Computer Architecture (Spring 2020)

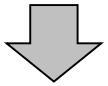
Instruction Set Principles

Dr. Duo Liu (刘铎)
Office: Main Building 0626
Email: liuduo@cqu.edu.cn

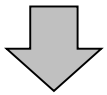
How Does a Machine Process Instructions?

- What does processing an instruction mean?
- Remember the von Neumann model

A = Architectural (programmer visible) state before an instruction is processed



Process instruction



A' = Architectural (programmer visible) state after an instruction is processed

- Processing an instruction: Transforming A to A' according to the ISA specification of the instruction

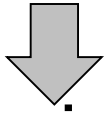
The "Process instruction" Step

- ISA specifies abstractly what A' should be, given an instruction and A
  - It defines an abstract finite state machine where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no "intermediate states" between A and A' during instruction execution
    - One state transition per instruction
- Microarchitecture implements how A is transformed to A'
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: multiple state transitions per instruction
    - Choice 1: A → A' (transform A to A' in a single clock cycle)
    - Choice 2: A → A+MS1 → A+MS2 → A+MS3 → A' (take multiple clock cycles to transform A to A')

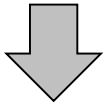
A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - *No intermediate, programmer-invisible state updates*

A = Architectural (programmer visible) state at the beginning of a clock cycle

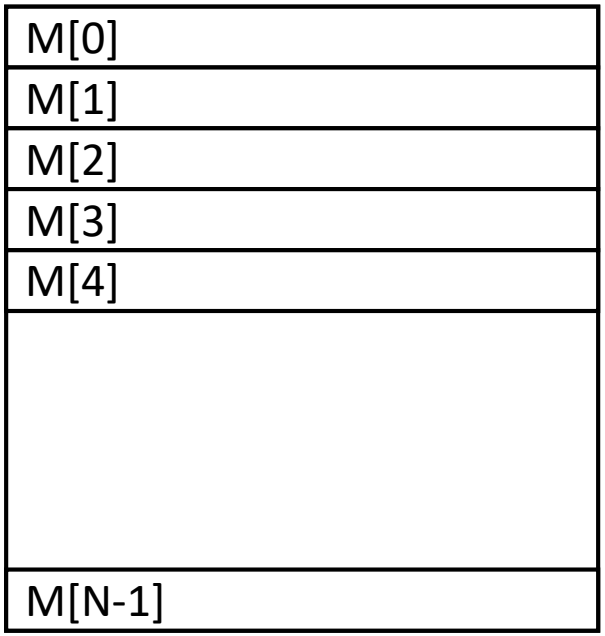


Process instruction in one clock cycle

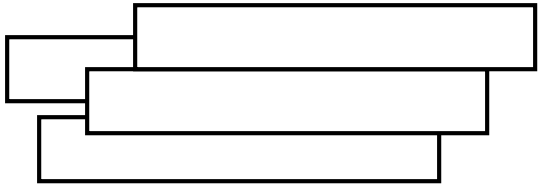


A' = Architectural (programmer visible) state at the end of a clock cycle

Remember: Programmer Visible (Architectural) State



Memory
array of storage locations
indexed by an address



Registers
- given special names in the ISA
  (as opposed to addresses)
- general vs. special purpose

Program Counter

memory address
of the current instruction

Instructions (and programs) specify how to transform the values of programmer visible state

Single-cycle vs. Multi-cycle Machines

- Single-cycle machines
  - Each instruction takes a single clock cycle
  - All state updates made at the end of an instruction's execution
  - Big disadvantage: The slowest instruction determines cycle time → long clock cycle time
- Multi-cycle machines
  - Instruction processing broken into multiple cycles/stages
  - State updates can be made during an instruction's execution
  - Architectural state updates made only at the end of an instruction's execution
  - Advantage over single-cycle: The slowest "stage" determines cycle time
  - Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level

Instruction Processing "Cycle"

- Instructions are processed under the direction of a "control unit" step by step.
- Instruction cycle: Sequence of steps to process an instruction
- Fundamentally, there are six phases:

- Fetch
- Decode
- Evaluate Address
- Fetch Operands
- Execute
- Store Result

- Not all instructions require all six stages

Instruction Processing "Cycle" vs. Machine Clock Cycle

- Single-cycle machine:
  - All six phases of the instruction processing cycle take a *single machine clock cycle* to complete
- Multi-cycle machine:
  - All six phases of the instruction processing cycle can take *multiple machine clock cycles* to complete
  - In fact, each phase can take multiple clock cycles to complete

Instruction Processing Viewed Another Way

- Instructions transform Data (A) to Data' (A')
- This transformation is done by functional units
  - Units that "operate" on data
- These units need to be told what to do to the data

- An instruction processing engine consists of two components
  - Datapath: Consists of hardware elements that deal with and transform data signals
    - functional units that operate on data
    - hardware structures (e.g. wires and muxes) that enable the flow of data into the functional units and registers
    - storage units that store data (e.g., registers)
  - Control logic: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data

Single-cycle vs. Multi-cycle: Control & Data

- Single-cycle machine:
  - Control signals are generated in the same clock cycle as data signals are operated on
  - Everything related to an instruction happens in one clock cycle
- Multi-cycle machine:
  - Control signals needed in the next cycle can be generated in the previous cycle
  - Latency of control processing can be overlapped with latency of datapath operation
- We will see the difference clearly in *microprogrammed multi-cycle microarchitecture*

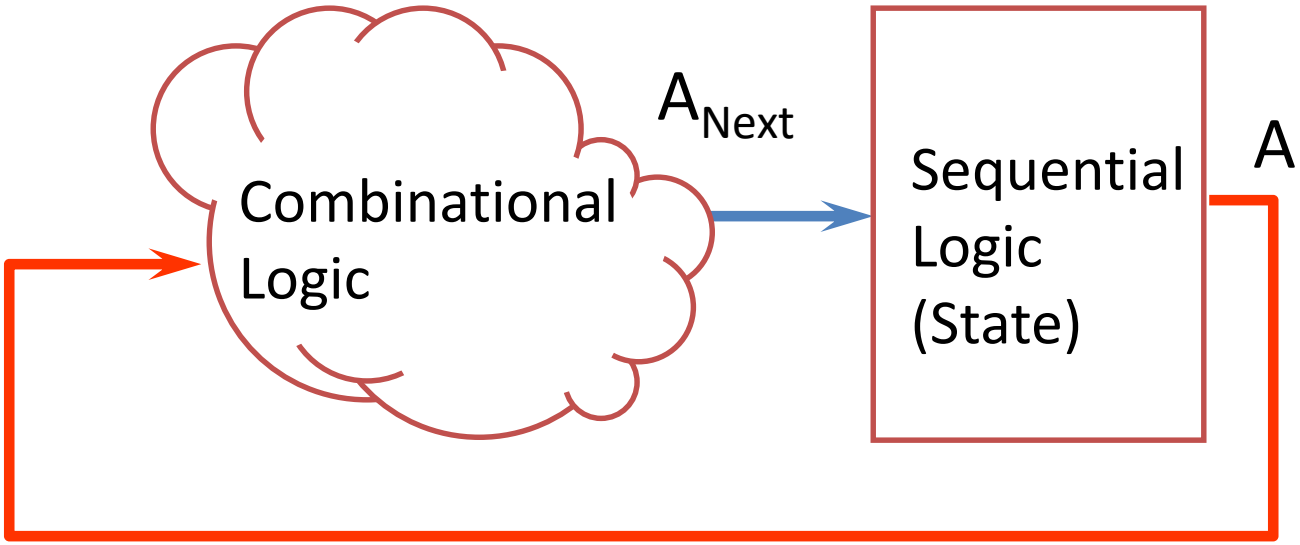
Performance Analysis

- Execution time of an instruction
  - {CPI} x {clock cycle time}
- Execution time of a program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - {# of instructions} x {Average CPI} x {clock cycle time}

