Final Review

Format

- 10 multiple choice – 8 points each
  - Make sure to show your work
  - Can write a description to the side as to why you think your answer is correct for possible partial credit
- 1 short answer – 20 points
  - Broken into 5 5point sub problems
- Same basic principals as the midterm

What to Know: Pre-Midterm

- 30% of multiple choice
- MIPS coding
- Emphasis on:
  - floating point
  - strings
  - arrays/loops

What to Know: Post-Midterm

- 70% of multiple choice and short answer
- How to generate truth tables
- How to write logic functions
- How to use k-maps
- How to generate simplified circuits
- How handle state based circuits
  - Counters, sequence detectors, etc
- How to show processor data flow for one or more instructions and those only
- How to generate control signals in the processor

What will NOT be on the exam

- Processor component specifics
  - ALU
  - Registers
  - Memory
- Verilog
- Gate diagrams
- Gates other than AND, OR, NOT, XOR
- Conceptual questions (everything will be problem solving like the homeworks)
- Non-MIPS processors

Semester Review

Interger Number Representations

- To convert an unsigned decimal number to binary: you divide the number N by 2, let the remainder be the first digit. Then divide the quotient by 2, then let the remainder be d1, then divide the quotient by 2, then let the remainder be d2, until the quotient is less than 2.
- 2's complement. To convert a negative number to binary: invert each bit, and then add 1.

Problem

The binary representation of -57_{ten} in 8 bits in 2's complement is

(a) 11000111

(b) 10011111

(c) 11010111

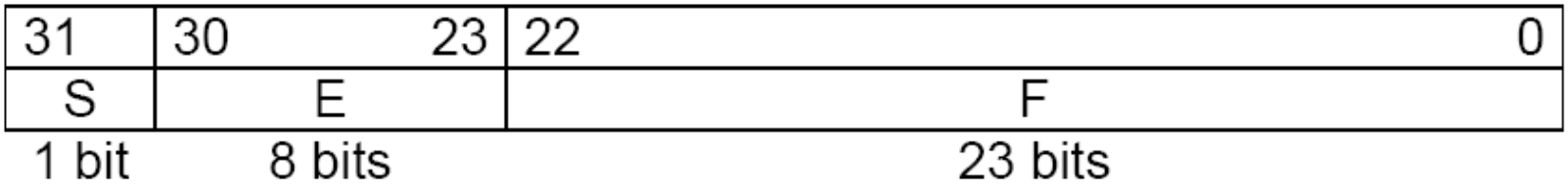
(d) None of the above.

Number with Fractions

- Numbers with fractions. Convert the integer part and the fraction part to binary separately, then put a dot in between.
  - To get the binary representation of the fraction, divide the fraction first by 0.5 (2^{-1}), take the quotient as the first bit of the binary fraction, then divide the remainder by 0.25 (2^{-2}), take the quotient as the second bit of the binary fraction, then divide the remainder by 0.125 (2^{-3}),…
- Floating numbers. Single precision. 32 bits.

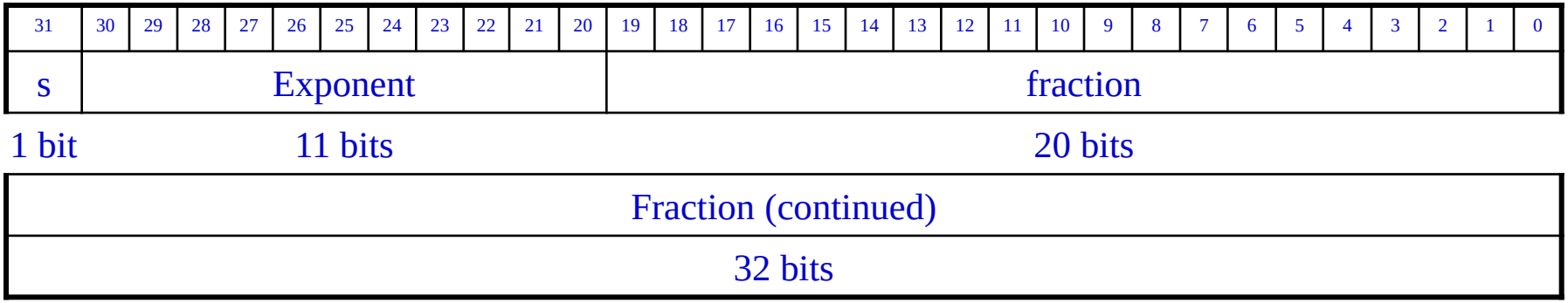
Floating Numbers

- Single precision. 32 bits.



$$(-1)^{S} \times (1 + 0.\text{Fraction}) \times 2^{(\text{Exponent} - 127)}$$

- Double precision. 64 bits. Bias is 1023.



Special Cases Considered

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	nonzero	0	nonzero	±denormalized number
1-254	anything	1-2046	anything	±floating-point number
255	0	2047	0	± infinity
255	nonzero	2047	nonzero	NaN (Not a number)

Problem

The single precision floating number representation of -22.75_{ten} is

(a) 1 10000111 011 0100 0000 0000 0000 0000

(b) 1 10000011 011 0110 0000 0000 0000 0000

(c) 0 10000100 011 1110 0000 0000 0000 0000

(d) None of the above.

MIPS

- MIPS registers

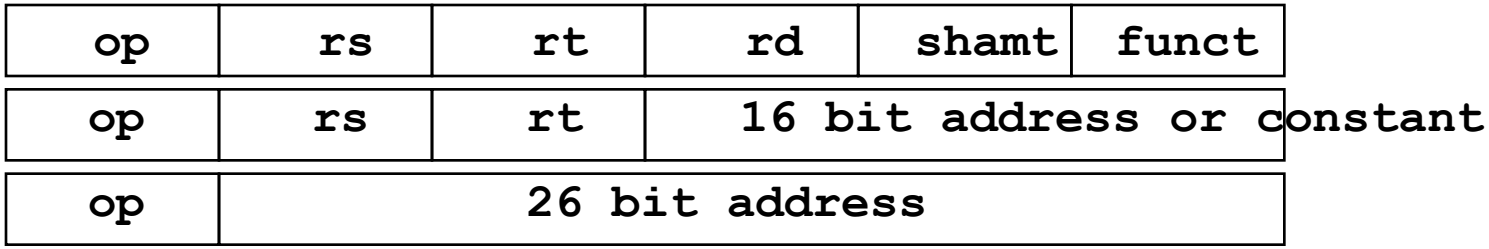
Register Number	Mnemonic Name	Conventional Use	Register Number	Mnemonic Name	Conventional Use
$0	$zero	Permanently 0	$24, $25	$t8, $t9	Temporary
$1	$at	Assembler Temporary (reserved)	$26, $27	$k0, $k1	Kernel (reserved for OS)
$2, $3	$v0, $v1	Value returned by a subroutine	$28	$gp	Global Pointer
$4–$7	$a0–$a3	Arguments to a subroutine	$29	$sp	Stack Pointer
$8–$15	$t0–$t7	Temporary (not preserved across a function call)	$30	$fp	Frame Pointer
$16–$23	$s0–$s7	Saved registers (preserved across a function call)	$31	$ra	Return Address

MIPS

- MIPS instructions (not complete)
  - R-type: add, sub, and, or, sll,…
    - add $t0, $t1, $t2 # add $1, $t2, put the result in $t0
  - Memory: lw, sw, lb, sb.
    - lw $t0, 4($t1) # read the data at the address of
      # $t1+4, put it in $t0
  - Branch: beq, bne, …
    - beq $t0, $t1, SOMEWHERE # if $t0 is equal to $t1, the
      # next instruction to be
      # executed is at the address
      # specified by SOEMWHERE
      # (PC+4+offset)
  - Jump: j, jal, jr
    - j SOMEWHERE # the next instruction should be at the address
      # specified by SOMEWHERE (The upper 4 bits from PC+4, the
      # lower 26 bits from the instruction, the last 2 bits 0)
  - Immediate type:
    - addi $t0, $t0, 4 # add $t0 by 4 and put it in $t0

MIPS Instruction Encoding

- Each MIPS instruction is exactly 32 bits
  - R-type (register type)
  - I-type (immediate type)
  - J-type (jump type)



MIPS Coding

- If else. Assume f to h are in $s0 to $s4.

```
if (i==j) f = g + h; else f = g - h;
```

```
     bne $s3,$s4,Else;      #go to Else if i <> j
     add $s0, $s1, $s2      #f = g + h
     j    Exit;             #go to the end of the if-then-else block
Else:
     sub $s0, $s1, $s2      #f = g -h
Exit:
```

while loop

- Assume that i and k correspond to registers $s3 and $s5 and base array save is in $s6

```
while (save[i] == k)
  i += 1;
```

```
Loop: if (save[i] != k) goto Exit;
      i = i + 1;
      goto Loop;
Exit:
```

```
Loop: sll  $t1, $s3, 2     # Temp reg $t1 = 4 * i
      add  $t1, $t1, $s6   # $t1 = address of save [i]
      lw   $t0, 0($t1)     # Temp reg $t0 = save[i]
      bne  $t0, $s5, Exit  # go to Exit if save[i] ≠ k
      addi $s3, $s3, 1     # i = i + 1
      j    Loop            # go to Loop
Exit:
```

Problem

If $t0 is holding 0, $t1 is holding 1, what will be the value stored in $t2 after the following instructions?

```
                srl $t1, $t1, 1
                bne $t0, $t1, L1
                addi $t2, $t0, 1
L1:             addi $t2, $t0, 2
```

(a) 1.

(b) 2.

(c) 3.

(d) None of the above.

