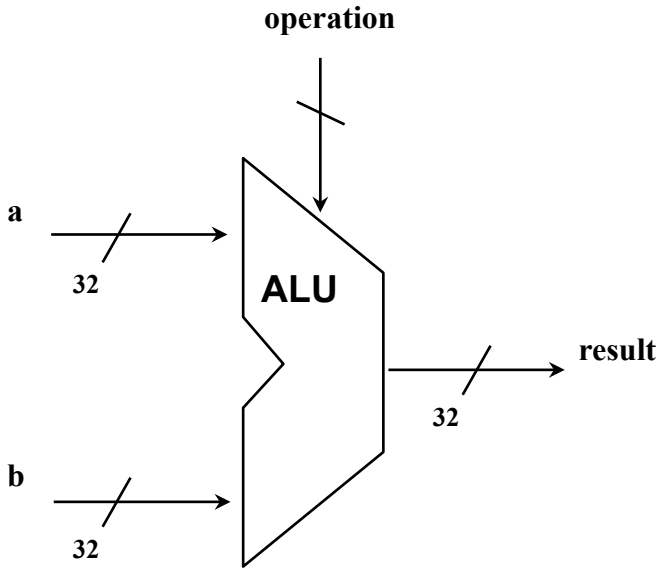
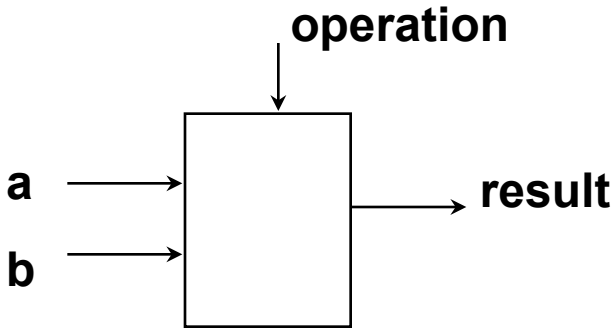
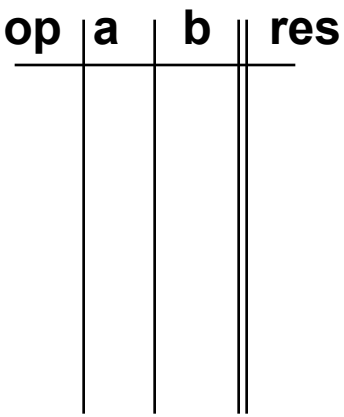
Lets Build a Processor

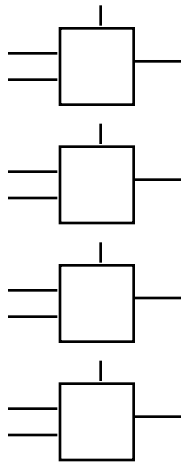


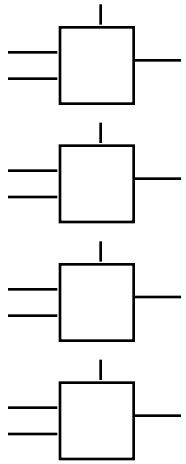
An ALU (arithmetic logic unit)

- Let's build an ALU to support the `andi` and `ori` instructions
  - we'll just build a 1 bit ALU, and use 32 of them





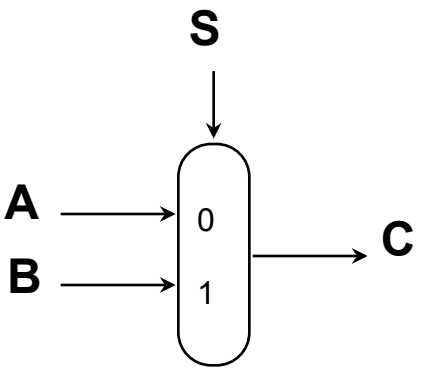




- Possible Implementation (sum-of-products):

Review: The Multiplexor

- Selects one of the inputs to be the output, based on a control input

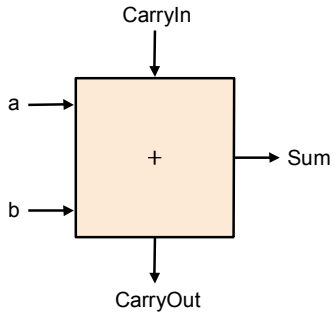


note: we call this a 2-input mux
even though it has 3 inputs!

- Lets build our ALU using a MUX:

Different Implementations

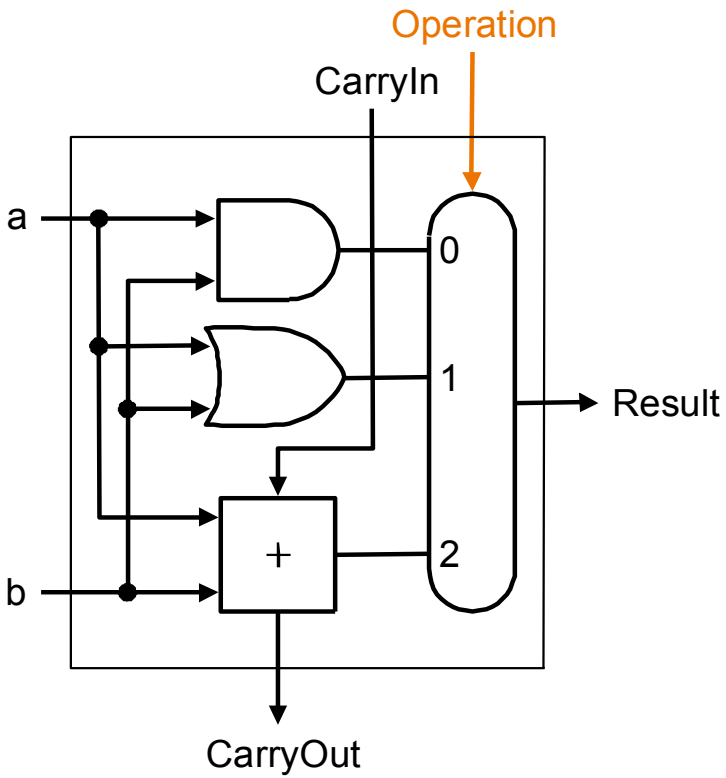
- Not easy to decide the "best" way to build something
  - Don't want too many inputs to a single gate
  - Don't want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:

