1. specified instruction list (R, I, J-type)
(1) Functions of instructions
(2) Format
(3) control signals

SE230
Computer Organization

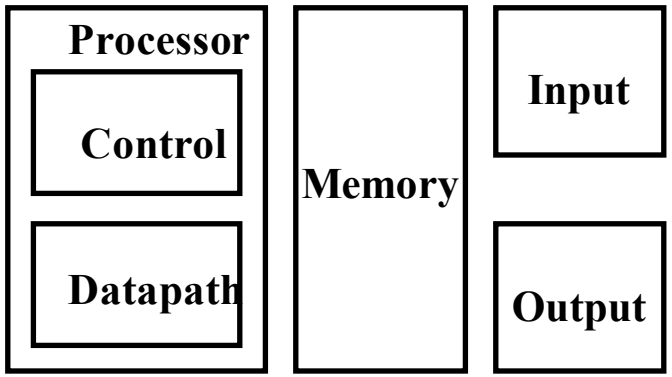
Lecture 04: Single Cycle Processor

Liang Yanyan

澳門科技大學
Macau of University of Science and Technology

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer



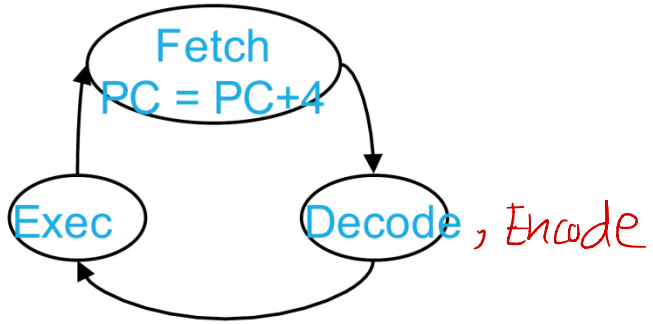
- Today's Topic: Design a Single Cycle Processor 1 instruction/cycle

The Big Picture: The Performance Perspective

- Performance of a machine is determined by:
  - Instruction count
  - Clock cycle time
  - Clock cycles per instruction (CPI)
- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- Single cycle processor:
  - Advantage: One clock cycle per instruction
  - Disadvantage: long cycle time

The Processor

- 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

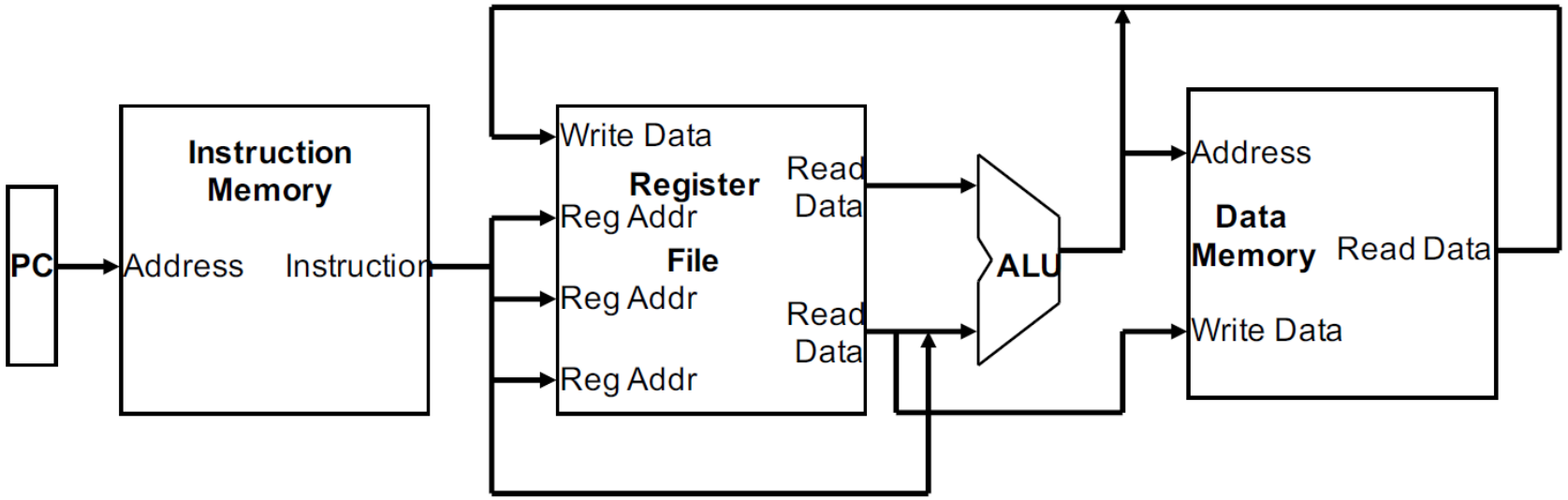


How to Design a Processor: step-by-step

1. Analyze instruction set => datapath requirements.
   - the meaning of each instruction
   - datapath must include storage element for ISA registers
     - E.g. 32 registers for MIPS
   - datapath must support each register transfer
2. Select set of datapath components.
3. Assemble datapath meeting the requirements.
4. Analyze implementation of each instruction to determine setting of control signals.
5. Assemble the control logic.

Abstract Implementation View

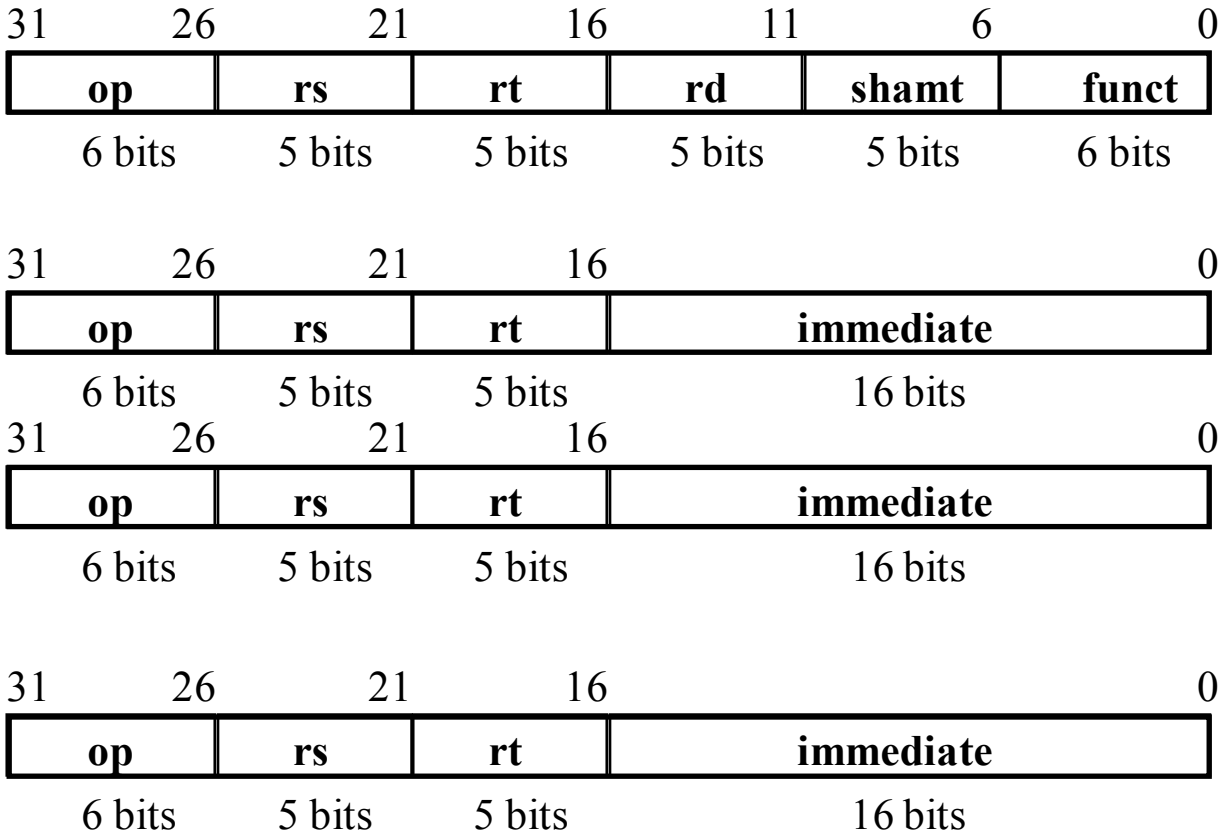
- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)



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

Step 1a: The MIPS subset for today

- ADD and SUB
  - add rd, rs, rt
  - sub rd, rs, rt
- ADDI Immediate:
  - addi rt, rs, imm16
- LOAD and STORE word
  - lw rt, imm16(rs)
  - sw rt, imm16(rs)
- BRANCH:
  - beq rs, rt, imm16



What does it mean?

inst	Register Transfers	Update program counter
ADD	R[rd] <= R[rs] + R[rt];	PC <– PC + 4
SUB	R[rd] <= R[rs] – R[rt];	PC <– PC + 4
ADDI	R[rt] <= R[rs] + sign_ext(imm16);	PC <– PC + 4
LOAD	R[rt] <= MEM[R[rs] + sign_ext(imm16)];	PC <– PC + 4
STORE	MEM[R[rs] + sign_ext(imm16)] <= R[rt];	PC <– PC + 4
BEQ	if (R[rs] == R[rt]) then PC <= PC + 4 + sign_ext(imm16) × 4 else PC <= PC + 4	

Step 1b: Datapath requirements

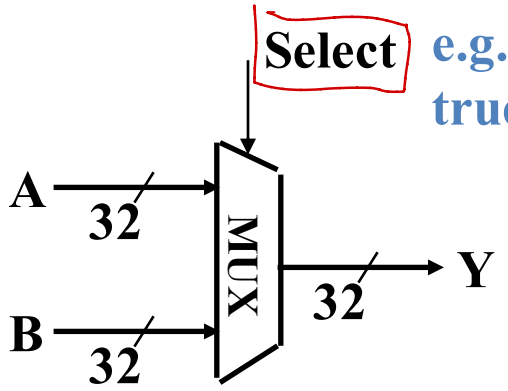
- Ability to perform computation
  - E.g. add & sub
- Storage element
  - Store instruction & data
- Store registers (32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- Program Counter (PC)
  - Point to the instruction to be loaded
  - Add 4 or extended immediate to PC
    - PC <= PC + 4
    - PC <= PC + 4 + [sign_ext(imm16)] $\times$ 4
- Extender
  - Sign extend, e.g. lw t1, imm16(t1)
  - Zero extend, e.g. addiu t1, t4, 5

Step 2: Components of the Datapath

- Computation: Arithmetic Logic Unit (ALU)
- Store instruction and data: Memory
- Store register: Register file
- Choose different PCs: Multiplexer (MUX)
  - if … else …
- Sign or Zero extend: Extender

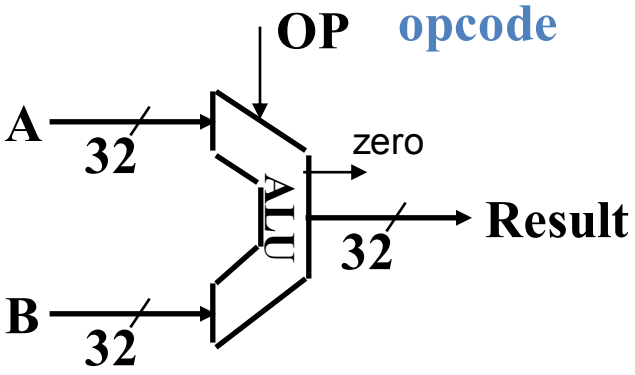
Combinational Logic Elements (Basic Building Blocks)

- MUX



e.g. condition of if statement, true, false

- ALU



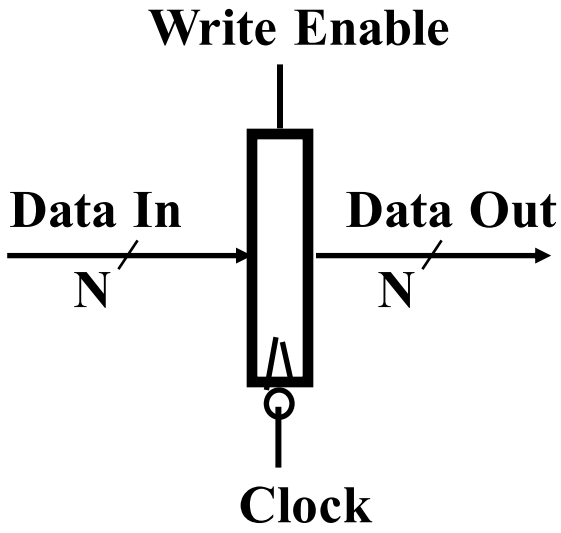
opcode

zero used to indicate whether ALU output is 0 or not.

zero = 1 if ALU output is 0, otherwise, Zero = 0.

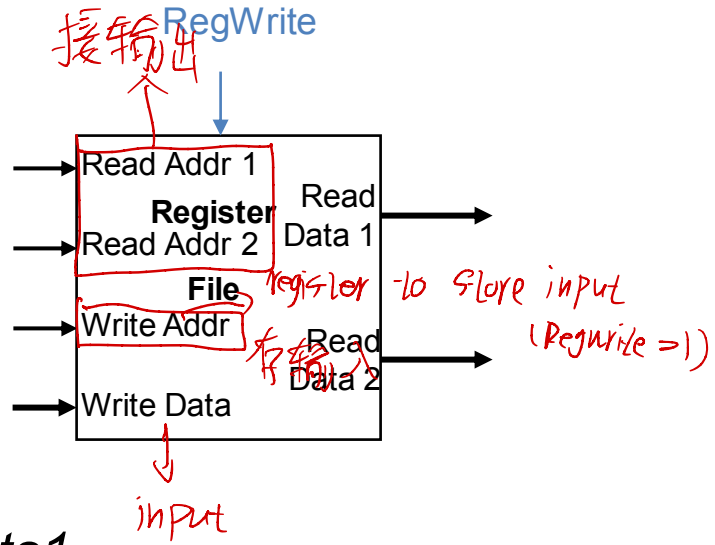
Storage Element: Register (Basic Building Block)

- Register
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable
  - Write Enable:
    - 0: register will NOT be updated
    - 1: content of the register will be updated at the rising edge of clock signal



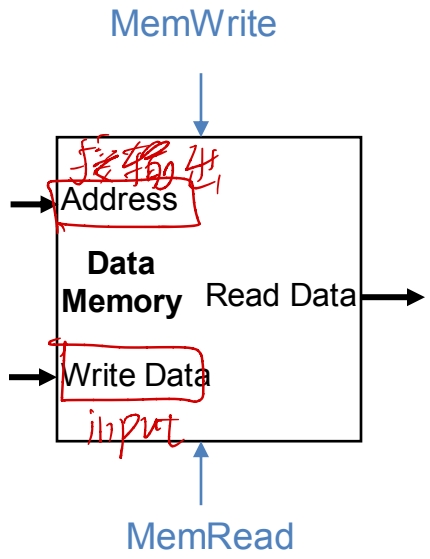
Storage Element: Register File

- Consists of 32 registers (locations):
  - Two 32-bit outputs:
    ReadData1 and *ReadData2.*
  - One 32-bit input: *WriteData.*
- *RegWrite*
  - Enable writing data to Register File or not.
- Register is selected by:
  - *ReadAddr1* (5 bits) selects the register to put on *ReadData1.*
  - *ReadAddr2* (5 bits) selects the register to put on *ReadData2.*
  - *WriteAddr* (5 bits) selects the register (location) to store data from *WriteData* when *RegWrite* is 1.



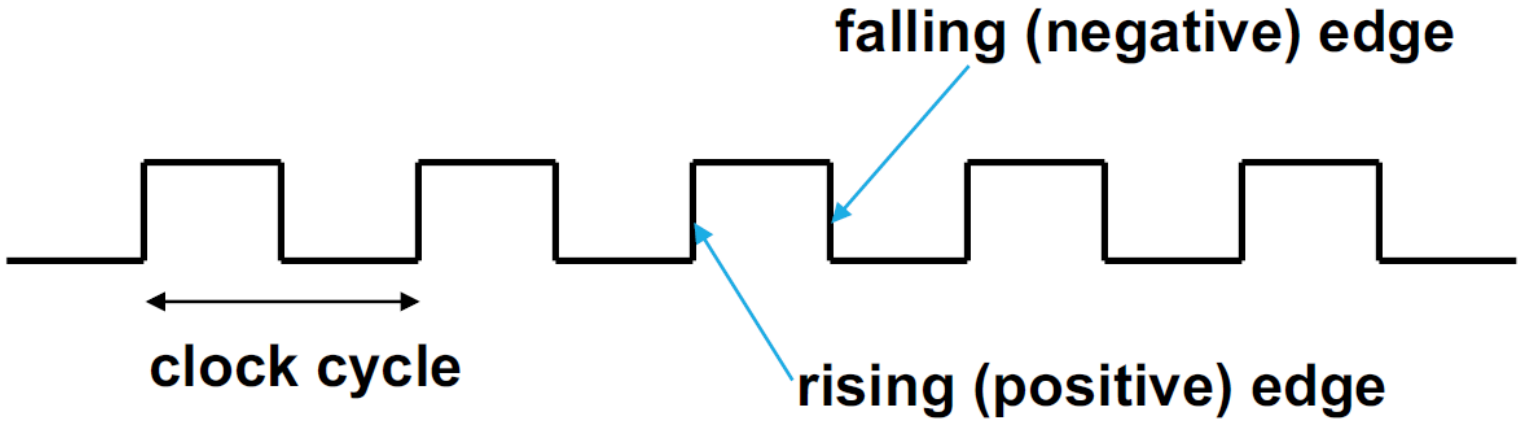
Storage Element: Memory

- Memory
  - One data input: *WriteData*.
  - One data output: *ReadData*.
- Memory word is selected by *Address*.
  - Address selects the word to put on *ReadData*.
  - When *MemWrite* == 1: *Address* selects the memory location to store *WriteData*.
  - When *MemRead* == 1: *Address* selects the memory location to put on *ReadData*.
- Number of address bits depends on number of memory locations.