Consider the following C code

```
if (a > b)
        a = A[b];
else
        A[a] = b;
```

where A is an integer array. Which of the following correctly implements the code above, assume a is in $t0, b is in $t1, and the starting address of A is in $s0? (bgt is “branch if greater than.”)

(a)

```
        bgt $t0, $t1, L1
        add $t2, $t0, $s0
        lw $t1, 0($t2)
L1:     add $t2, $t1, $s0
        sw $t0, 0($t2)
Exit:
```

(b)

```
        bgt $t0, $t1, L1
        add $t2, $t0, $s0
        sw $t1, 0($t2)
        j Exit
L1:     add $t2, $t1, $s0
        lw $t0, 0($t2)
Exit:
```

(c)

```
        bgt $t0, $t1, L1
        sll $t2, $t0, 2
        add $t2, $t2, $s0
        lw $t1, 0($t2)
        j Exit
L1:     sll $t2, $t1, 2
        add $t2, $t2, $s0
        sw $t0, 0($t2)
Exit:
```

(d) None of the above.

MIPS Function

- jal Funct:
  - The next instruction will be at address specified by Funct
  - PC+4 will be stored in $ra
- jr $ra:
  - The next instruction will be the one at address equal to the content in $ra
- Calling a function is more like going to a function and then come back

```
.data
array:
        .word 12, 34, 67, 1, 45, 90, 11, 33, 67, 19
msg_done:
        .asciiz "done!\n"

        .text
        .globl main
main:
        la $s7, array
        li $s0, 0 #i
        li $s1, 0 #res
        li $s6, 9
loop:
        sll $t0, $s0, 2
        add $t0, $t0, $s7
        lw $a0, 0($t0)
        lw $a1, 4($t0)
        jal addfun
        add $s1, $s1, $v0
        addi $s0, $s0, 1
        beq $s0, $s6, done
        j loop

done:
        li $v0,4
        la $a0,msg_done
        syscall
        jr $ra


addfun:
        add $v0, $a0, $a1
        jr $ra
```

Problem

Consider the following code segment. What will the code do?

```
        li $ra, 0x04000000
        jal f1
        other instructions…
f1:     addi $ra, -8
        jr $ra
```

(a) It will enter a loop and can never come out.

(b) It will jump to the instruction located at address 0x04000000.

(c) It will call f1 once, then continue to execute other instructions following the jal f1 instruction.

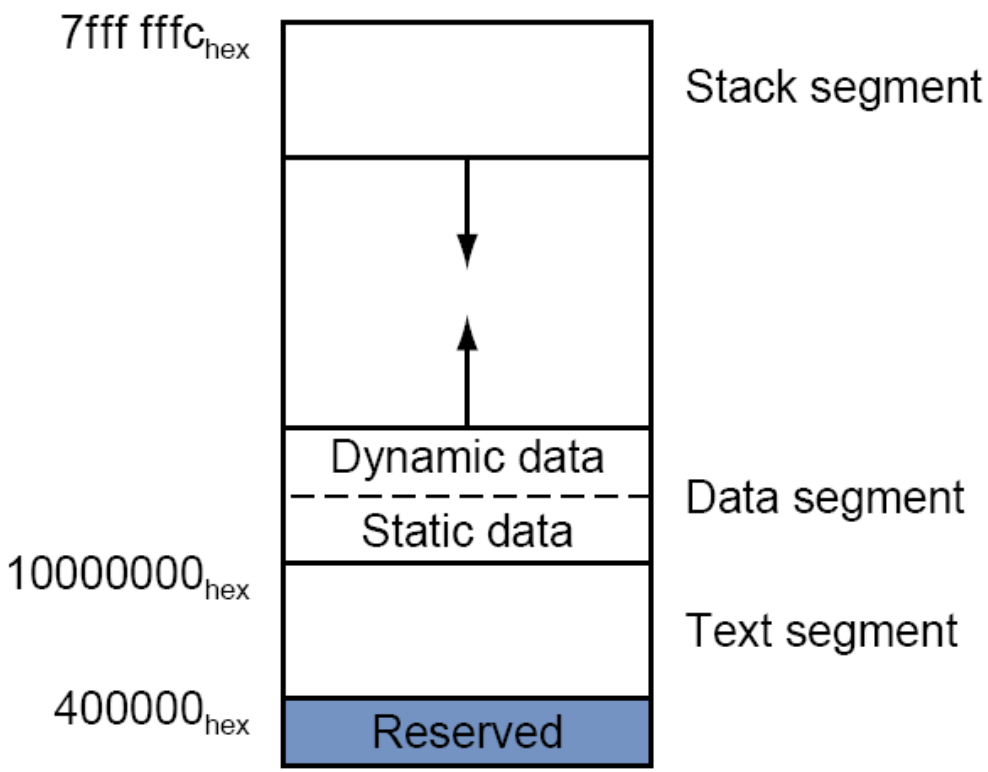
(d) None of the above.

MIPS Calling Conventions

- MIPS assembly follows the following convention in using registers
  - $a0 - $a3: four argument registers in which to pass parameters
  - $v0 - $v1: two value registers in which to return values
  - $ra: one return address register to return to the point of origin

MIPS stack

- The stack in MIPS is a memory space starting at 0x7ffffffc and growing DOWN.
- The top of the stack is always pointed by the stack pointer, $sp (the address of the first element space in the stack should always be in $sp).
- A function should save the registers it touches on the stack before doing anything, and restore it before returning.



MIPS Calling Conventions - more

- MIPS software divides 18 of the registers into two groups
  - $t0 - $t9: 10 temporary registers that are not preserved by the callee on a procedure call
    - These are caller-saved registers since the caller must save the ones it is using
  - $s0 - $s7: 8 saved registers that must be preserved on a procedure call
    - These are callee-saved registers since the callee must save the ones it uses
- In general,
  - if there is a register that the callee may change, and the caller still needs it after calling the callee, the caller should save it and restore it before using it, such as $ra.
  - If there is a register that the caller is not expected to change after calling the callee, the callee should save it, such as $s0.

Saving $s0

```
        .globl  leaf_example
leaf_example:
        addi    $sp, $sp, -4    #make space on the stack for three items
        sw      $s0, 0($sp)     #save register $s0
        add     $t0, $a0, $a1   #register $t0 contains g + h
        add     $t1, $a2, $a3   #register $t1 contains i + j
        sub     $s0, $t0, $t1   #f = (g + h) - (i + j)
        add     $v0, $s0, $0    #returns f
        lw      $s0, 0($sp)     #restore register $s0
        addi    $sp, $sp, 4     #adjust the stack before the return
        jr      $ra             #return to the calling program
```

MIPS interrupt

- For external interrupt, your code is executing, and if an event happens that must be processed,
  - The address of the instruction that is about to be executed is saved into a special register called EPC
  - PC is set to be 0x80000180, where the interrupt handlers are located
  - Then, after processing this interrupt, call “eret” to set the value of the PC to the value stored in EPC
  - Note the difference between an interrupt and a function call. In a function call, the caller is aware of going to another address. In interrupt, the “main program” is not.

Supporting floating point. Load and Store

- Load or store from a memory location (pseudoinstruction). Just load the 32 bits into the register.
  - l.s $f0, val
  - s.s $f0, val
- Load immediate number (pseudoinstruction)
  - li.s $f0, 0.5

Arithmetic Instructions

- abs.s $f0, $f1
- add.s $f0, $f1, $f2
- sub.s $f0, $f1, $f2
- mul.s $f0, $f1, $f2
- div.s $f0, $f1, $f2
- neg.s $f0, $f1

Data move

- mov.s $f0, $f1
- mfc1 $t0, $f0
- mtc1 $t0, $f0

Convert to integer and from integer

- cvt.s.w $f0, $f0 # convert the 32 bit in $f0 currently representing an integer to float of the same value
- cvt.w.s $f0, $f0 # the reverse

Comparison instructions

- c.lt.s $f0,$f1 #set a flag in coprocessor 1if $f0 < $f1, else clear it. The flag will stay until set or cleared next time
- c.le.s $f0,$f1 #set flag if $f0 <= $f1, else clear it
- bc1t L1 # branch to L1 if the flag is set
- bc1f L1 # branch to L1 if the flag is 0

Read the MIPS code and answer the following questions. What does function f1 do? What is the value returned in $v0 after the function is called?

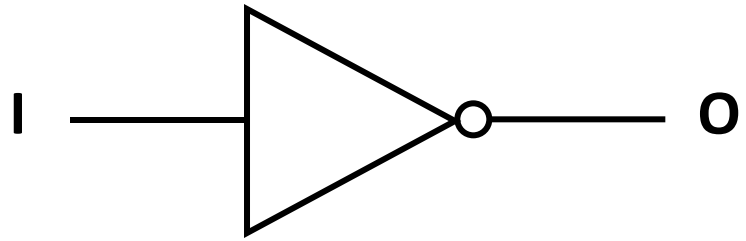
```
        .data
AA:     .word 11,2,33,4,5,6,2,10,7,2,

        .text
        .globl main
main:

        la $a0, AA
        li $a1, 10
        li $a2, 6
        jal fun

fun:
        li $t8, 1000000
        li $t9, -1
        li $t0, 0
funL0:
        sll $t1, $t0, 2
        add $t1, $t1, $a0
        lw $t1, 0($t1)
        bgt $t1, $a2, funL1
        sub $t2, $a2, $t1
        j funL2
funL1:
        sub $t2, $t1, $a2
funL2:
        bgt $t2, $t8, funL3
        ori $t8, $t2, 0
        ori $t9, $t0, 0
funL3:
        addi $t0, $t0, 1
        bne $t0, $a1, funL0

        ori $v0, $t9, 0
        jr $ra
```

