Computer Organization and Design Processor

MIPS single cycle processor

[adapted from Mary Jane Irwin slides]

Reading assignment

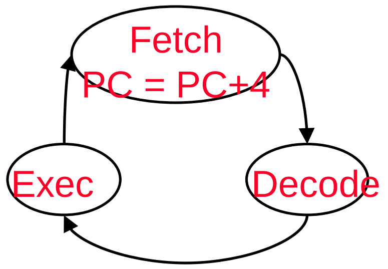
- Designing a MIPS single cycle processor
  - PH(4) 4.1-4.4
  - Appendix C The Basics of Logic Design C.1-C.3

Review: Design Principles

- Simplicity favors regularity
  - fixed size instructions – 32-bits
  - only three instruction formats
- Good design demands good compromises
  - three instruction formats
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands

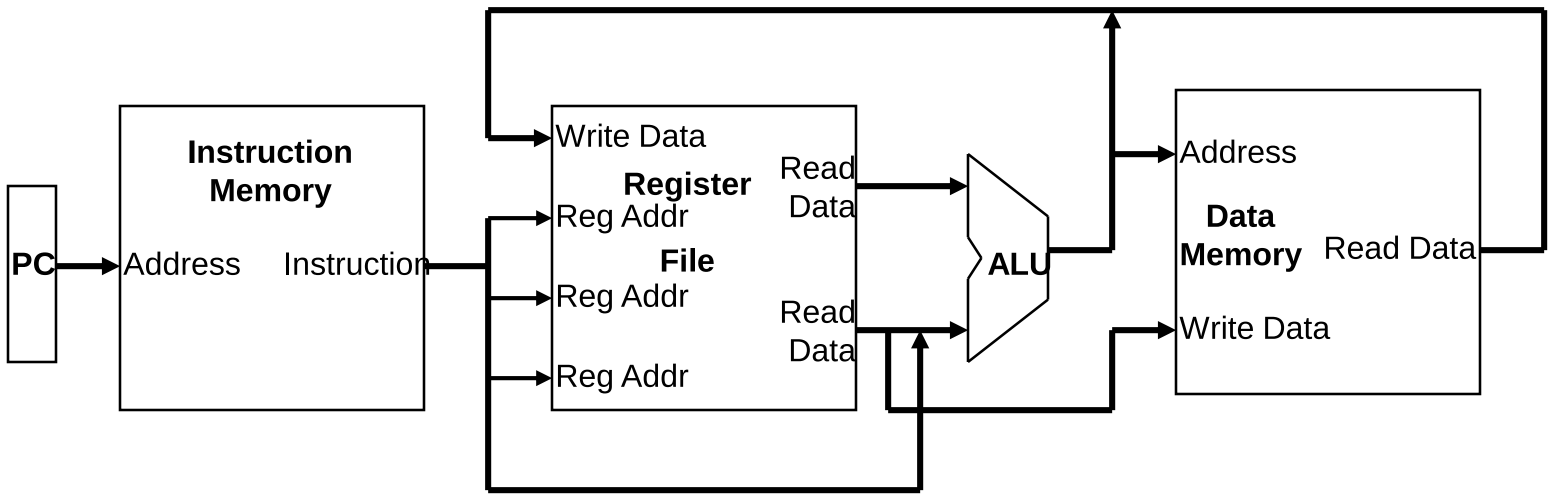
The Processor: Datapath & Control

- We're ready to look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, addu, sub, subu, and, or, xor, nor, slt, sltu`
  - arithmetic-logical immediate instructions: `addi, addiu, andi, ori, xori, slti, sltiu`
  - 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



Abstract Implementation View

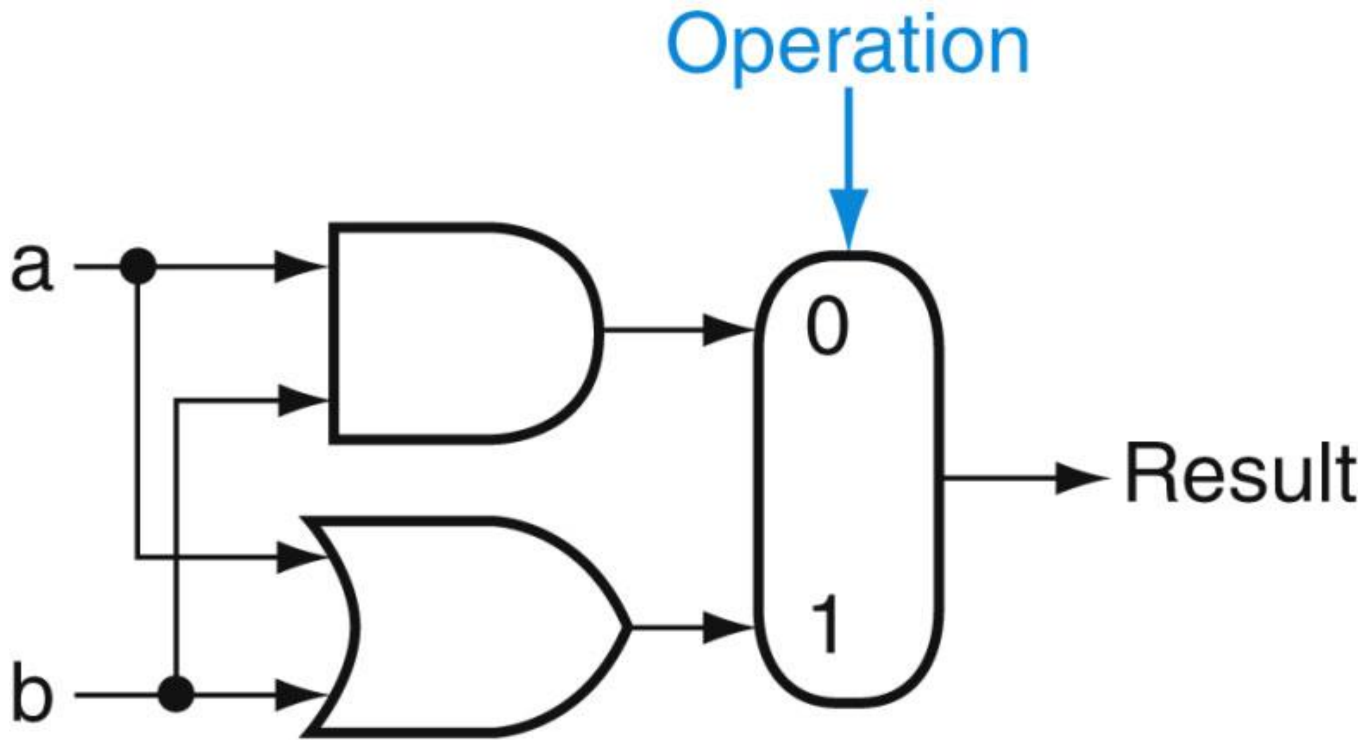
- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)



- Single cycle operation
- Split memory (Harvard) model - one memory for instructions and one for data

ALU

- AND和OR的1位逻辑单元。
- 根据Operation的值，多路选择器选择（a AND b）或（a OR b）。

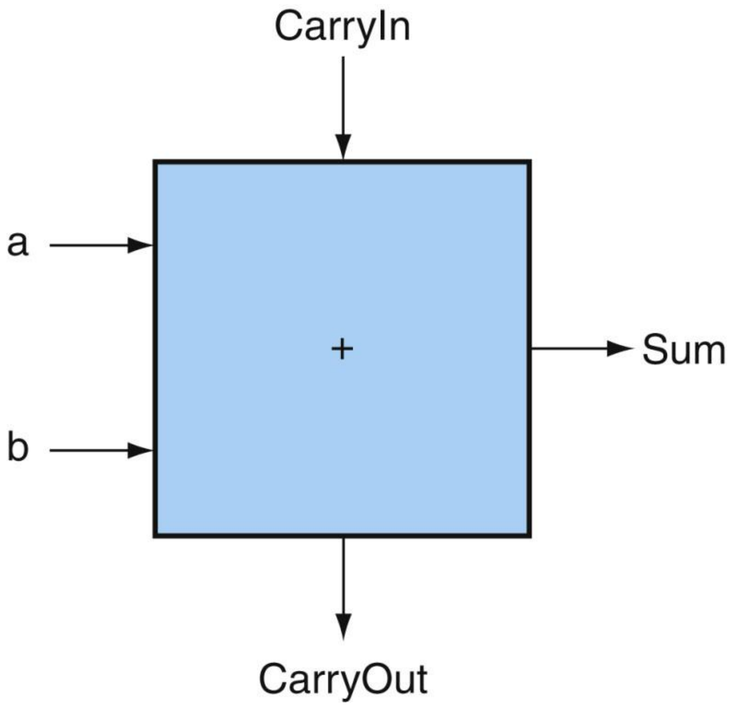


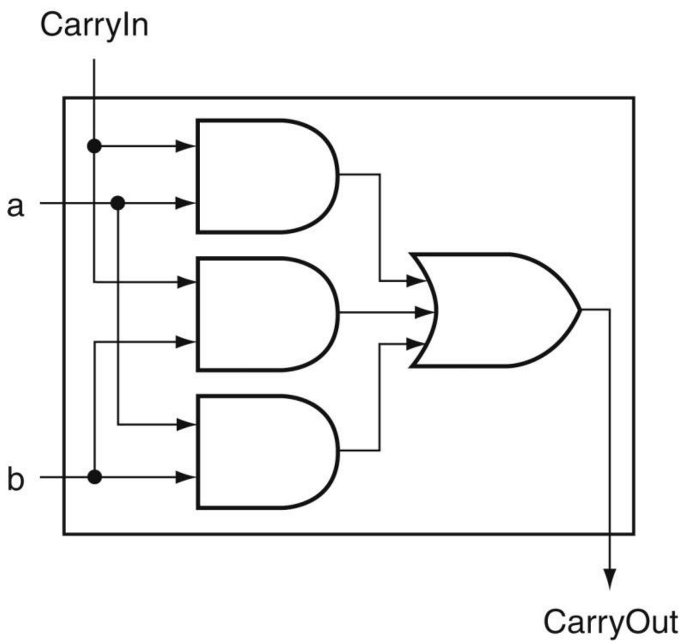
ALU

- 1位加法器，全加器，（3，2）加法器

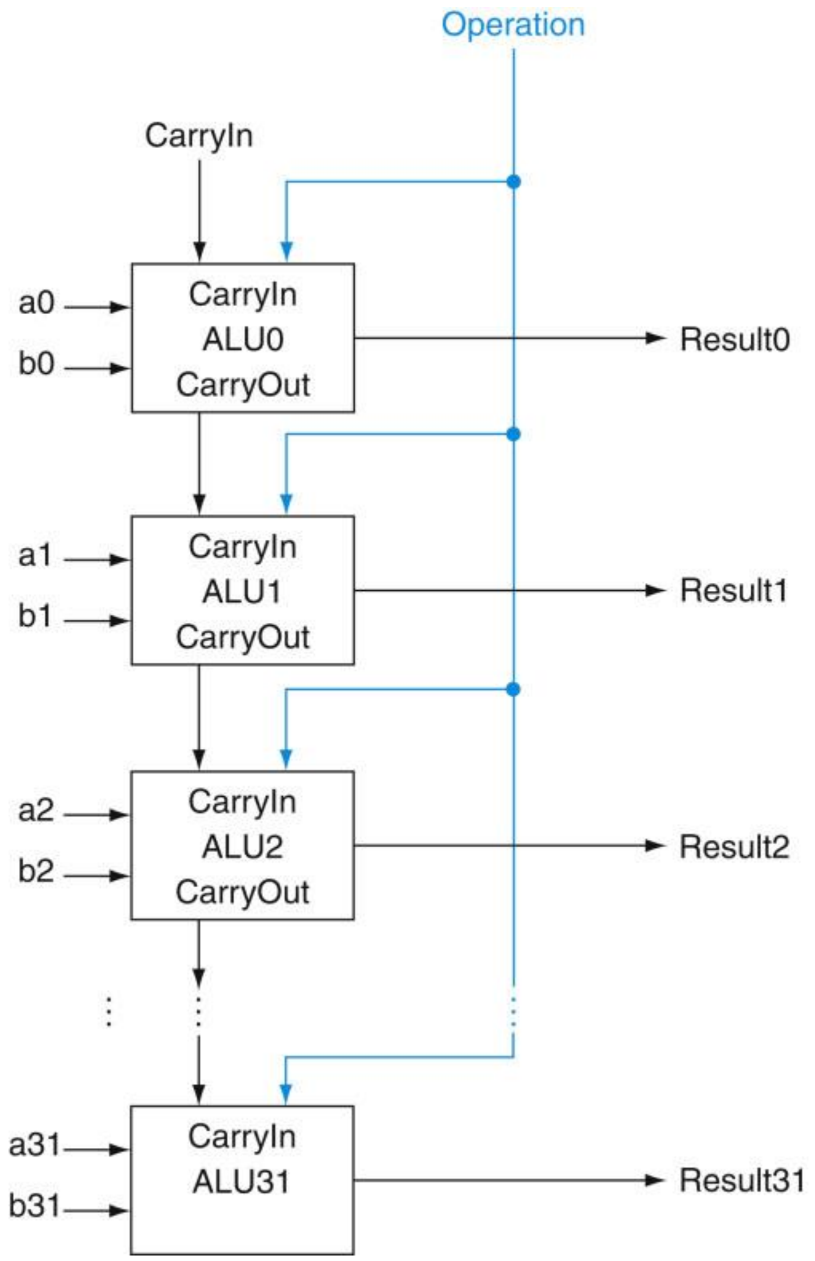
$$CarryOut = (b \times CarryIn) + (a \times CarryIn) + (a \times b)$$

$$Sum = (a \times \overline{b} \times \overline{CarryIn}) + (\overline{a} \times b \times \overline{CarryIn}) + (\overline{a} \times \overline{b} \times CarryIn) + (a \times b \times CarryIn)$$

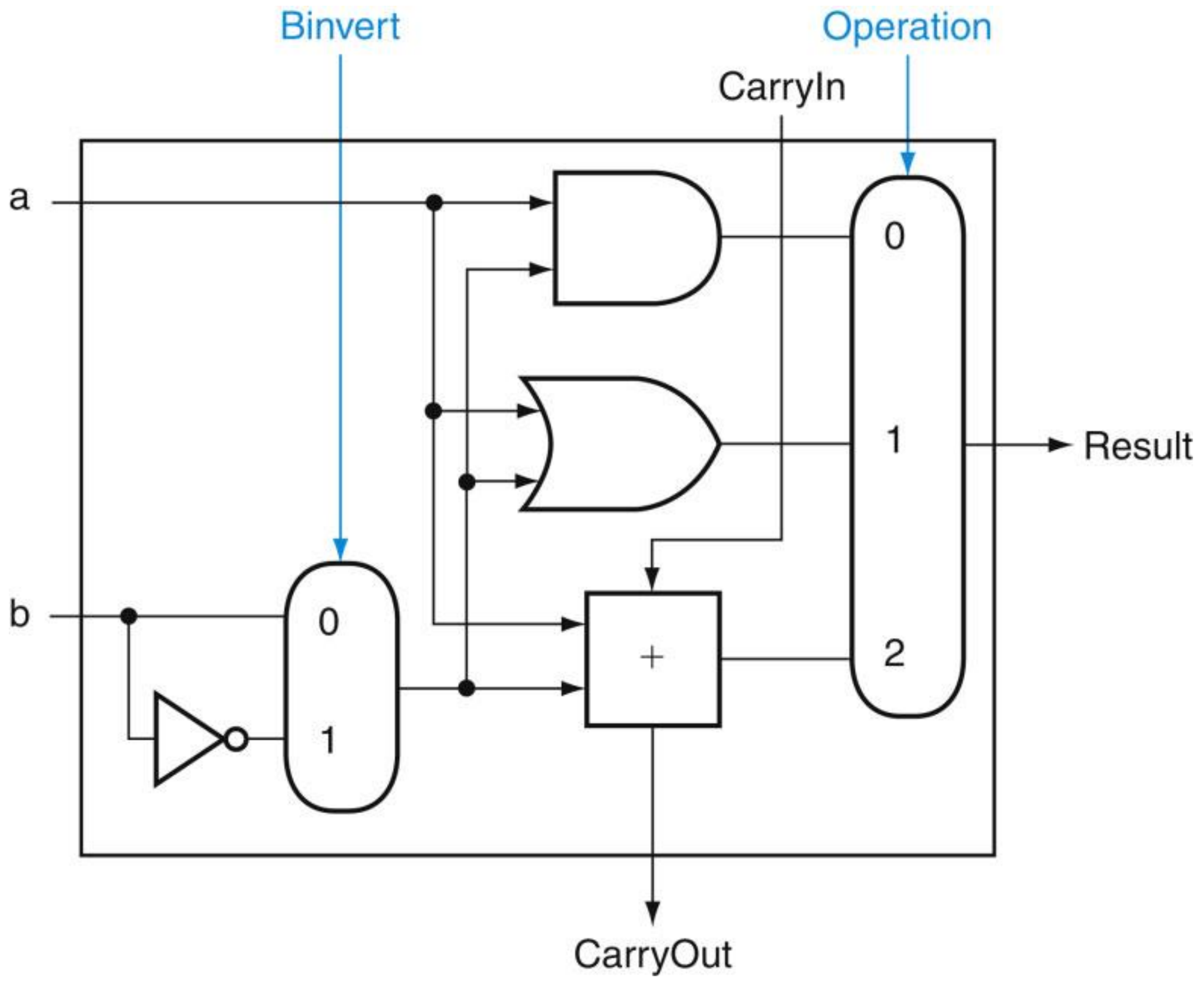




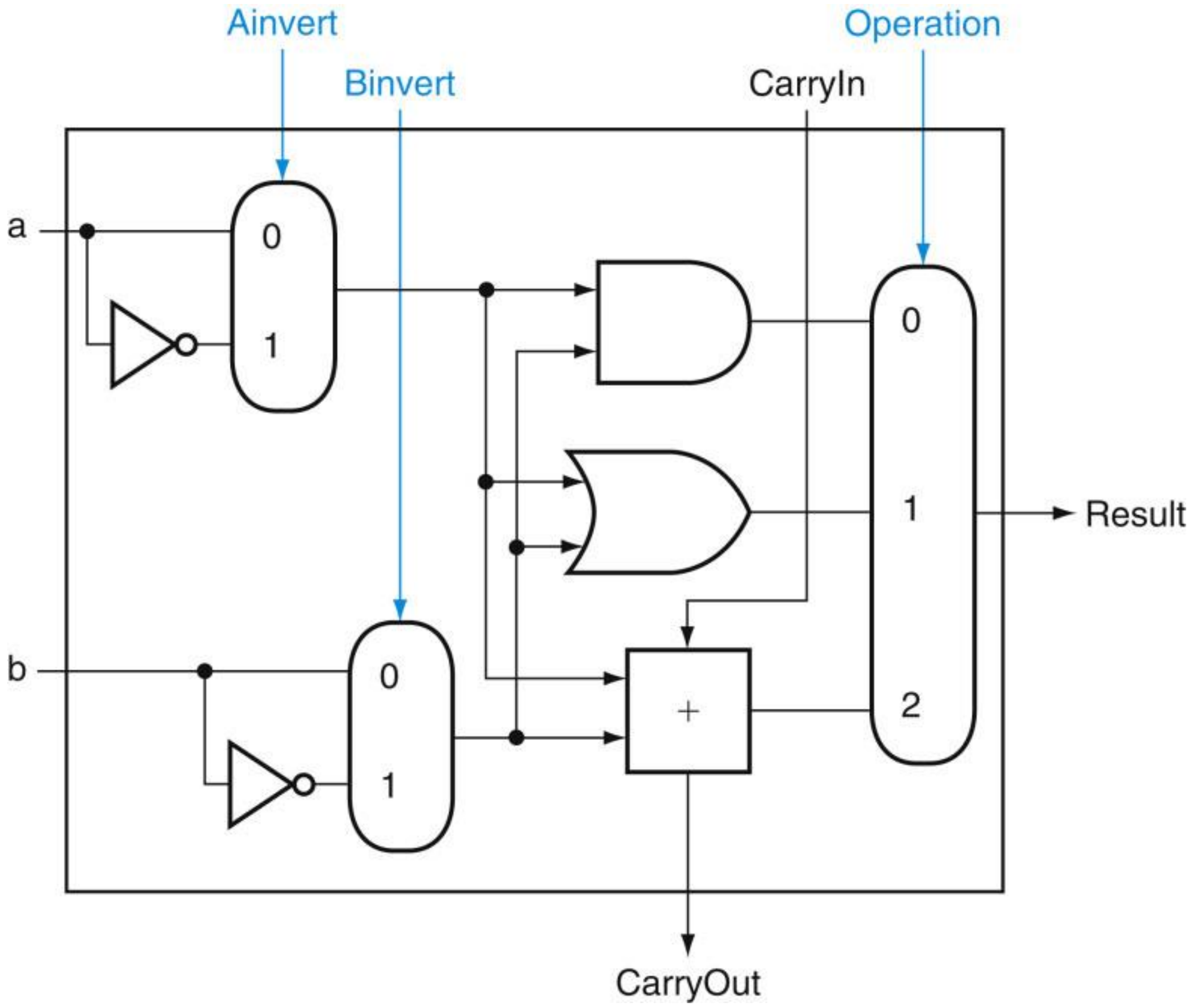
32位 ALU


Operation
CarryIn
a0
b0
CarryIn
ALU0
CarryOut
Result0
a1
b1
CarryIn
ALU1
CarryOut
Result1
a2
b2
CarryIn
ALU2
CarryOut
Result2
a31
b31
CarryIn
ALU31
Result31