Clocking Methodologies

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



clock rate = 1/(clock cycle)

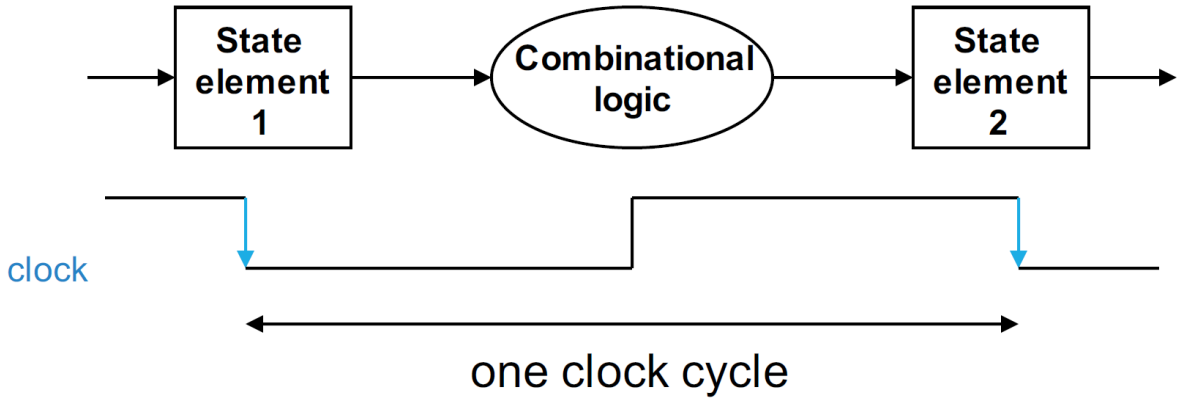
e.g.,10 nsec clock cycle = 100 MHz clock rate

1 nsec clock cycle = 1 GHz clock rate

- State element design choices
  - master-slave and edge-triggered flipflops: A clocking methodology defines when signals can be read and written – would NOT 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
  - send values through some combinational logic
  - write results to one or more state elements



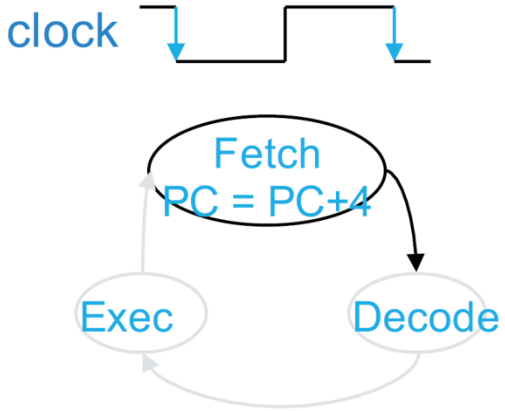
- Assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - write occurs only when both the write control is asserted and the clock edge occurs

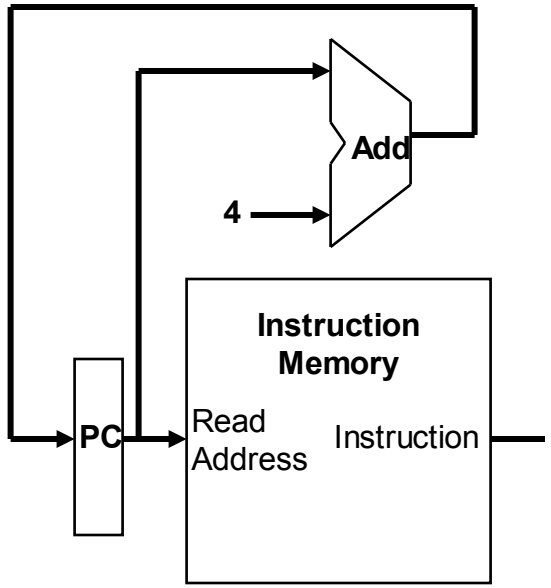
Step 3: Datapath Assembly

- a. Instruction Fetch
- b. Instruction execution
  - Read Operands
  - Execute Operation
    - Register e.g. R[rd] <= R[rs] op R[rt]
    - Immediate e.g. R[rt] <= R[rs] + sign_ext(Imm16)
    - Load e.g. R[rt] <= MEM[R[rs] + sign_ext(Imm16)];
    - Store e.g. MEM[R[rs] + sign_ext(Imm16)] <= R[rt];
    - Branch e.g. PC <= PC + 4 + [sign_ext(Imm16)] $\times$ 4
- Add datapath step by step

3a: Data path for instruction fetch

- Fetch the instructions from Instruction Memory.
  - An address input: *ReadAddress*.
  - A data output: *Instruction*.
  - *ReadAddress* selects the instruction to put on *Instruction*.

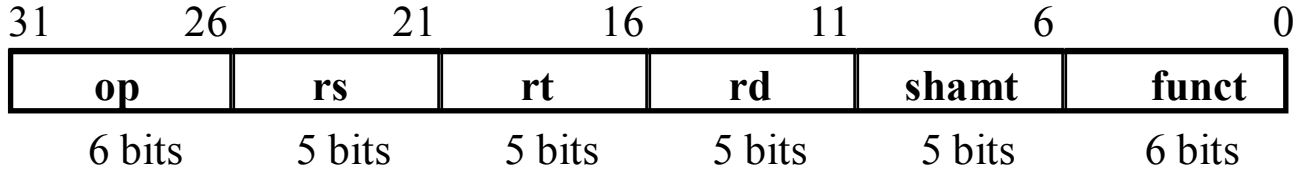


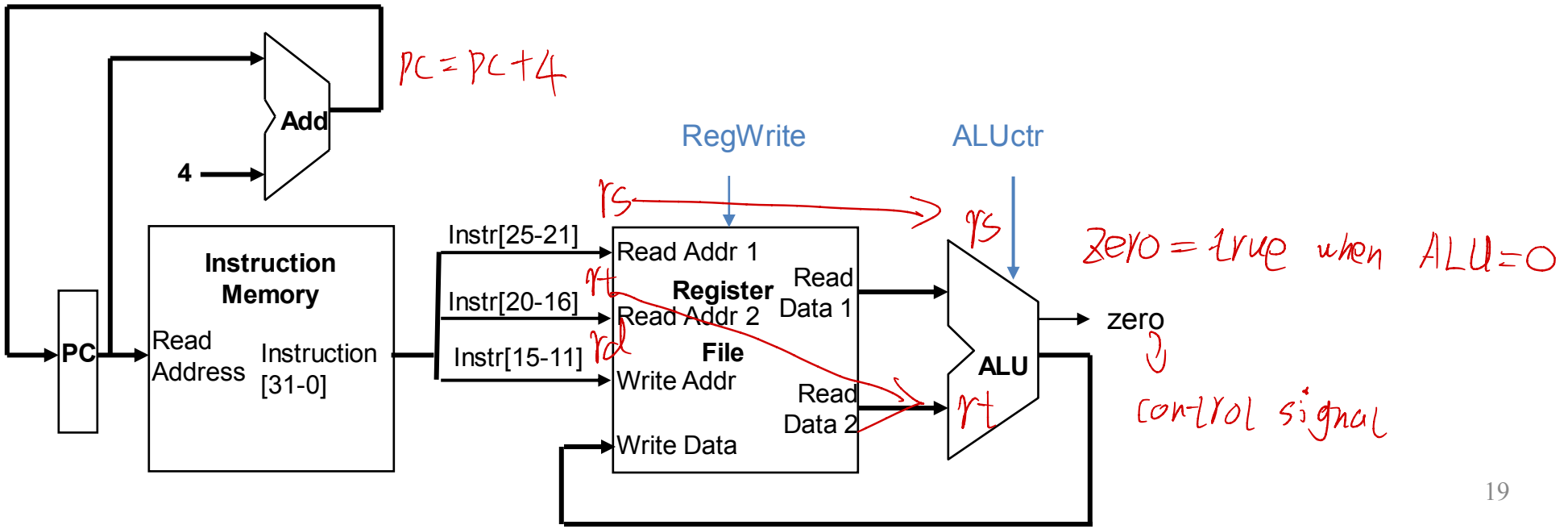


- Update the program counter (PC):
  - Sequential Code: PC <= PC + 4
  - Branch and Jump: PC <= “something else”
- Connect PC to Instruction Memory.

3b: Execute operation: Register

- R[rd] <= R[rs] op R[rt] Example: add rd, rs, rt rd = rs + rt
  - Register is selected by instruction's rs, rt, and rd fields.
  - Data are sent from Register File to ALU for computation.
  - Result is written back to Register File from ALU.





3c: Execute operation: Immediate

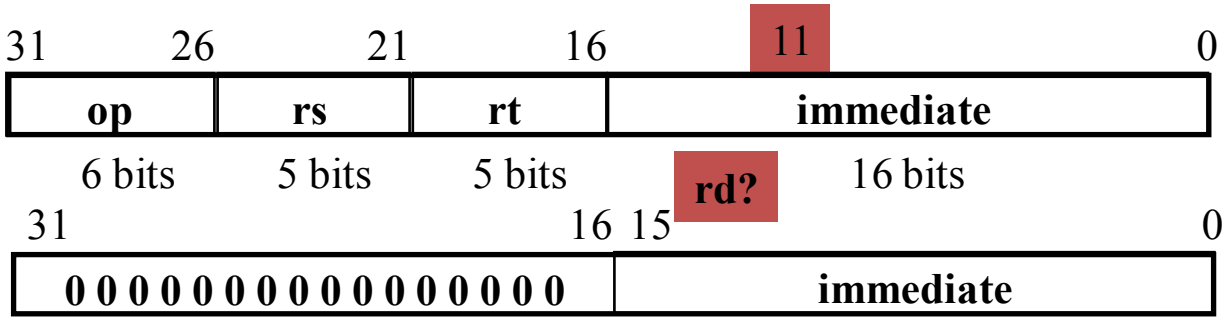
rt = rs + imm

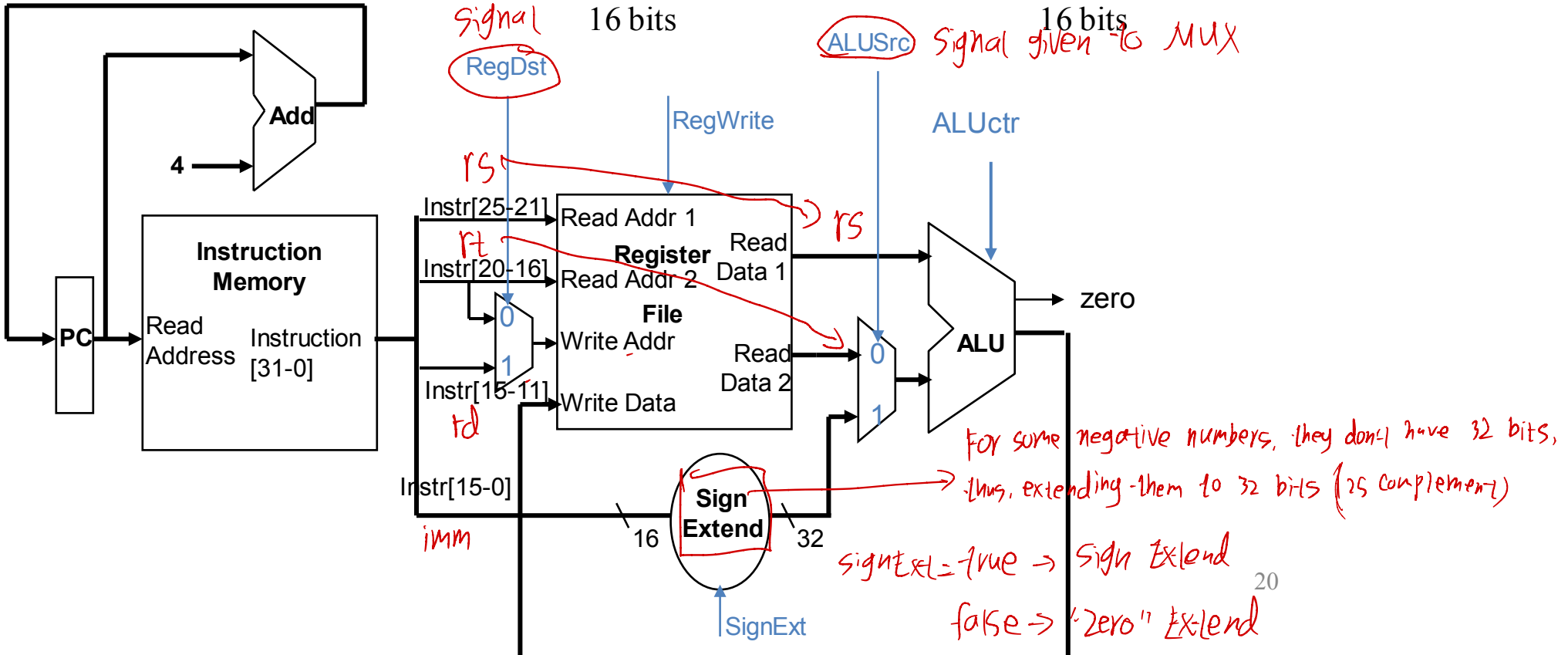
- R[rt] <= R[rs] op SignExt[imm16]] Example: addi t0, t1, -10

Changes compared with 3b:

1. ALU input can come from register or immediate => add sign extender and Mux

2. Destination can be Rd or Rt





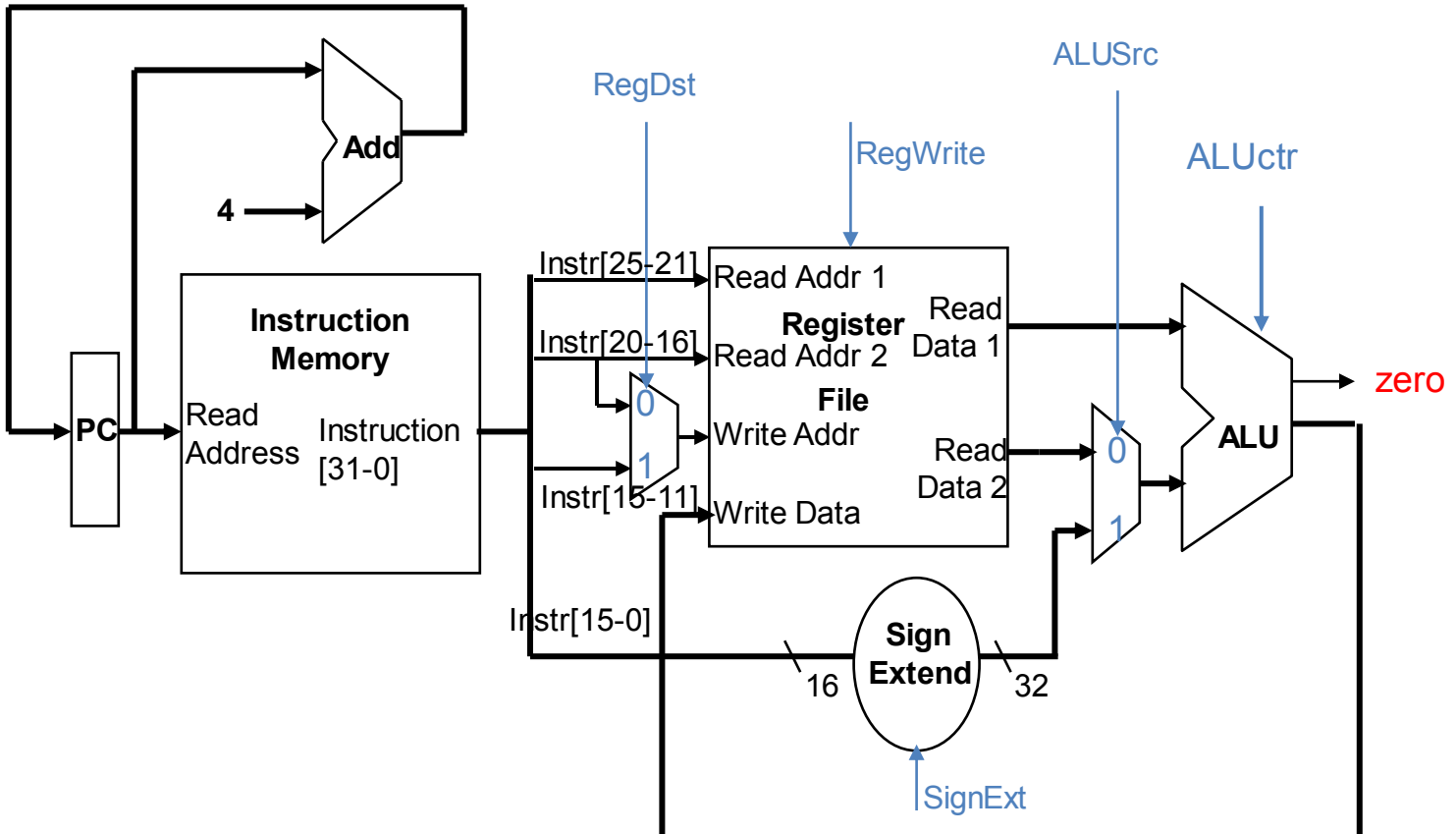
Consider the slt Instruction

- Remember the R format instruction `slt`

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

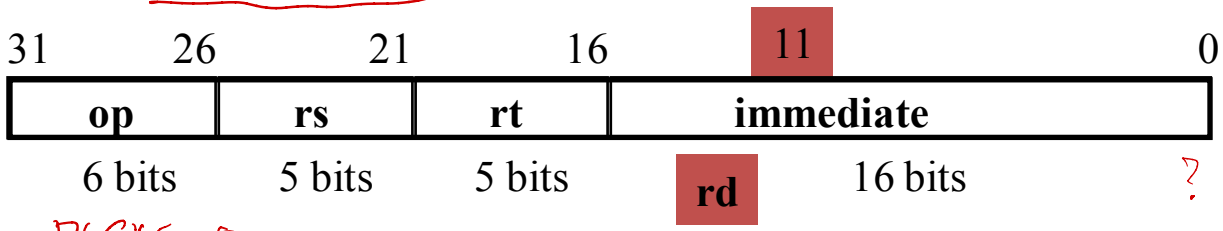
MSB(Vs-Vt) 最高位
executed in ALU

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



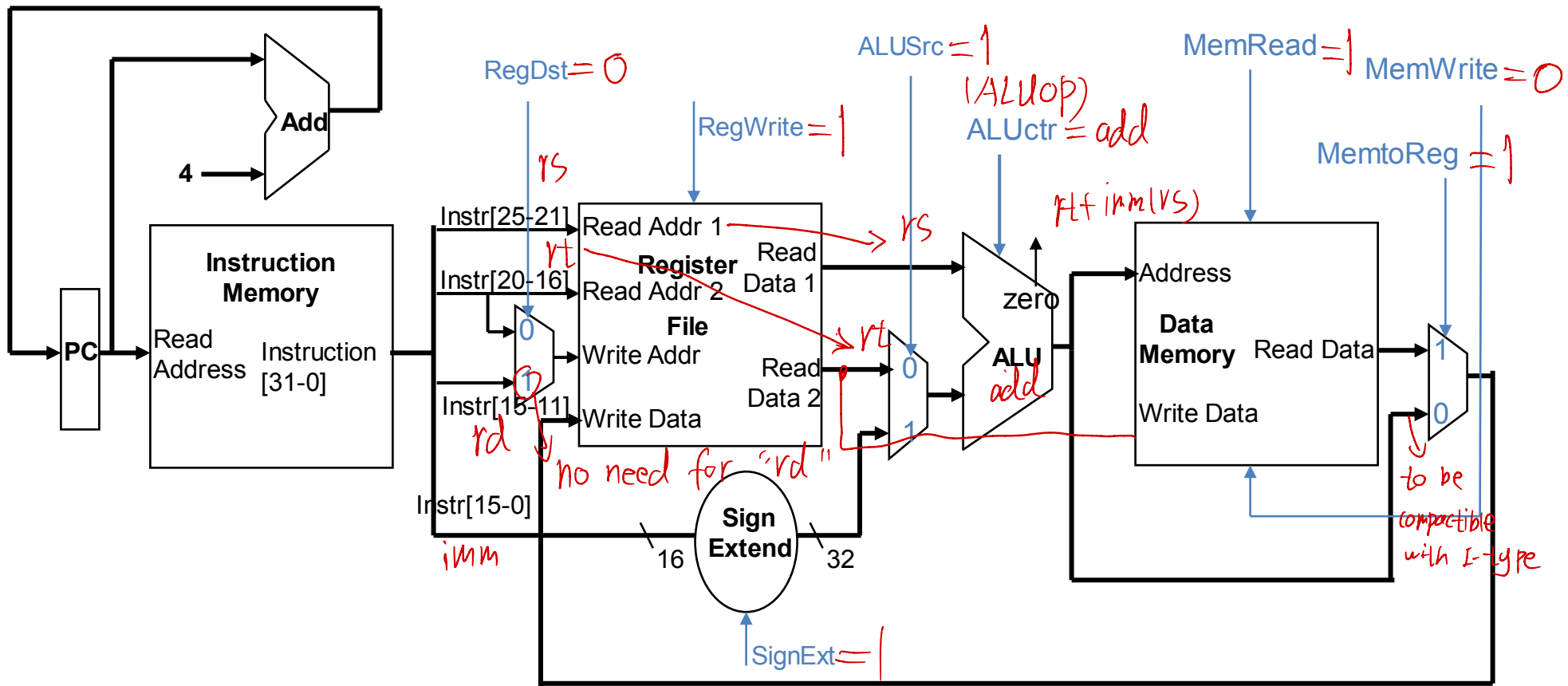
3d: Execute operation: Load

- R[rt] <= Mem[R[rs] + SignExt[imm16]] Example: lw rt, imm16(rs)