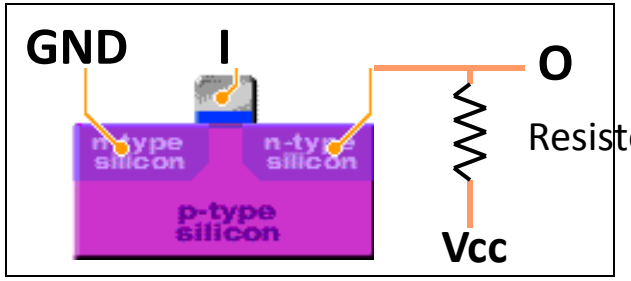
Digital Logic, gates

- Basic Gate: Inverter

Truth Table

I	O
0	**1**
1	**0**





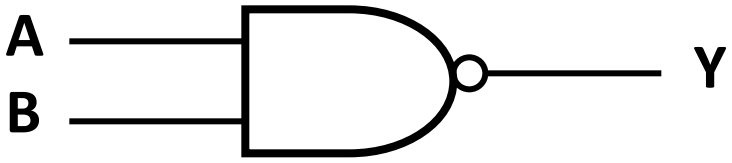
Resister (limits conductivity)

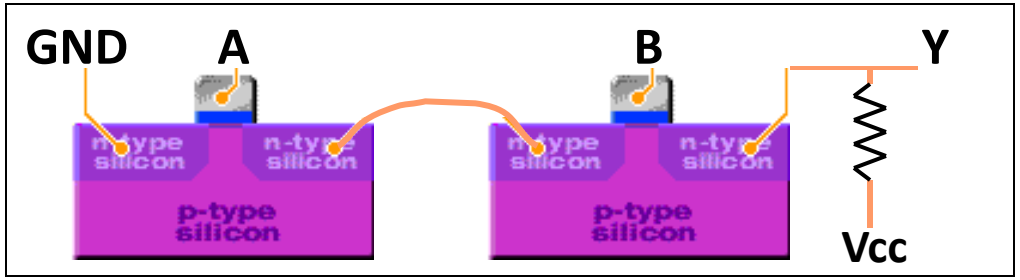
Abstractions in CS (gates)

- Basic Gate: NAND (Negated AND)

Truth Table

A	B	Y
0	**0**	**1**
0	**1**	**1**
1	**0**	**1**
1	**1**	**0**



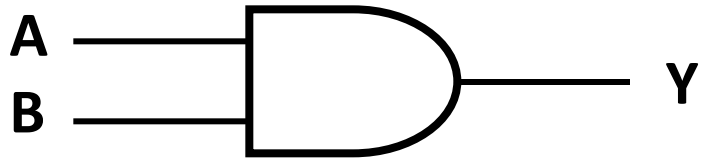


Abstractions in CS (gates)

- Basic Gate: AND

Truth Table

A	B	Y
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

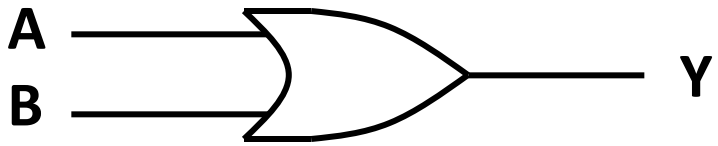


Abstractions in CS (gates)

- Other Basic Gates: OR gate

Truth Table

A	B	Y
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

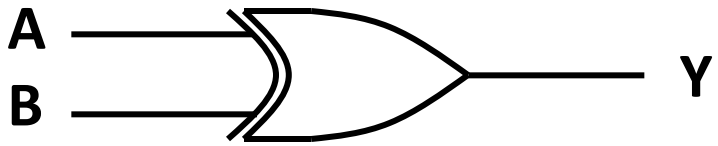


Abstractions in CS (gates)

- Other Basic Gates: XOR gate

Truth Table

A	B	Y
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**0**



Design flow

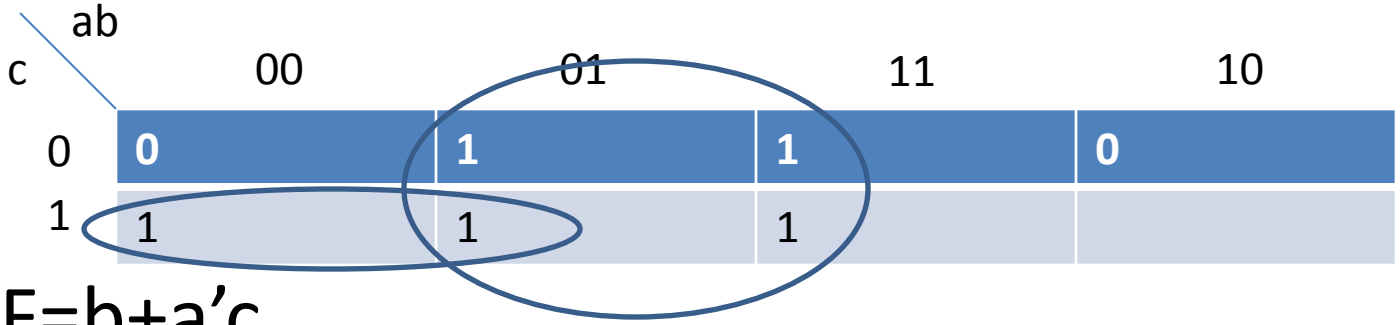
- Given any function, first get the truth table.
- Based on the truth table, use the Karnaugh Map to simplify the circuit.

Karnaugh Map

- Draw the map. Remember to make sure that the adjacent rows/columns differ by only one bit.
- According to the truth table, write 1 in the boxes.
- Draw a circle around a rectangle with all 1s. The rectangle must have size 2,4,8,16…Then, reduce the term by writing down the variables that the values does not change. For example, if there is a rectangle with two 1s representing abc’ and abc, you write a term as ab.
- A term may be covered in multiple circles.
- The rectangle can wrap-around!
- Use the minimum number of circles. A single `1’ is also counted as a circle.

K-map

- F=a’bc’+a’bc+abc’+abc+a’b’c



- F=b+a’c

Problem

- A digital circuit has three inputs (X2, X1, X0), and one output O. The output is `1' if the inputs interpreted as an unsigned integer is an even number less than 5. Which of the following implements the function?

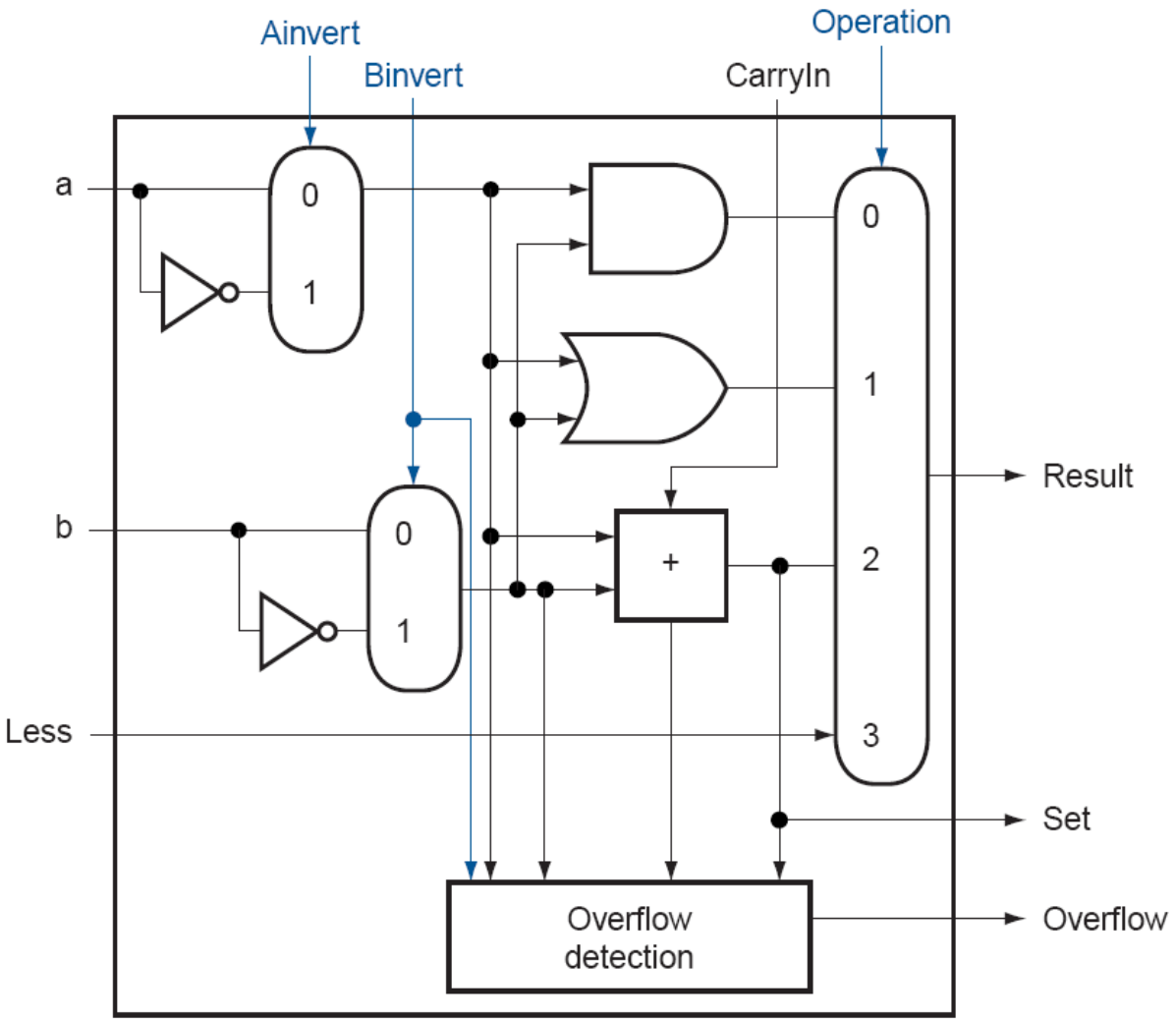
a. O = (~X2&~X0) | (X2&~X1&X0)

b. O = (~X2&~X0) | (~X1&~X0)