- Single cycle microarchitecture performance
  - CPI = 1
  - Clock cycle time = long
- Multi-cycle microarchitecture performance
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

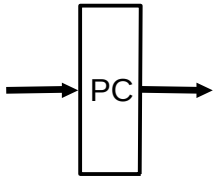
A Single Cycle Microarchitecture

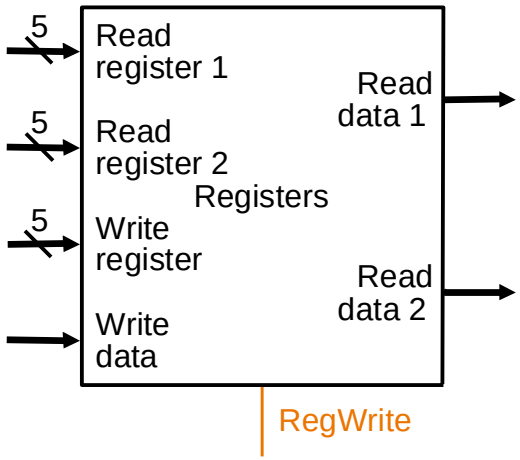
- Single-cycle machine

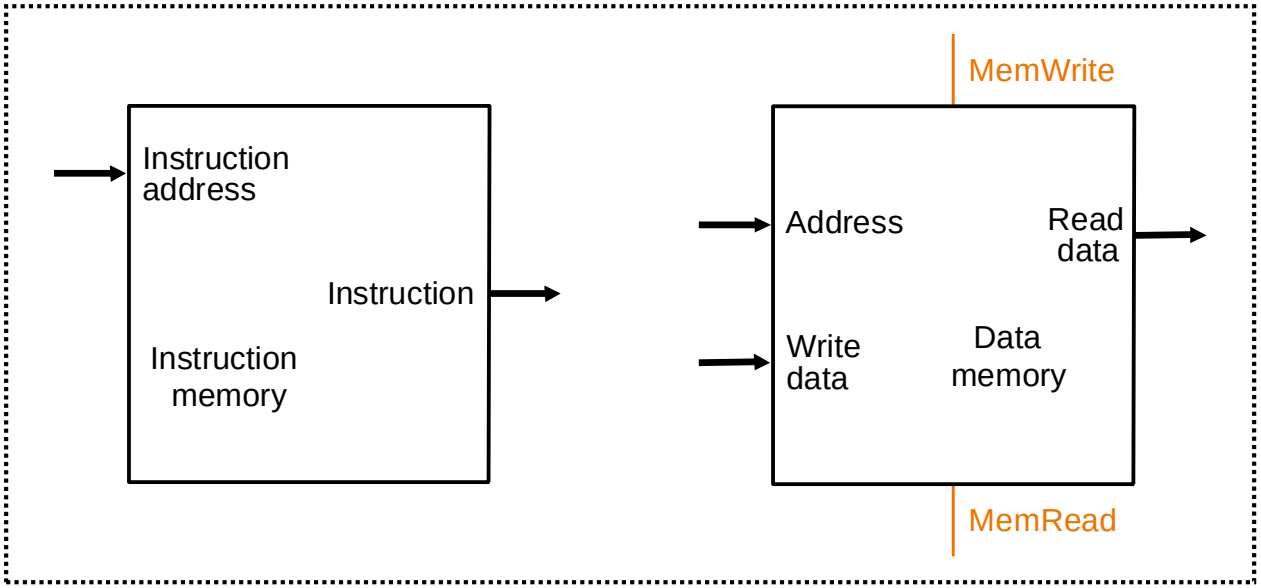


Let's Start with the State Elements

- Data and control inputs





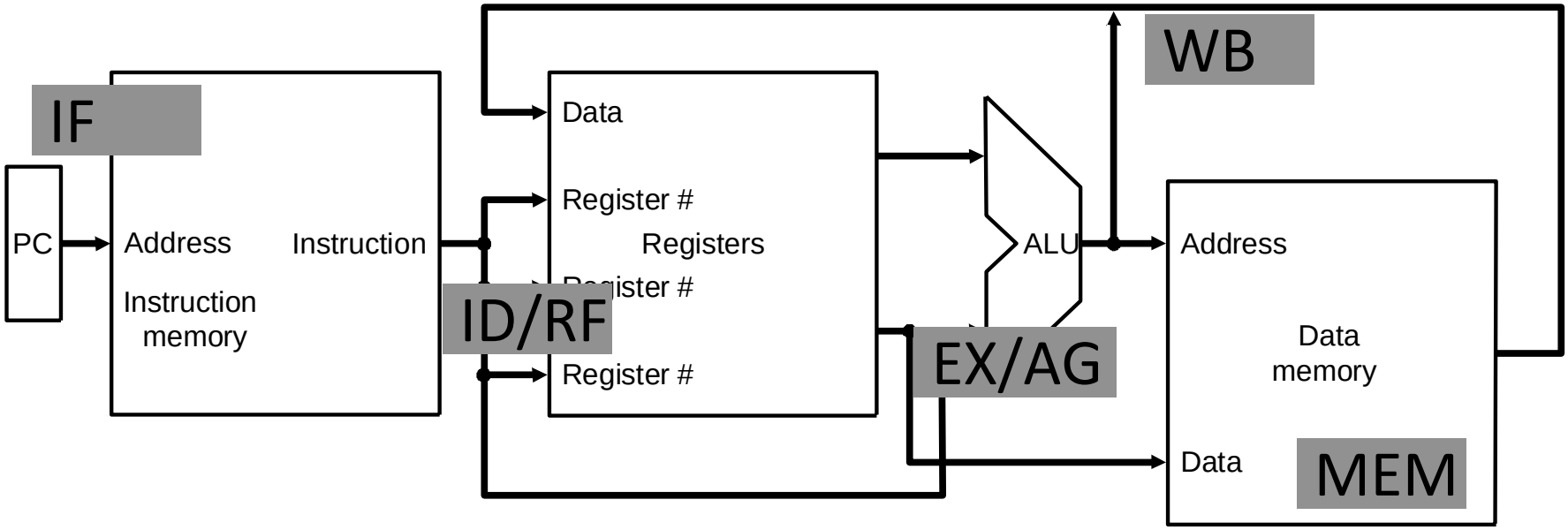


For Now, We Will Assume

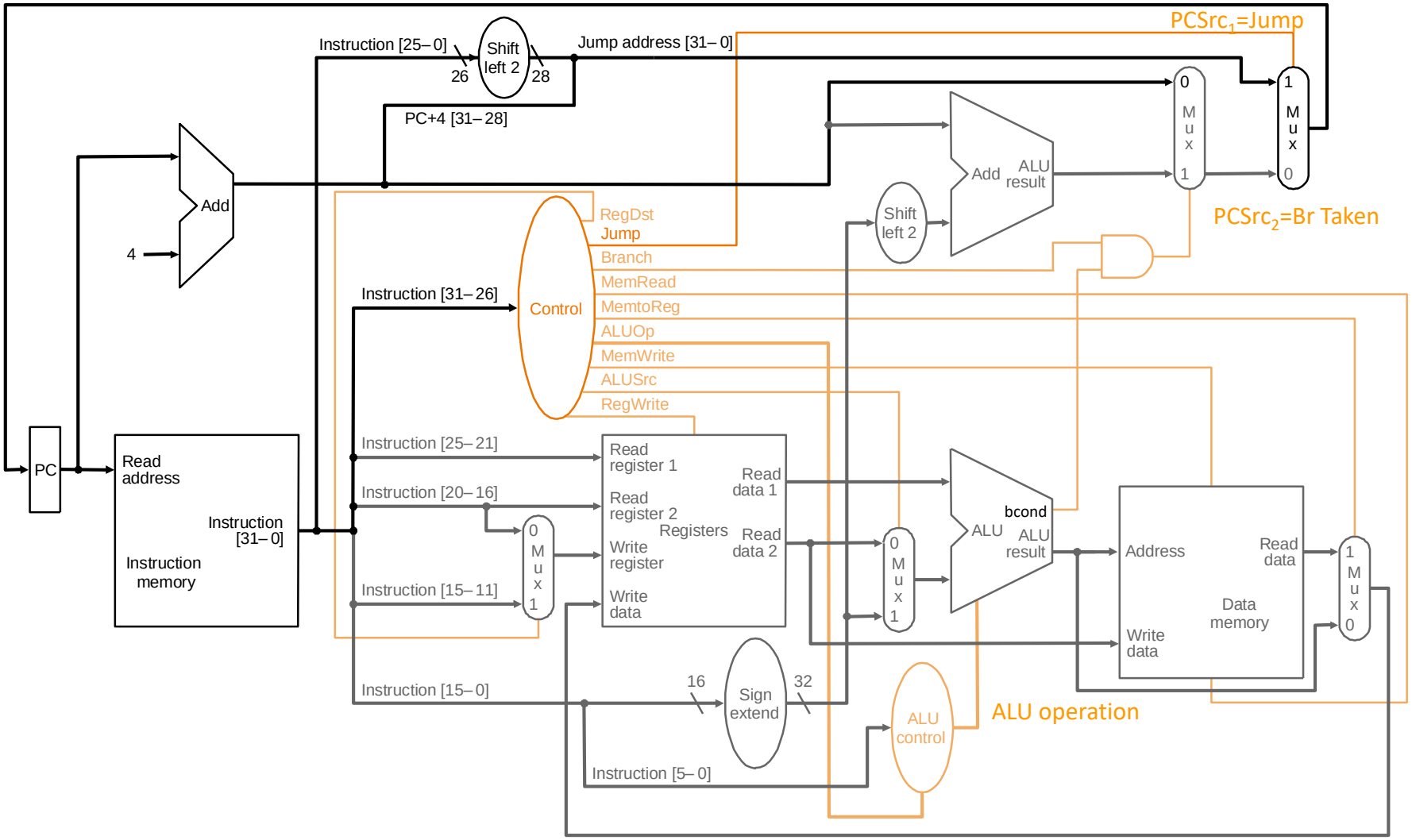
- "Magic" memory and register file
- Combinational read
  - output of the read data port is a combinational function of the register file contents and the corresponding read select port
- Synchronous write
  - the selected register is updated on the positive edge clock transition when write enable is asserted
    - Cannot affect read output in between clock edges