Changes compared with 3c:

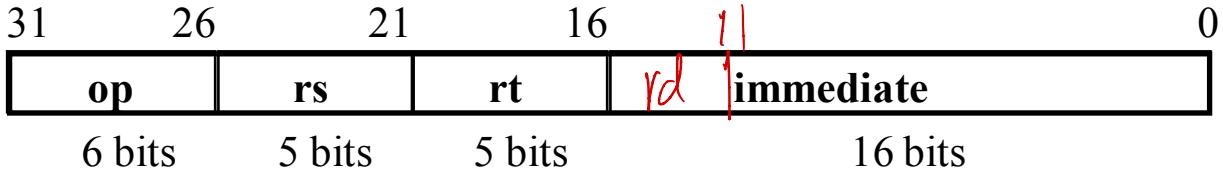
Data to Register File can come from ALU result or memory output => add memory and Mux



3e: Execute operation: Store

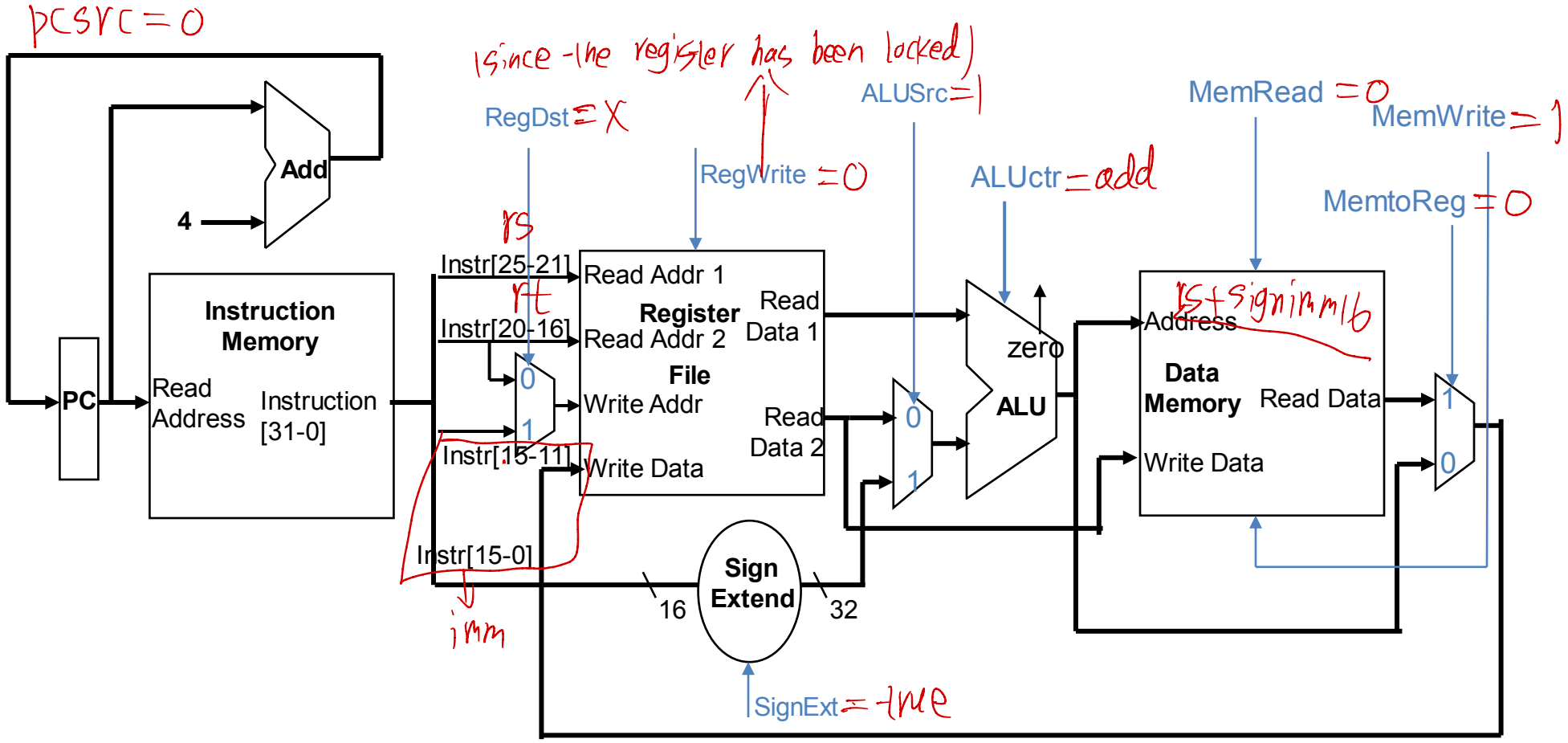
mem[rs+signimm16] = R[rt]

- Mem[R[rs] + SignExt[imm16]] <= R[rt]] Example: sw rt, imm16(rs)

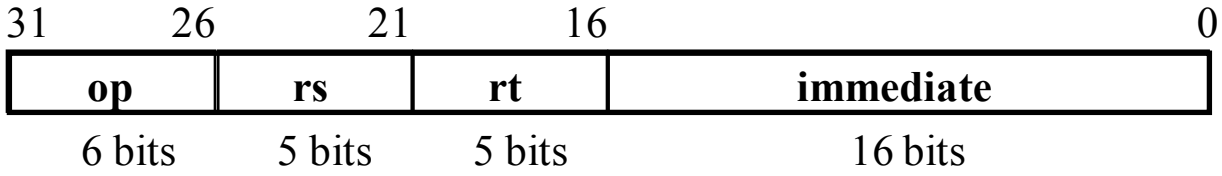


Changes compared with 3d:

1. Can store data from register to memory => add datapath



3f: The Branch Instruction



- beq rs, rt, Label
  - mem[PC] Fetch the instruction from memory
  - EQUAL <= if (R[rs] == R[rt]) ? Calculate the branch condition
    - EQUAL <= 1 if they are equal, EQUAL <= 0 if they are not equal
  - if (EQUAL eq 1) Calculate the next instruction's address
    - PC <- PC + 4 + (SignExt(imm16) x 4)
  - else
    - PC <- PC + 4

3f: Execute operation: Branch

beq rs, rt, Label

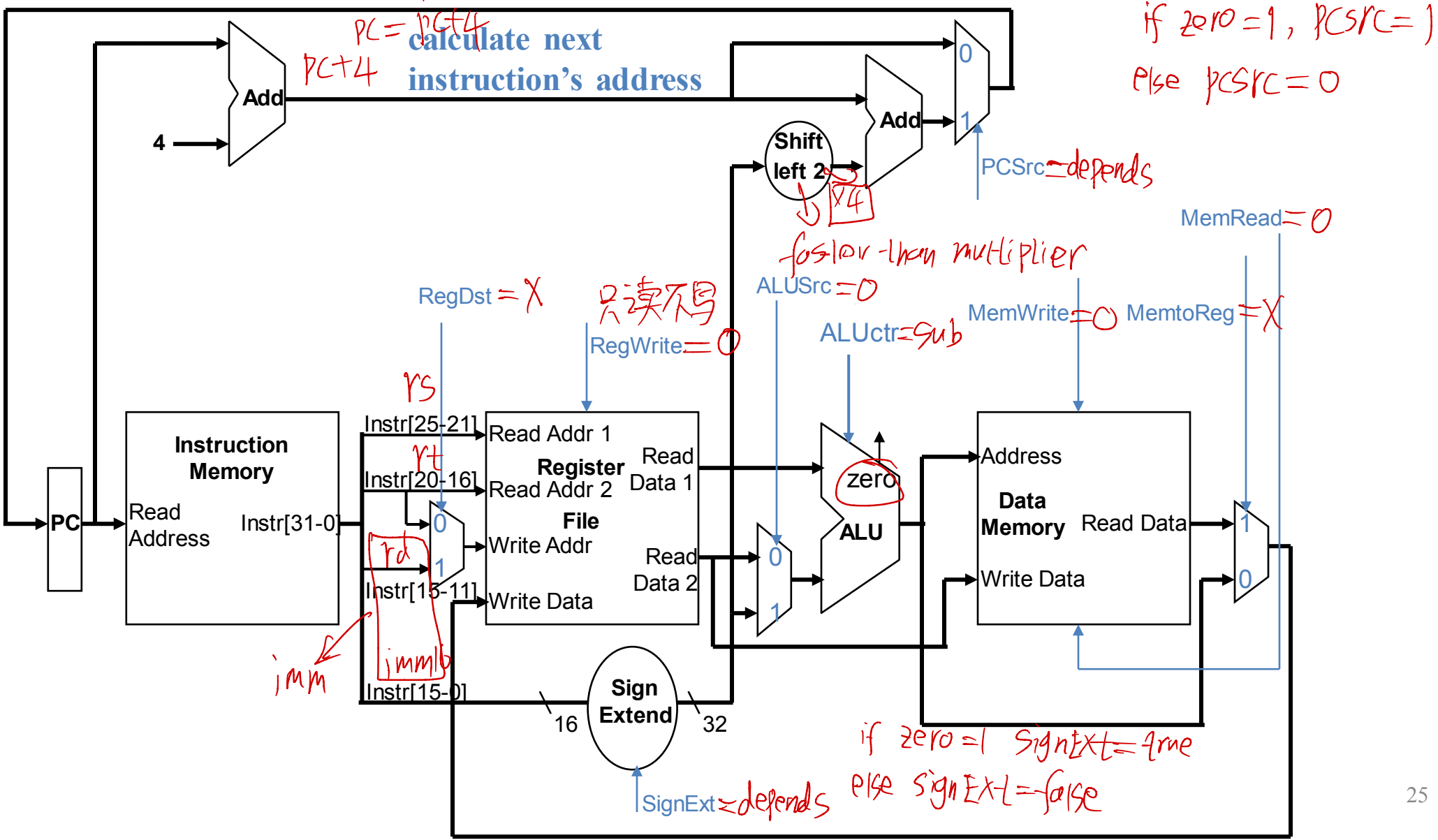
rs = rt ⇔ sub rs rt

pc = pc+4+imm*4 or pc = pc+4

1. New PC can be (PC+4) or (PC+4+4ximm16), => add adder and Mux

2. Check equal, adjust PCsrc based on Zero

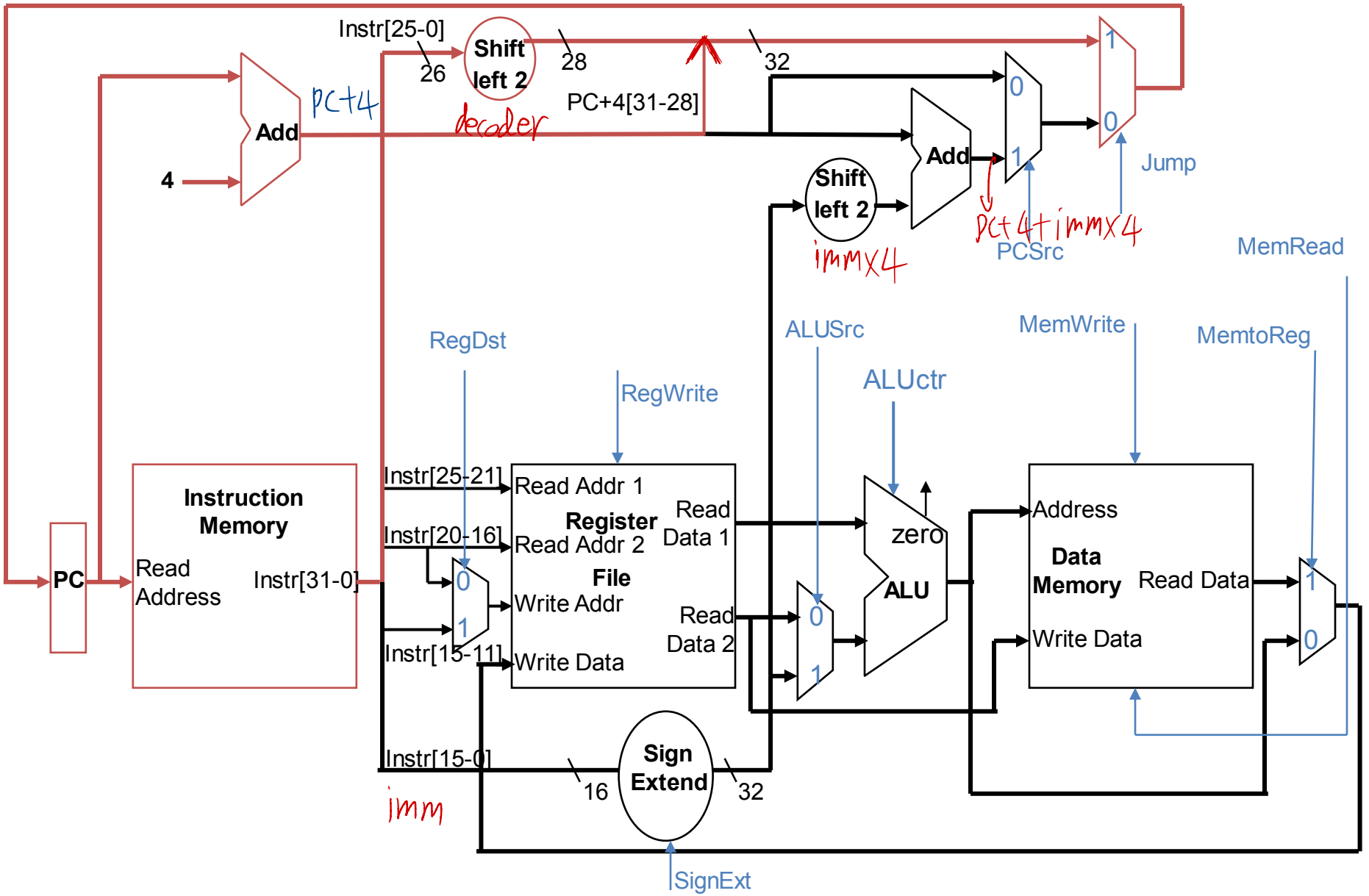
if zero=1, PCsrc=1
else PCsrc=0



3g: Execute operation: Jump

op	26 bit offset value

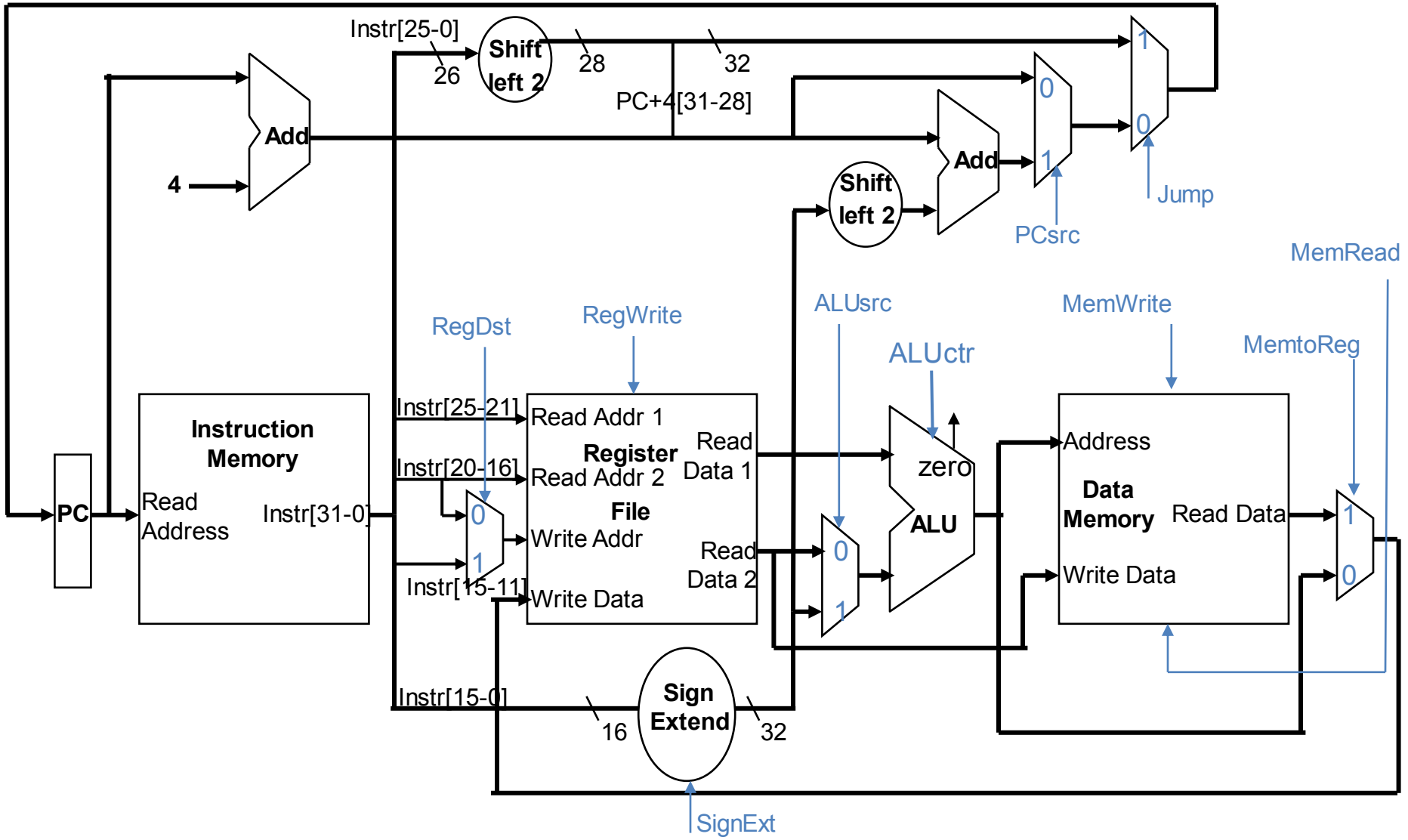
Address of target inst. (target PC) = (upper 4 bits of (PC + 4) : offset) << 2



