Microarchitecture (Fundamentals & Design)

Microarchitecture: Implementation of the ISA under specific design constrains and goals

How Does a Machine Process Instructions?

- What does processing an instruction mean?
- We will assume the von Neumann model (for now)

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

Process instruction

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

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

The "Process Instruction" Step

- ISA specifies abstractly what AS' should be, given an instruction and AS
  - 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 AS and AS' during instruction execution
    - One state transition per instruction
- Microarchitecture implements how AS is transformed to AS'
  - 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: AS → AS' (transform AS to AS' in a single clock cycle)
    - Choice 2: AS → AS+MS1 → AS+MS2 → AS+MS3 → AS' (take multiple clock cycles to transform AS to AS')

microactual State.

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*

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

Process instruction in **one clock cycle**

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

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 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

All six phases of the instruction processing circle take a single machine clock circle to complete

All six phases of the instruction processing circle can take multiple machine clock circles to complete

In fact, each phase can take multiple clock circles to complete.

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

- Single-cycle machine:
  - Control signals are generated in the same clock cycle as the one during which data signals are operated on
  - Everything related to an instruction happens in one clock cycle (serialized processing)
- Multi-cycle machine:
  - Control signals needed in the next cycle can be generated in the current cycle
  - Latency of control processing can be overlapped with latency of datapath operation (more parallelism)

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 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, 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 datapath and 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

Above are the fundamentals of microarchitecture

Next we will step into the design.

Performance Analysis:

Execution time of a single instruction:

{CPI} × {clock circle time}

↓

cycles per instruction

Excution time of an entire program

Sum over all instructions [{CPI} × {clock cycle time}]

Also: {# of instructions} × {Average CPI} × {clock circle time}

↓

number

Single-cycle uarch performance

CPI = 1

clock cycle time = longggg

Multi-cycle uarch performance

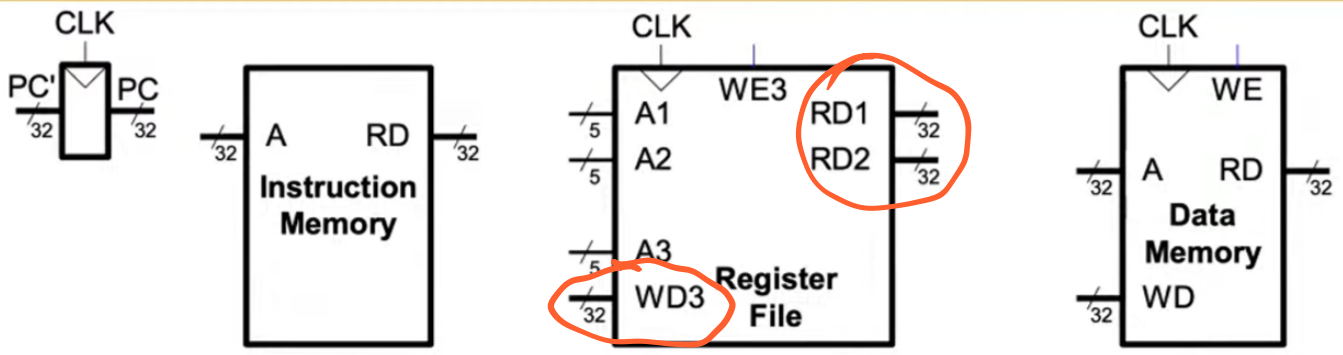
CPI = different for each instruction

clock cycle time = short

Average CPI → hopefully small

we have 2 degrees of freedom to optimize independently

MIPS State Elements

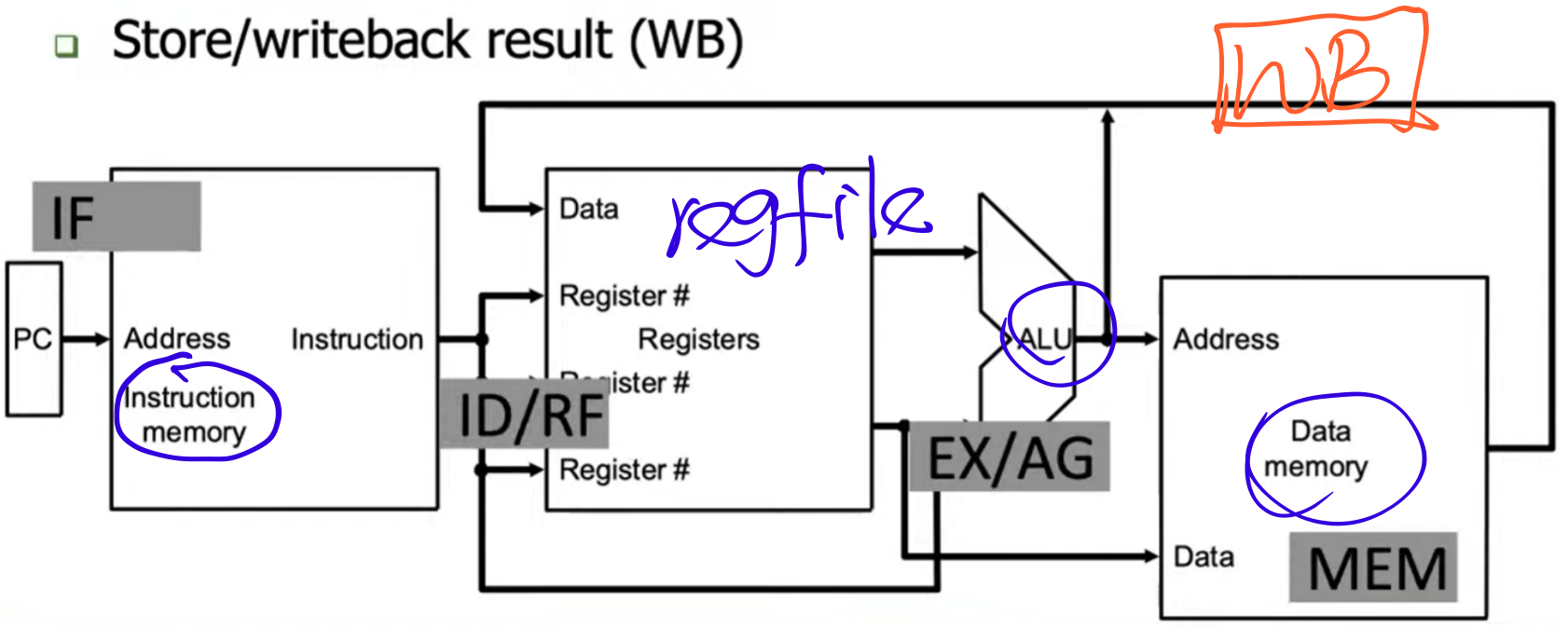


- **Program counter:**
  32-bit register
- **Instruction memory:**
  Takes input 32-bit address A and reads the 32-bit data (i.e., instruction) from that address to the read data output RD
- **Register file:**
  The 32-element, 32-bit register file has 2 read ports and 1 write port
- **Data memory:**
  If the write enable, WE, is 1, it writes 32-bit data WD into 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 memory (and reg file)
Combinational read
Output of the read data port is a combinational function of the memory contents and the corresponding input address.
Synchronous write
target location is updated at the positive edge clock transition when write enable is asserted.
Single-cycle, synchronous memory
Contrast this with memory that tells when the data is ready (i.e. Read singal: indicating the r/w is done)

Instruction Processing

- 5 generic steps (P&H book)
  - 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)



