KING'S College LONDON

Hardware Design

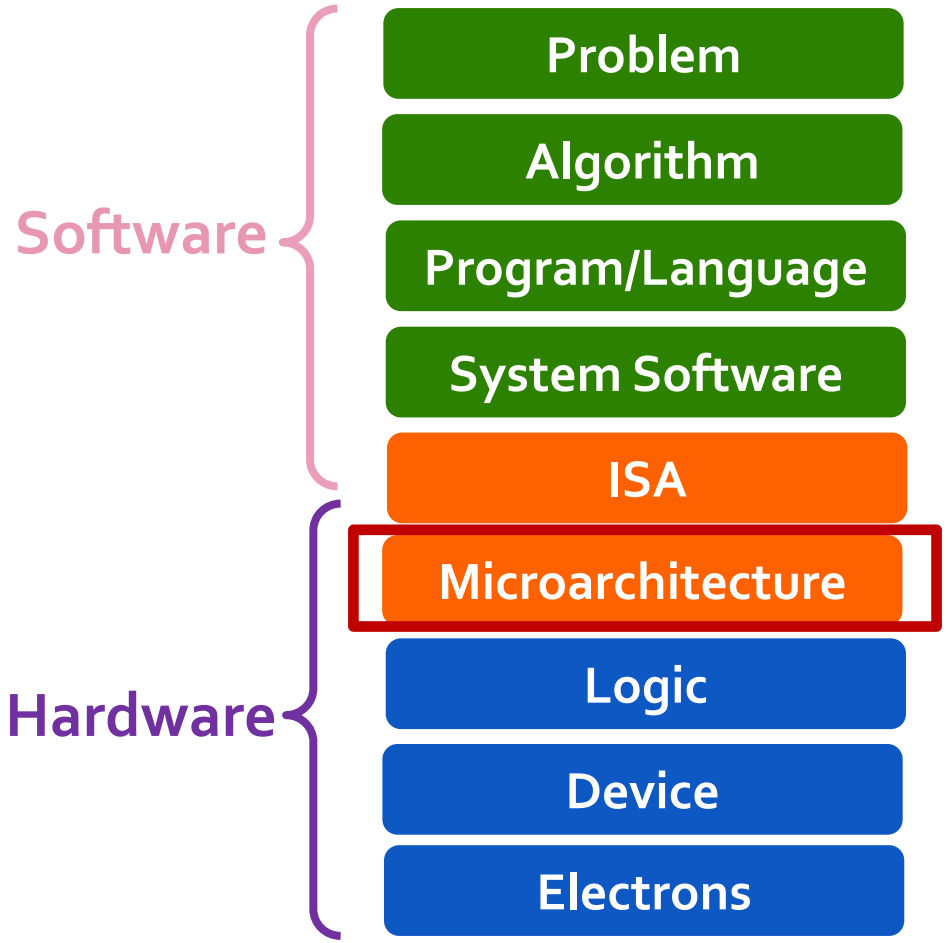
Lecture 4: Single-Cycle Microarchitecture: build the whole CPU once

Dr. Haiyu Mao

19.02.2026

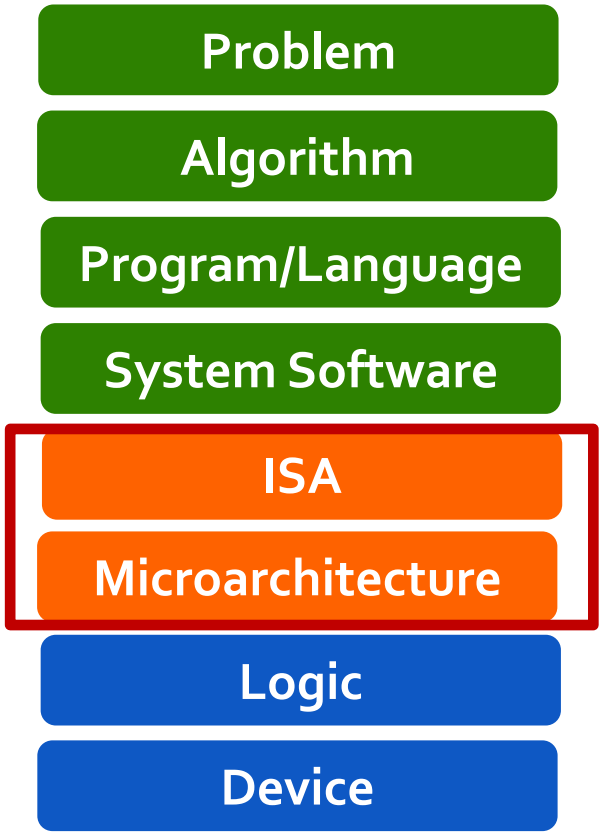
What We Learned & Will Learn

- The von Neumann model
- Instruction Set Architectures (ISA)
- Assembly programming: LC-3 and MIPS
- Microarchitecture: basics
- **Microarchitecture: Single-cycle**
- Microarchitecture: Multi-cycle
- Pipelining
- Cache and Memory



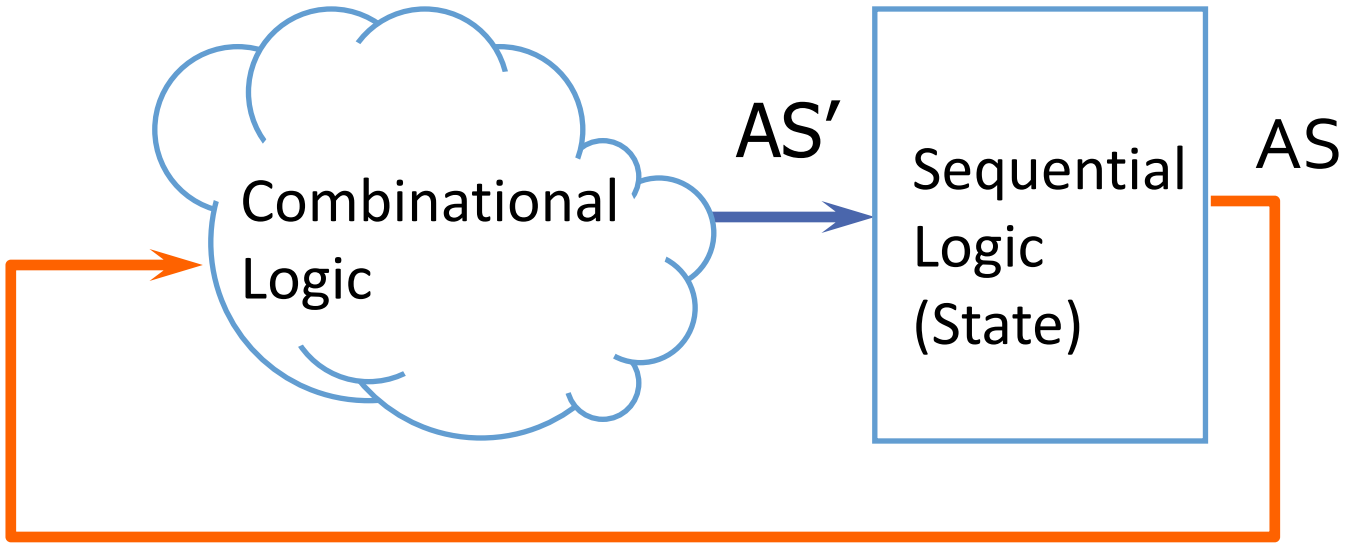
Recall: ISA vs. Microarchitecture

- ISA
  - Agreed upon the interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs
- Microarchitecture
  - **Specific implementation of an ISA**
  - **Not visible to the software**
- Microprocessor
  - **ISA, uarch**, circuits
  - "Computer Architecture" = ISA + microarchitecture



Recall: A Very Basic Instruction Processing Engine

- Single-cycle machine



Recall: Single-cycle vs. Multi-cycle Machines

- Single-cycle machines
  - Each instruction takes a single clock cycle
  - All state updates are made at the end of an instruction's execution
  - Big disadvantage: The slowest instruction determines the cycle time → long clock cycle time

- Multi-cycle machines
  - Instruction processing is broken into multiple cycles/stages
  - State updates can be made during an instruction's execution
  - Architectural state updates are made 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 Viewed Another Way

- Instructions transform Data (AS) to Data' (AS')
- This transformation is done by functional units
  - Units that "operate" on data
- These units need to be told what to do with 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, muxes, decoders, tri-state bufs) 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

Many Ways of Datapath and Control Design

- There are many ways of designing the datapath and control logic

- Example ways
  - Single-cycle, multi-cycle, pipelined, out-of-order datapath & control
  - Single-bus vs. multi-bus datapaths
  - Hardwired/combinational vs. microcoded/microprogrammed control
    - Control signals generated by combinational logic versus
    - Control signals stored in a memory structure

- Control signals and structure depend on the datapath design

Flash-Forward: Performance Analysis

- Execution time of a single instruction
  - **{CPI} x {clock cycle time}** CPI: Cycles Per Instruction
- Execution time of an entire 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

In multi-cycle, we have two degrees of freedom to optimize independently

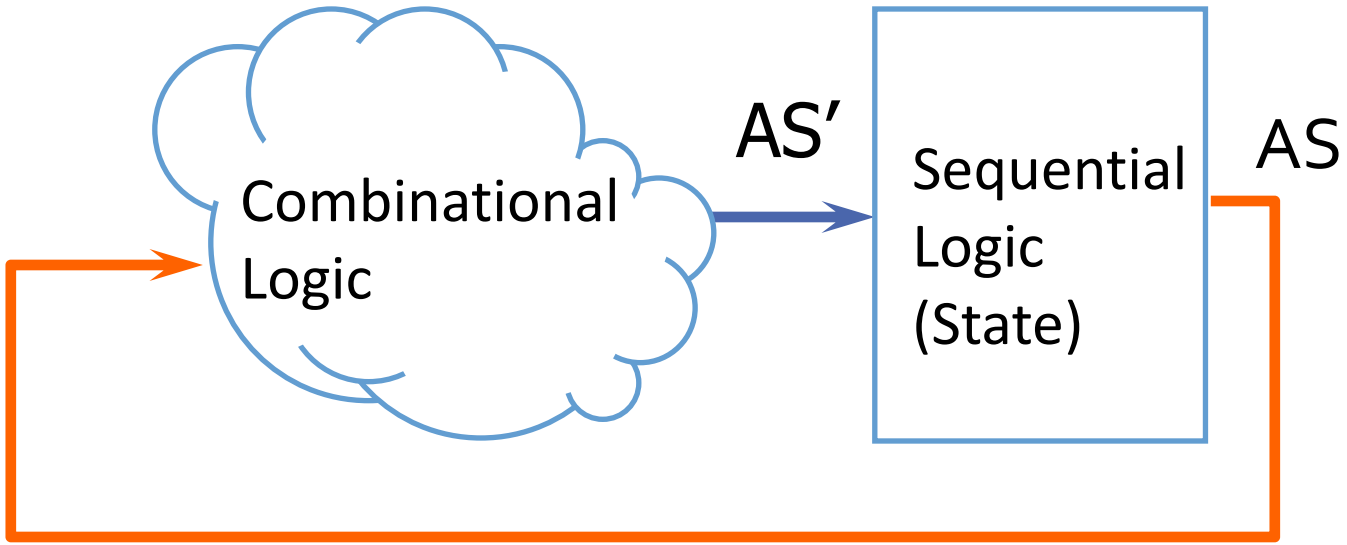
Hardware Design

A Single-Cycle Microarchitecture

From the Ground Up

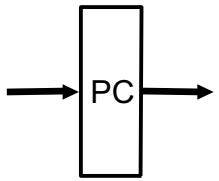
Remember:

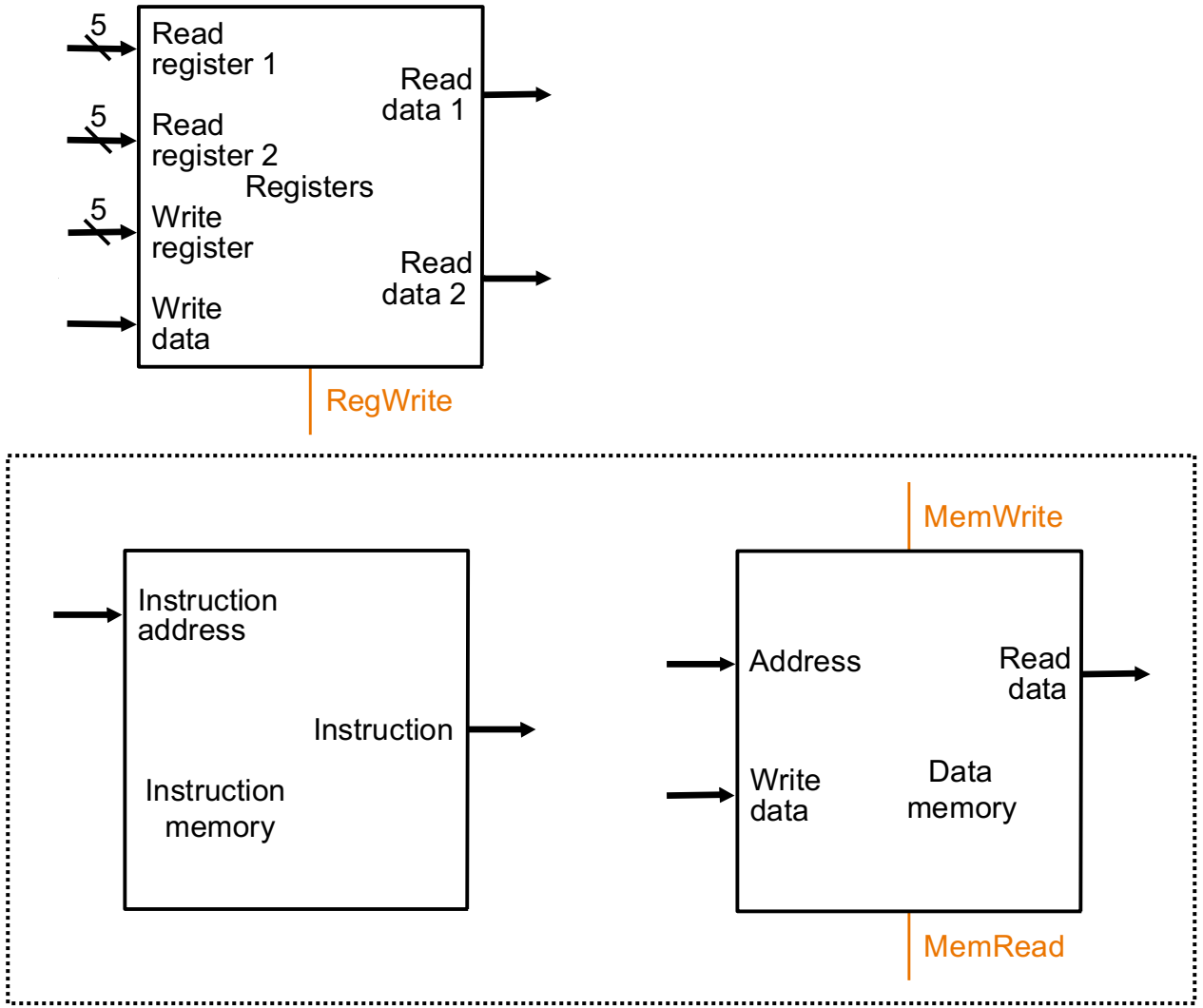
- Single-cycle machine



Let's Start with the State Elements (MIPS)

