Lecture 13

CMPEN 331

Review questions solved

Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating Point Multiplication Example

- Multiply

$$(0.5 = 1.0000 \times 2^{-1}) \times (-0.4375 = -1.1100 \times 2^{-2})$$

- Step 0: Hidden bits restored in the representation above
- Step 1: Add the exponents (not in bias would be -1 + (-2) = -3
- and in bias would be (-1+127) + (-2+127) – 127 = (-1 -2) + (127+127-127) = -3 + 127 = 124
- Step 2: Multiply the significands
  1.0000 x 1.110 = 1.110000
- Step 3: Normalized the product, checking for exp over/underflow
  1.110000×2^{-3} is already normalized
- Step 4: The product is already rounded, so we're done
- Step 5: Rehide the hidden bit before storing

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

IEEE 754 encoding of floating-point numbers

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)

MIPS Floating Point Instructions

- MIPS has a separate Floating Point Register File (`$f0, $f1, …, $f31`) (whose registers are used in *pairs* for double precision values) with special instructions to load to and store from them

```
lwc1   $f1,54($s2)   #$f1 = Memory[$s2+54]
swc1   $f1,58($s4)   #Memory[$s4+58] = $f1
```

- And supports IEEE 754 single

```
add.s $f2,$f4,$f6   #$f2 = $f4 + $f6
```

and double precision operations

```
add.d $f2,$f4,$f6   #$f2||$f3 =
                              $f4||$f5 + $f6||$f7
```

similarly for `sub.s, sub.d, mul.s, mul.d, div.s, div.d`

MIPS Floating Point Instructions, Con't

- And floating point single precision comparison operations

```
c.x.s $f2,$f4        #if($f2 < $f4) cond=1;
                                    else cond=0
```

where `x` may be `eq, neq, lt, le, gt, ge`

and double precision comparison operations

```
c.x.d $f2,$f4        #$f2||$f3 < $f4||$f5
                                    cond=1; else
                                    cond=0
```

- And floating point branch operations

```
bclt  25             #branch, true if(cond==1)
                                    go to PC+4+25

bclf  25             #branch, false if(cond==0)
                                    go to PC+4+25
```

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - `fahr` in $f12, result in $f0, literals in global memory space

- Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp) # $f16 =5.0
     lwc1  $f18, const9($gp) # $f18 =9.0
     div.s $f16, $f16, $f18  # $f16 =5.0/9.0
     lwc1  $f18, const32($gp) # $f18 =32.0
     sub.s $f18, $f12, $f18  # $f18 =fahr-32.0
     mul.s $f0,  $f16, $f18
     jr    $ra
```

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism

Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i
- Right shift divides by 2^i?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4
    - $1111111111111111111111111111011_2$ >> 2

    = $00111111111111111111111111111110_2$

    = 1073741822
  - A solution would be to have an arithmetic right shift that extends the sign bit instead of shifting in 0s. A 2-bit arithmetic shift right of -5_{ten} produces
    - $11111111111111111111111111111110_2$

    = -2 instead of -1; (close)

Review

- Overflow (floating point) happens when a positive exponent becomes too large to fit in the exponent field
- Underflow (floating point) happens when a negative exponent becomes too large to fit in the exponent field
- IEEE 754 floating point standard

$$(-1)^{sign} \times (1+F) \times 2^{E-bias}$$

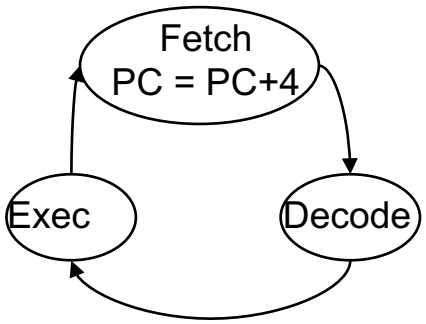
  - Formats for both single and double precision
  - F is stored in normalized format where the msb in F is 1, called the hidden bit
