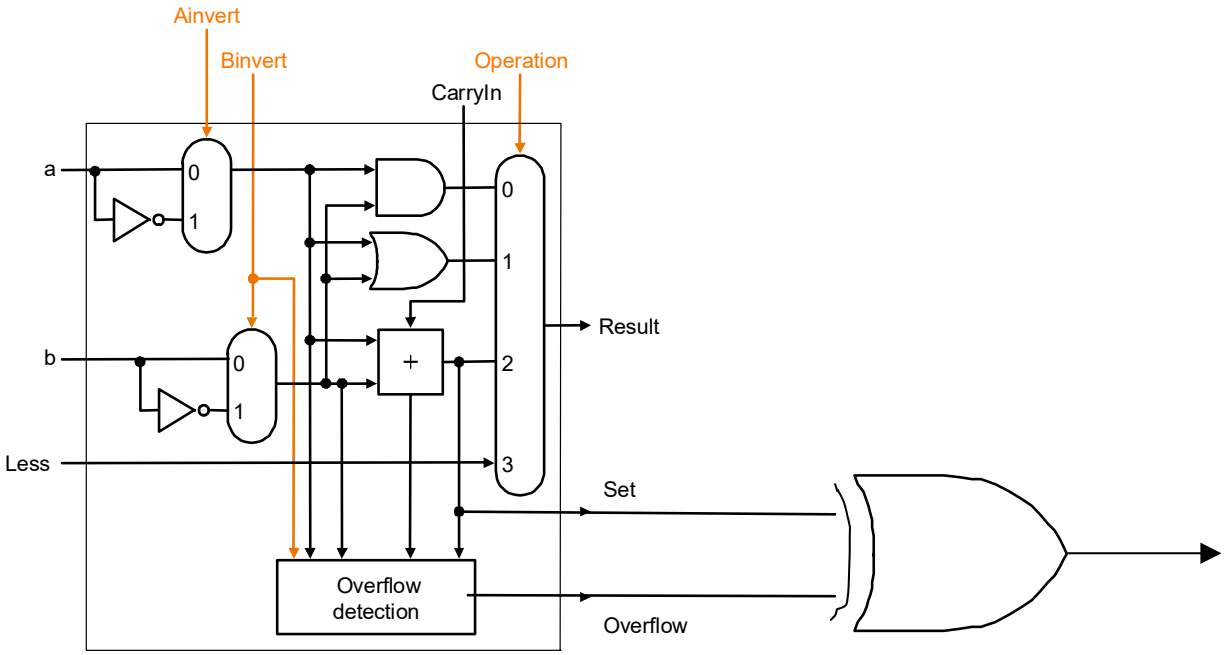
CS151B/EE116C – Solutions to Homework #2

Problem (1) B.24

If there is no overflow, the circuitry shown in Figure B.5.10 is sufficient – the "Set" output from bit 31 (the sign bit) can be used as the "Less" input for bit 0. However, if there is an overflow, the inverse of the sign bit must be used as the "Less" input for bit 0. Both cases are handled correctly by the following circuitry (replacing the bottom part – bit 31 – of Figure B.5.10):



"Less" input of bit 0

Problem (2) B.25

Given that a number that is greater than or equal to zero is termed positive and a number that is less than zero is negative, inspection reveals that the last two rows of Figure 3.2 restate the information of the first two rows. Because A – B = A + (–B), the operation A – B when A is positive and B negative is the same as operation A + B when A is positive and B is positive. Thus the third row restates the conditions of the first. The second and fourth rows refer also to the same condition.