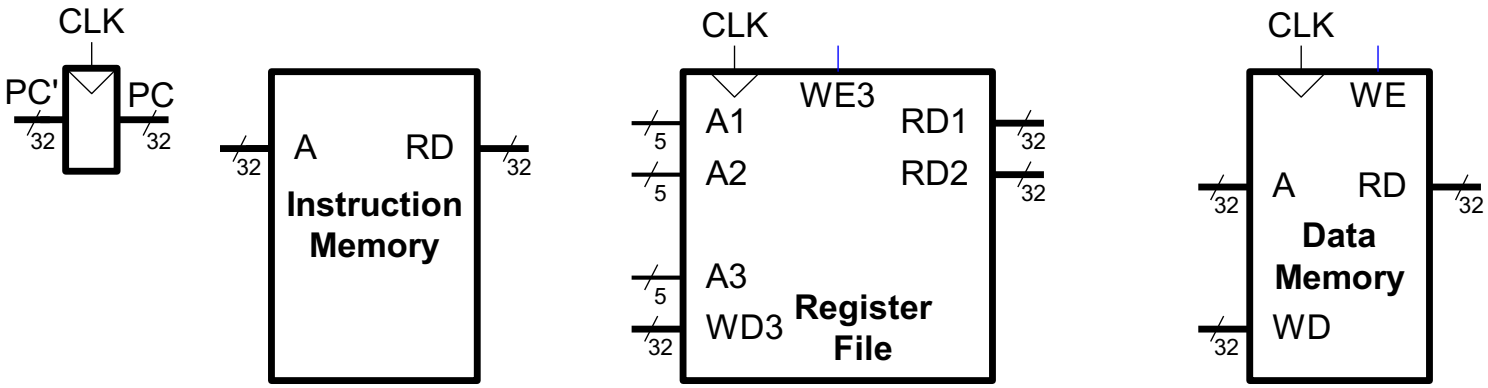
- Data and control inputs





We will use this notation in lectures

MIPS State Elements



- Program counter:

  32-bit register

- Instruction memory:

  Takes input 32-bit address A and reads the 32-bit data (i.e., instruction) from that address to the read data output RD

- Register file:

  The 32-element, 32-bit register file has 2 read ports and 1 write port

- Data memory:

  If the write enable, WE, is 1, it writes 32-bit data WD into the memory location at 32-bit address A on the rising edge of the clock.

  If the write enable is 0, it reads 32-bit data from address A onto RD.

For Now, We Will Assume

- Ultra-fast (i.e., single-cycle) memory and register file

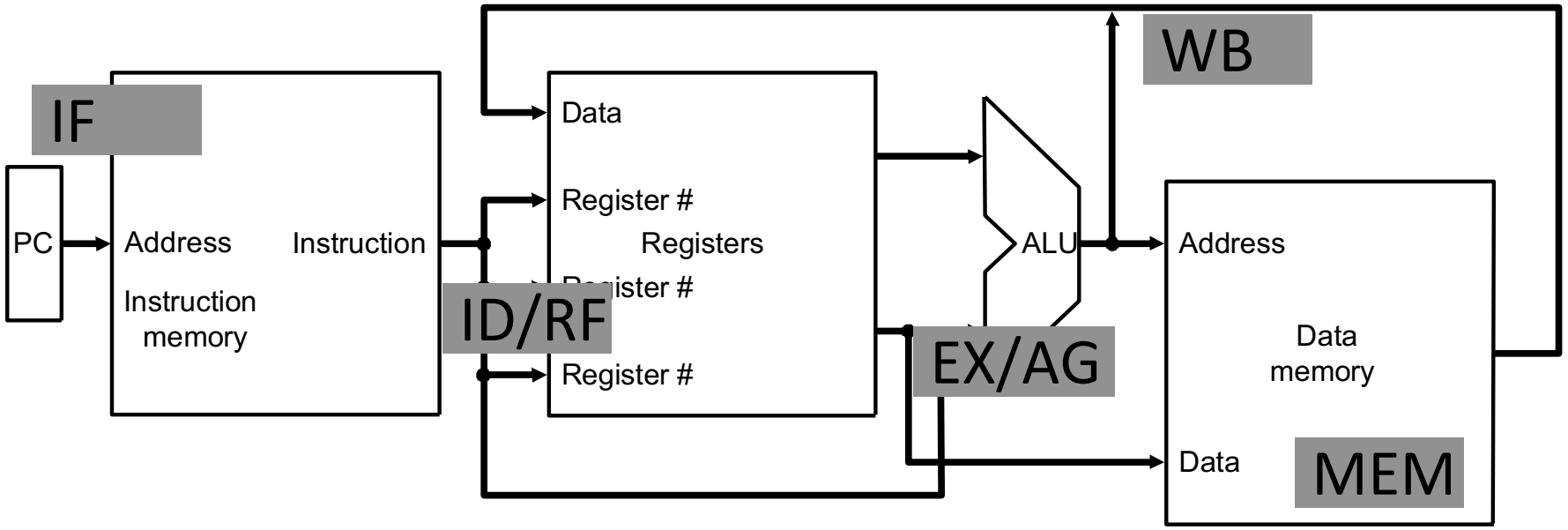
- Combinational read
  - The output of the read data port is a combinational function of the memory contents and the corresponding input address

- Synchronous write
  - The target location is updated at the positive edge clock transition when the write enable signal is asserted
    - Cannot affect read output in-between positive clock edges

- Single-cycle memory
  - Contrast this with a memory that tells when the data is ready
    - i.e., Ready signal: 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)



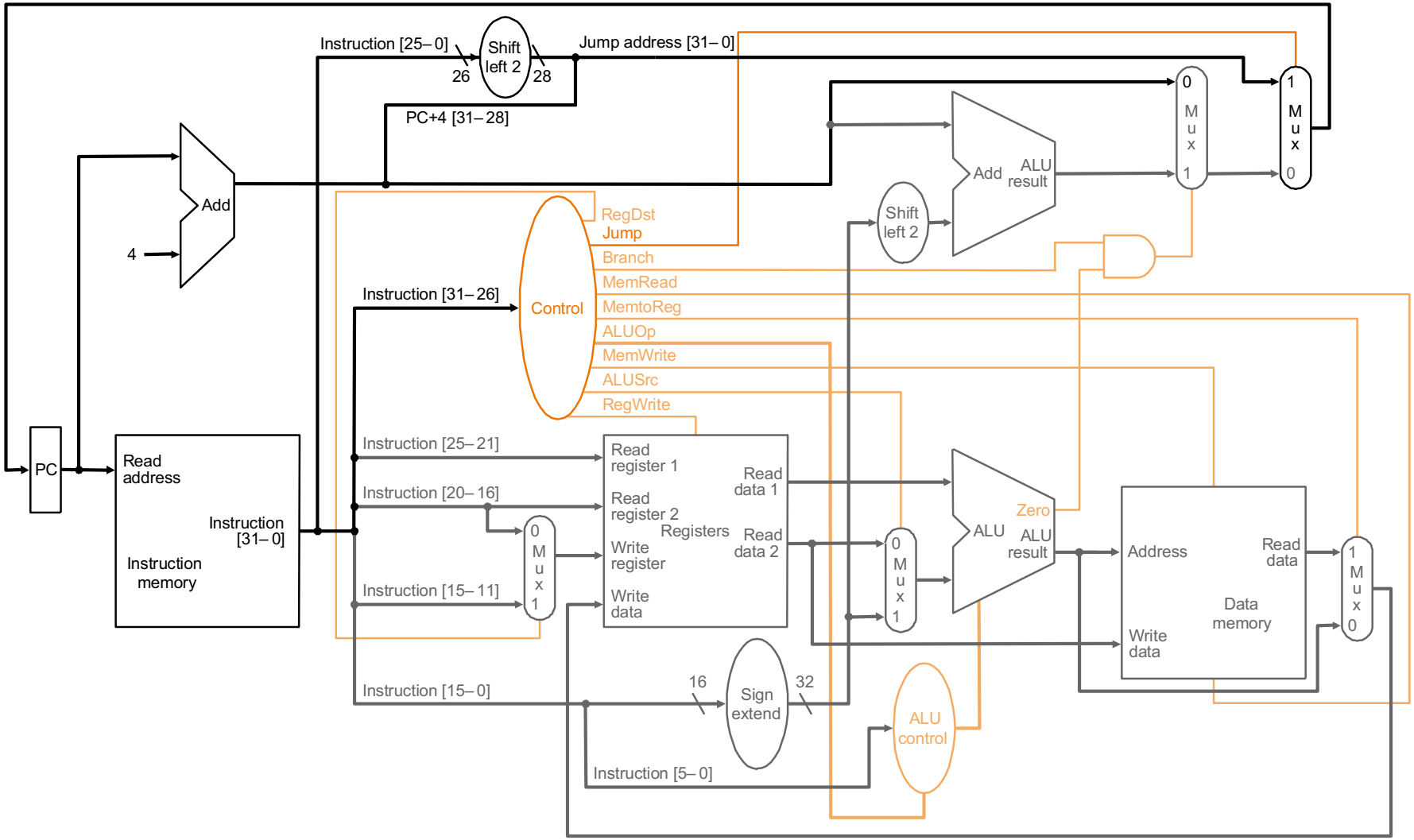
Datapath and Control Logic

We Need to Provide the

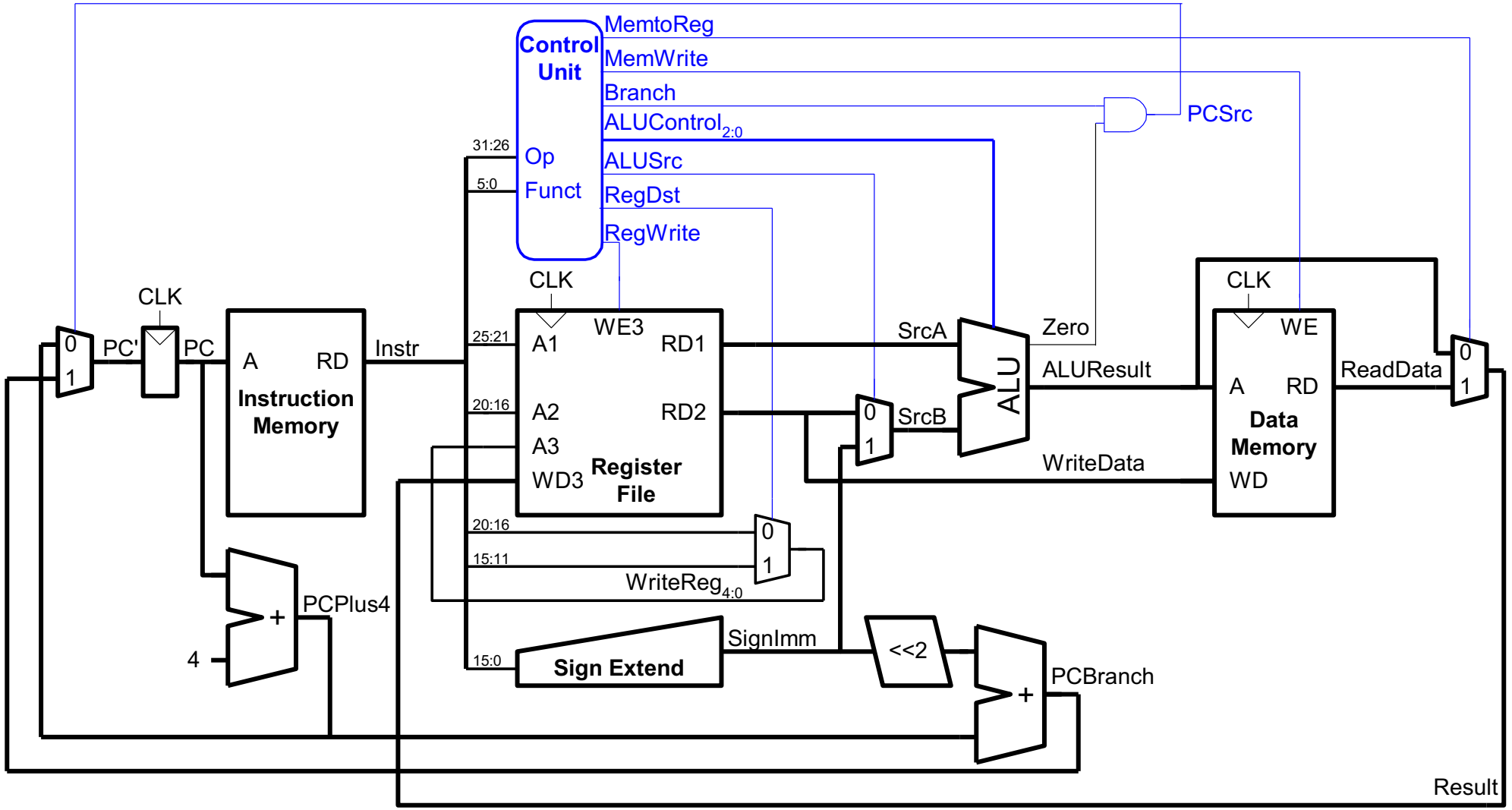
Datapath & Control Logic

to Execute All ISA Instructions

What Is To Come: Single-Cycle MIPS Processor



Another Complete Single-Cycle Processor



Hardware Design

Single-Cycle Datapath for Arithmetic and Logical Instructions

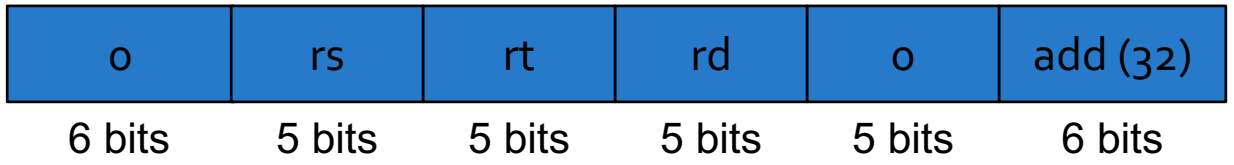
R-Type ALU Instructions

- R-type: 3 register operands

MIPS assembly (e.g., register-register signed addition)

```
add  $s0, $s1, $s2          #$s0=rd, $s1=rs, $s2=rt
```