  - To simplify sorting FP numbers, E is represented in excess (biased) notation where the bias is -127 (-1023 for double precision) so the most negative is 00000001 = $2^{1-127} = 2^{-126}$ and the most positive is 11111110 = $2^{254-127} = 2^{+127}$

Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`

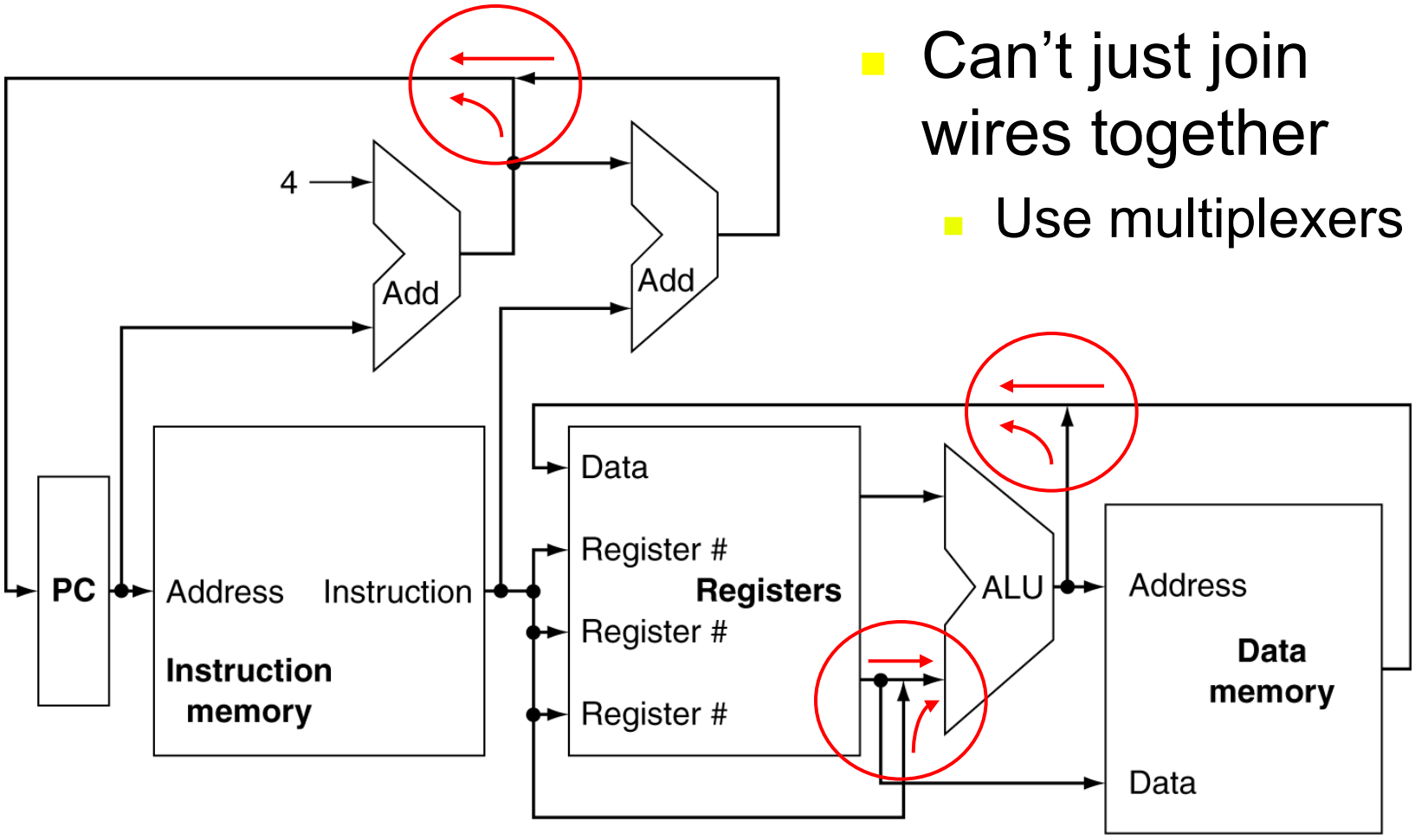
Instruction Execution

- Generic implementation
  - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction



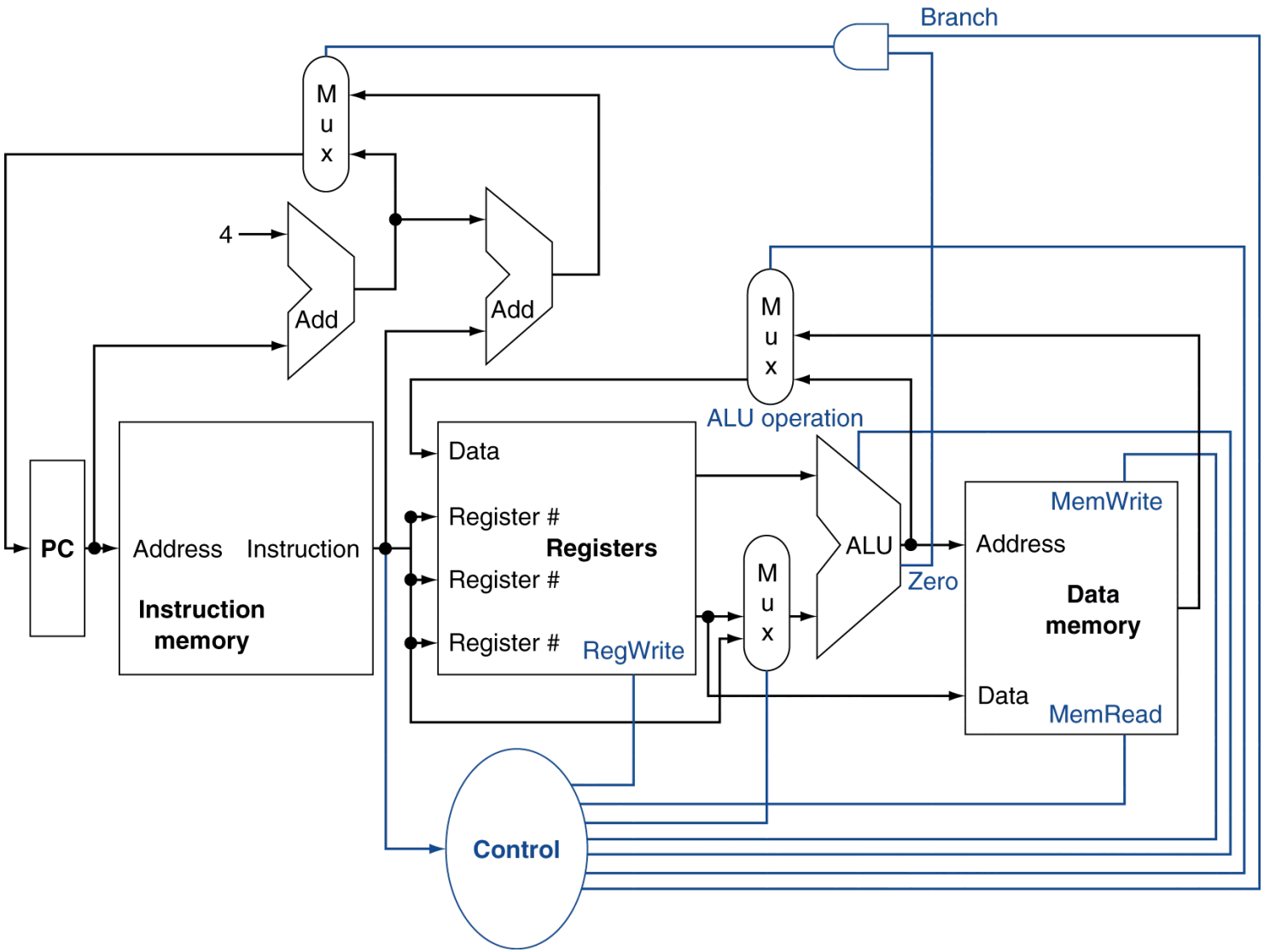
- All instructions (except `j`) use the ALU after reading the registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC ← target address or PC + 4

CPU Overview



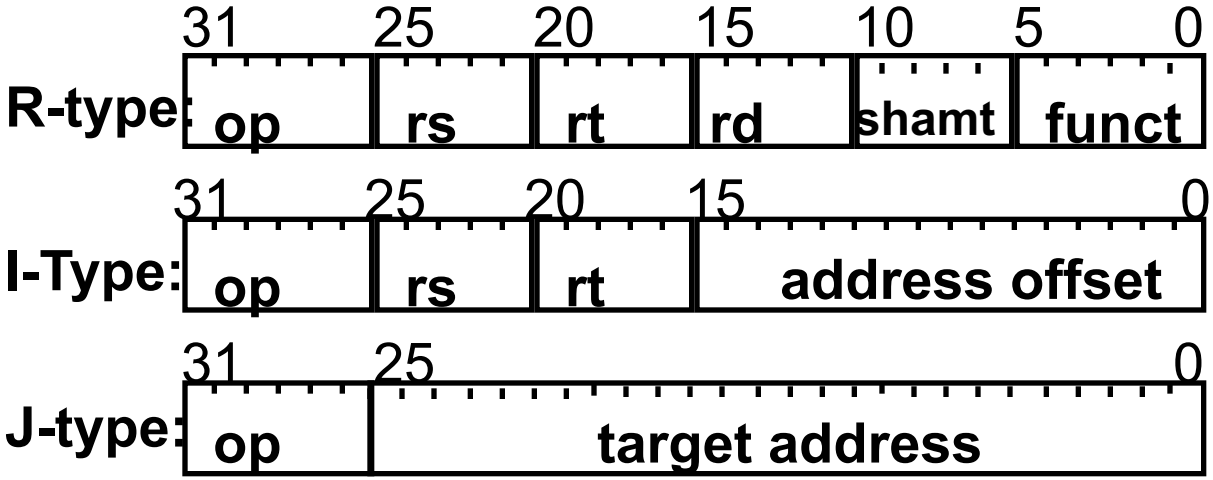
- Can't just join wires together
  - Use multiplexers

Control


Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Adding the Control

- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)



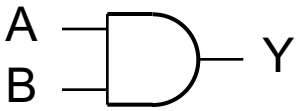
- Observations
  - op field always in bits 31-26
  - address of registers to be read are always specified by the rs field (bits 25-21) and rt field (bits 20-16); for lw and sw rs is the base register
  - address of register to be written is in one of two places – in rt (bits 20-16) for lw; in rd (bits 15-11) for R-type instructions
  - offset for beq, lw, and sw always in bits 15-0

Logic Design Basics

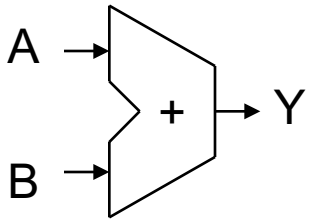
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