Because subtraction of two's complement numbers is performed by addition, a complete examination of overflow conditions for addition suffices to show also when overflow will occur for subtraction. Begin with the first two rows of Figure 3.2, and add rows for A and B with opposite signs. Build a table that shows all possible combinations of Sign and CarryIn to the sign bit position and derive the CarryOut, Overflow, and related information. Thus,

Sign A	Sign B	Carry In	Carry Out	Sign of Result	Correct Sign of Result	Overflow ?	Carry In XOR Carry Out	Notes
0	0	0	0	0	0	No	0	
0	0	1	0	1	0	Yes	1	Carries differ
0	1	0	0	1	1	No	0	|A| < |B|
0	1	1	1	0	0	No	0	|A| > |B|
1	0	0	0	1	1	No	0	|A| > |B|
1	0	1	1	0	0	No	0	|A| < |B|
1	1	0	1	0	1	Yes	1	Carries differ
1	1	1	1	1	1	No	0	

From this table an Exclusive OR (XOR) of the CarryIn and CarryOut of the sign bit serves to detect overflow. When the signs of A and B differ, the correct sign of result is determined by the relative magnitudes of A and B, as listed in the Notes column.

Problem (3)

The stuck-at-1 error at the bit 1 of the control lines will cause incorrect results for all instructions that require the ALU actions that needs the bit 1 of the control lines to be 0. Specifically:

A) The `and` instruction will compute Rs + Rt instead of Rs & Rt.
B) The `or` instruction will set Rd to 1 when Rs + Rt < 0, and will set Rd to 0 otherwise.
C) The `nor` instruction will set Rd to – Rs – 1 – Rt instead of Rs NOR Rt.

Problem (4) 4.6.4

The fault will potentially change the function performed by the processor only for an instruction where the MemRead signal has to be 1 but the RegDst signal may be 0. This is the case only for the lw instruction.

If we could be sure that when MemRead is negated, the "Read data" output of the data memory is 0, we could use the following test:

The state of the processor before starting the test:

1) The value stored in the PC is 0.
2) The value stored in registers $2 is 0.
3) The value stored in register $3 is 0x4000.
4) The values in all the registers not mentioned in items 1-3 do not matter.
5) The word stored in the data memory beginning from address 0x4000 is 0x00000001.
6) The state of the data memory not mentioned in item 5 does not matter.
7) For the instruction memory, the instruction used in the test is stored in word 0 and the rest of the contents do not matter

Test Instruction:

Address	Instruction	Machine code
0x00	lw $2, 0($3)	100011 00011 00010 0000000000000000

After the execution of the test instruction, the expected value in the $2 is 0x01. However, if the fault we are testing for is present, the value in register $2 will be 0.

Unfortunately, when MemRead=0, the memory is not read and the value on the "Read data" output of the data memory is random (unpredictable). Hence, it is possible for this value to be the same as the value stored in data memory at location 0x4000 (i.e., 0x01). Thus, there is no test that can detect this fault with 100% probability. A test that includes several lw instructions, loading several different values from memory, is highly likely to detect the fault.

Problem (5) 4.6.5

To test for this fault, we need an instruction for which the Jump control signal is 1. Hence, it has to be the "`j`" (jump) instruction. However, for the jump instruction, the RegDst signal is a don't care because the instruction does not write to any registers. Hence, the implementation may or may not set RegDst to 0. As a result, we cannot reliably test for this fault.

Problem (6)

The PC and $R[IR_{20..16}]$ are changed:

$PC \leftarrow PC + 4$

$R[IR_{20..16}] \leftarrow R[IR_{25..21}] + R[IR_{20..16}]$

Problem (7)

No additions to the datapath are required. The problem specifies that the instruction will use the R-format. Since an opcode is not specified, we must pick an opcode that is currently unused in order to avoid a conflict with existing instructions. Based on Figure A.10.2, page A-50, we choose the opcode 0x37:

`lwrr rd, rt, rs`

110111	Rs	Rt	Rd	0
6	5	5	5	11

A new row should be added to the truth table in Figure 4.18. Since this is an R-format instruction, the destination register is Rd (RegDst = 1). The instruction is similar to `lw` in that it reads the memory (RegWrite = 1, MemtoReg = 1, MemRead = 1, MemWrite = 0), but, unlike `lw`, it uses the ALU to compute Rs + Rt (ALUSrc = 0, ALUOp = 00).

Instr	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
lwrr	1	0	1	1	1	0	0	0	0

Problem (8)

`jr rs`

0	rs	0	0x8
6	5	15	6

0x8 = 001000

Instruction `jr` has the same opcode as R-type instructions. However, `jr` can be distinguished from the other implemented R-type instructions based on the most significant bit of the funct field. Specifically, bit 5 of the `jr` instruction is 0 while, as shown in Figure 4.12 in the textbook, bit 5 of all the implemented R-type instructions is 1.