    - Can affect read output at clock edges (but who cares?)
- Single-cycle, synchronous memory
  - Contrast this with memory that tells when the data is ready
  - i.e., Ready bit: indicating the read or write is done

Instruction Processing

- 5 generic steps
  - Instruction fetch (IF)
  - Instruction decode and register operand fetch (ID/RF)
  - Execute/Evaluate memory address (EX/AG)
  - Memory operand fetch (MEM)
  - Store/writeback result (WB)



What Is To Come: The Full Datapath



JAL, JR, JALR omitted

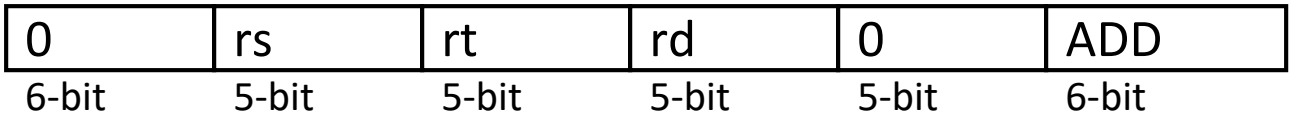
Single-Cycle Datapath for *Arithmetic and Logical Instructions*

R-Type ALU Instructions

- Assembly (e.g., register-register signed addition)

  ADD rd_{reg} rs_{reg} rt_{reg}

- Machine encoding



R-type

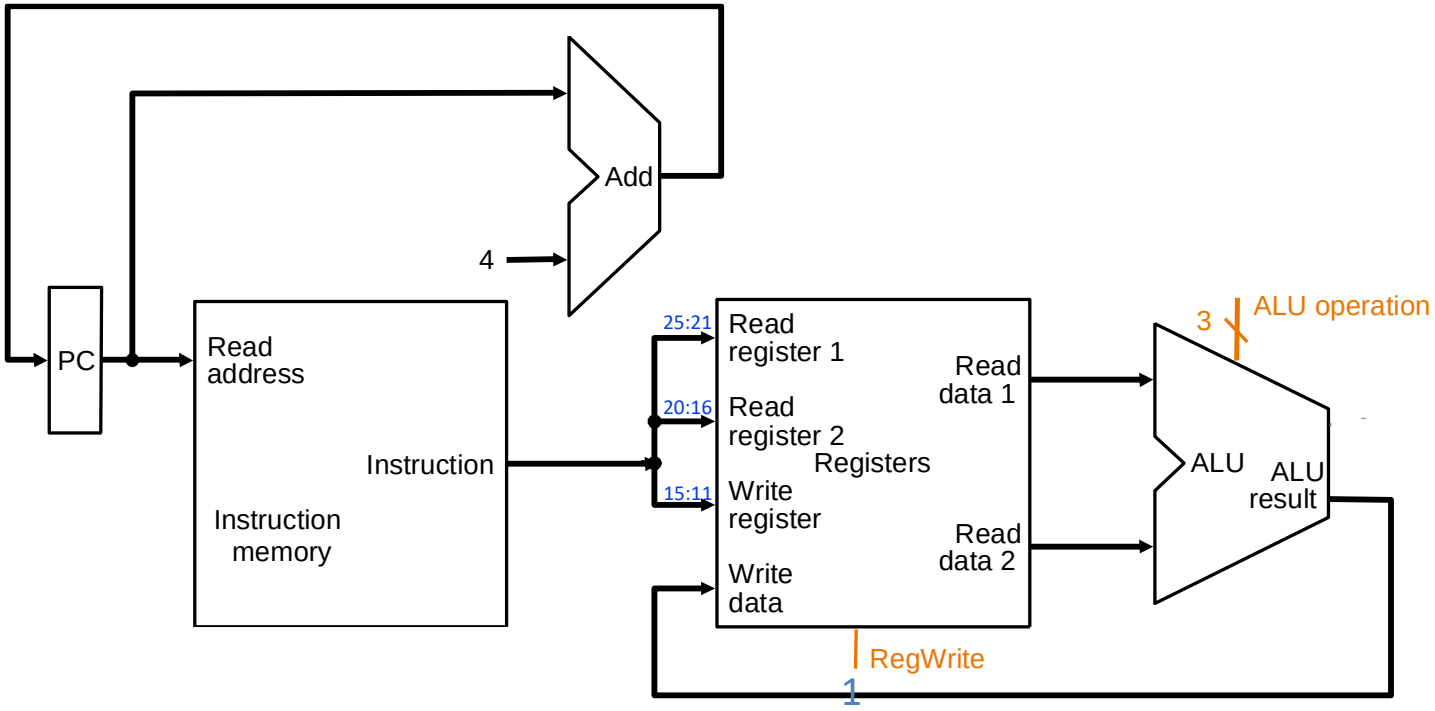
- Semantics

if MEM[PC] == ADD rd rs rt

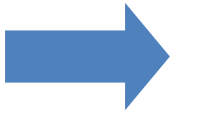
GPR[rd] ← GPR[rs] + GPR[rt]