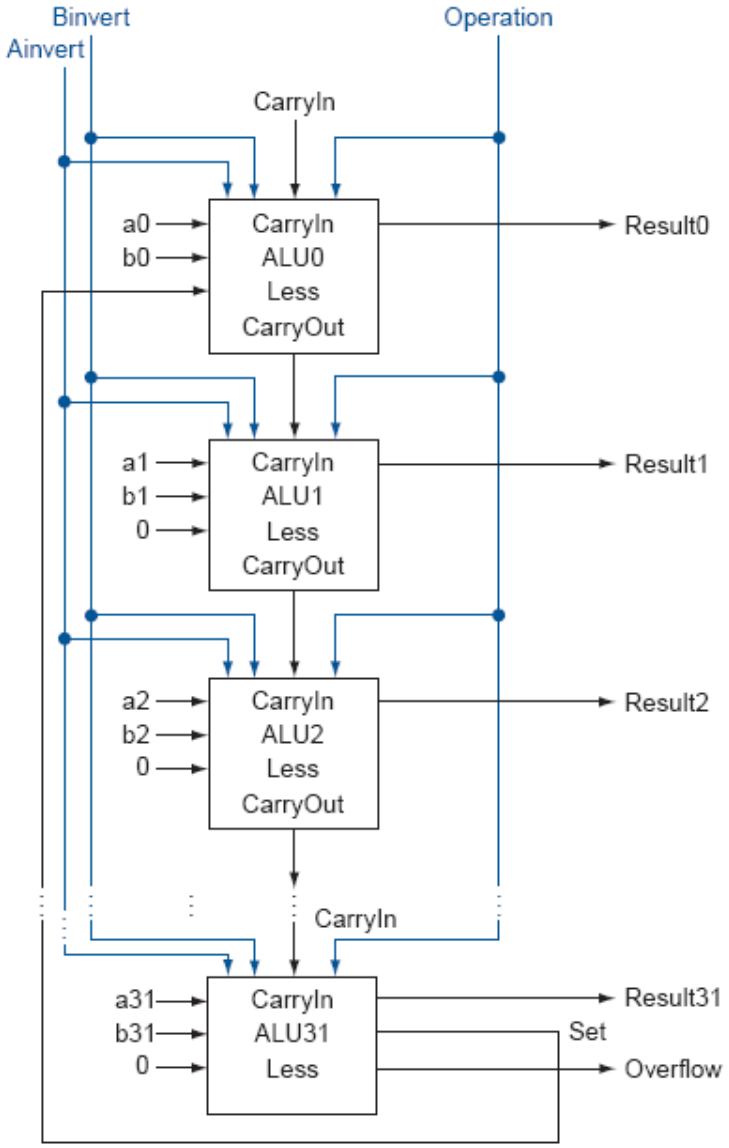
c. O = (~X2&~X1&~X0) | (~X2&X1&~X0) | (X2&~X1&X0)

d. None of the above.

MIPS ALU unit



32-bit ALU that Supports Set Less Than



Problems

The MIPS ALU is controlled by 4 bits as shown below. Consider the MIPS processor supporting the R-type, lw, sw, and beq instructions. What should the control signal be in case of the lw and beq instructions?

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

(a) lw: 0010, beq: 0010

(b) lw: 0001, beq: 0000

(c) lw: 0010, beq: 0110

(d) None of the above.

Verilog Data Types

- A wire specifies a combinational signal.
- A reg (register) holds a value, which can vary with time. A reg need not necessarily correspond to an actual register in an implementation, although it often will.

Constants

- Constants is represented by prefixing the value with a decimal number specifying its size in bits.
- For example:
  - 4’b0100 specifies a 4-bit binary constant with the value 4, as does 4’d4.

Values

- The possible values for a register or wire in Verilog are
  - 0 or 1, representing logical false or true
  - x, representing unknown, the initial value given to all registers and to any wire not connected to something
  - z, representing the high-impedance state for tristate gates

Operators

- Verilog provides the full set of unary and binary operators from C, including
  - the arithmetic operators (+, –, *, /),
  - the logical operators (&, |, ~),
  - the comparison operators (==, !=, >, <, <=, >=),
  - the shift operators (<<, >>)
  - Conditional operator (?, which is used in the form condition ? expr1 :expr2 and returns expr1 if the condition is true and expr2 if it is false).

Structure of a Verilog Program

- A Verilog program is structured as a set of modules, which may represent anything from a collection of logic gates to a complete system.
- A module specifies its input and output ports, which describe the incoming and outgoing connections of a module.
- A module may also declare additional variables.
- The body of a module consists of
  - initial constructs, which can initialize reg variables
  - continuous assignments, which define only combinational logic
  - always constructs, which can define either sequential or combinational logic
  - instances of other modules, which are used to implement the module being defined

The half-adder. Example of continuous assignments

```
module half_adder (A,B,Sum,Carry);
input A,B;
output Sum, Carry;
assign Sum = A ^ B;
assign Carry = A & B;
endmodule
```

- assign: continuous assignments. Any change in the input is reflected immediately in the output.
- Wires may be assigned values only with continuous assignments.

One-bit Full Adder

```
module full_adder (A,B,Cin,Sum, Cout);
        input A,B,Cin;
        output Sum, Cout;

        assign Sum = (A & B & Cin) | (~A & ~B & Cin) | (~A & B & ~Cin) | (A & ~B & ~Cin);
        assign Cout = (A & Cin) | (A & B) | (B & Cin);

endmodule
```

Four-bit Adder

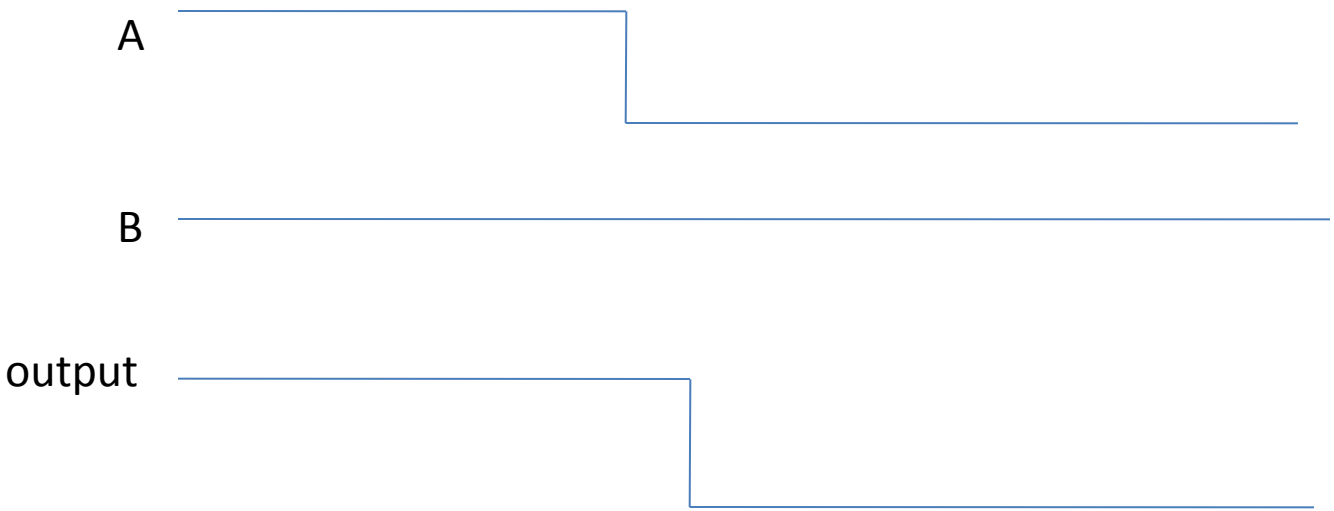
```
module four_bit_adder (A,B,Cin,Sum, Cout);
        input [3:0] A;
        input [3:0] B;
        input Cin;
        output [3:0] Sum;
        output Cout;

        wire C0, C1, C2;

        full_adder FA1(A[0], B[0], Cin, Sum[0], C0);
        full_adder FA2(A[1], B[1], C0, Sum[1], C1);
        full_adder FA3(A[2], B[2], C1, Sum[2], C2);
        full_adder FA4(A[3], B[3], C2, Sum[3], Cout);

endmodule
```

D-flip-flop

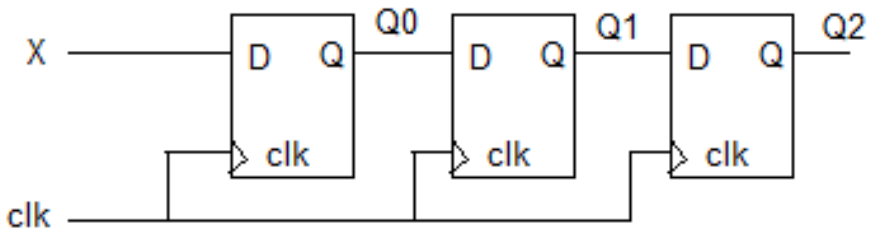
```
module Dff1 (D, clk, Q, Qbar);
        input D, clk;
        output reg Q, Qbar;

        initial begin
                Q = 0;
                Qbar = 1;
        end

        always @(posedge clk) begin
                #1
                Q = D;
                #1
                Qbar = ~Q;
        end
endmodule
```

Delay

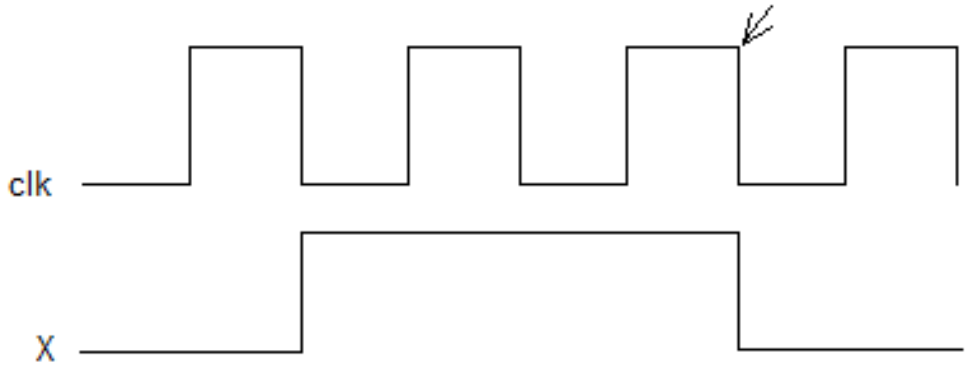
- Real circuits have delays caused by charging and discharging.
- So, once the input to a gate changes, the output will change after a delay, usually in the order of nano seconds. An and gate:



Consider the following circuit.