Machine Encoding

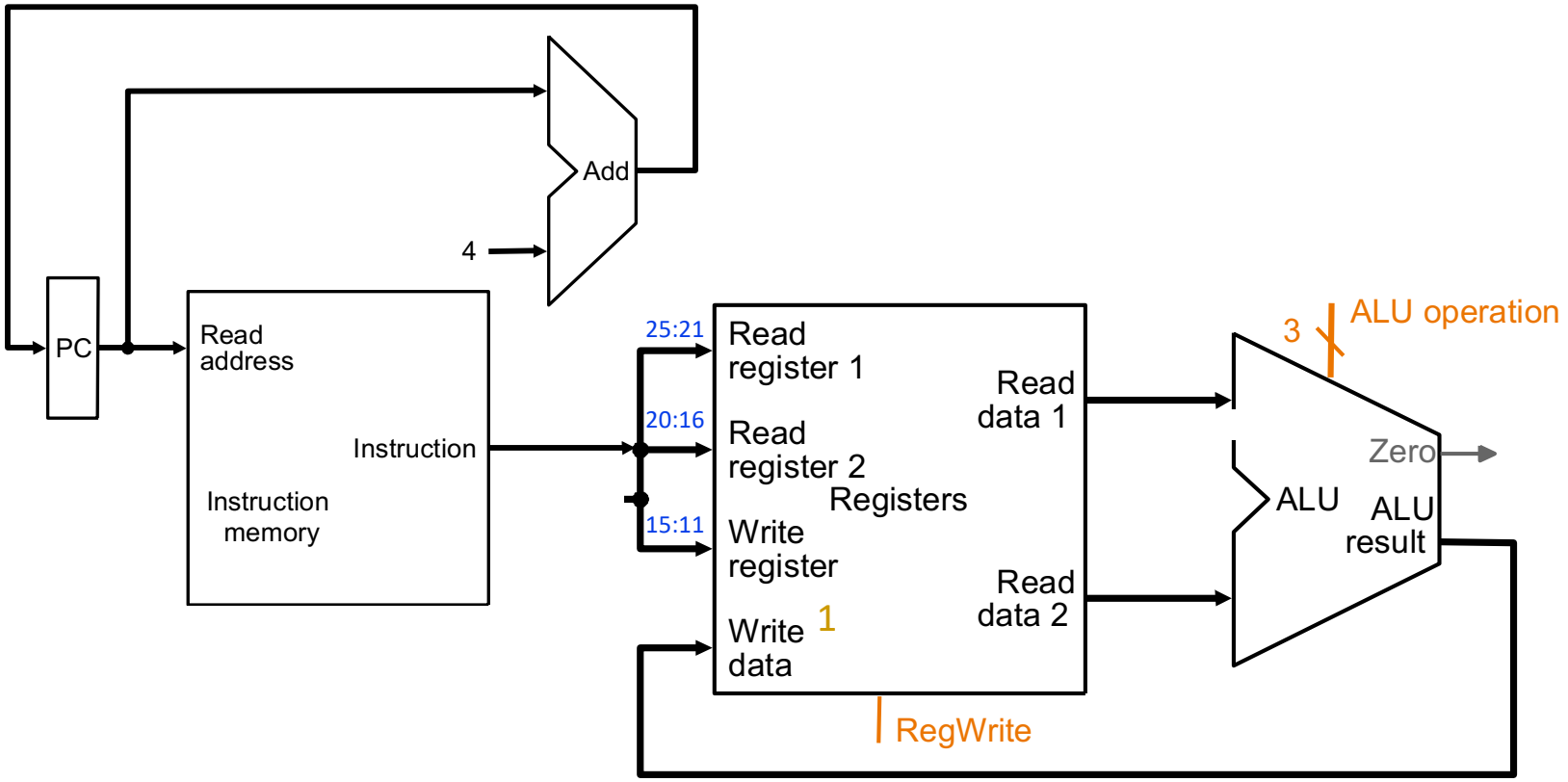


R-Type

- Semantics

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

(R-Type) ALU Datapath

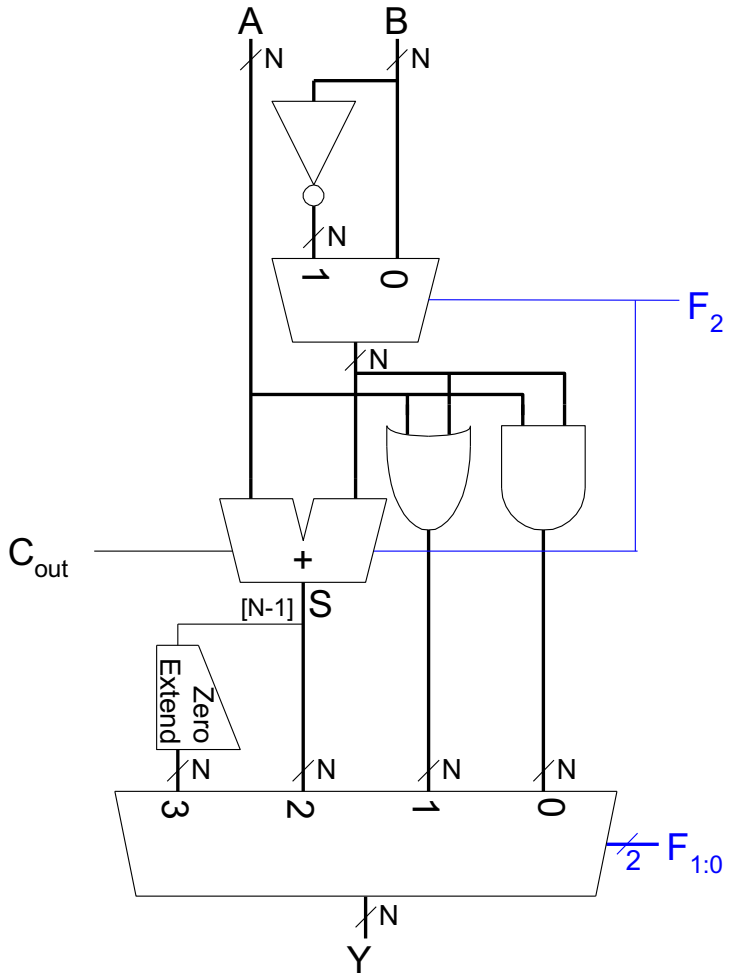



```
if MEM[PC] == ADD rd rs rt
        GPR[rd] ← GPR[rs] + GPR[rt]
        PC ← PC + 4
```

Combinational state update logic

Example: ALU Design

- ALU operation ($F_{2:0}$) comes from the control logic



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

I-Type ALU Instructions

- I-type: 2 register operands and 1 immediate

MIPS assembly (e.g., register-immediate signed addition)

```
addi $s0, $s1, 5                #$s0=rt, $s1=rs
```

Machine Encoding

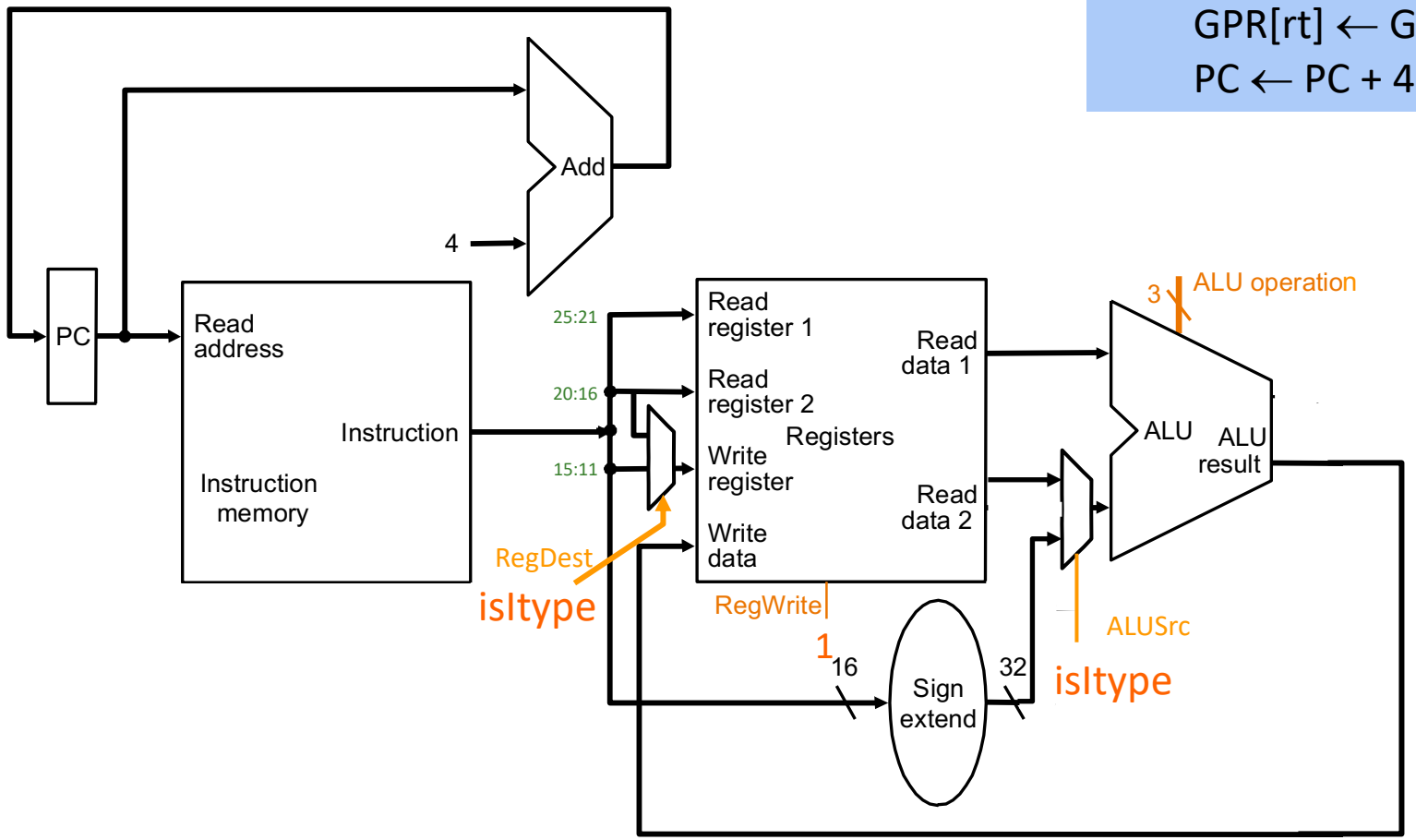
I-Type

- Semantics

```
if MEM[PC] == addi rs rt immediate
    PC ← PC + 4
    GPR[rt] ← GPR[rs] + sign-extend(immediate)
```

Datapath for R- and I-Type ALU Insts.

if MEM[PC] == ADDI rt rs immediate
GPR[rt] ← GPR[rs] + sign-extend (immediate)
PC ← PC + 4



Combinational state update logic

Hardware Design

Single-Cycle Datapath for Data Movement Instructions

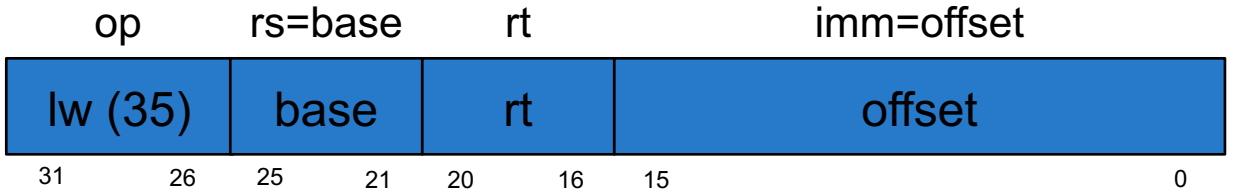
Load Instructions

- Load 4-byte word

MIPS assembly

```
lw   $s3, 8($s0)                    #$s0=rs, $s3=rt
```

Machine Encoding

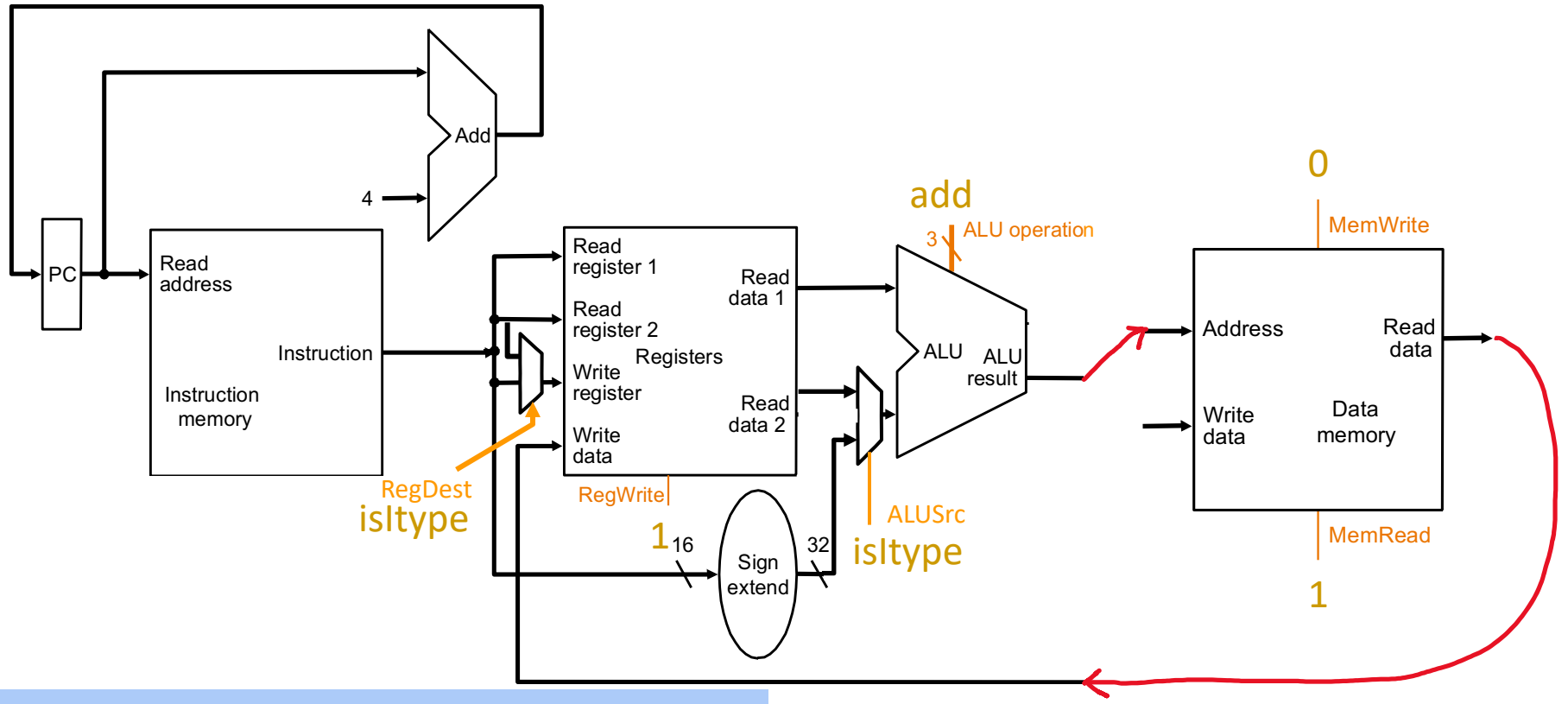


I-Type

- Semantics

if MEM[PC] == lw rt offset_{16} (base)
    PC ← PC + 4
    address = sign-extend(offset) + GPR(base)
    GPR[rt] ← MEM[address]

lw Datapath



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

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

GPR[rt] ← MEM[address]