PC ← PC + 4

ALU Datapath




```
if MEM[PC] == ADD rd rs rt
        GPR[rd] ← GPR[rs] + GPR[rt]
        PC ← PC + 4
```



IF	ID	EX	MEM	WB

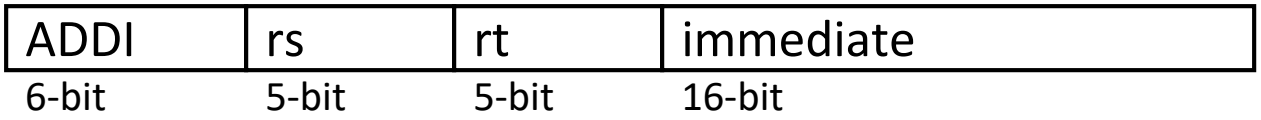
Combinational state update logic

I-Type ALU Instructions

- Assembly (e.g., register-immediate signed additions)

  ADDI rt_{reg} rs_{reg} $immediate_{16}$

- Machine encoding



I-type

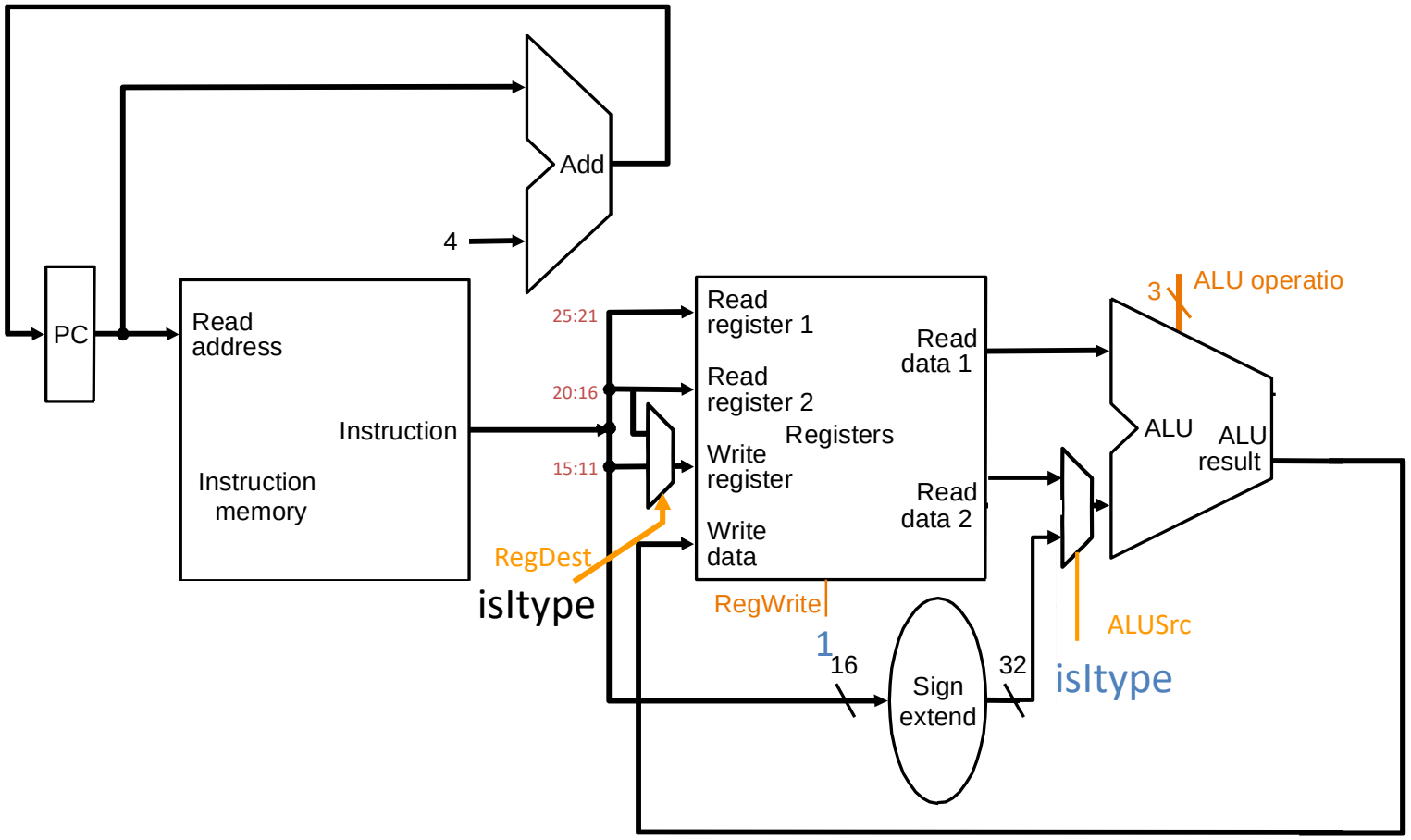
- Semantics

  if MEM[PC] == ADDI rt rs immediate

  GPR[rt] ← GPR[rs] + sign-extend (immediate)

  PC ← PC + 4

Datapath for R and I-Type ALU Insts.



IF	ID	EX	MEM	WB

if MEM[PC] == ADDI rt rs immediate

GPR[rt] ← GPR[rs] + sign-extend (immediate)

PC ← PC + 4

Combinational state update logic

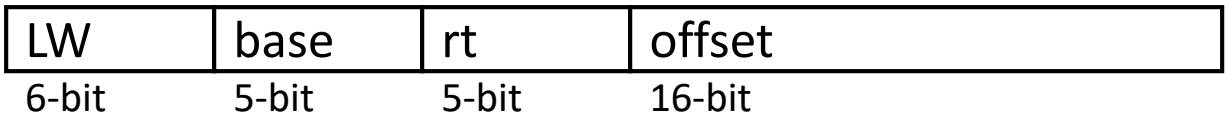
Single-Cycle Datapath for *Data Movement Instructions*

Load Instructions

- Assembly (e.g., load 4-byte word)

  LW rt_{reg} $offset_{16}$ ($base_{reg}$)