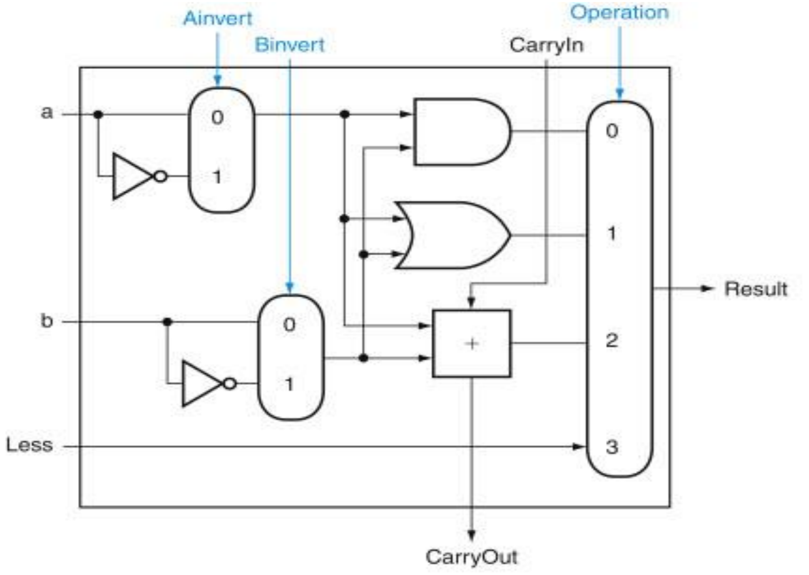
执行AND, OR, a+b, or a+$\overline{b}$的1位ALU

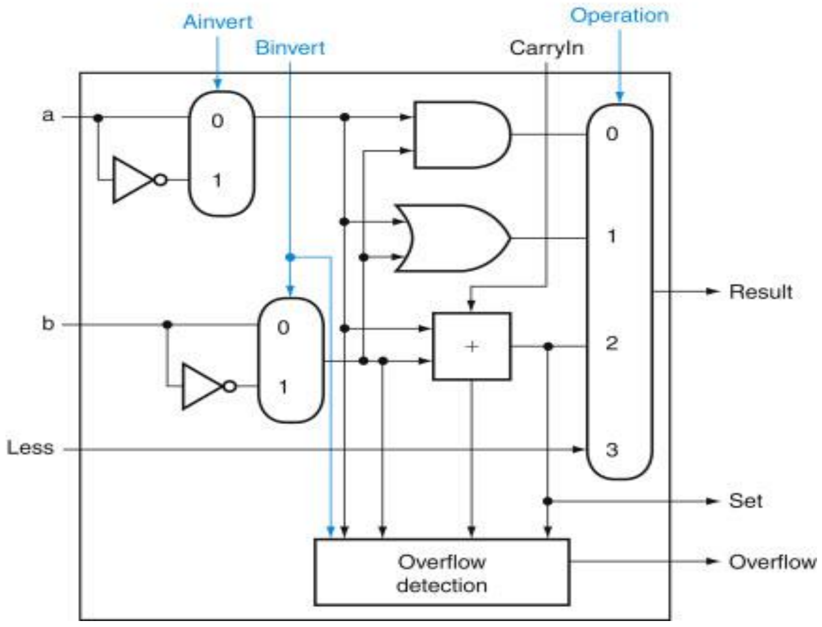


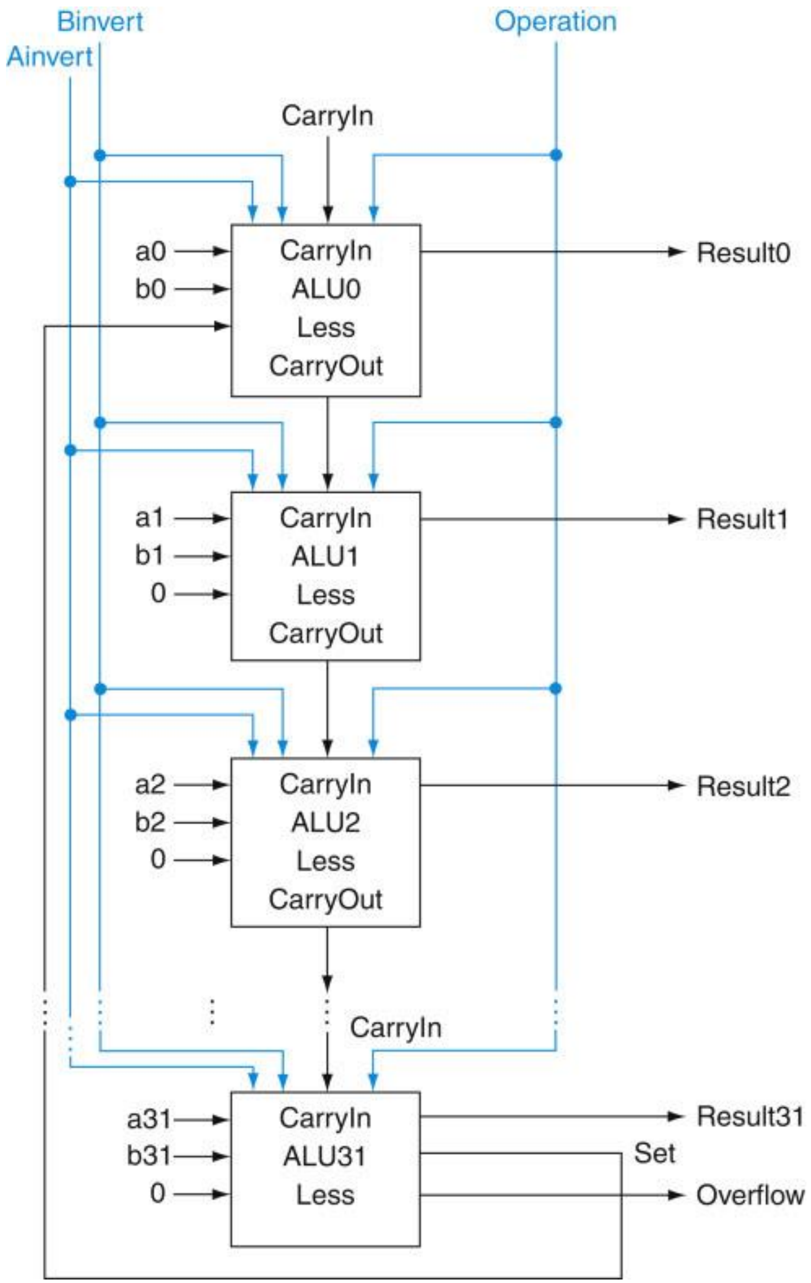
执行NOR的1位ALU



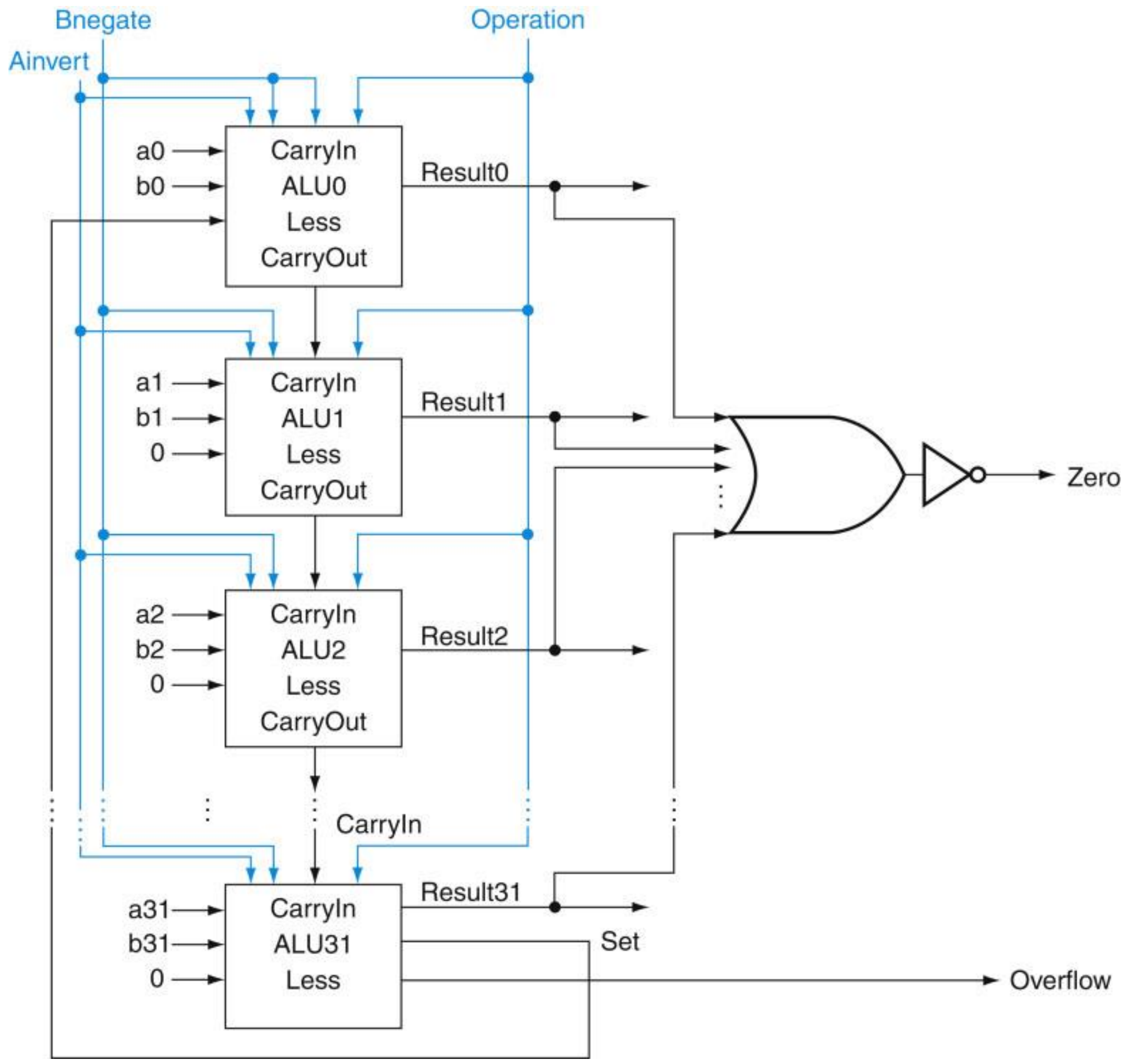
执行SLT的ALU


Ainvert
Binvert
Operation
CarryIn
a
b
Less
0
1
0
1
0
1
2
3
+
Result
CarryOut

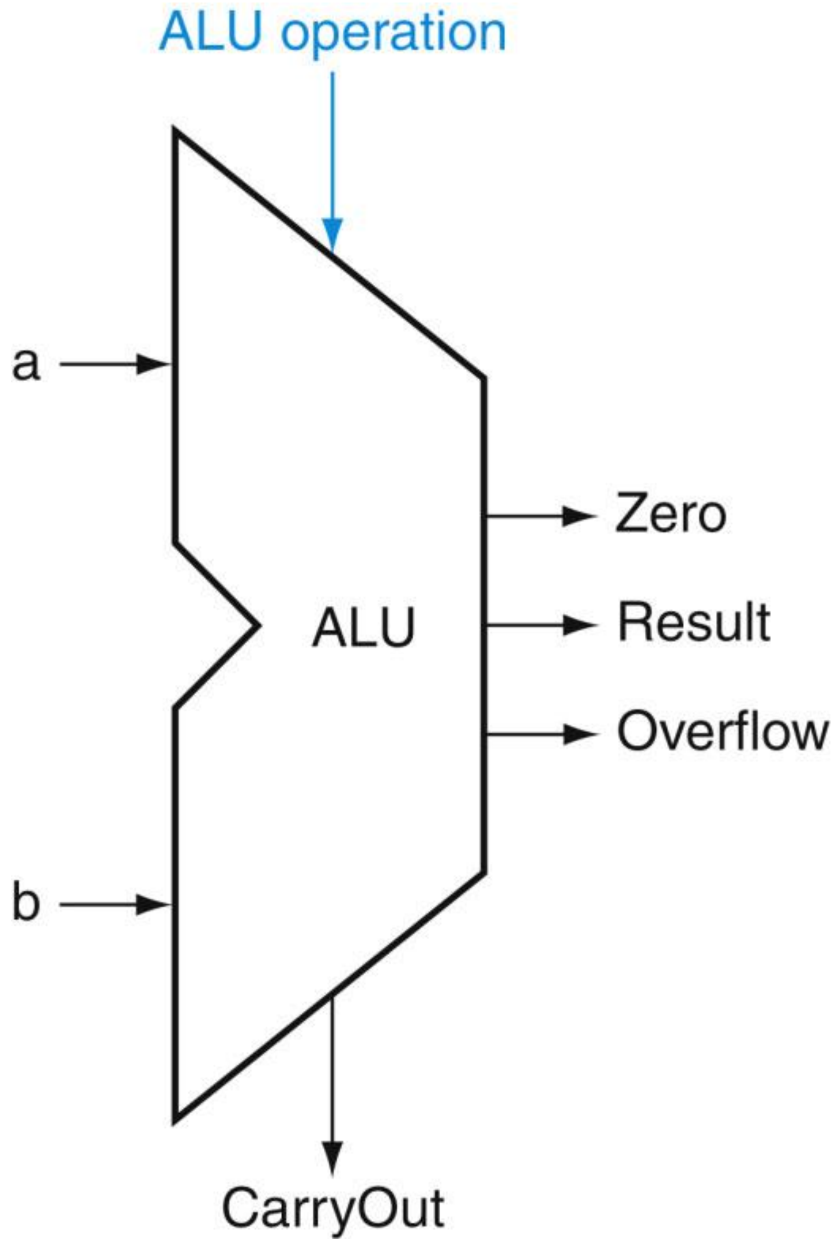

Ainvert
Binvert
Operation
CarryIn
a
b
Less
0
1
0
1
0
1
2
3
+
Result
Set
Overflow detection
Overflow


Binvert
Ainvert
Operation
CarryIn
a0
b0
CarryIn
ALU0
Less
CarryOut
Result0
a1
b1
0
CarryIn
ALU1
Less
CarryOut
Result1
a2
b2
0
CarryIn
ALU2
Less
CarryOut
Result2
CarryIn
a31
b31
0
CarryIn
ALU31
Less
Result31
Set
Overflow

执行BEQ的ALU

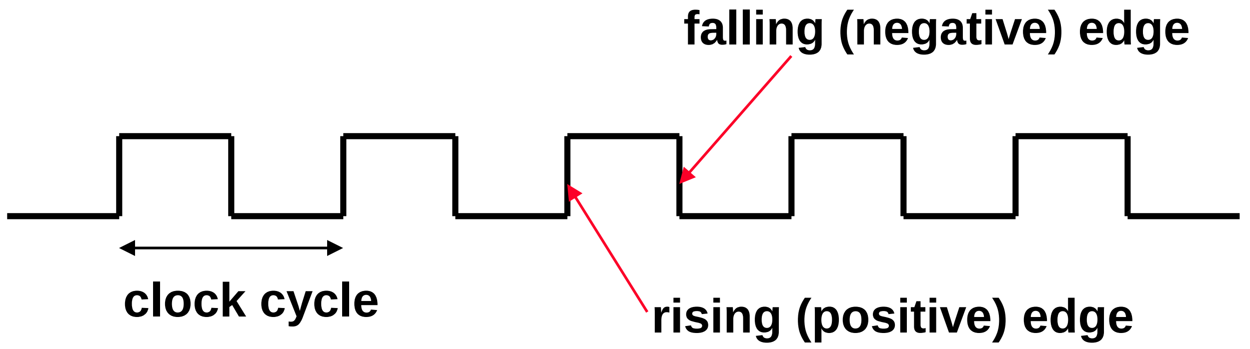

Bnegate
Operation
Ainvert
a0
b0
CarryIn
ALU0
Less
CarryOut
Result0
a1
b1
0
CarryIn
ALU1
Less
CarryOut
Result1
Zero
a2
b2
0
CarryIn
ALU2
Less
CarryOut
Result2
CarryIn
a31
b31
0
CarryIn
ALU31
Less
Result31
Set
Overflow

表示ALU的符号



Clocking Methodologies

- Clocking methodology defines when signals can be read and when they can be written

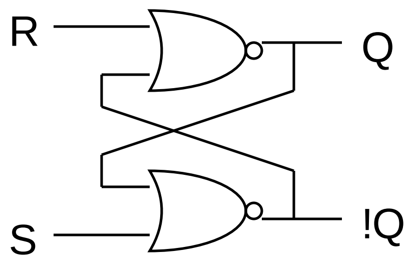


clock rate = 1/(clock cycle)
e.g., 10 nsec clock cycle = 100 MHz clock rate
1 nsec clock cycle = 1 GHz clock rate

- State element design choices
  - level sensitive latch
  - master-slave and edge-triggered flipflops