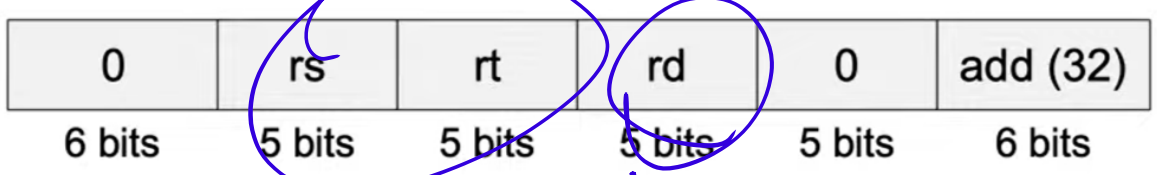
R-Type ALU Instructions

- 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

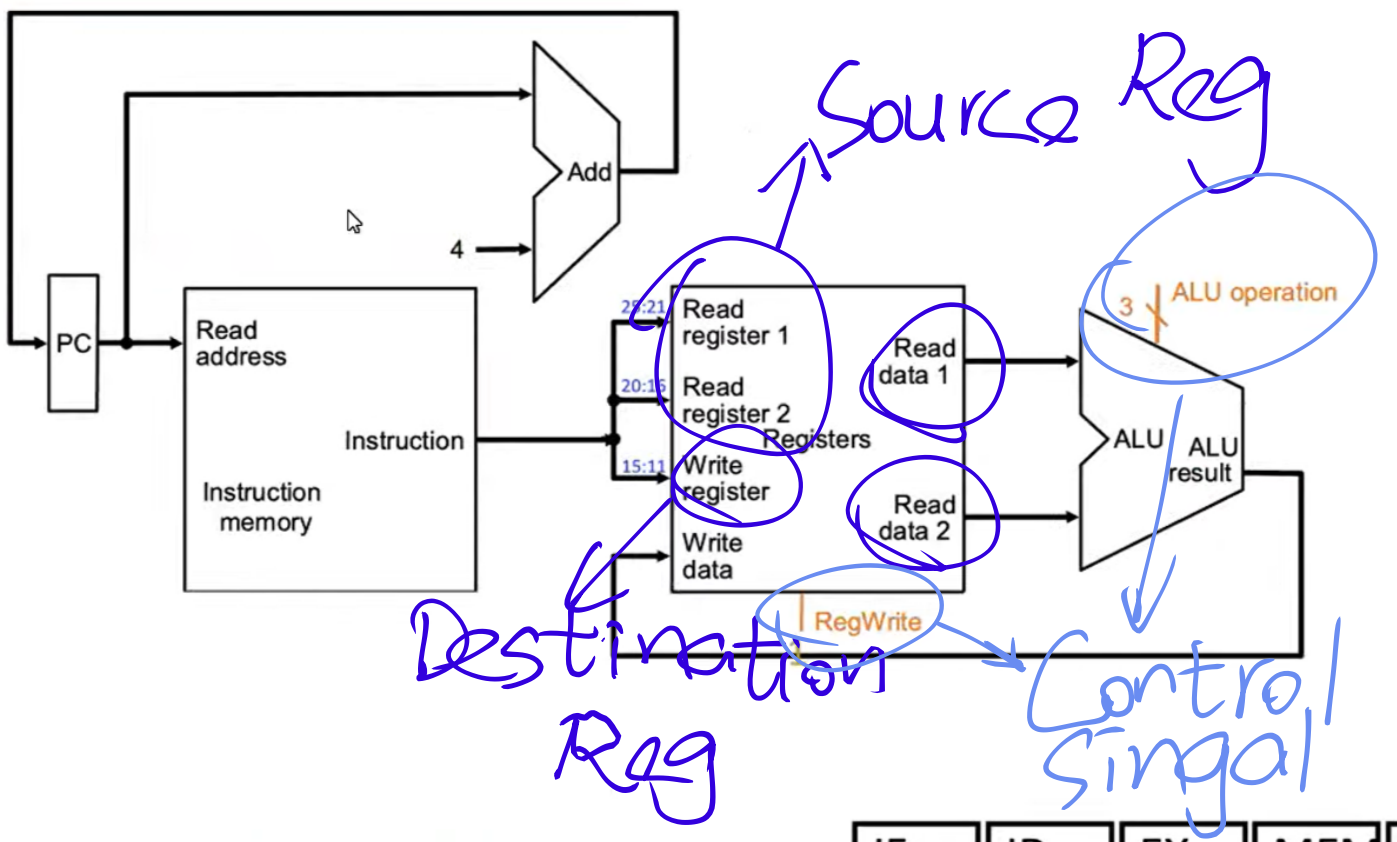
Source

destination

- Semantics

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

(R-Type) ALU Datapath



IF | ID | EX | MEM | WB

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

→ Combinational state update logic

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
```

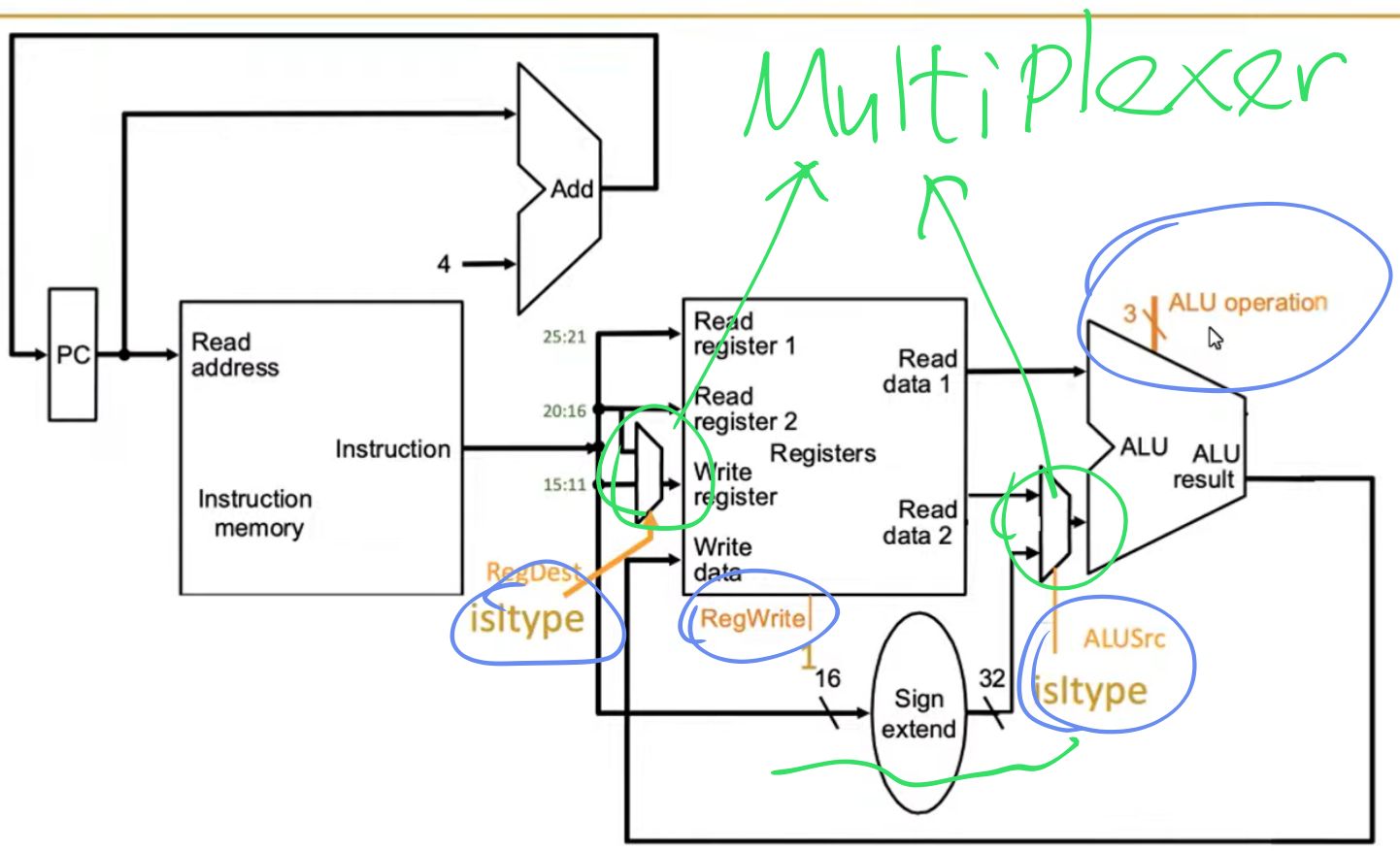
Machine Encoding

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
```