- Machine encoding



I-type

- Semantics

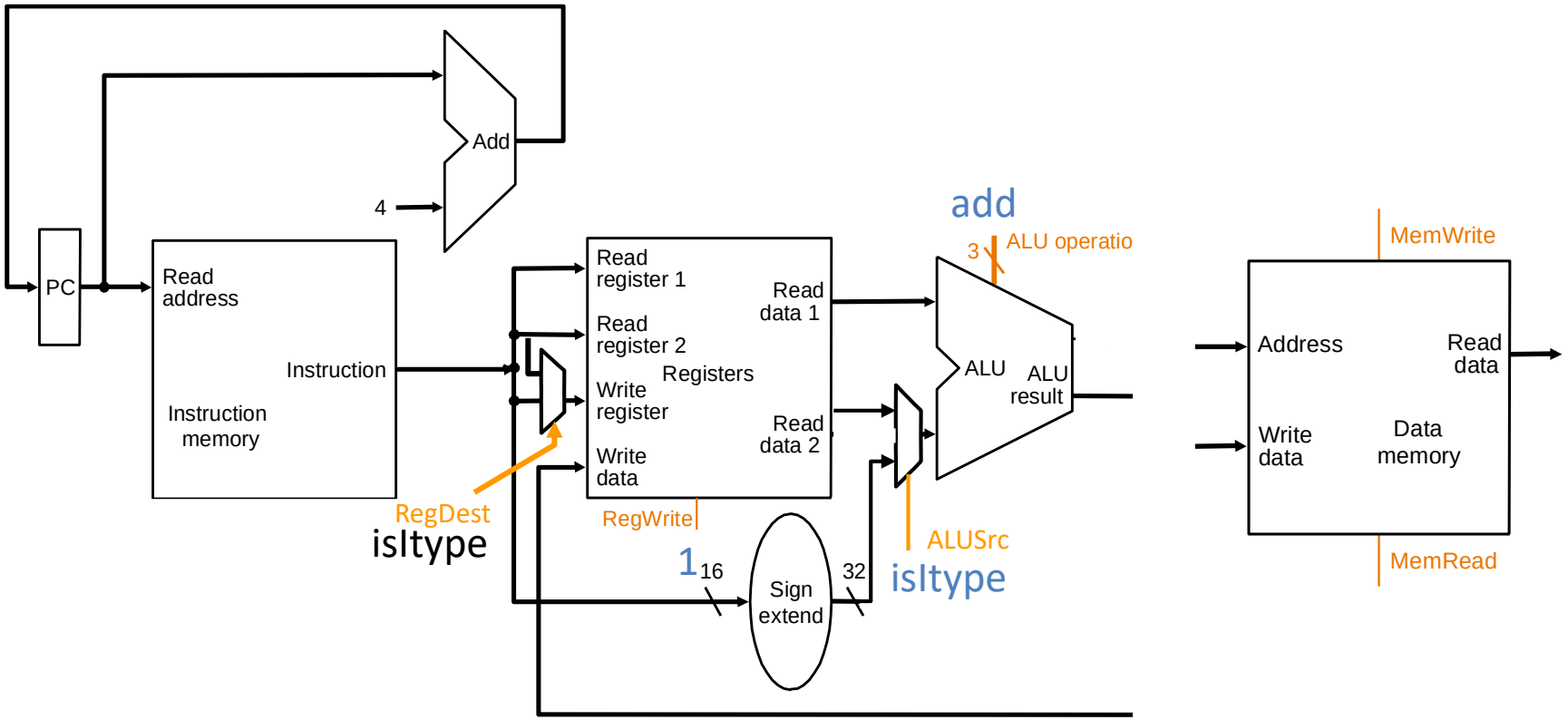
  if MEM[PC]==LW rt $offset_{16}$ (base)

  EA = sign-extend(offset) + GPR[base]

  GPR[rt] ← MEM[translate(EA)]

  PC ← PC + 4

LW Datapath



if MEM[PC]==LW rt offset$_{16}$ (base)

EA = sign-extend(offset) + GPR[base]

GPR[rt] ← MEM[translate(EA)]

PC ← PC + 4

IF	ID	EX	MEM	WB

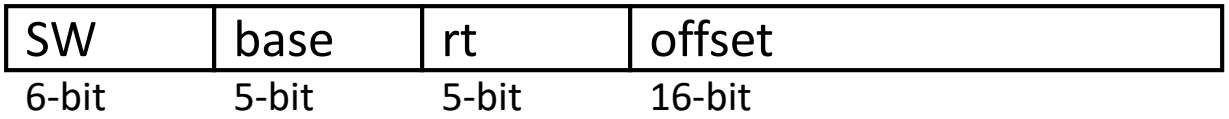
Combinational state update logic

Store Instructions

- Assembly (e.g., store 4-byte word)

  SW rt_{reg} $offset_{16}$ ($base_{reg}$)

- Machine encoding



I-type

- Semantics

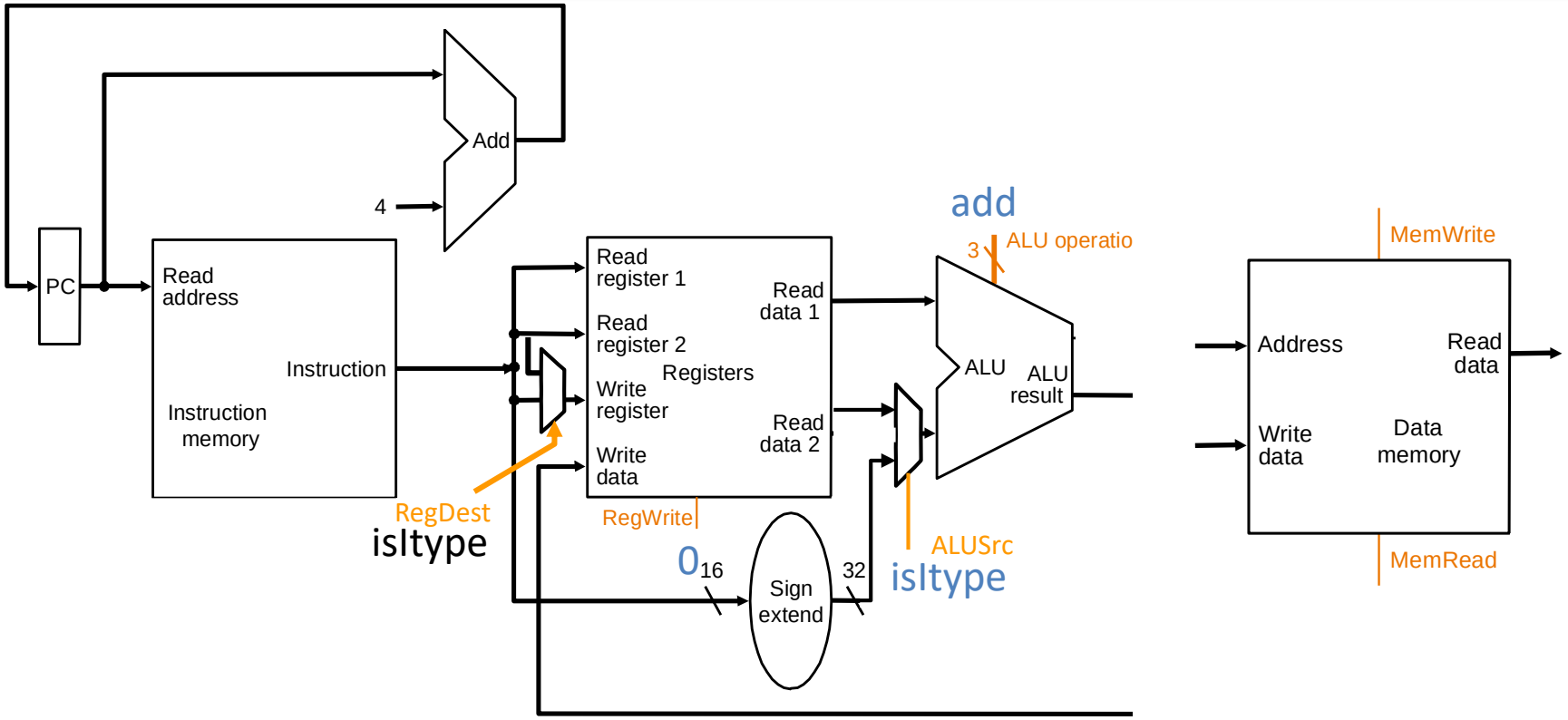
  if MEM[PC]==SW rt $offset_{16}$ (base)

  EA = sign-extend(offset) + GPR[base]

  MEM[translate(EA)] ← GPR[rt]

  PC ← PC + 4

SW Datapath



if MEM[PC]==SW rt offset$_{16}$ (base)

EA = sign-extend(offset) + GPR[base]

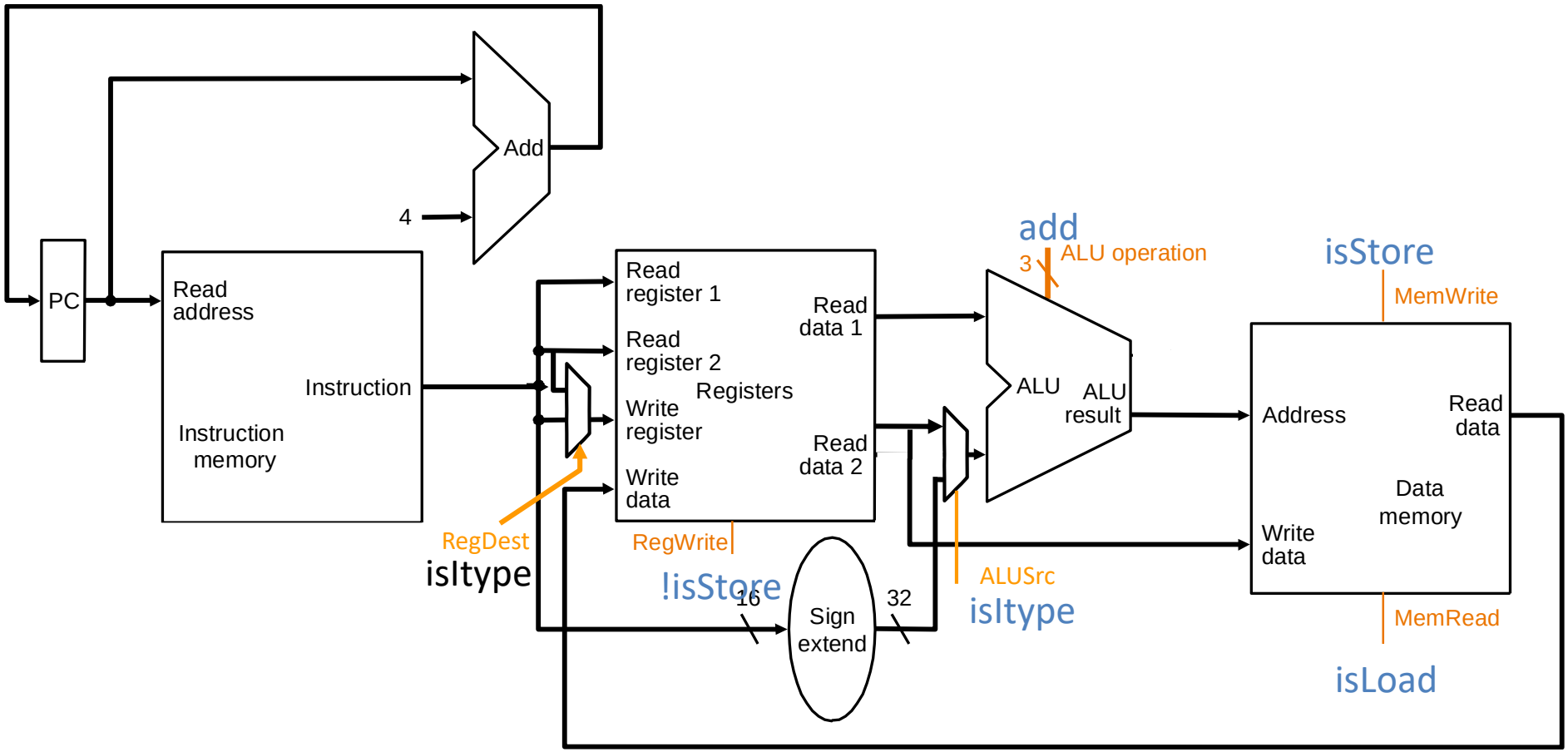
MEM[translate(EA)] ← GPR[rt]

PC ← PC + 4

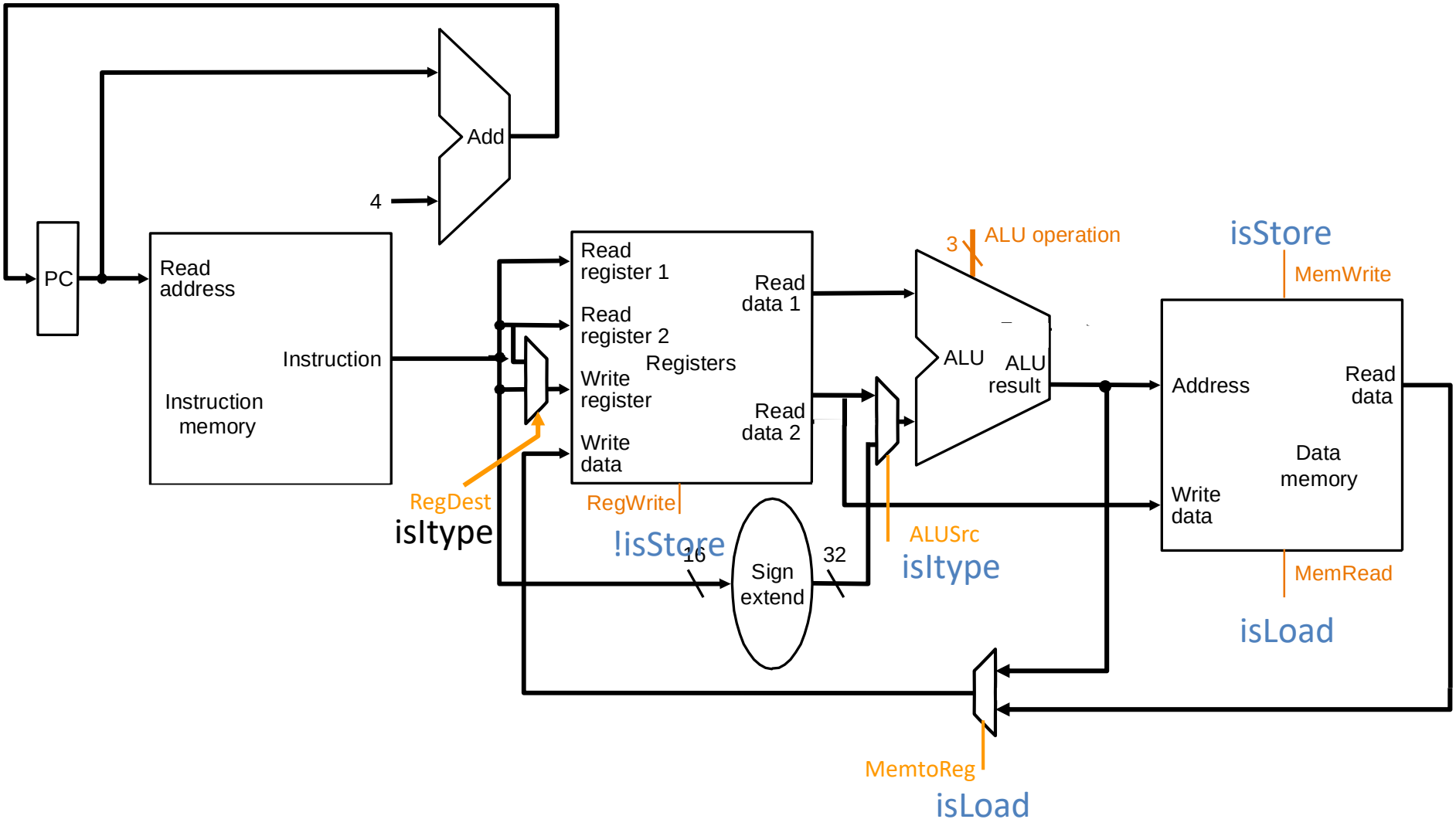
IF	ID	EX	MEM	WB

Combinational state update logic

Load-Store Datapath



Datapath for Non-Control-Flow Insts.



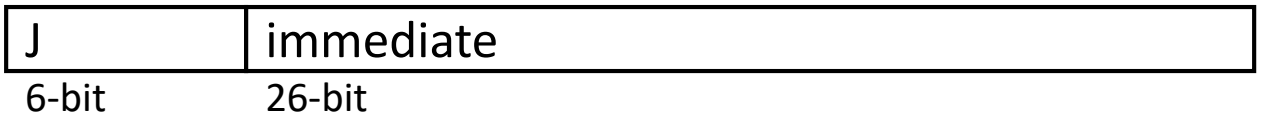
Single-Cycle Datapath for *Control Flow Instructions*

Unconditional Jump Instructions

- Assembly

  J $immediate_{26}$

- Machine encoding



J-type

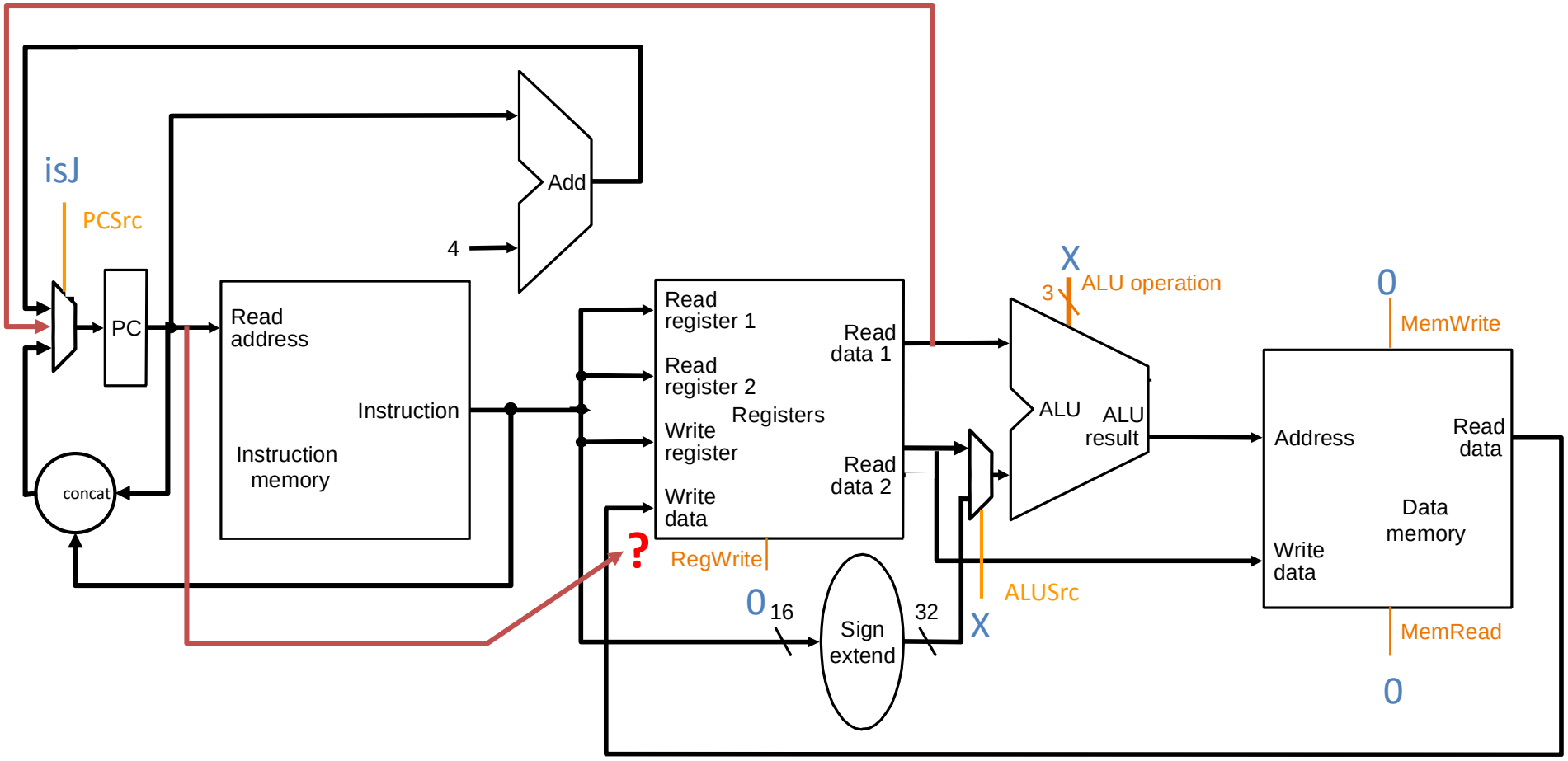
- Semantics

  if MEM[PC]==J $immediate_{26}$

  target = { PC[31:28], $immediate_{26}$, 2'b00 }

  PC ← target

Unconditional Jump Datapath



if MEM[PC]==J immediate26
PC = { PC[31:28], immediate26, 2'b00 }

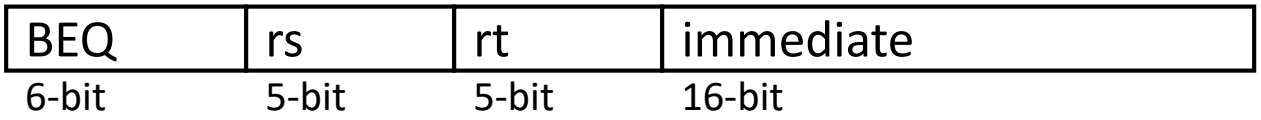
What about JR, JAL, JALR?