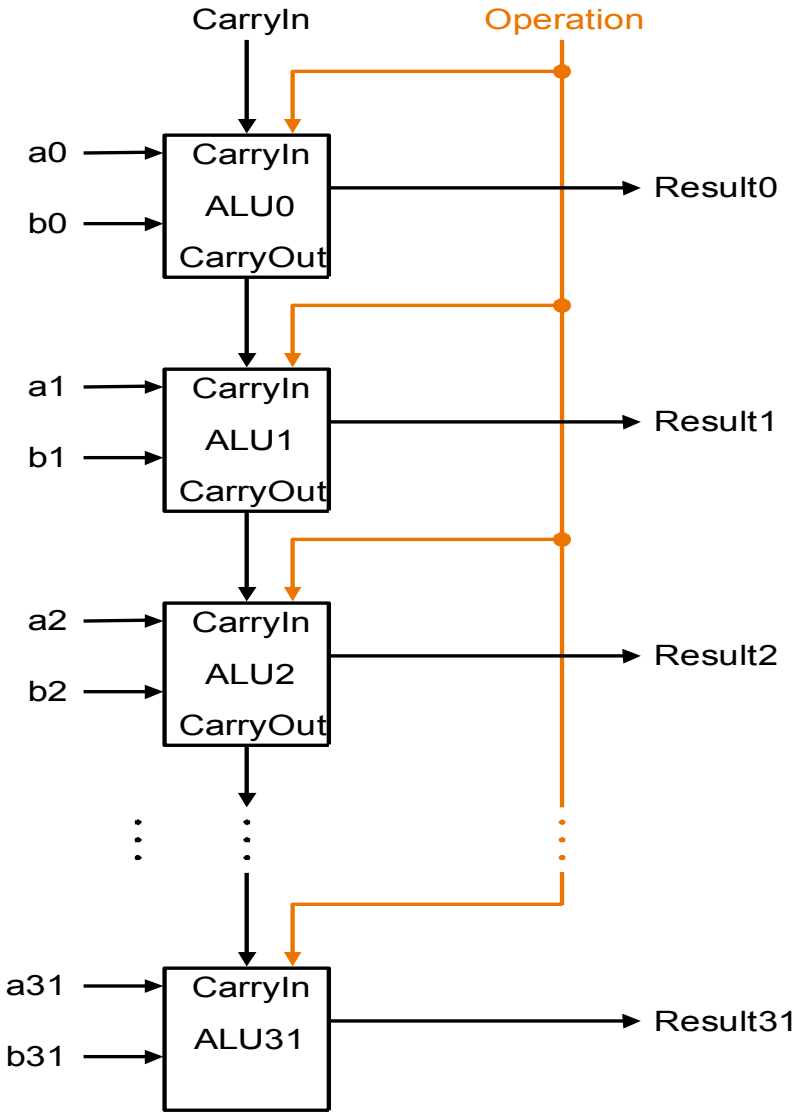


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

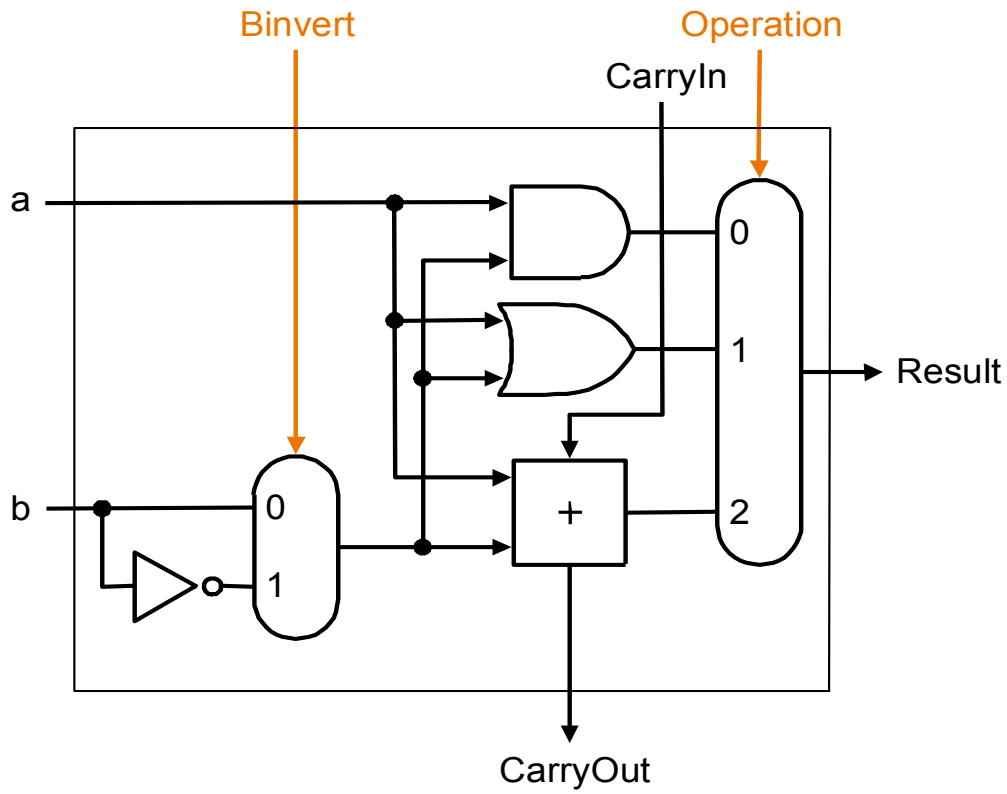
Building a 32 bit ALU





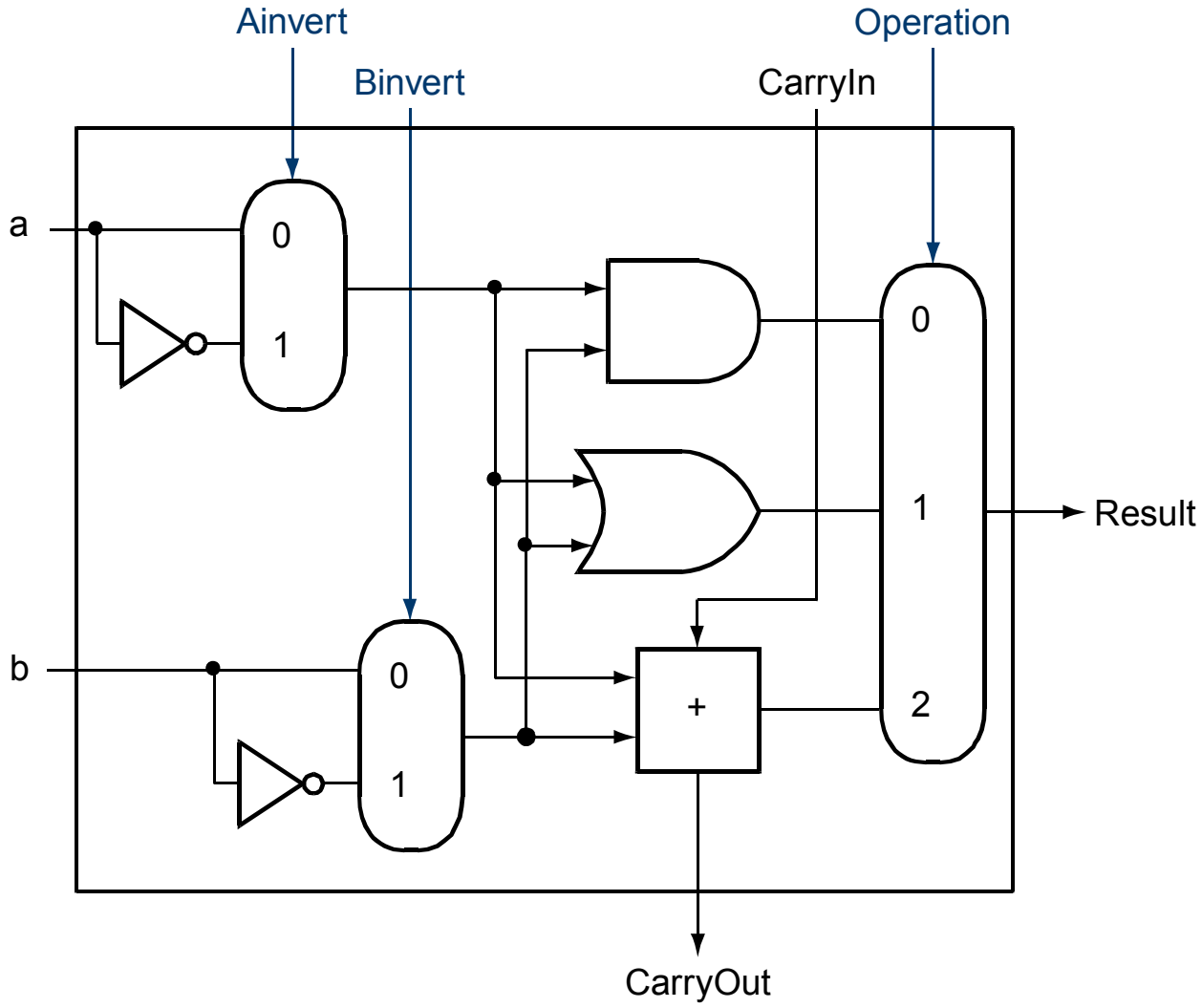
What about subtraction (a – b) ?

- Two's complement approach: just negate b and add.
- How do we negate?
- A very clever solution:



Adding a NOR function

- Can also choose to invert a. How do we get "a NOR b"?

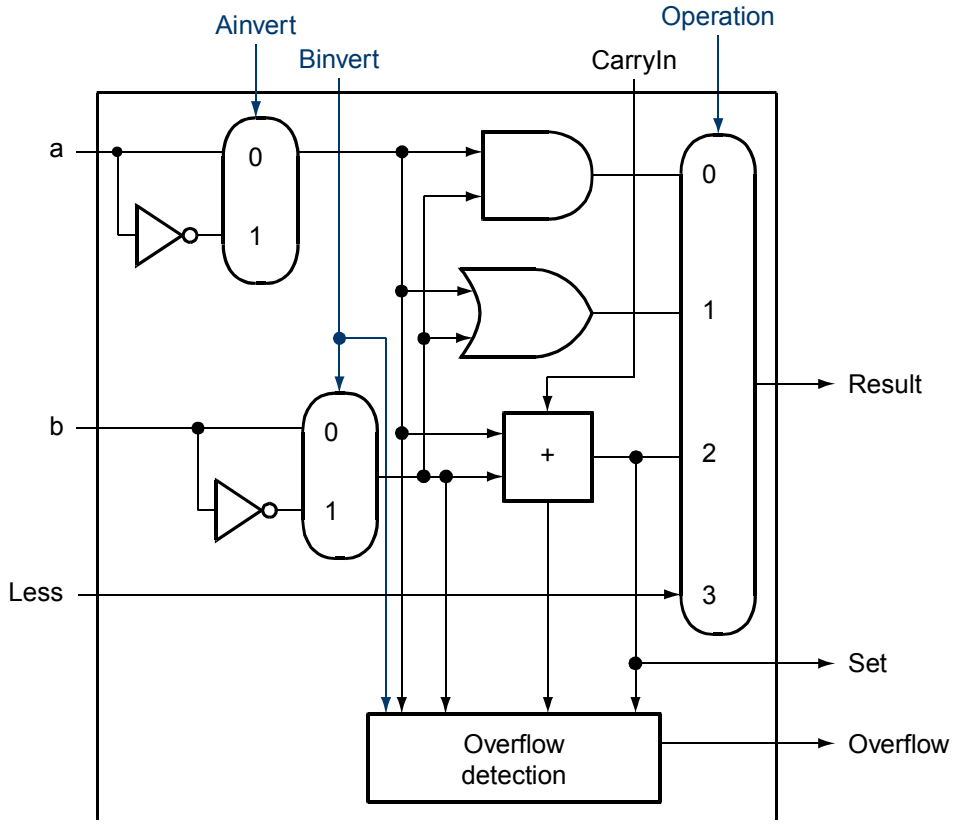


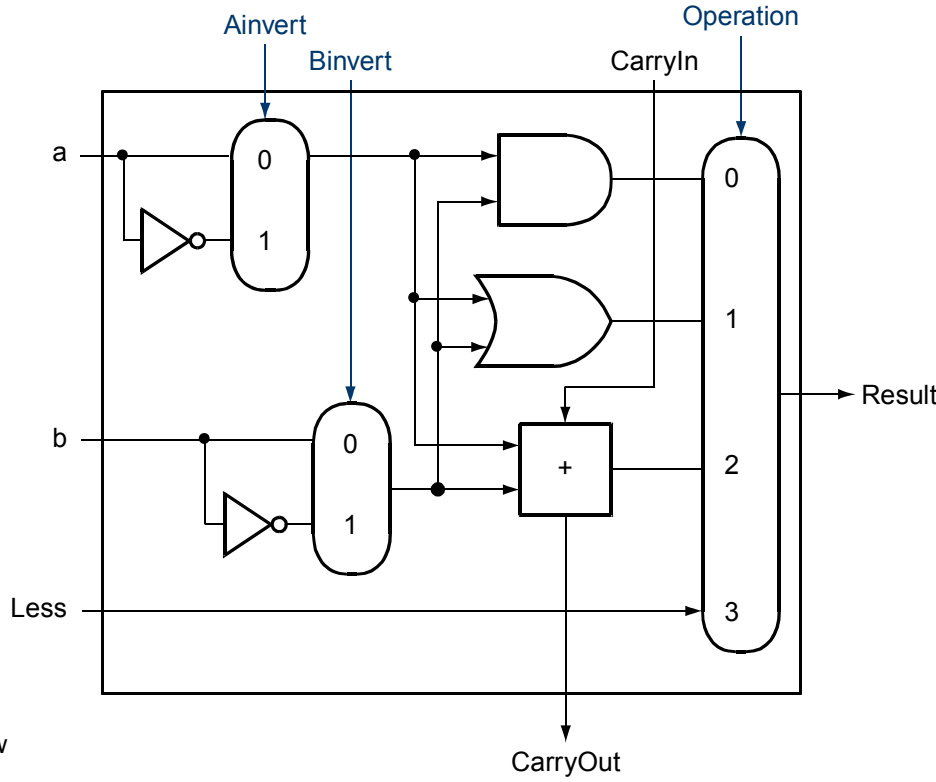
Tailoring the ALU to the MIPS

- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (beq $t5, $t6, $t7)
  - use subtraction: (a-b) = 0 implies a = b

Supporting slt

- Can we figure out the idea?

