MemtoReg = 0	Zero = 0
Loads				
Stores				
R-type				
Branch				
Jump				

2. [5] This exercise is similar to problem 1, but this time consider stuck-at-1 faults (the signal is always 1).

Working Instructions for stuck-at-1 faults

	RegDst = 1	ALUSrc = 1	MemtoReg = 1	Zero = 1
Loads				
Stores				
R-type				
Branch				
Jump				

(Fig 1)


ADDER
4
ADDER
Shift left 2
0 M U X 1
AND
PCSrc
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Inst [31-26]
Inst [25-21]
Inst [20-16]
Inst [15-11]
Inst [15-0]
Inst [5-0]
PC
Read Address
Instruction [31-0]
Instruction Memory
Read Register 1
Read Register 2
Registers
Write Register
Write data
Read data 1
Read data 2
ALU
Address
Read data
Write data
Data Memory
1 M U X 0
16
Sign extend
32
ALU control

3. [20] Say the critical path that sets the clock cycle length of a CPU with a **multicycle datapath** is the memory access stage for loads and stores (*not* for instructions). This has caused this particular CPU to run at a clock rate of 500 MHz rather than the intended target clock rate of 750 MHz. However, if all the cycles that access memory are broken into two clock cycles, then the machine can run at its target clock rate. Using the gcc mixes shown in Table 1, how many times faster the machine with the two-cycle memory accesses is compared with the 500-MHz machine with single-cycle memory access. Assume that all jumps and branches take the same number of cycles and that the set instructions and arithmetic immediate instructions are implemented as R-type instructions. (Minor Hint: What does `lui` do?)

(Table 1)

Core MIPS	Name	gcc	Arithmetic core + MIPS I	Name	gcc
add	`add`	0%	branch on equal (zero)	`beq`	9%
add immediate	`addi`	0%	branch on not equal (zero)	`bne`	8%
add unsigned	`addu`	9%	jump and link	`jal`	1%
add immediate unsigned	`addiu`	17%	jump register	`jr`	1%
subtract unsigned	`subu`	0%	set less than	`slt`	2%
and	`and`	1%	set less than immediate	`slti`	1%
and immediate	`andi`	2%	set less than unsigned	`sltu`	1%
shift left logical	`sll`	5%	set less than imm. Uns.	`sltiu`	1%
shift right logical	`srl`	0%	shift right arithmetic	`sra`	2%
load upper immediate	`lui`	2%	load half	`lh`	1%
load word	`lw`	21%	branch less than zero	`bltz`	1%
store word	`sw`	12%	branch greater or equal zero	`bgez`	1%
load byte	`lb`	1%	branch less or equal zero	`blez`	0%
store byte	`sb`	1%			

4. [10] Identify all the data dependencies in the following code. Which dependencies are data hazards that will be resolved via forwarding?