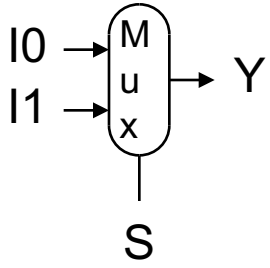
- AND-gate
  - Y = A & B



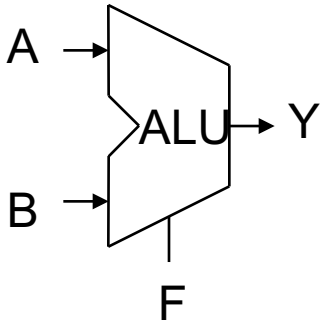
- Adder
  - Y = A + B



- Multiplexer
  - Y = S ? I1 : I0

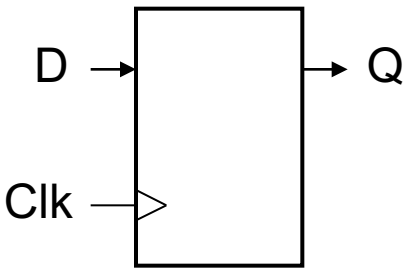


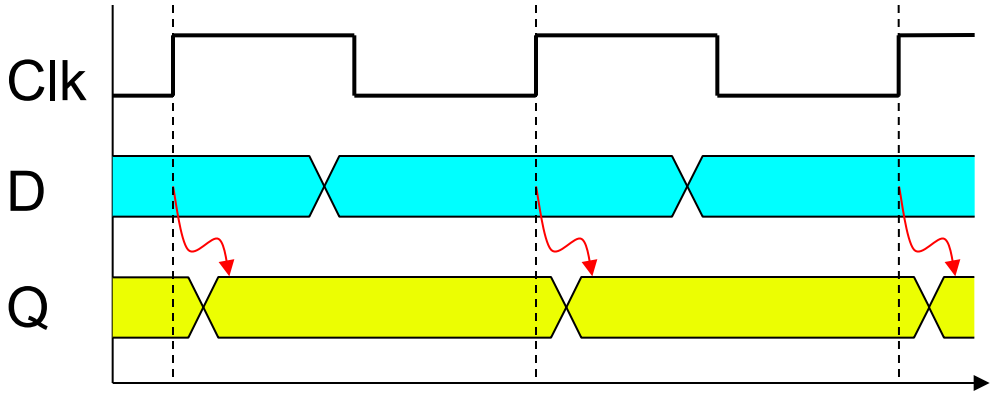
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

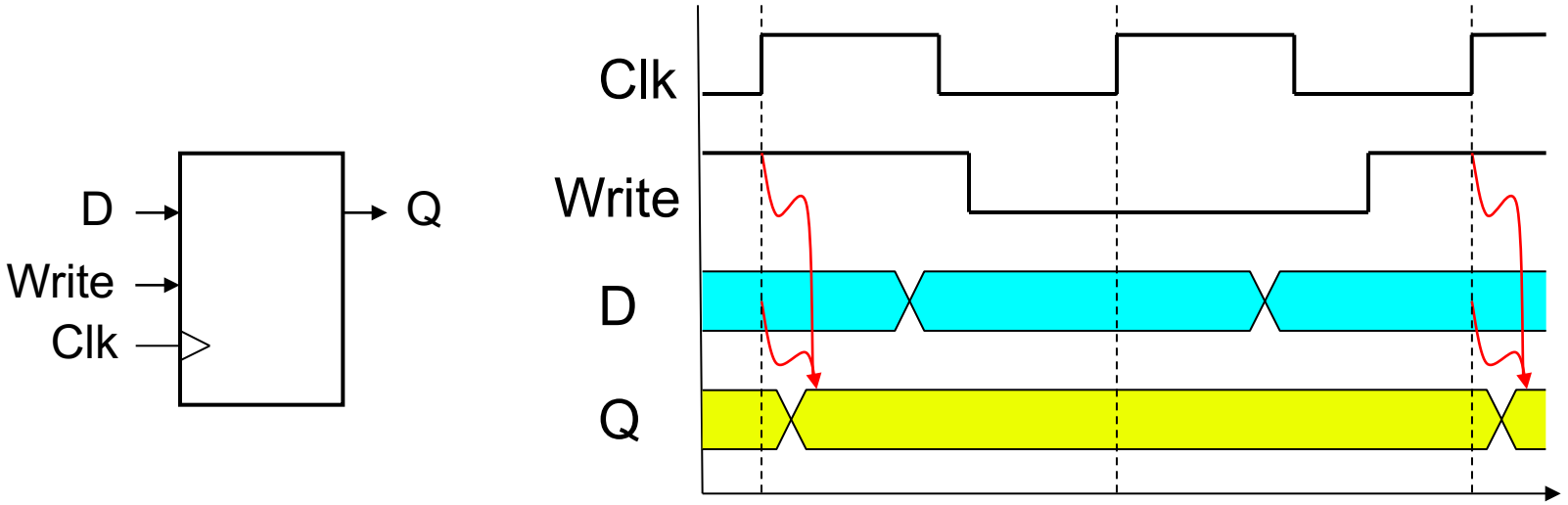
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1





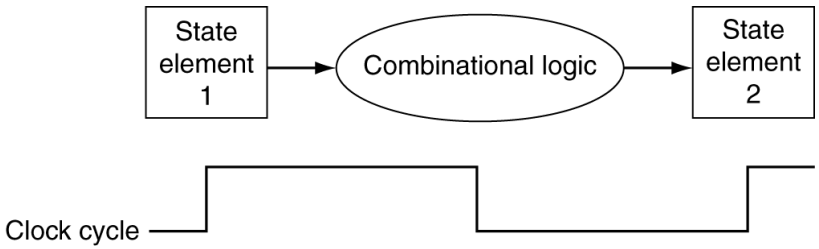
Sequential Elements

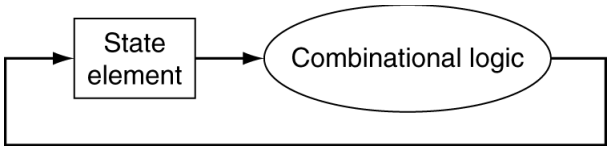
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



Clocking Methodology

- The clocking methodology defines when data in a state element is valid and stable relative to the clock
  - State elements - a memory element such as a register
  - Edge-triggered – all state changes occur on a clock edge
- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period





Lecture 14

CMPEN 331

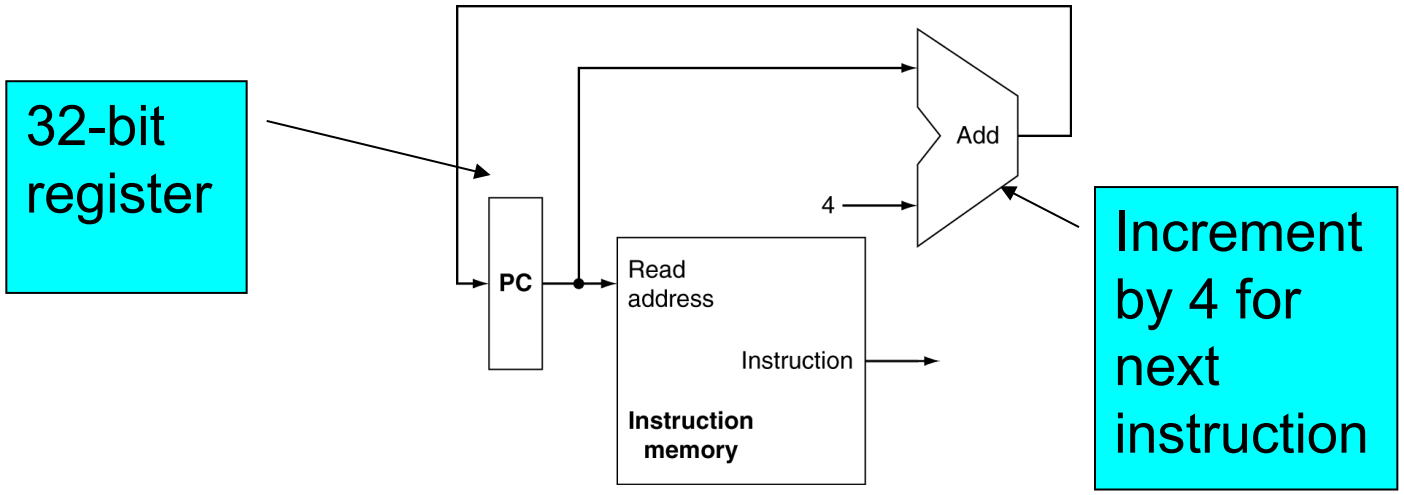
Chapter 4

Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

Instruction Fetch

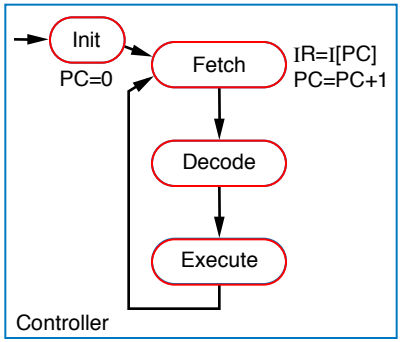
- Fetching instructions involves
  - Reading the instruction from the Instruction Memory
  - Updating the PC value to be the address of the next (sequential) instruction

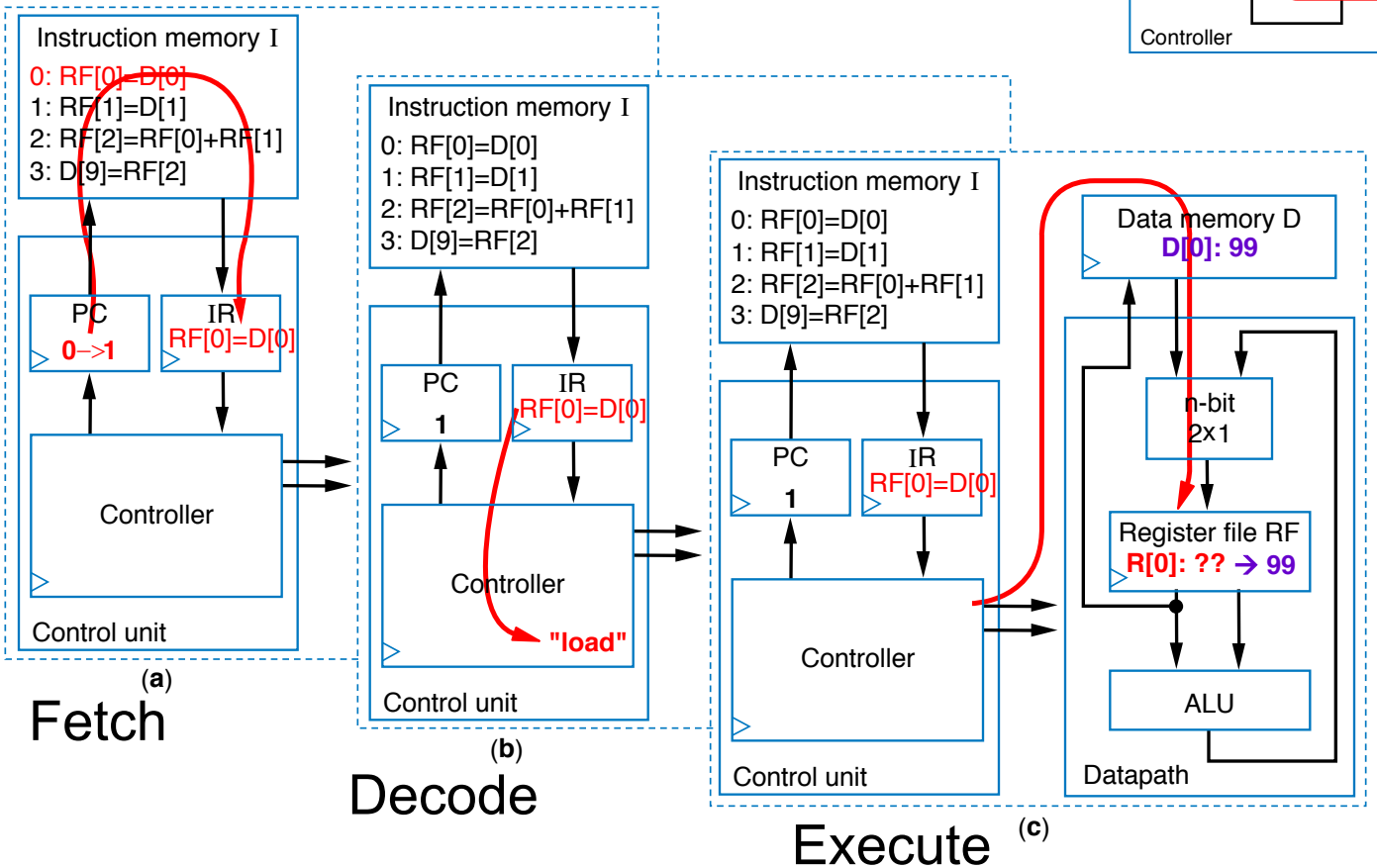


- PC is updated every clock cycle, so it does not need an explicit write control signal just a clock signal