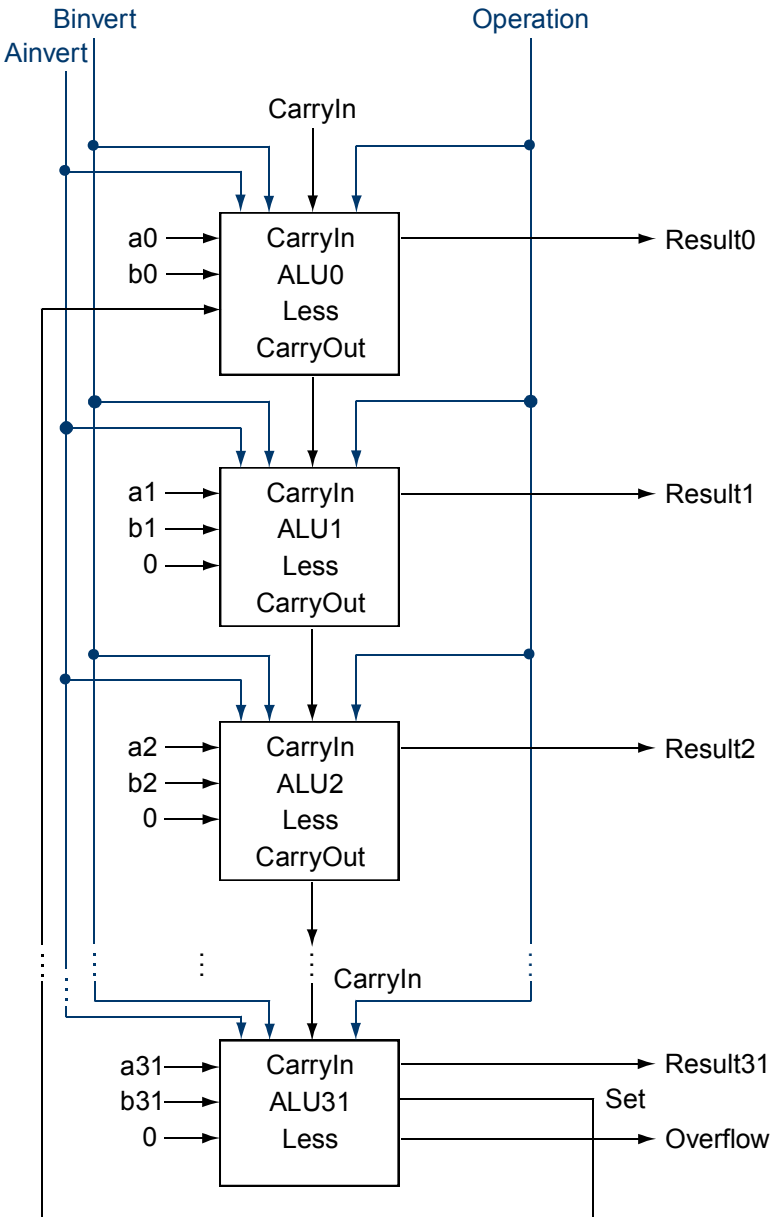




Use this ALU for most significant bit

all other bits

Supporting slt

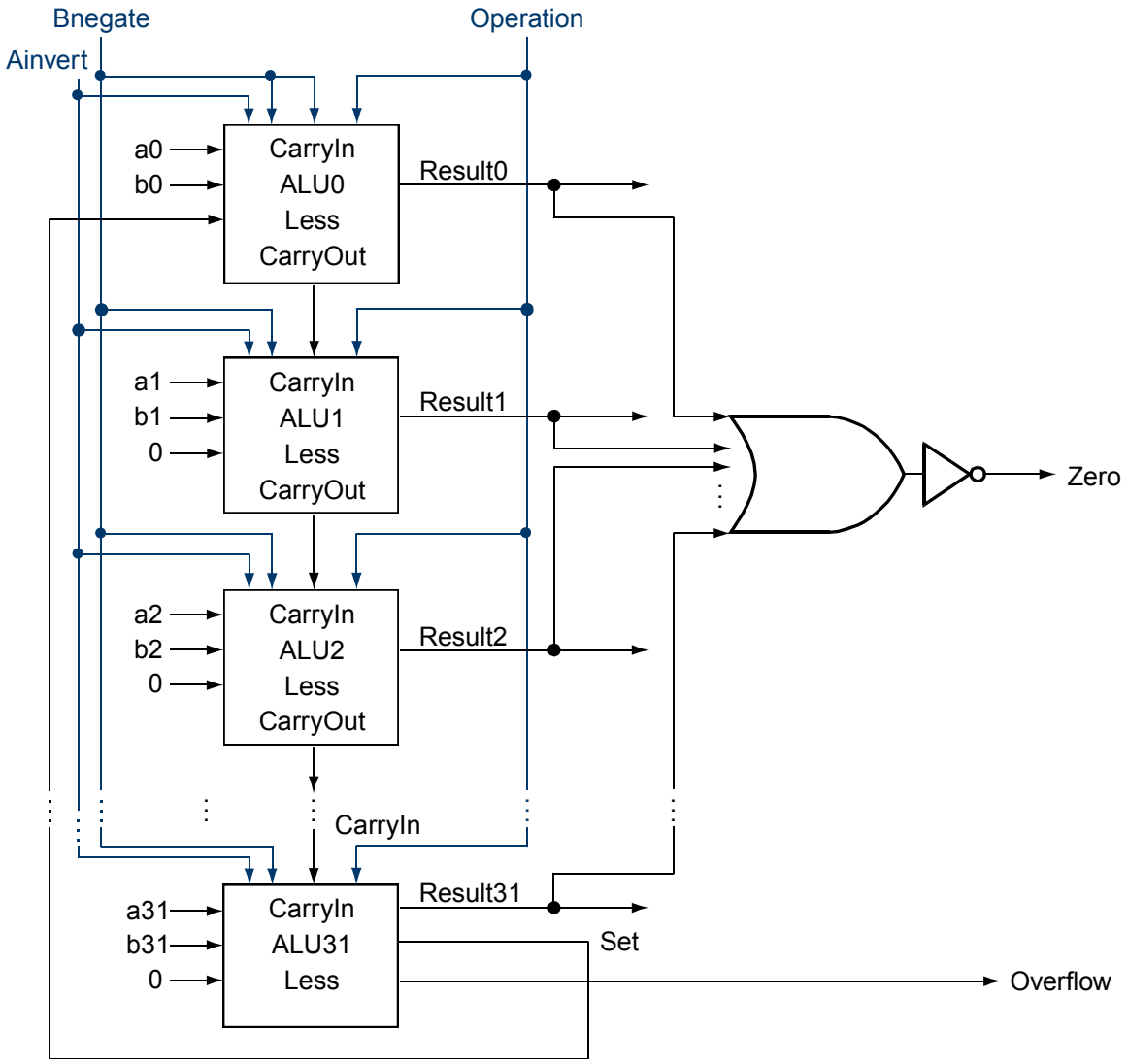


Test for equality

- Notice control lines:

```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
1100 = NOR
```

Note: zero is a 1 when the result is zero!



Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU

- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")

- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)

ALU Summary

- We can build an ALU to support MIPS addition
- Our focus is on comprehension, not performance
- Real processors use more sophisticated techniques for arithmetic (e.g. do not use ripple carry adder)
- Where performance is not critical, hardware description languages allow designers to completely automate the creation of hardware!

```
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; goes anywhere
   always @(ALUctl, A, B) //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; //default to 0, should not happen;
      endcase
endmodule
```

FIGURE B.4.3 A Verilog behavioral definition of a MIPS ALU. This could be synthesized using a module library containing basic arithmetic and logical operations.

Basic MIPS Architecture Review

[Adapted from Mary Jane Irwin for
Computer Organization and Design,
Patterson & Hennessy, © 2005, UCB]

Review: THE Performance Equation

- Our basic performance equation is then

$$\text{CPU time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle}$$

or

$$\text{CPU time} = \frac{\text{Instruction\_count} \times \text{CPI}}{\text{clock\_rate}}$$

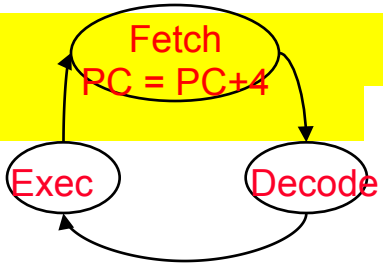
- These equations separate the three key factors that affect performance
  - Can measure the CPU execution time by running the program
  - The clock rate is usually given in the documentation
  - Can measure instruction count by using profilers/simulators without knowing all of the implementation details
  - CPI varies by instruction type and ISA implementation for which we must know the implementation details

So the first area of craftsmanship is in trading function for size. … The second area of craftsmanship is space-time trade-offs. For a given function, the more space, the faster.

The Mythical Man-Month, Brooks, pg. 101

The Processor: Datapath & Control

- Our implementation of the MIPS is simplified
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- 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

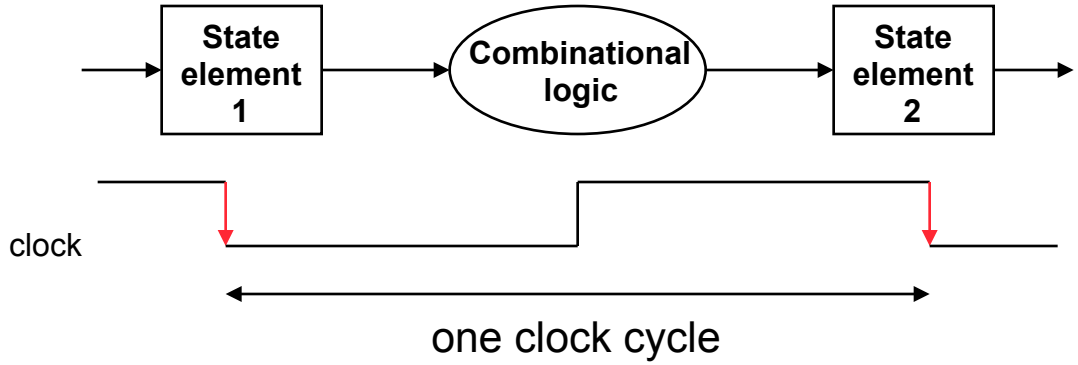


- All instructions (except `j`) use the ALU after reading the registers

How? memory-reference? arithmetic? control flow?

Clocking Methodologies

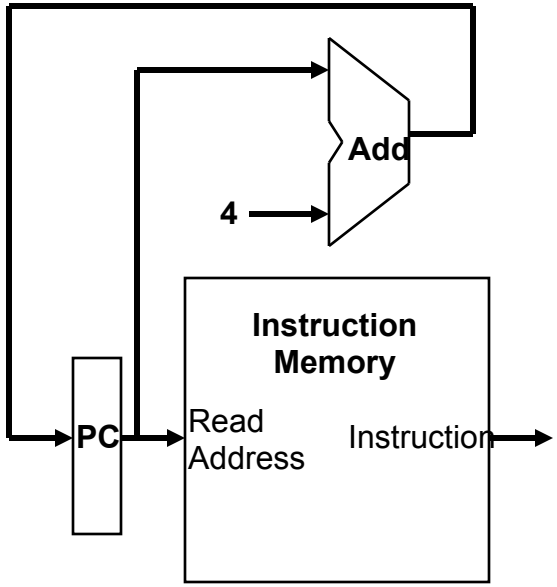
- The clocking methodology defines when signals can be read and when they are written
  - An edge-triggered methodology
- Typical execution
  - read contents of state elements
  - send values through combinational logic
  - write results to one or more state elements



- Assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - write occurs only when both the write control is asserted and the clock edge occurs

Fetching Instructions

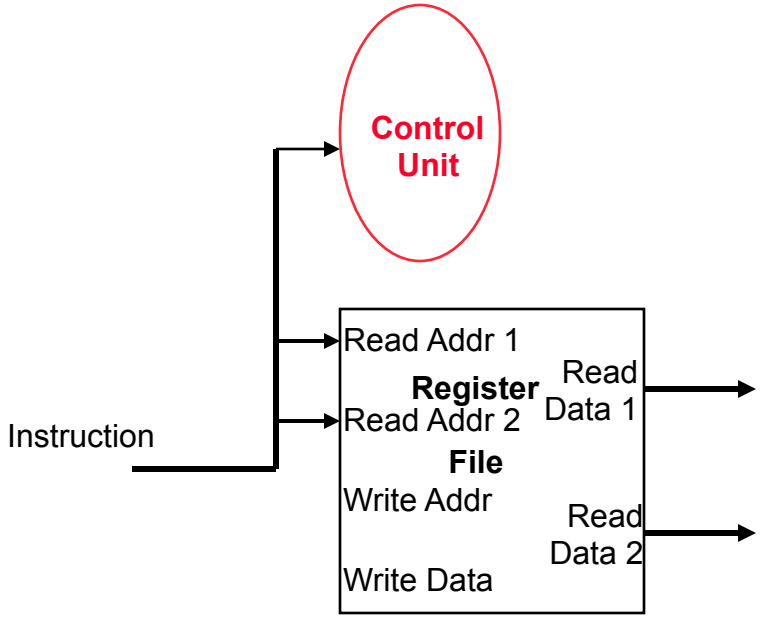
- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC to hold the address of the next instruction



  - PC is updated every cycle, so it does not need an explicit write control signal
  - Instruction Memory is read every cycle, so it doesn't need an explicit read control signal

Decoding Instructions

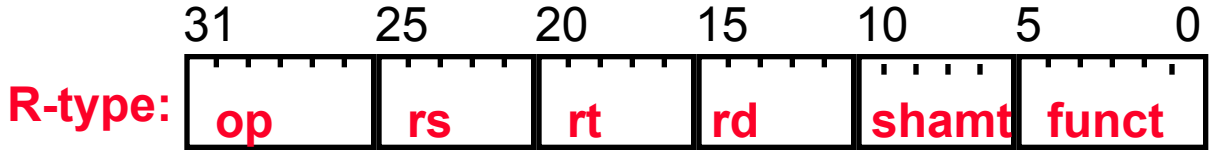
- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit



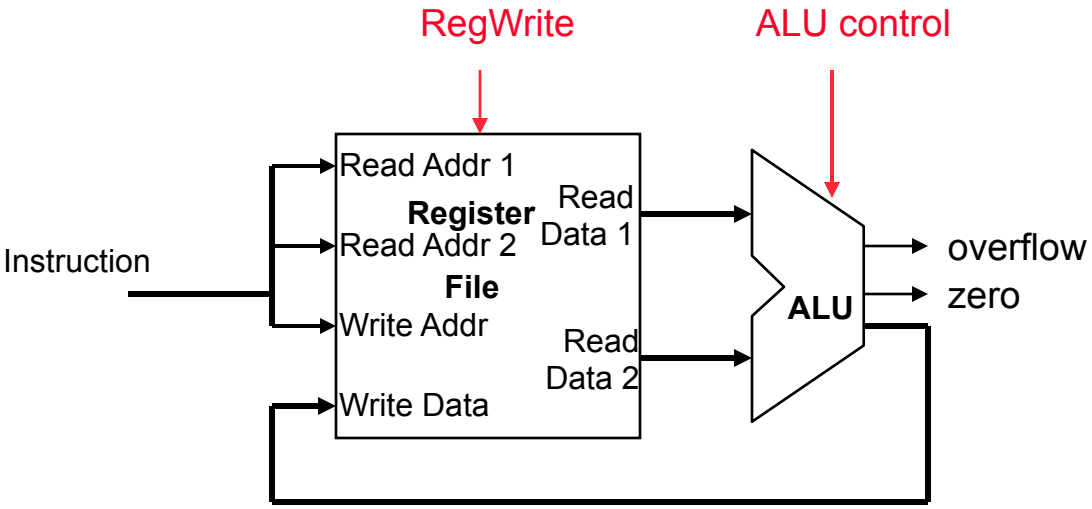
  - reading two values from the Register File
    - Register File addresses are contained in the instruction