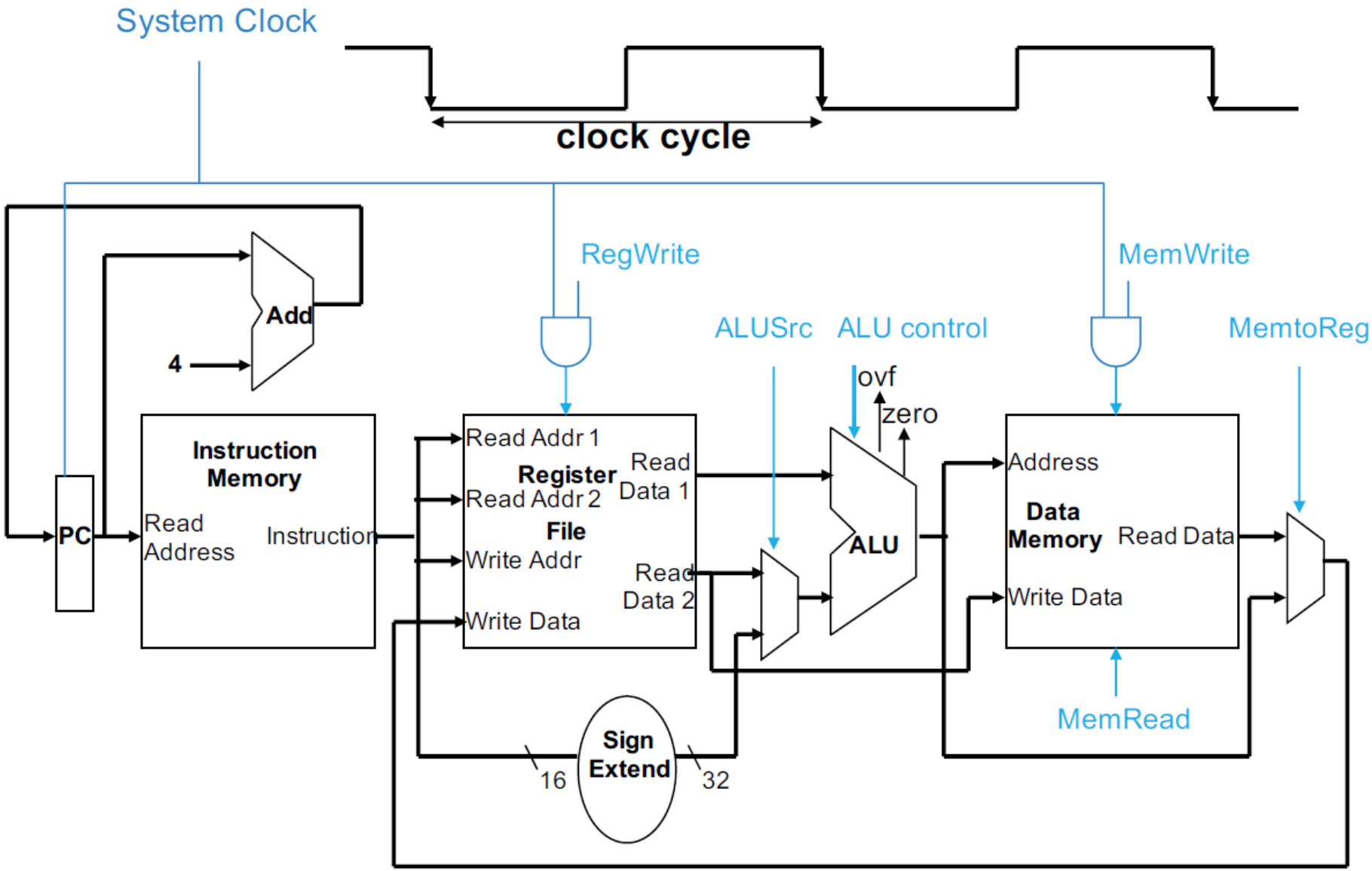
Creating a Single Datapath from the Parts

- Assemble the datapath elements
- Fetch 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

Putting it All Together: A Single Cycle Datapath



Clock Distribution



What have been done in one clock cycle for lw?

- Instruction fetch
- Instruction decode
- Load data from register Rs
- Select immediate (imm16) and put to ALU
- Calculate memory address
  - R[rs] + imm16
- Load data from memory
- Write data to register R[rt]
- Calculate next PC

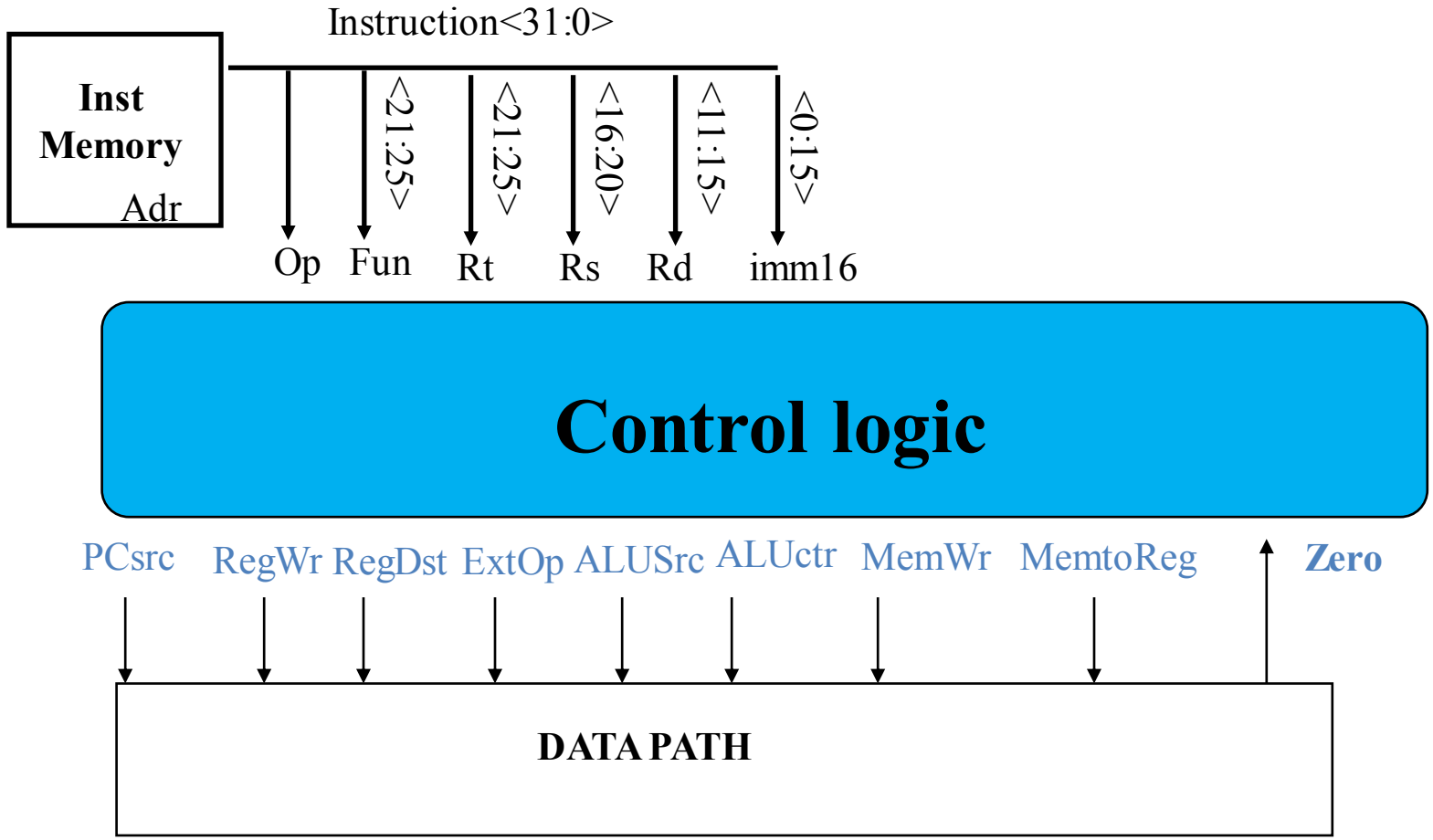
Long cycle time!!

Step 4: Designing control signals

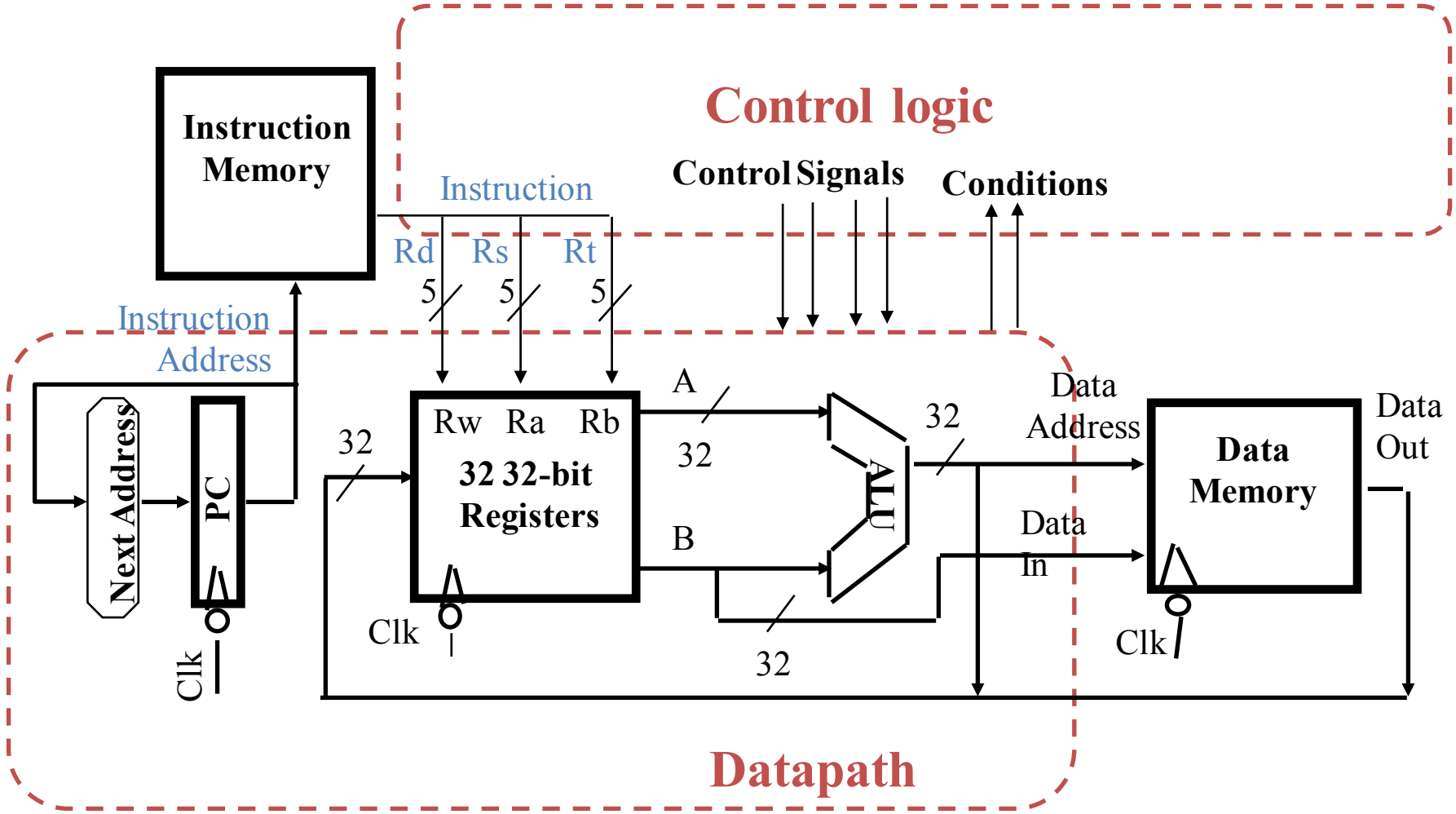
- Control signals are used to control the datapath to perform necessary operations.
- Need to decide what control signals are required such that we can fully control our processor to perform calculation.
- Example: add t0, t1, t2
  - Need the ALU to perform addition.
  - As a result, we need a control signal “ALUctr” to control the ALU to perform necessary operation.
- Will be addressed in next lecture.

Step 5: Adding control logic

- Control logic is a component used to determine the value of control signals based on instruction type.
  - E.g. ALUctr, RegDst, ……



An abstract view of the processor

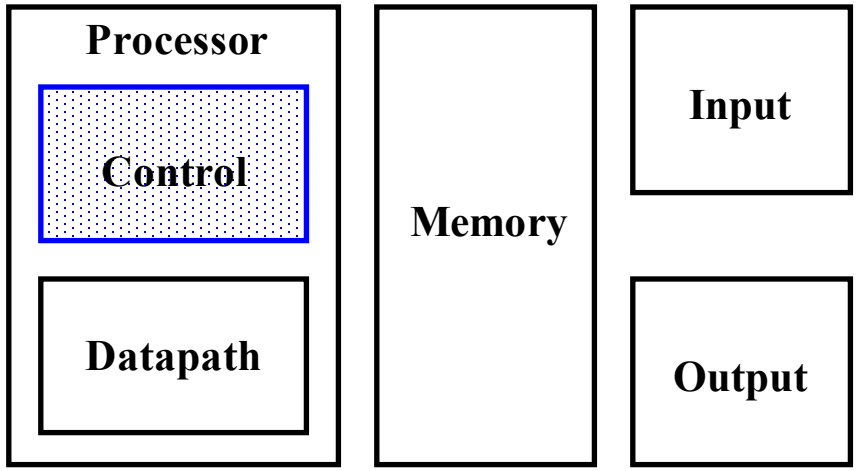


Summary

- 5 steps to design a processor
  - 1. Analyze instruction set => datapath requirements
  - 2. Select set of datapath components
  - 3. Assemble datapath meeting the requirements
  - 4. Analyze implementation of each instruction to determine setting of control signals
  - 5. Assemble the control logic
- MIPS makes it easier
  - Instructions same size
  - Source registers always in same place, R[rt], R[rs]
  - Immediates same size
  - ALU input data always come from registers/immediates
- Single cycle datapath => CPI=1, Cycle time => long
- Next time: implementing control logic.

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer



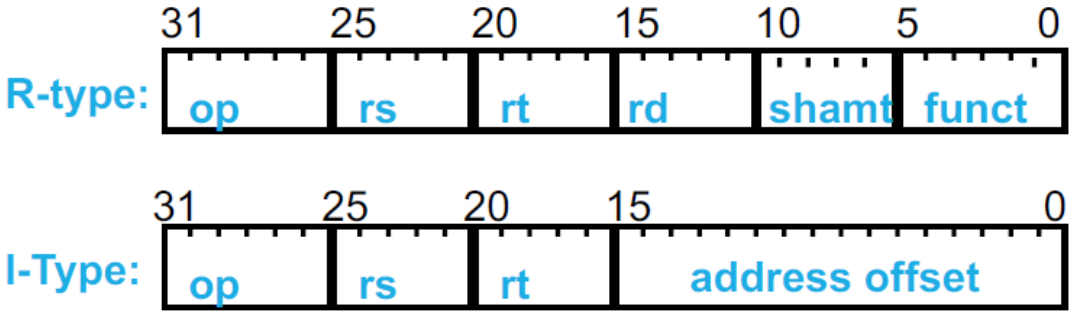
- Today's Topic: Designing the Control for the Single Cycle Datapath

Step 4: Designing control signals

- Control signals are used to control the datapath to perform necessary operations.
- Need to decide what control signals are required such that we can fully control our processor to perform calculation.
- Example: add t0, t1, t2
  - Need the ALU to perform addition.
  - As a result, we need a control signal “ALUctr” to control the ALU to perform necessary operation.

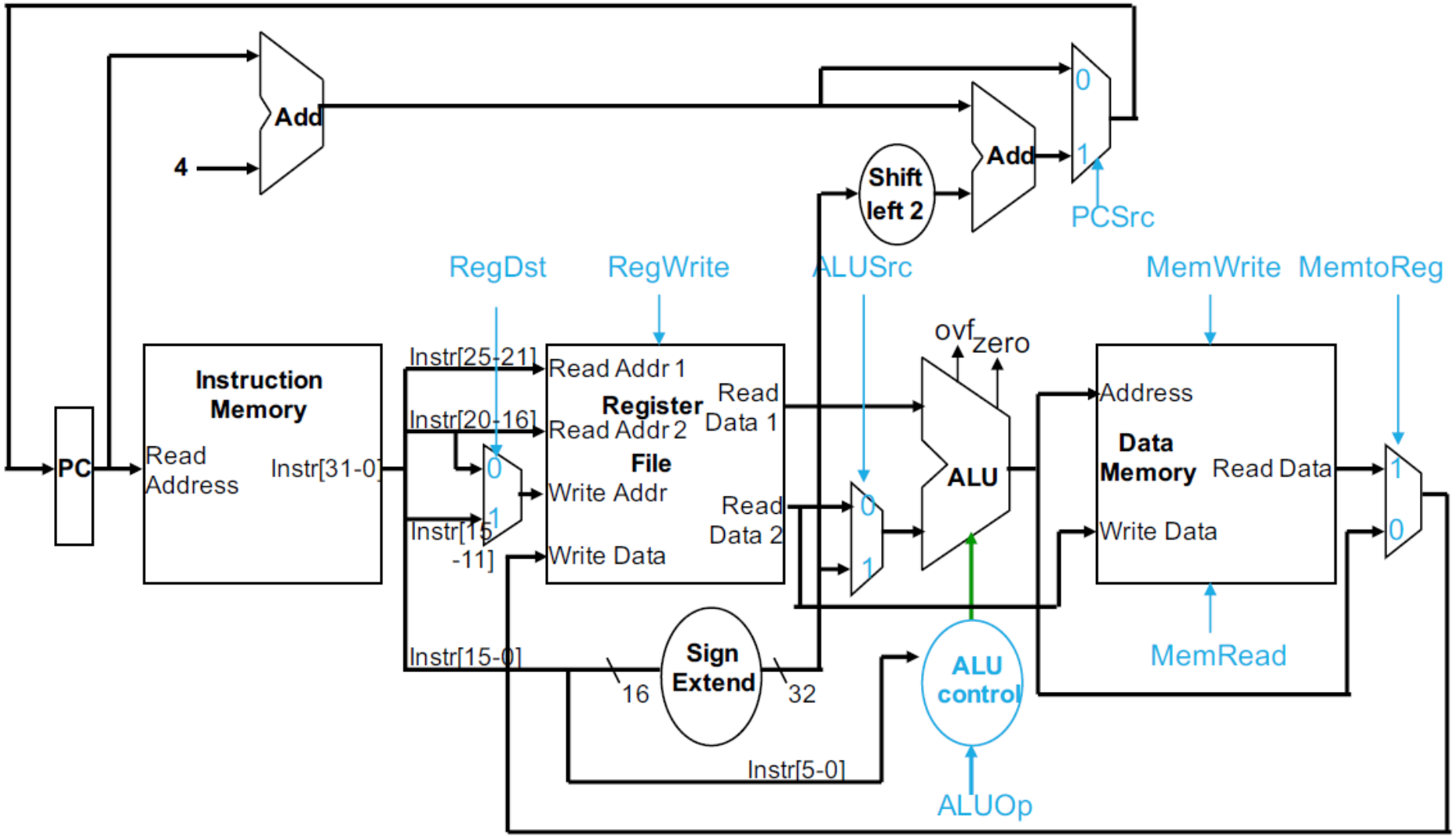
Adding the Control

- 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

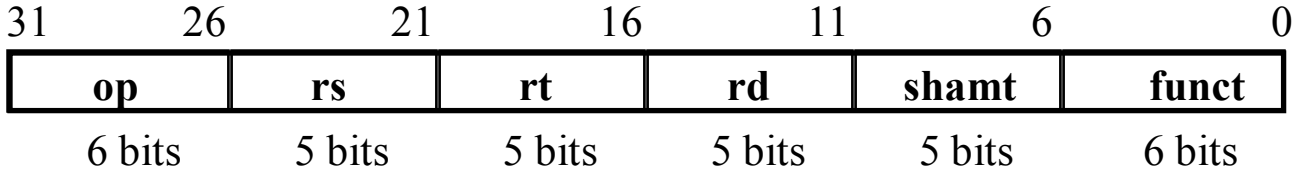
(Almost) Complete Single Cycle Datapath



Meaning of the Control Signals

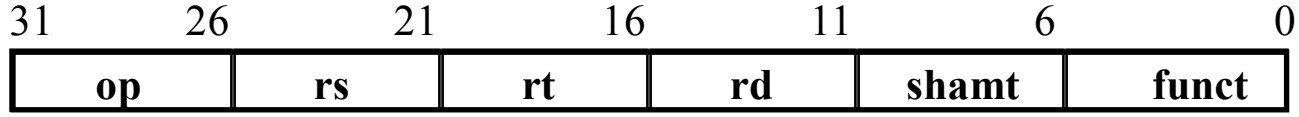
- RegDst: Register destination. 0 → rt, 1 → rd.
- RegWrite: Write register file or not. 0 → no, 1 → yes.
- ALUctr: Operation of ALU, “add”, “sub”, “or”, etc.
- ALUsrc: Select data for ALU. 0 → Register file, 1 → Immediate.
- MemWrite: Write data memory or not. 0 → no, 1 → yes.
- MemRead: Read data memory or not. 0 → no, 1 → yes.
- MemtoReg: Select data for writing register file. 0 → ALU, 1 → data memory.
- PCsrc: Branch or not. 0 → no, 1 → yes.
- Jump: Jump or not. 0 → no, 1 → yes.
- SignExt: Perform sign or zero extend. “sign” or “zero”.