- Reading from the Instruction Memory is a combinational activity, so it doesn't need an explicit read control signal

Basic Architecture – Control Unit

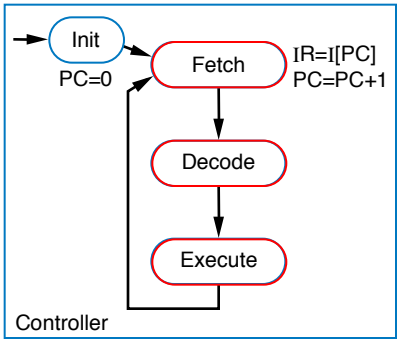
- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath

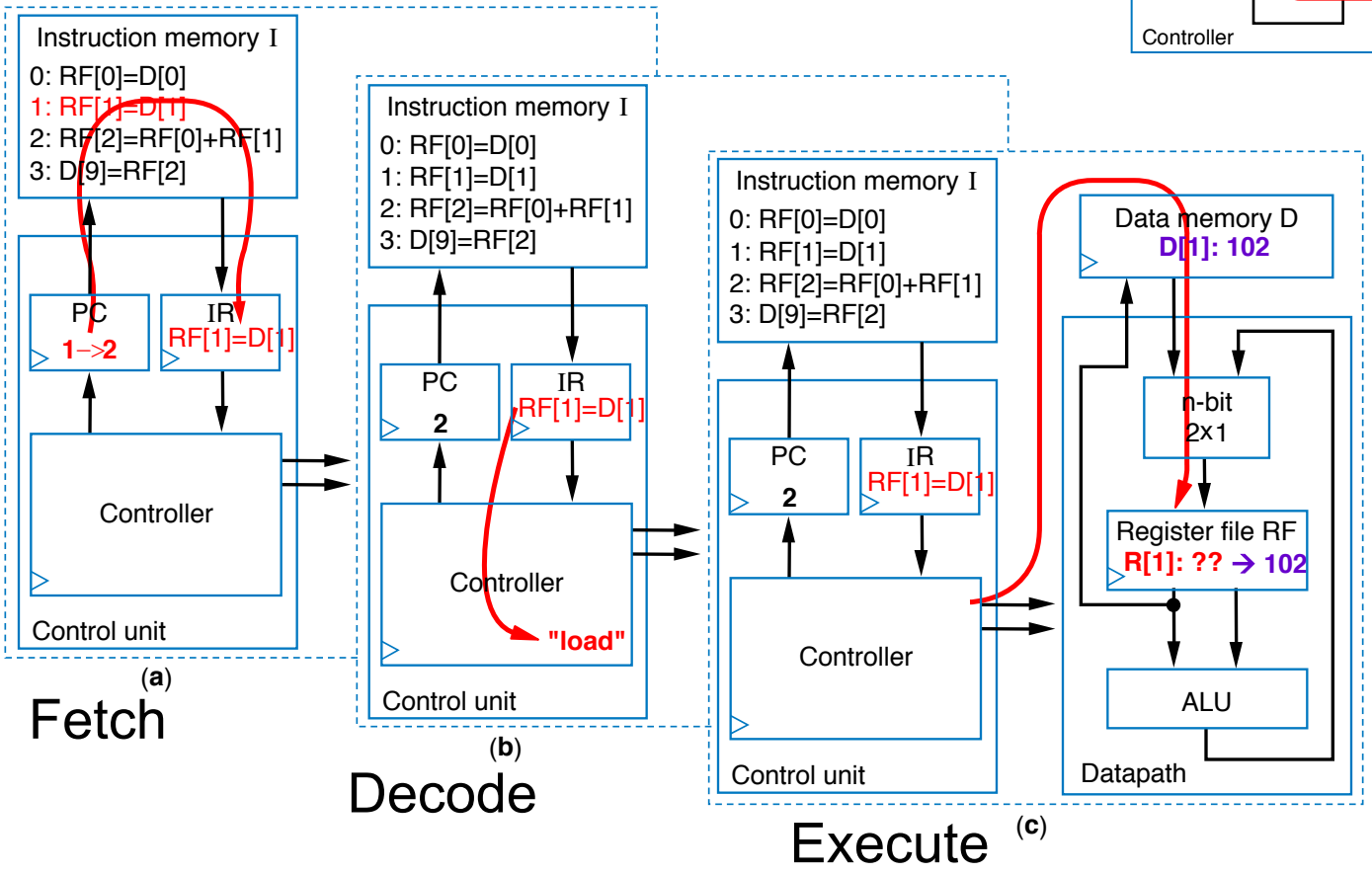




Basic Architecture – Control Unit

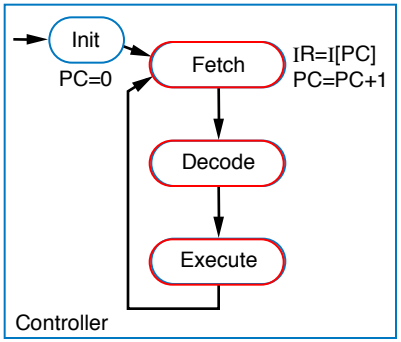
- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath

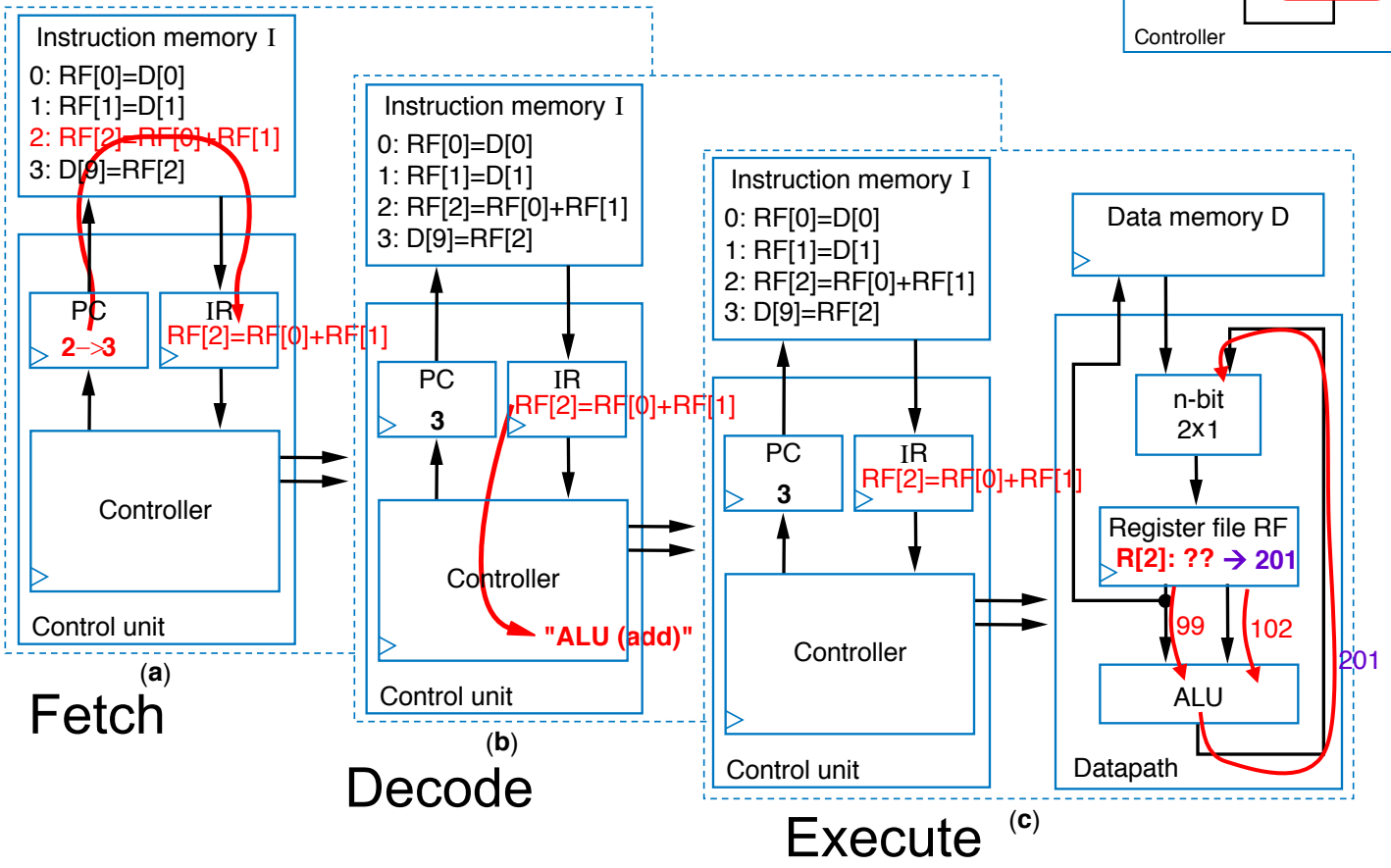




Basic Architecture – Control Unit

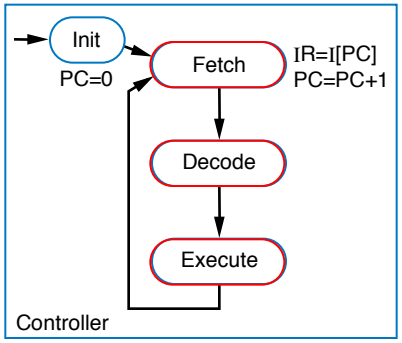
- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath

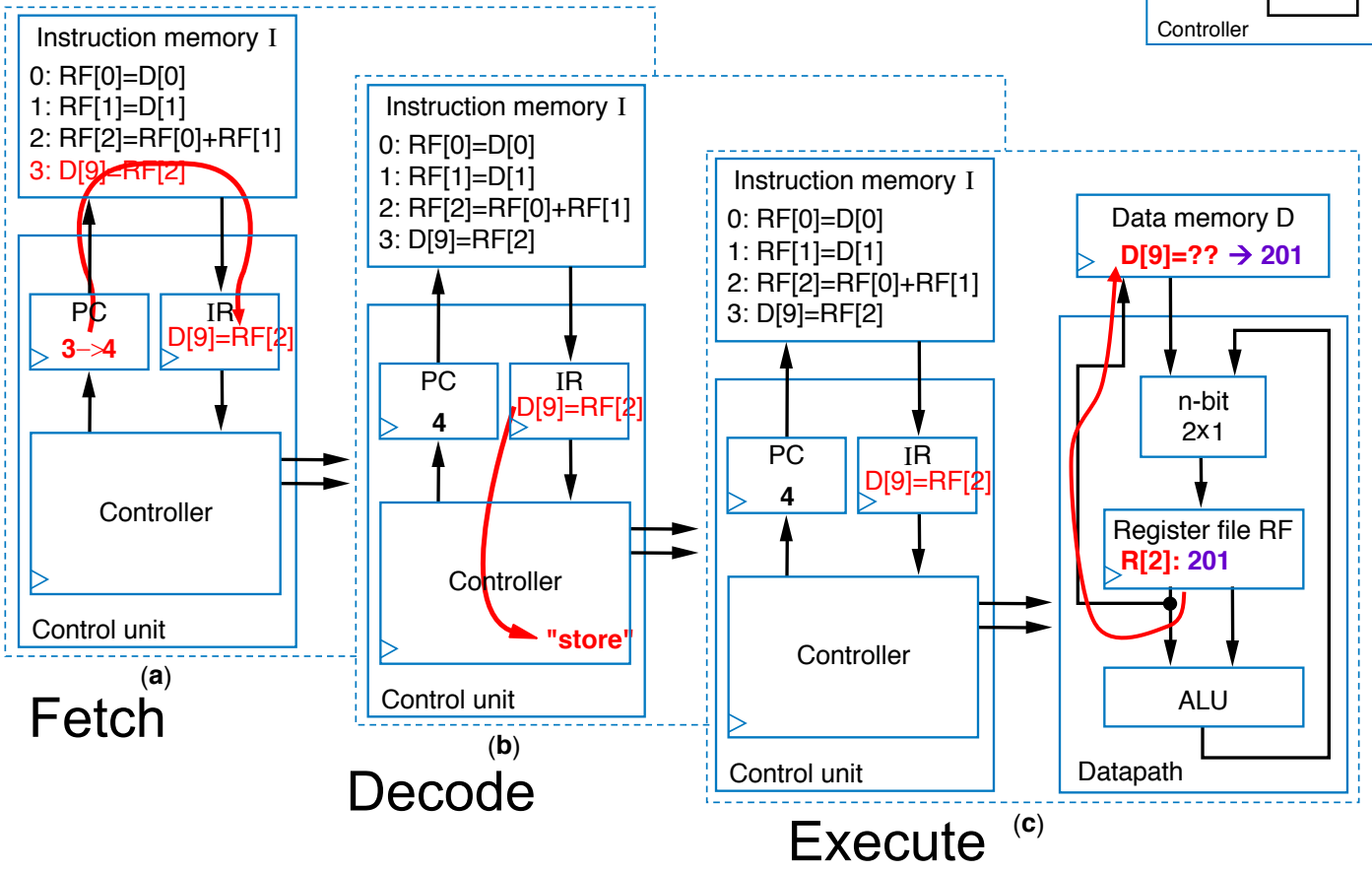




Basic Architecture – Control Unit

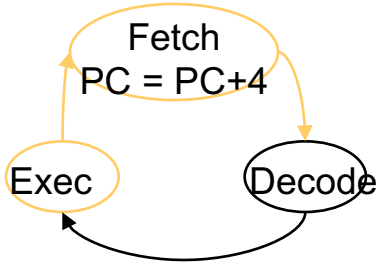
- To carry out *each instruction*, the control unit must:
  - Fetch – Read instruction from inst. mem.
  - Decode – Determine the operation and operands of the instruction
  - Execute – Carry out the instruction's operation using the datapath

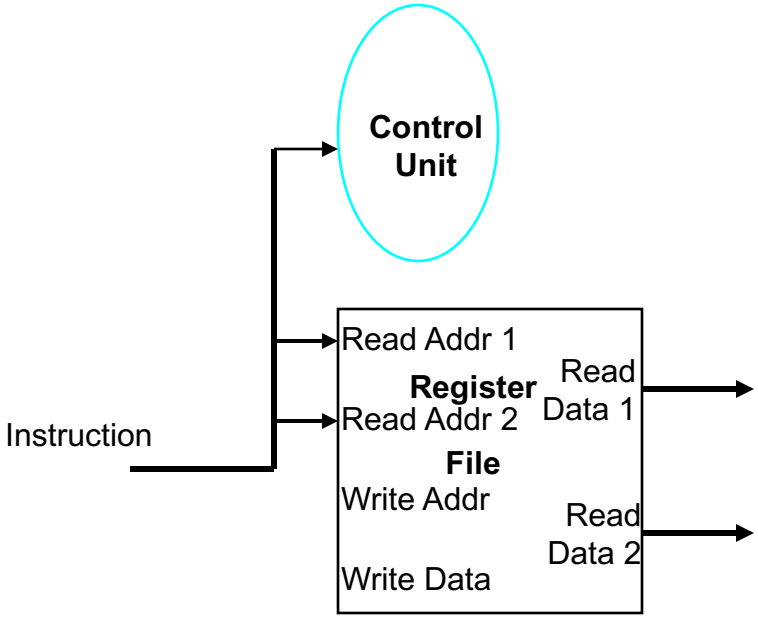




Decoding Instructions

- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit



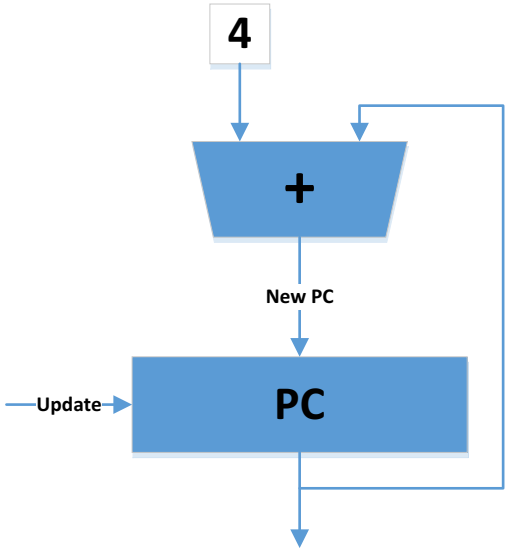


and

  - reading two values from the Register File
    - Register File addresses are contained in the instruction

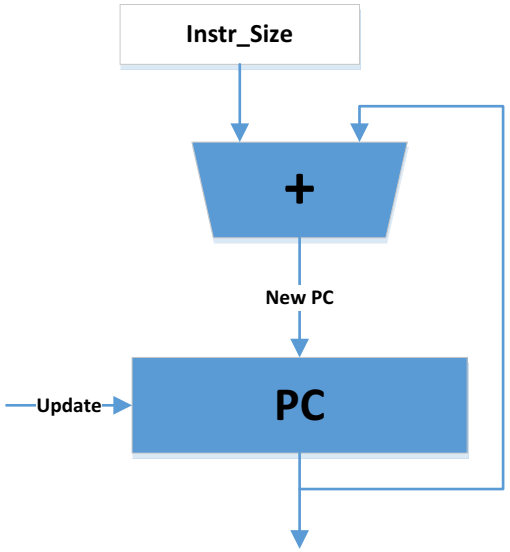
Fixed Program Counter