IF	ID	EX	MEM	WB

Combinational state update logic

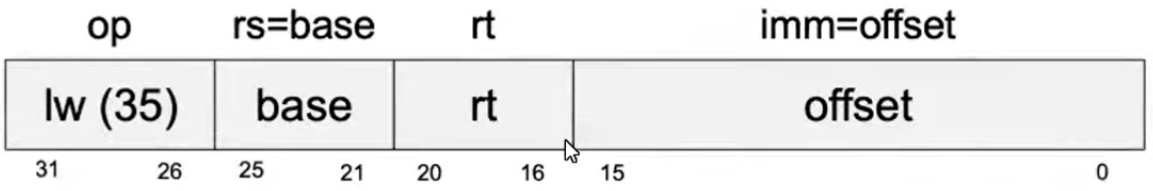
Load Instructions

- Load 4-byte word

MIPS assembly

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

Machine Encoding



I-Type

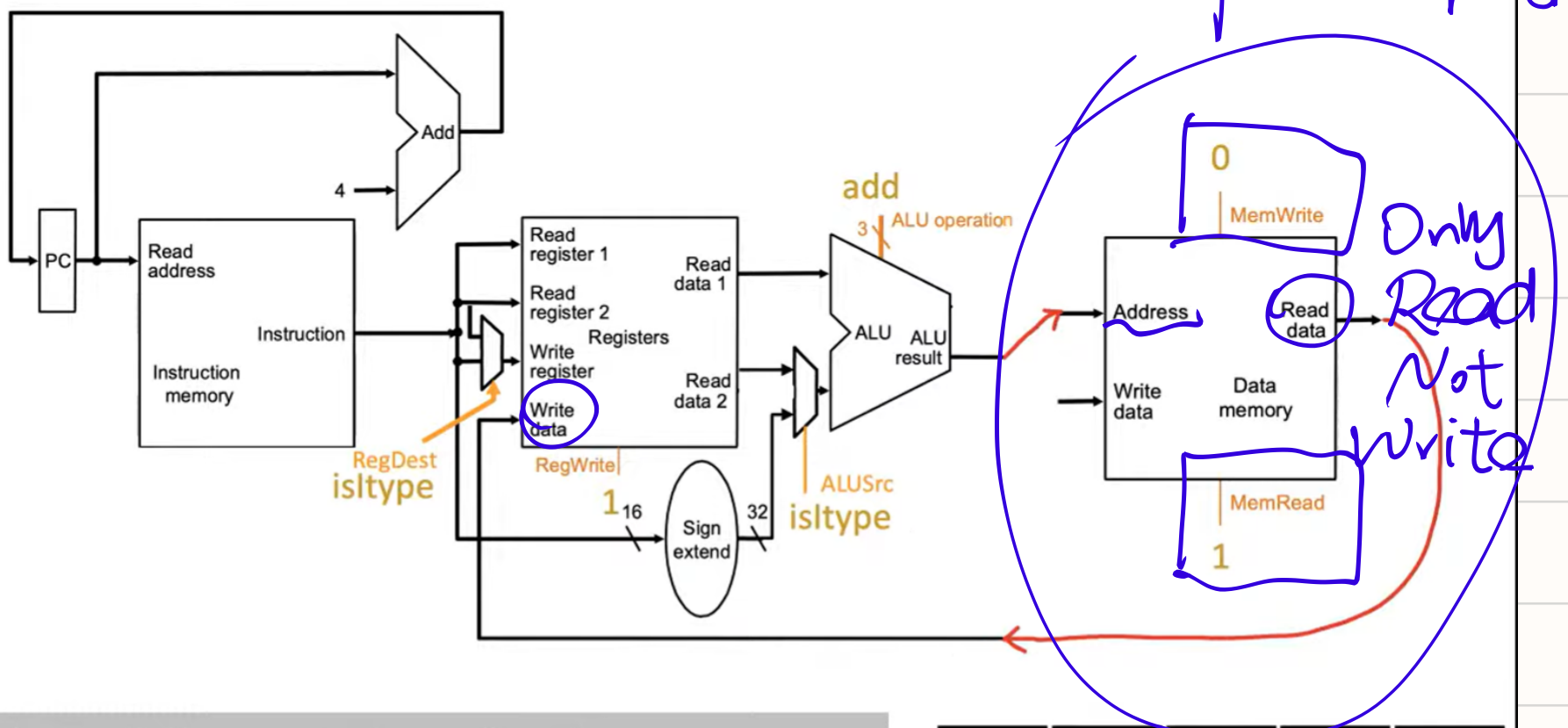
- Semantics

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

General Purpose Register.

Add this part based on I-type Datapath

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
```

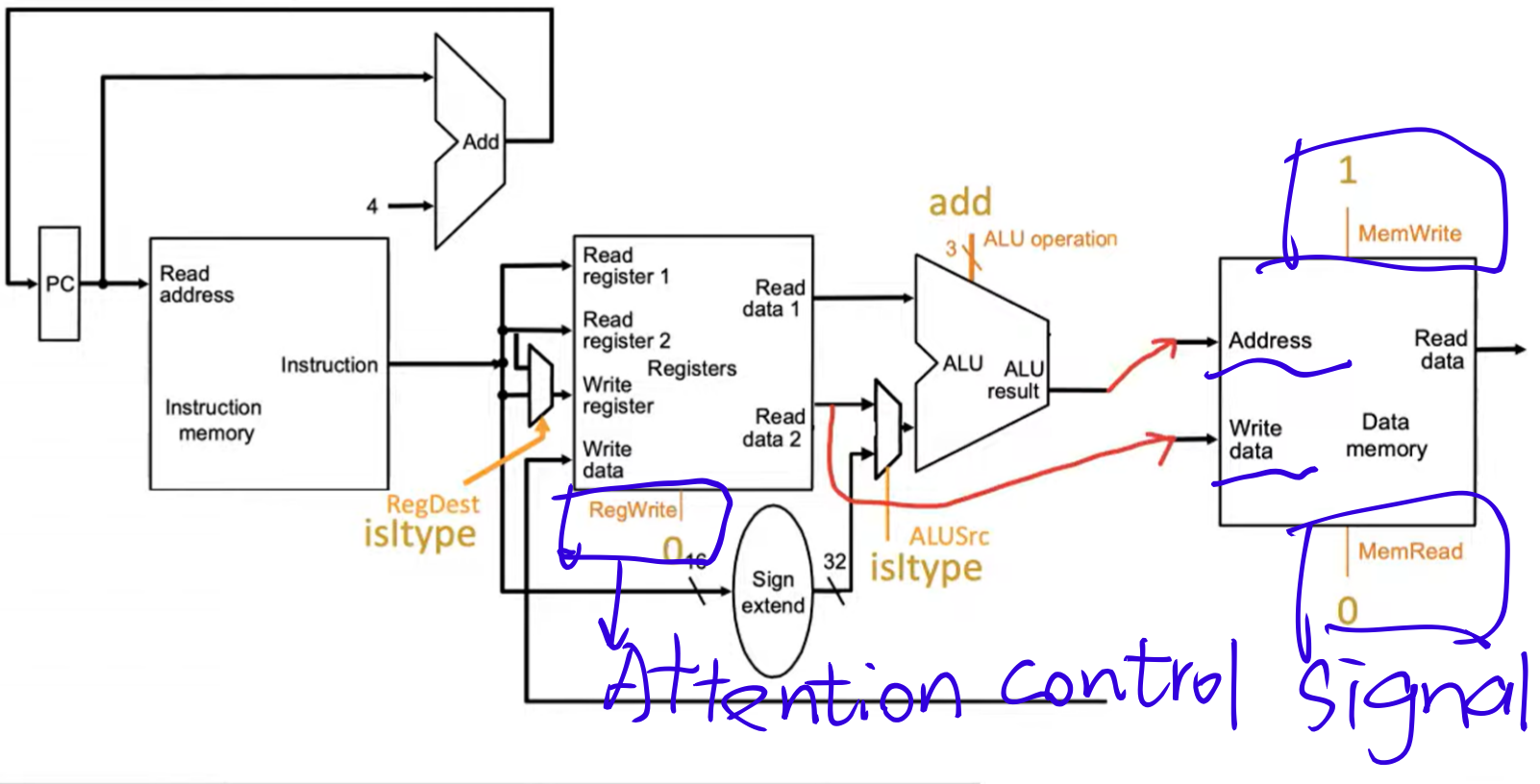
Machine Encoding

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



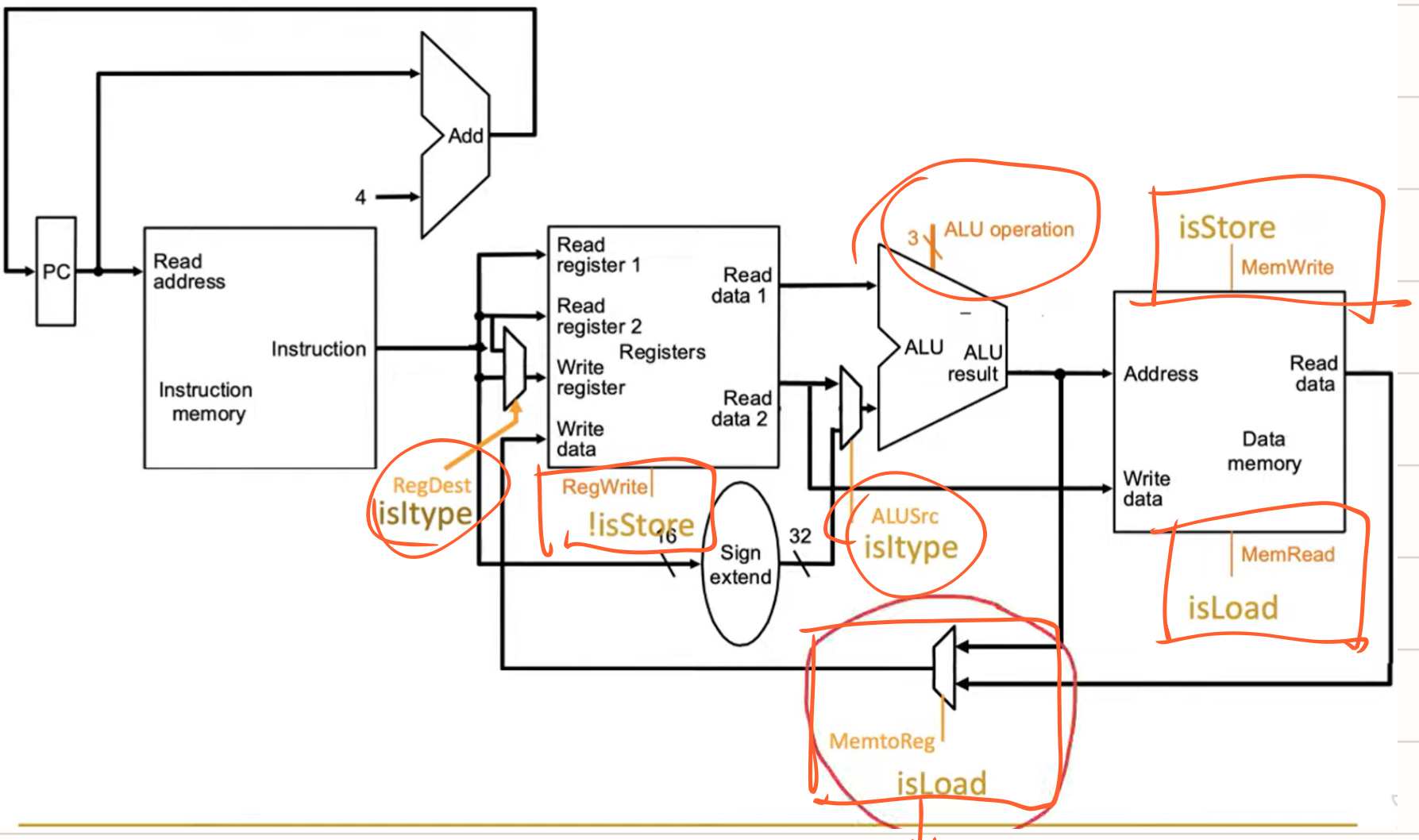
if MEM[PC]==SW rt offset_{16} (base)
address = sign-extend(offset) + GPR[base]
MEM[address] ← GPR[rt]
PC ← PC + 4

IF	ID	EX	MEM	WB

Combinational
state update logic

We combine all the above together!

Datapath for Non-Control-Flow Insts.



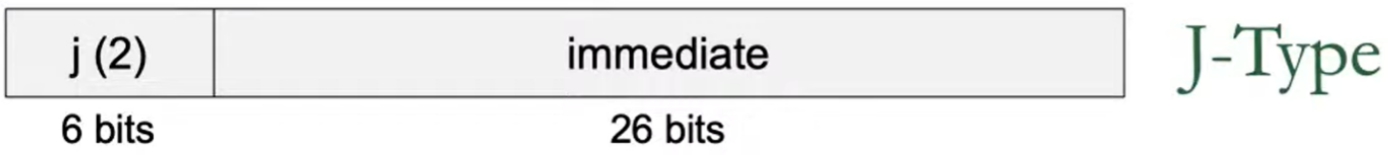
The data written back can come from either the "ALU result" or "Read data"

Control Flow Instruction

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 $^{\uparrow}$[31:28], immediate$_{26}$, 2' b00 }
  PC ← target

Computational logic

Bit Slicing!

Similar

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 $^{\uparrow}$[31:28], immediate$_{26}$, 2' b00 }
    PC ← target

Store ahead in order to return in future

Similar

- 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
```