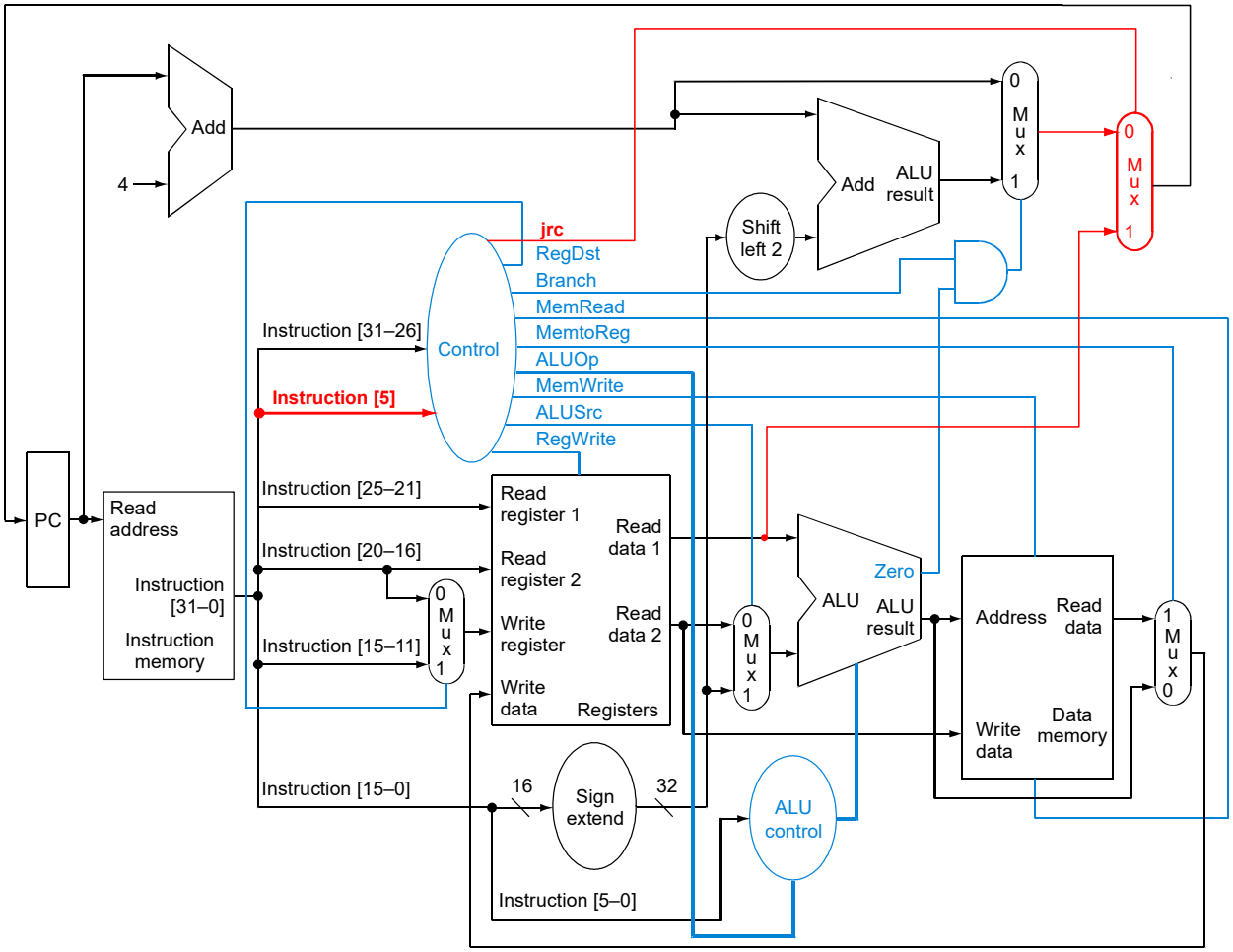
Thus, bit 5 from the output of the instruction memory must be a new input to the main part of the control unit.