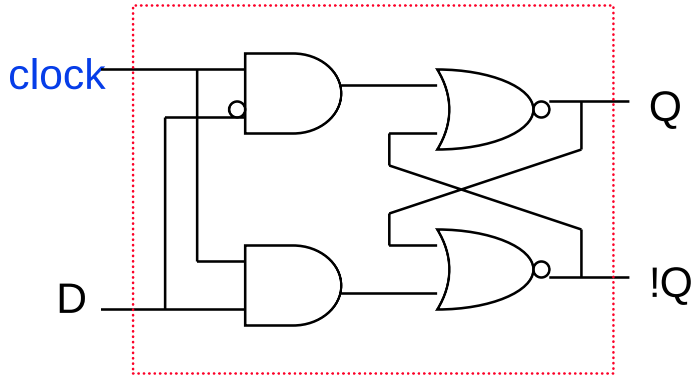
State Elements

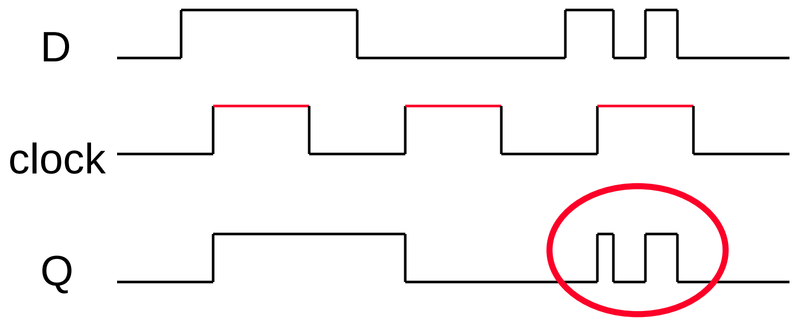
- Set-reset latch



R	S	Q(t+1)	!Q(t+1)
1	0	0	1
0	1	1	0
0	0	Q(t)	!Q(t)
1	1	0	0

- Level sensitive D latch

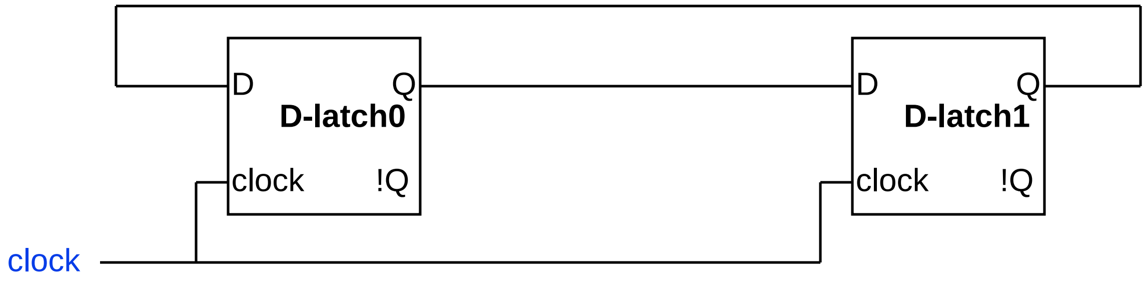




  - latch is transparent when clock is high (copies input to output)

Two-Sided Clock Constraint

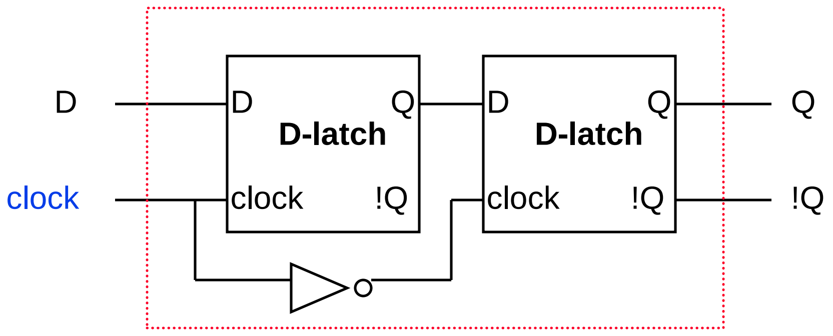
- Race problem with latch based design …

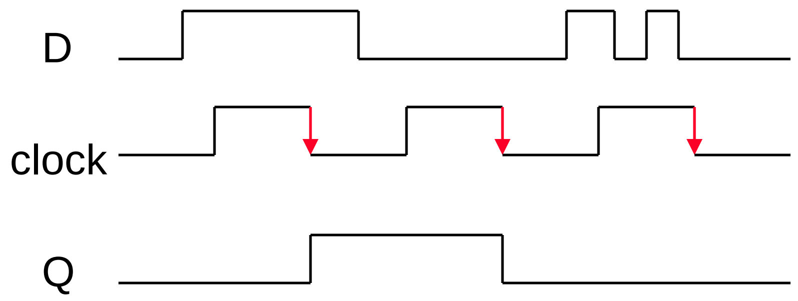


- Consider the case when D-latch0 holds a 0 and D-latch1 holds a 1 and you want to transfer the contents of D-latch0 to D-latch1 and vica versa
  - must have the clock high long enough for the transfer to take place
  - must not leave the clock high so long that the transferred data is copied back into the original latch
- Two-sided clock constraint

State Elements, con't

- Solution is to use flipflops that change state (Q) only on clock edge (master-slave)

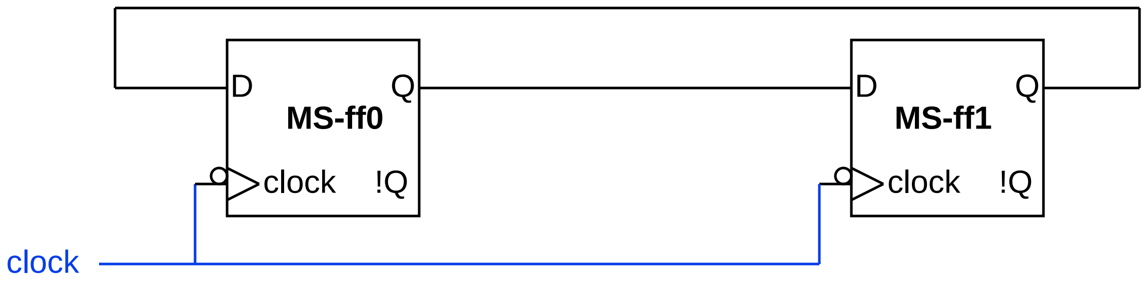




- master (first D-latch) copies the input when the clock is high (the slave (second D-latch) is locked in its memory state and the output does not change)
- slave copies the master when the clock goes low (the master is now locked in its memory state so changes at the input are not loaded into the master D-latch)

One-Slided Clock Constraint

- Master-slave (edge-triggered) flipflops removes one of the clock constraints



- Consider the case when MS-ff0 holds a 0 and MS-ff1 holds a 1 and you want to transfer the contents of MS-ff0 to MS-ff1 and vica versa
  - must have the clock cycle time long enough to accommodate the worst case delay path
- One-sided clock constraint

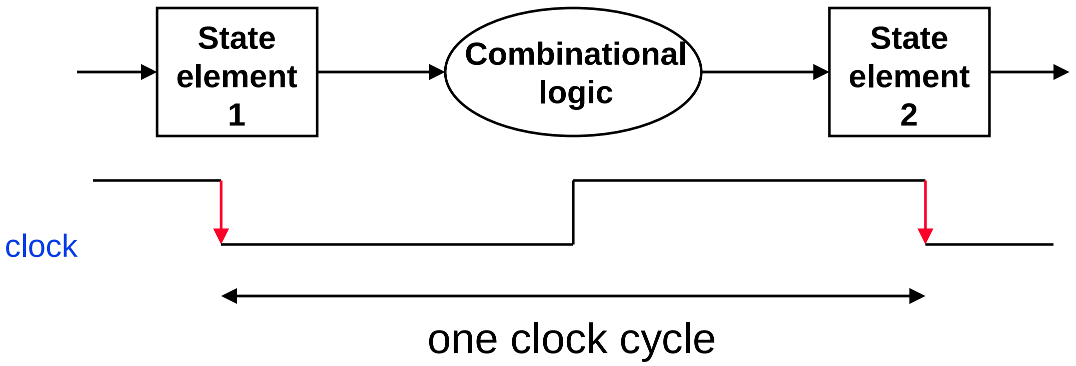
Latches vs Flipflops

- Output is equal to the stored value inside the element
- Change of state (value) is based on the clock
  - Latches: output changes whenever the inputs change and the clock is asserted (level sensitive methodology)
    - Two-sided timing constraint
  - Flip-flop: output changes only on a clock edge (edge-triggered methodology)
    - One-sided timing constraint

A clocking methodology defines when signals can be read and written – wouldn’t want to read a signal at the same time it was being written

Our Implementation

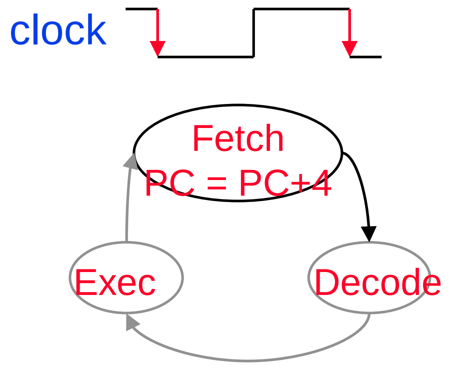
- An edge-triggered methodology, typical execution
  - read contents of some state elements (combinational activity, so no clock control signal needed)
  - send values through some combinational logic
  - write results to one or more state elements on clock edge

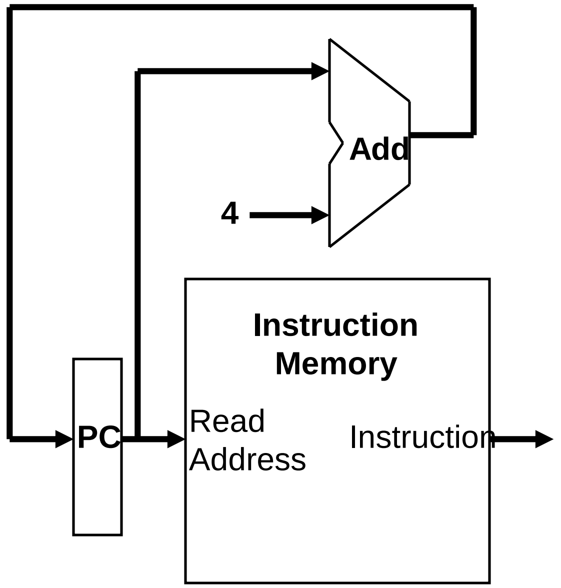


- Assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - write occurs only when both the write control is asserted and the clock edge occurs

Fetching Instructions

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

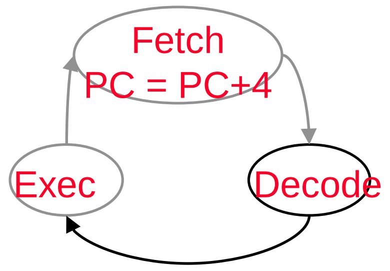


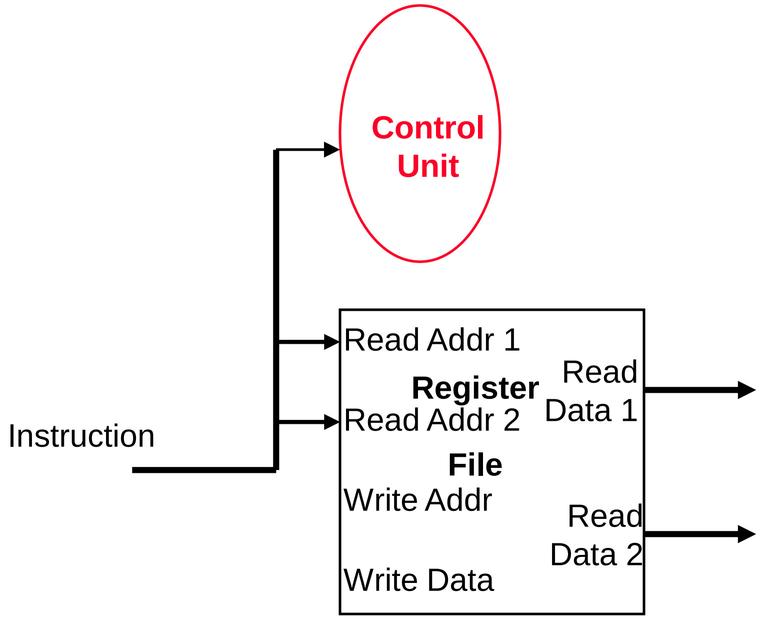


  - PC is updated every clock cycle, so it does not need an explicit write control signal just a clock signal
  - Reading from the Instruction Memory is a combinational activity, so it doesn't need an explicit read control signal

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

Reading Registers “Just in Case”

- Note that both RegFile read ports are active for all instructions during the Decode cycle using the rs and rt instruction field addresses
  - Since haven’t decoded the instruction yet, don’t know what the instruction is !
  - *Just in case* the instruction uses values from the RegFile do “work ahead” by reading the two source operands

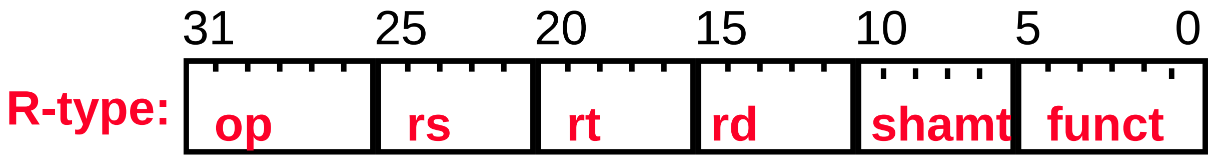
Which instructions *do* make use of the RegFile values?

- Also, all instructions (except `j`) use the ALU after reading the registers

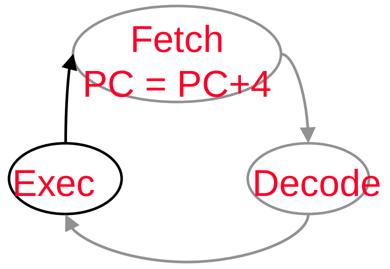
Why? memory-reference? arithmetic? control flow?

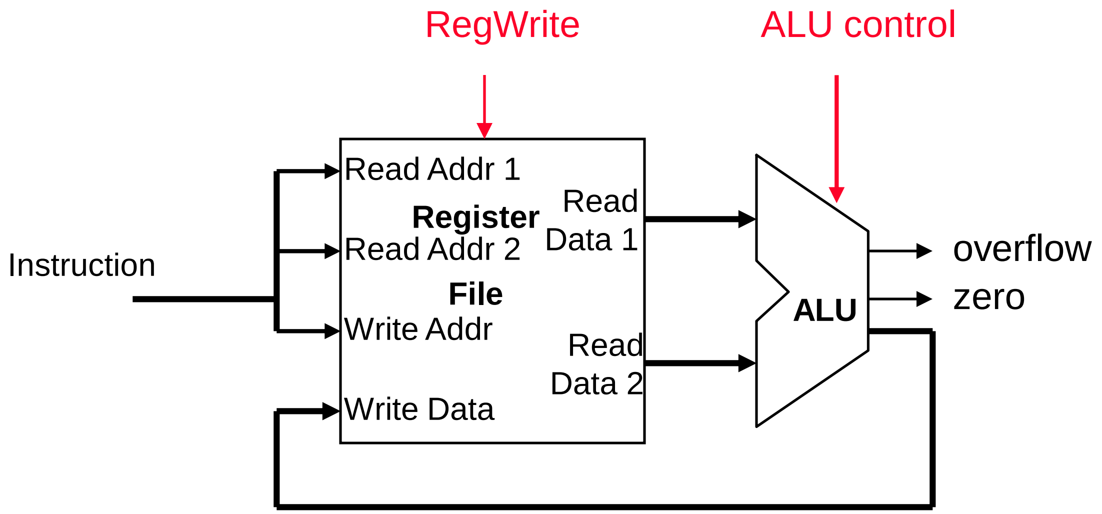
Executing R Format Operations

- 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)

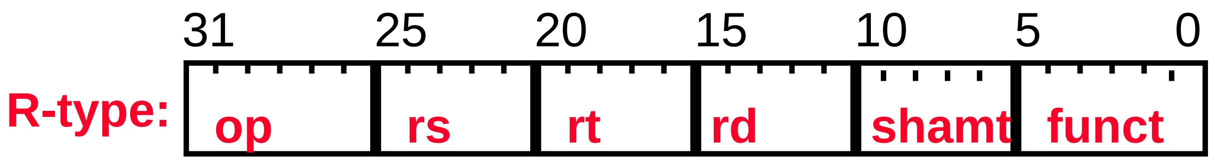




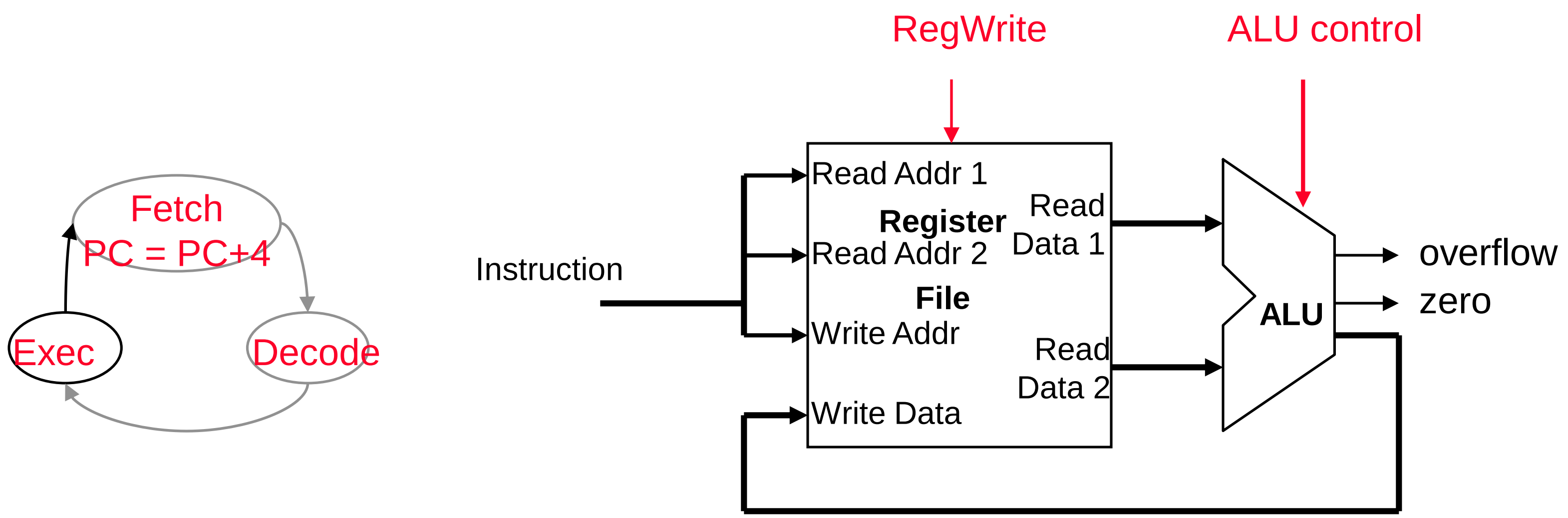
- Note that Register File is not written every cycle (e.g. **`sw`**), so we need an explicit write control signal for the Register File

Consider `slt` Instruction

- 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)



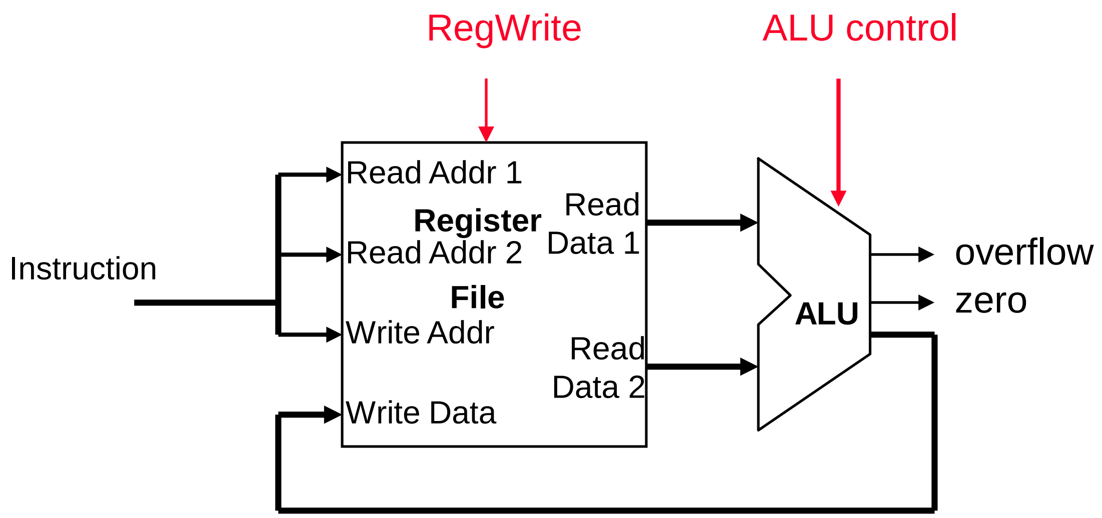
- Note that Register File is not written every cycle (e.g. `sw`), so we need an explicit write control signal for the Register File

Consider the `slt` Instruction

- Remember the R format instruction `slt`

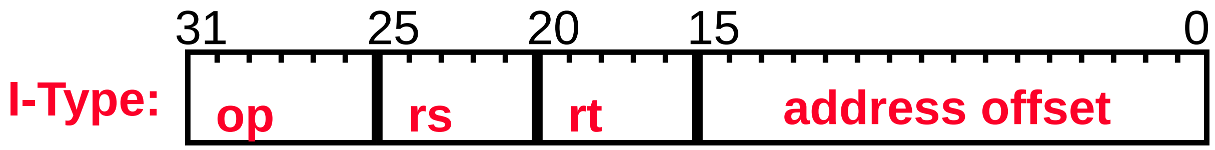
```
slt $t0, $s0, $s1  # if $s0 < $s1
                   #      then  $t0 = 1
                   #      else  $t0 = 0
```

- Where does the 1 (or 0) come from to store into $t0 in the Register File at the end of the execute cycle?