beq (4)	rs	rt	immediate=offset
6 bits	5 bits	5 bits	16 bits

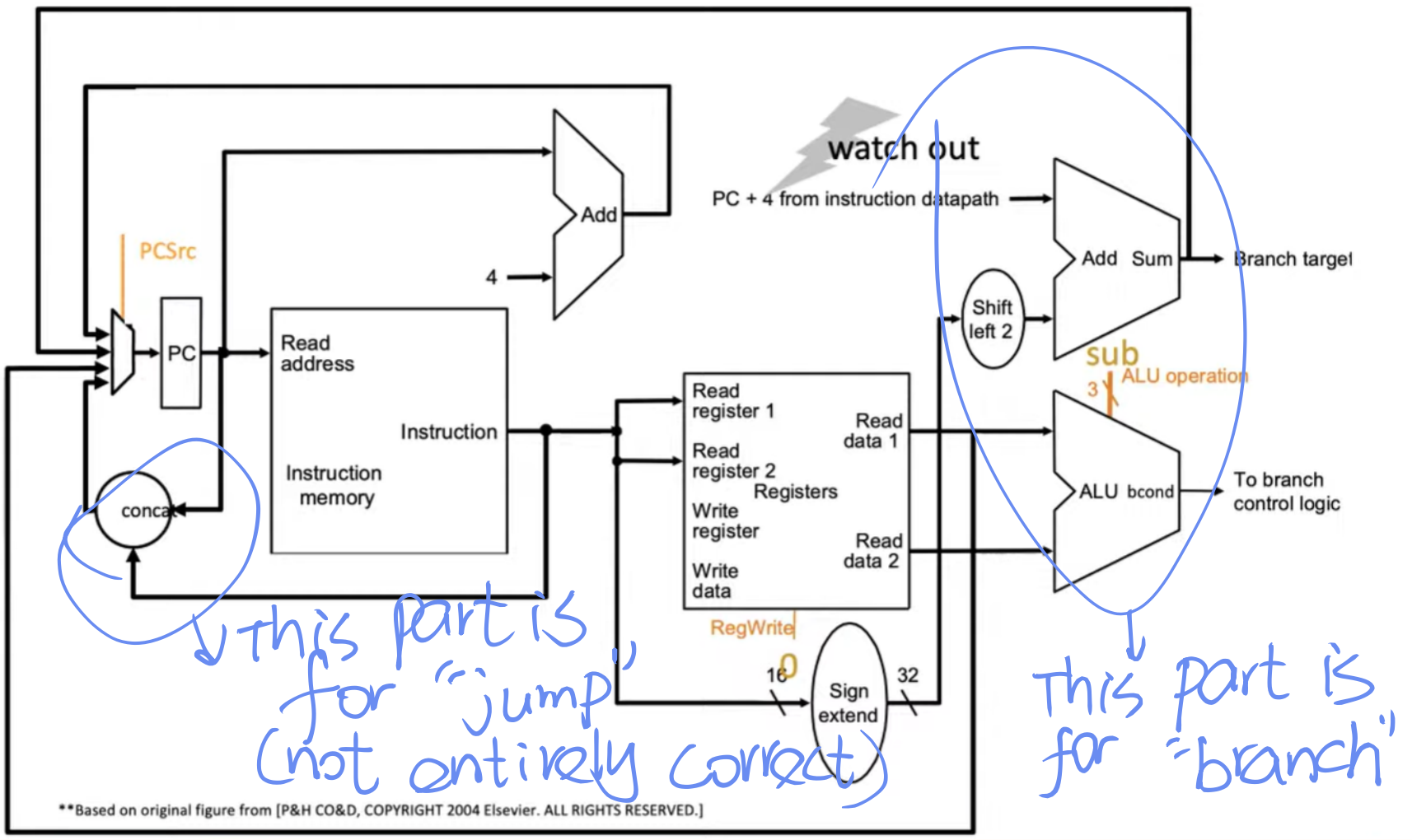
I-Type

- Semantics (assuming no branch delay slot)

  if MEM[PC] == beq rs rt immediate$_{16}$
  target = PC† + sign-extend(immediate) x 4 → left shifted 2 bits
  if GPR[rs]==GPR[rt] then PC ← target
  else PC ← PC + 4

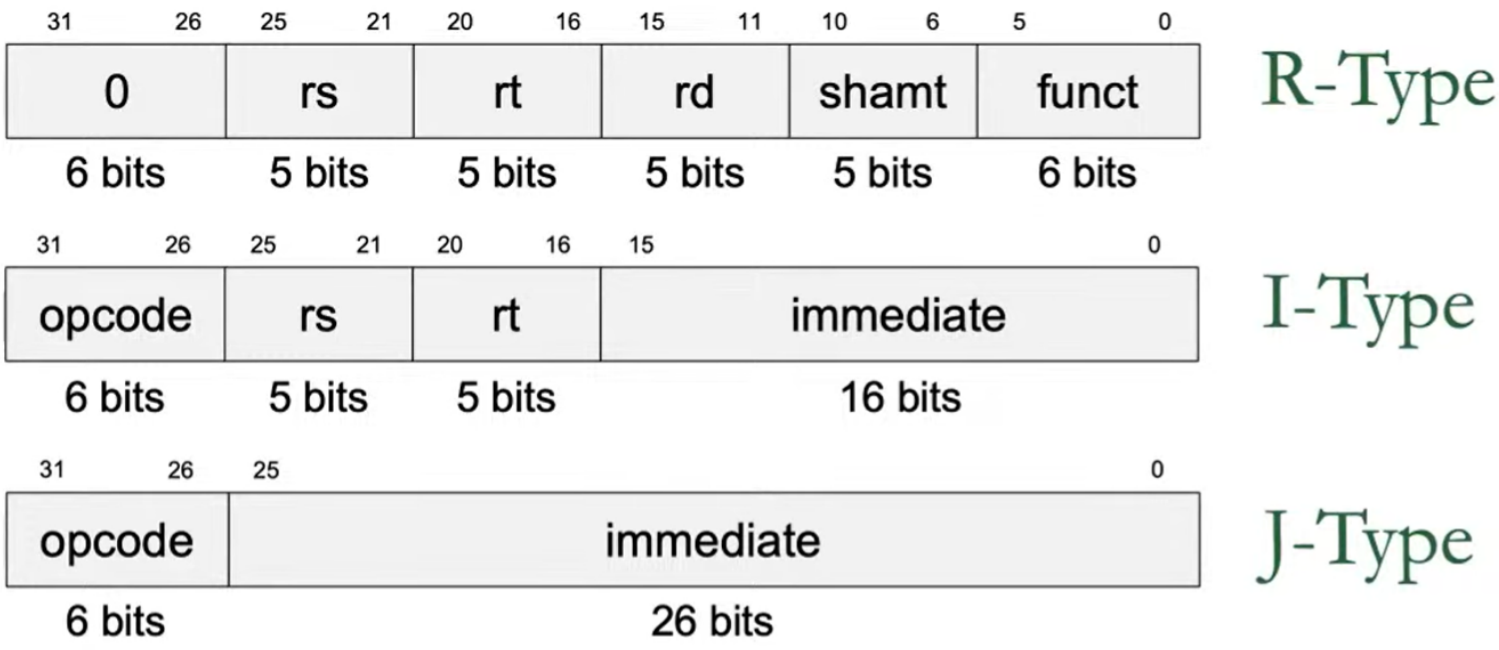
  - Variations: beq, bne, blez, bgtz

Conditional Branch Datapath (for you to finish)



Single-Cycle Hardwired Control

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



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

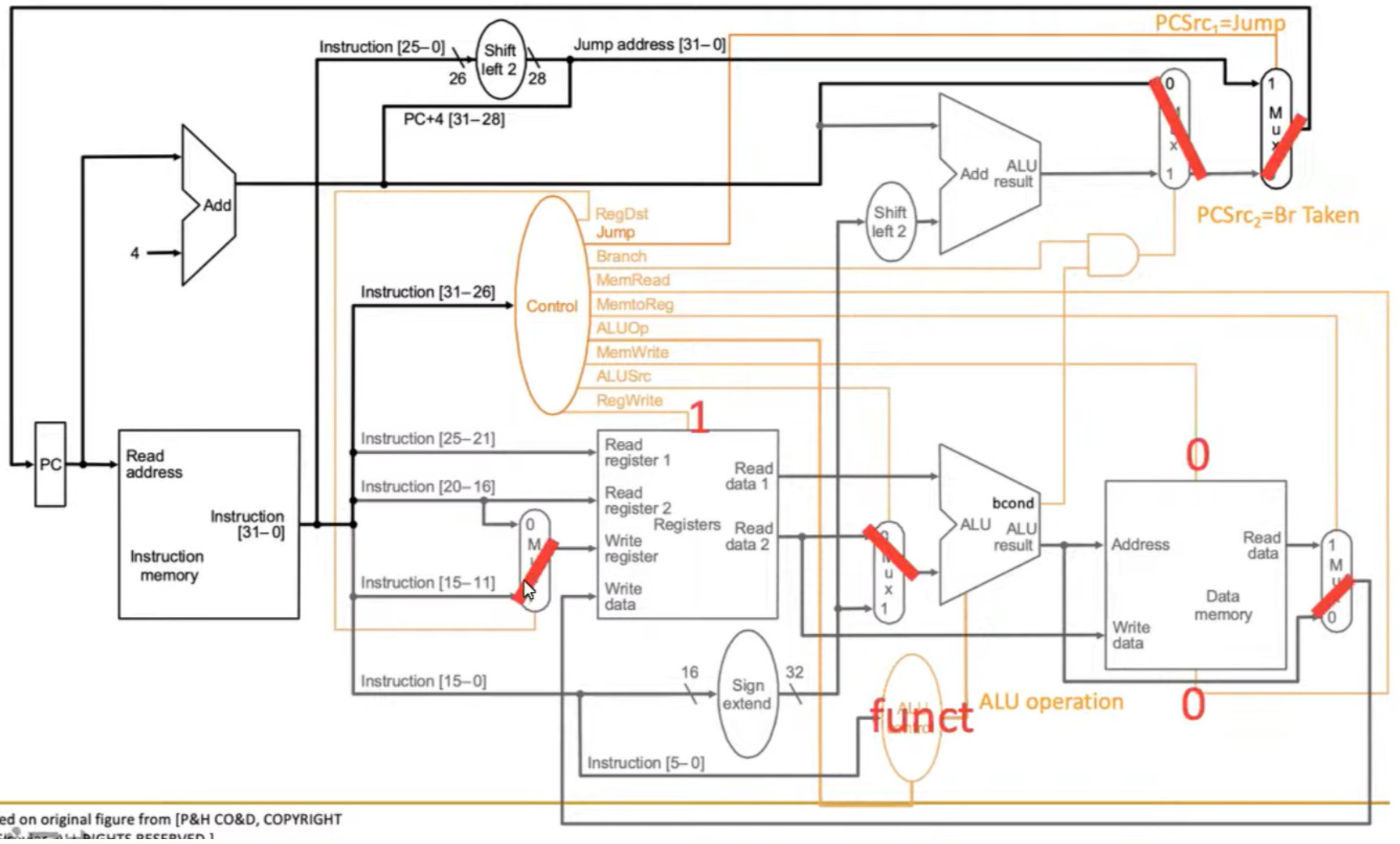
Single-Bit Control Signals (I)

	When De-asserted	When asserted	Equation
RegDest	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))

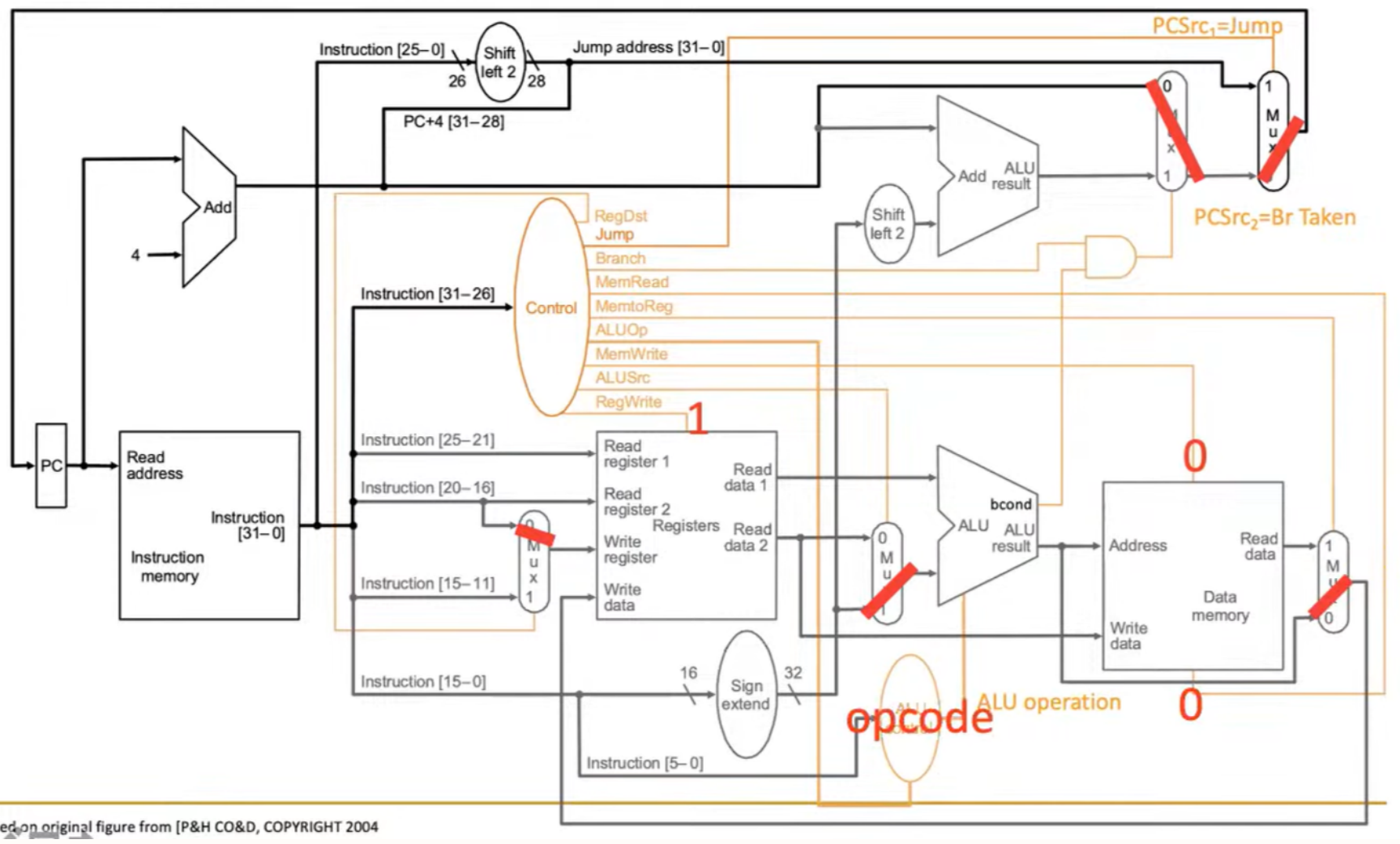
Single-Bit Control Signals (II)

	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"

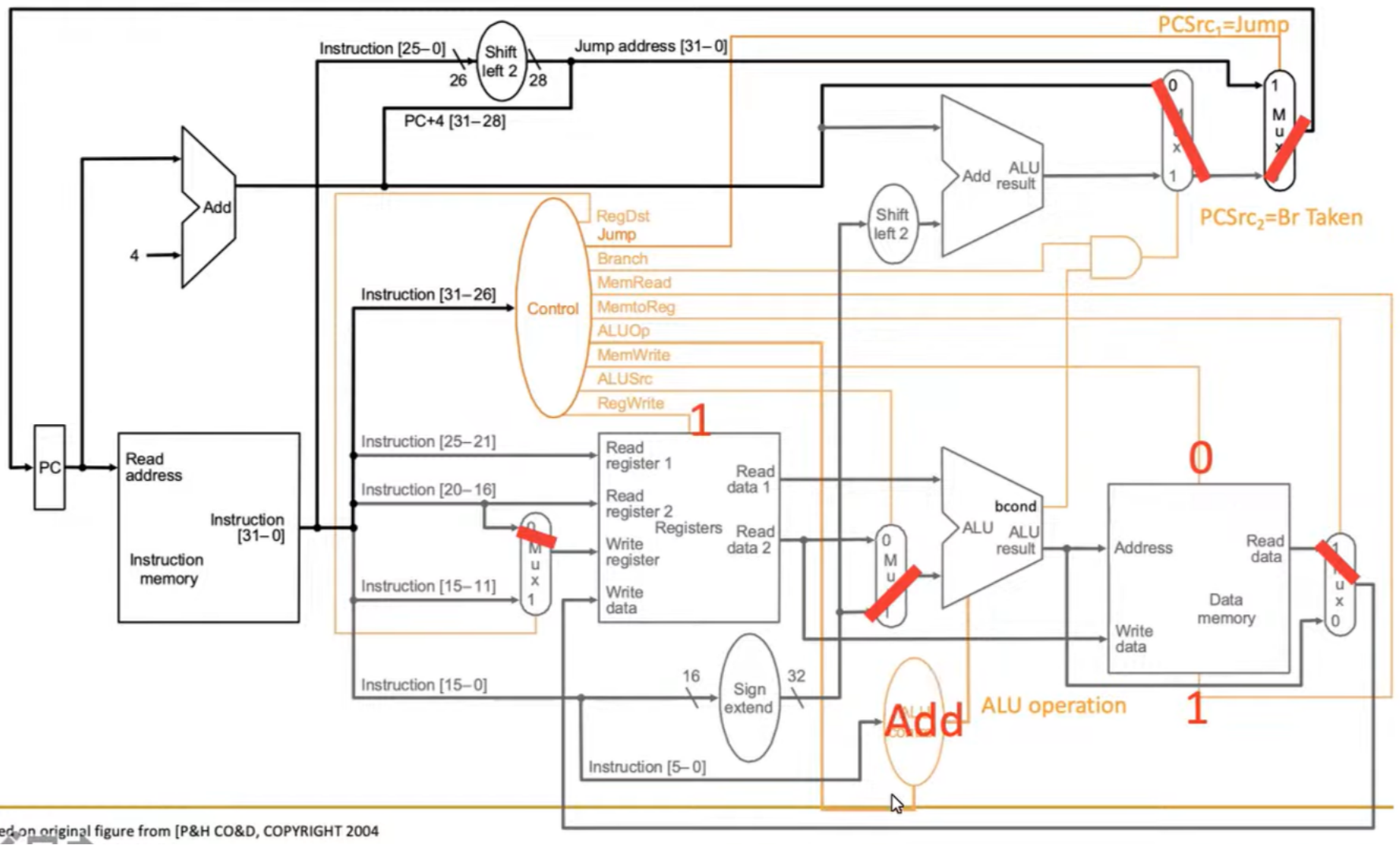
R-Type ALU



I-Type ALU

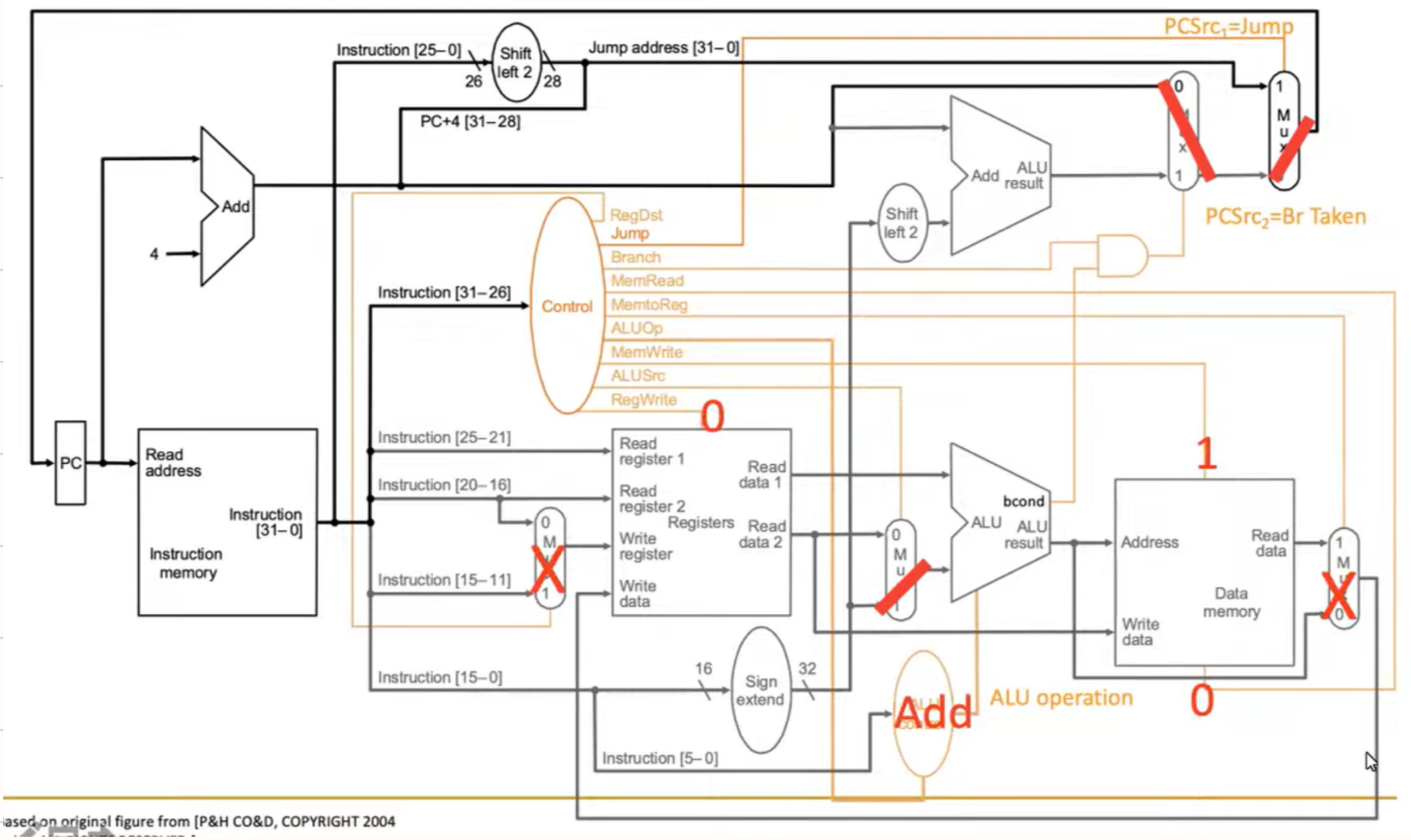


LW



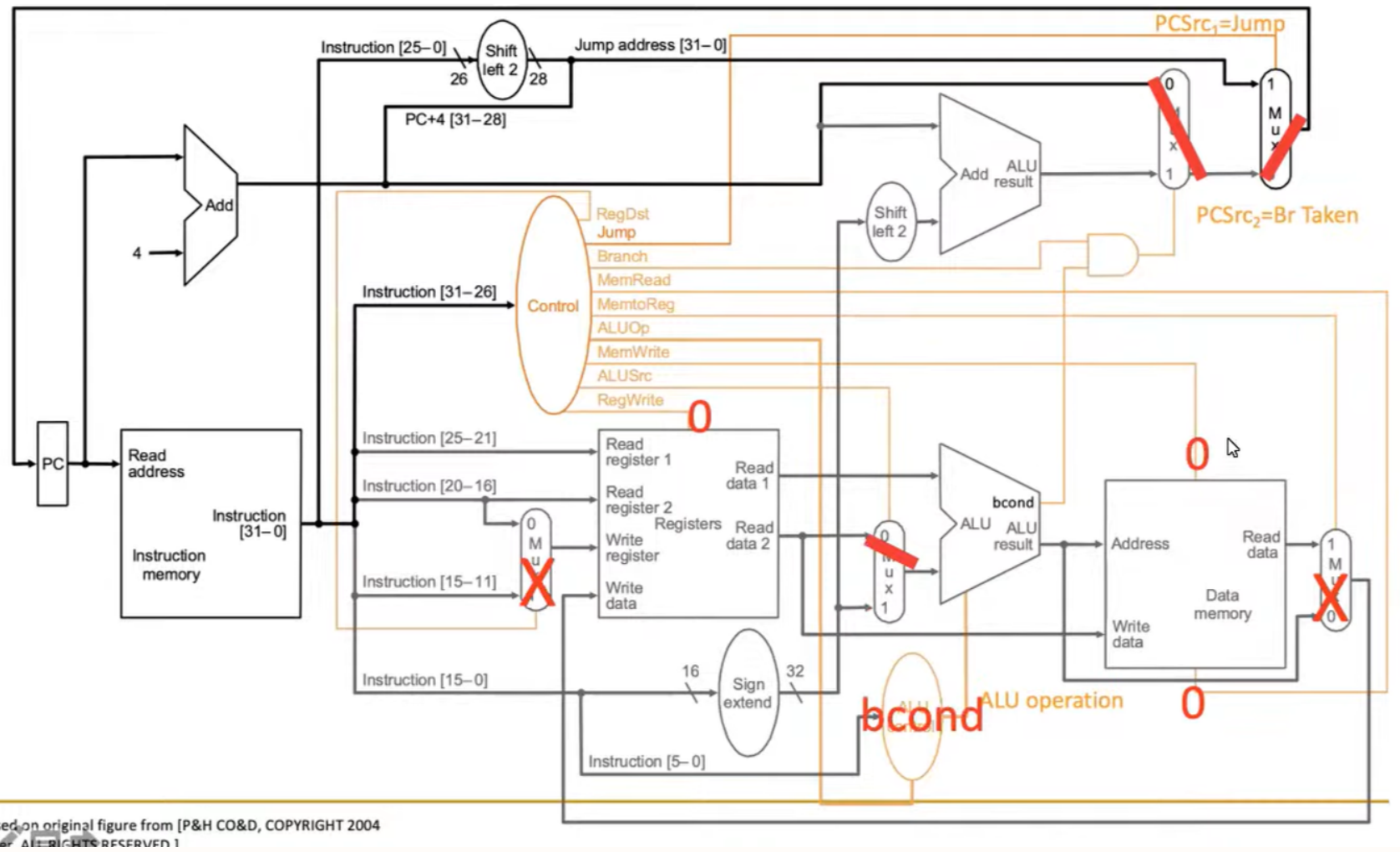
Based on original figure from [P&H CO&D, COPYRIGHT 2004

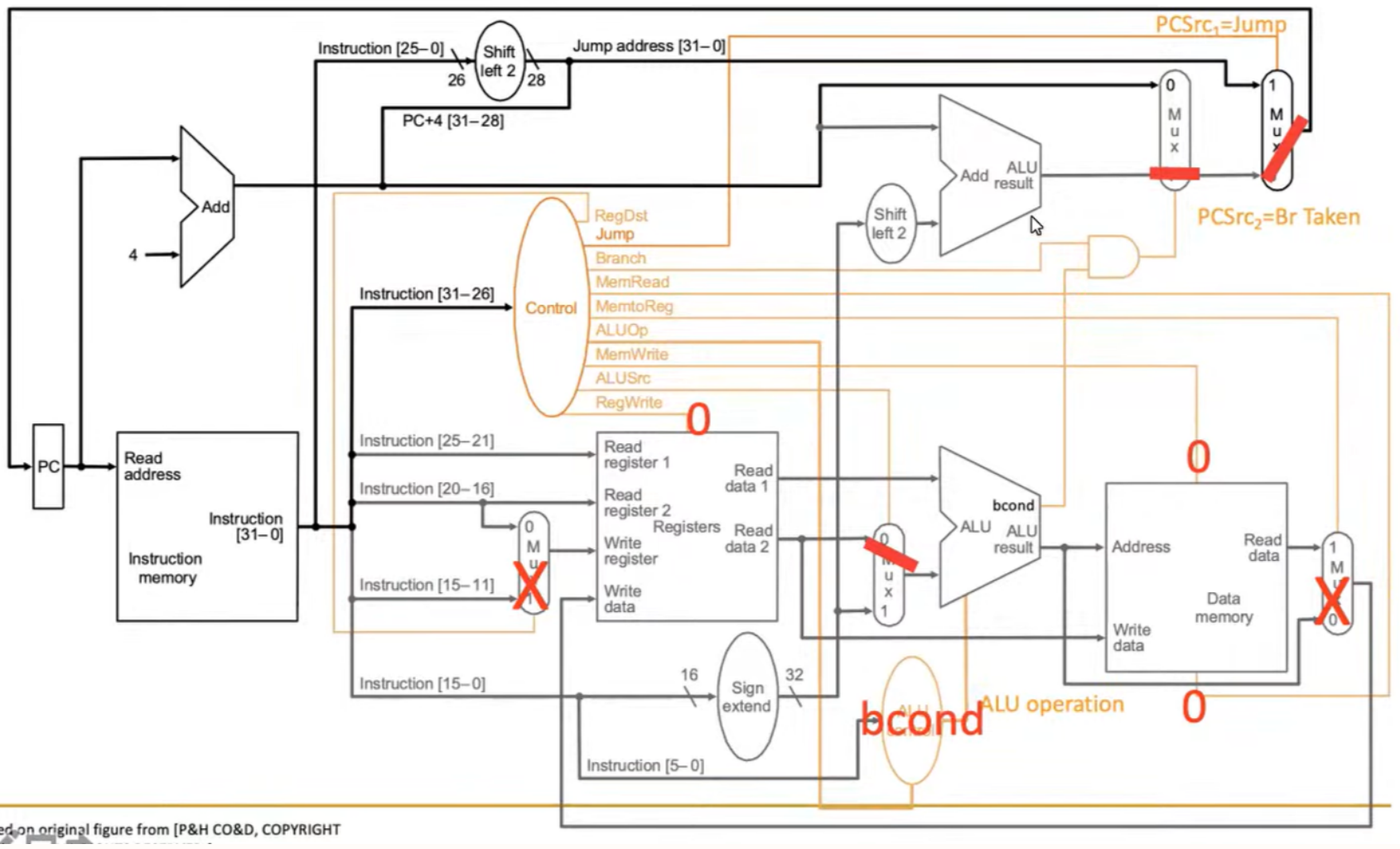
SW



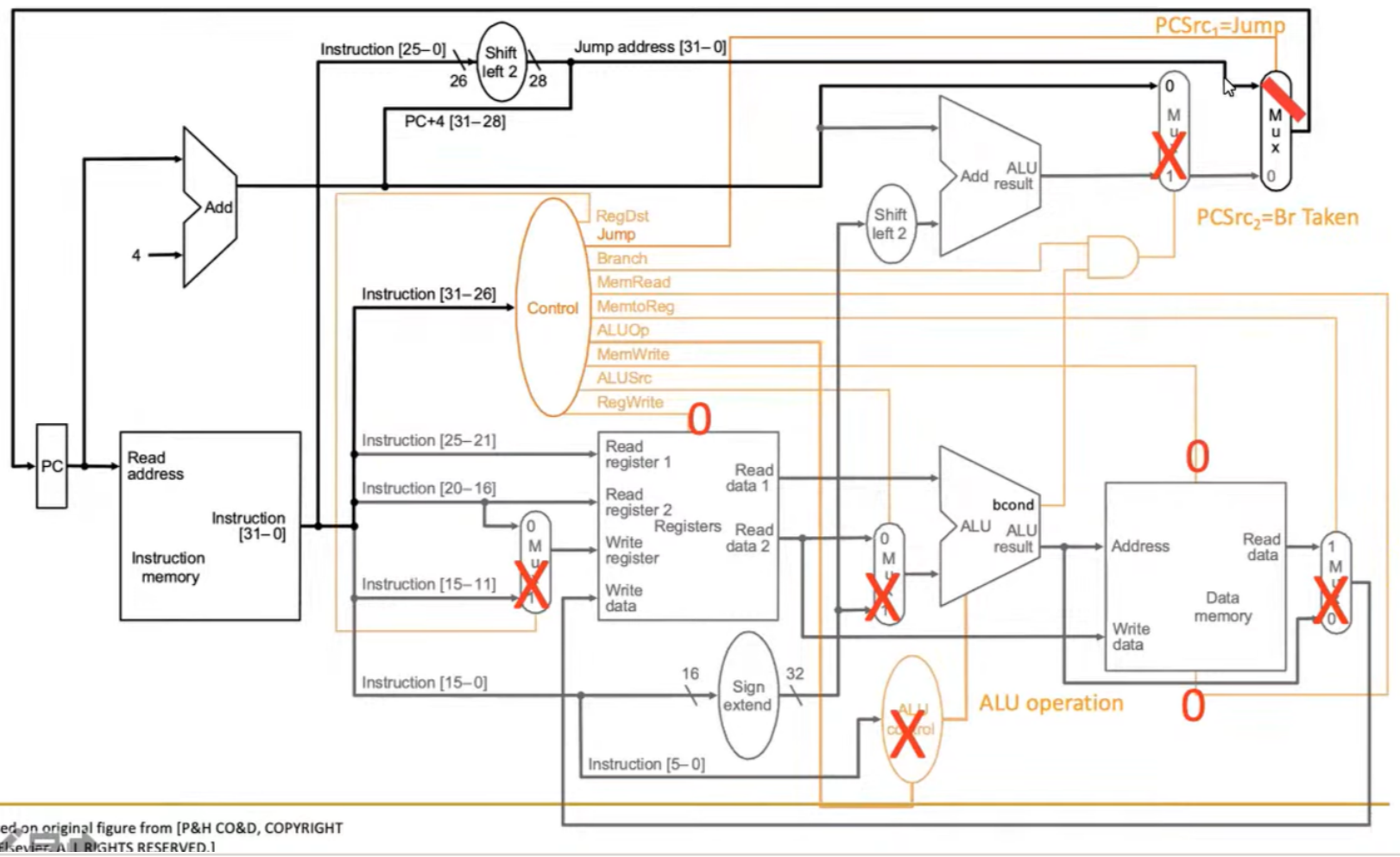
ased on original figure from [P&H CO&D, COPYRIGHT 2004

Branch (Not Taken)





Jump



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

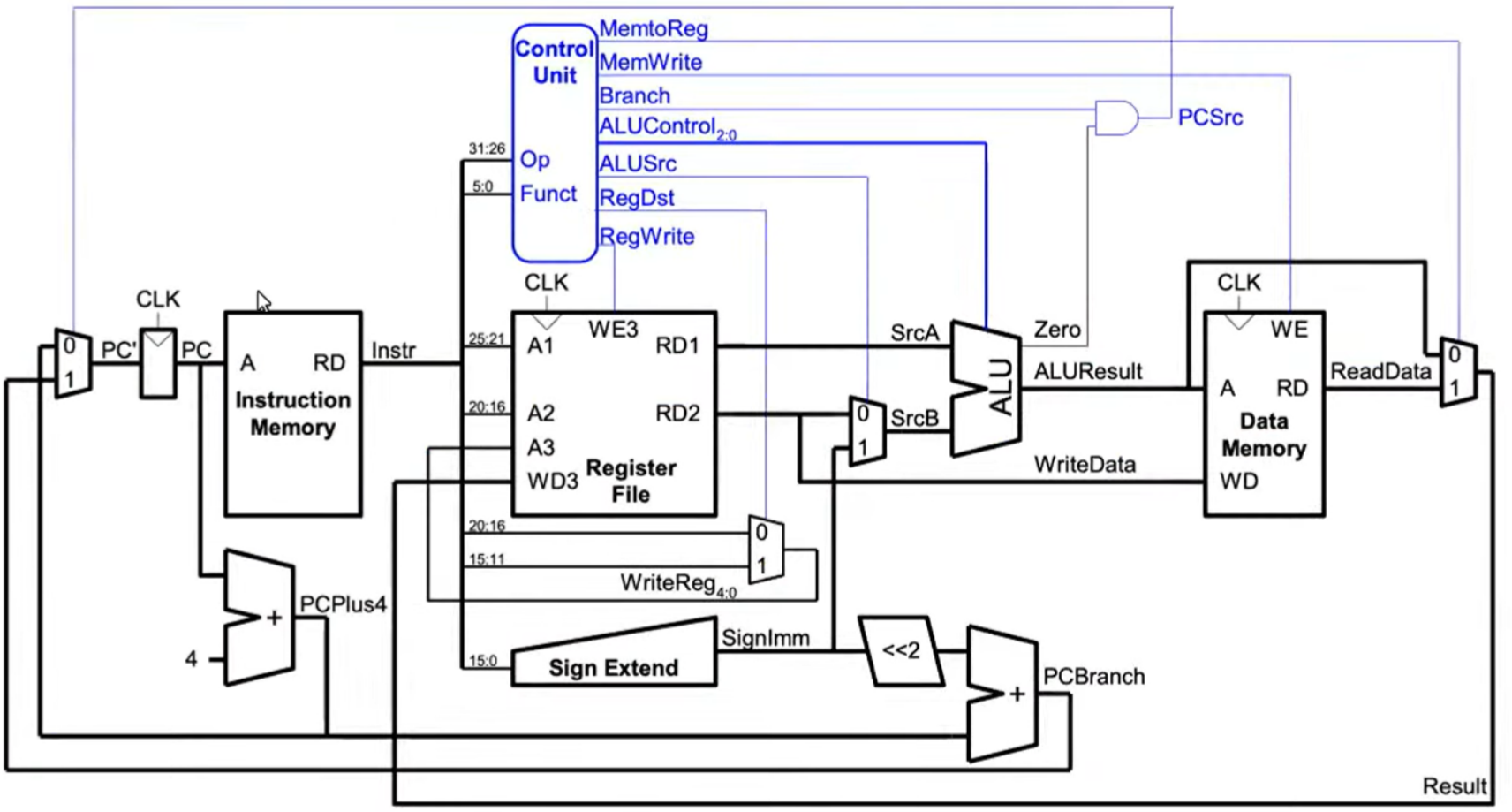
What is in That Control Box?

Combinational Logic → Hardwired control
Control signals generated combinationally based on bits in instruction encoding

Sequential Logic → Sequential control
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

Another Complete Single-Cycle Processor



Similarly, We Need to Design the Control Unit

- Control signals are generated by the decoder in control unit

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1