PC ← PC + 4

IF	ID	EX	MEM	WB

Combinational state update logic

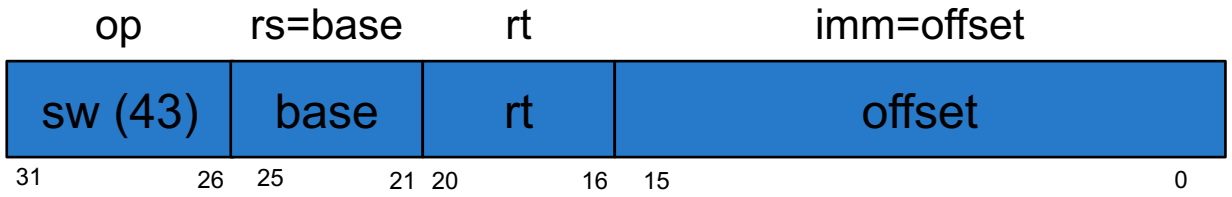
Store Instructions

- Store 4-byte word

MIPS assembly

```
sw    $s3, 8($s0)                    #$s0=rs, $s3=rt
```

Machine Encoding



I-Type

- Semantics

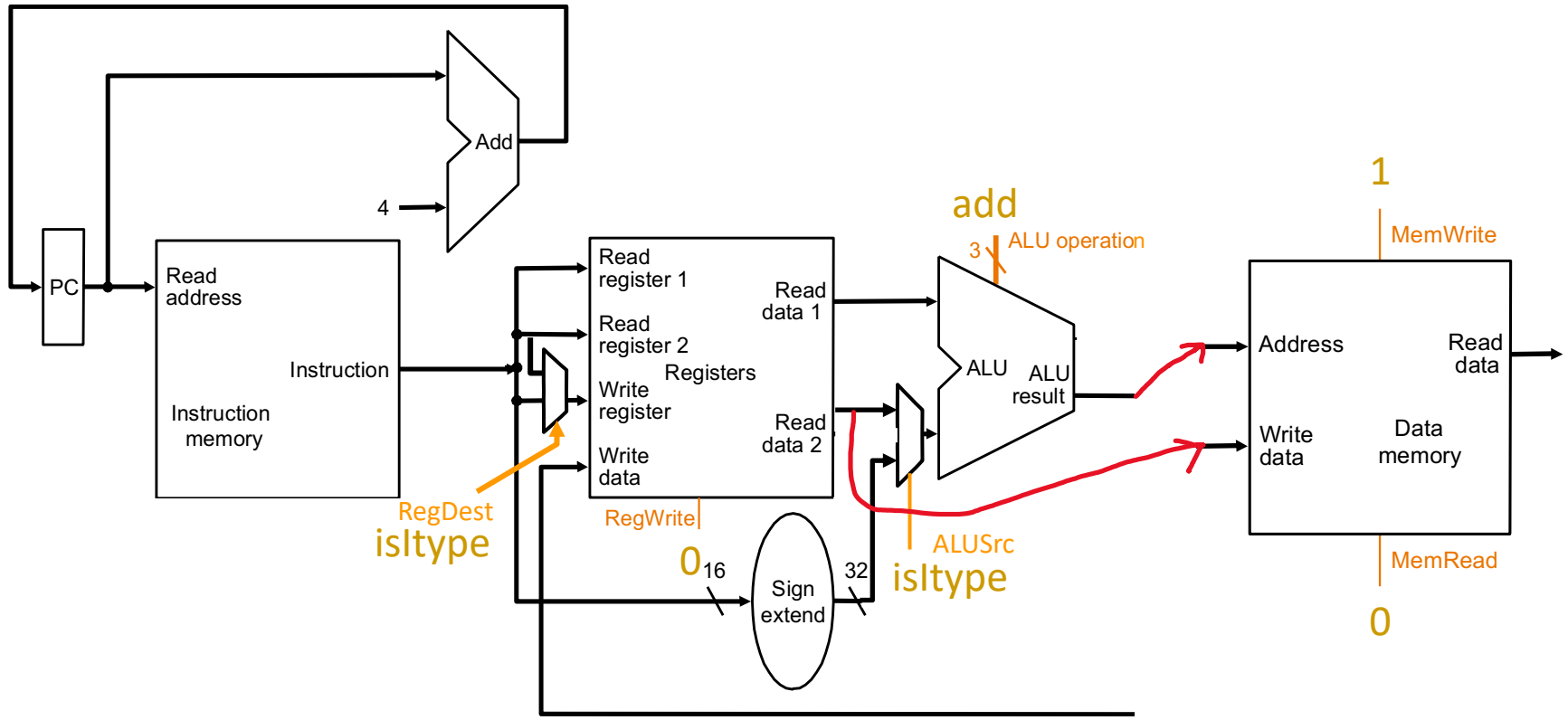
if Mem[PC] == sw rt offset_{16} (base)

PC ← PC + 4

address = sign-extend(offset) + GPR(base)

MEM[address] ← GPR[rt]

SW Datapath

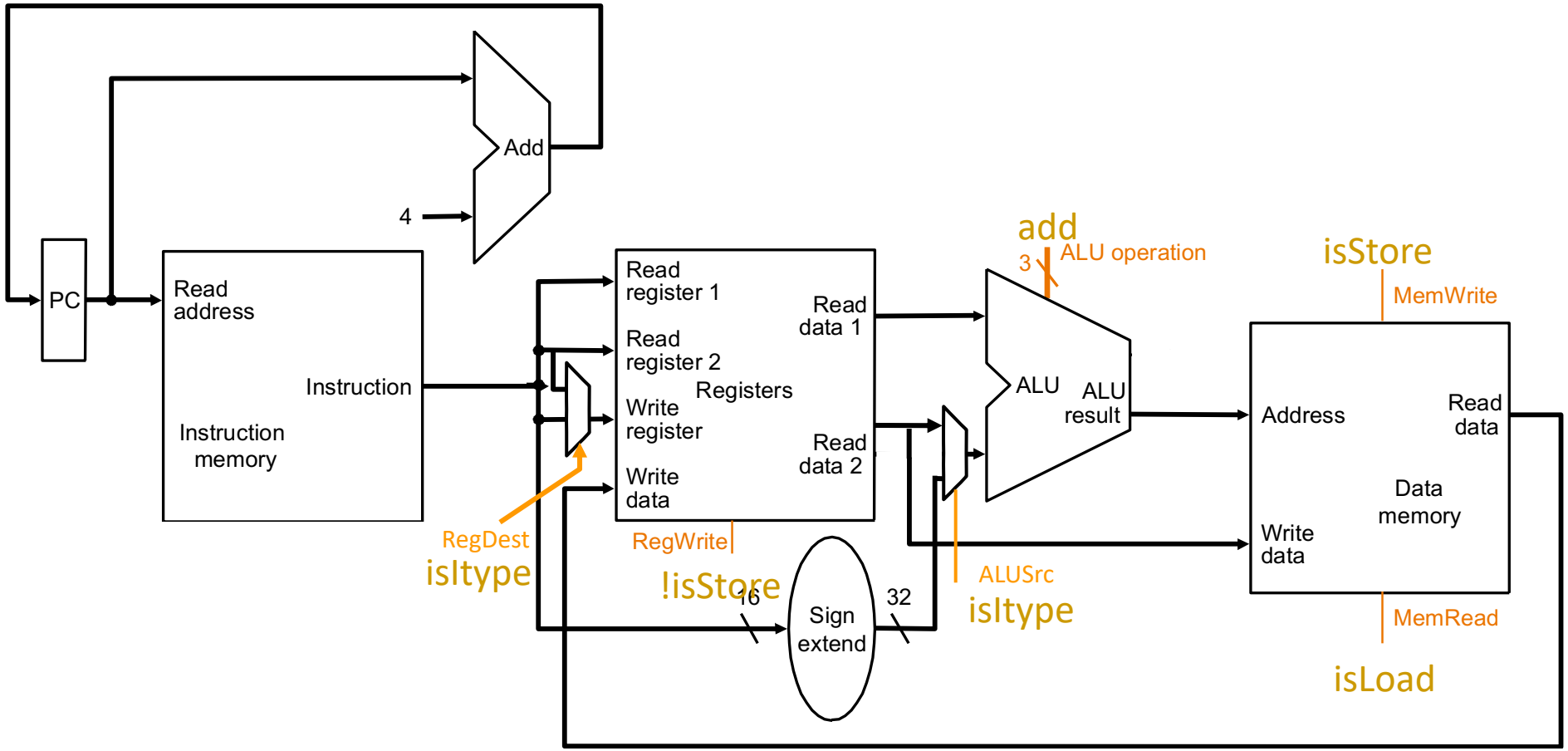


if MEM[PC]==SW rt $offset_{16}$ (base)
address = sign-extend(offset) + GPR[base]
MEM[address] ← GPR[rt]
PC ← PC + 4

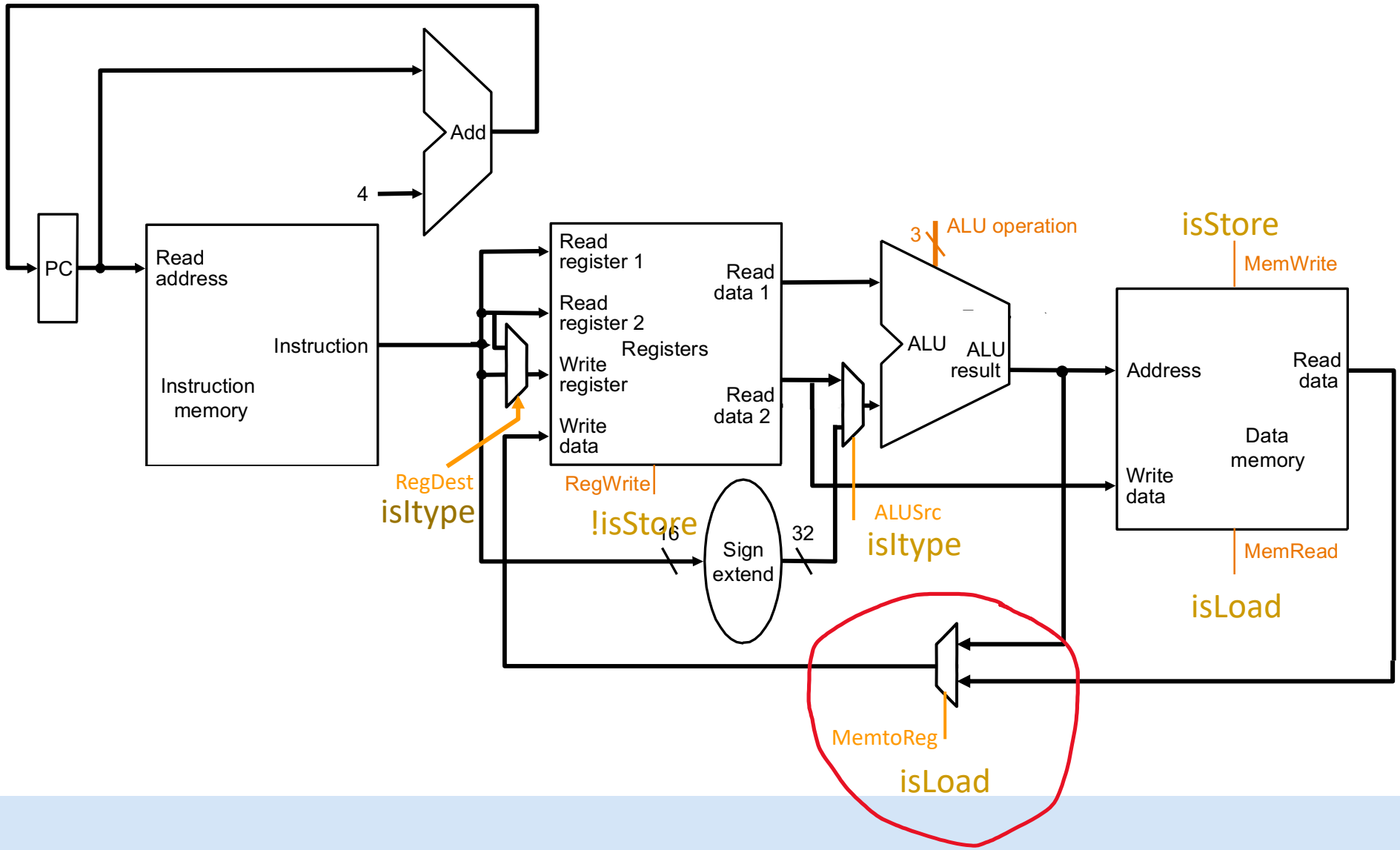
IF	ID	EX	MEM	WB

Combinational state update logic

Load-Store Datapath



Datapath for Non-Control-Flow Instructions



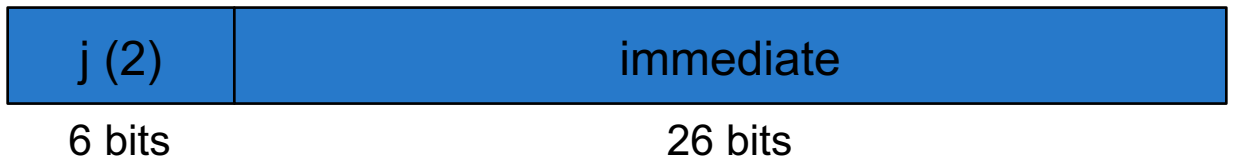
Hardware Design

Single-Cycle Datapath for Control Flow Instructions

Jump Instruction

- Unconditional branch or jump

```
j   target
```



J-Type

  - 2 = opcode
  - immediate (target) = target address

- Semantics

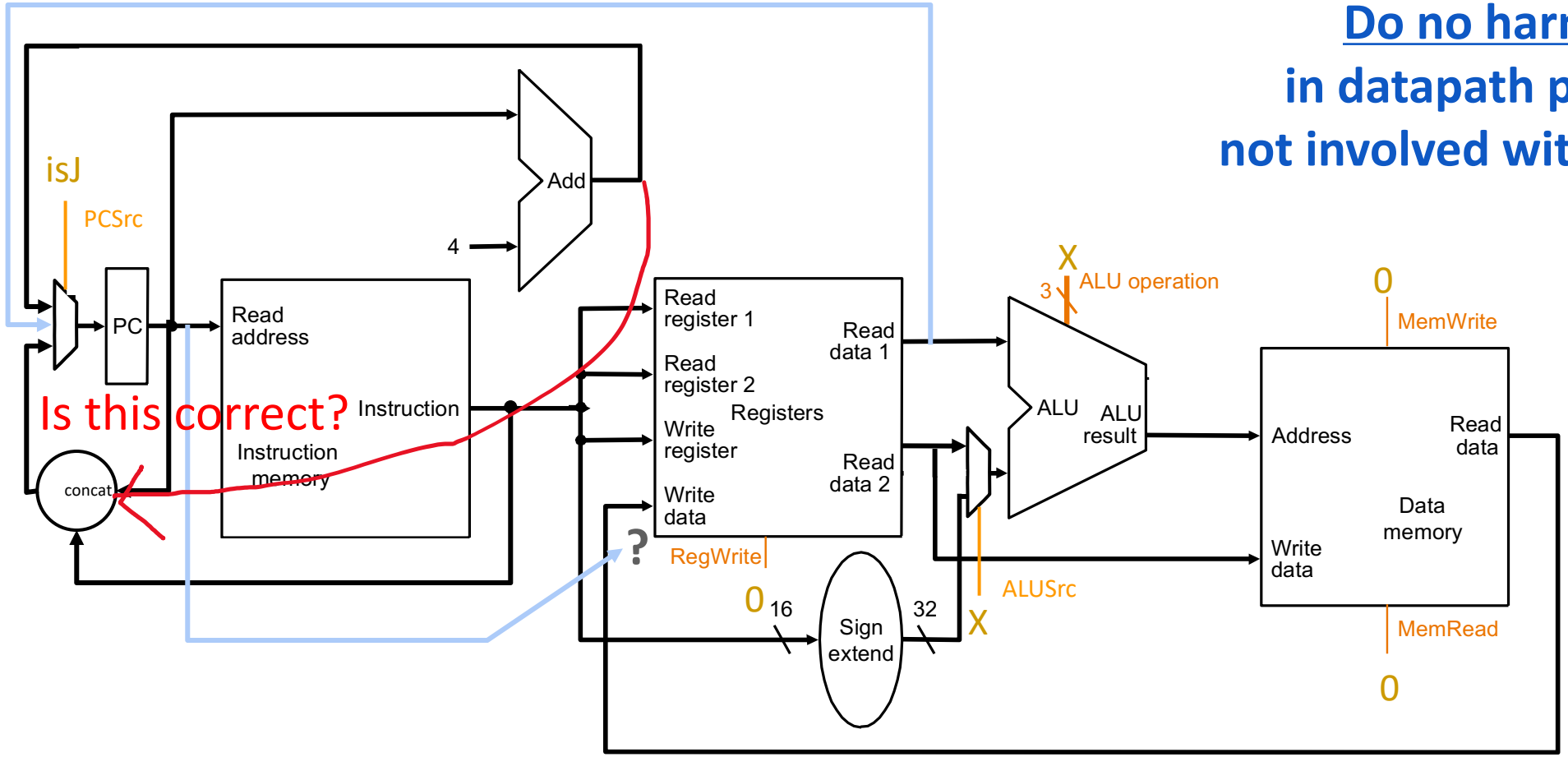
if MEM[PC]== j $immediate_{26}$

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

PC ← target

† This is the incremented PC

Unconditional Jump Datapath



Do no harm
in datapath parts
not involved with jump

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

What about JR, JAL, JALR?

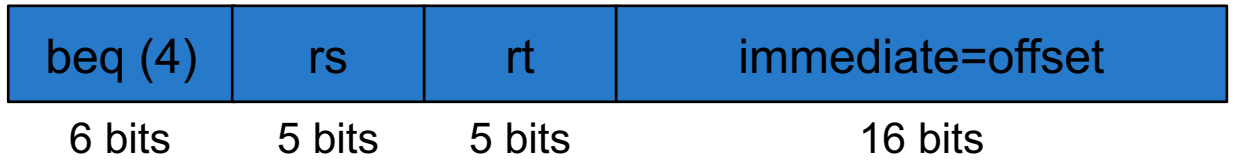
Other Jumps in MIPS

- jr: jump register
  - Semantics
  - if MEM[PC]== jr rs
    - PC ← GPR(rs)
- jal: jump and link (function calls)
  - Semantics
  - if MEM[PC]== jal immediate_{26}
    - $ra ← PC + 4
    - target = { PC † [31:28], immediate_{26}, 2’ boo }
    - PC ← target
- jalr: jump and link register
  - Semantics
  - if MEM[PC]== jalr rs
    - $ra ← PC + 4
    - PC ← GPR(rs)

Conditional Branch Instructions

- beq (Branch if Equal)

```
beq   $s0, $s1, offset                    #$s0=rs,$s1=rt
```



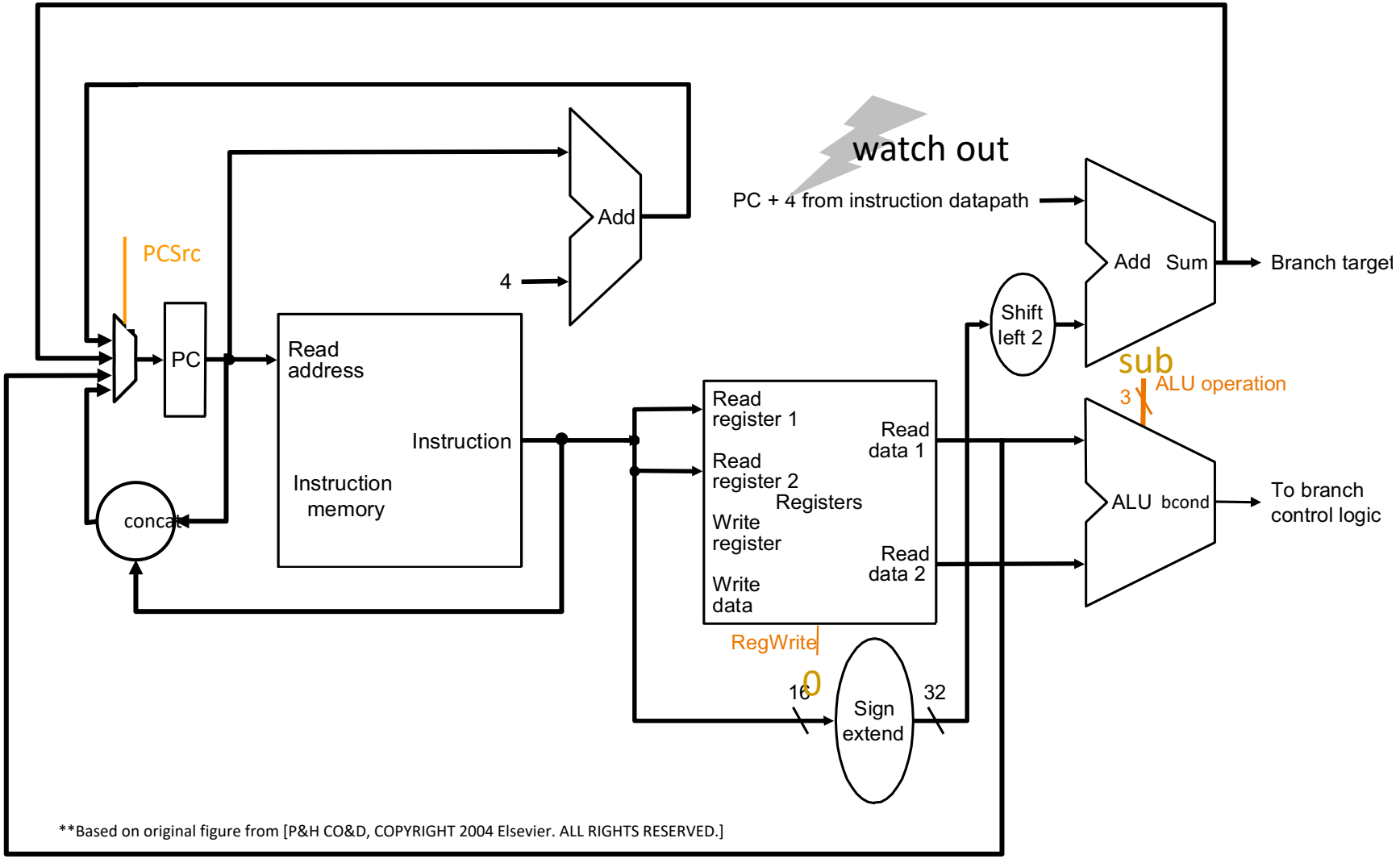
I-Type

- Semantics (assuming no branch delay slot)

```
if MEM[PC] == beq rs rt immediate16
    target = PC+ + sign-extend(immediate) x 4
    if GPR[rs]==GPR[rt] then PC ← target
    else PC ← PC + 4
```

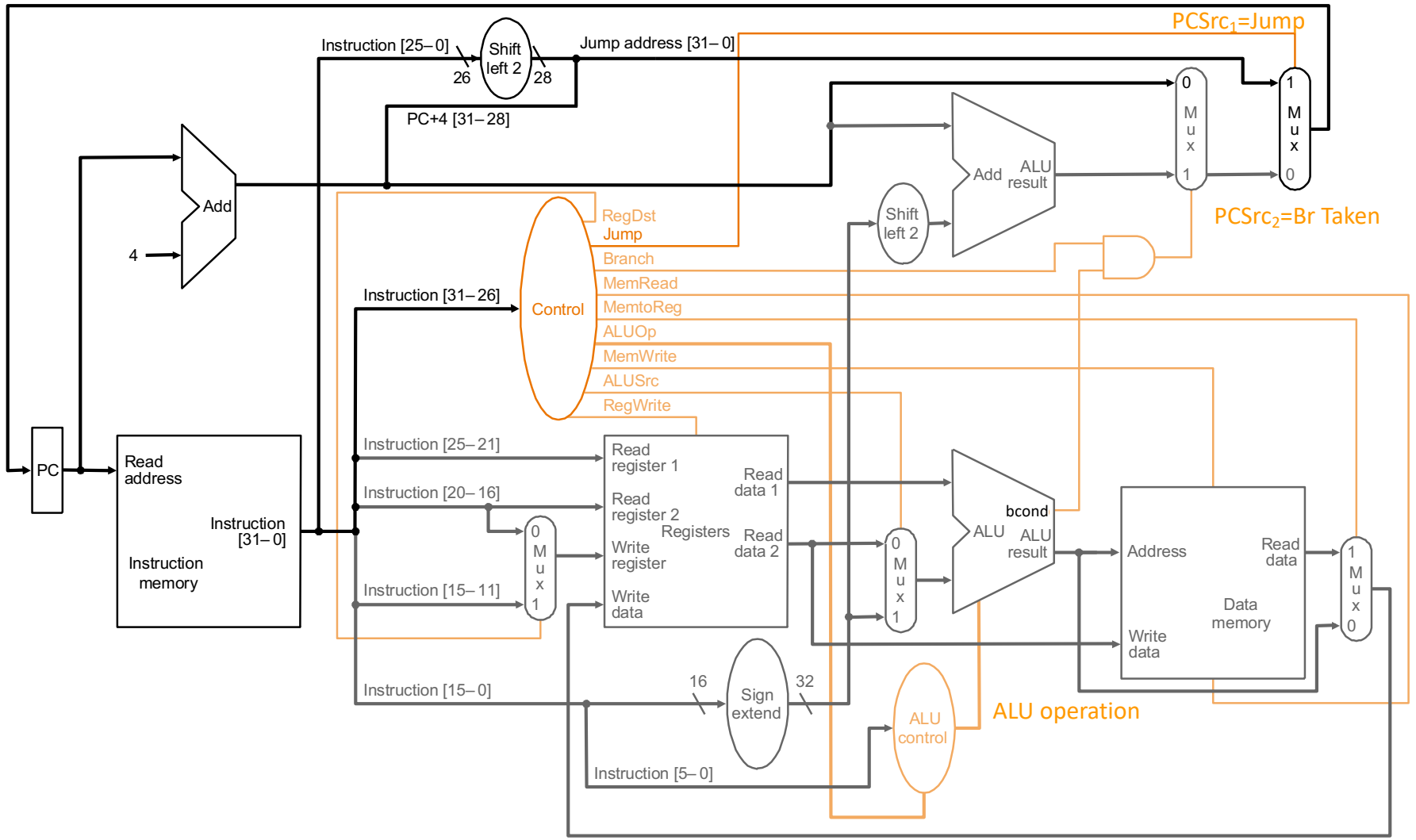
  - Variations: beq, bne, blez, bgtz

Conditional Branch Datapath (for you to finish)