And suppose the input X is



At the time pointed by the arrow, what should Q2Q1Q0 (notice the order) be?

(a) 011.

(b) 110.

(c) 000.

(d) None of the above.

Sequential Circuits

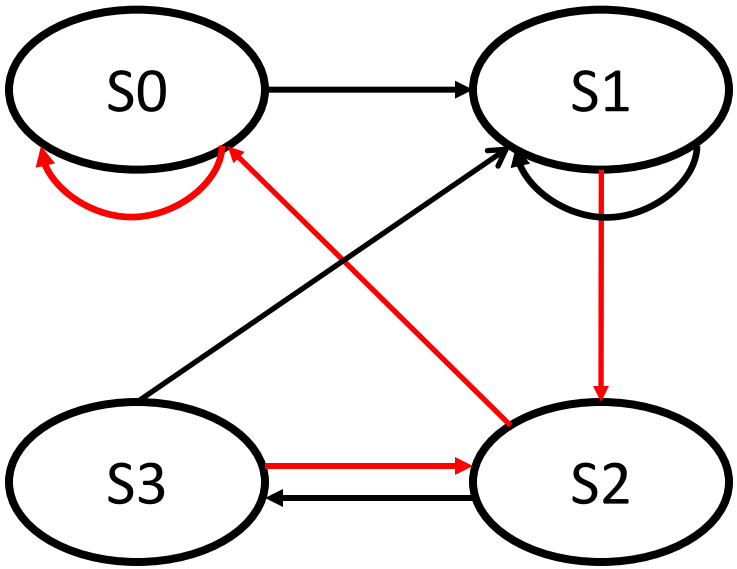
- A three-bit counter.
- First, get the next state table. Then, generate D2, D1, D0.

Q2	Q1	Q0	D2	D1	D0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

FSM example – A sequence detector

- One input X, and one output O.
- X may change every clock cycle. The change happens at the falling edge.
- The circuit samples the input at every rising edge of the clock. If the input is 1, consider as read a 1, else read a 0.
- O is 1 (for one clock cycle, from positive edge to positive edge) if the last three bits read are 101.

4 states



X = 1

X = 0

- S0: got nothing. The initial state.
- S1: got 1.
- S2: got 10.
- S3: got 101.

Assign states

- S0 = 00
- S1 = 01
- S2 = 10
- S3 = 11

Next State Function

Q1	Q0	X	D1	D0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	0
1	1	1	0	1

D1 = (Q0&~X)|(Q1&~Q0&X)

D0 = X

The output function

- Clearly, O = Q1&Q0.

Please read the following Verilog module:

```
module statem (clk, O);
       input clk;
       output [1:0] O;

       wire D1, D0, Q1, Q0, Q1bar, Q0bar;

       assign D0 = Q1 & Q0;
       Dff1 C0 (D0, clk, Q0, Q0bar);

       assign D1 = ~Q0;
       Dff1 C1 (D1, clk, Q1, Q1bar);

       assign O[1] = Q1;
       assign O[0] = Q0;
endmodule
```

Given the D-flip-flops were in the 0 state at the beginning, which of the following statements is true, if Q1Q0 are interpreted as a 2-bit unsigned integer number?

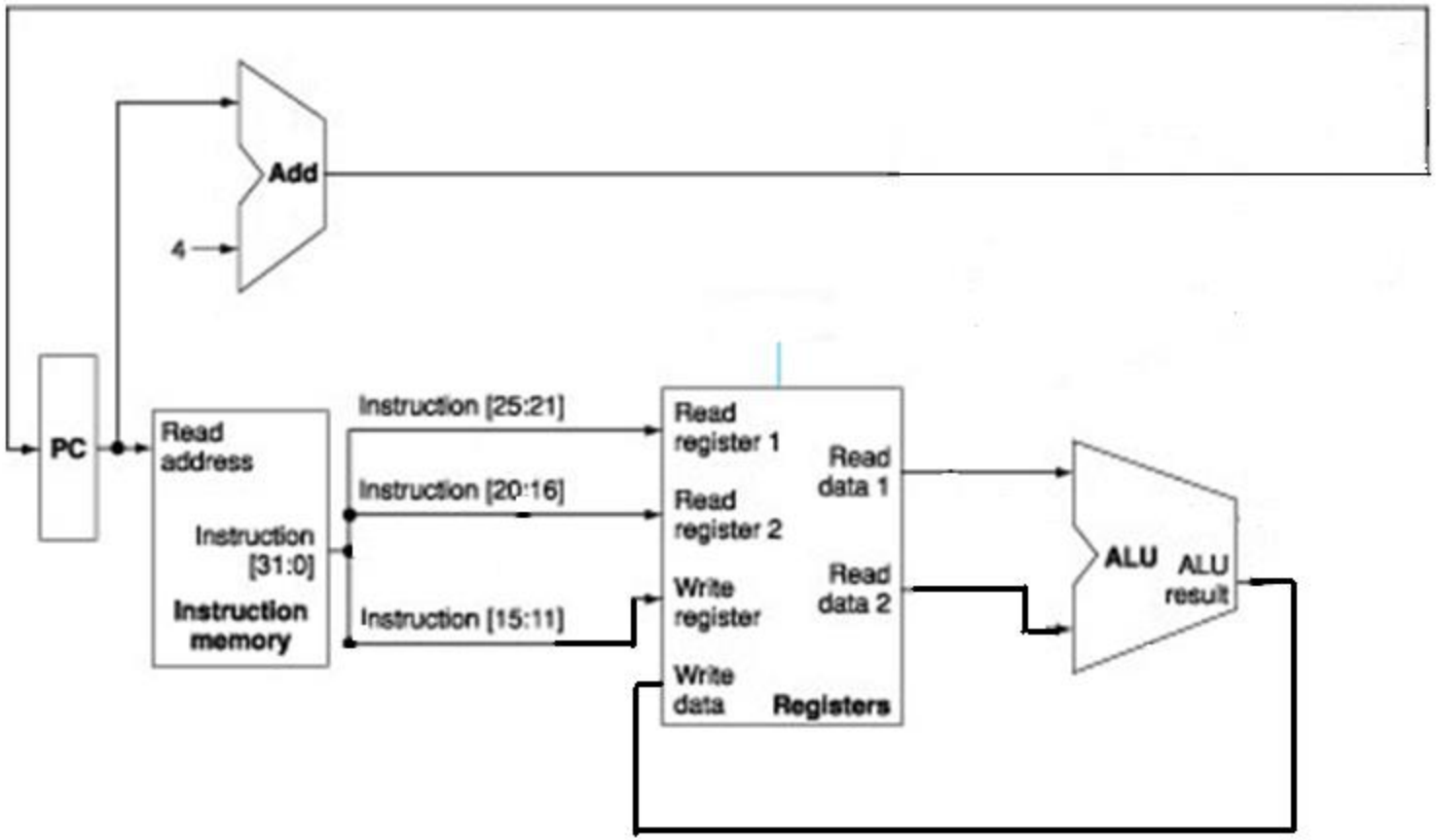
(a) It is a two-bit counter, counting as "01230123..."

(b) It is not a counter, and generates sequence "0202020..."