Executing R Format Operations

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



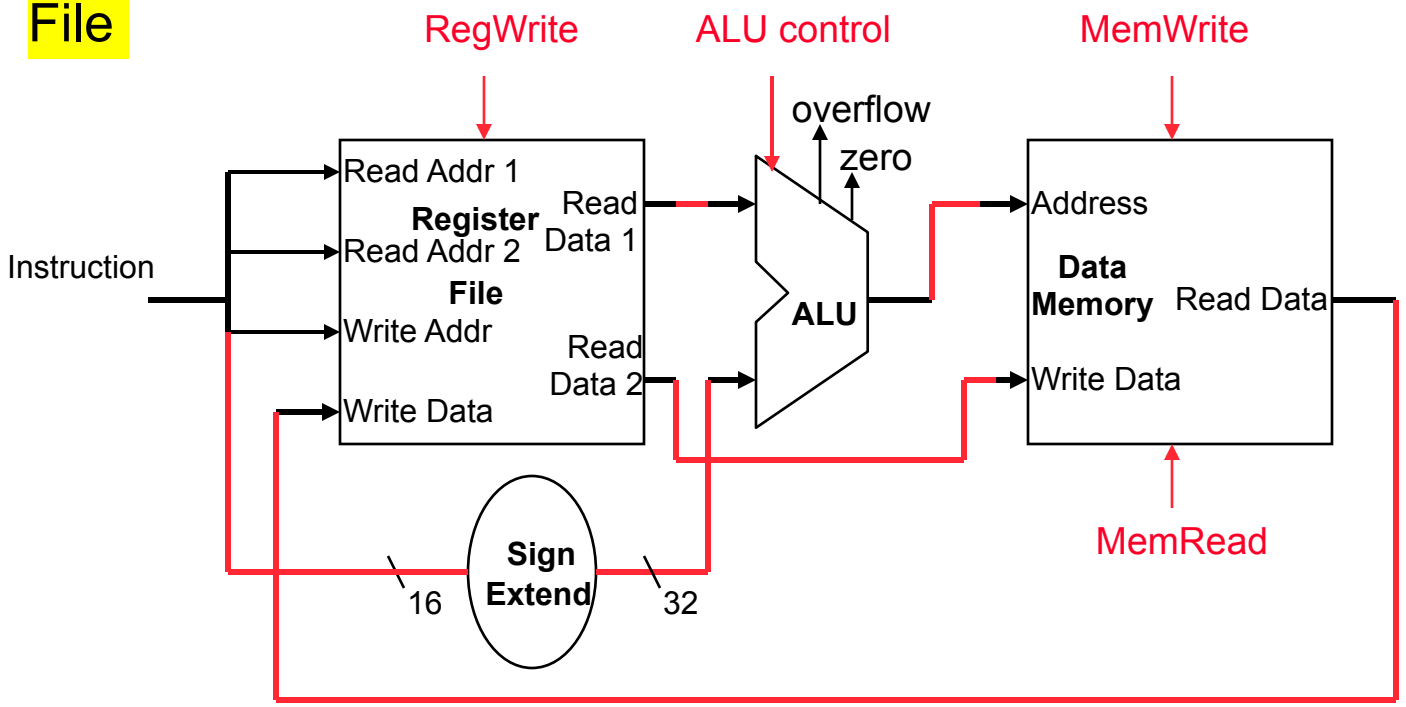
- perform the (op and funct) operation on values in rs and rt
- store the result back into the Register File (into location rd)



- The Register File is not written every cycle (e.g. `sw`), so we need an explicit write control signal for the Register File

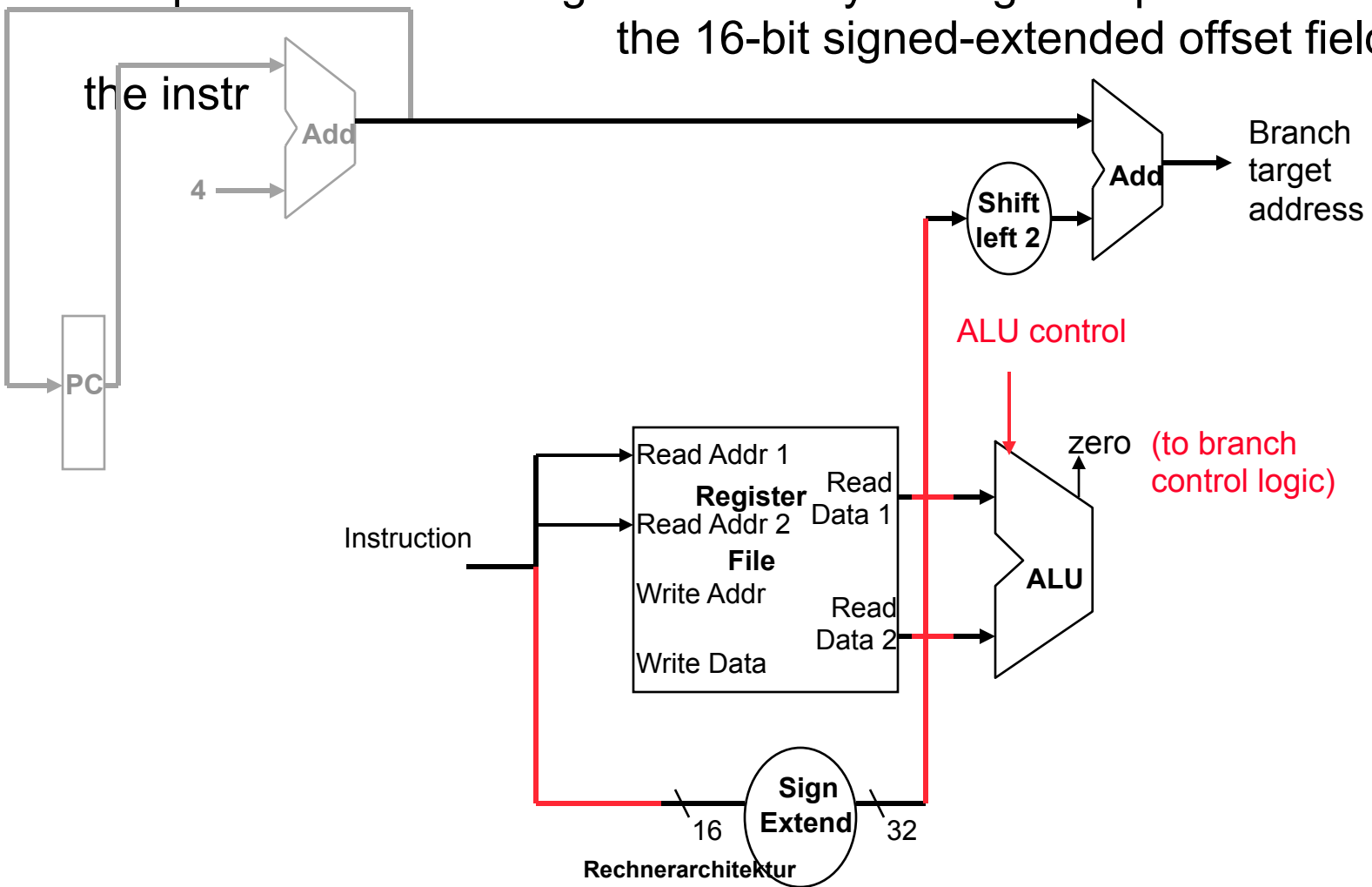
Executing Load and Store Operations

- 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



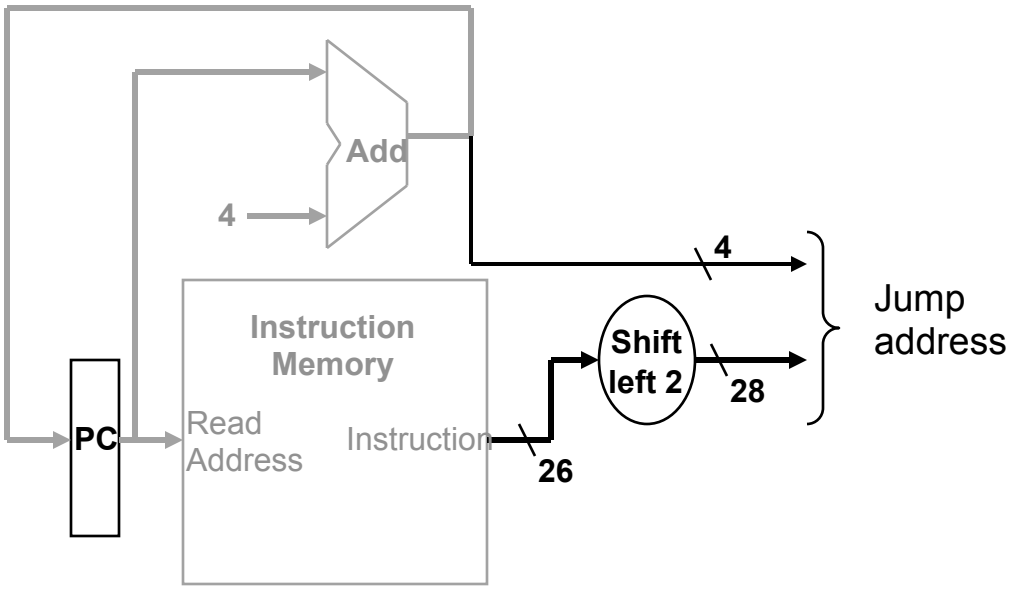
Executing Branch Operations

- Branch operations involves
  - compare the operands read from the Register File during decode for equality (`zero` ALU output)
  - compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instr



Executing Jump Operations

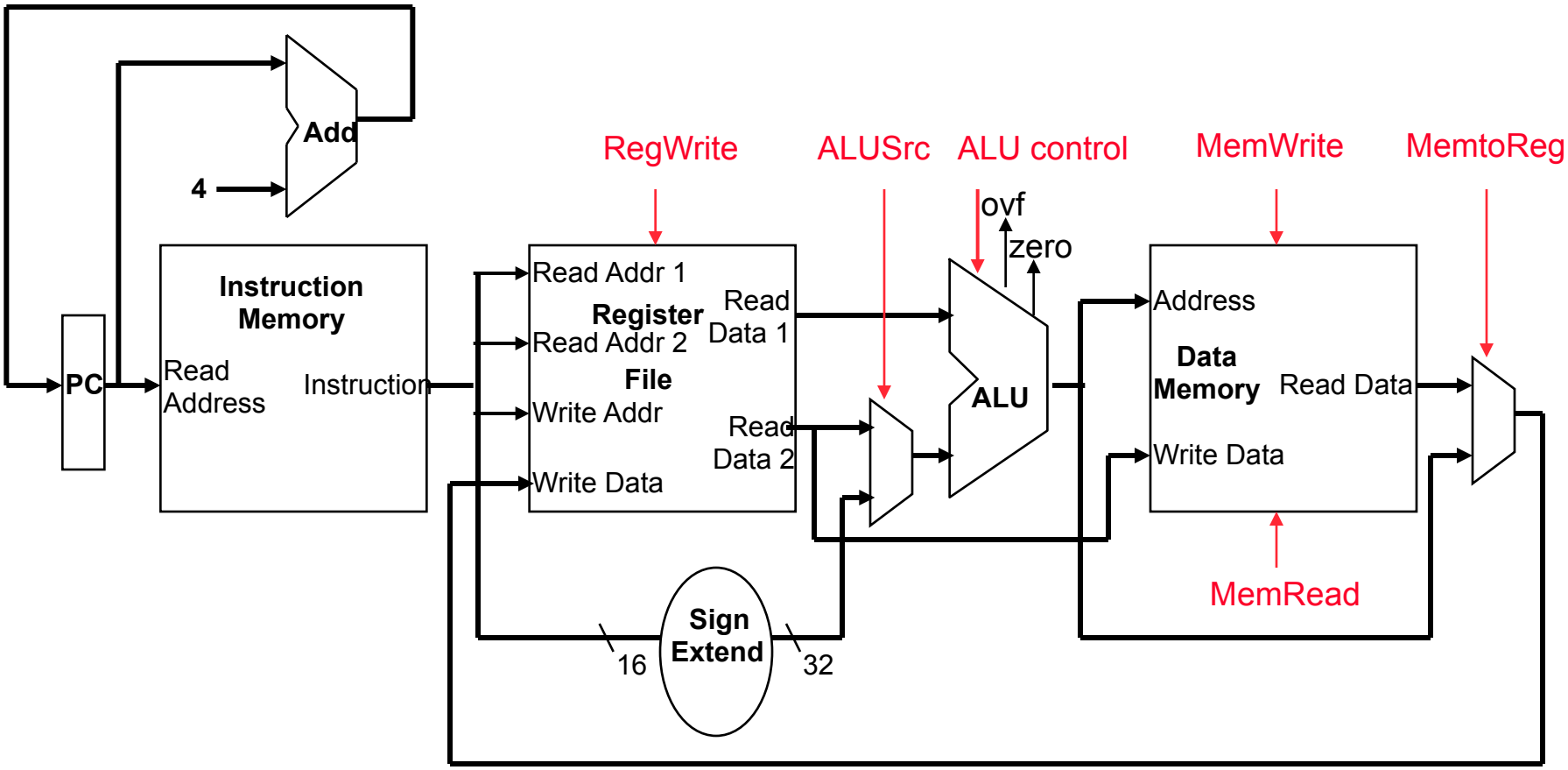
- Jump operation involves
  - replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits



Creating a Single Datapath from the Parts

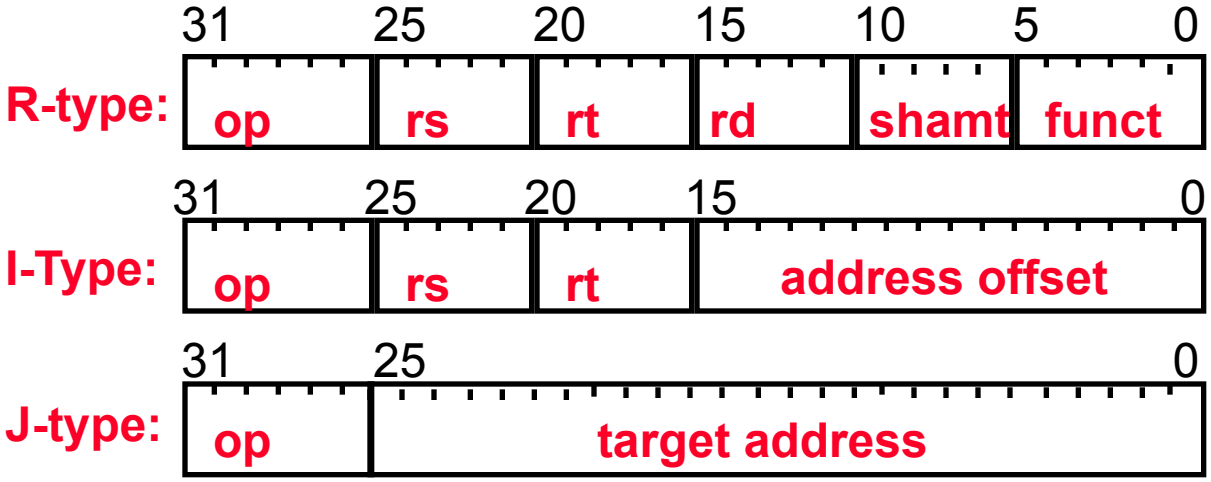
- Assemble the datapath segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instructions in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexors needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the Register File and Data Memory
- Cycle time is determined by length of the longest path

Fetch, R, and Memory Access Portions



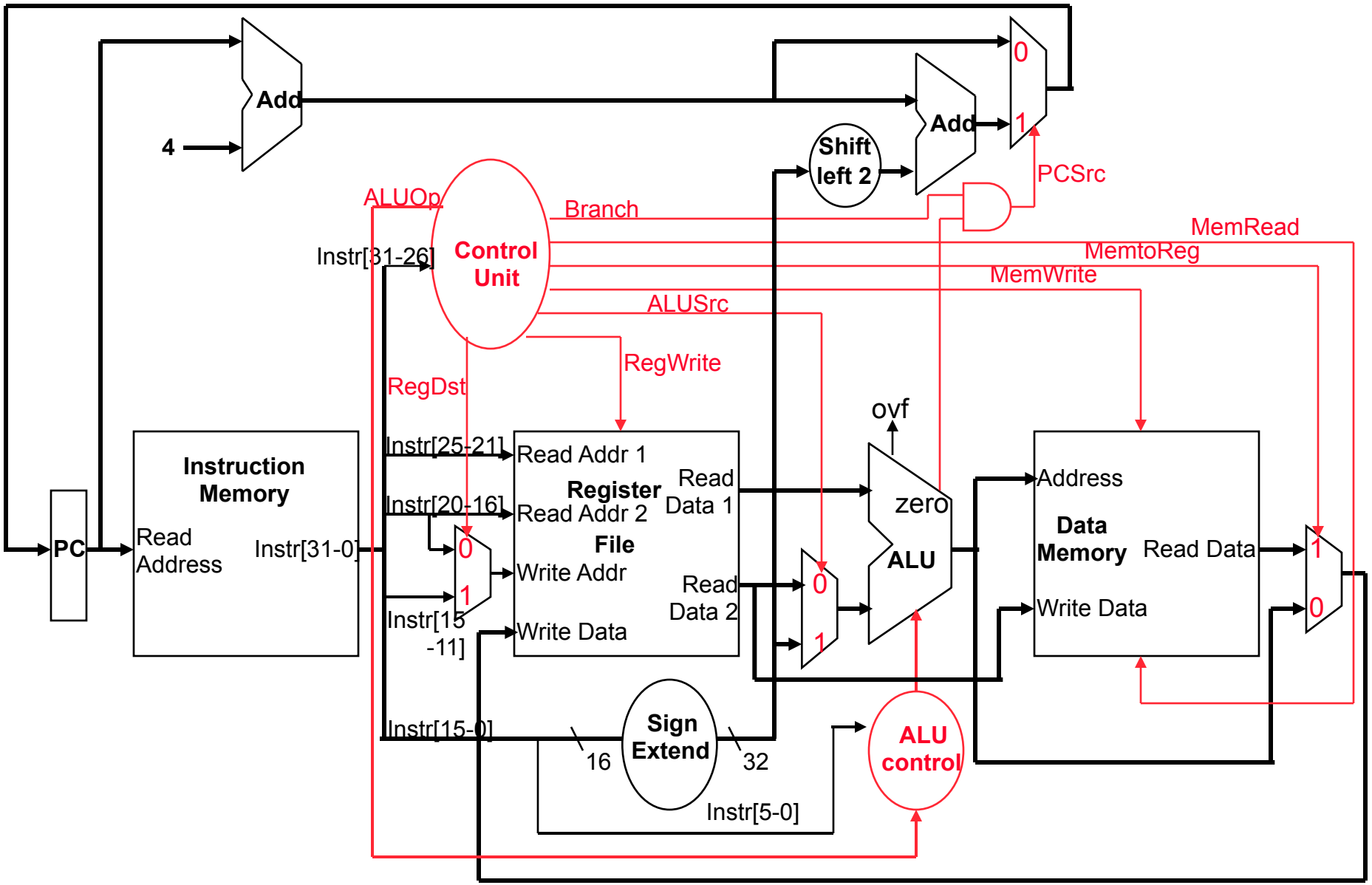
Adding the Control

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

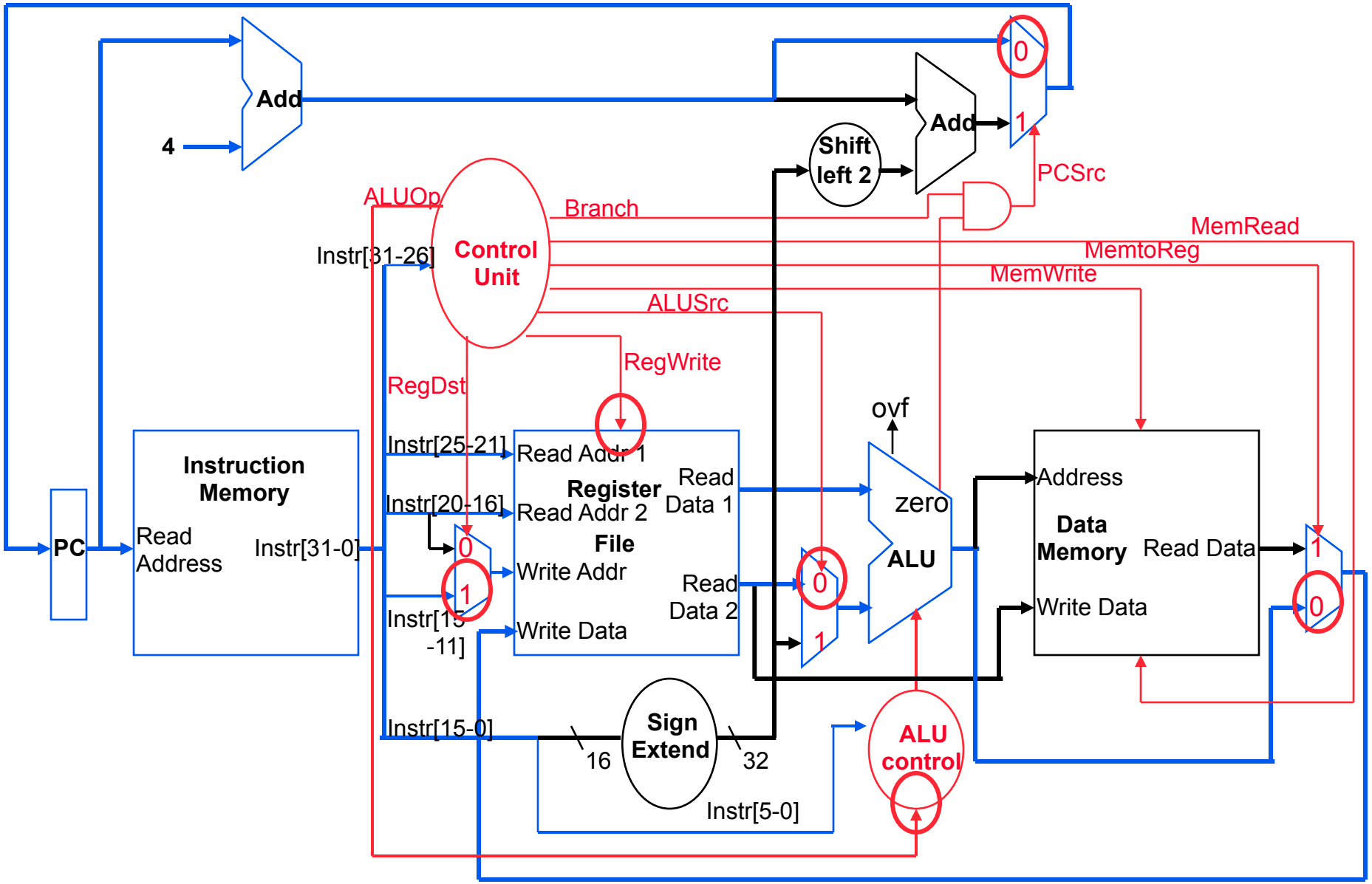


- Observations
  - op field always in bits 31-26
  - addr of registers to be read are always specified by the rs field (bits 25-21) and rt field (bits 20-16); for lw and sw rs is the base register
  - addr. of register to be written is in one of two places – in rt (bits 20-16) for lw; in rd (bits 15-11)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