Conditional Branch Instructions

- Assembly (e.g., branch if equal)

  BEQ rs_{reg} rt_{reg} $immediate_{16}$

- Machine encoding



I-type

- Semantics (assuming no branch delay slot)

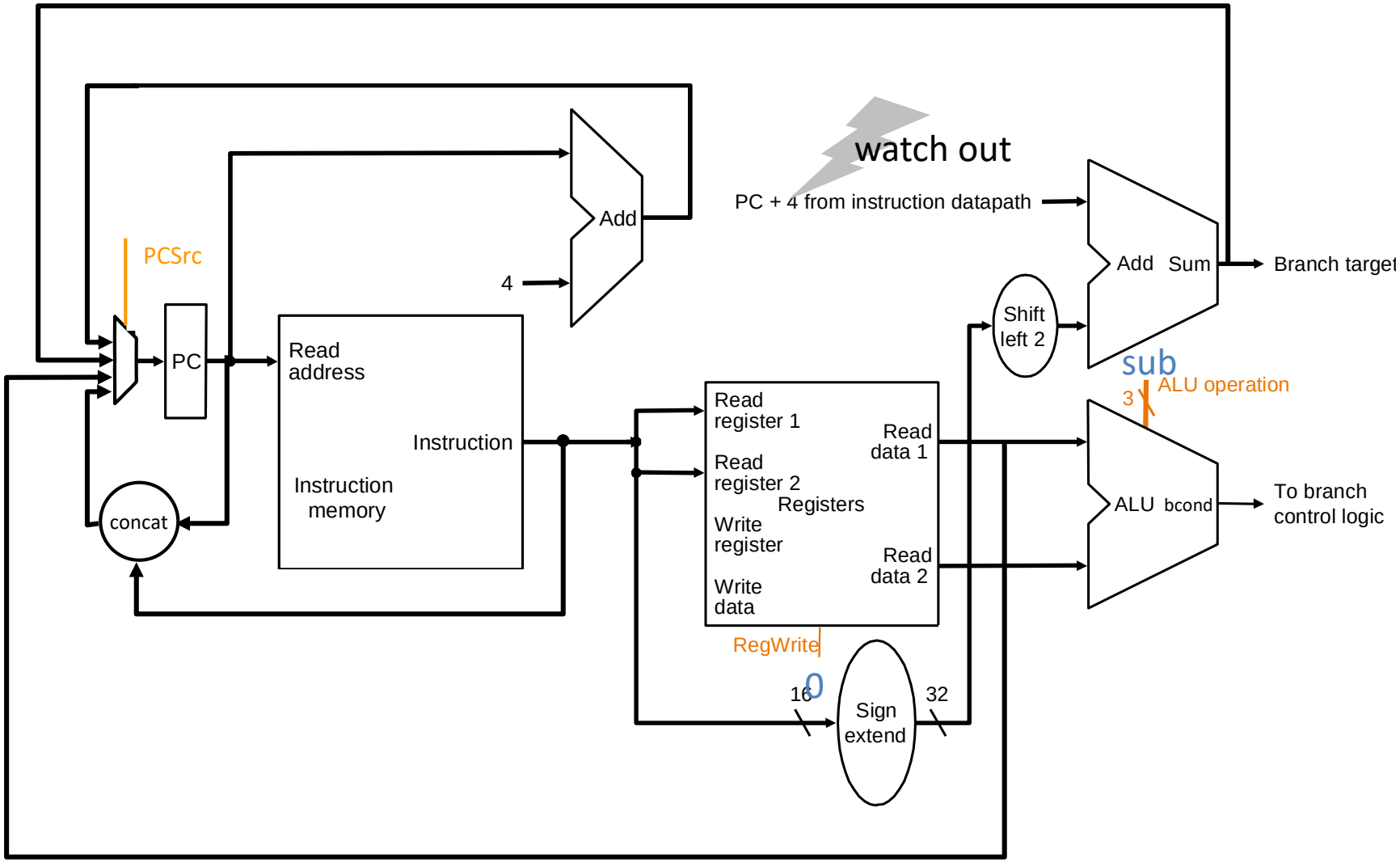
  if MEM[PC]==BEQ rs rt $immediate_{16}$

  target = PC + 4 + sign-extend(immediate) x 4

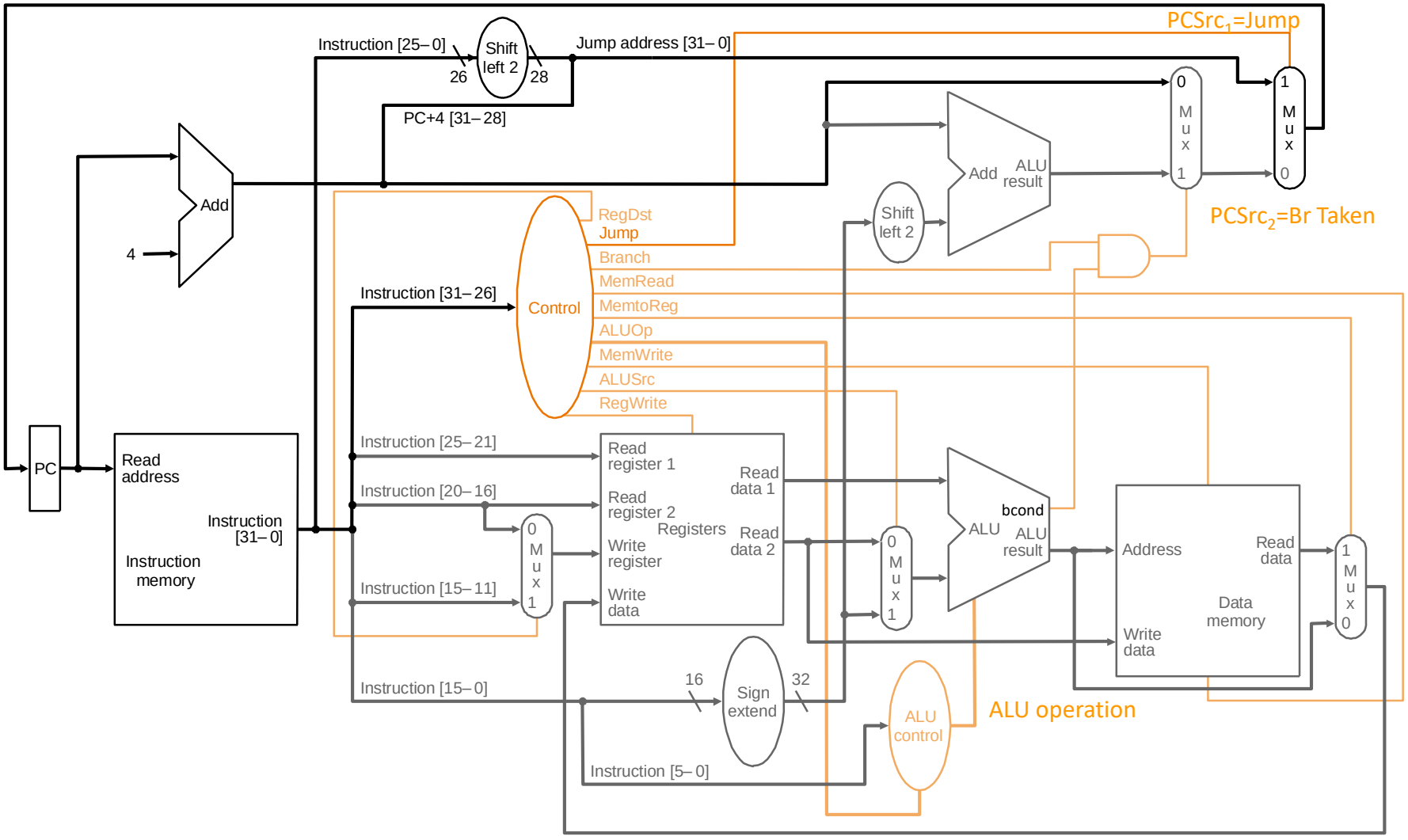
  if GPR[rs]==GPR[rt] then PC ← target

  else PC ← PC + 4

Conditional Branch Datapath (For You to Fix)



Putting It All Together



JAL, JR, JALR omitted