Steps in `add` instruction



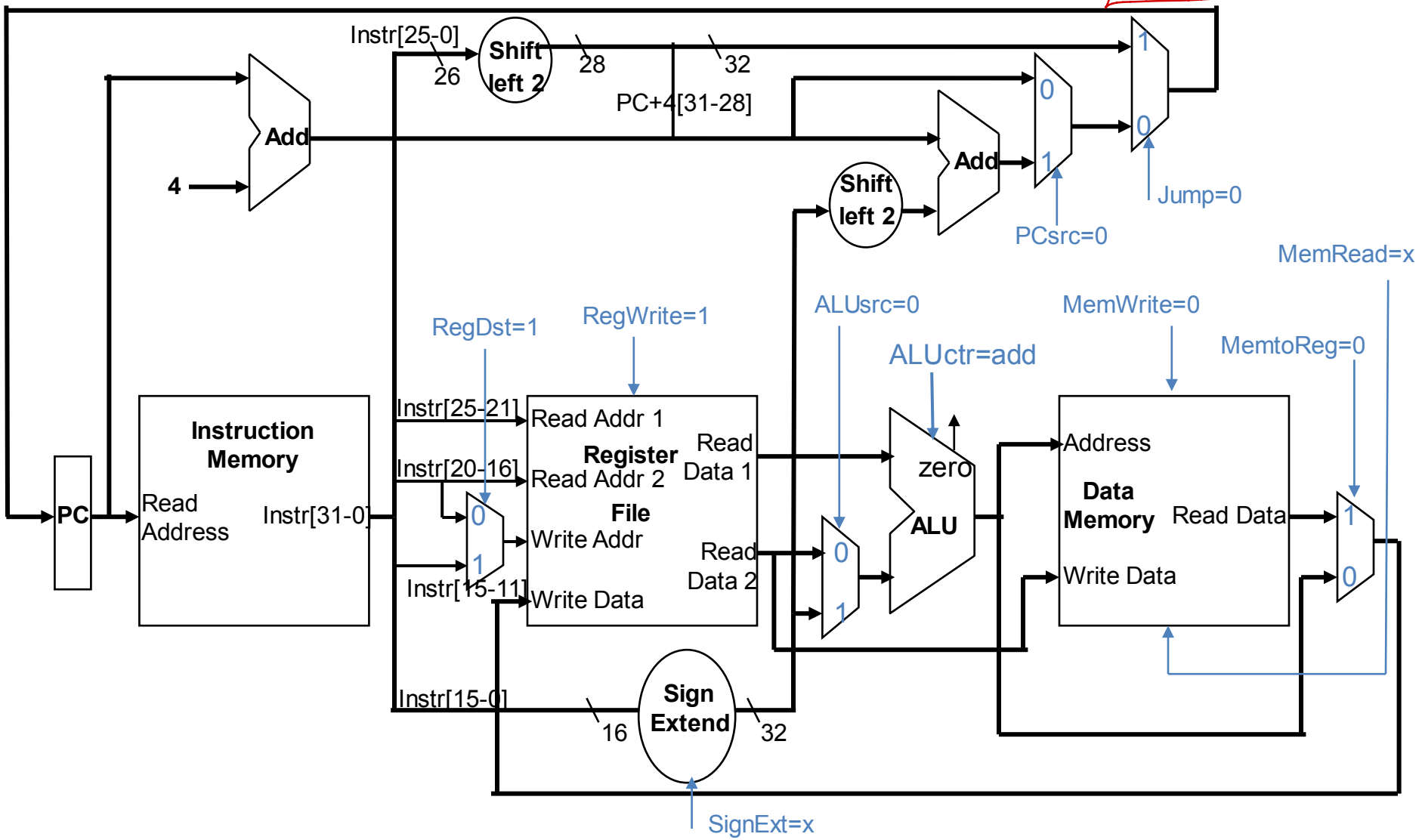
- add rd, rs, rt
  - mem[PC] Fetch the instruction from memory
  - R[rd] <- R[rs] + R[rt] The actual operation
  - PC <- PC + 4 Calculate the next instruction's address

The Single Cycle Datapath during `add`

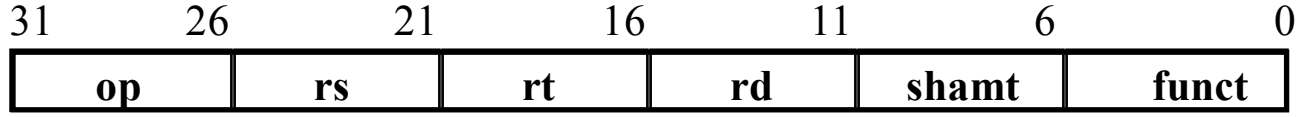


- R[rd] <- R[rs] + R[rt]

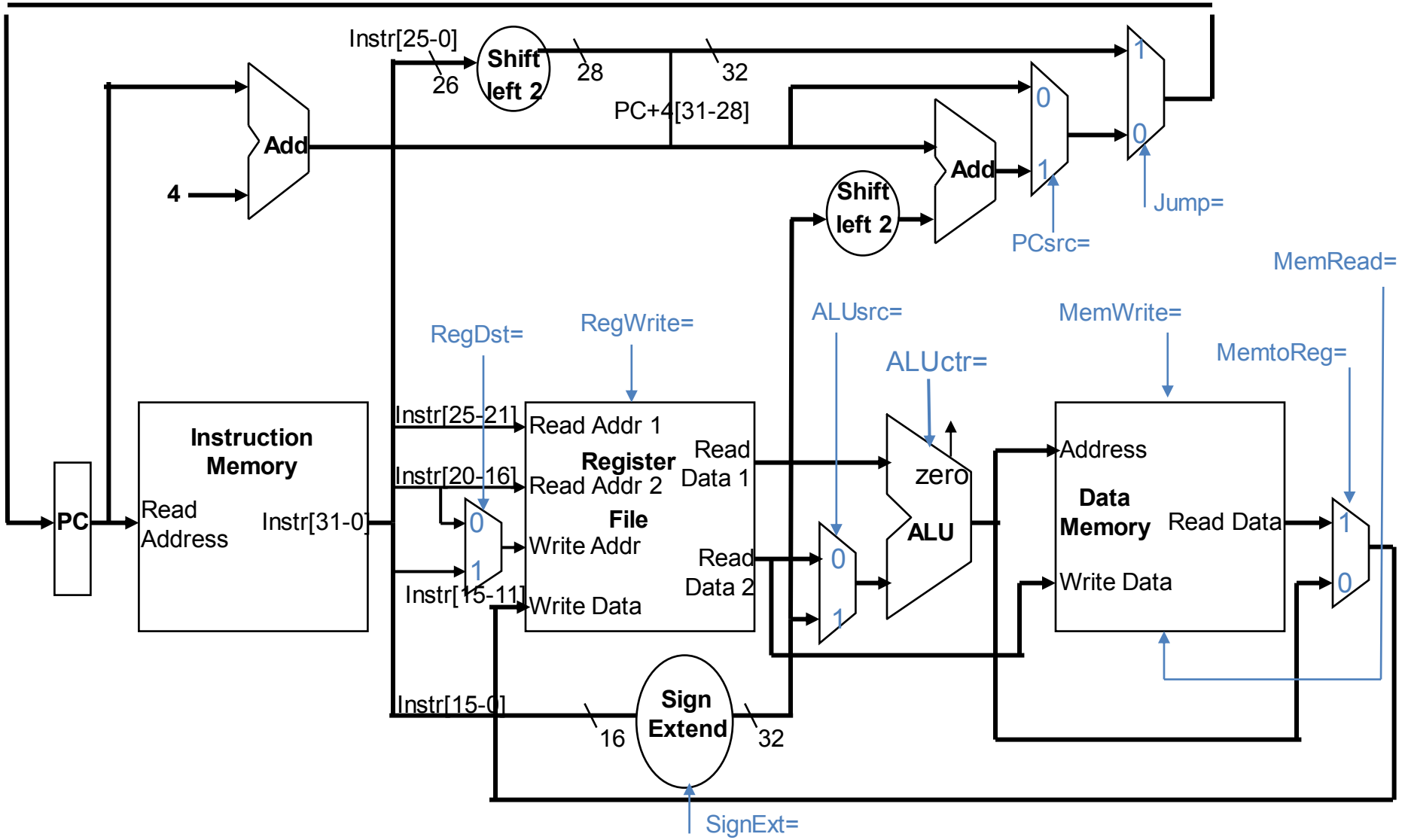
x: means we don't care



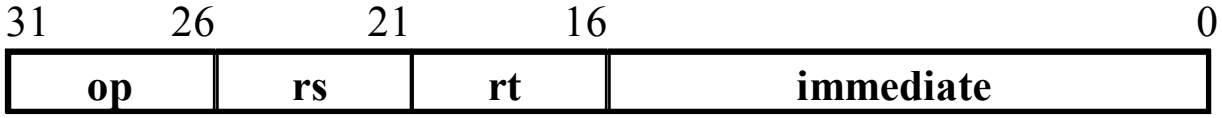
The Single Cycle Datapath during `sub`??



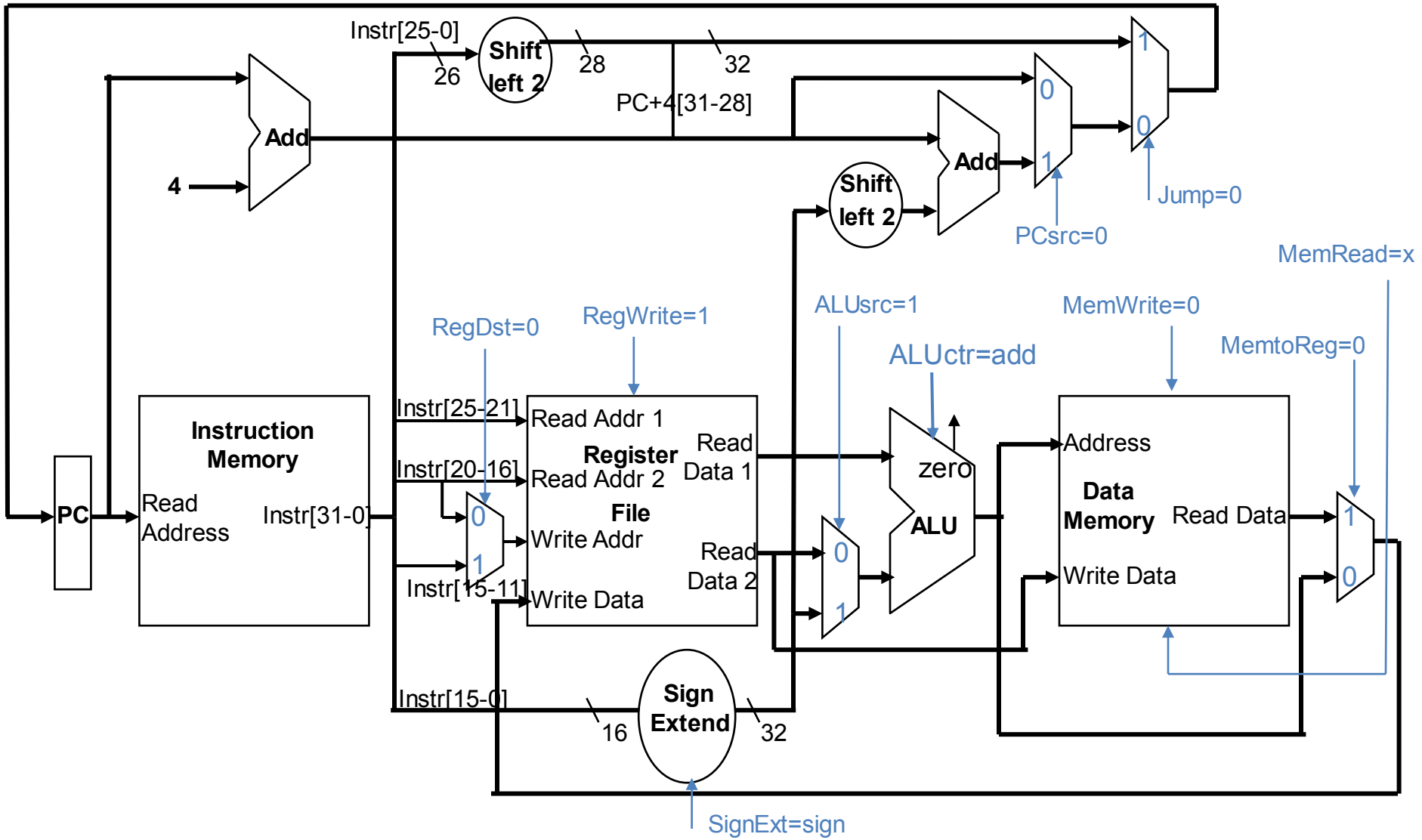
- R[rd] <- R[rs] - R[rt] (sub rd, rs, rt)



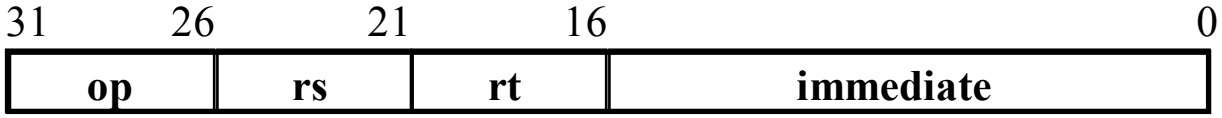
The Single Cycle Datapath during `addi`



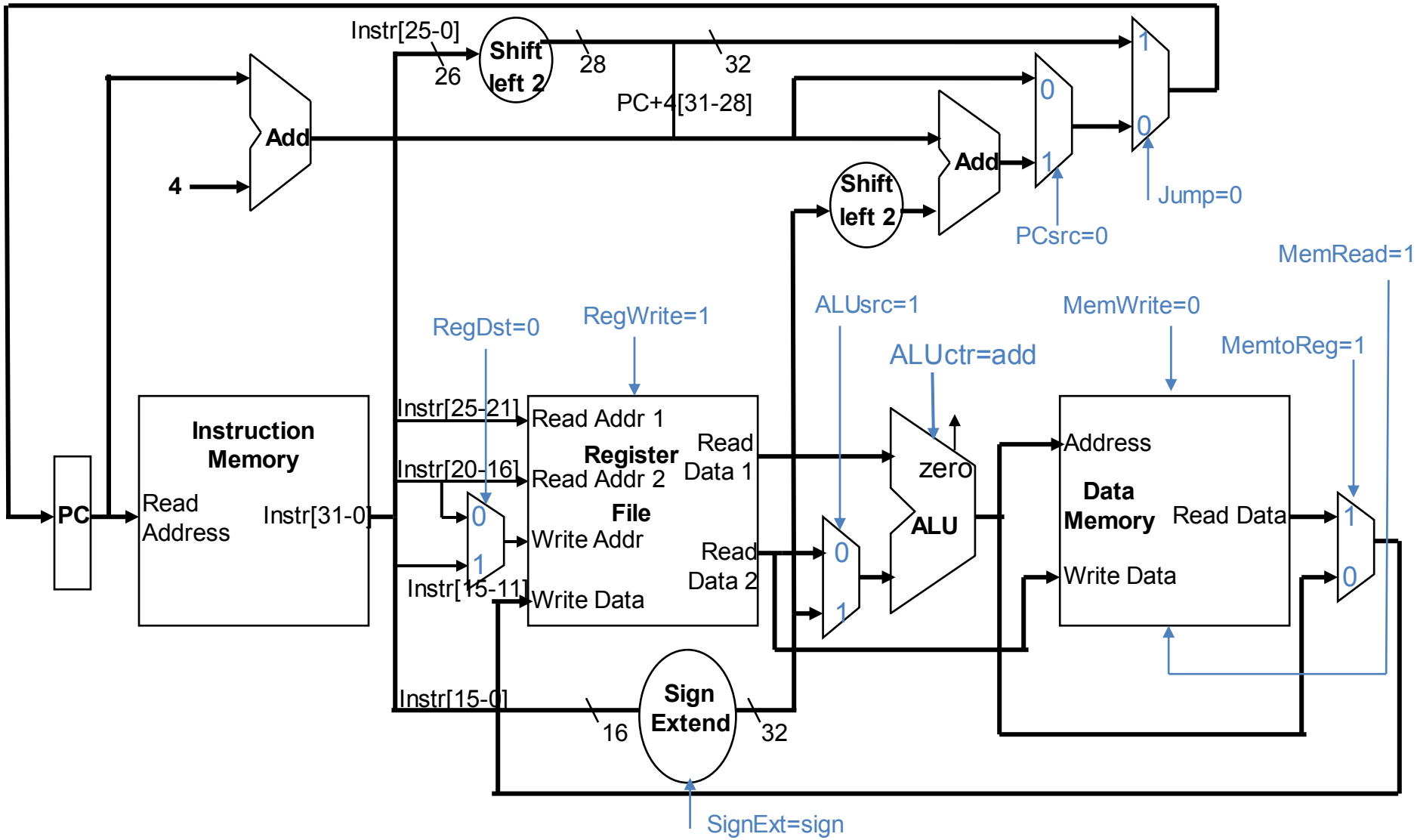
- R[rt] <- R[rs] or SignExt[imm16] (addi t0, t1, -10)