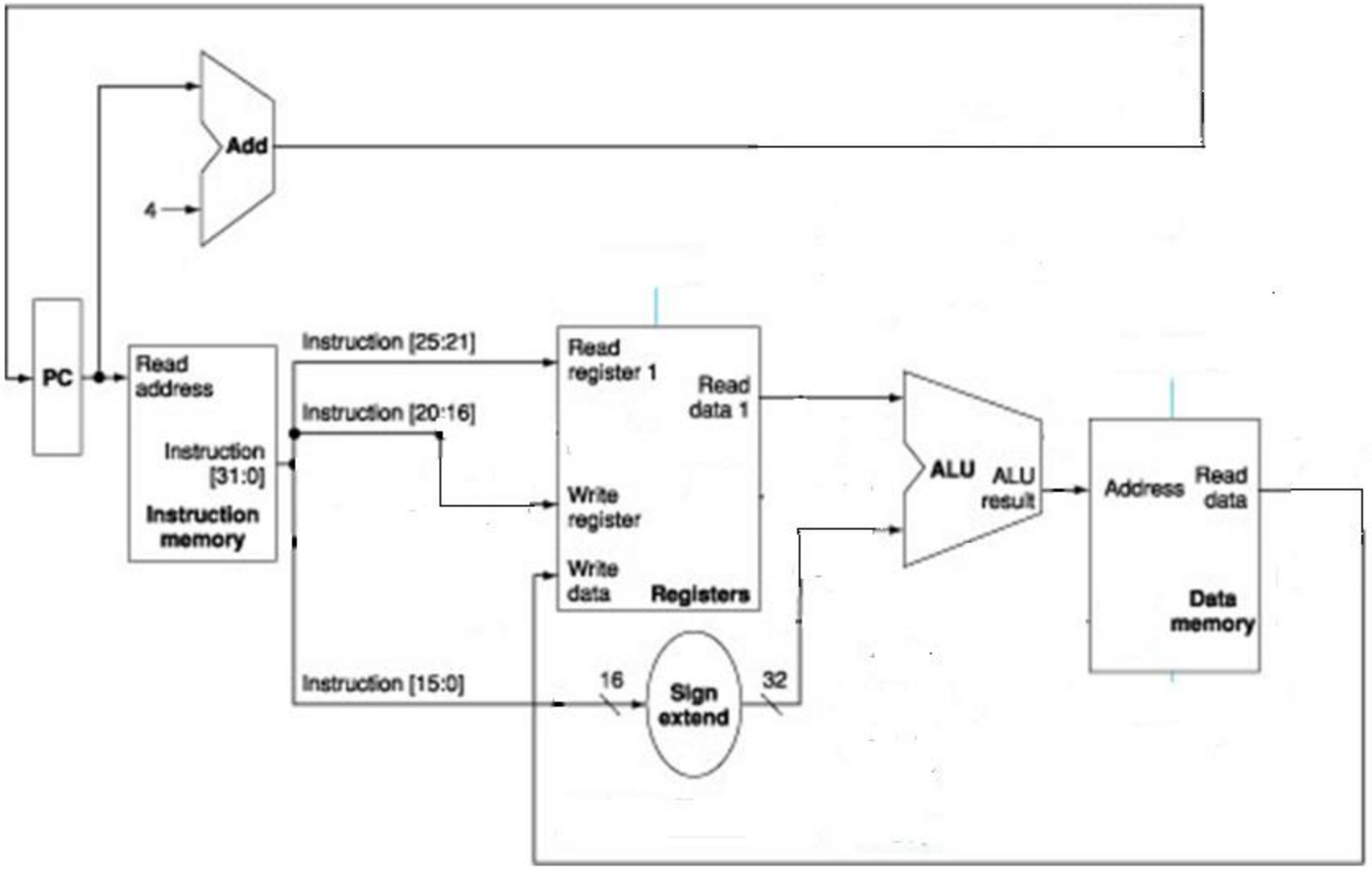
(c) It will stay at state 0.

(d) None of the above.