**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

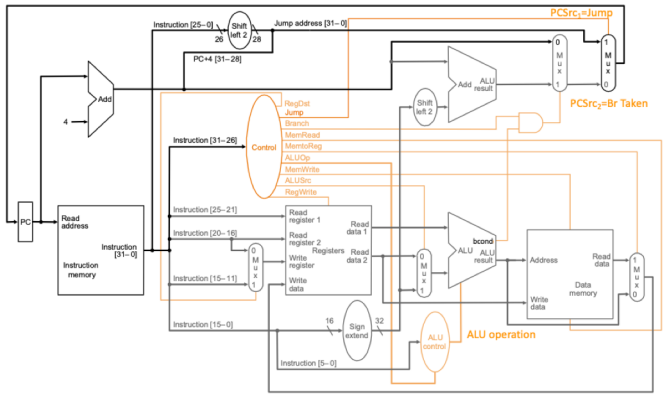
Putting It All Together



JAL, JR, JALR omitted

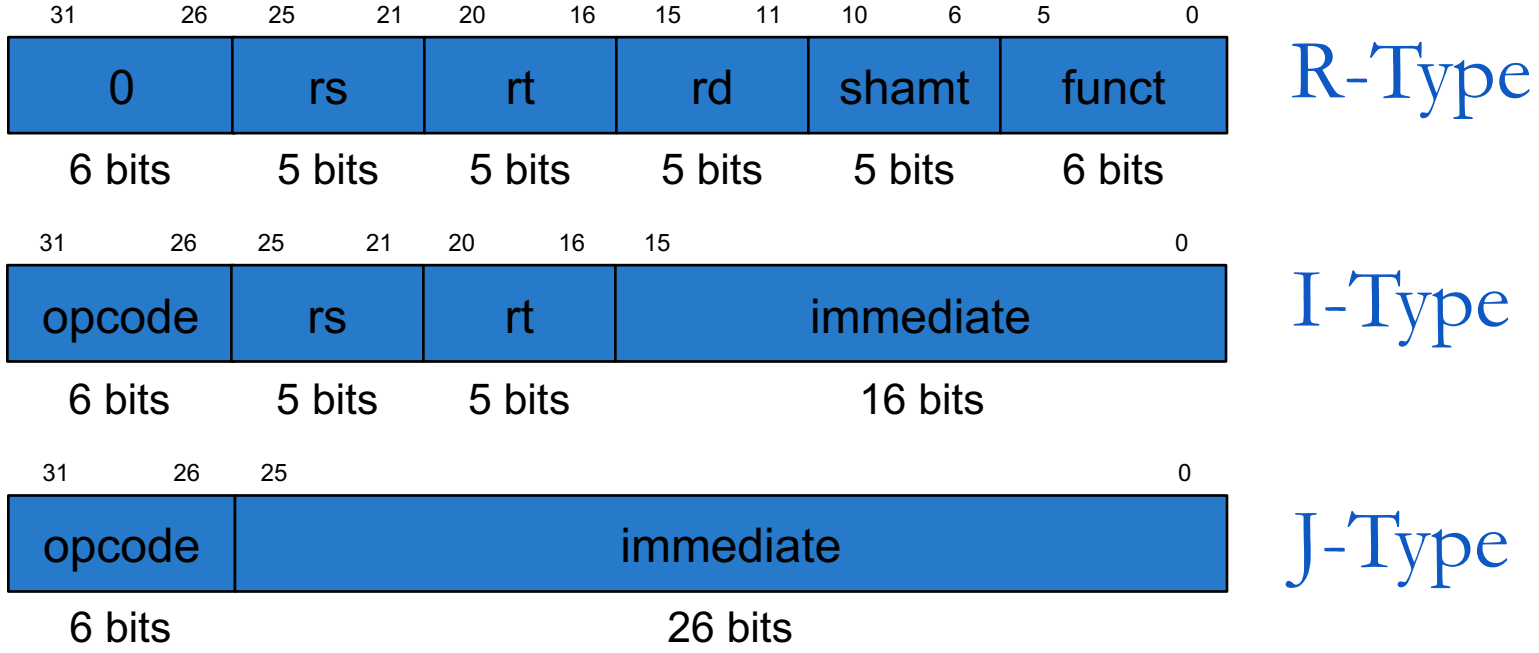
Hardware Design

Single-Cycle Control Logic



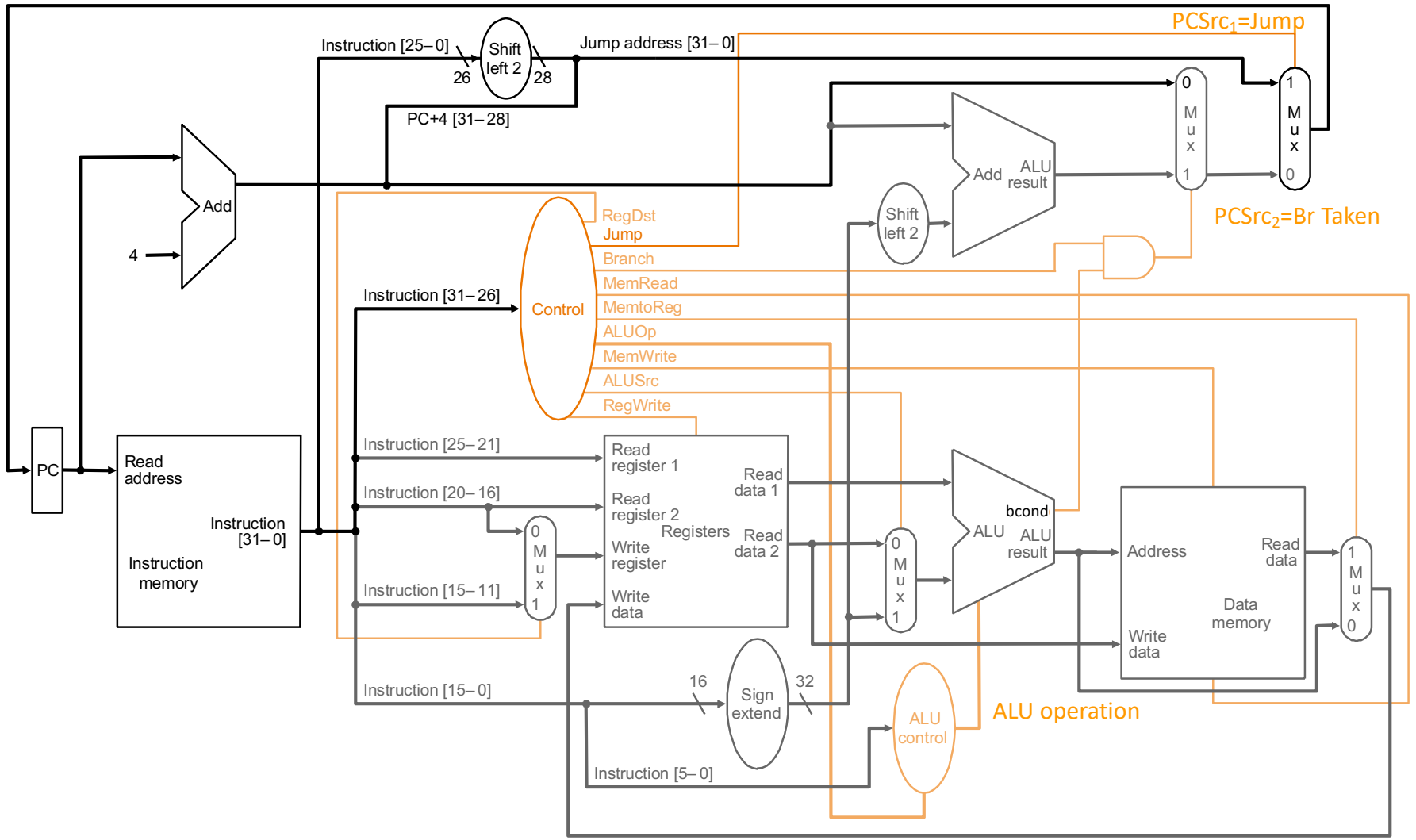
Single-Cycle Hardwired Control

- **As a combinational function** of Inst=MEM[PC]



- Consider
  - All R-type and I-type ALU instructions
  - lw and sw
  - beq, bne, blez, bgtz
  - j, jr, jal, jalr omitted

Generate Control Signals (in Orange Color)



JAL, JR, JALR omitted

Single-Bit Control Signals (I)

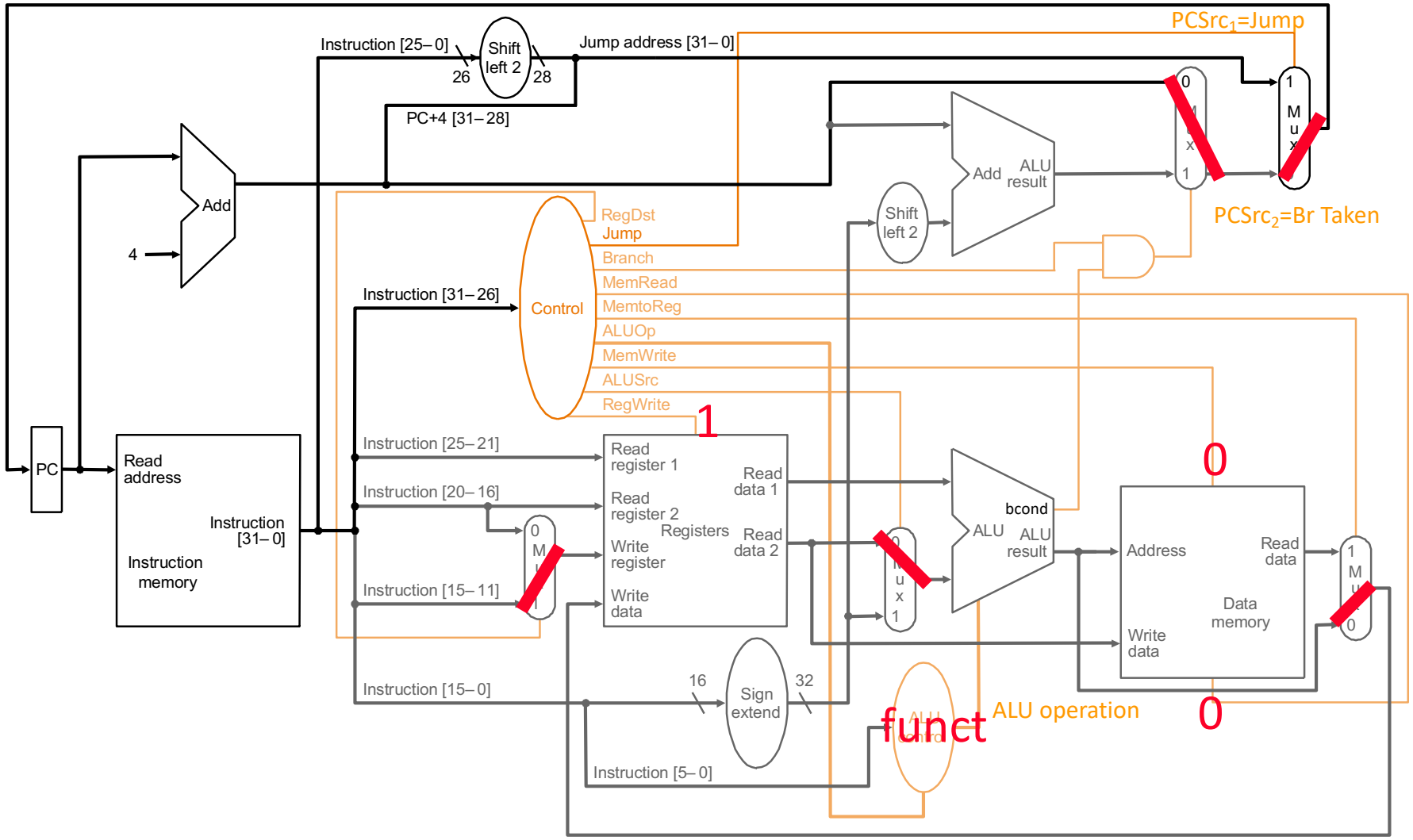
	When De-asserted	When asserted	Equation
RegDst	GPR write select according to **rt**, i.e., inst[20:16]	GPR write select according to **rd**, i.e., inst[15:11]	opcode==0
ALUSrc	2nd ALU input from 2nd **GPR** read port	2nd ALU input from sign-extended 16-bit **immediate**	(opcode!=0) && (opcode!=BEQ) **&&** (opcode!=BNE)
MemtoReg	Steer **ALU result** to GPR write port	Steer **memory output** to GPR write port	opcode==LW
RegWrite	GPR **write disabled**	GPR **write enabled**	(opcode!=SW) && (opcode!=Bxx) && (opcode!=J) && (opcode!=JR))

JAL and JALR require additional RegDst and MemtoReg options

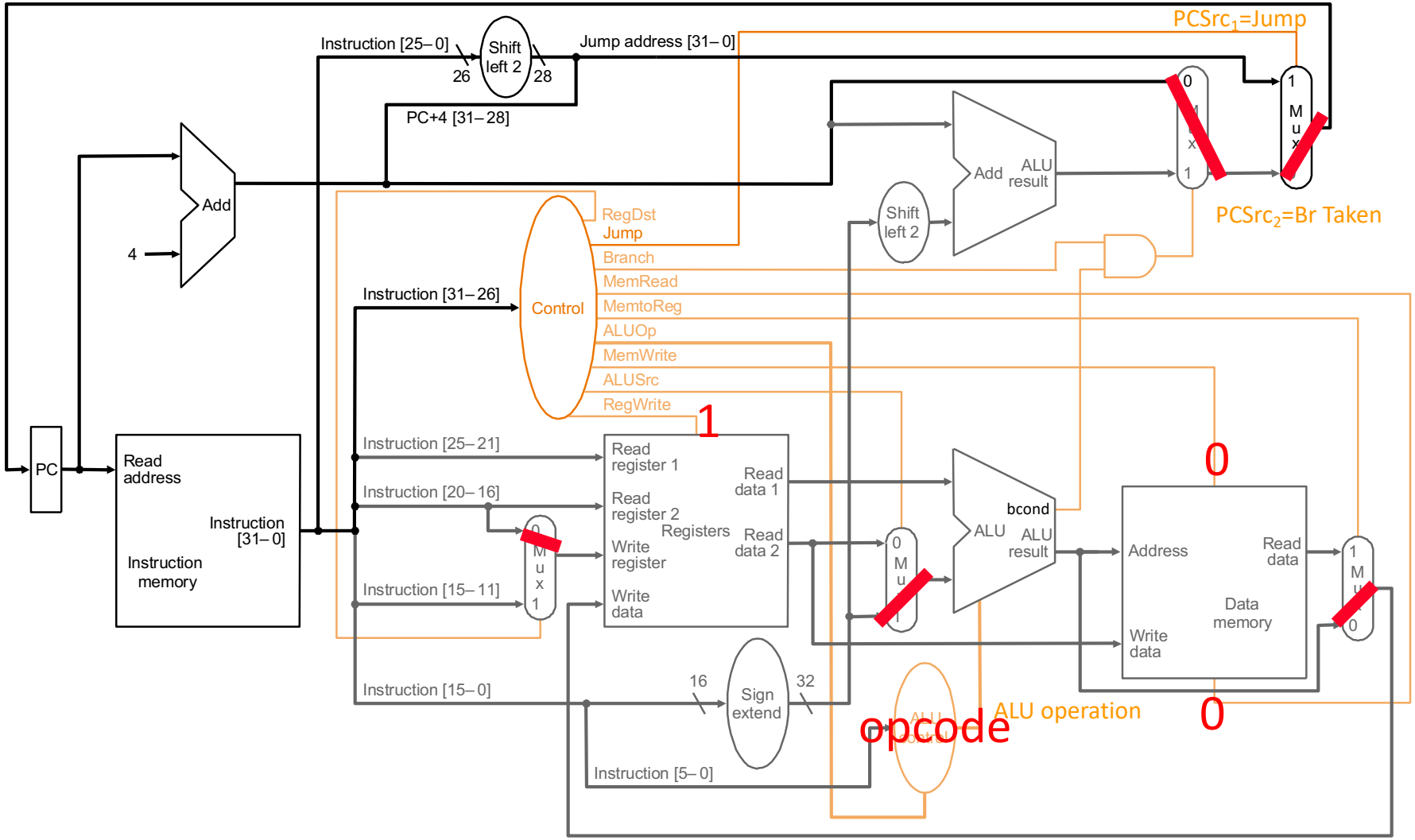
Single-Bit Control Signals (II)

	When De-asserted	When asserted	Equation
MemRead	Memory **read disabled**	Memory **read port returns load value**	opcode==LW
MemWrite	Memory **write disabled**	Memory **write enabled**	opcode==SW
$PCSrc_1$	According to **$PCSrc_2$**	next PC is based on **26-bit immediate jump target**	(opcode==J) || (opcode==JAL)
$PCSrc_2$	next PC = **PC + 4**	next PC is based on **16-bit immediate branch target**	(opcode==Bxx) && "bcond is satisfied"

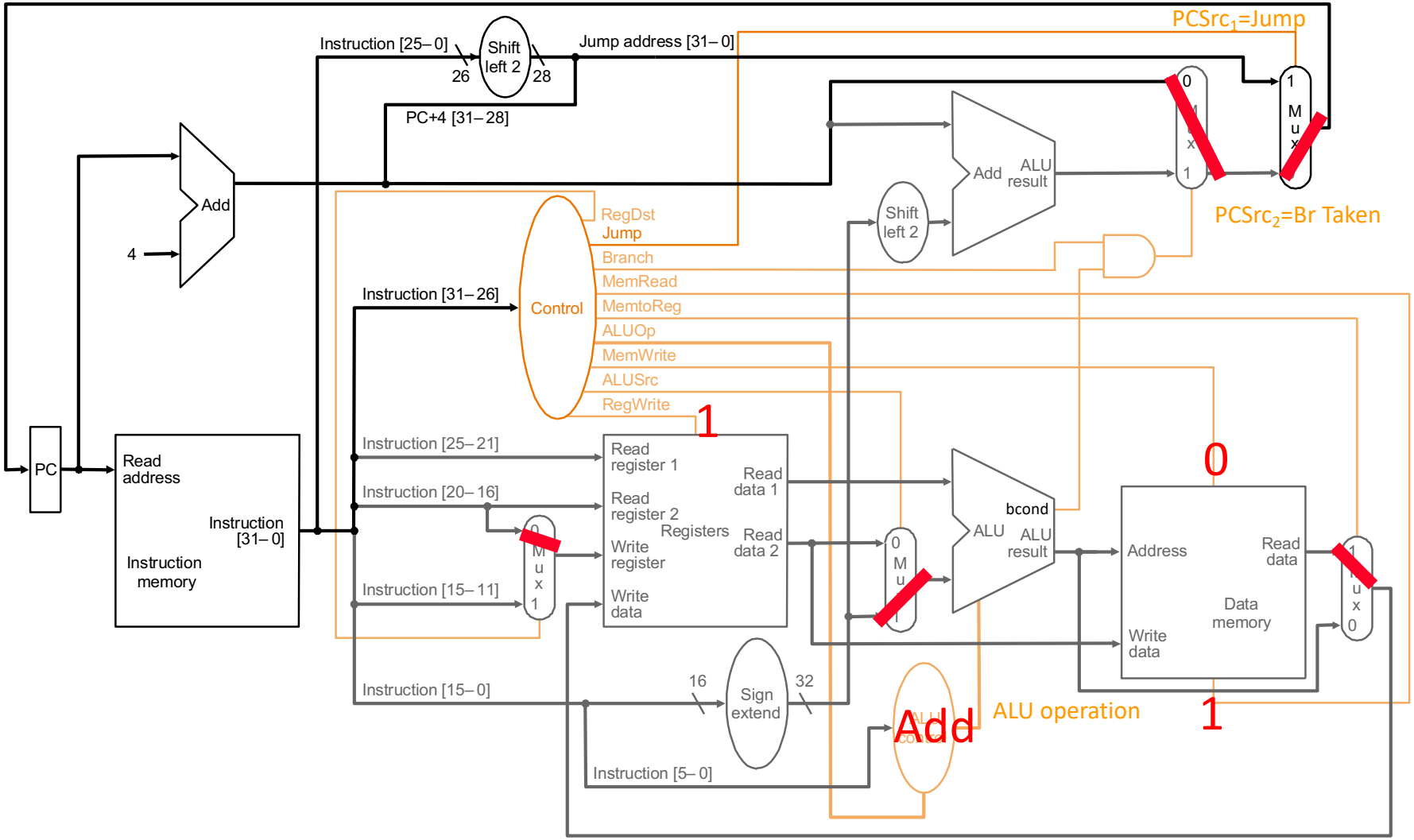
R-Type ALU


PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
0
M
u
x
1
1
M
u
x
Add
ALU result
Shift left 2
PCSrc2=Br Taken
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
1
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
bcond
ALU
ALU result
0
M
u
x
1
0
Address
Read data
Data memory
Write data
1
M
u
x
0
0
funct
ALU operation
0

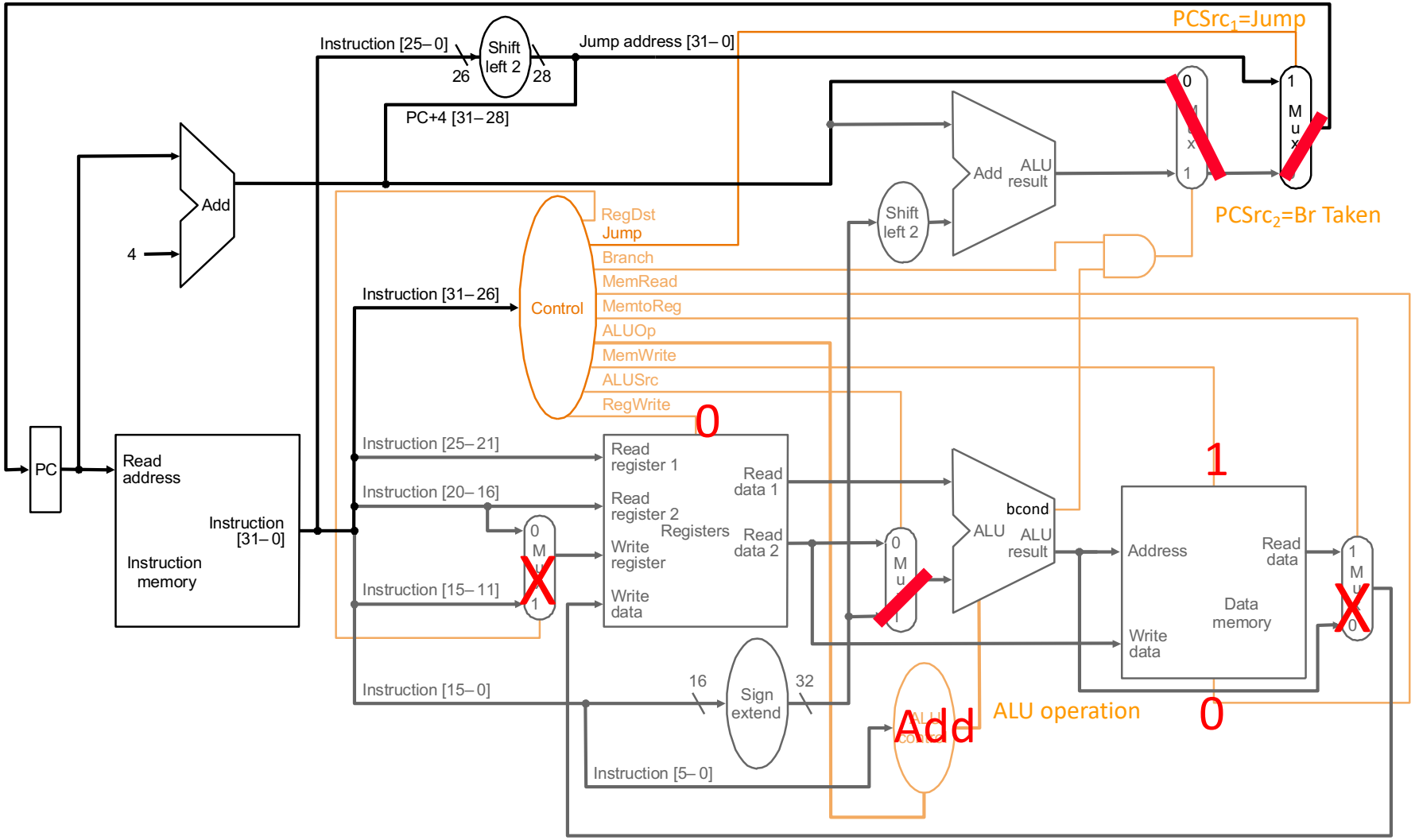
I-Type ALU


PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Shift left 2
Add
ALU result
PCSrc2=Br Taken
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
1
0
ALU
bcond
ALU result
Address
Read data
Data memory
Write data
16
Sign extend
32
opcode
ALU operation
0
0

lw

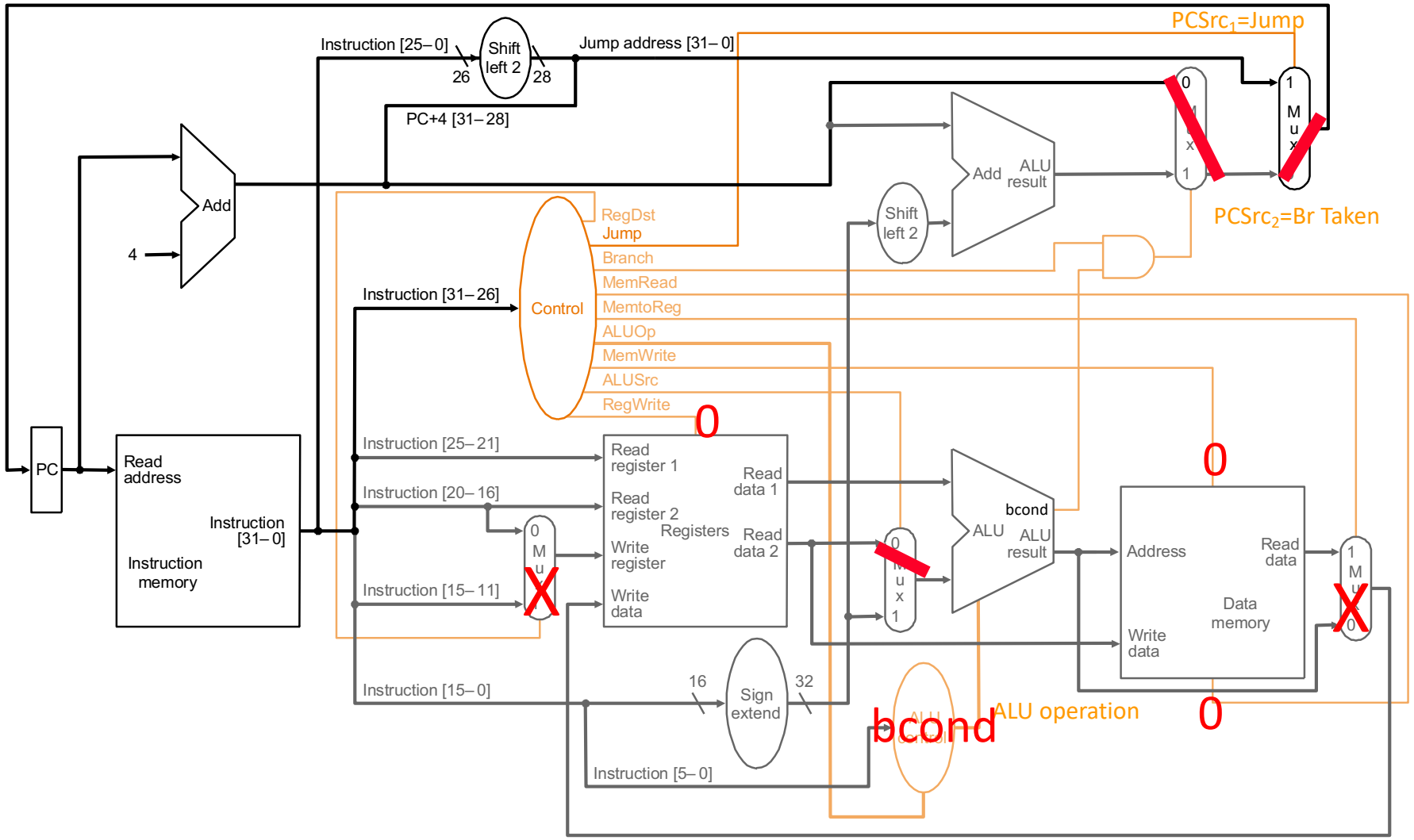

PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
M u x
0
1
Add
ALU result
Shift left 2
PCSrc2=Br Taken
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
1
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
bcond
ALU
ALU result
Add
ALU operation
0
1
Address
Read data
Data memory
Write data

SW


PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
M u x
0
1
Add
ALU result
Shift left 2
PCSrc2=Br Taken
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
0
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
ALU
bcond
ALU result
Add
ALU operation
1
Address
Read data
Data memory
Write data
0