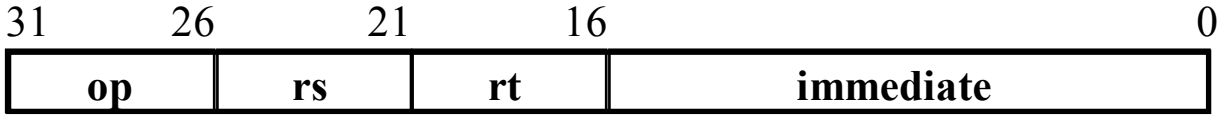
The Single Cycle Datapath during Load



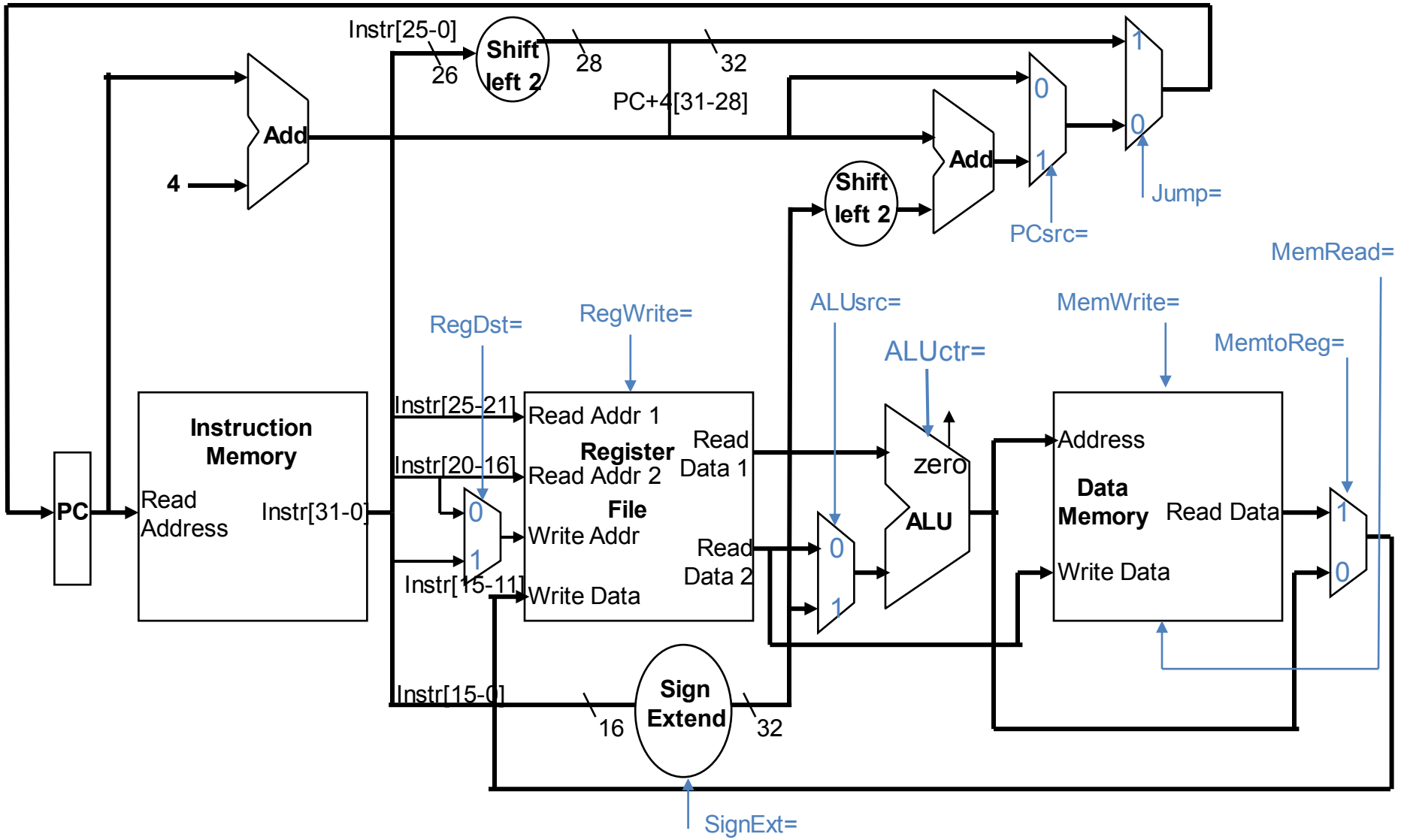
- R[rt] <- Data Memory {R[rs] + SignExt[imm16]} lw rt, imm16(rs)



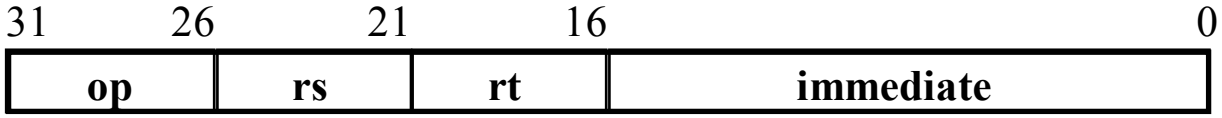
The Single Cycle Datapath during `Store`??



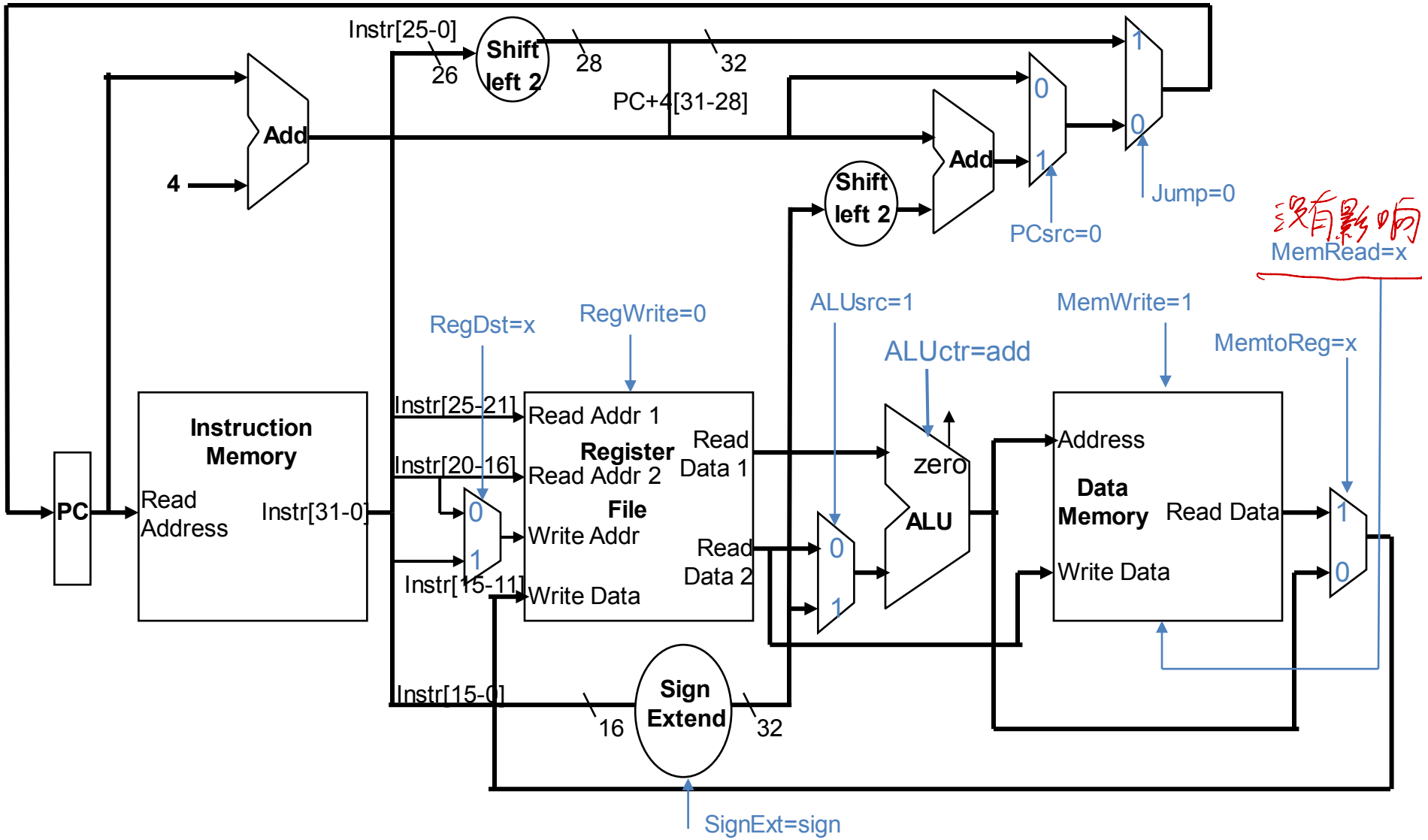
- Data Memory {R[rs] + SignExt[imm16]} <- R[rt] sw rt, imm16(rs)



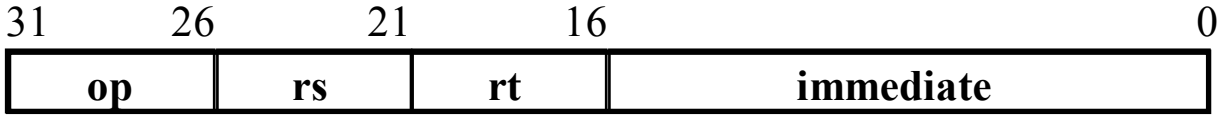
The Single Cycle Datapath during Store



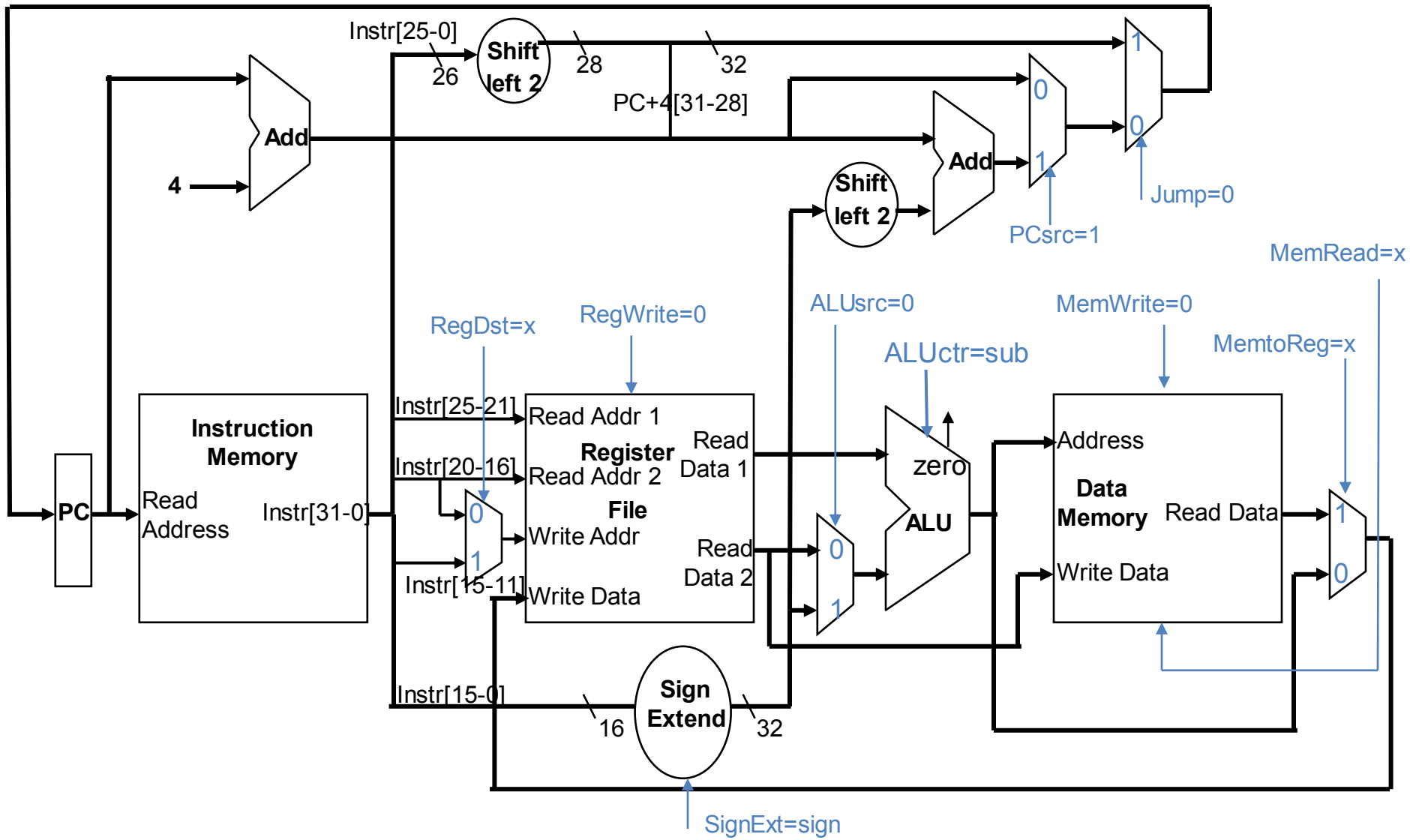
- Data Memory {R[rs] + SignExt[imm16]} <- R[rt] sw rt, imm16(rs)



The Single Cycle Datapath during Branch

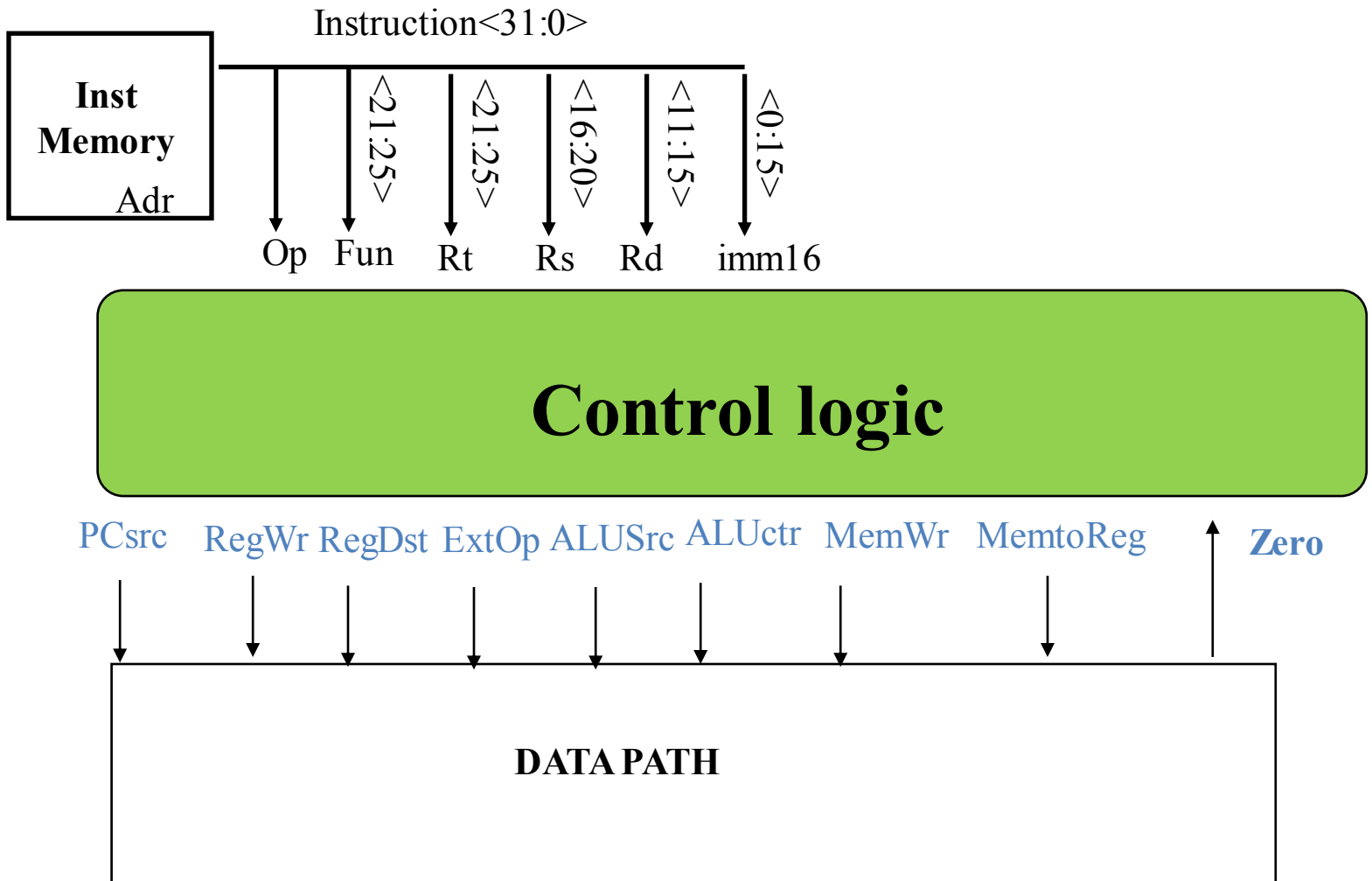


- if (R[rs] - R[rt] == 0) then zero = 1; else zero = 0 beq rs, rt, Label

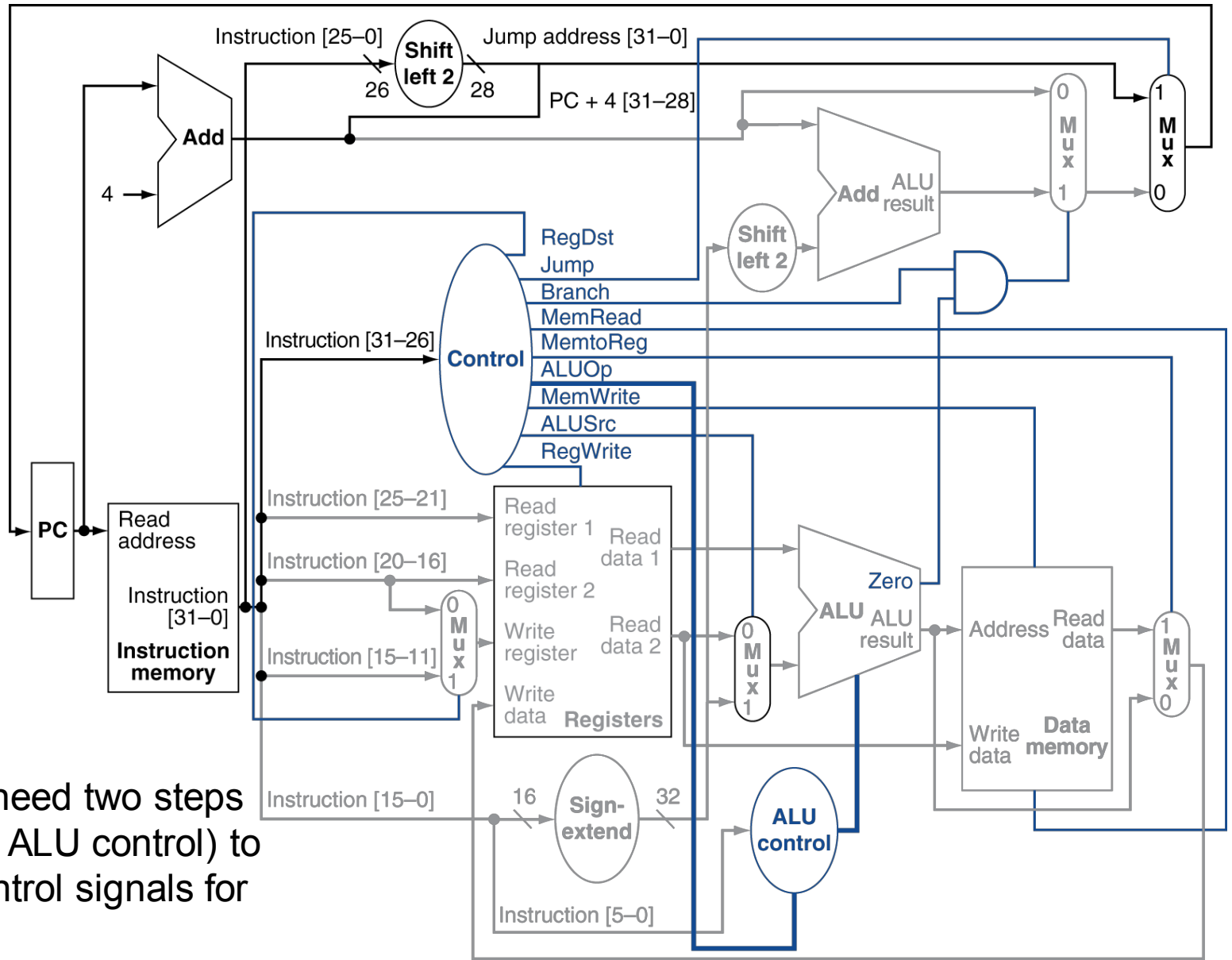


Step 5: Adding control logic

- Control logic is a component used to determine the value of control signals based on instruction type.
  - E.g. ALUctr, RegDst, ……



Datapath after adding control logic: Control, ALU control, and



Why do we need two steps (Control and ALU control) to generate control signals for the ALU?