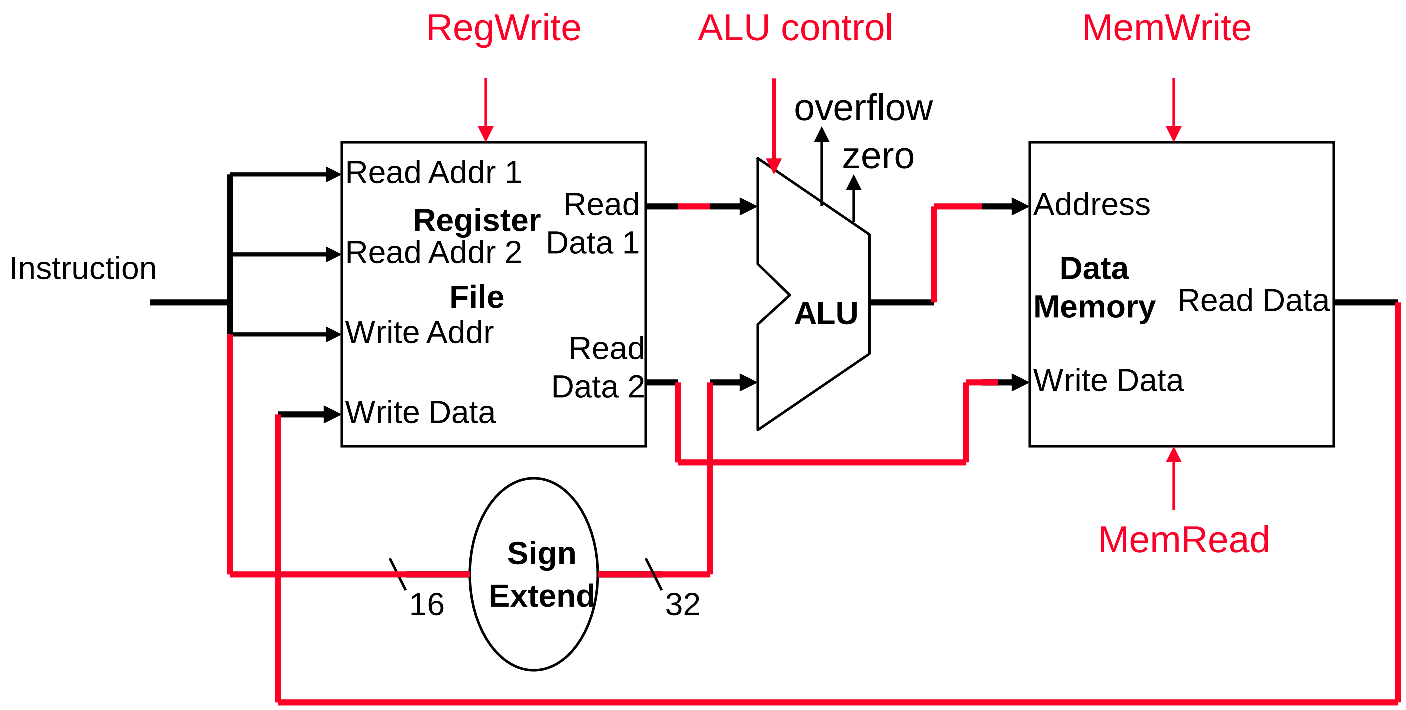
Executing Load and Store Operations

- Load and store operations have to



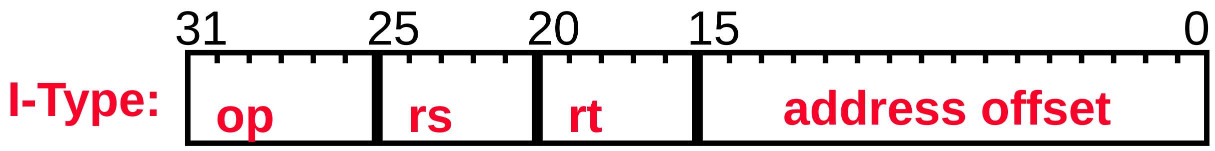
- compute a memory address by adding the base register (in rs) to the 16-bit signed offset field in the instruction
  - base register was read from the Register File during decode
  - offset value in the low order 16 bits of the instruction must be sign extended to create a 32-bit signed value
- store value, read from the Register File during decode, must be written to the Data Memory
- load value, read from the Data Memory, must be stored in the Register File

Executing Load and Store Operations, con't



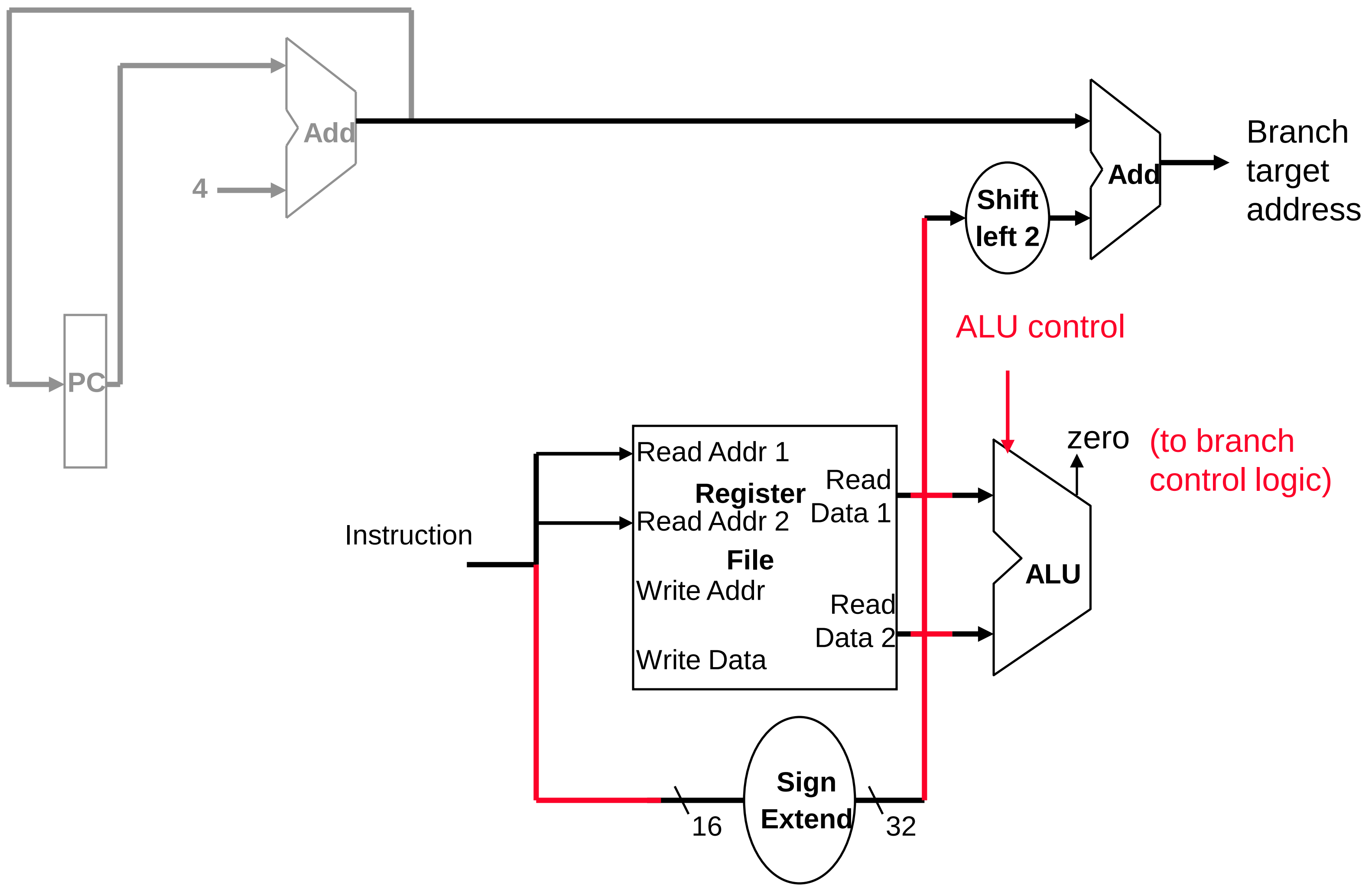
Executing Branch Operations

- Branch operations have to



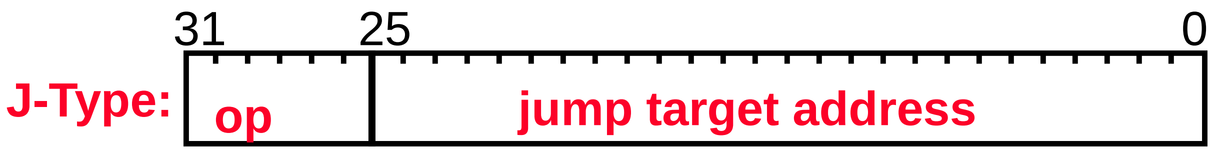
  - compare the operands read from the Register File during decode (rs and rt values) for equality (`zero` ALU output)
  - compute the branch target address by adding the updated PC to the sign extended16-bit signed offset field in the instruction
    - "base register" is the updated PC
    - offset value in the low order 16 bits of the instruction must be sign extended to create a 32-bit signed value and then shifted left 2 bits to turn it into a word address

Executing Branch Operations, con't

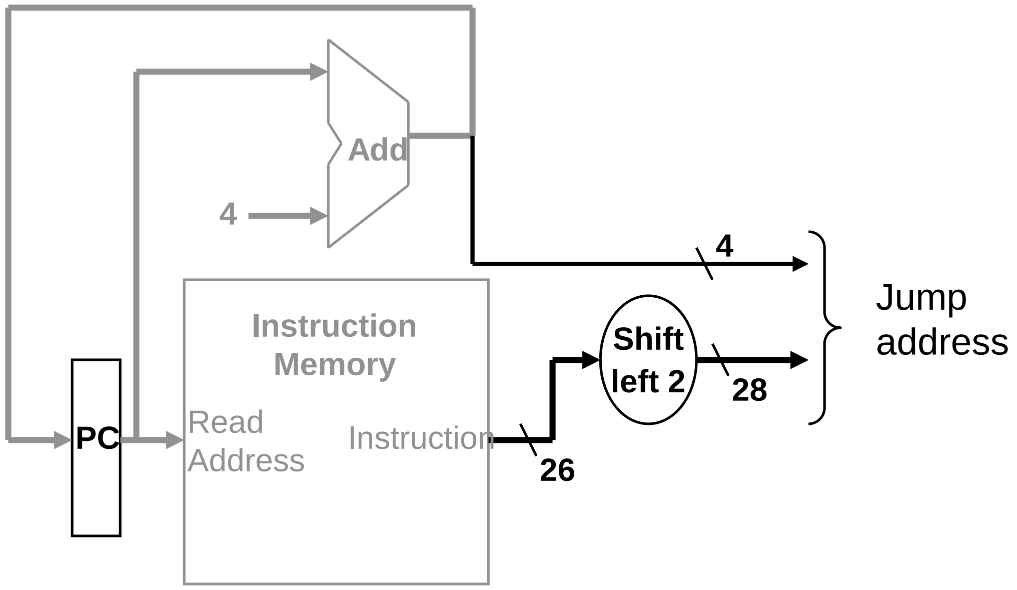


Executing Jump Operations

- Jump operations have to



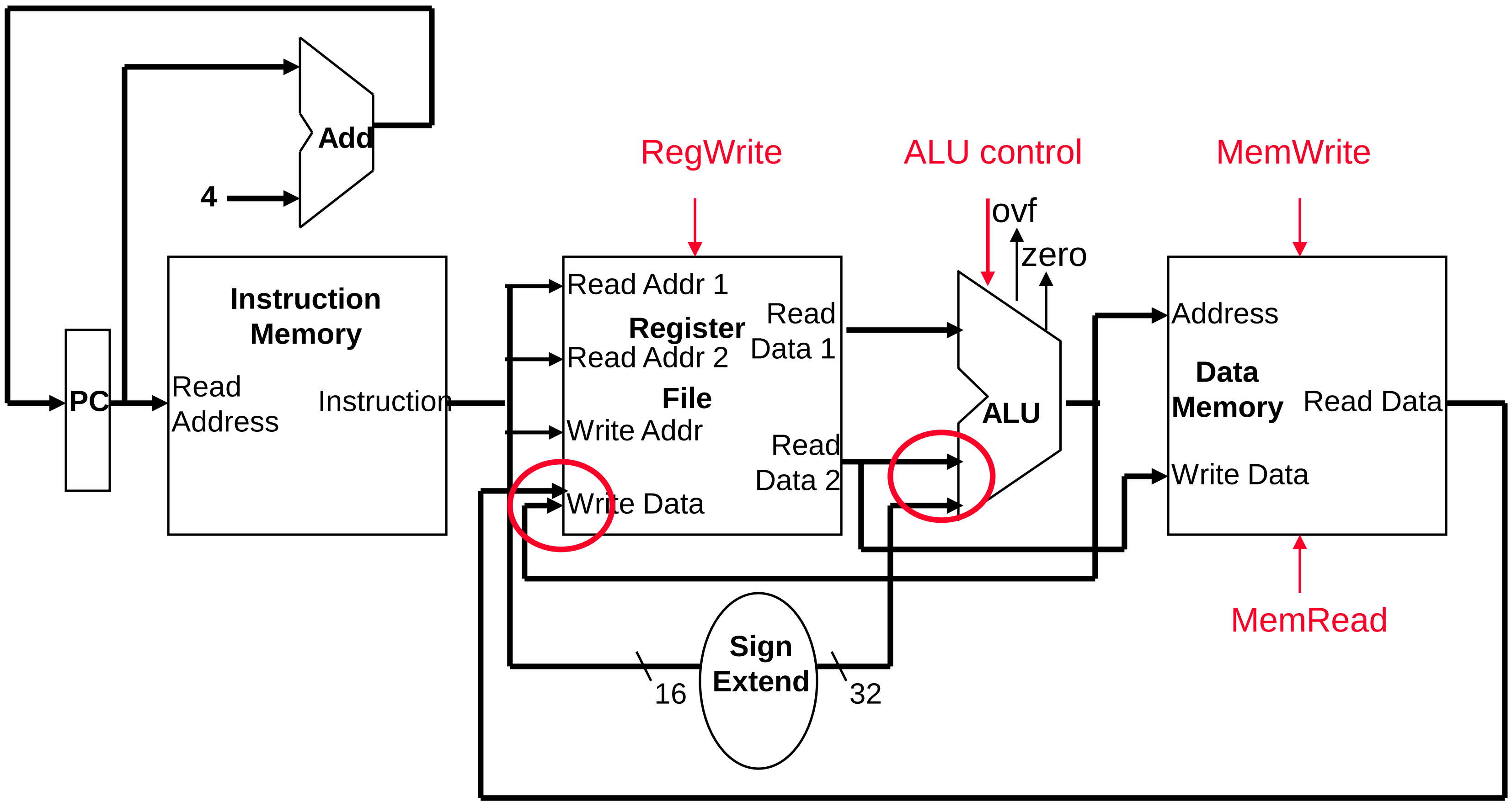
  - replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits



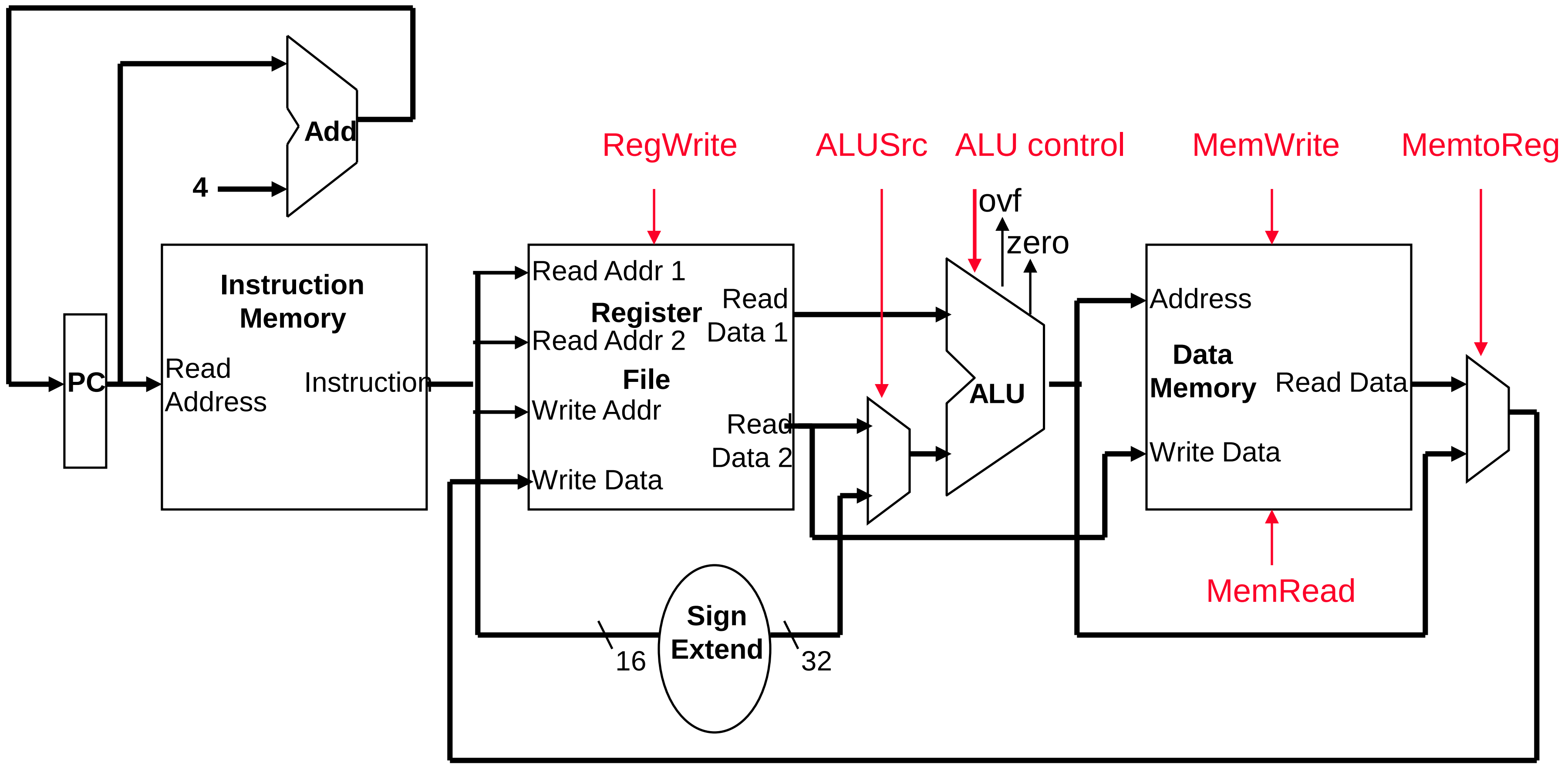
Creating a Single Datapath from the Parts

- Assemble the datapath elements, add control lines as needed, and design the control path
- Fetch, decode and execute each instructions in one clock cycle – single cycle design
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., why we have a separate Instruction Memory and Data Memory)
  - to share datapath elements between two different instruction classes will need multiplexors at the input of the shared elements with control lines to do the selection
- Cycle time is determined by length of the longest path

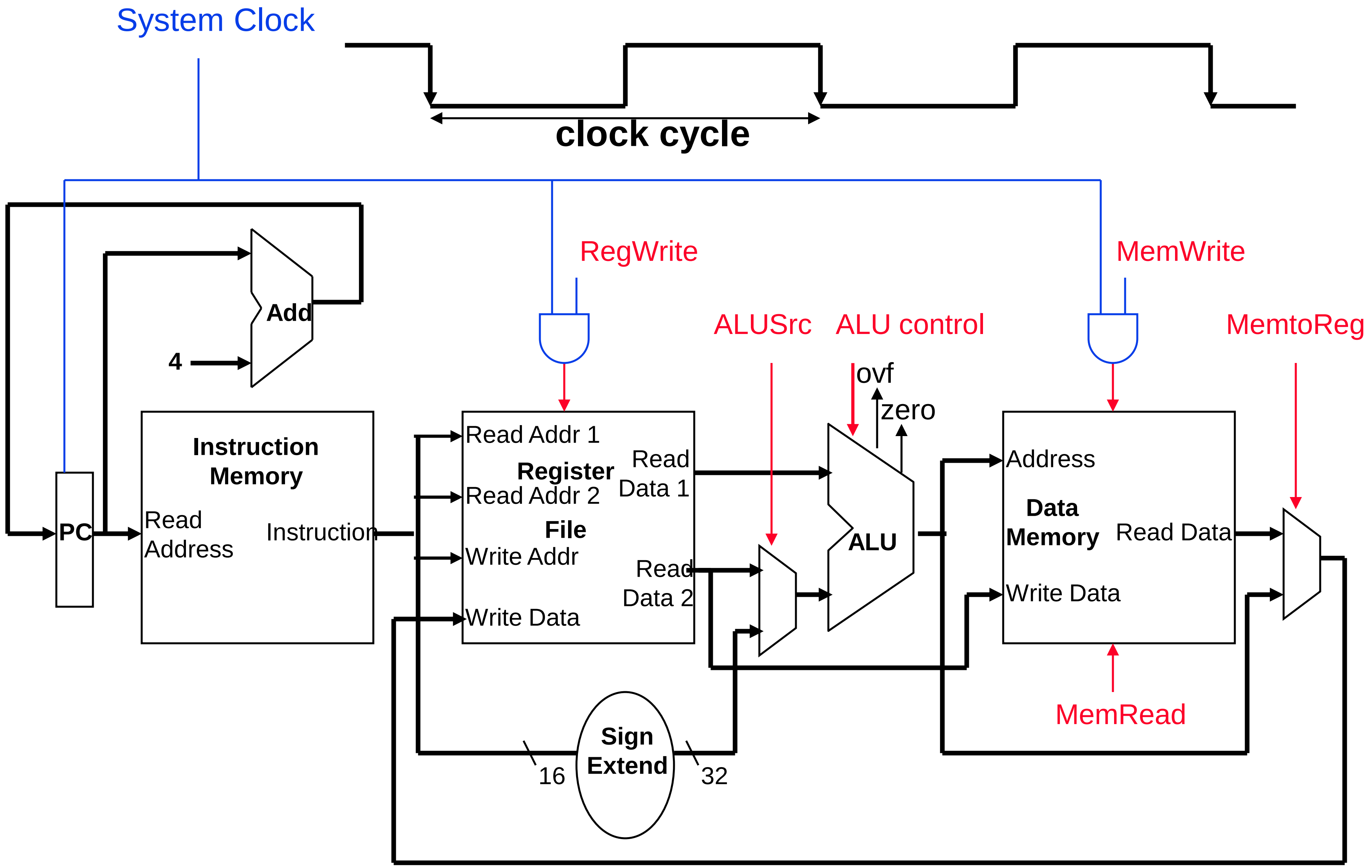
Fetch, R, and Memory Access Portions



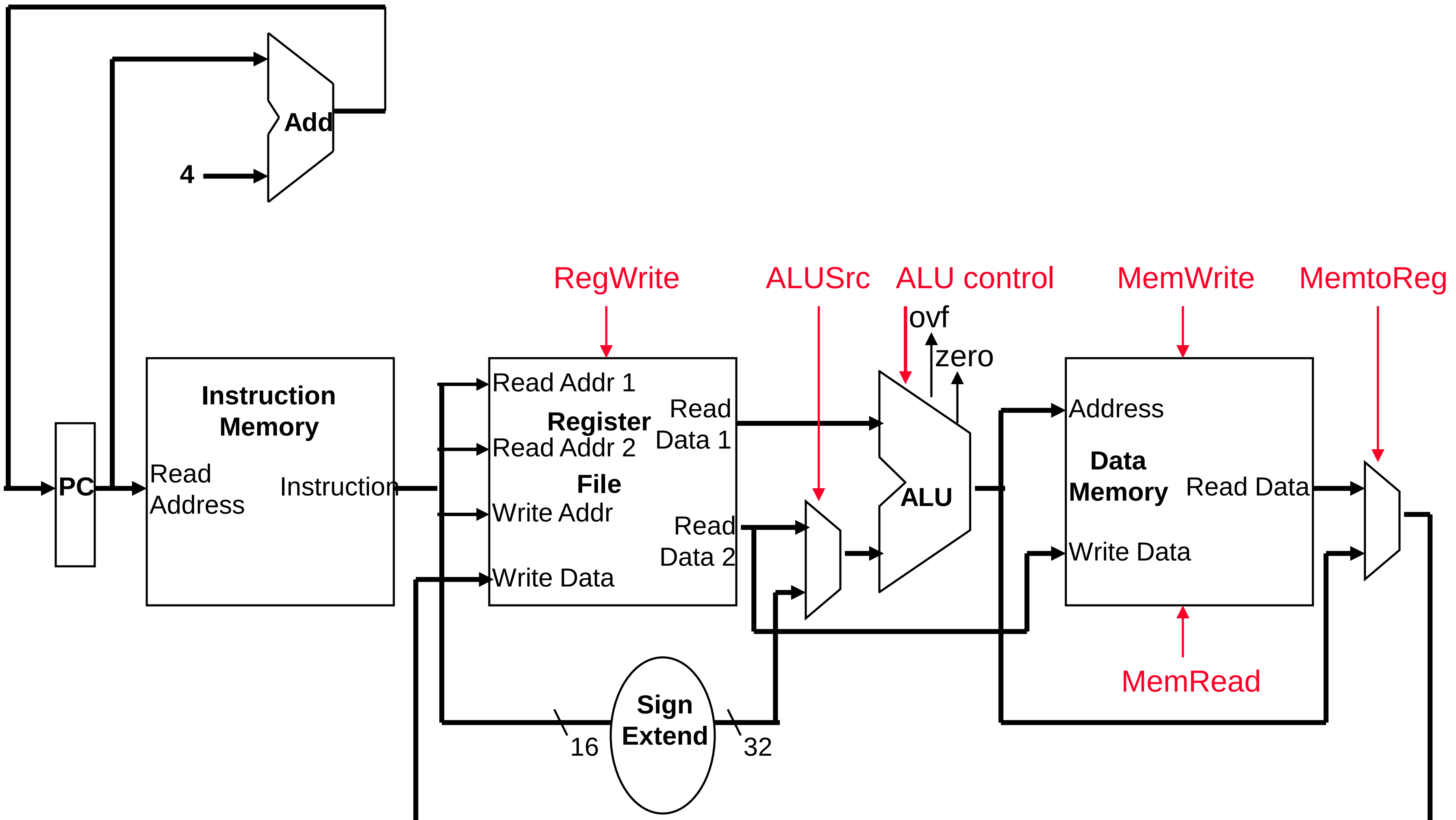
Multiplexor Insertion



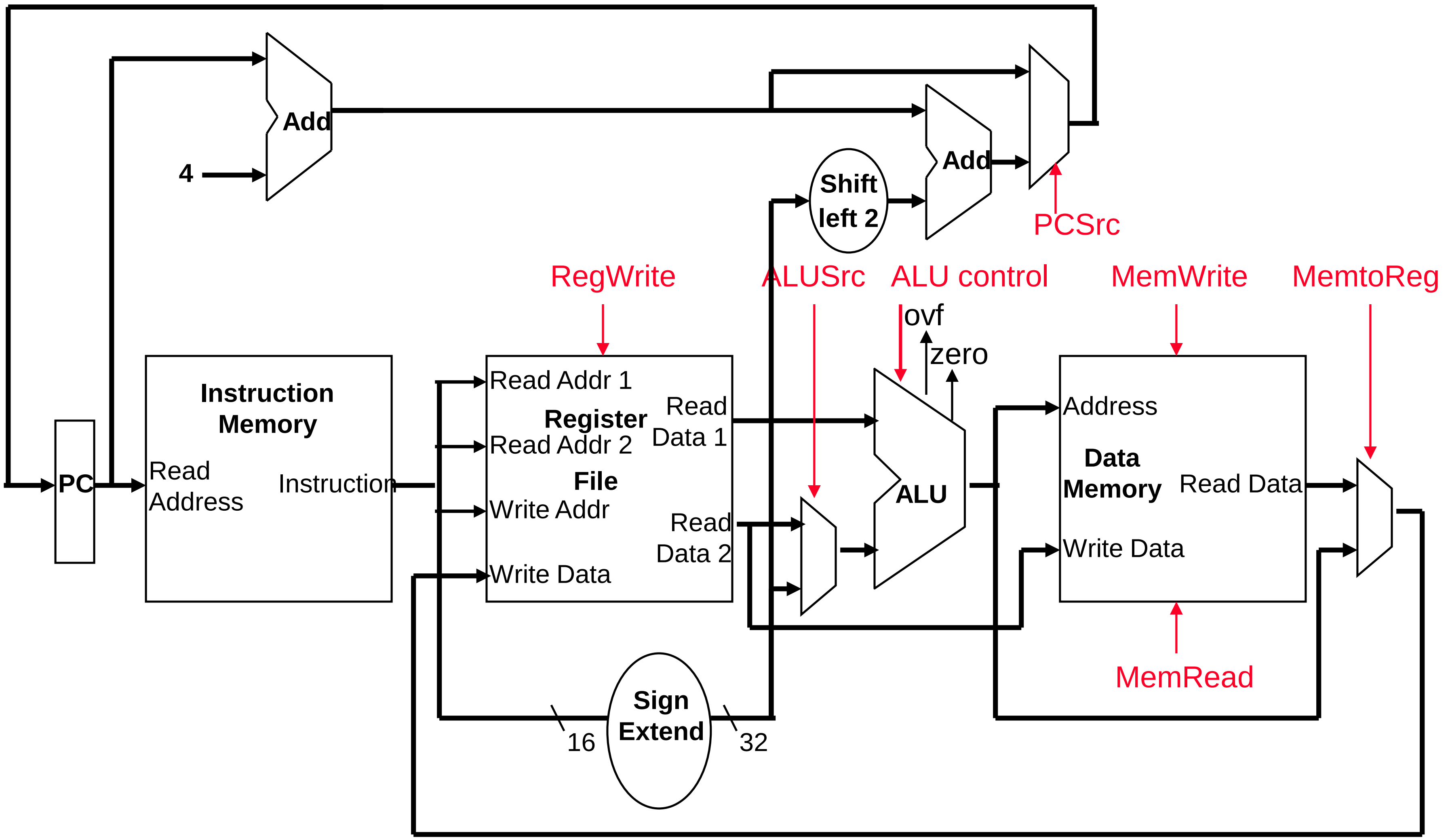
Clock Distribution



Adding the Branch Portion



Adding the Branch Portion



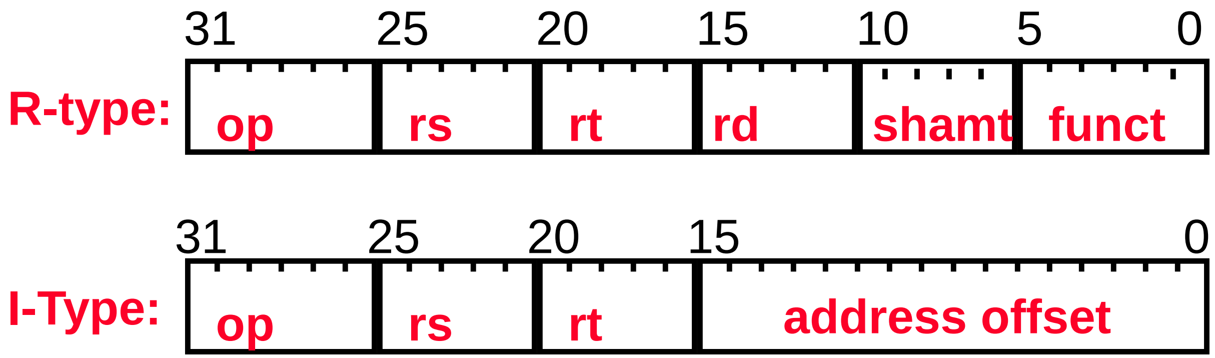
Our Simple Control Structure

- We wait for everything to settle down
  - ALU might not produce “right answer” right away
  - Memory and RegFile reads are combinational (as are ALU, adders, muxes, shifter, signextender)
  - Use write signals along with the clock edge to determine when to write to the sequential elements (to the PC, to the Register File and to the Data Memory)
- The clock cycle time is determined by the logic delay through the longest path

We are ignoring some details like register setup and hold times

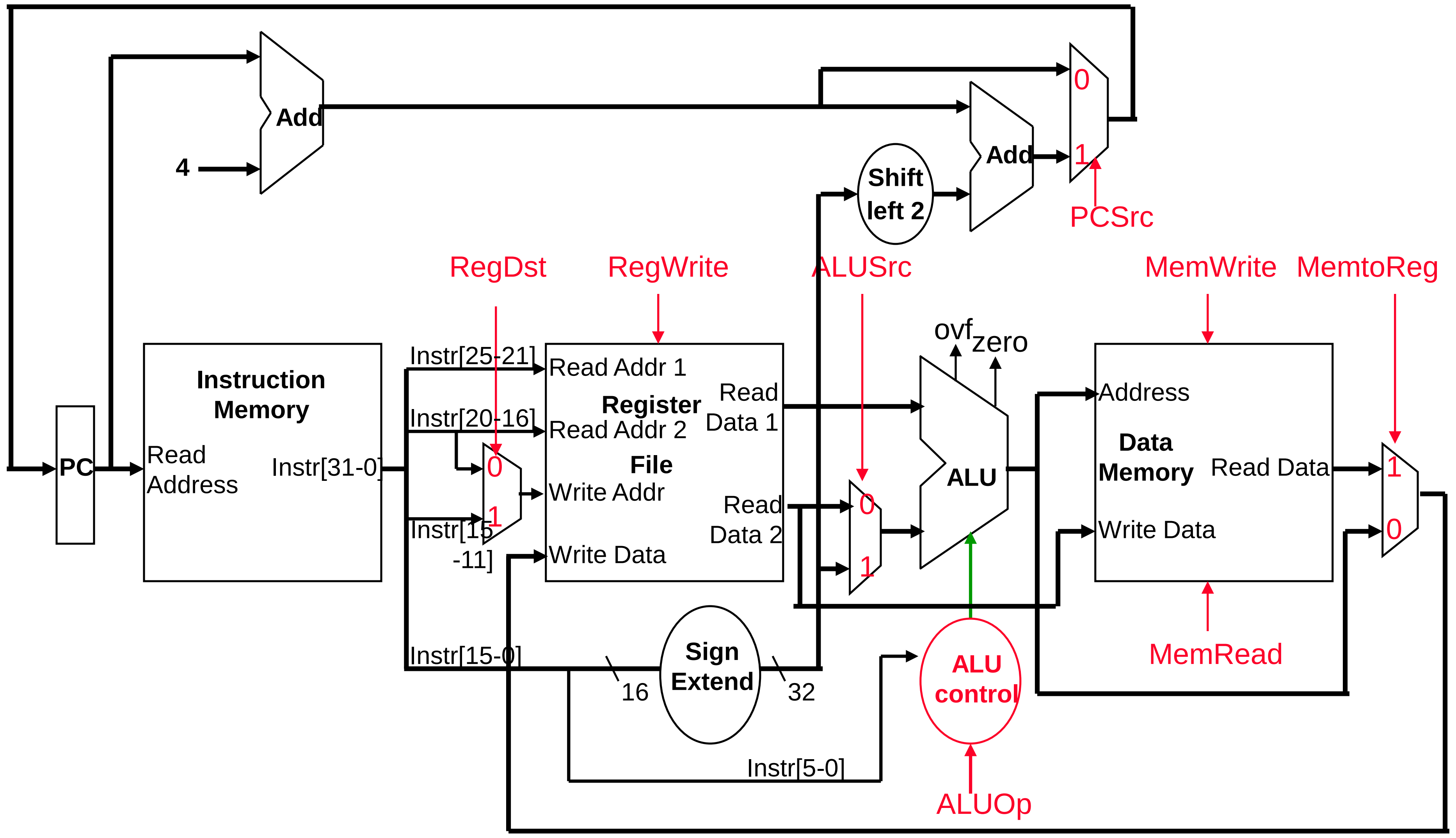
Adding the Control

- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction



- Observations
  - op field always in bits 31-26
  - addr of two registers to be read are always specified by the rs and rt fields (bits 25-21 and 20-16)
  - base register for lw and sw always in rs (bits 25-21)
  - 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