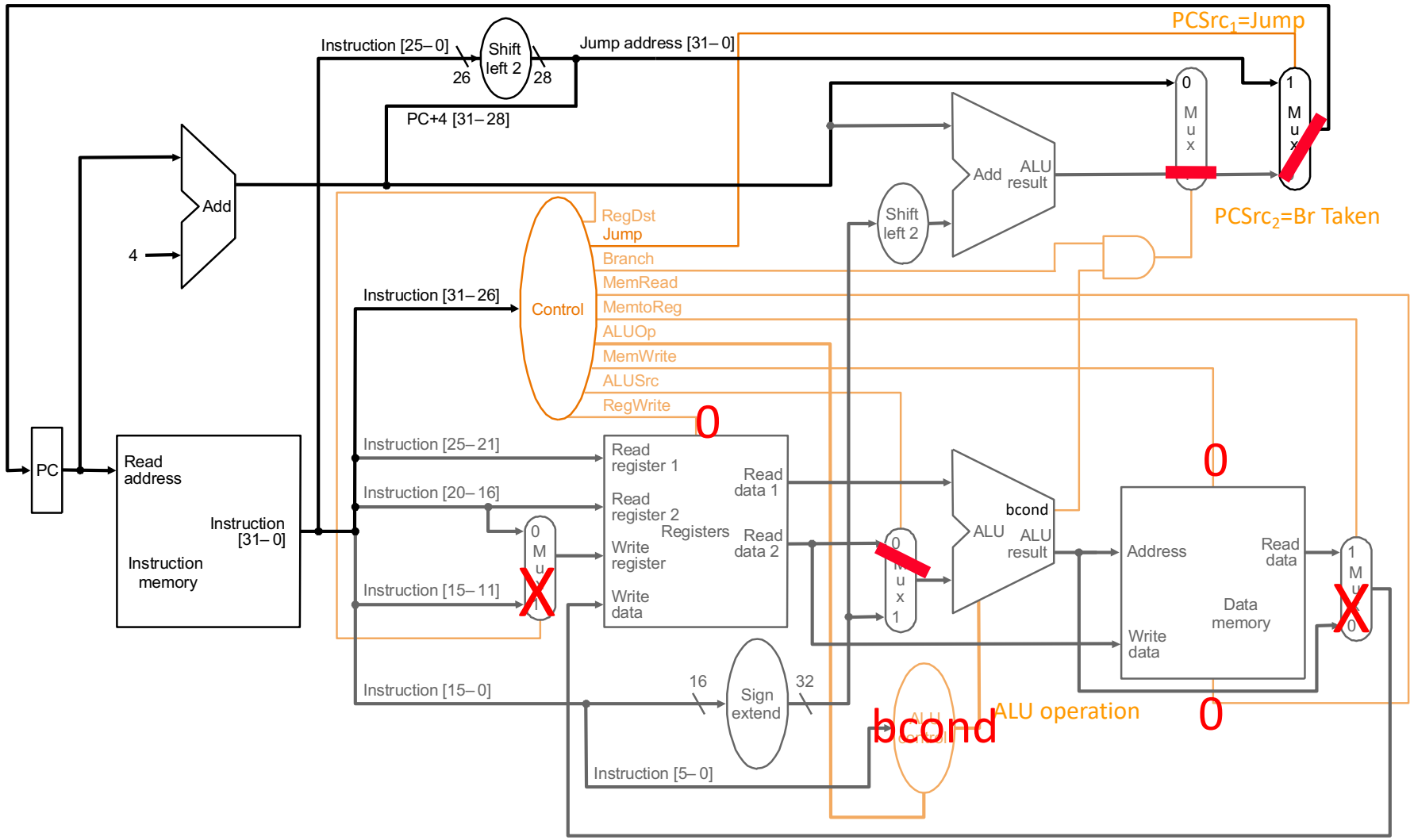
Branch (Not Taken)

Some control signals are dependent on the processing of data

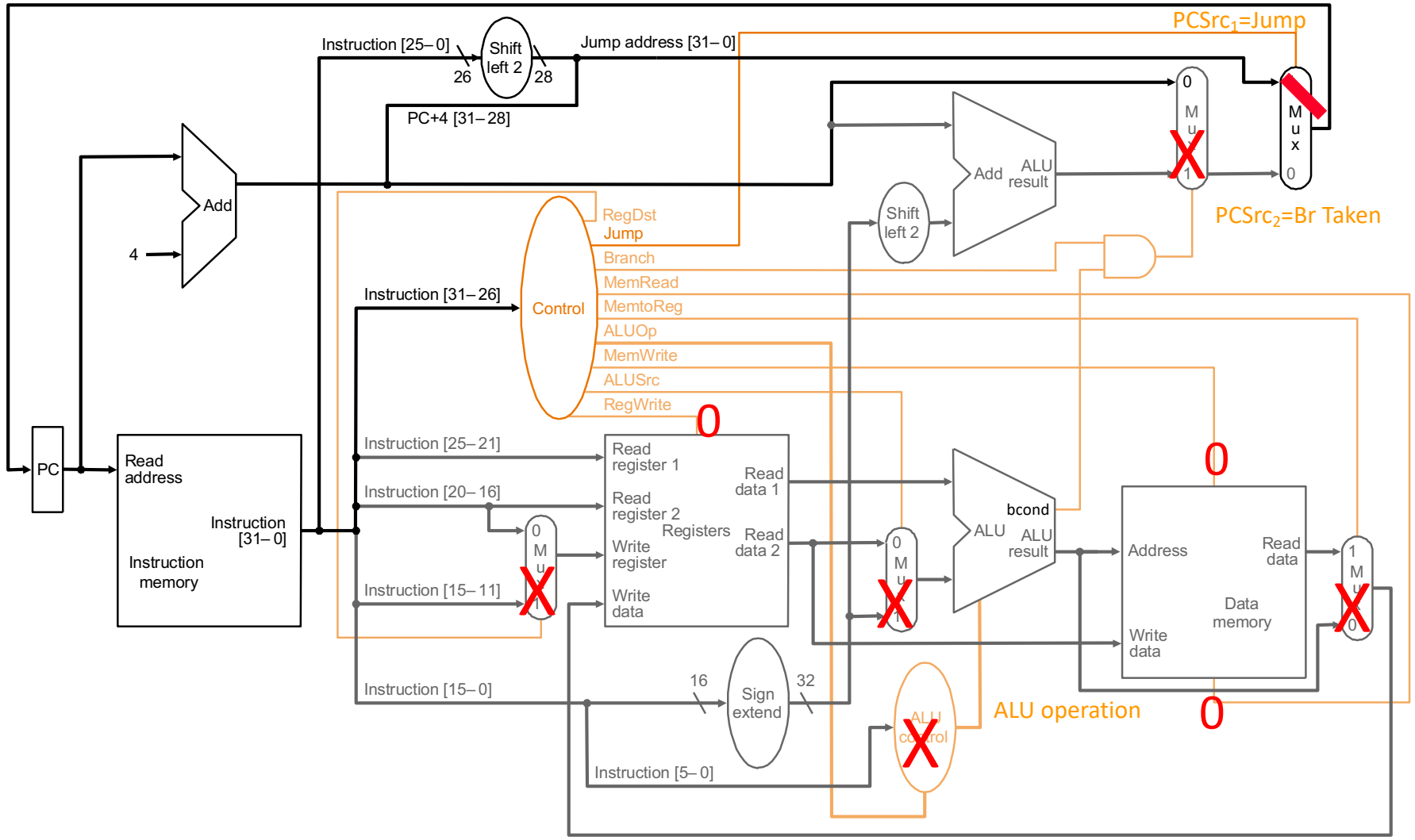


Branch (Taken)

Some control signals are dependent on the processing of data



Jump


PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
0
M
u
x
1
Mux
0
Add
ALU result
Shift left 2
PCSrc2=Br Taken
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
ALU
bcond
ALU result
ALU control
ALU operation
Address
Read data
Data memory
Write data
0
0
0

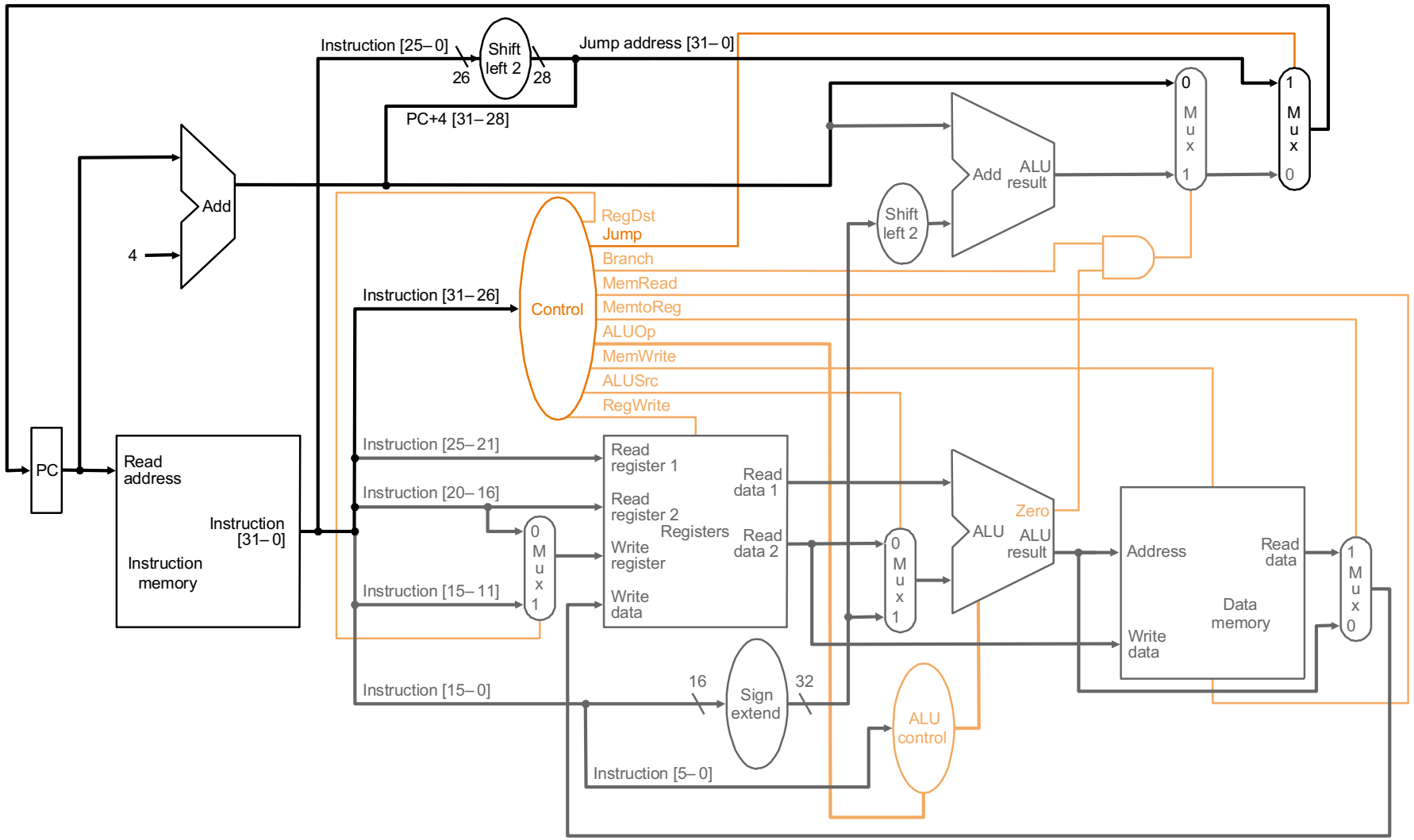
What is in That Control Box?

- Combinational Logic → Hardwired Control
  - Idea: Control signals generated combinationally based on bits in instruction encoding

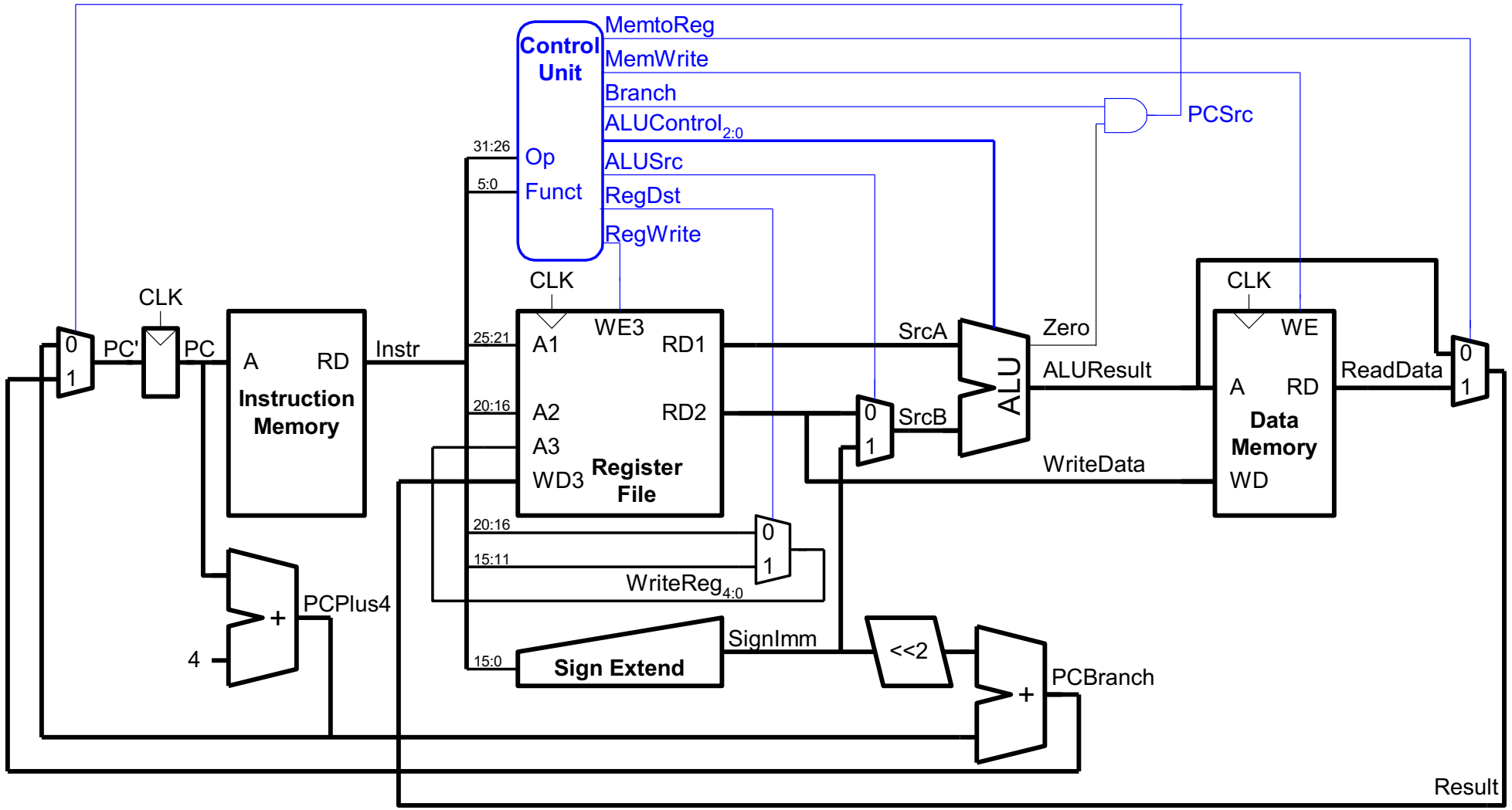
- Sequential Logic → Sequential Control
  - Idea: A memory structure contains the control signals associated with an instruction
    - Called Control Store

- Both types of control structure can be used in single-cycle processors
  - Choice depends on the latency of each structure + how much the critical path control signal generation is, etc.

Review: Complete Single-Cycle Processor



Another Complete Single-Cycle Processor



KING'S College LONDON

Hardware Design

Lecture 4: Single-Cycle Microarchitecture: build the whole CPU once

Dr. Haiyu Mao

19.02.2026