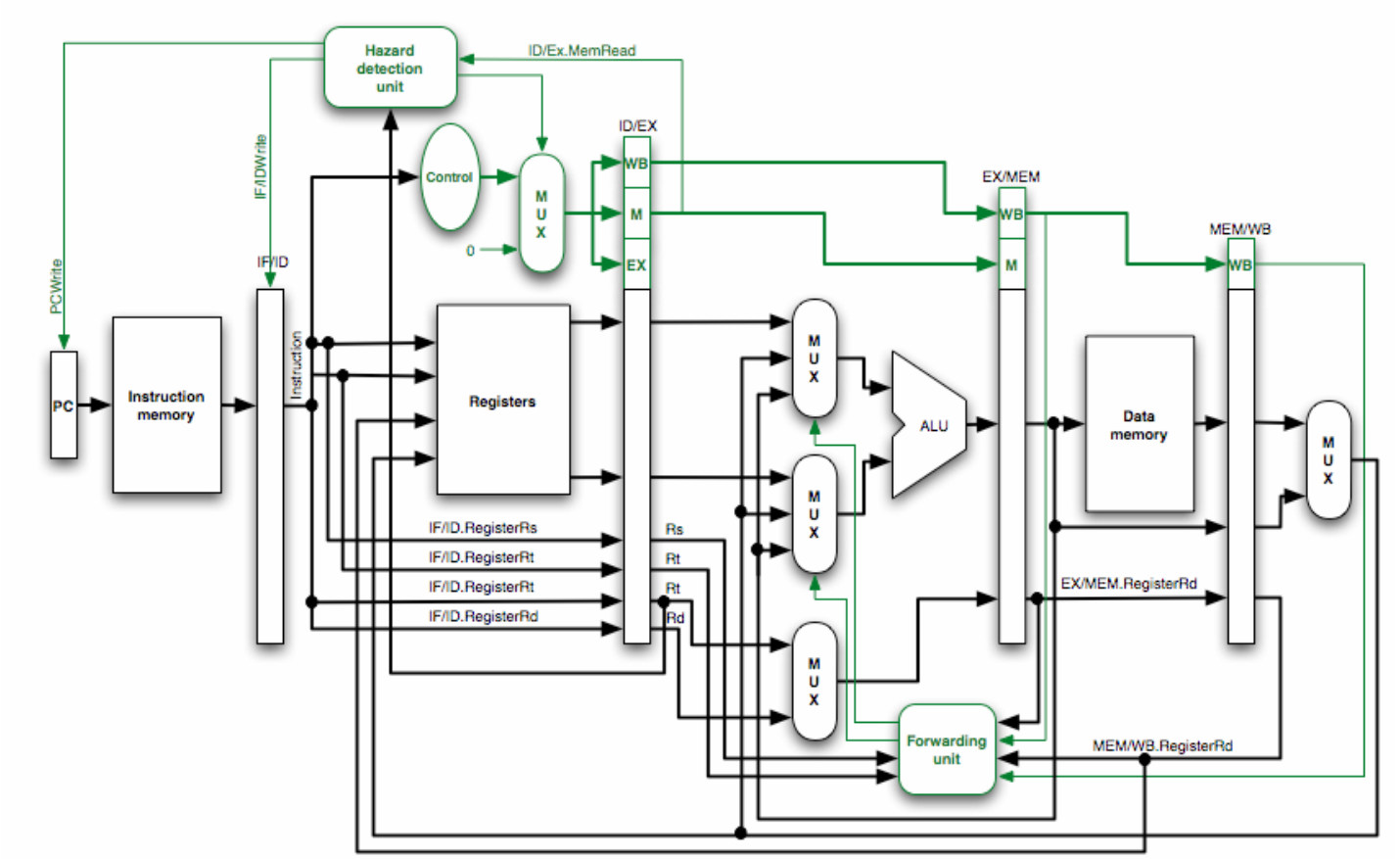
```
add $2, $5, $4
add $4, $2, $5
sw  $5, 100($2)
add $3, $2, $4
```

5. [5] With regard to the following program executing on the pipelined datapath of Figure 2.

```
add $1,  $2,  $3
add $4,  $5,  $6
add $7,  $8,  $9
add $10, $11, $12
add $13, $14, $15
```

Explain what the forwarding unit is doing *during* the fifth cycle of execution. If any comparisons are being made, mention them.

(Fig 2)



6. [20] Consider an instruction sequence used for a memory-to-memory copy:

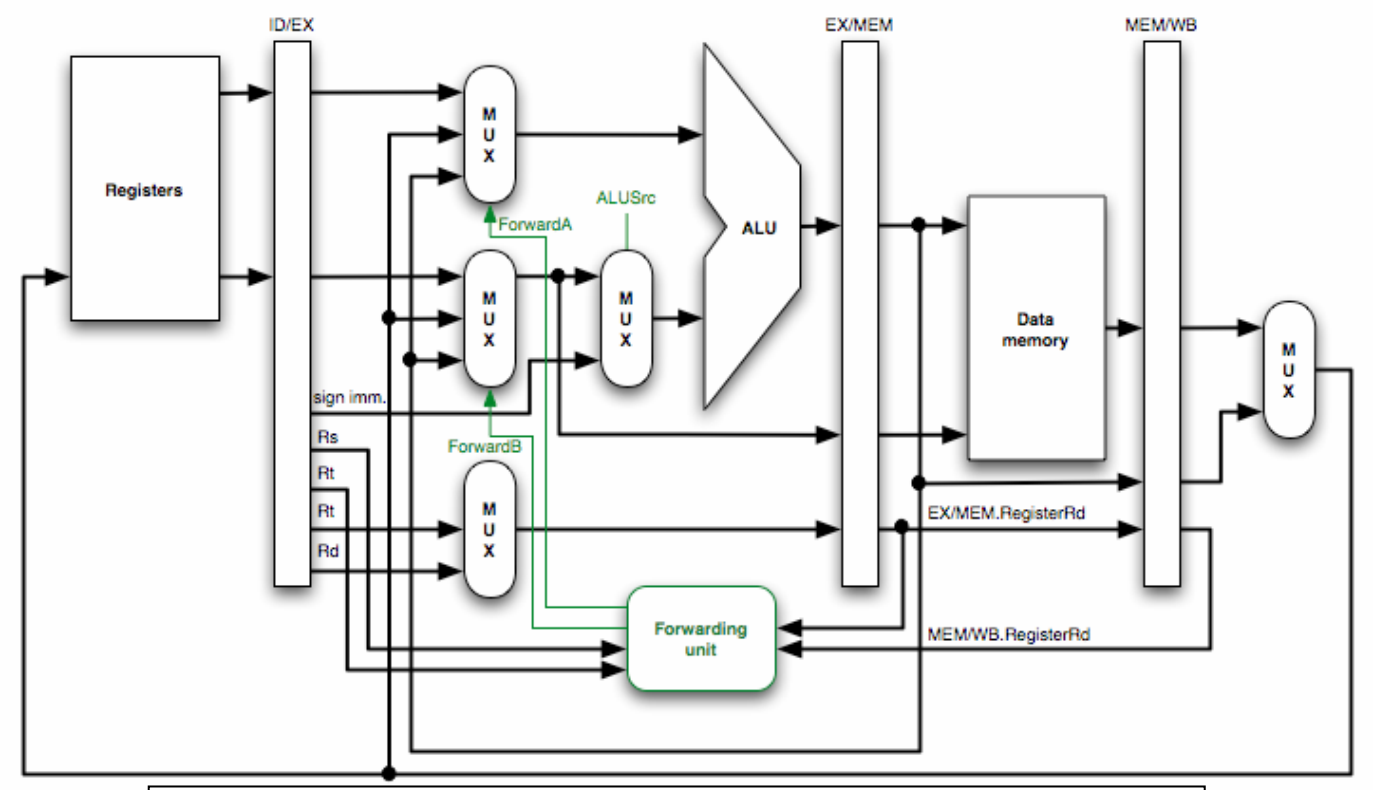
```
lw  $2, 100($5)
sw  $2, 200($6)
```

The elaboration discusses this situation and states that additional forwarding hardware can improve its performance. Show the necessary additions to the datapath of Figure 3 to allow code like this to run without stalling. Include forwarding equations (such as the ones appearing on the next page) for all of the control signals for any new or modified multiplexors in your datapath. Finally, rewrite the stall formula on the next page so that this code sequence won't stall.

Elaboration: Forwarding can also help with hazards when store instructions are dependent on other instructions. Since they use just one data value during the MEM stage, forwarding is easy. But consider loads immediately followed by stores. We need to add more forwarding hardware to make memory-to-memory copies run faster. With a `sw` that immediately follows a `lw`, a stall can be avoided since the data exists in the MEM/WB register of a load instruction in time for its use in the MEM stage of a store instruction. We would need to add forwarding into the memory access stage for this option.

The signed-immediate input to the ALU, needed by loads and stores has been added in the datapath in Figure 3. Since central control decides between register and immediate, and since the forwarding unit chooses the pipeline register input to the ALU, the easiest solution is to add a 2:1 multiplexor that chooses between the ForwardB multiplexor output and the signed immediate. Figure 3 shows this addition. Note that the solution differs from a modification where the multiplexor controlled by line ALUSrcB was expanded to include the immediate input. This solution also solves store forwarding by connecting the forwarding multiplexor output-containing store data in this case-to the EX/MEM pipeline register.

(Fig 3)



Forwarding Equation Examples:

EX hazard:

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

MEM hazard:

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

Stall Formula:

If (ID/EX.MemRead and
((ID/EX.RegisterRt = IF/ID.RegisterRs) or
(ID/EX.RegisterRt = IF/ID.RegisterRt)))
stall the pipeline