(Almost) Complete Single Cycle Datapath



ALU Control

- ALU's operation based on instruction type and function code

ALU control input	Function
0000	and
0001	or
0010	xor
0011	nor
0110	add
1110	subtract
1111	set on less than

- Notice that we are using different encodings than in the book

ALU Control, Con’t

- Controlling the ALU uses of multiple decoding levels
  - main control unit generates the ALUOp bits
  - ALU control unit generates ALUcontrol bits

Instr op	funct	ALUOp	action	ALUcontrol
lw	xxxxxx	00	add	0110
sw	xxxxxx	00	add	0110
beq	xxxxxx	01	subtract	1110
add	100000	10	add	0110
sub	100010	10	subtract	1110
and	100100	10	and	0000
or	100101	10	or	0001
xor	100110	10	xor	0010
nor	100111	10	nor	0011
slt	101010	10	slt	1111

ALU Control Truth Table

Our ALU m control input

F5	F4	F3	F2	F1	F0	ALU Op_1	ALU Op_0	ALU $control_3$	ALU $control_2$	ALU $control_1$	ALU $control_0$
X	X	X	X	X	X	0	0	0	1	1	0
X	X	X	X	X	X	0	1	1	1	1	0
X	X	0	0	0	0	1	0	0	1	1	0
X	X	0	0	1	0	1	0	1	1	1	0
X	X	0	1	0	0	1	0	0	0	0	0
X	X	0	1	0	1	1	0	0	0	0	1
X	X	0	1	1	0	1	0	0	0	1	0
X	X	0	1	1	1	1	0	0	0	1	1
X	X	1	0	1	0	1	0	1	1	1	1

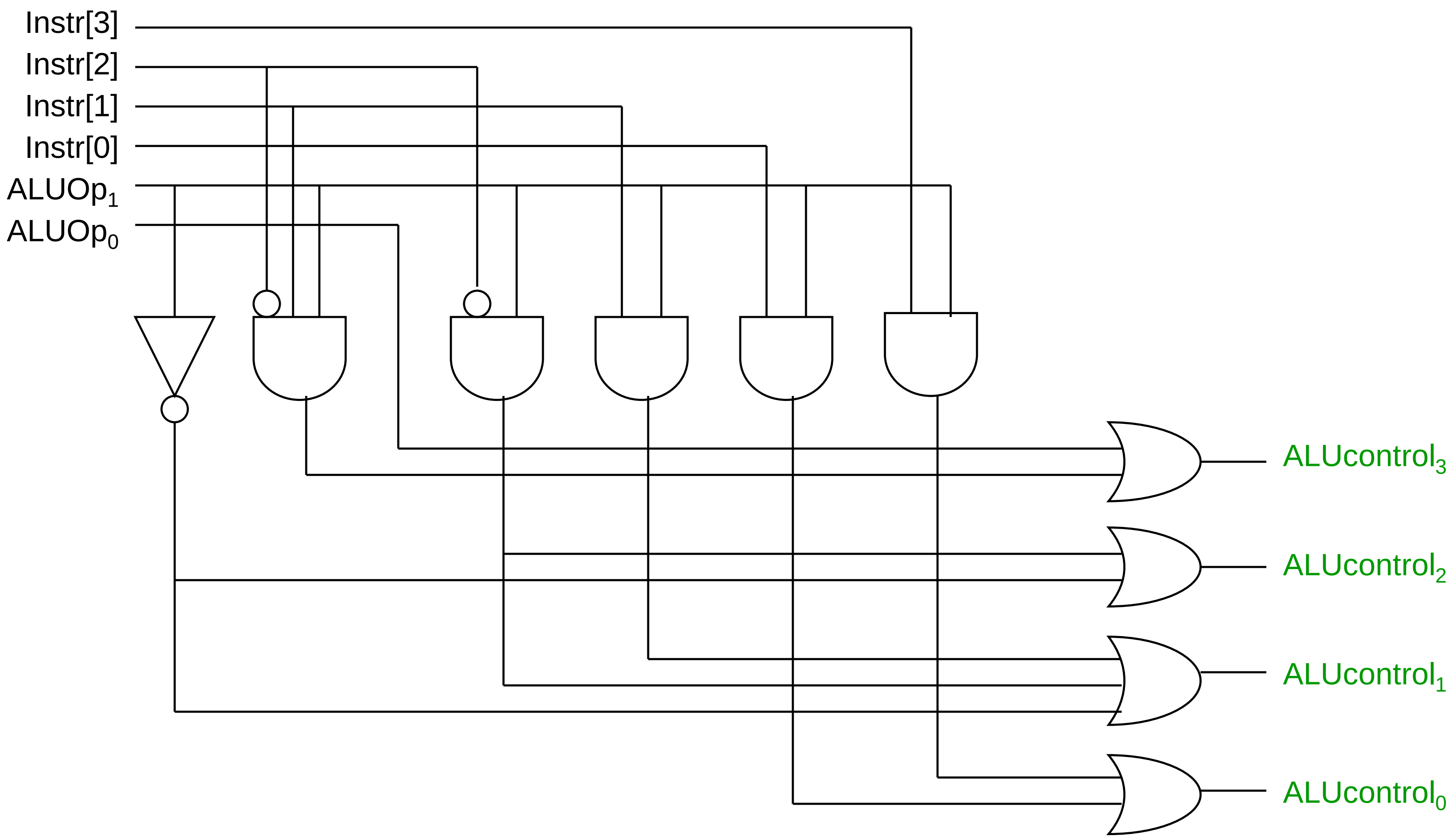
Add/subt

Mux control

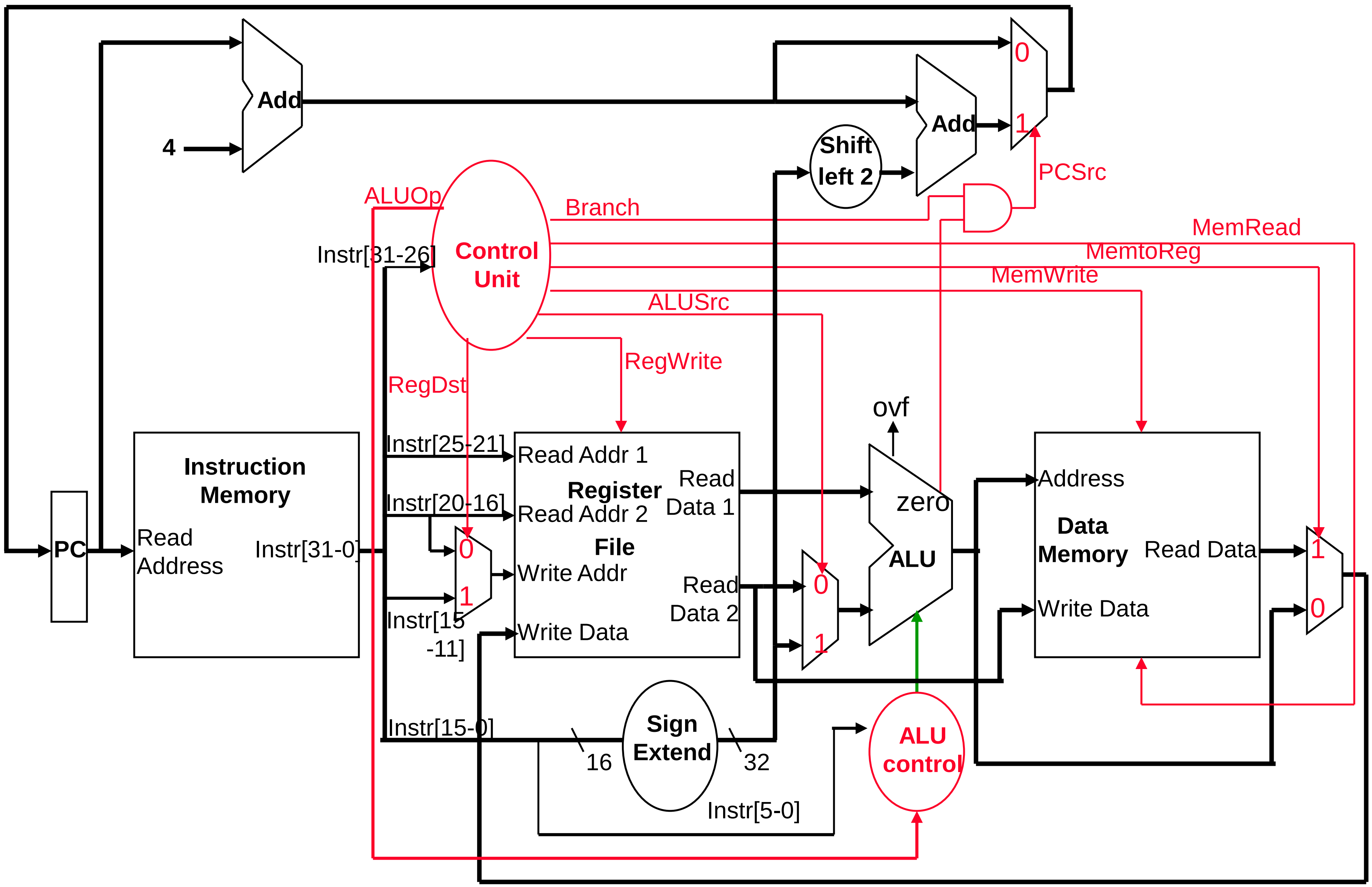
- Four, 6-input truth tables

ALU Control Logic

- From the truth table can design the ALU Control logic



(Almost) Complete Datapath with Control Unit

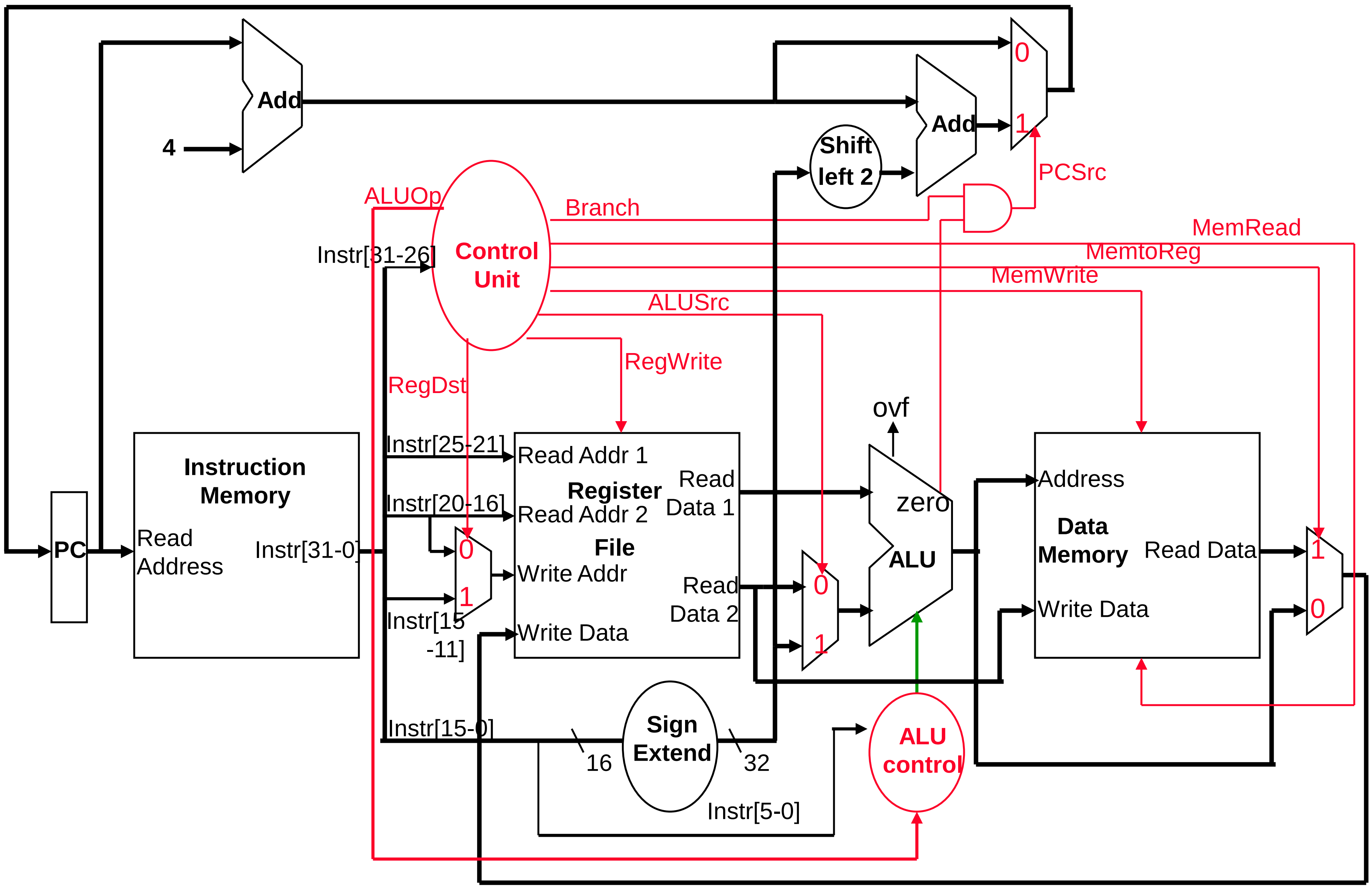


Main Control Unit

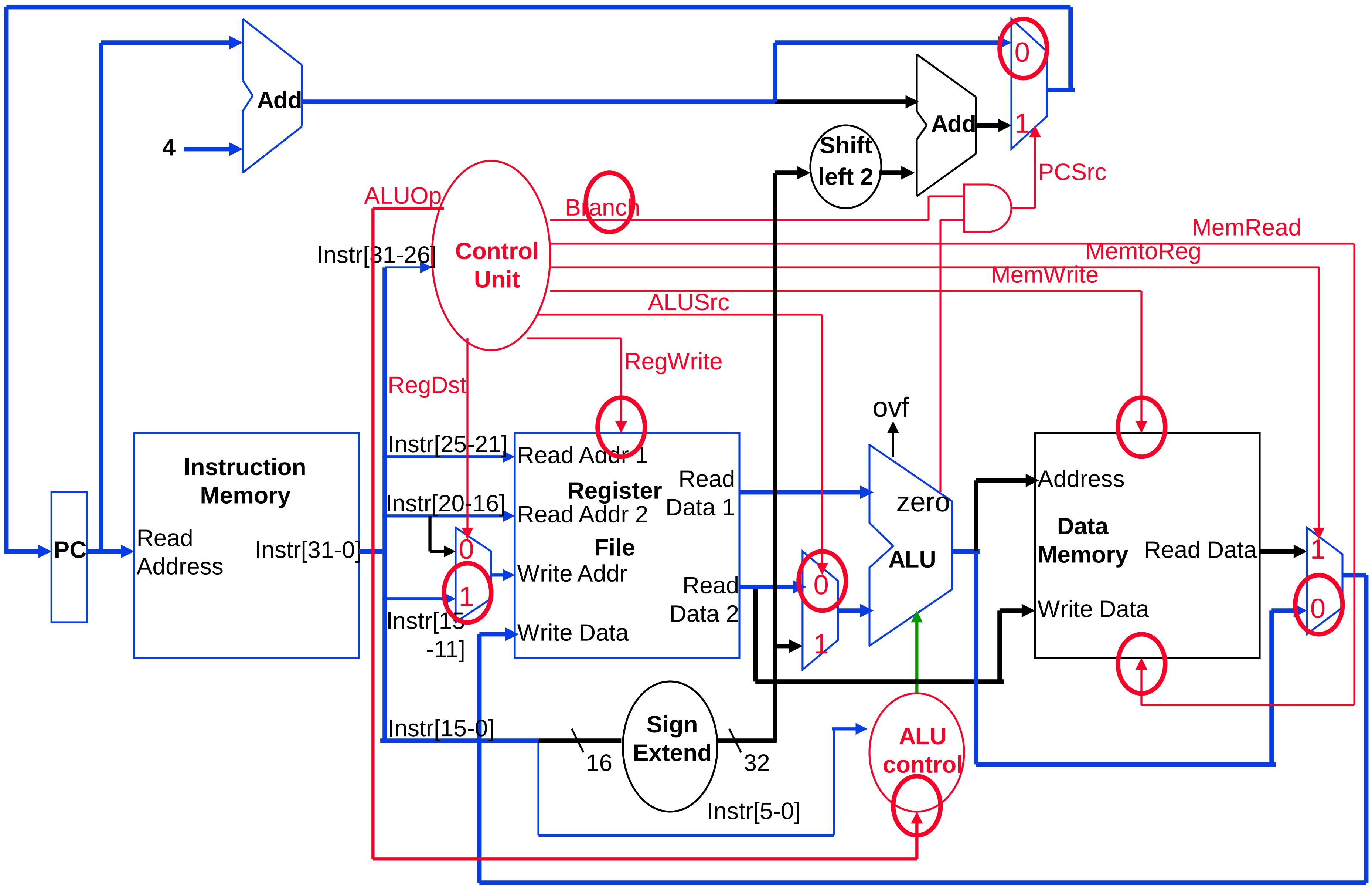
Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp
R-type 000000								
lw 100011								
sw 101011								
beq 000100								

- Completely determined by the instruction opcode field
  - Note that a multiplexor whose control input is 0 has a definite action, even if it is not used in performing the operation

R-type Instruction Data/Control Flow



R-type Instruction Data/Control Flow

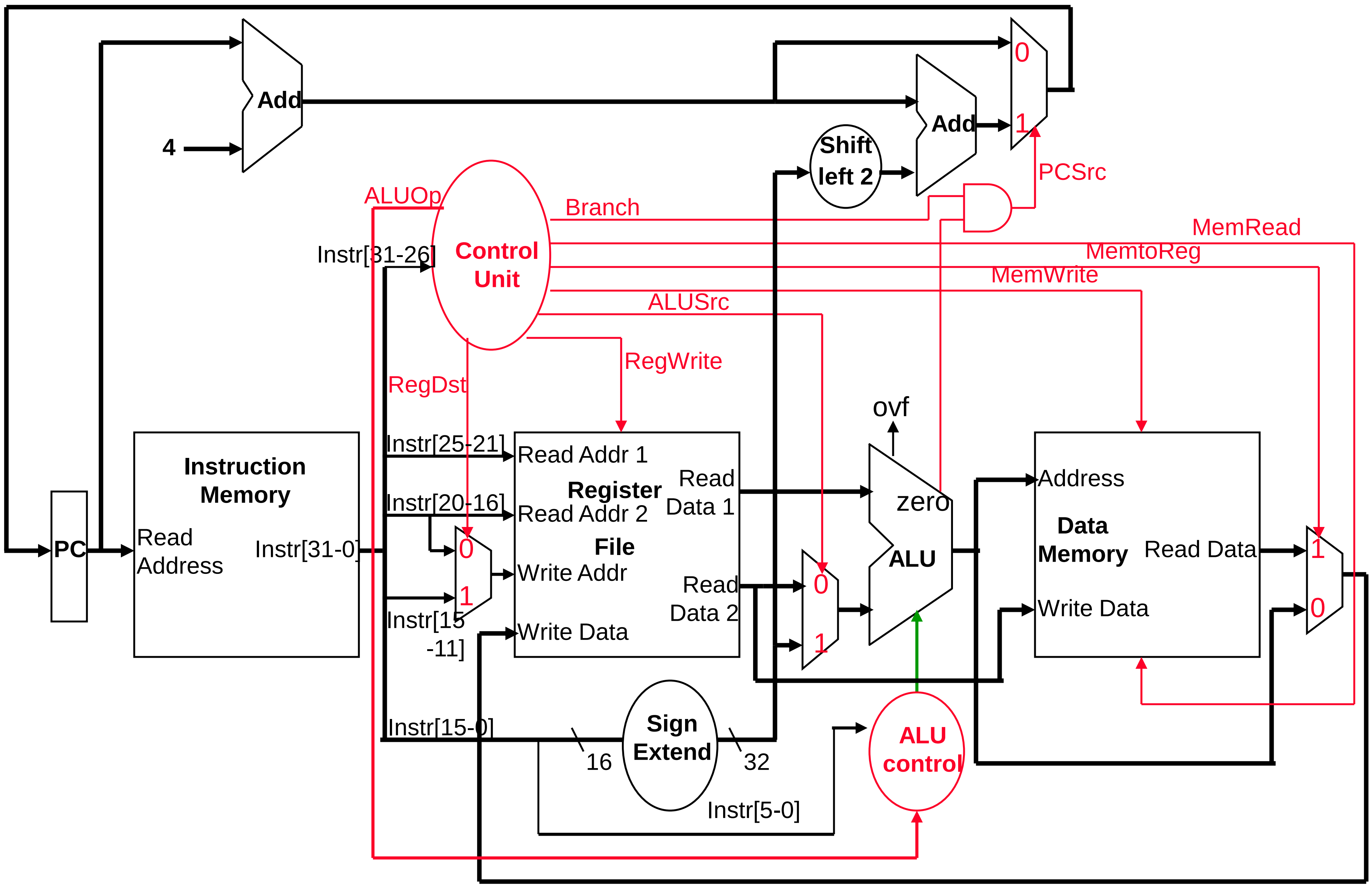


Main Control Unit

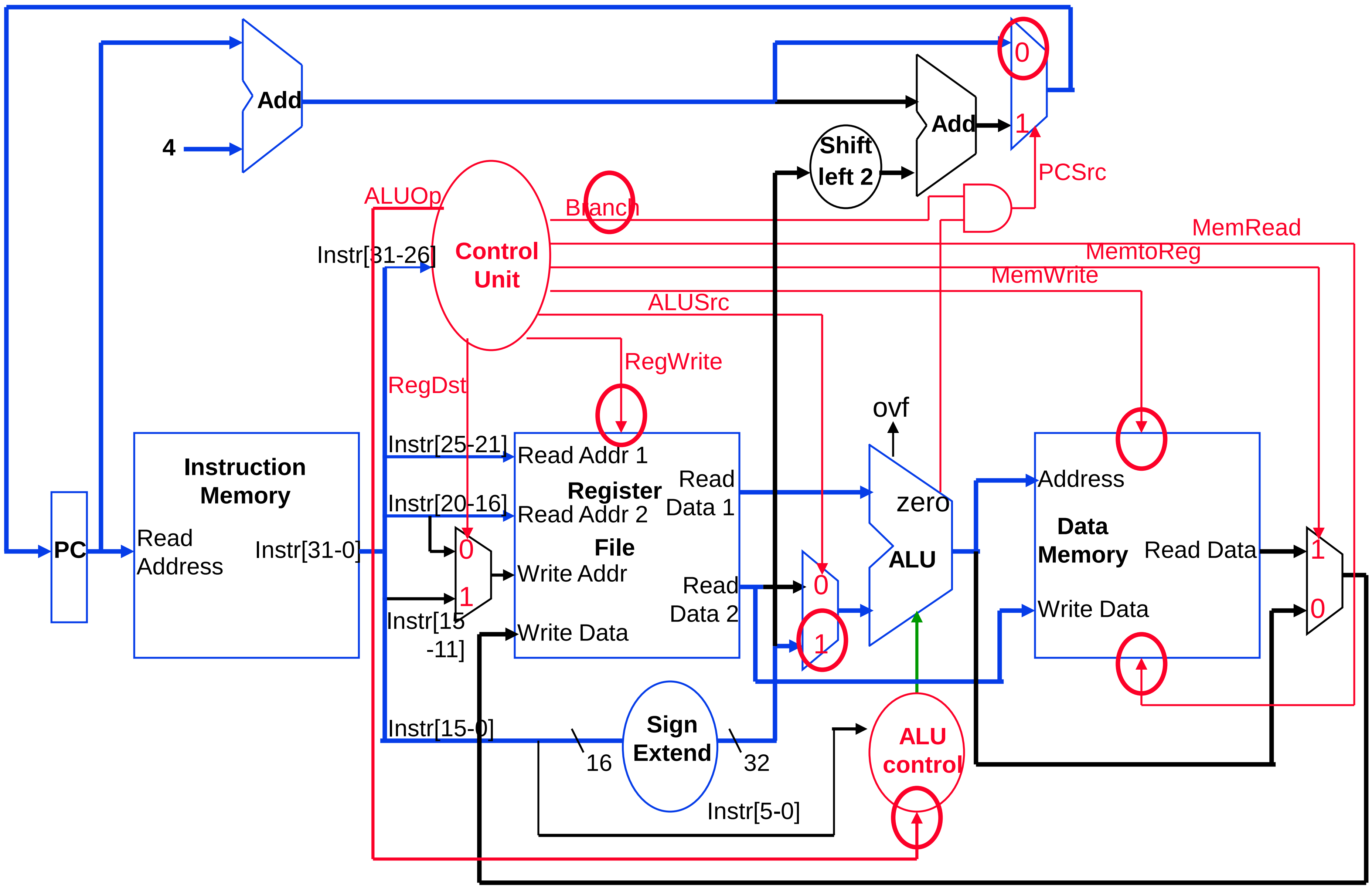
Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp
R-type 000000	1	0	0	1	0	0	0	10
lw 100011								
sw 101011								
beq 000100								

- Setting of the MemRd signal (for R-type, sw, beq) depends on the memory design (could have to be 0 or could be a X (don't care))

Store Word Instruction Data/Control Flow



Store Word Instruction Data/Control Flow

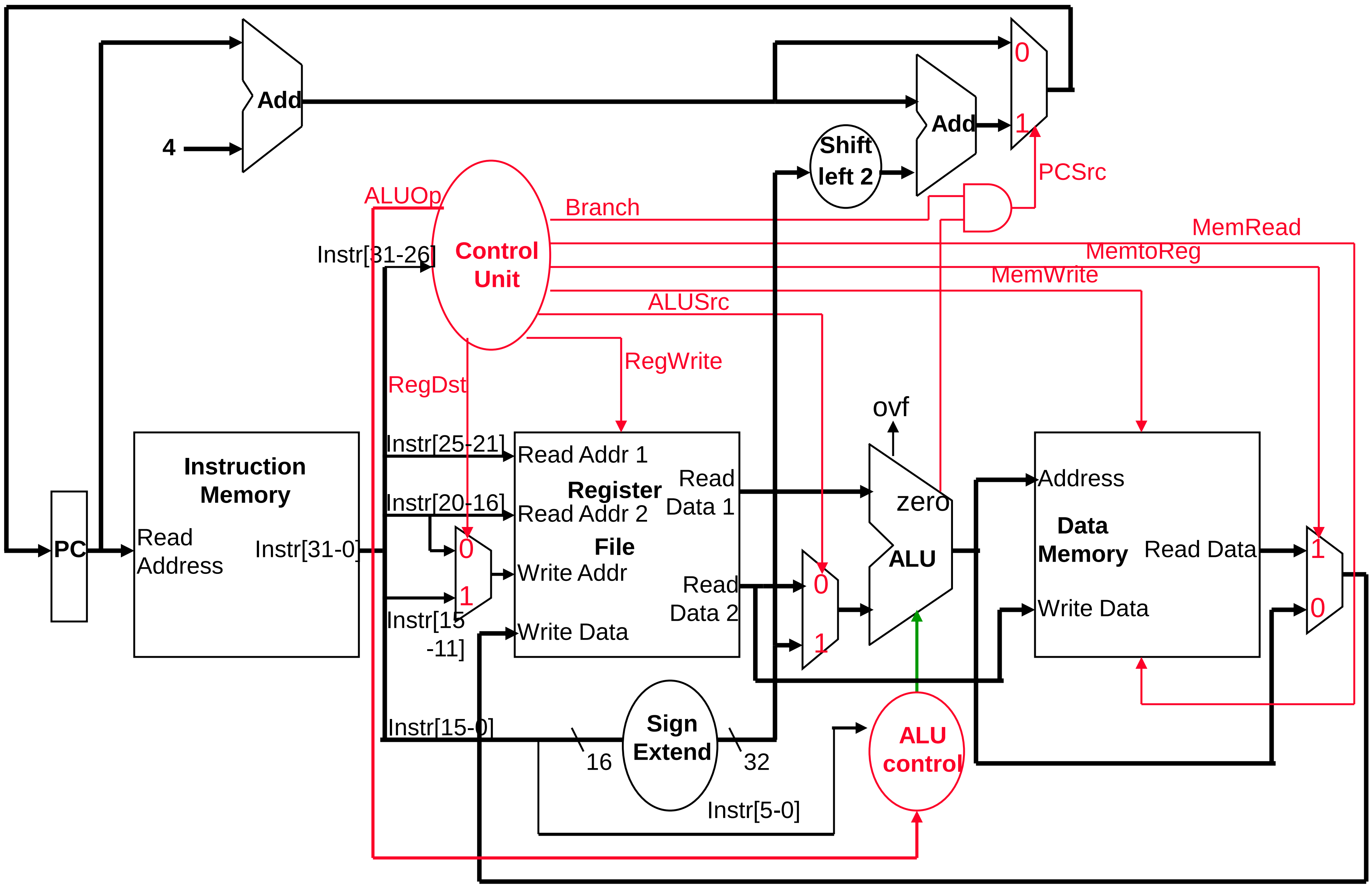


Main Control Unit

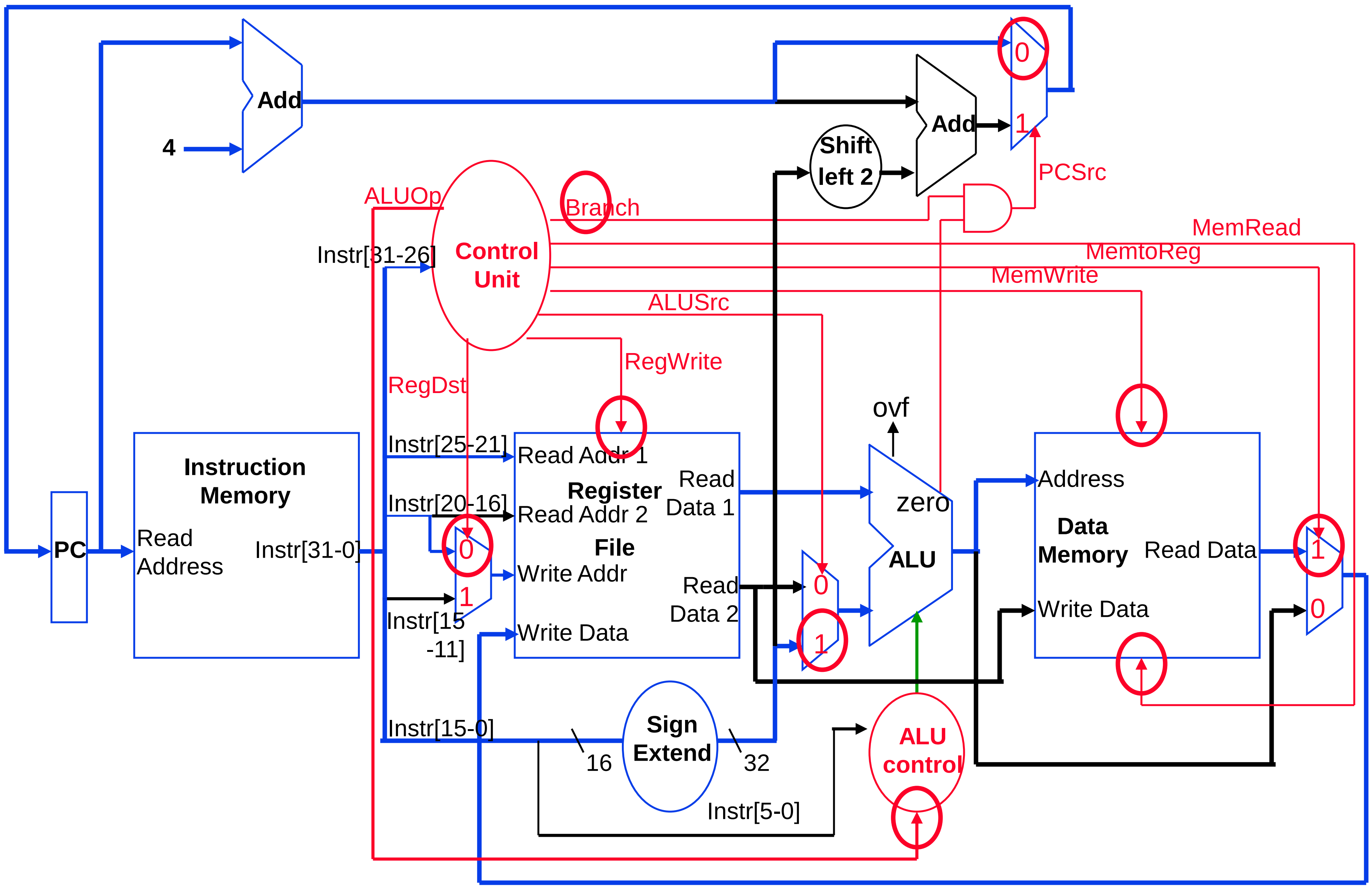
Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp
R-type 000000	1	0	0	1	0	0	0	10
lw 100011								
sw 101011	X	1	X	0	0	1	0	00
beq 000100								

- Setting of the MemRd signal (for R-type, sw, beq) depends on the memory design (could have to be 0 or could be a X (don't care))

Load Word Instruction Data/Control Flow



Load Word Instruction Data/Control Flow

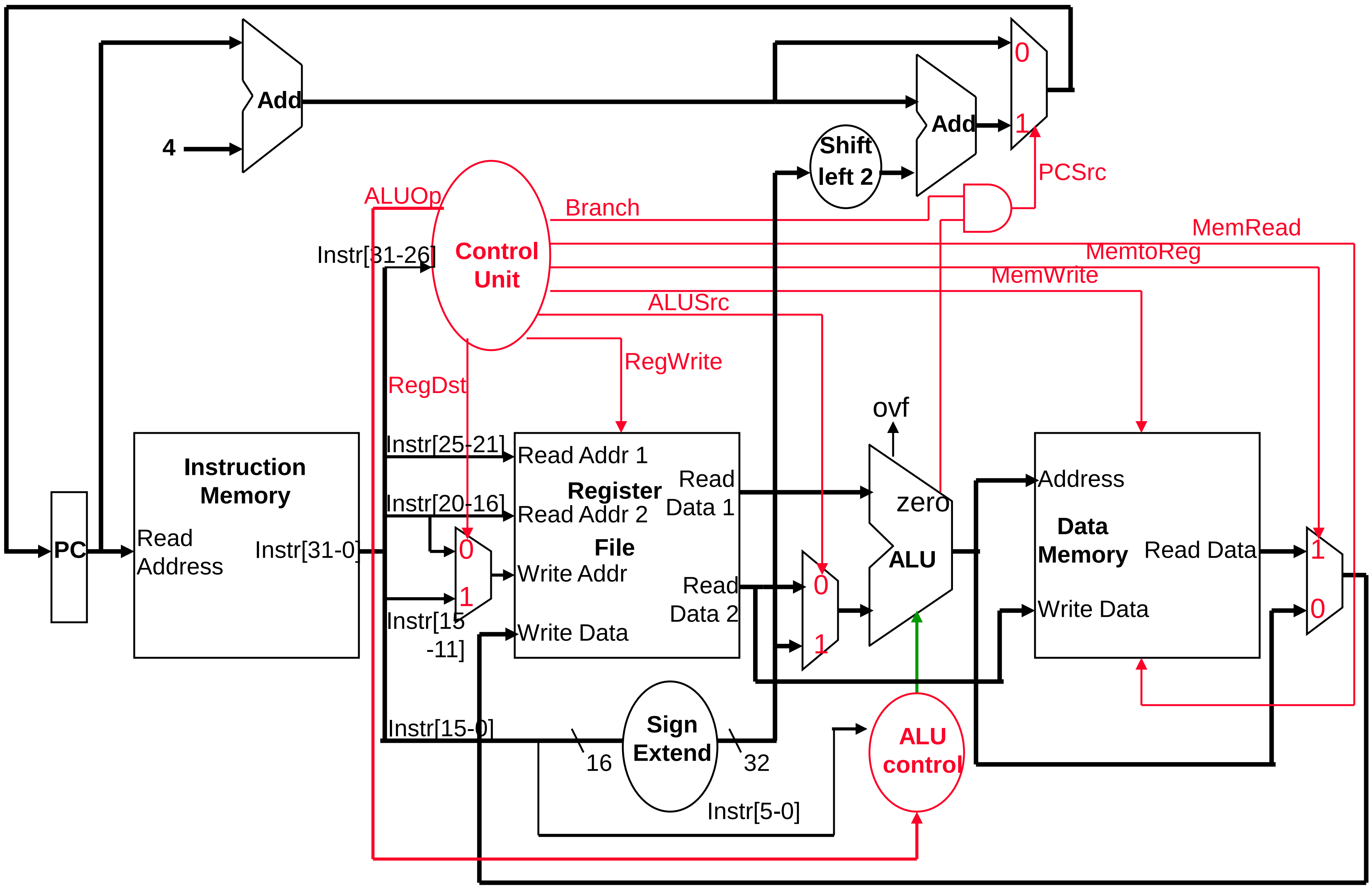


Main Control Unit

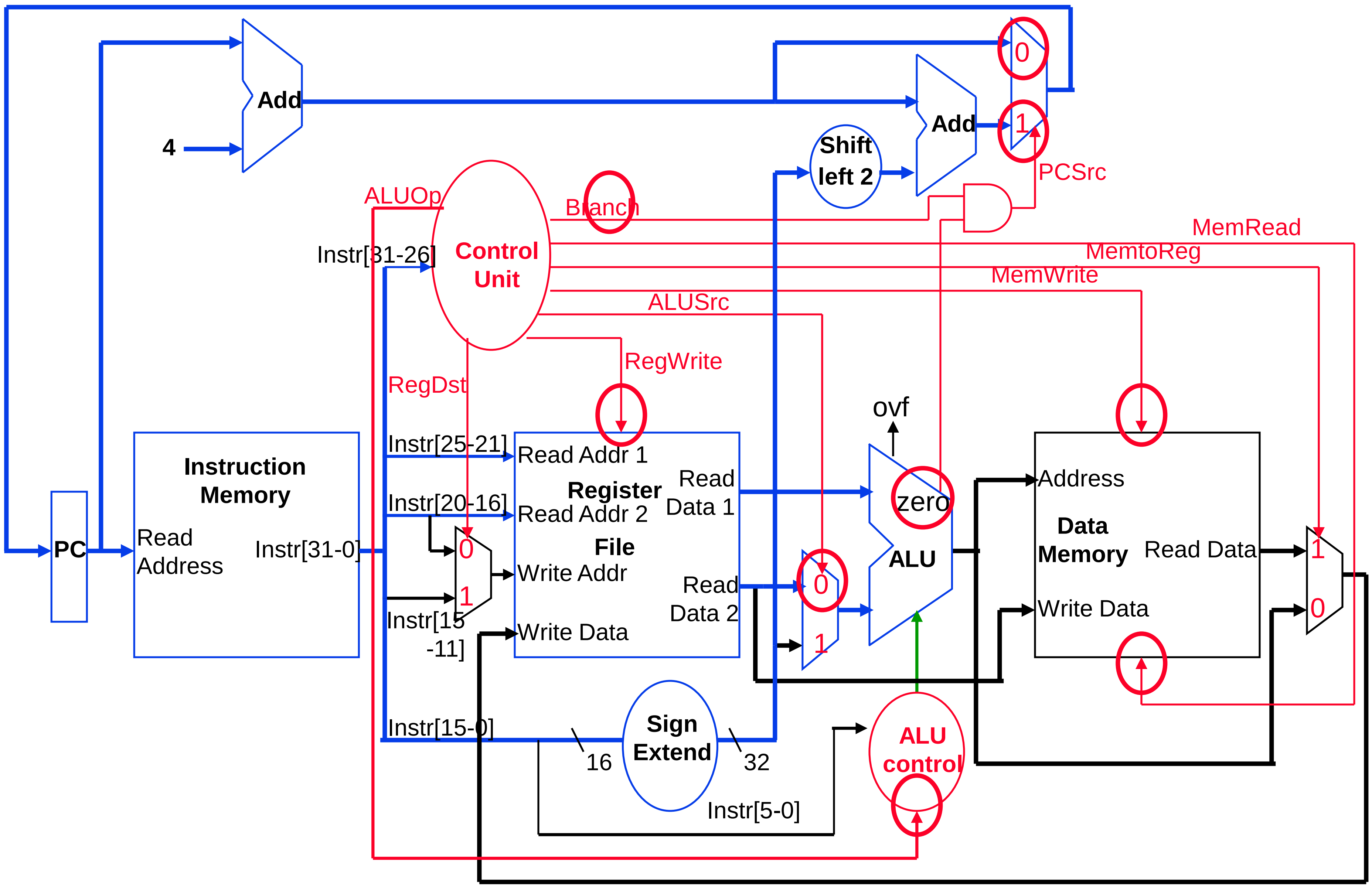
Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp
R-type 000000	1	0	0	1	0	0	0	10
lw 100011	0	1	1	1	1	0	0	00
sw 101011	X	1	X	0	0	1	0	00
beq 000100								