```
module program_counter
(
      input                      update,
      input                      clk,
      input                      rst,
      output   reg [31:0]        pc
);

parameter INCREMENT_AMOUNT = 32'd4;

always @(posedge clk or posedge rst)
begin
      if (rst)
            pc <= 0;
      else if (update)
            pc <= pc + INCREMENT_AMOUNT;
end
```



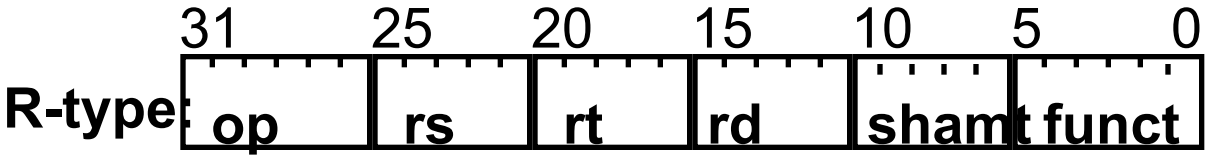
Variable Program Counter

```
module program_counter
(
      input                   update,
      input         [31:0]    instruction_size,
      input                   clk,
      input                   rst,
      output   reg [31:0]     pc
);

always @(posedge clk or posedge rst)
begin
      if (rst)
            pc <= 0;
      else if (update)
            pc <= pc + instruction_size;
end
```

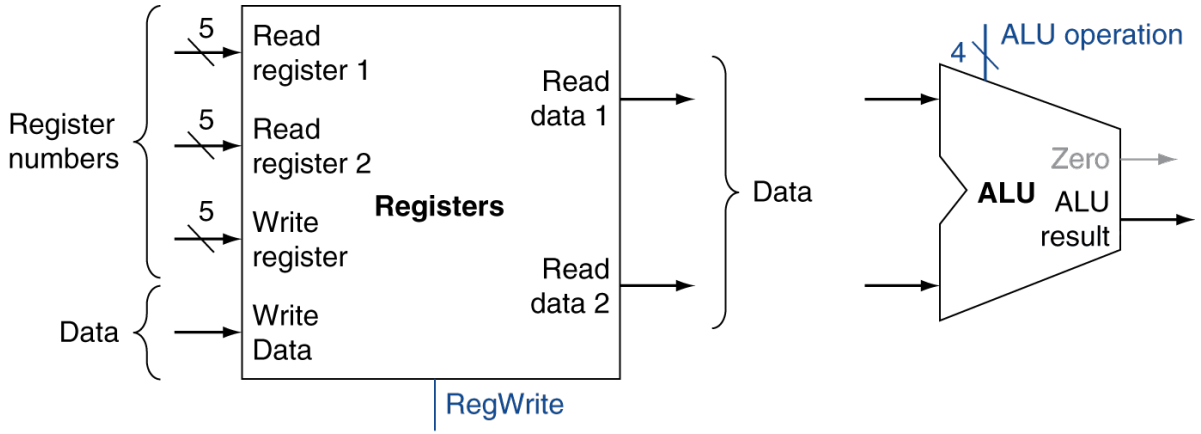


R-Format Instructions

- R format operations (**`add, sub, slt, and, or`**)



  - perform operation (op and funct) on values in rs and rt
  - store the result back into the Register File (into location rd)
  - Note that Register File is not written every cycle (e.g. **`sw`**), so we need an explicit write control signal for the Register File

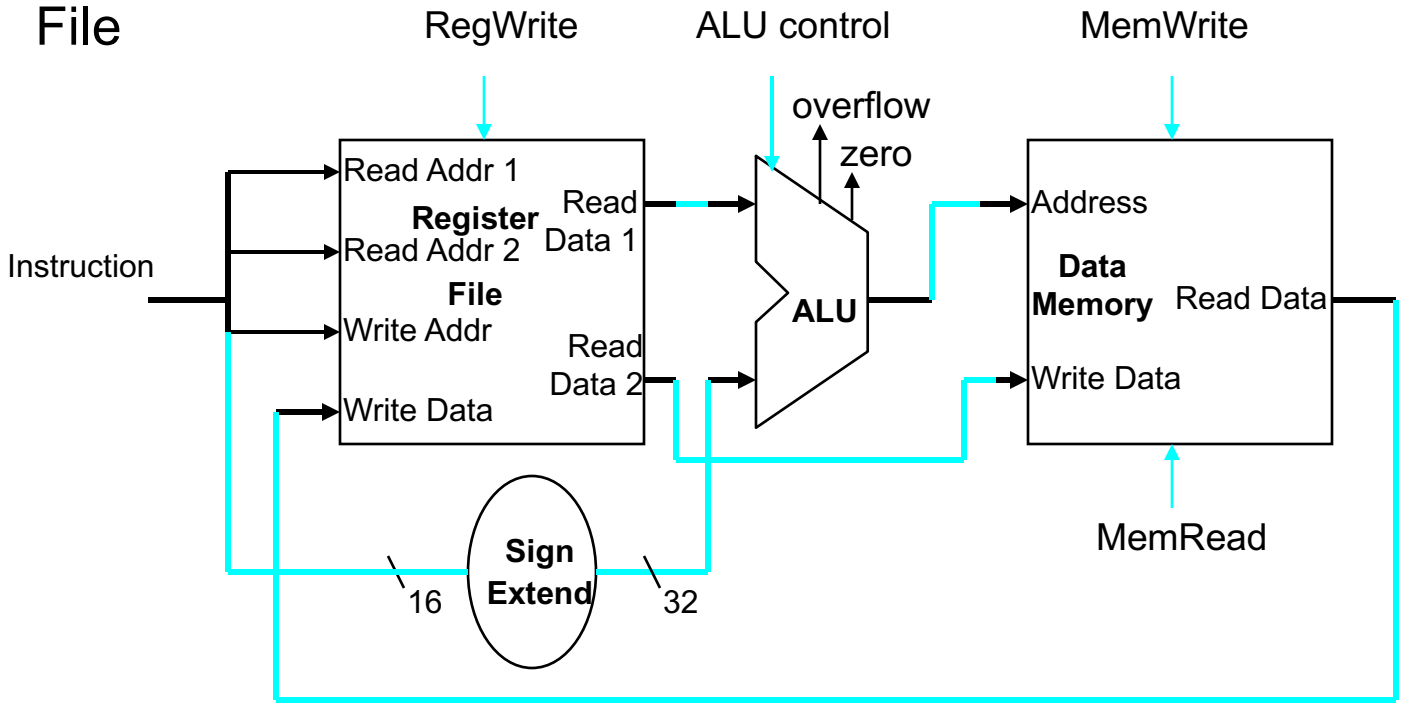


a. Registers

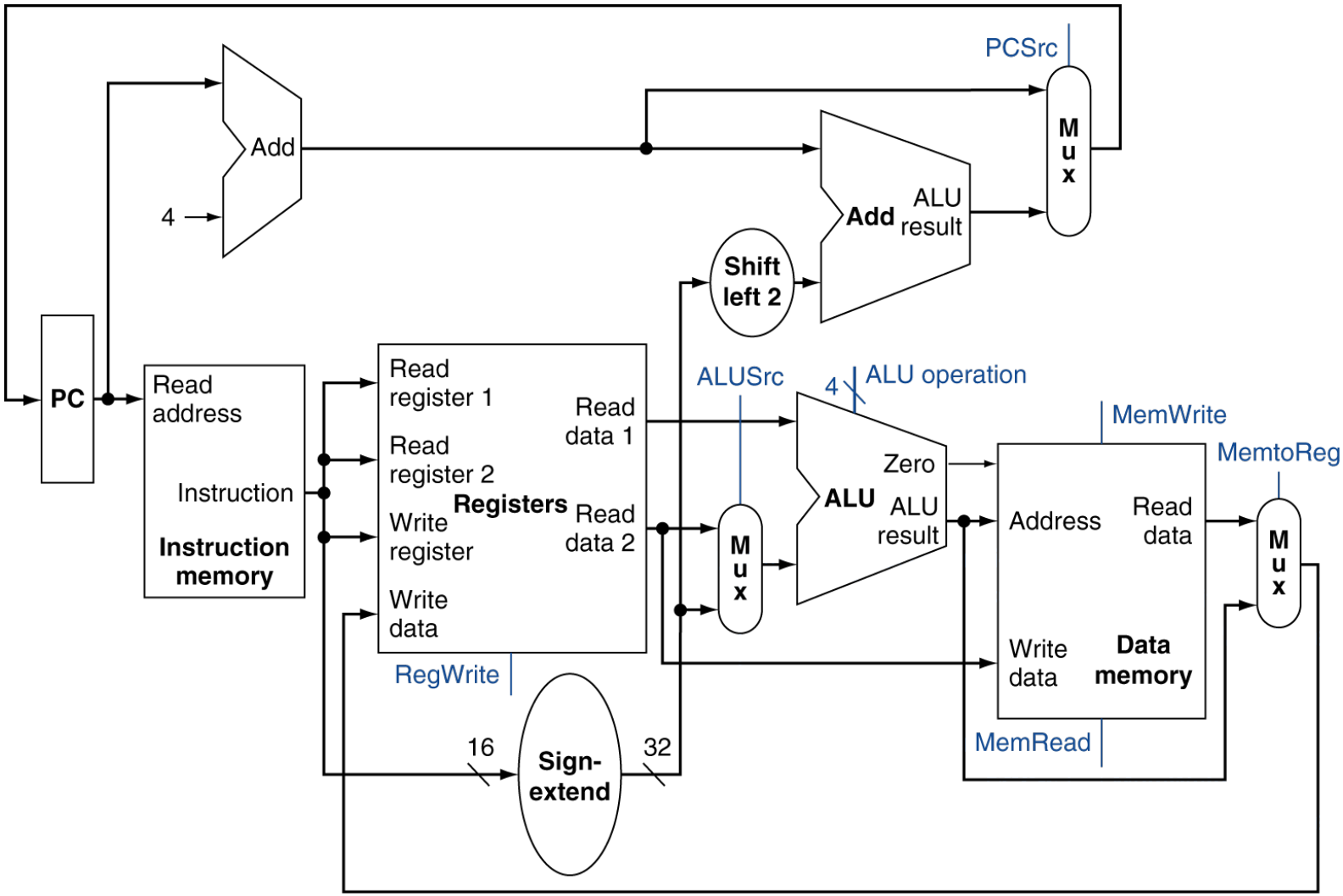