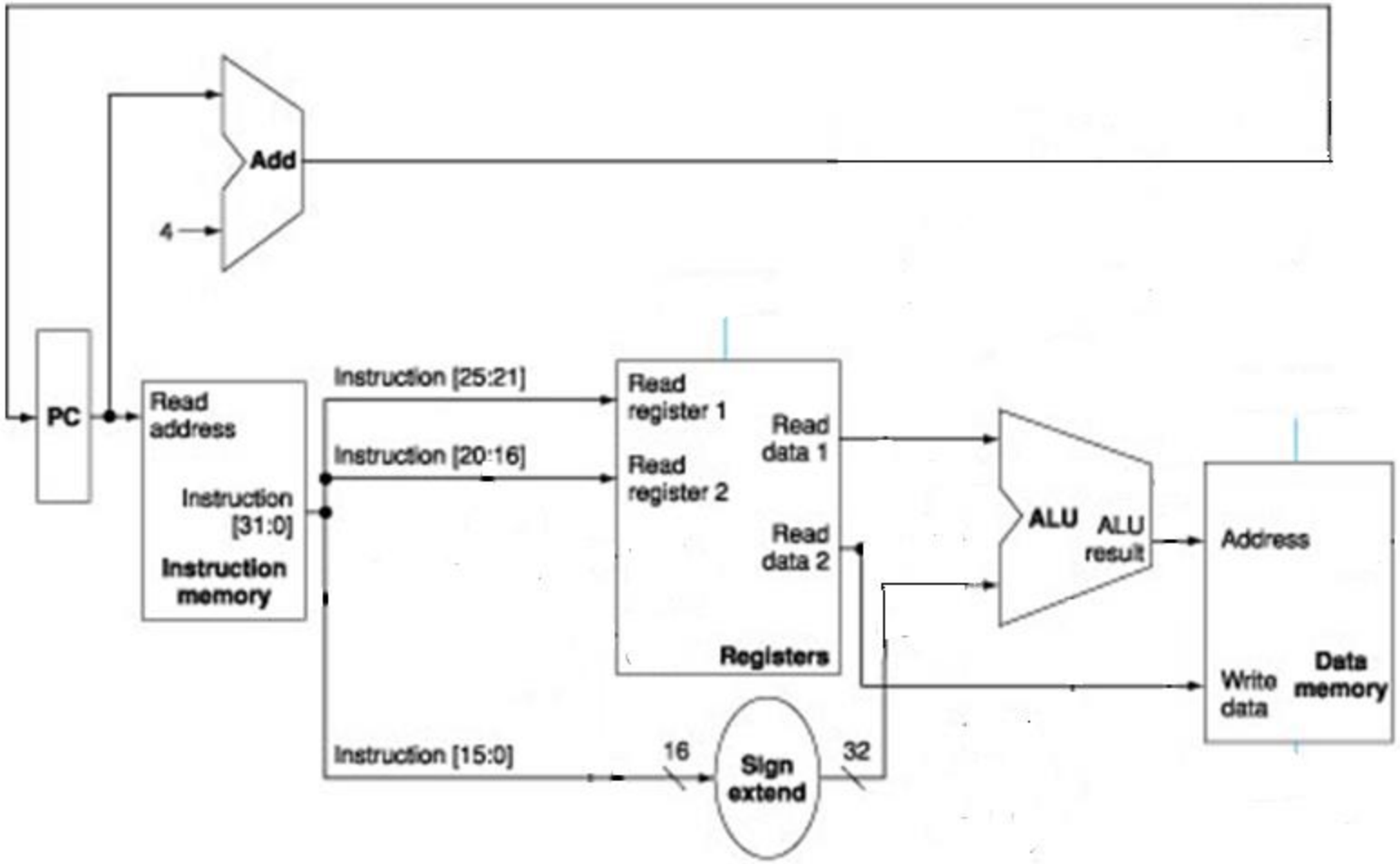
Datapath only for R-type instructions



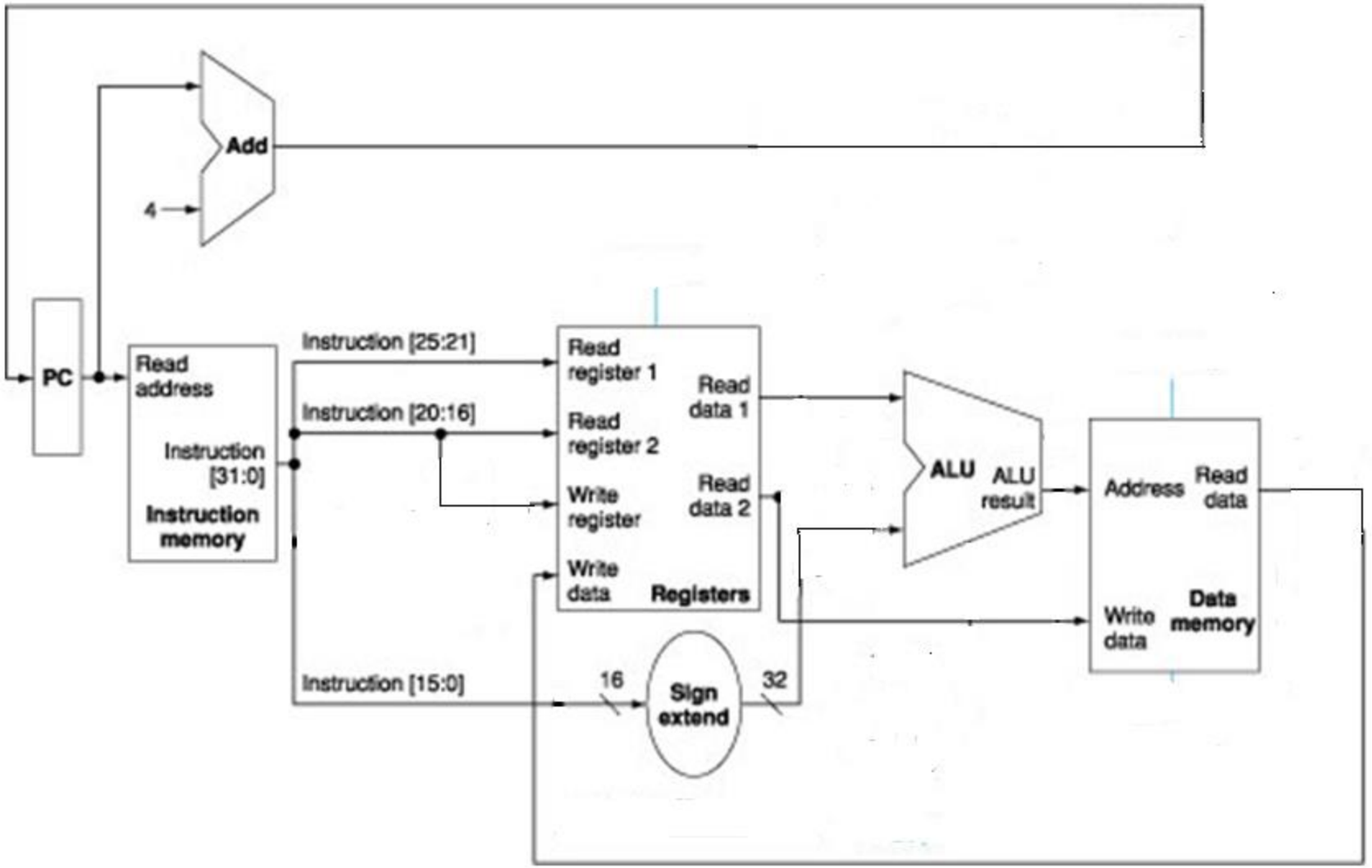
Data path only for lw



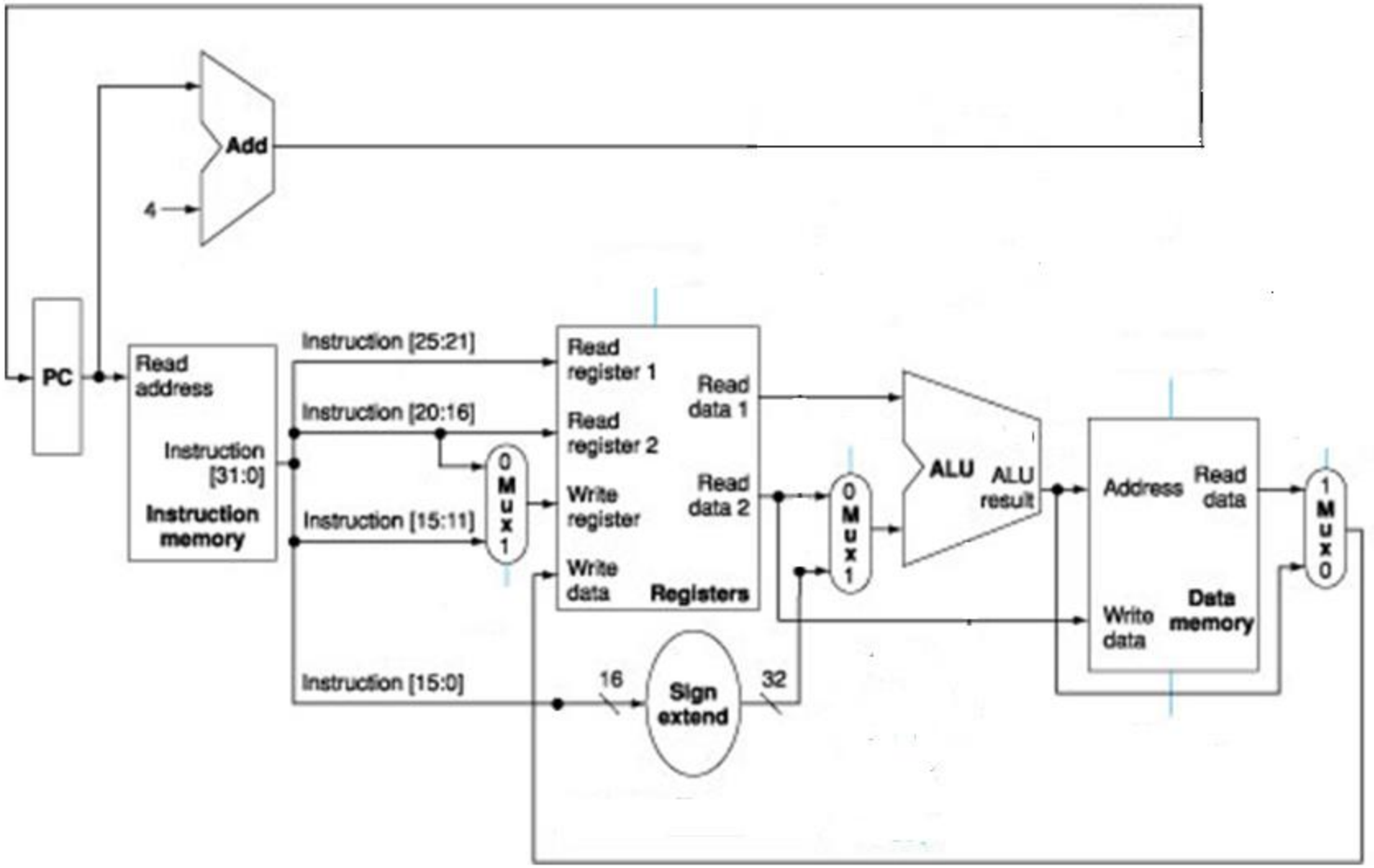
Data path only for sw



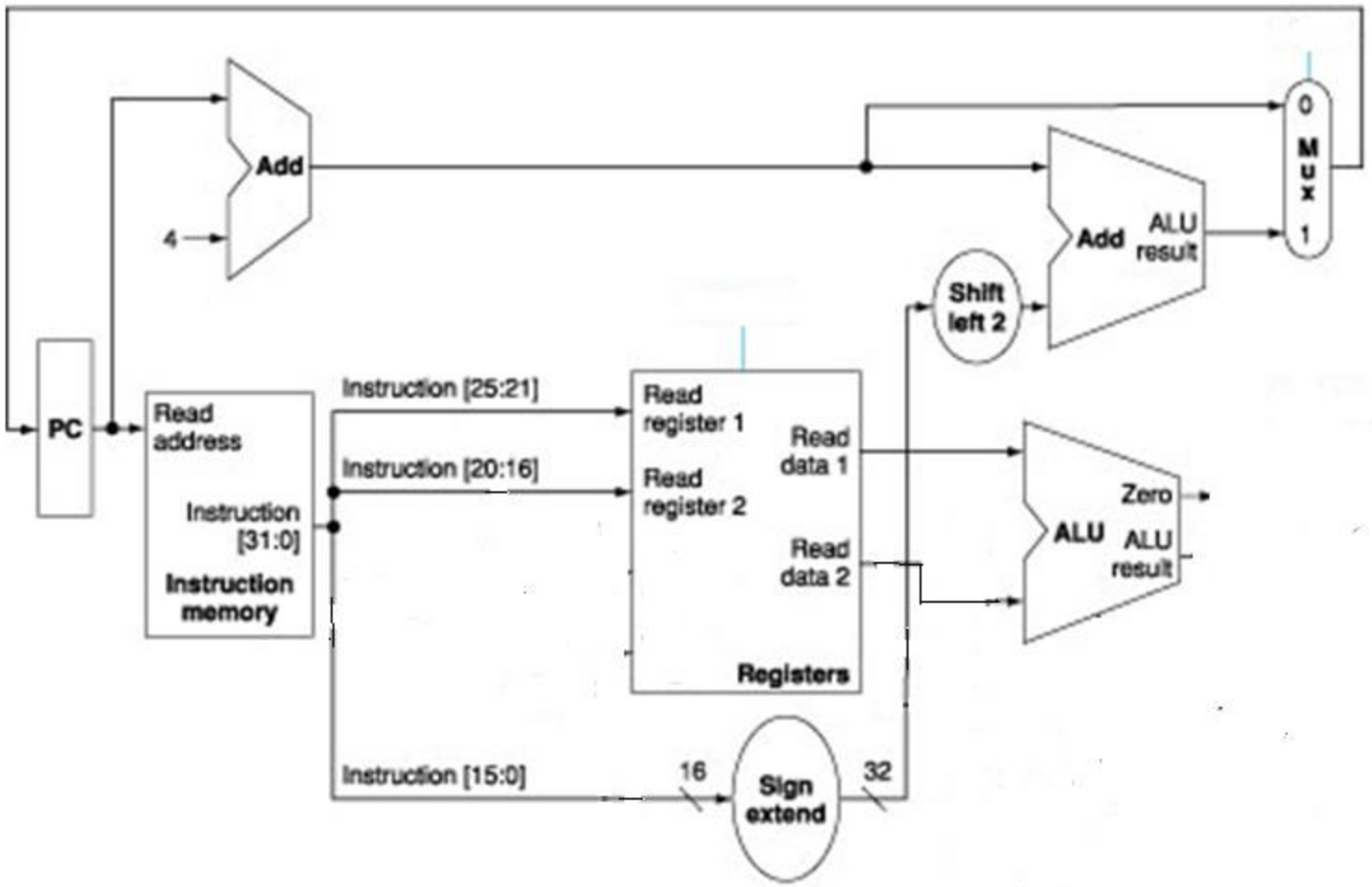
Data path only for lw and sw



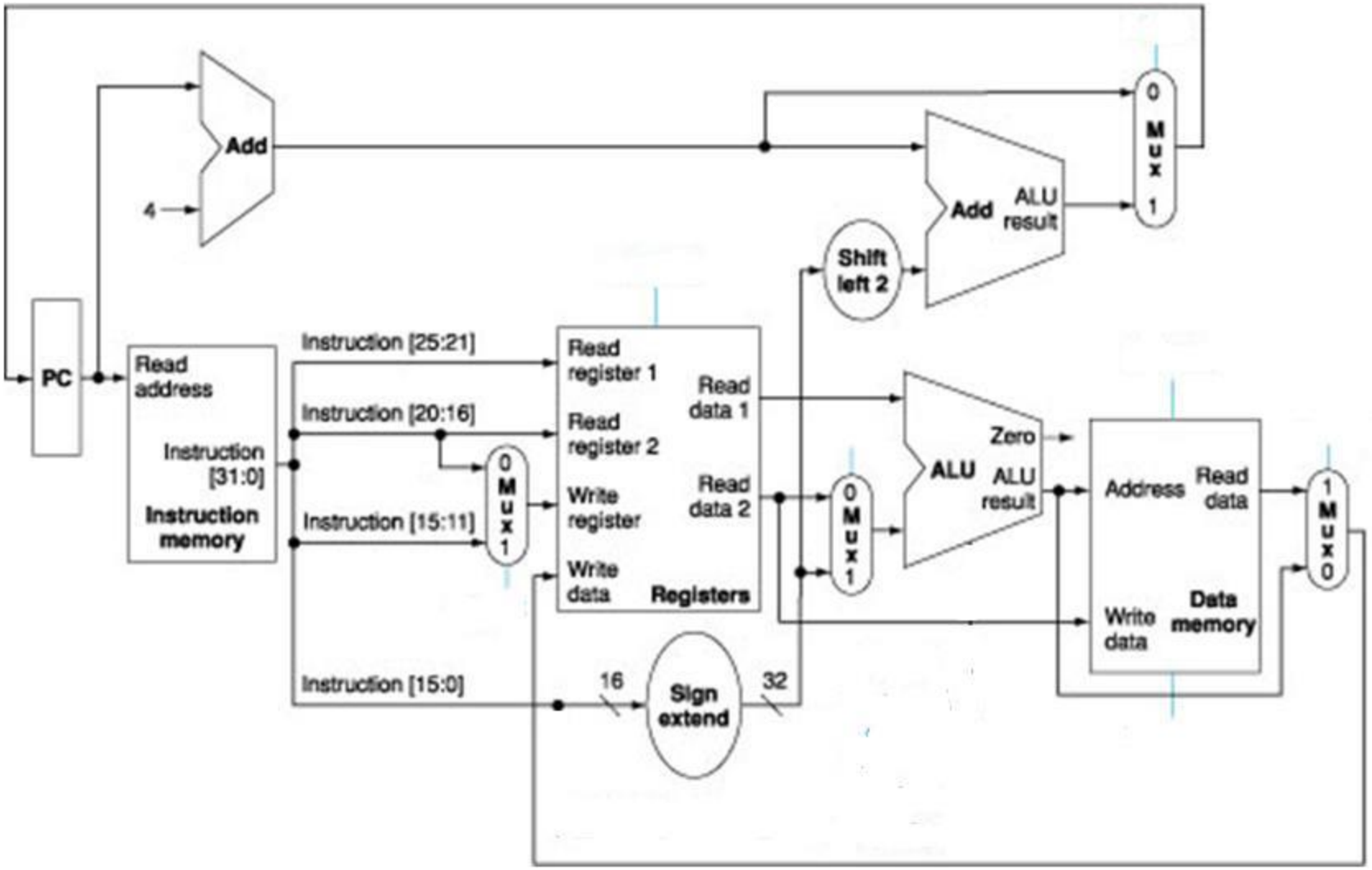
Datapath for Memory and R-type Instructions



Datapath only for beq



Datapath for R-type, memory, and branch operations



Problem

In addition to "beq," MIPS also has the "bne" instruction. Suppose we already have a MIPS processor supporting the R-type, lw, sw, and beq instructions. To support the bne instruction, which of the following statements is true?

(a) A new data path must be created from one of the outputs of the register file to the PC.

(b) It just needs a few more gates; no addition data path is needed.

(c) It can be totally supported by software based on beq and no additional hardware is needed.

(d) None of the above.

Problem

Which of the following is **not** a reason that the MIPS processor **cannot** support instruction such as ``add $t0, $t1, $t2, $t3,'' that is, adding `$t1, $t2, $t3` and put the result in `$t0`?

(a) Because the ALU cannot add three numbers.

(b) Because the register file can only read two registers.

(c) Because the instruction size (32 bits) is not large enough to specify these many registers for this instruction.

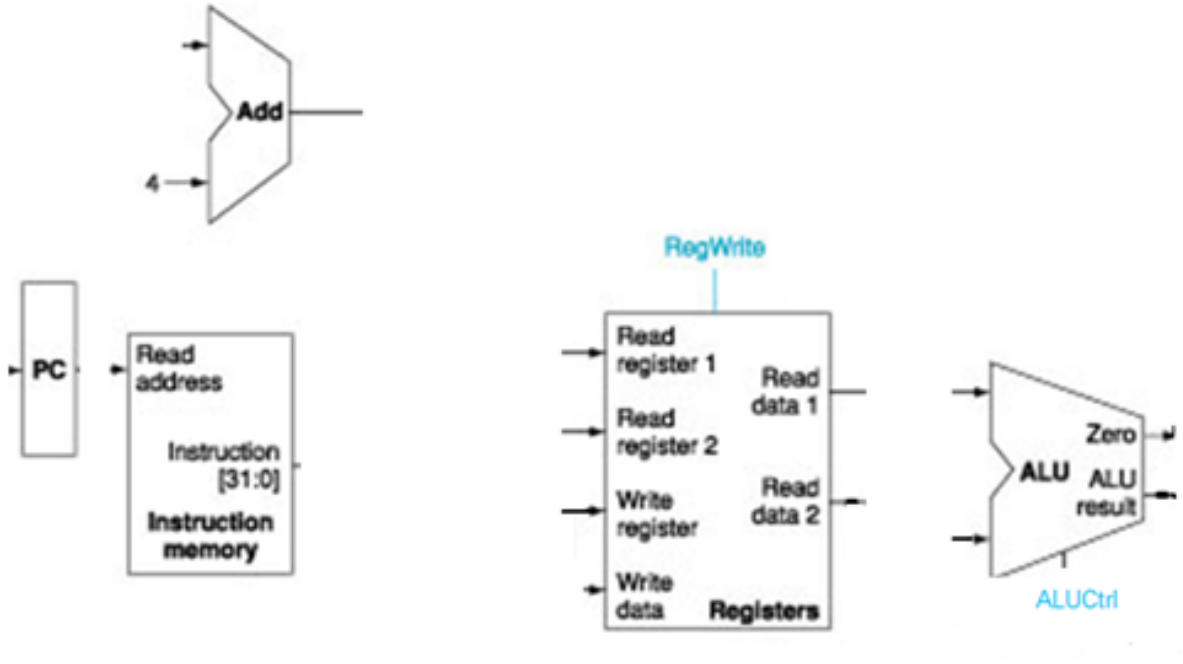
(d) All of the above are correct reasons that the MIPS processor cannot support this instruction.

Problem 3 (20 points) Design a MIPS processor supporting **only** the `R-type` and the `jreq rs, rt, rd` instruction. The `jreq rs, rt, rd` instruction does the following: if `rs==rt`, the address of the next instruction should be the content of `rs`, and the content of `rd` should be set to be the content of `rt`; otherwise, nothing happens, go to the next instruction. For this problem, assume that the leading bit of the `opcode` of all `R-type` instructions is 0 and the leading bit of the `opcode` of `jreq` is 1.

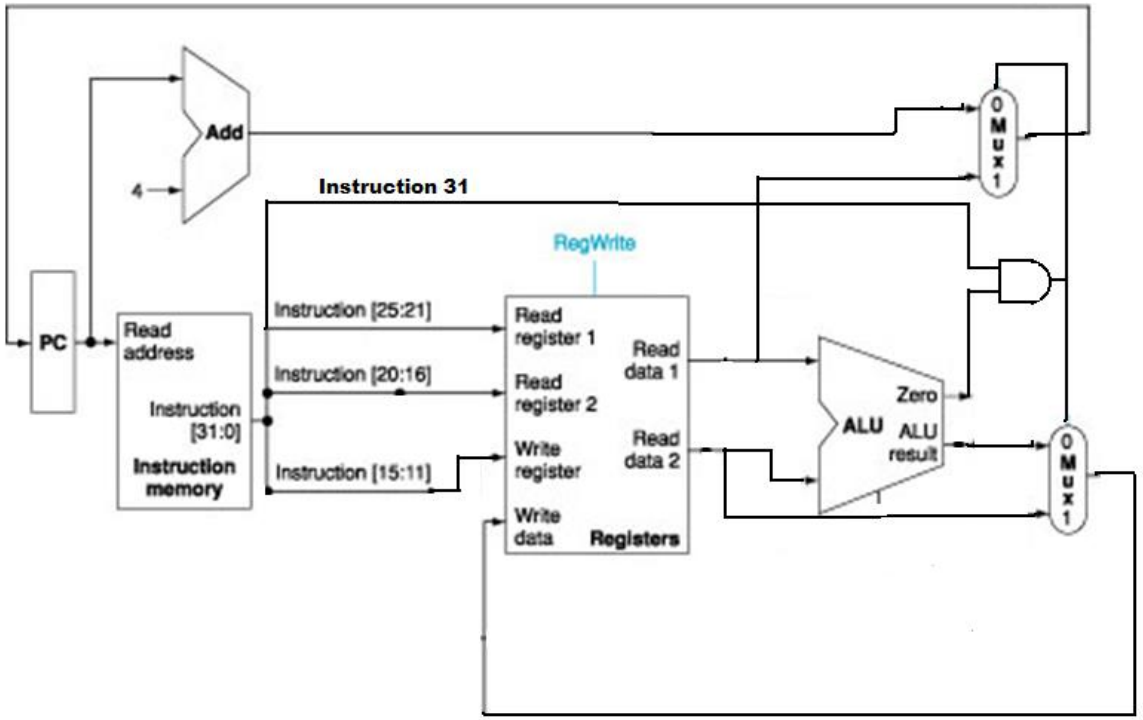
(a) (12 points) Show the data path of this processor, add 2-1 MUXes when necessary. Besides a group of wires, please show clearly the indices of the bits.

(b) (8 points) The control signals include `RegWrite`, `ALUCtrl`, and the signals to control the added 2-1 MUXes. Show how to generate all control signals except `ALUCtrl` by drawing down the circuit.

Note: This instruction is my invention and I admit it seems to be useless in practice. But let's see if you can implement it.



Answer



Also, RegWrite = ~Instruct[31] | (Instruct[31] & zero)