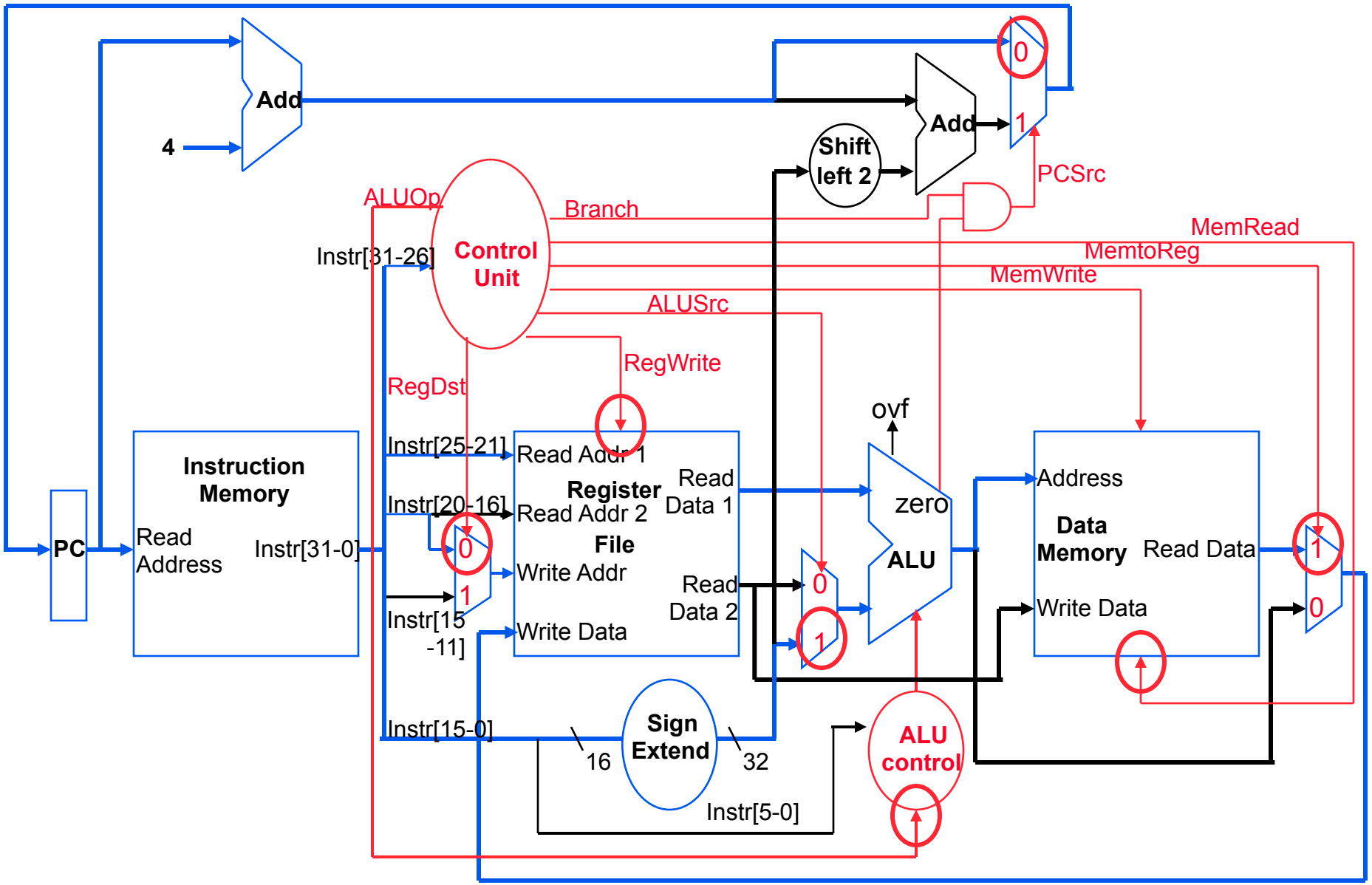
Single Cycle Datapath with Control Unit



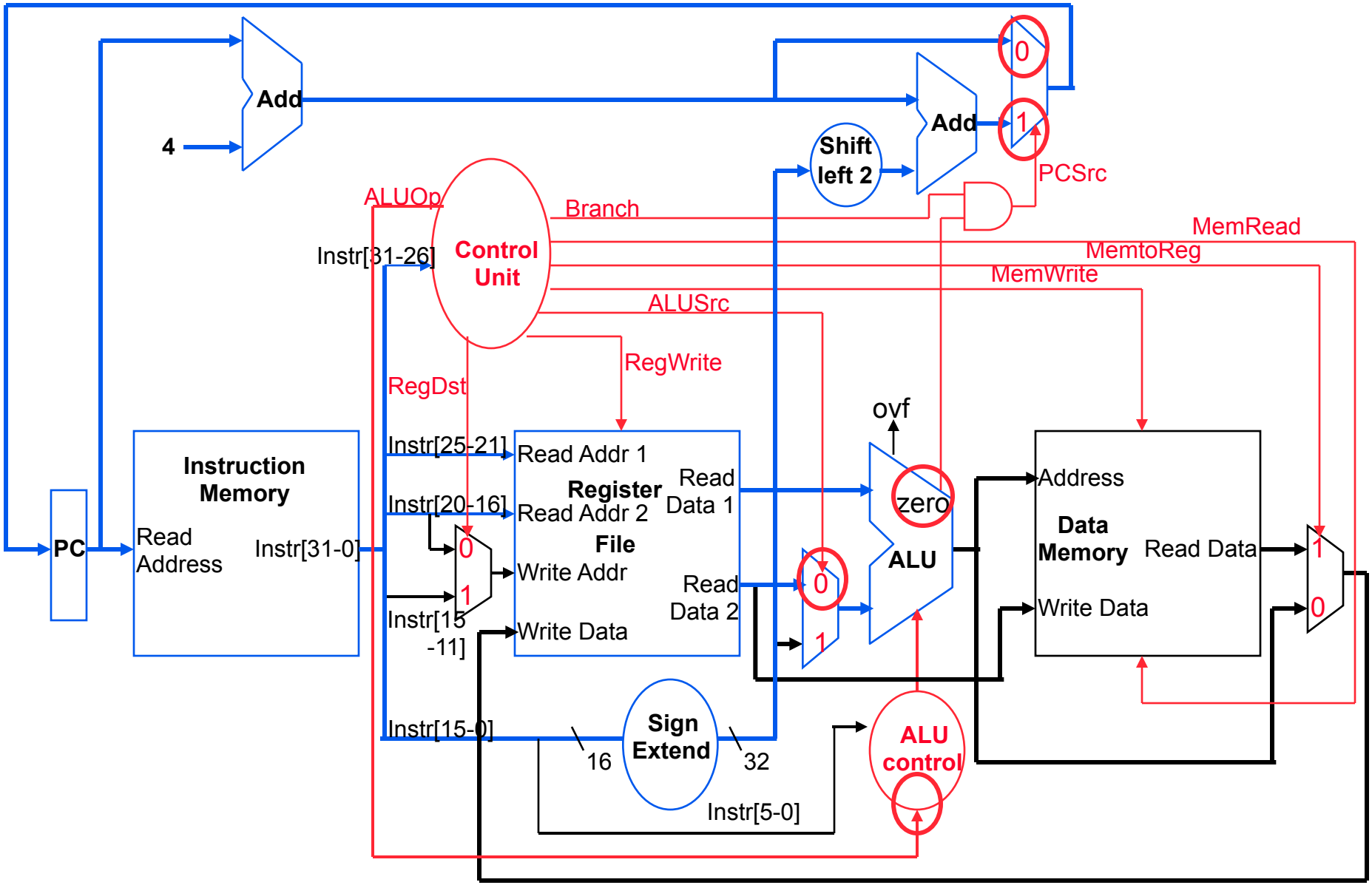
R-type Instruction Data/Control Flow



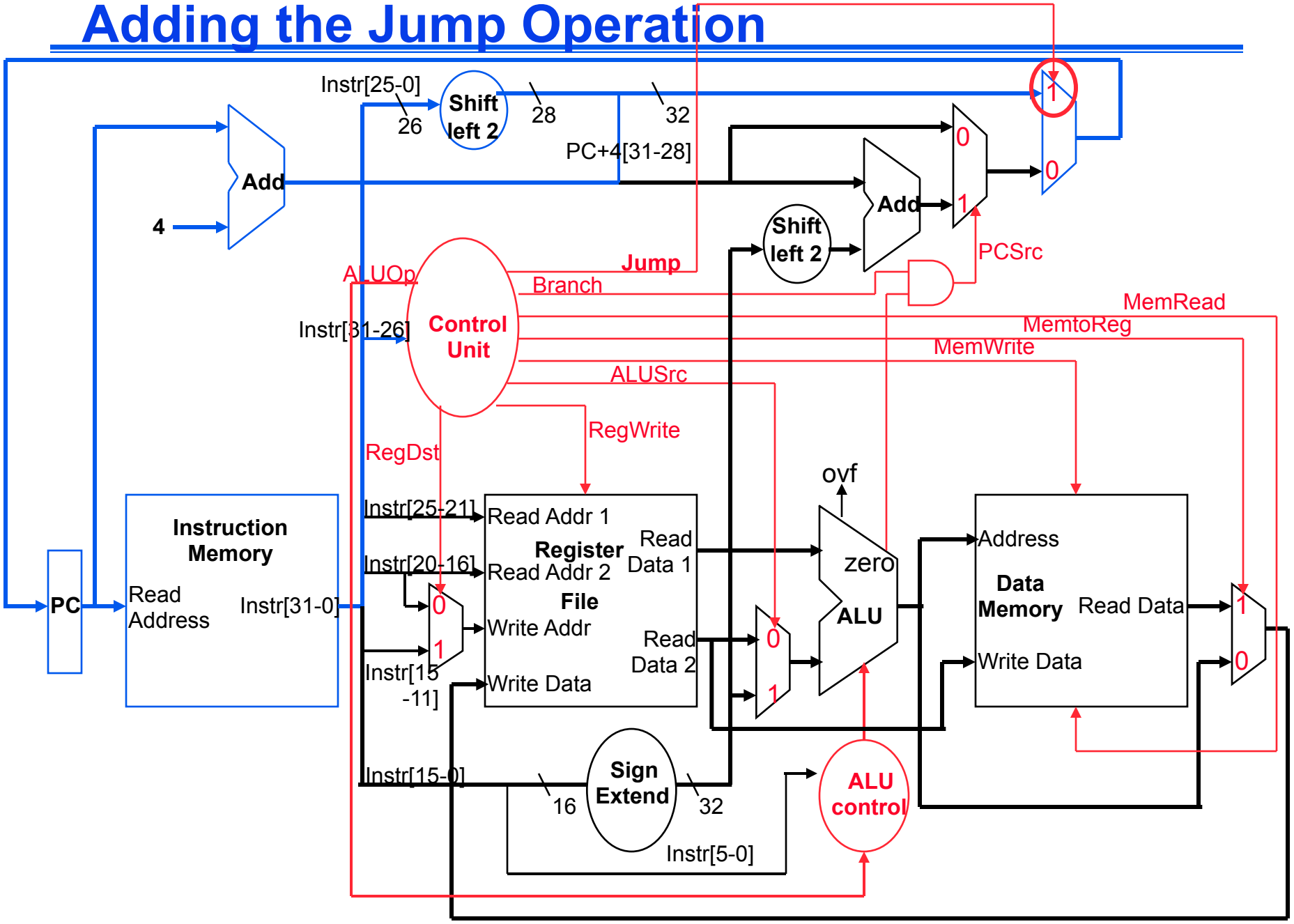
Load Word Instruction Data/Control Flow



Branch Instruction Data/Control Flow

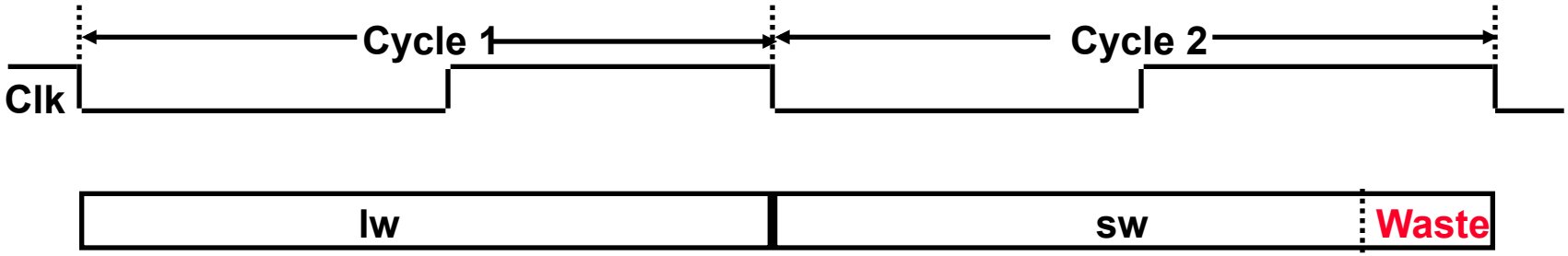


Adding the Jump Operation



Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point multiply



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

- Is simple and easy to understand

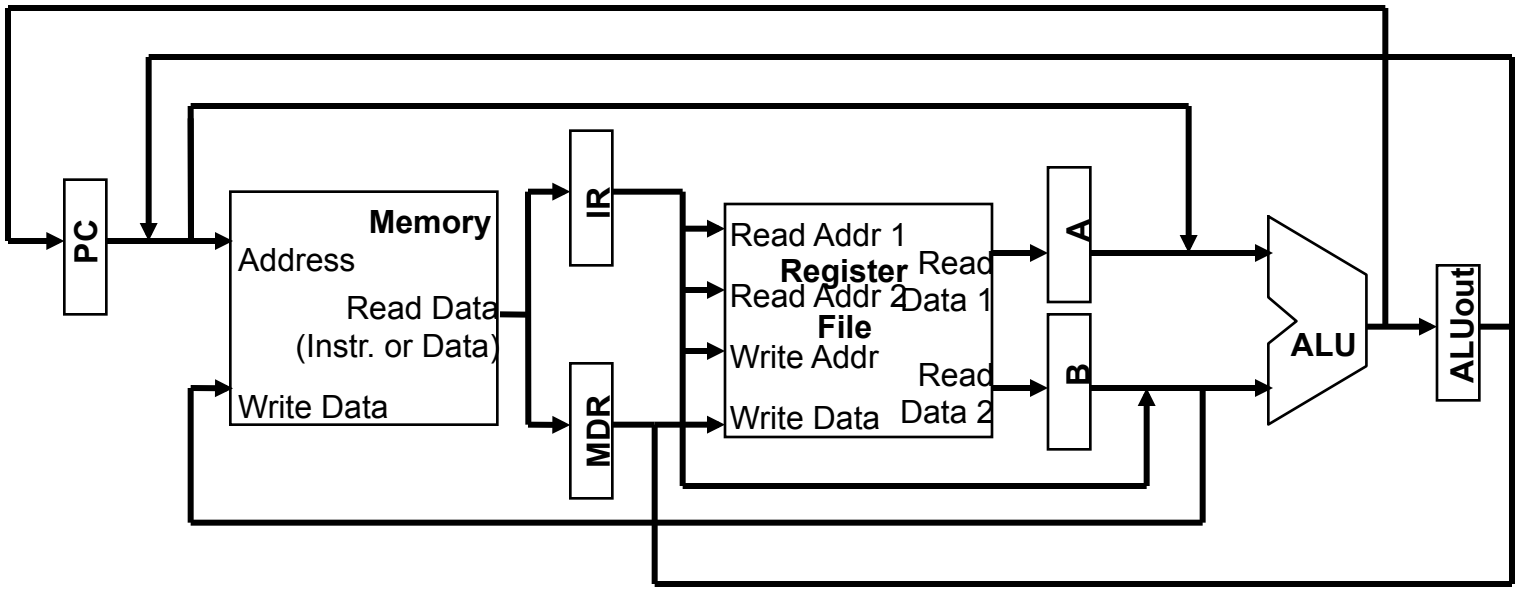
Multicycle Datapath Approach

- Let an instruction take more than 1 clock cycle to complete
  - Break up instructions into steps where each *step* takes a cycle while trying to
    - balance the amount of work to be done in each step
    - restrict each cycle to use only one major functional unit
  - Not every instruction takes the *same* number of clock cycles

- In addition to faster clock rates, multicycle allows functional units that can be used more than once per instruction as long as they are used on *different* clock cycles, as a result
  - only need one memory – but only one memory access per cycle
  - need only one ALU/adder – but only one ALU operation per cycle

Multicycle Datapath Approach, con't

- At the end of a cycle
  - Store values needed in a later cycle by the current instruction in an internal register (not visible to the programmer). All (except IR) hold data only between a pair of adjacent clock cycles (no write control signal needed)



IR – Instruction Register

MDR – Memory Data Register

A, B – regfile read data registers

ALUout – ALU output register

- Data used by subsequent instructions are stored in programmer visible registers (i.e., register file, PC, or memory)

Instructions from ISA perspective

- Consider each instruction from perspective of ISA.
- Example:
  - The add instruction changes a register.
  - Register specified by bits 15:11 of instruction.
  - Instruction specified by the PC.
  - New value is the sum ("op") of two registers.