b. ALU

Executing Load and Store Operations

- Load and store operations involves
  - compute memory address by adding the base register (read from the Register File during decode) to the 16-bit signed-extended offset field in the instruction
  - store value (read from the Register File during decode) written to the Data Memory
  - load value, read from the Data Memory, written to the Register File



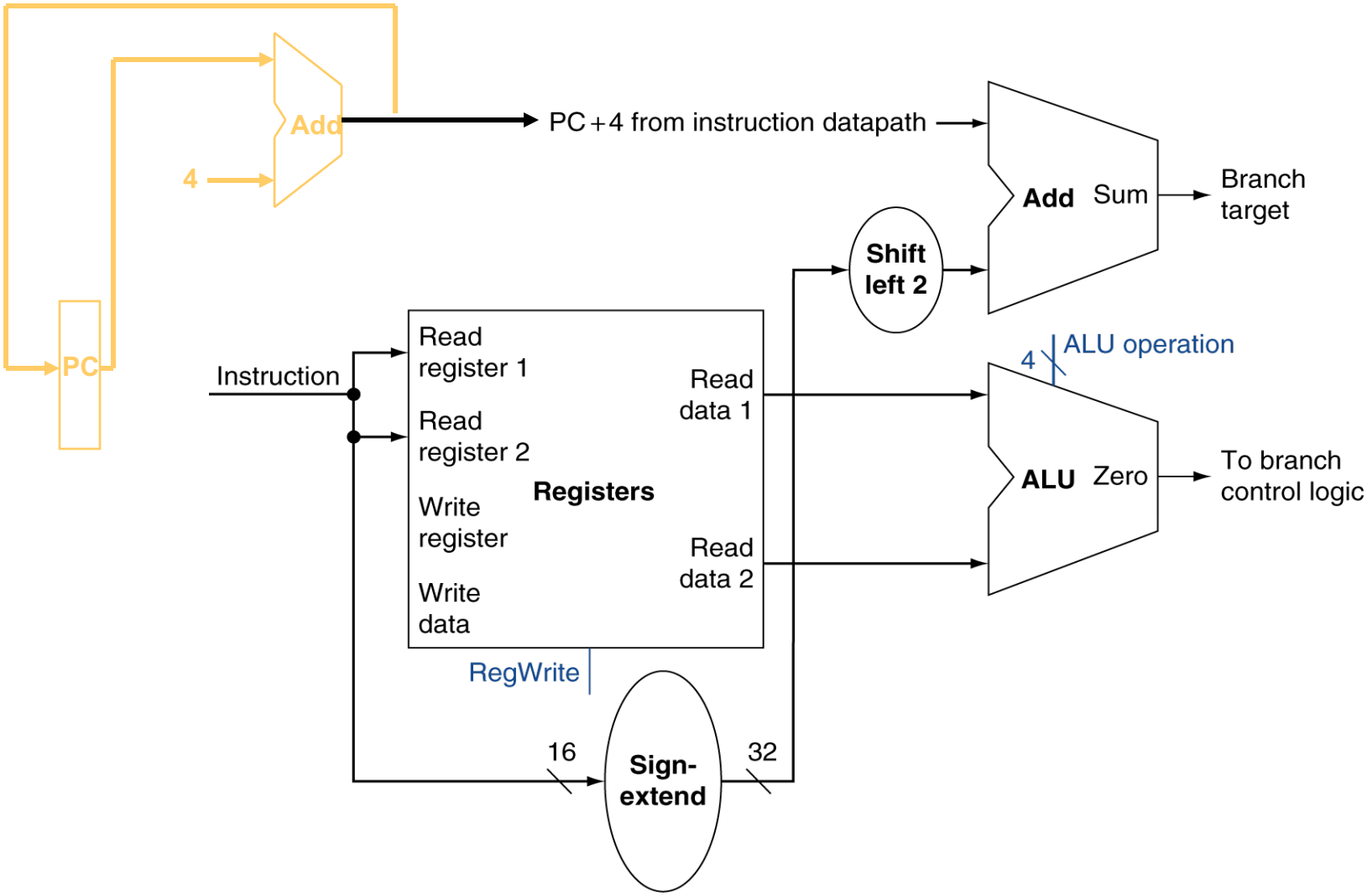
Data path


PCSrc
Add
4
M
u
x
Add
ALU
result
Shift
left 2
PC
Read
address
Instruction
Instruction
memory
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
RegWrite
ALUSrc
4
ALU operation
M
u
x
ALU
Zero
ALU
result
MemWrite
MemtoReg
Address
Read
data
Write
data
Data
memory
MemRead
M
u
x
16
Sign-
extend
32

Branch Instructions

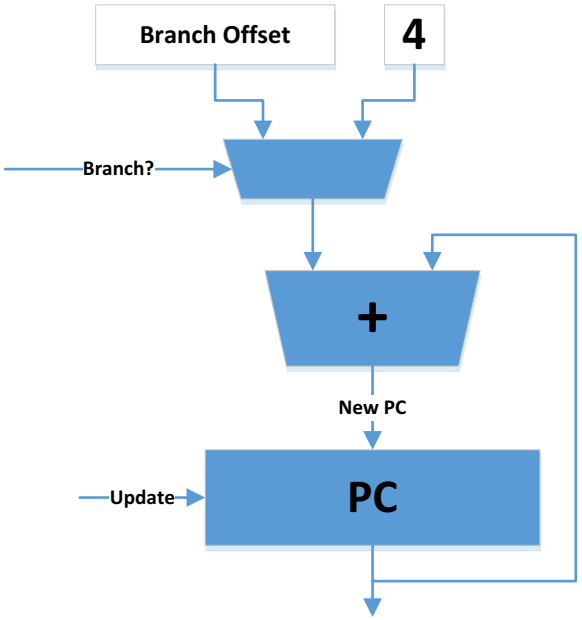
- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Branch Instructions



Fixed Program Counter with Offset Branching

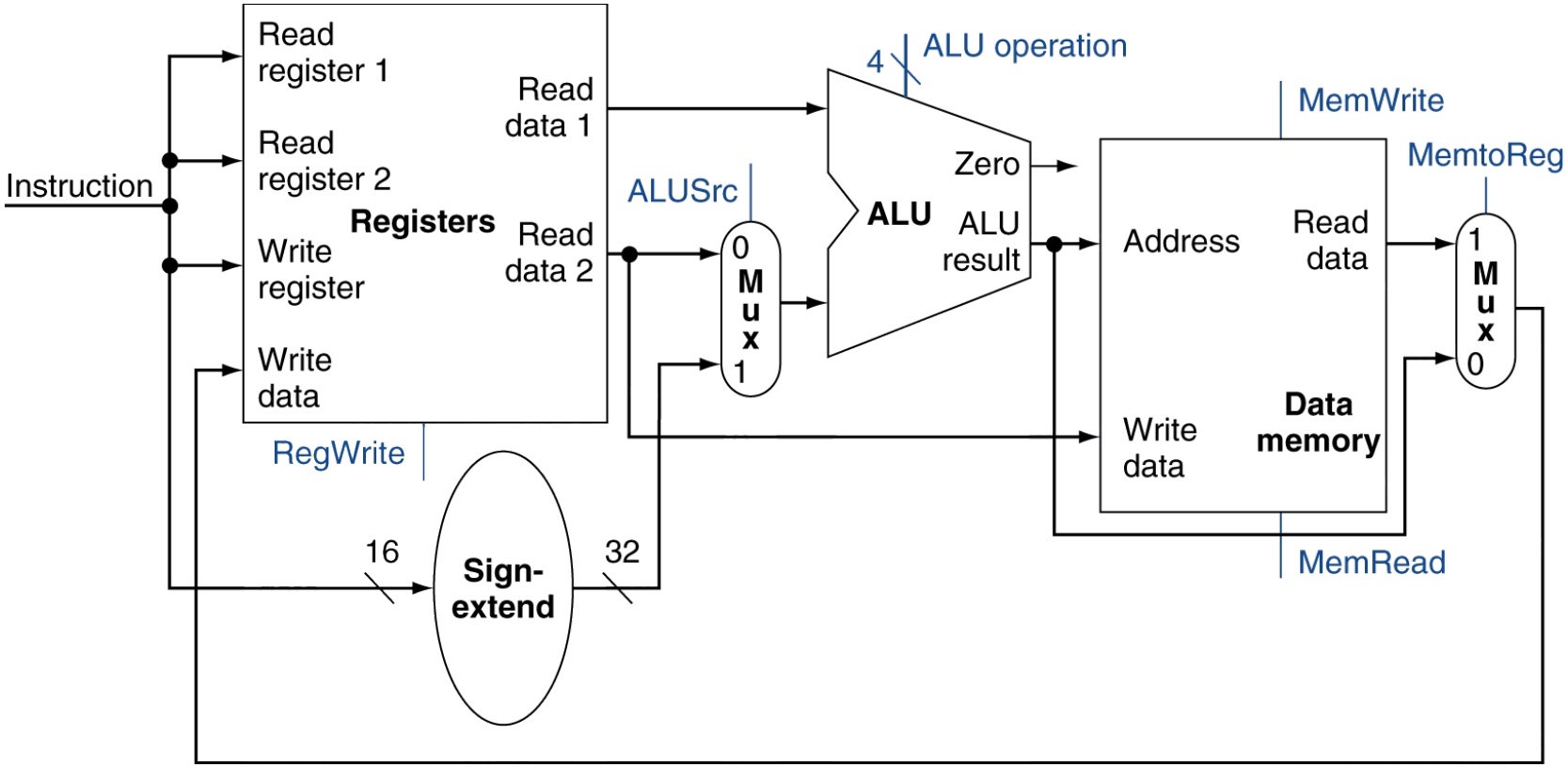
```
module program_counter
(
        input                           update,
        input                           branch,
        input                   [15:0]branch_offset,
        input                           clk,
        input                           rst,
        output      reg [31:0]          pc
);