- Setting of the MemRd signal (for R-type, sw, beq) depends on the memory design (could have to be 0 or could be a X (don't care))

Branch Instruction Data/Control Flow



Branch Instruction Data/Control Flow



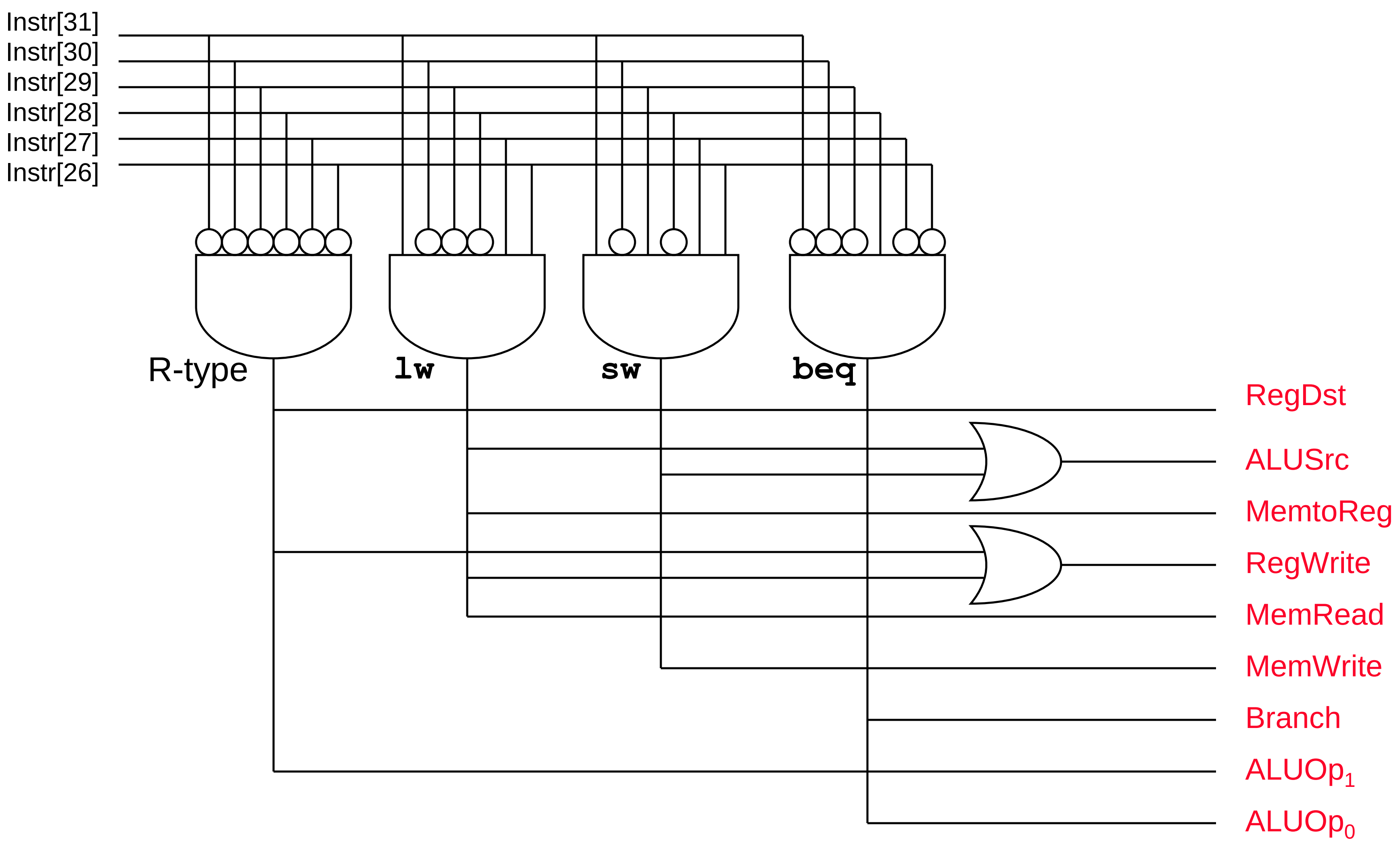
Main Control Unit

Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp
R-type 000000	1	0	0	1	0	0	0	10
lw 100011	0	1	1	1	1	0	0	00
sw 101011	X	1	X	0	0	1	0	00
beq 000100	X	0	X	0	0	0	1	01

- Setting of the MemRd signal (for R-type, sw, beq) depends on the memory design (could have to be 0 or could be a X (don’t care))

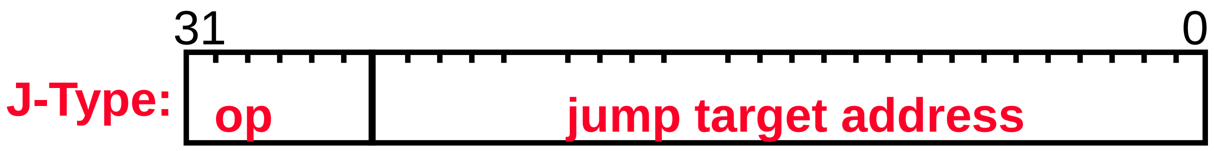
Control Unit Logic

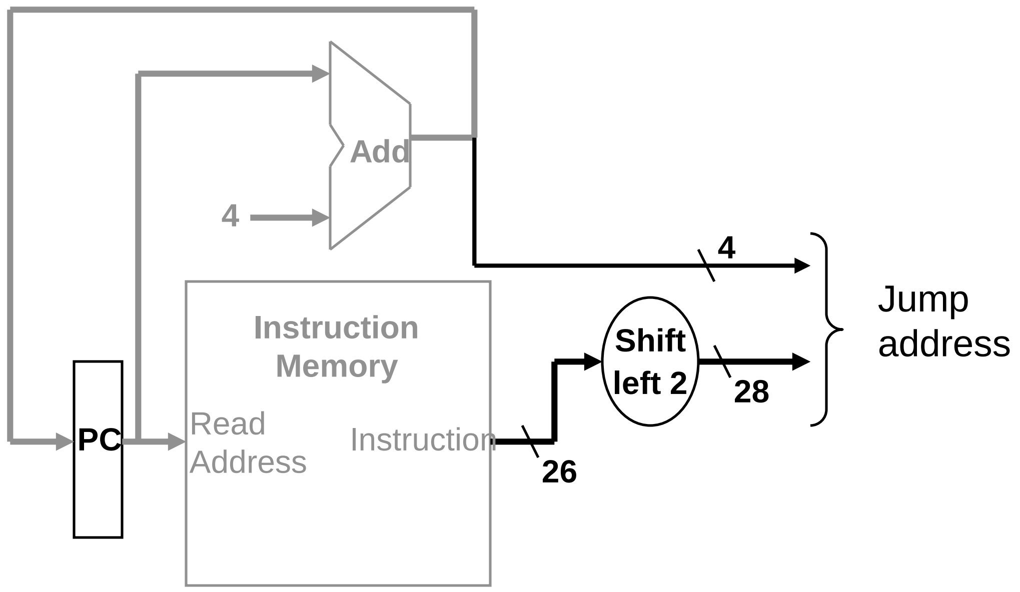
- From the truth table can design the Main Control logic



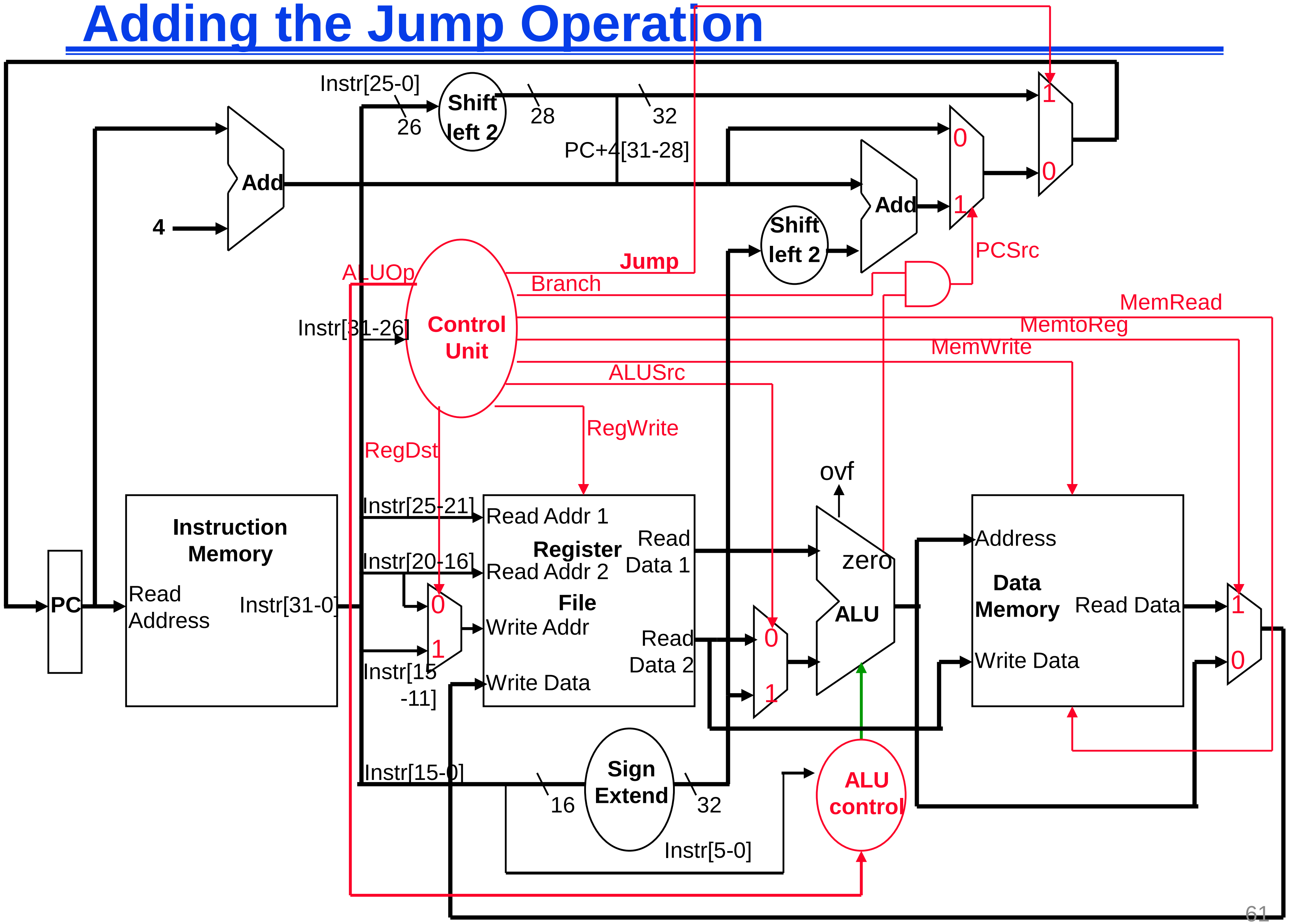
Review: Handling Jump Operations

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

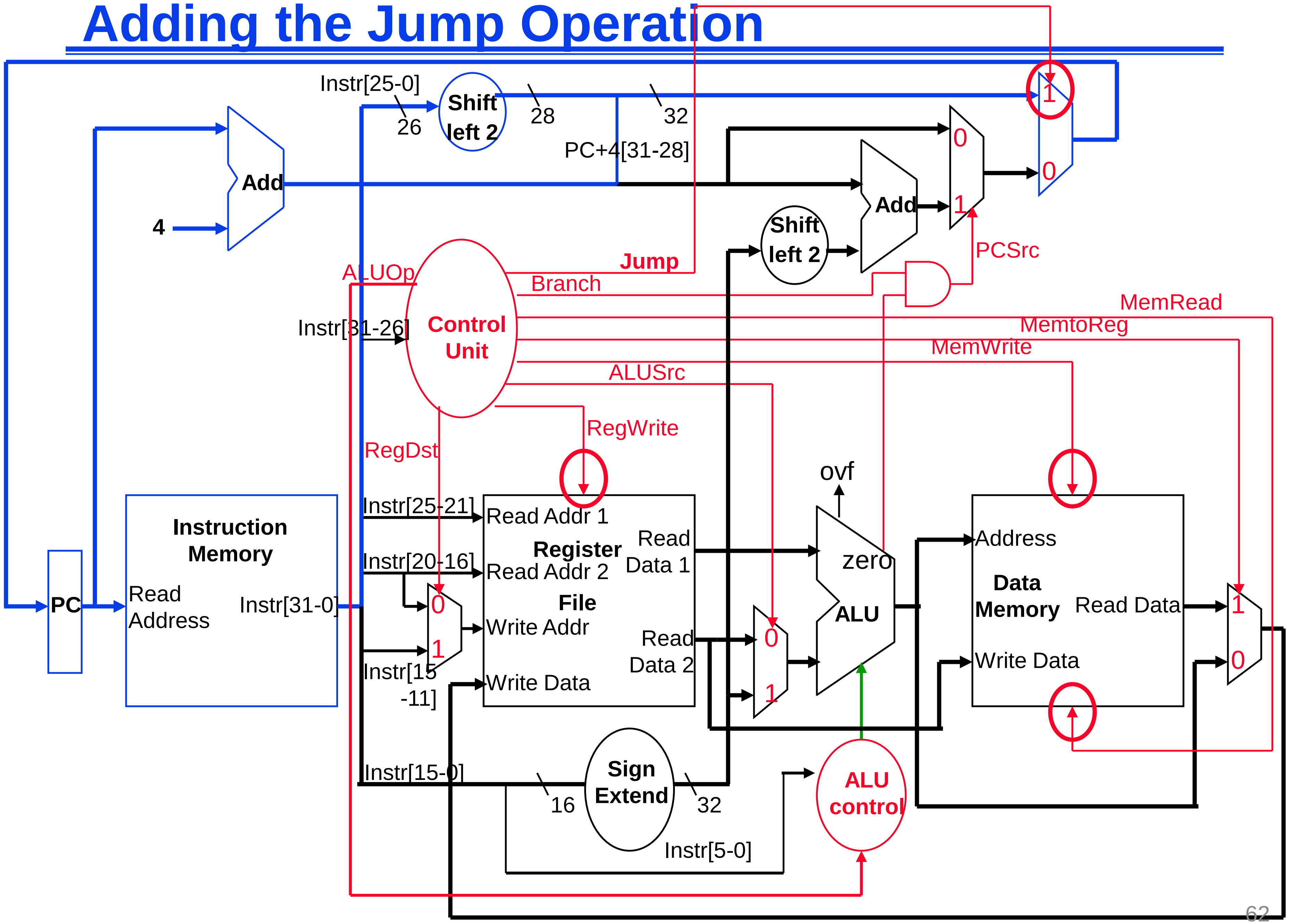




Adding the Jump Operation



Adding the Jump Operation



Main Control Unit

Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp	Jump
R-type 000000									
lw 100011									
sw 101011									
beq 000100									
j 000010									

- Setting of the MemRd signal (for R-type, sw, beq) depends on the memory design

Main Control Unit

Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp	Jump
R-type 000000	1	0	0	1	0	0	0	10	0
lw 100011	0	1	1	1	1	0	0	00	0
sw 101011	X	1	X	0	0	1	0	00	0
beq 000100	X	0	X	0	0	0	1	01	0
j 000010	X	X	X	0	0	0	X	XX	1

- Setting of the MemRd signal (for R-type, sw, beq) depends on the memory design

Single Cycle Implementation Cycle Time

- Unfortunately, though simple, the single cycle approach is not used because it is very slow
- Clock cycle must have the same length for every instruction

- What is the longest (slowest) path (slowest instruction)?

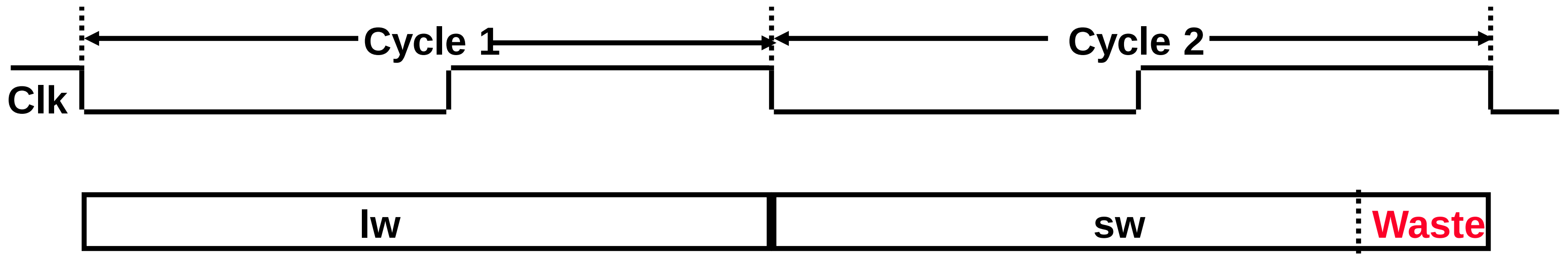
Instruction Critical Paths

- Calculate cycle time assuming negligible delays (for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times) except:
  - Instruction and Data Memory (200 ps)
  - ALU and adders (200 ps)
  - Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
beq	200	100	200			500
jump	200					200

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instr
  - 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

- It is simple and easy to understand