A) Modifications to the datapath:
Add a new multiplexor to determine whether the next PC value will be as before (the output of the PCSrc MUX) or the value of register R[Rs]. Bit 5 from the instruction memory must be connected to the main part of the control unit. (see next page)

B) jrc (jr control) – controls the new multiplexor.

C) The control logic is modified. The 5 bit of instruction is added to the control logic input to distinguish jr instruction from other R-type instructions. New control signal jrc is added to the control logic output. When an instruction is jr, jrc sets to 1, RegWrite and MemWrite are deasserted, and other control signals are don't cares.

Input or output	Signal name	R-format	lw	sw	beq	jr
Inputs	Op5	0	1	1	0	**0**
	Op4	0	0	0	0	**0**
	Op3	0	0	1	0	**0**
	Op2	0	0	0	1	**0**
	Op1	0	1	1	0	**0**
	Op0	0	1	1	0	**0**
	Instruction[5]	**1**	**X**	**X**	**X**	**0**
Outputs	RegDst	1	0	X	X	**X**
	ALUSrc	0	1	1	0	**X**
	MemtoReg	0	1	X	X	**X**
	RegWrite	1	1	0	0	**0**
	MemRead	0	1	0	0	**X**
	MemWrite	0	0	1	0	**0**
	Branch	0	0	0	1	**X**
	ALUOp1	1	0	0	0	**X**
	ALUOp0	0	0	0	1	**X**
	jrc	**0**	**0**	**0**	**0**	**1**



Problem (8) Datapath

Solutions to Homework #2 Practice Problems

Problem (9)

$$A \text{ OR } \overline{B}$$

With control lines set to 0101, the `Binvert` input to the ALU would be set and the output of the OR gates is selected as the result.

Problem (10)

The only change will be on the result from the `slt` instruction. Specifically, instead of generating 000…00000 or 000…00001 as the comparison result, the instruction will generate 000…01000 or 000…01001, respectively.

Problem (11)

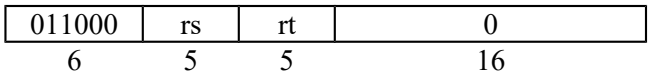
No additions to the datapath are required. A new row should be added to the truth table in Figure 4.18. The new control is similar to load word because we want to use the ALU to add the immediate to a register (and thus RegDst = 0, ALUSrc = 1, ALUOp = 00). The new control is also similar to an R-format instruction, because we want to write the result of the ALU into a register (and thus MemtoReg = 0, RegWrite = 1), and of course we are not branching or using memory (Branch = 0, MemRead = X, MemWrite = 0).

Instr	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
addi	0	1	0	1	X	0	0	0	0

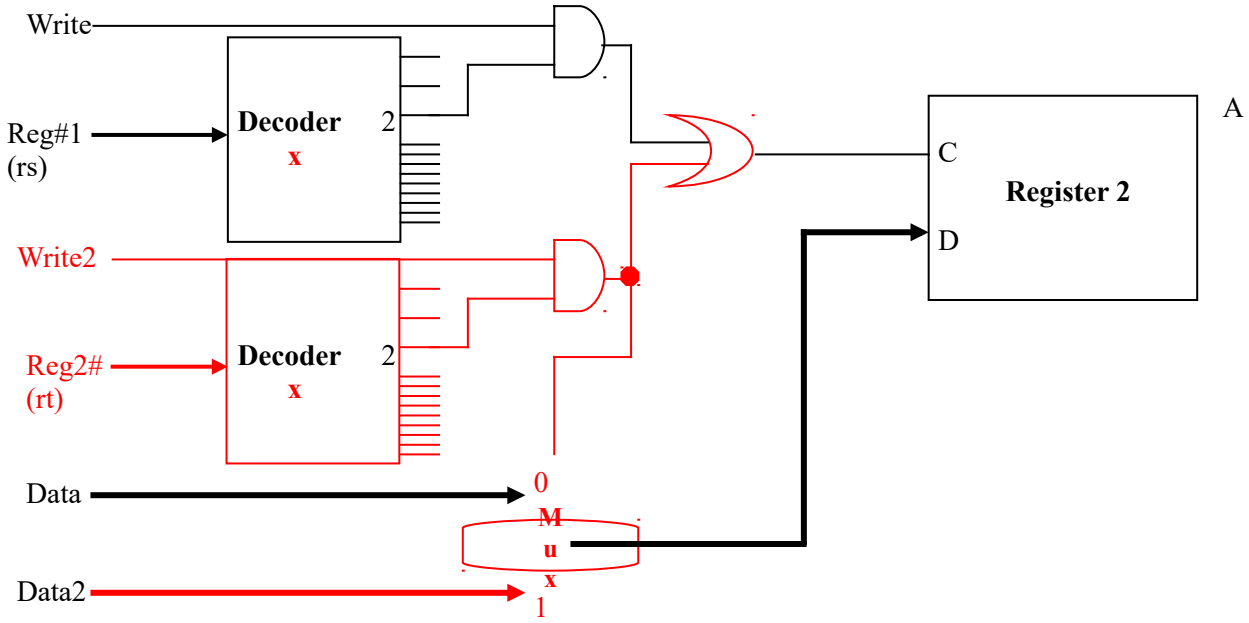
Problem (12)