parameter INCREMENT_AMOUNT = 32'd4;

always @(posedge clk or posedge rst)
begin
        if (rst)
                pc <= 0;
        else if (update)
                if (branch)
                        pc <= pc + {16'd0,branch_offset};
                else
                        pc <= pc + INCREMENT_AMOUNT;
end
```



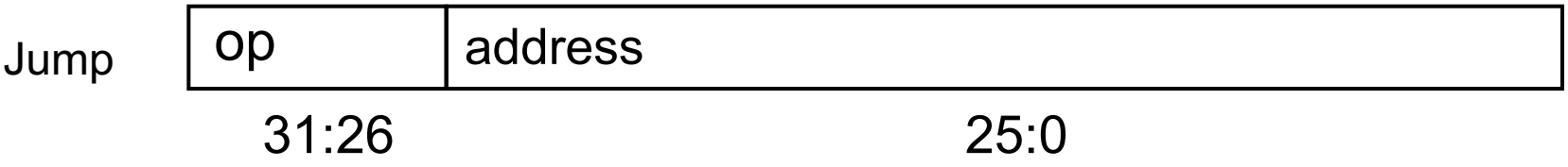
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each data path element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath

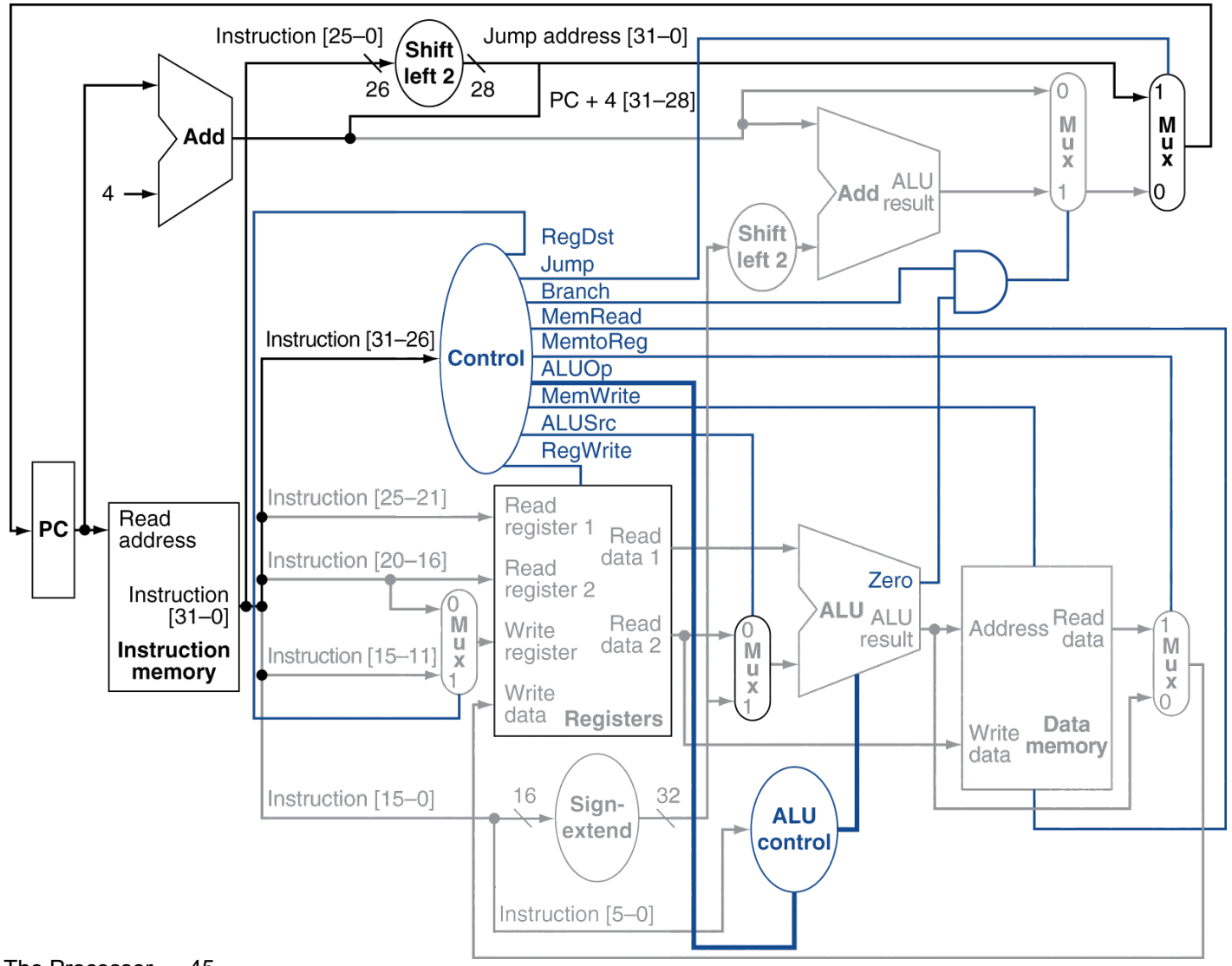


Implementing Jumps



- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

Datapath With Jumps Added



ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

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

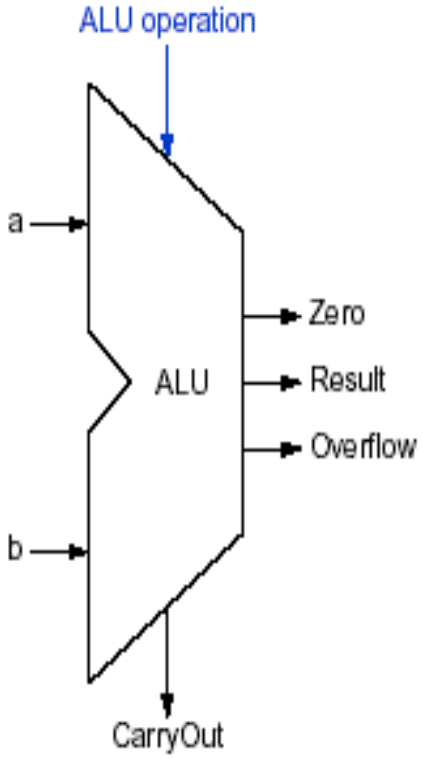
ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

MIPS ALU in Verilog

•The ALU has 7 ports.



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

A Verilog behavioral definition of a MIPS ALU

Pseudo code

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
input [3:0] ALUctl;
input [31:0] A,B;
output reg [31:0] ALUOut;
output Zero;
assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
always @(ALUctl, A, B)
  begin //reevaluate if these change
      case (ALUctl)
            0: ALUOut <= A & B;
            1: ALUOut <= A | B;
            2: ALUOut <= A + B;
            6: ALUOut <= A - B;
            7: ALUOut <= A < B ? 1 : 0;
            12: ALUOut <= ~(A | B); //result is nor default: ALUOut <= 0;
      endcase
  end
endmodule
```

The MIPS ALU control

This is a combinational control logic. (Pseudo code)

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
input [1:0] ALUOp;
input [5:0] FuncCode;
output [3:0] reg ALUCtl;
always @(*)
case (FuncCode)
        32: ALUCtl <=2; // add
        34: ALUCtl <=6; //subtract
        36: ALUCtl <=0; // and
        37: ALUCtl <=1; // or
        39: ALUCtl <=12; // nor
        42: ALUCtl <=7; // slt
    default: ALUCtl <=15; // should not happen
endcase
endmodule
```

The Main Control Unit

- Control signals derived from instruction