  - Registers specified by bits 25:21 and 20:16 of the instruction

```
Reg[Memory[PC][15:11]] <=  Reg[Memory[PC][25:21]]
op                                 Reg[Memory[PC]
[20:16]]
```

  - In order to accomplish this we must break up the instruction. (kind of like introducing variables when programming)

Breaking down an instruction

- ISA definition of arithmetic:

```
Reg[Memory[PC][15:11]] <= Reg[Memory[PC]
[25:21]]  op
                          Reg[Memory[PC][20:16]]
```

- Could break down to:
  - `IR <= Memory[PC]`
  - `A <= Reg[IR[25:21]]`
  - `B <= Reg[IR[20:16]]`
  - `ALUOut <= A op B`
  - `Reg[IR[15:11]] <= ALUOut`

- We forgot an important part of the definition of arithmetic!
  - `PC <= PC + 4`

Idea behind multicycle approach

- We define each instruction from the ISA perspective (do this!)
- Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)
- Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)
- Finally try and pack as much work into each step
  (avoid unnecessary cycles)
  while also trying to share steps where possible
  (minimizes control, helps to simplify solution)

Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR <= Memory[PC];
PC <= PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A <= Reg[IR[25:21]];

B <= Reg[IR[20:16]];

ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
```

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type

- Memory Reference:

```
ALUOut <= A + sign-extend(IR[15:0]);
```

- R-type:

```
ALUOut <= A op B;
```

- Branch:

```
if (A==B) PC <= ALUOut;
```

Step 4 (R-type or memory-access)

- Loads and stores access memory

```
MDR <= Memory[ALUOut];
      or
Memory[ALUOut] <= B;
```

- R-type instructions finish

```
Reg[IR[15:11]] <= ALUOut;
```

Write-back step

- `Reg[IR[20:16]] <= MDR;`

Only Load instruction needs this cycle.

Summary:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]),2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

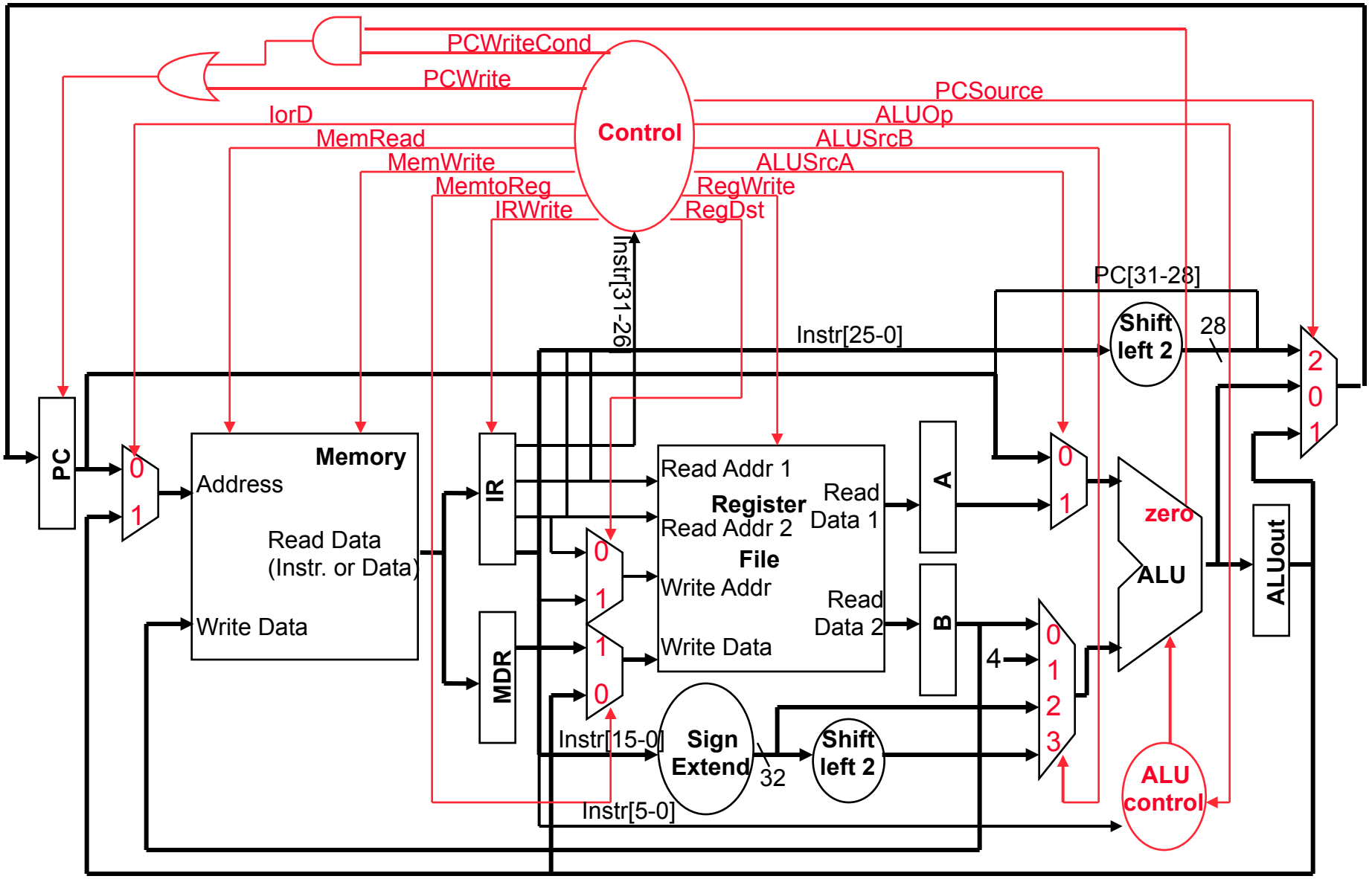
Simple Questions

- How many cycles will it take to execute this code?