We pick an unused opcode, 0x18, and we choose the I-format with zero for the immediate:

`swap rs, rt`



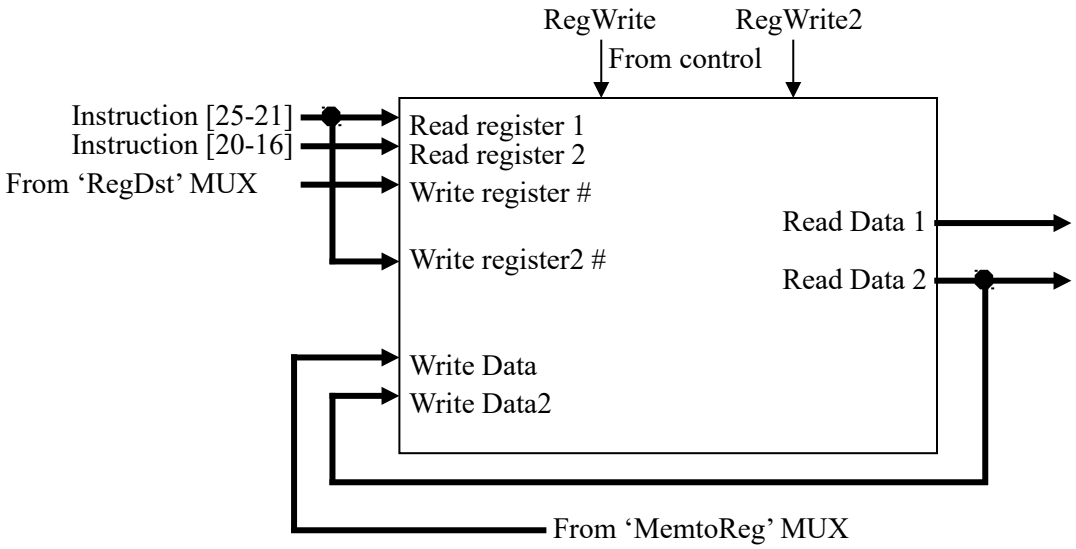
The original register file needs to be modified because it cannot write to two registers in the same cycle. We change the register file to have two write ports: two ‘register number’, two ‘register data’ and two ‘write’ inputs. Modifications relevant to register number 2 are shown on the following diagram, note that all registers have to be modified similarly.



'MUX', an 'OR' gate and two 'AND' gates are needed for each register. Two decoders are needed.

Instr	Reg-Dst	ALU-Src	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALU-Op1	ALU-Op0	Reg Write2
R-type	1	0	0	1	0	0	0	1	0	**0**
Lw	0	1	1	1	1	0	0	0	0	**0**
Sw	X	1	X	0	0	1	0	0	0	**0**
Beq	X	0	X	0	0	0	1	0	1	**0**
Swap	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**X**	**1**

Note: Read register 1 is Rs (Inst[25-21]) and Read register 2 is Rt (Inst[20-16]), but Write register 1 is Rt and Write register 2 is Rs. Therefore, Read Data 1 should be connected to Write Data and Read Data 2 should be connected to Write Data2 (as shown).



Problem (13)

Looking at Figure 4.18, we see that MemtoReg and MemRead are identical except for `sw` and `beq`, for which MemtoReg is a "don't care". Thus, the modification will work for the single-cycle datapath.

Problem (14)

The Zero output from the ALU is used by the `beq` instruction to decide whether to take the branch or not. If the Zero output is always 1, the `beq` instruction is always taken; it becomes an unconditional branch.

Problem (15)

Add a new control signal Eq which is 1 if executing `beq` instruction and 0 otherwise (`bne` instruction).

Instr	Reg Dst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALU-Op1	ALU-Op0	Eq
R-type	1	0	0	1	0	0	0	1	0	**X**
lw	0	1	1	1	1	0	0	0	0	**X**
sw	X	1	X	0	0	1	0	0	0	**X**
beq	X	0	X	0	0	0	1	0	1	**1**
bne	**X**	**0**	**X**	**0**	**X**	**0**	**1**	**0**	**1**	**0**

The signal Eq is combined with the 'zero' output of the ALU to produce a ConditionMet signal, which signifies that the branching condition has been met, with the following truth table:

Eq	Zero	Not Zero	ConditionMet
0	0	1	1
0	1	0	0
1	0	1	0
1	1	0	1

The ConditionMet signal is ANDed together with the Branch signal to control the multiplexor, as shown in the following diagram:

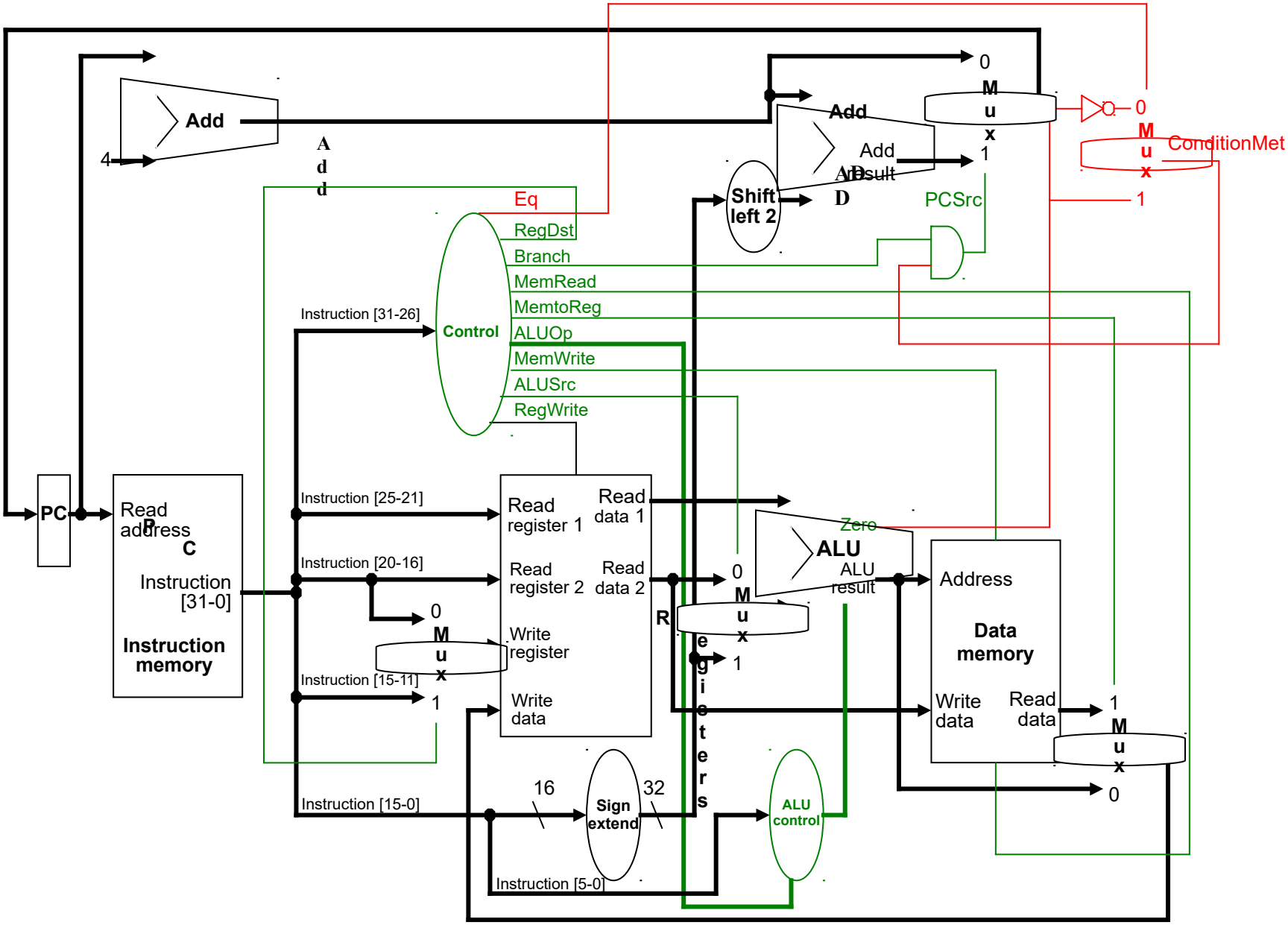


Figure for Problem (15) Datapath and control for problem 15. The changes relative to Figure 4.17 in the book are: the MUX at the top-right corner and the Eq control signal.

Problem (16)

The register file input "Write Register" is the register number of the destination register of R-type instructions (e.g., add, sub) and some I-type instructions (e.g., lw).

If bit 0 of the input Write Register (signal a) is stuck-at-0, then any instruction that specifies an odd number register as the destination, will, instead, write to an even number register:
original destination register number -1.

The state of the processor before starting the test:

8) The value stored in the PC is 0.
9) The value stored in registers $4, and $5 is 0.
10) The value stored in register $10 is 1.
11) The values in all the registers not mentioned in items 1-3 do not matter.
12) The state of the data memory does not matter.
13) For the instruction memory, the instruction used in the test is stored in word 0 and the rest of the contents do not matter

Test Instruction:

Address	Instruction	Machine code
0x00	add $5, $0, $10	000000 00000 01010 00101 00000 100000

After the execution of the test instruction, the expected value in the $5 is 0x01. However, if the fault we are testing for is present, the value in register $5 will be 0.

Problem (17)

The register file input "Write Register" is the register number of the destination register of R-type instructions (e.g., add, sub) and some I-type instructions (e.g., lw).

If bit 0 of the input Write Register (signal a) is stuck-at-1, then any instruction that specifies an even number register as the destination, will, instead, write to an odd number register:
original destination register number +1.

The state of the processor before starting the test:

1) The value stored in the PC is 0.
2) The value stored in registers $4, and $5 is 0.
3) The value stored in register $10 is 1.
4) The values in all the registers not mentioned in items 1-3 do not matter.
5) The state of the data memory does not matter.
6) For the instruction memory, the instruction used in the test is stored in word 0 and the rest of the contents do not matter

Test Instruction:

Address	Instruction	Machine code
0x00	add $4, $0, $10	000000 00000 01010 00101 00000 100000

After the execution of the test instruction, the expected value in the $4 is 0x01. However, if the fault we are testing for is present, the value in register $5 will be 1.

In general, to detect a stuck-at-0 fault on some signal X, the test uses an initial state and a test instruction that, in a fault-free circuit, will drive a 1 onto X. Similarly, to detect a stuck-at-1 fault on some signal X, the test uses an initial state and a test instruction that, in a fault-free circuit, will drive a 0 onto X. Since it is not possible to simultaneously drive both a 0 and a 1 onto the same signal, it is impossible for a single test to detect both a stuck-at-0 fault and a stuck-at-1 fault on the same signal.