ALU Control

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

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

ALU Control Truth Table

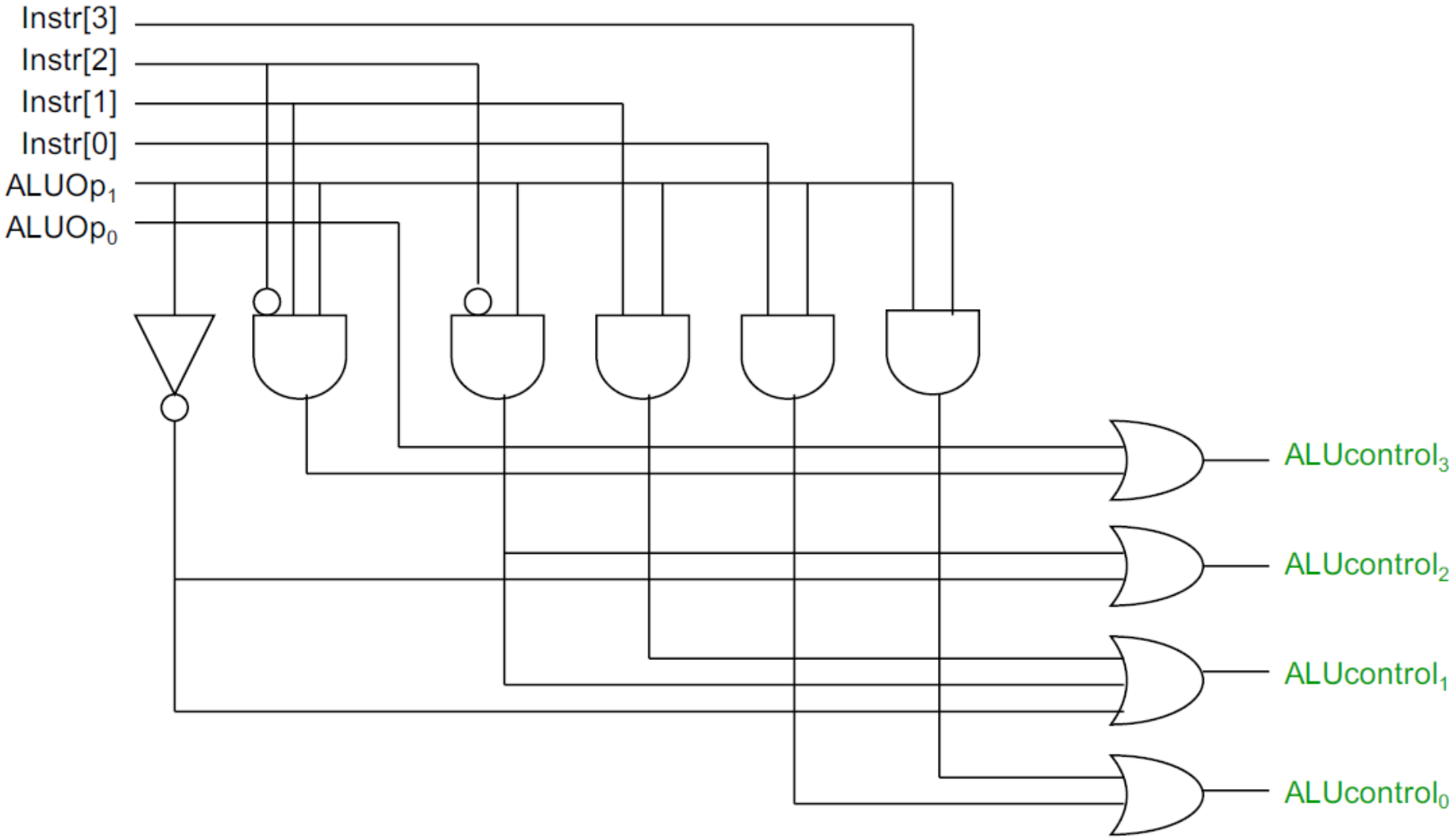
- ALU's operation based on instruction type and function code.

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

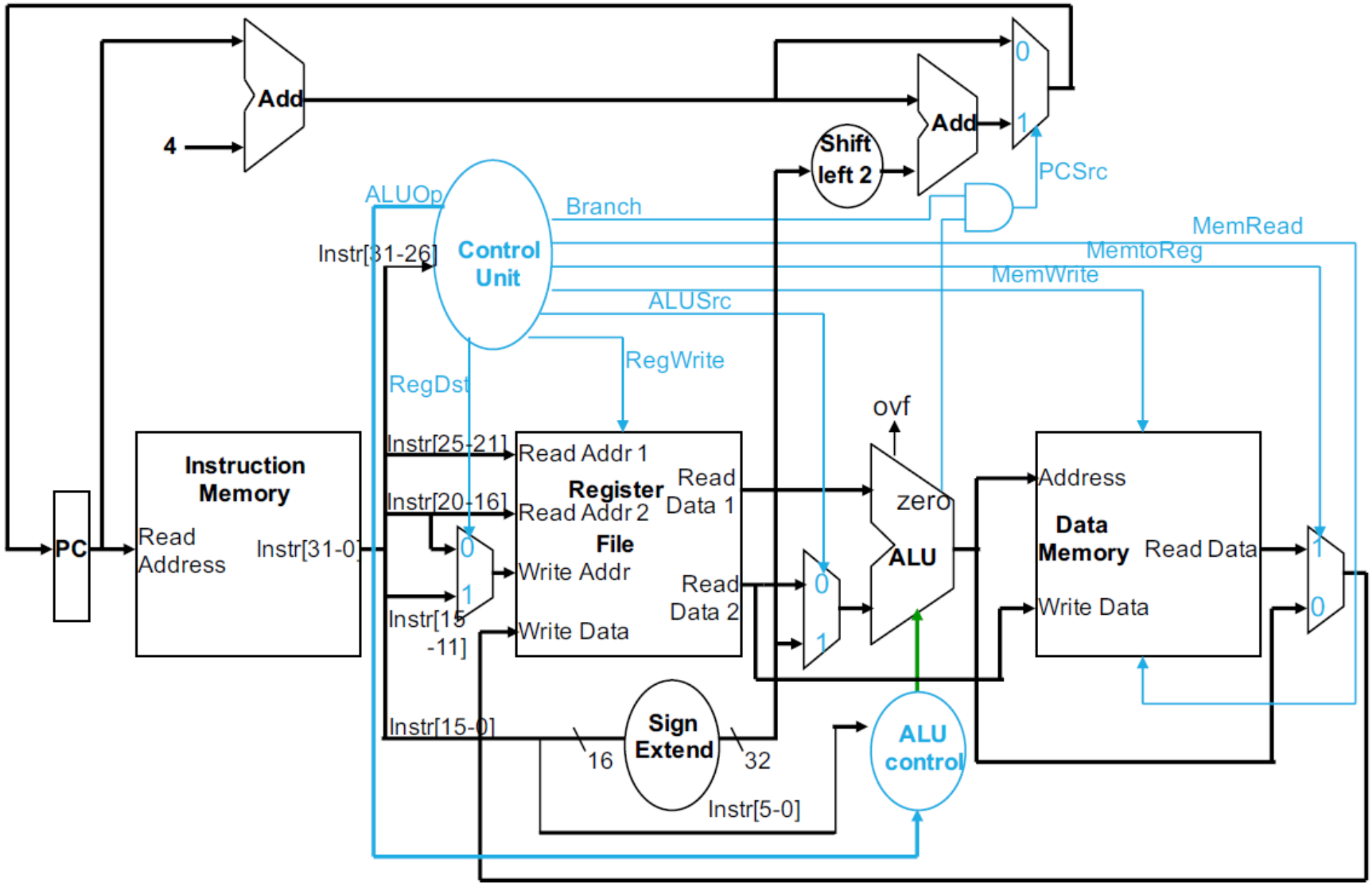
ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

ALU Control Logic

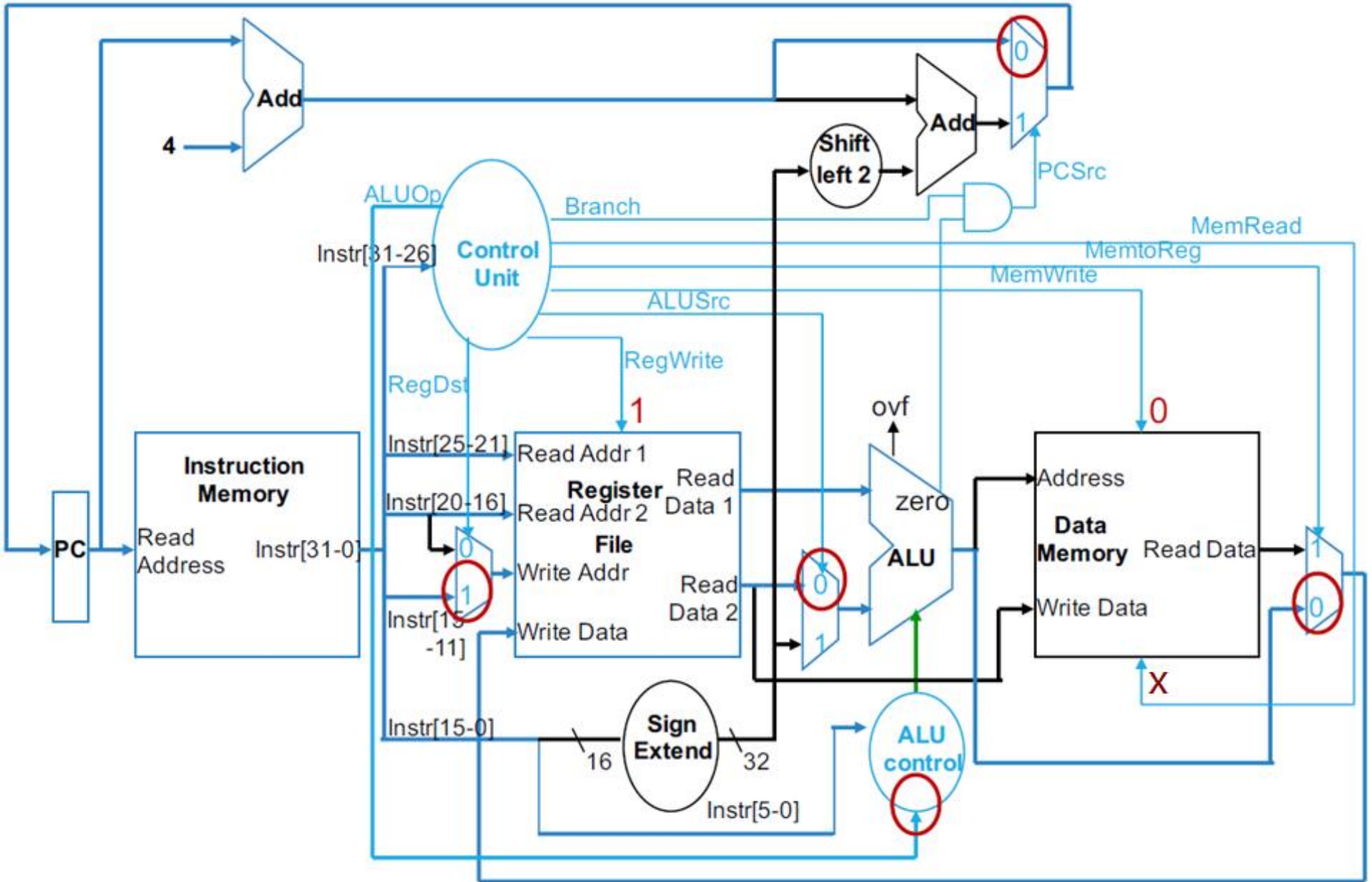
- From the truth table can design the ALU Control logic



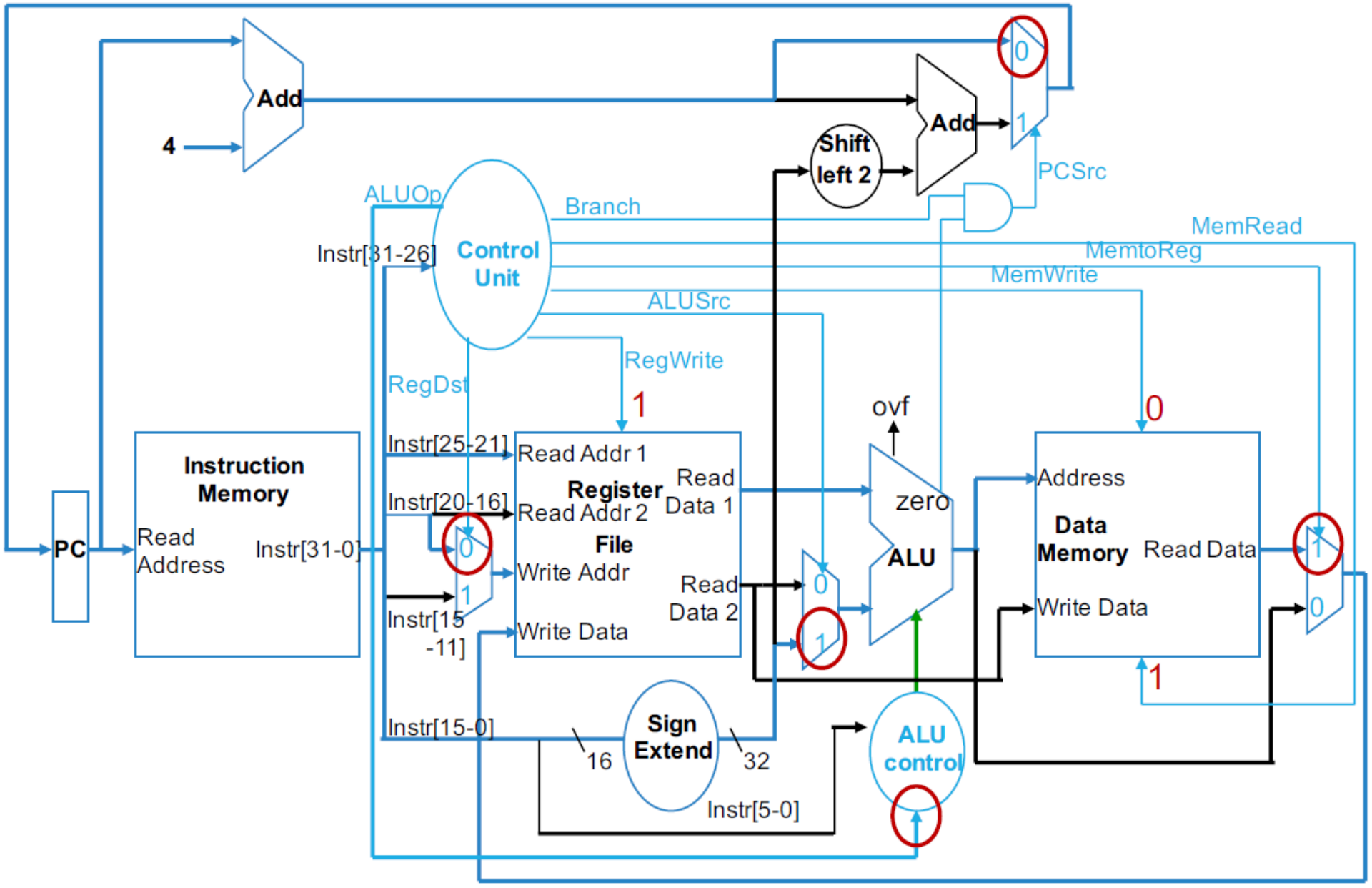
Complete Datapath with Control Unit



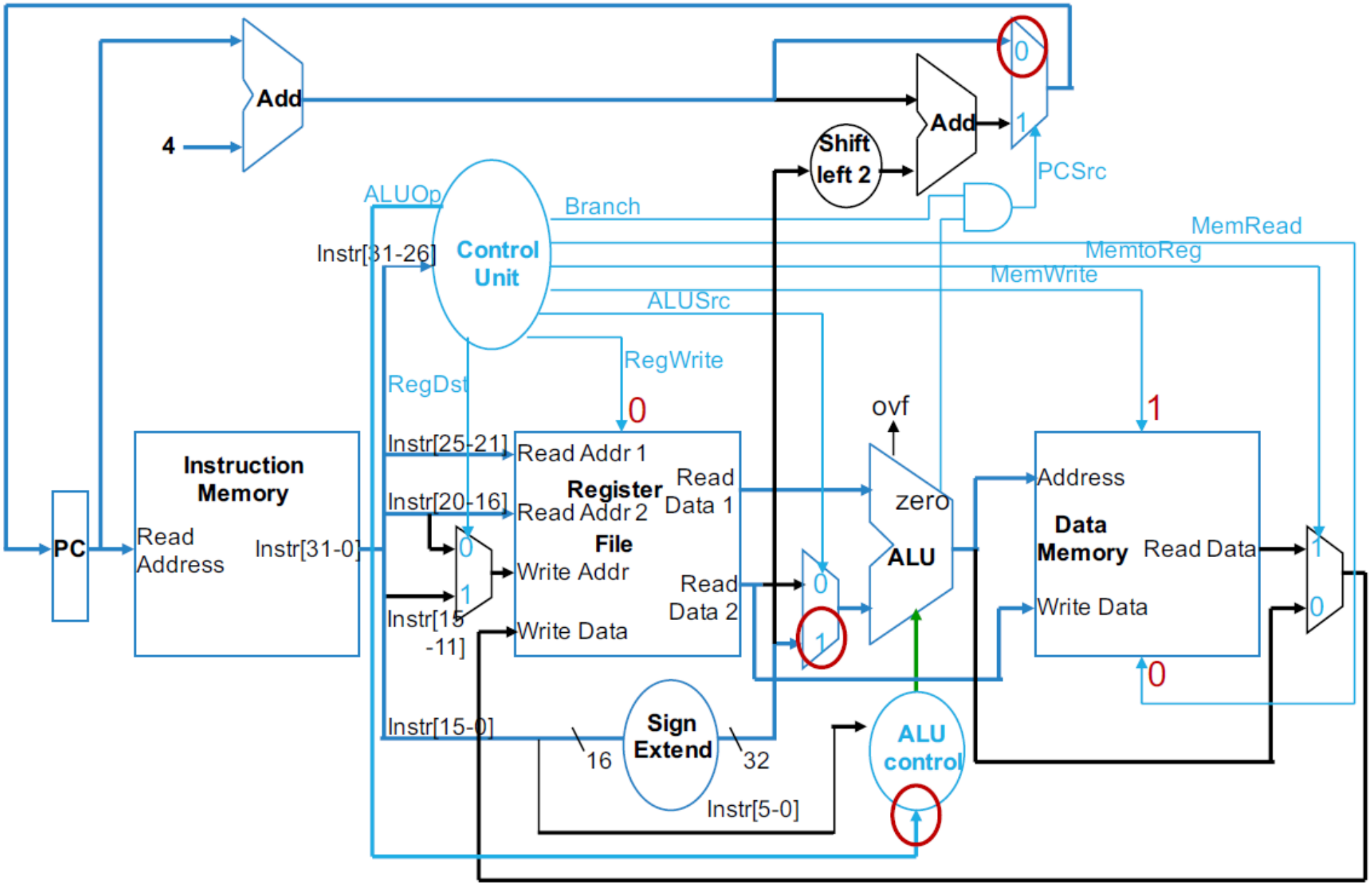
R-type Instruction Data/Control Flow



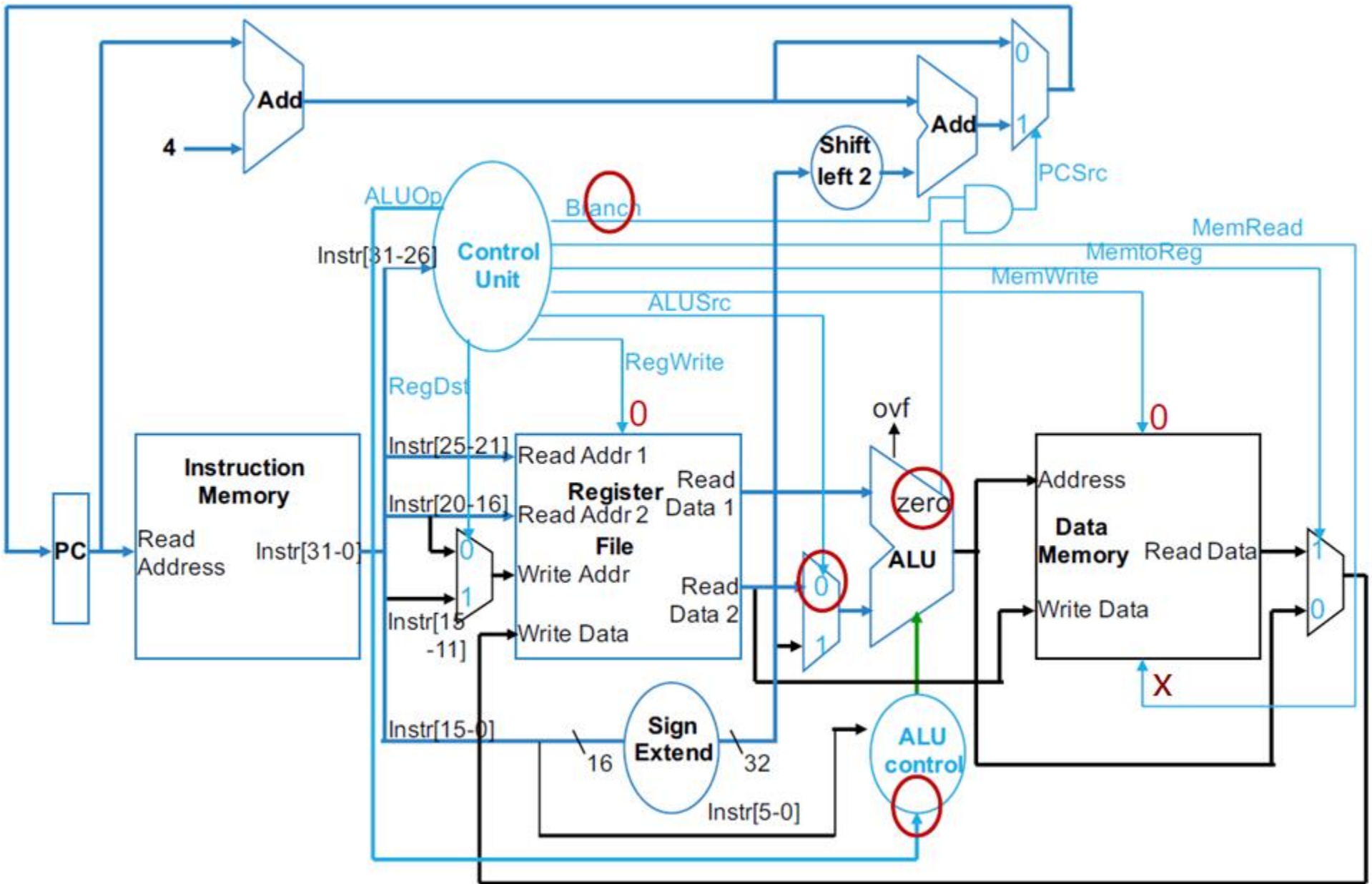
Load Word Instruction Data/Control Flow



Store Word Instruction Data/Control Flow



Branch Instruction Data/Control Flow

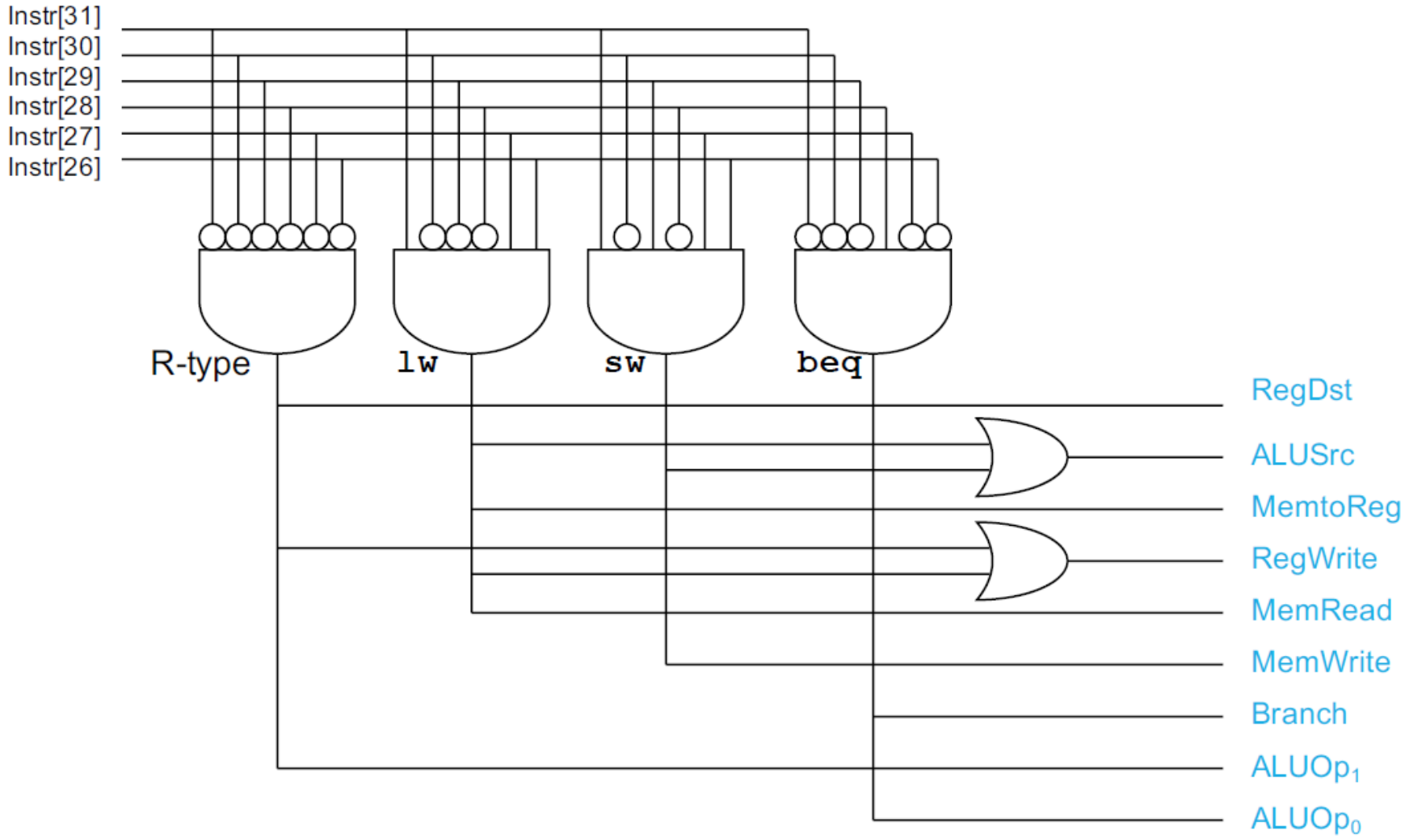


Main Control Unit

Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp
R-type 000000	1	0	0	1	X	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	X	0	1	01

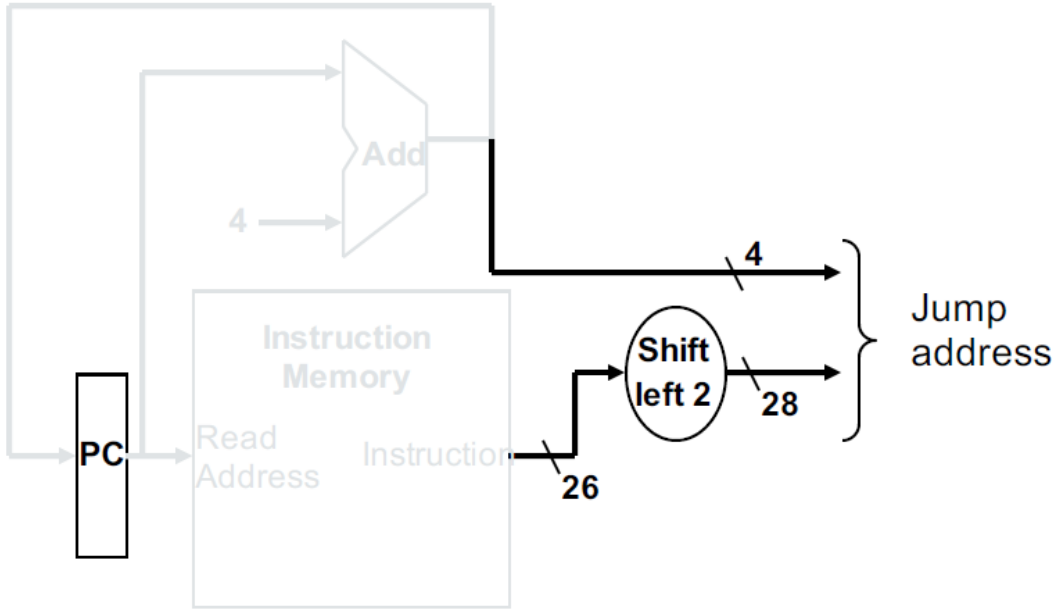
Control Unit Logic

- From the truth table can design the Main Control logic

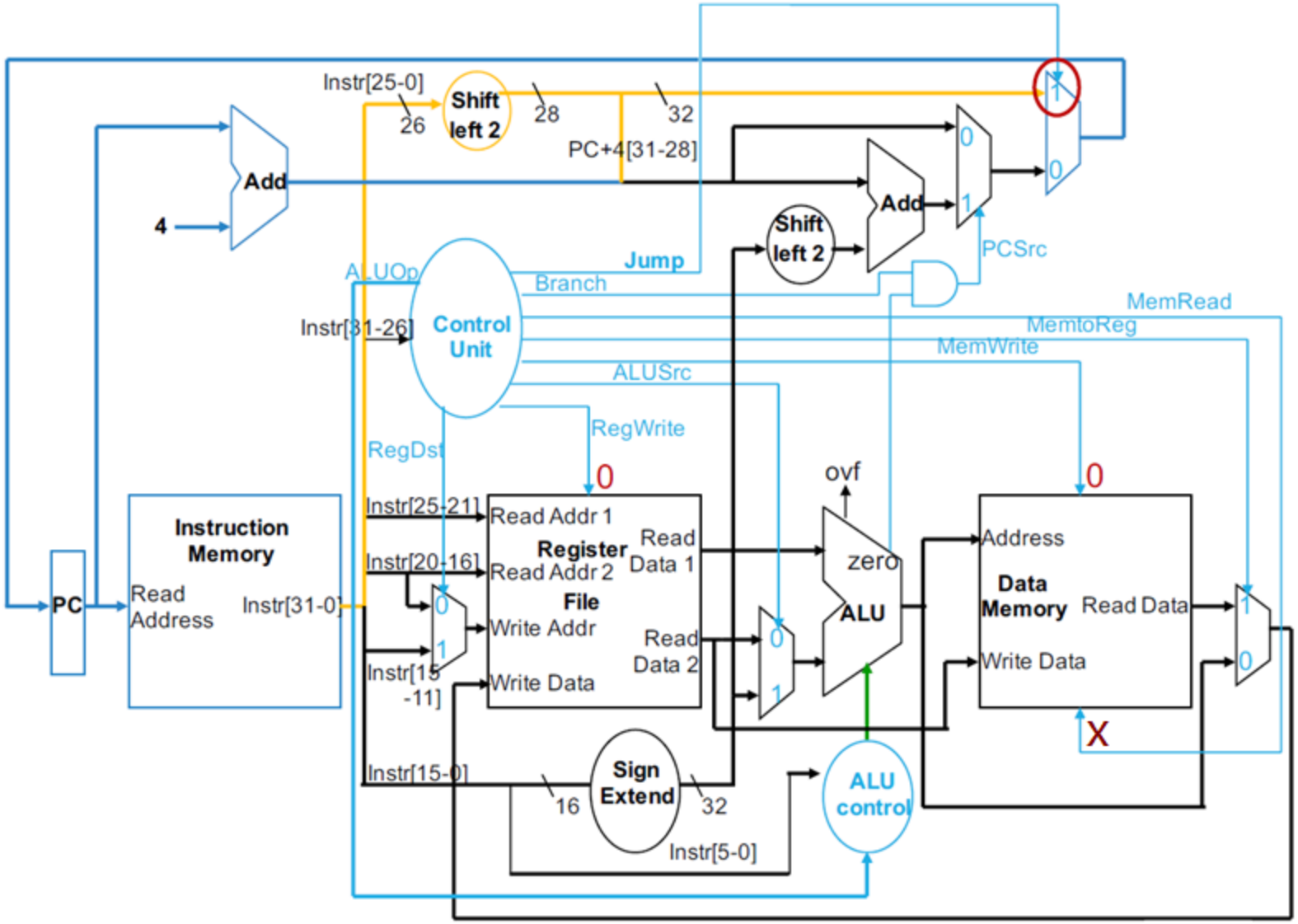


Review: Handling Jump Operations

- 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



Main Control Unit of j

Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp	Jump
R-type 000000	1	0	0	1	X	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	X	0	1	01	0
j 000010									1

Single Cycle Implementation Cycle Time

- Single Cycle Implementation Cycle Time approach is not used because it is very slow.
- Clock cycle must have the same length for every instruction.

- What is the longest path (slowest instruction)?

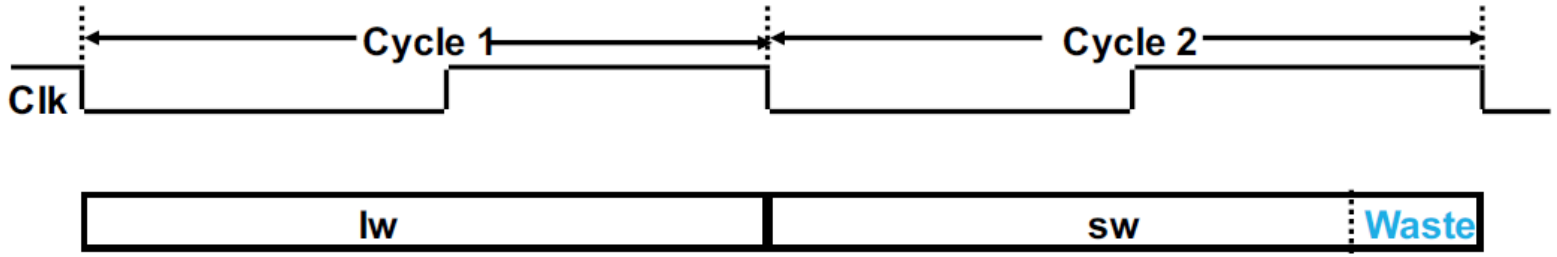
Instruction Critical Paths

- Calculate cycle time assuming negligible delays (for muxes, control unit, sign extend, PC access, shift left 2, wires) except:
  - Instruction and Data Memory (4 ns)
  - ALU and adders (2 ns)
  - Register File access (reads or writes) (1 ns)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type (4)	4 +1	1 +1	2 +1		1 +1	8
load (5)	4	1	2	4	1	12
store	4					
beq						
jump						

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 it is simple and easy to understand.

Summary

- Single cycle datapath => CPI=1
  - Long cycle time
  - Cycle time for load is much longer than needed for all other instructions
- 5 steps to design a processor
  - 1. Analyze instruction set => datapath requirements
  - 2. Select set of datapath components
  - 3. Assemble datapath meeting the requirements
  - 4. Analyze implementation of each instruction to determine setting of control signals
  - 5. Assemble the control logic
- Usually control is the hard part
  - Complex control => long cycle time