```
        lw $t2, 0($t3)
        lw $t3, 4($t3)
        beq $t2, $t3, Label #assume not
        add $t5, $t2, $t3
        sw $t5, 8($t3)
Label:  ...
```

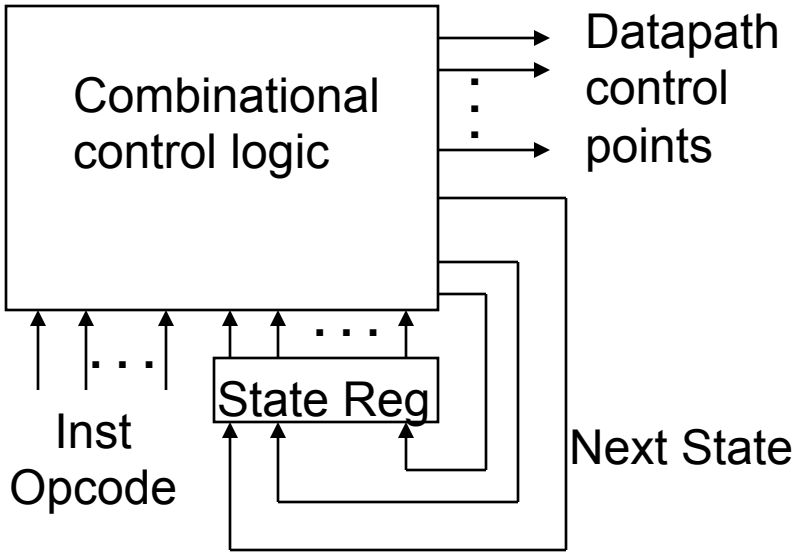
- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of `$t2` and `$t3` takes place?

The Multicycle Datapath with Control Signals

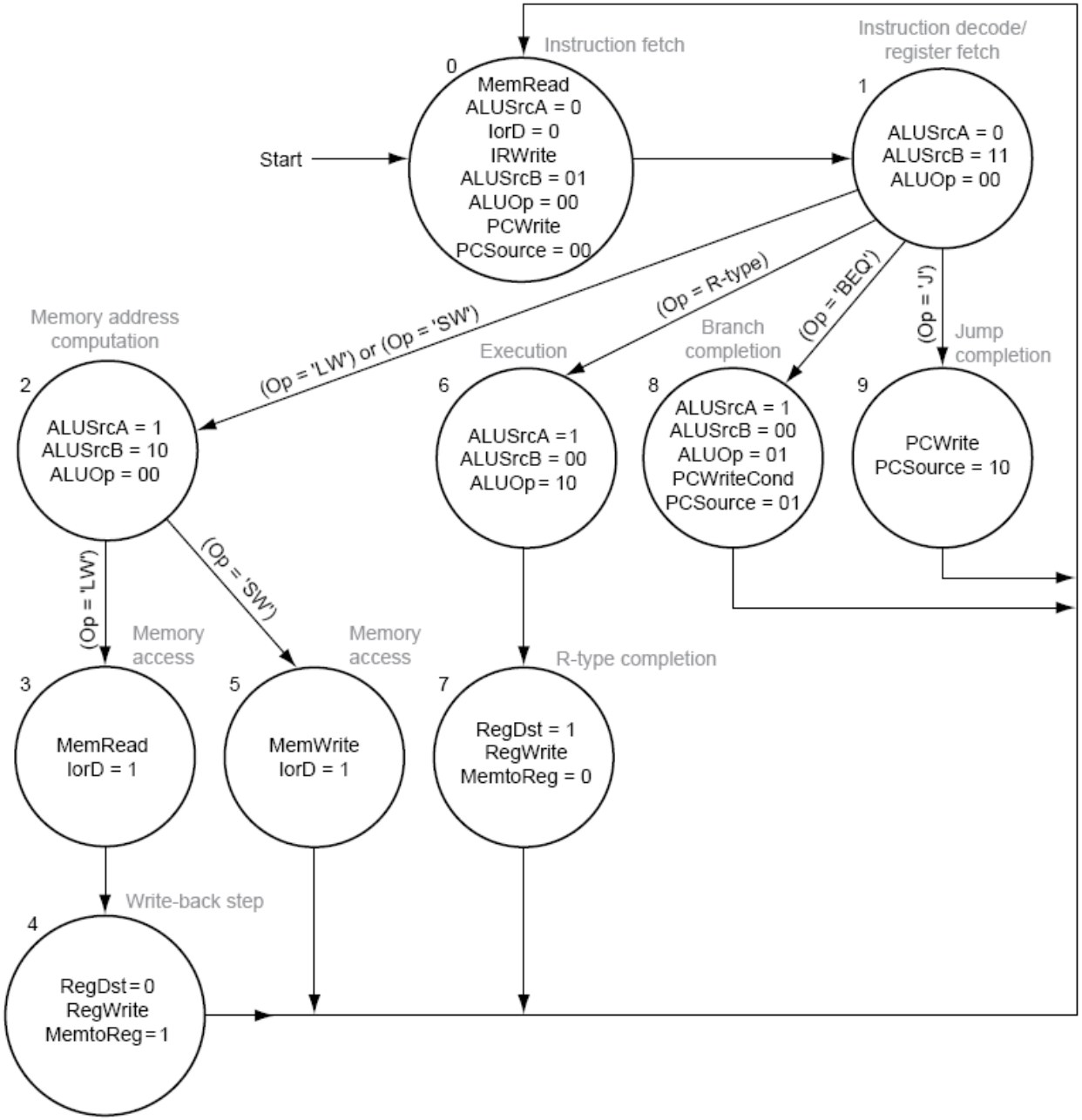


Multicycle Control Unit

- Multicycle datapath control signals are not determined solely by the bits in the instruction
  - e.g., op code bits tell what operation the ALU should be doing, but *not* what instruction cycle is to be done next
- Must use a finite state machine (FSM) for control
  - a set of states (current state stored in State Register)
  - next state function (determined by current state and the input)
  - output function (determined by current state and the input)

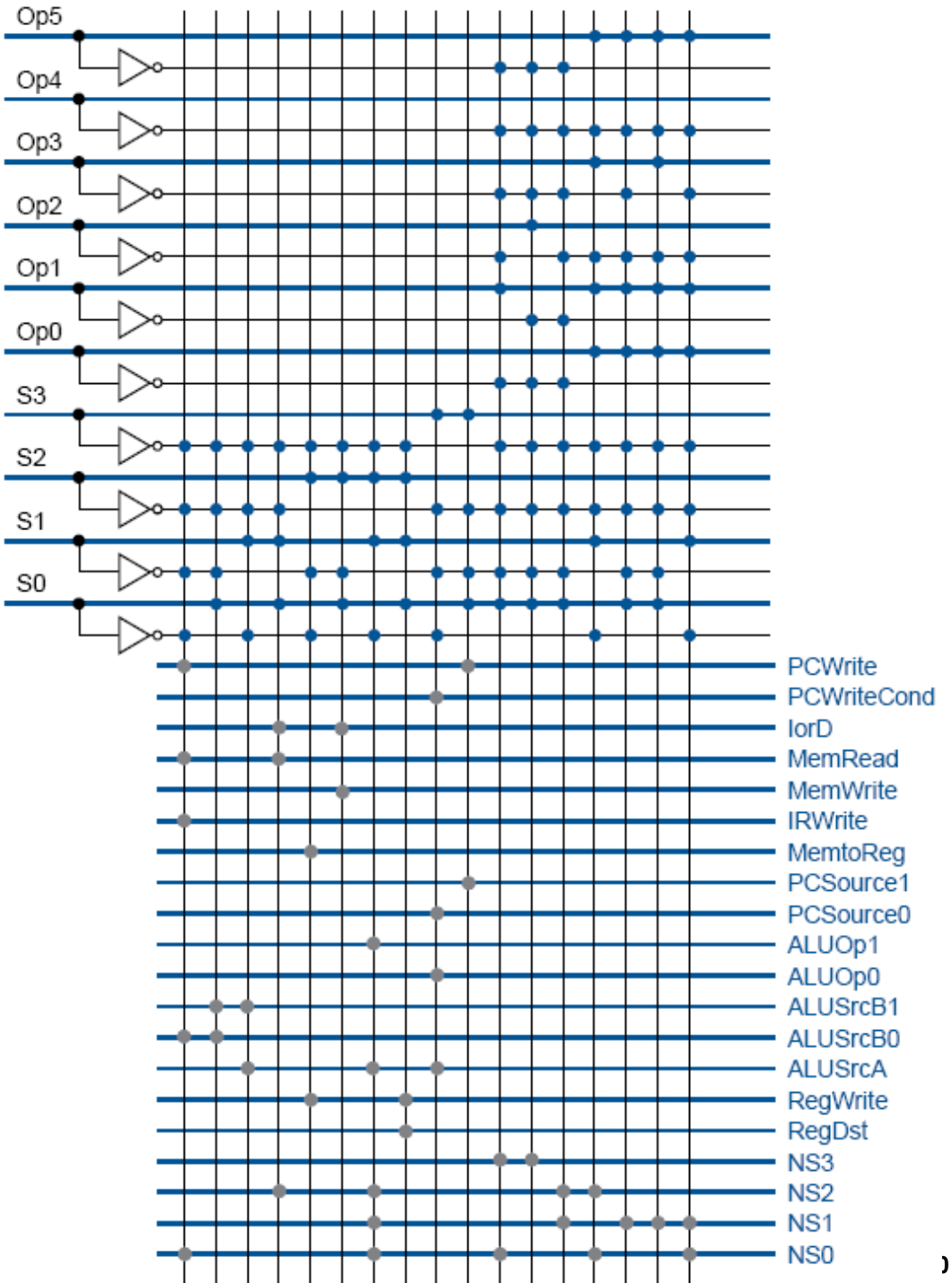


Multicycle Control Unit: Finite State Machine

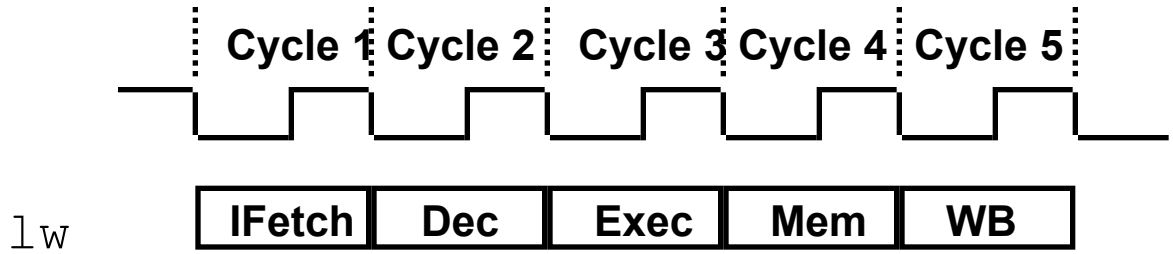


Control Unit: PLA

PLA implementation of the finite state machine for the multicycle control unit



The Five Steps of the Load Instruction

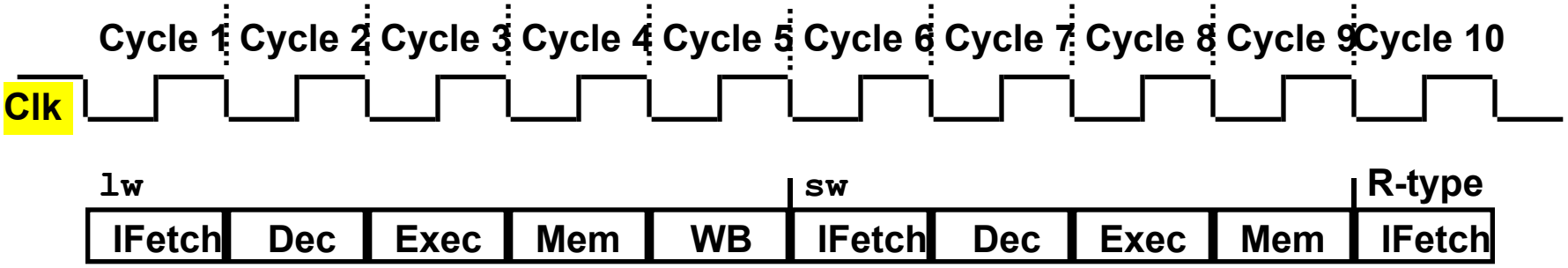


- IFetch: Instruction Fetch and Update PC
- Dec: Instruction Decode, Register Read, Sign Extend Offset
- Exec: Execute R-type; Calculate Memory Address; Branch Comparison; Branch and Jump Completion
- Mem: Memory Read; Memory Write Completion; R-type Completion (RegFile write)
- WB: Memory Read Completion (RegFile write)

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Multicycle Advantages & Disadvantages

- Uses the clock cycle efficiently – the clock cycle is timed to accommodate the slowest instruction step



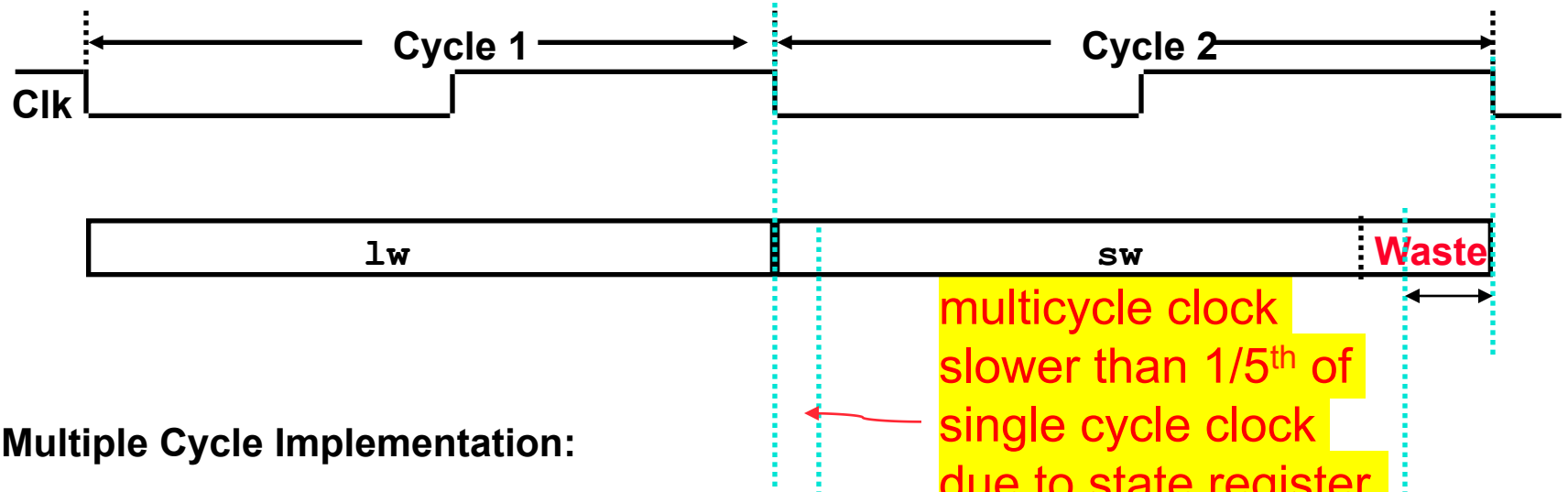
- Multicycle implementations allow functional units to be used more than once per instruction as long as they are used on different clock cycles

but

- Requires additional internal state registers, more muxes, and more complicated (FSM) control

Single Cycle vs. Multiple Cycle Timing

Single Cycle Implementation:



multicycle clock slower than 1/5th of single cycle clock due to state register overhead

Multiple Cycle Implementation: