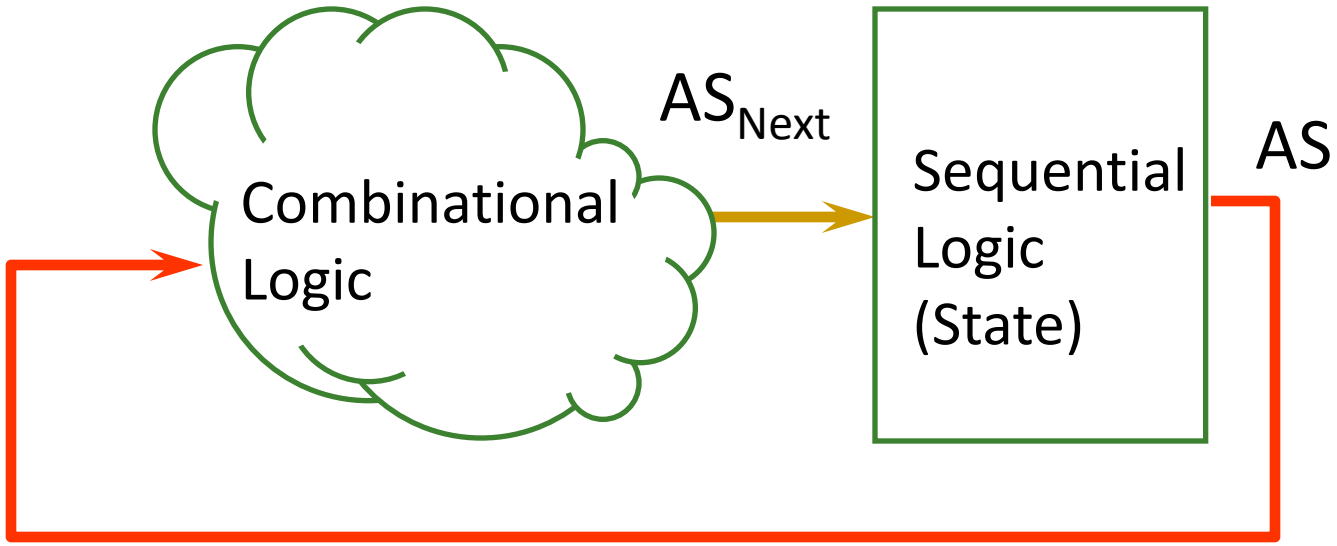
18-447
Computer Architecture
Lecture 5: Single-Cycle and Multi-Cycle Microarchitectures

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2014, 1/24/2014

A Single-Cycle Microarchitecture
A Closer Look

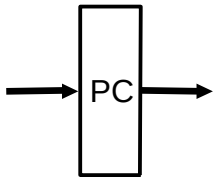
Remember…

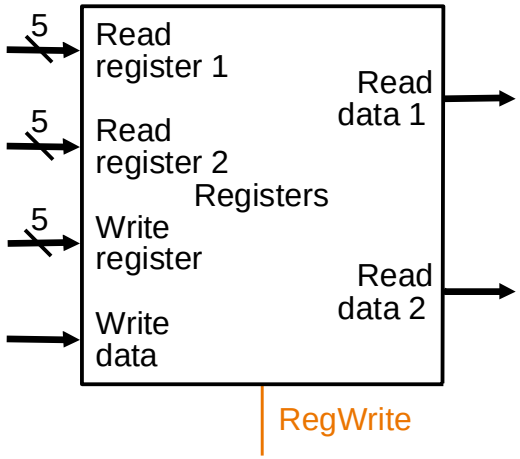
- Single-cycle machine

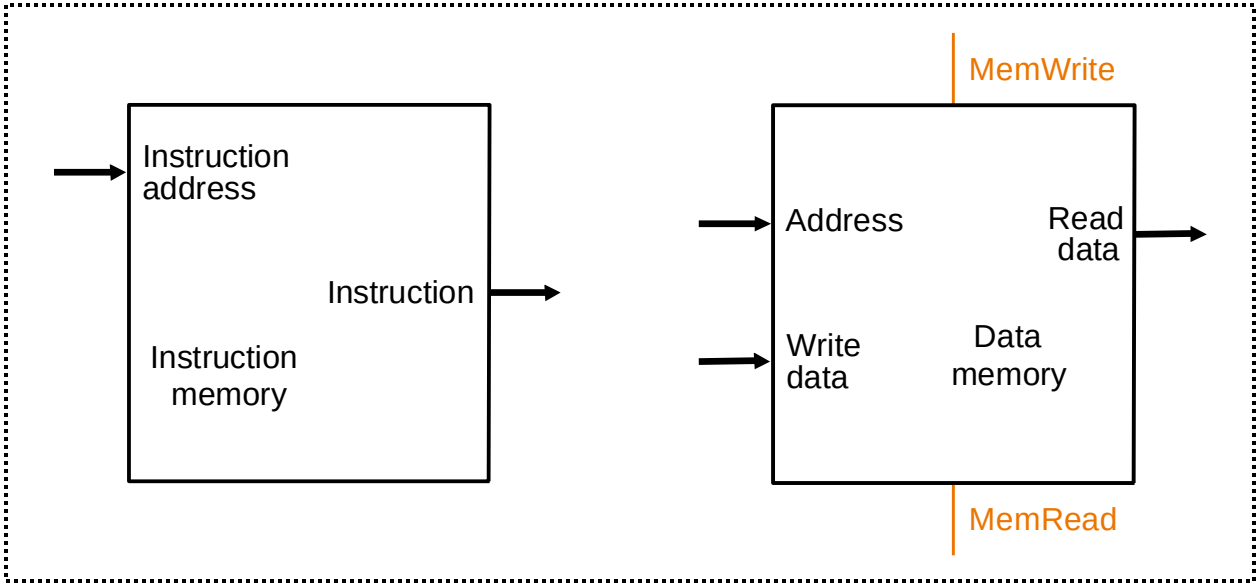


Let's Start with the State Elements

- Data and control inputs





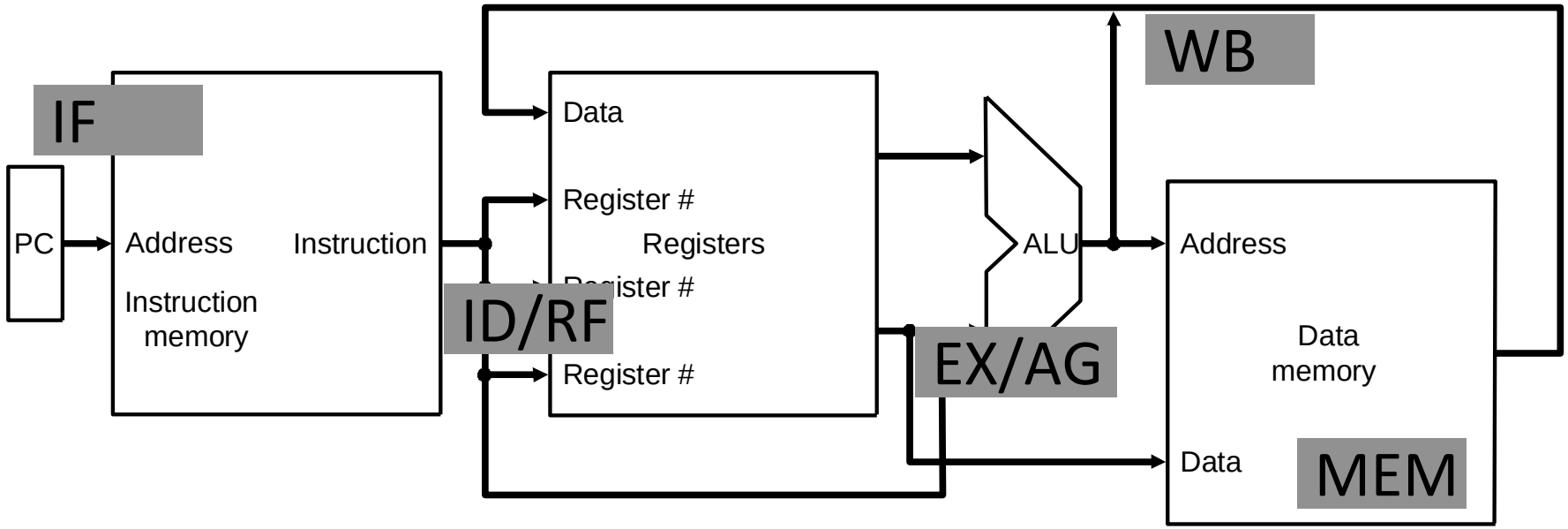


For Now, We Will Assume

- "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
- 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

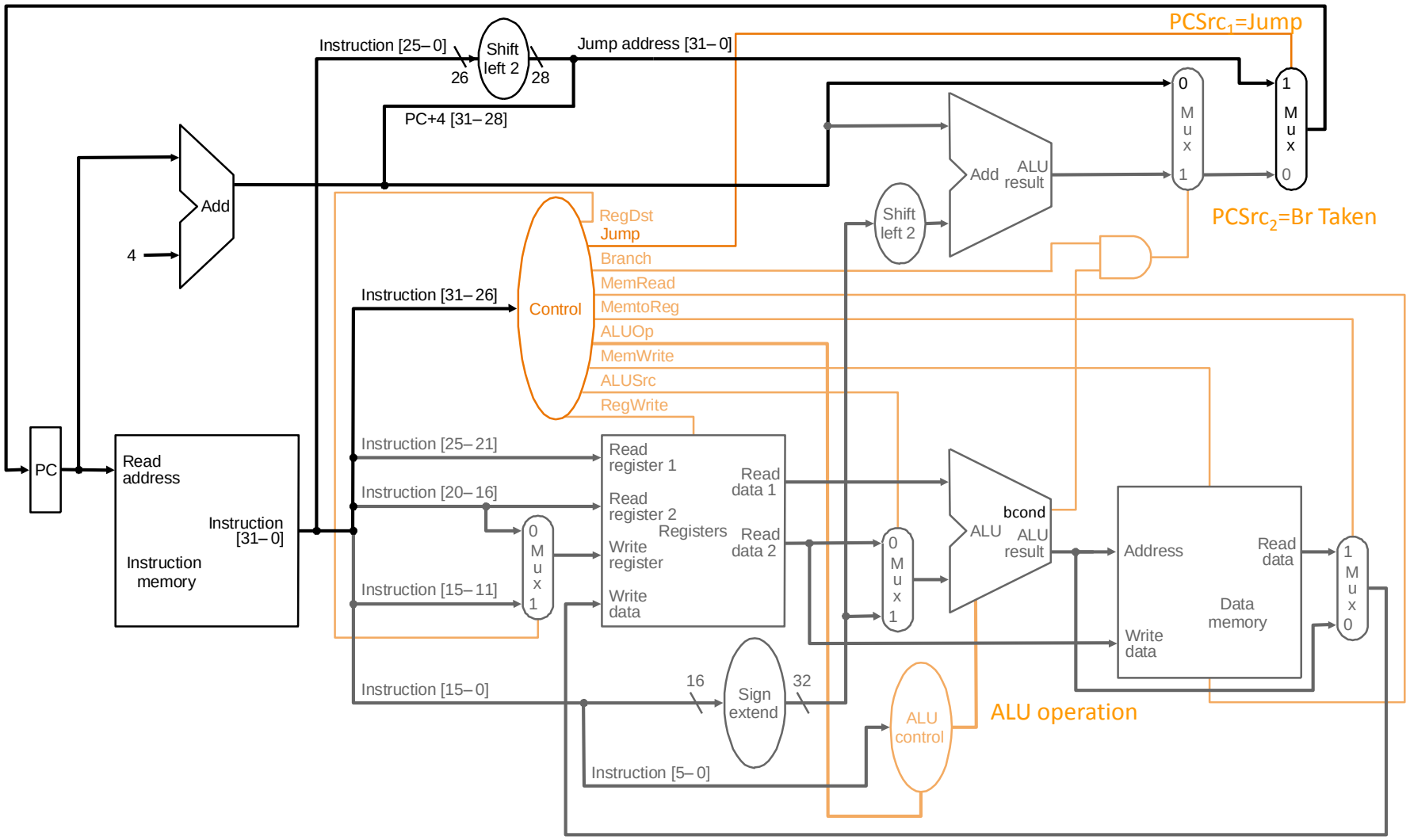
Instruction Processing

- 5 generic steps (P&H)
  - 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)



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

What Is To Come: The Full MIPS Datapath



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

JAL, JR, JALR omitted

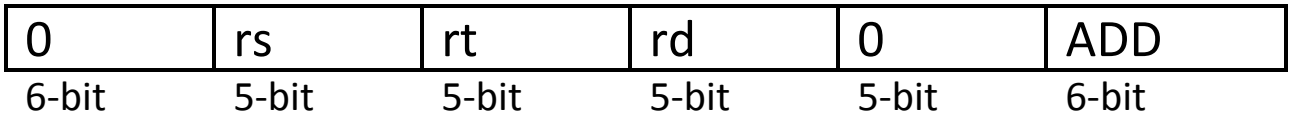
Single-Cycle Datapath for *Arithmetic and Logical Instructions*

R-Type ALU Instructions

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

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

- Machine encoding

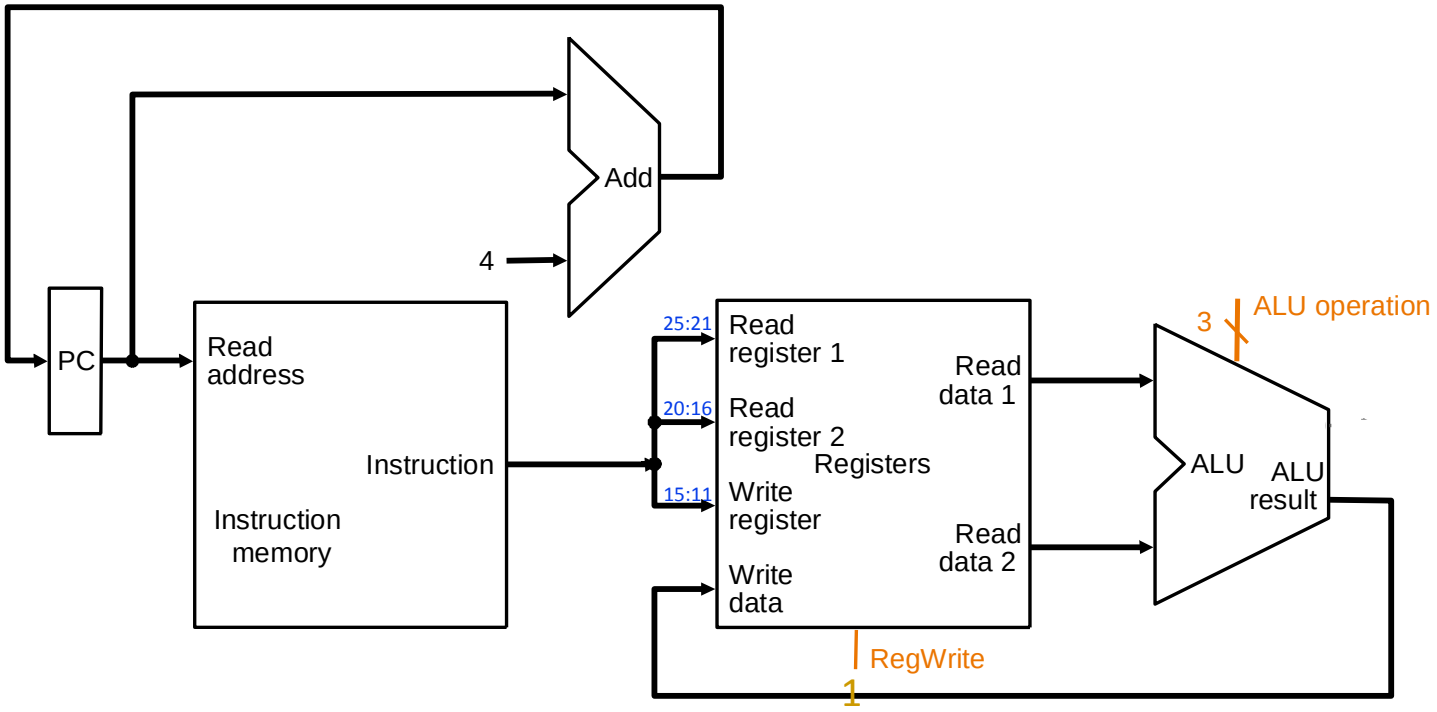


R-type

- 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

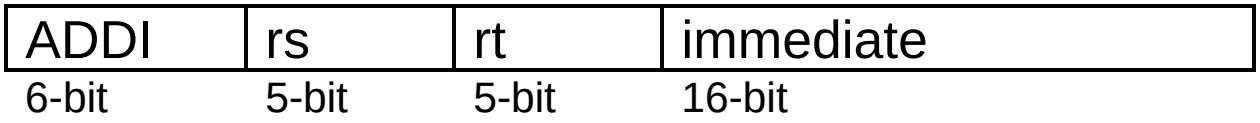
Combinational state update logic

I-Type ALU Instructions

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

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

- Machine encoding



I-type

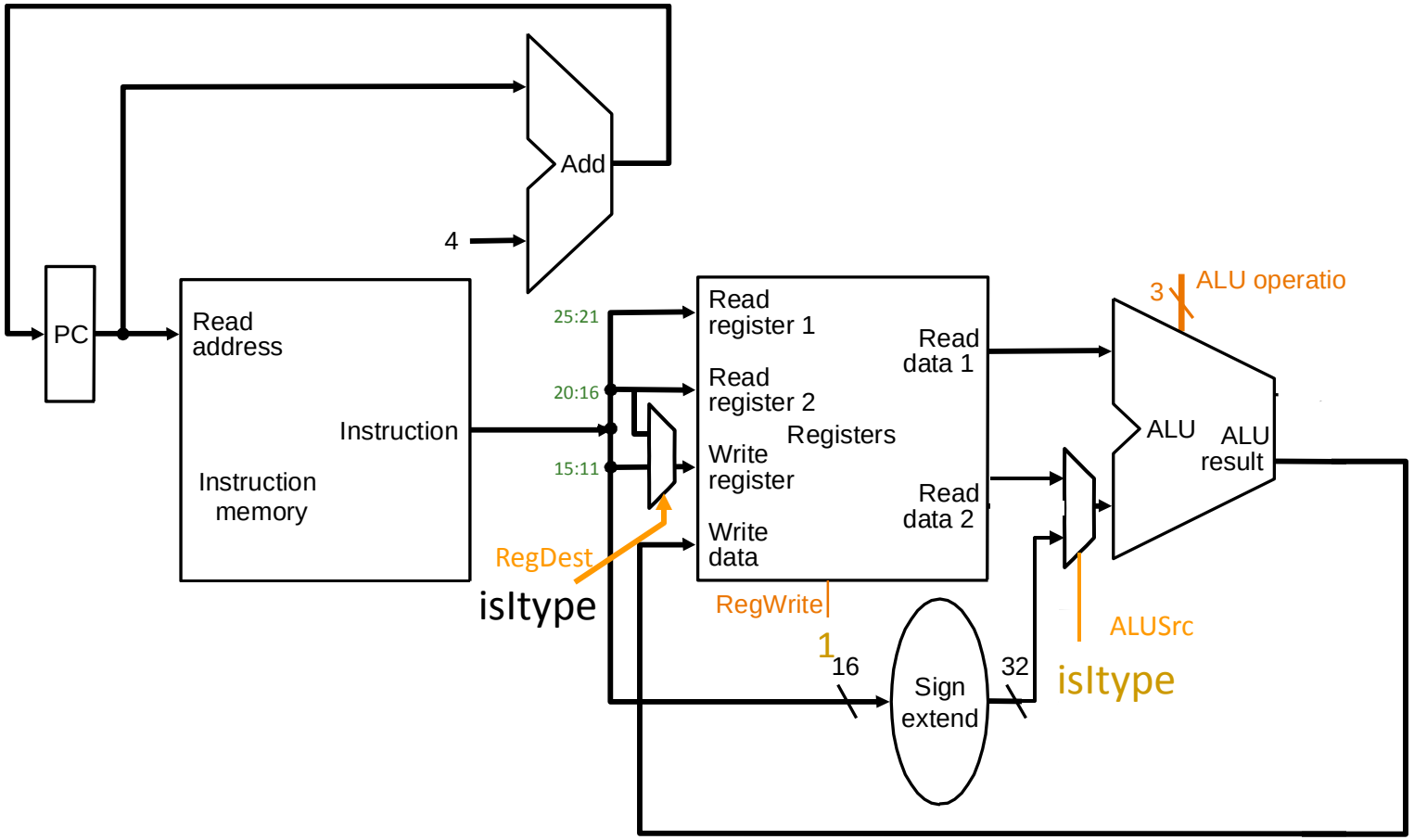
- Semantics

  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

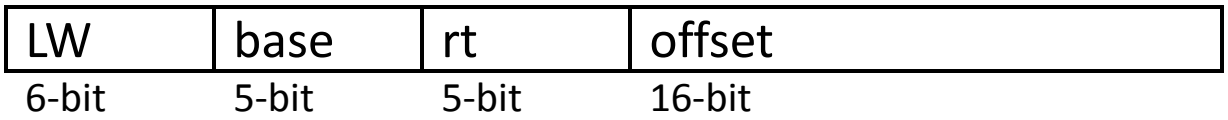
Single-Cycle Datapath for *Data Movement Instructions*

Load Instructions

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

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

- Machine encoding



I-type

- Semantics

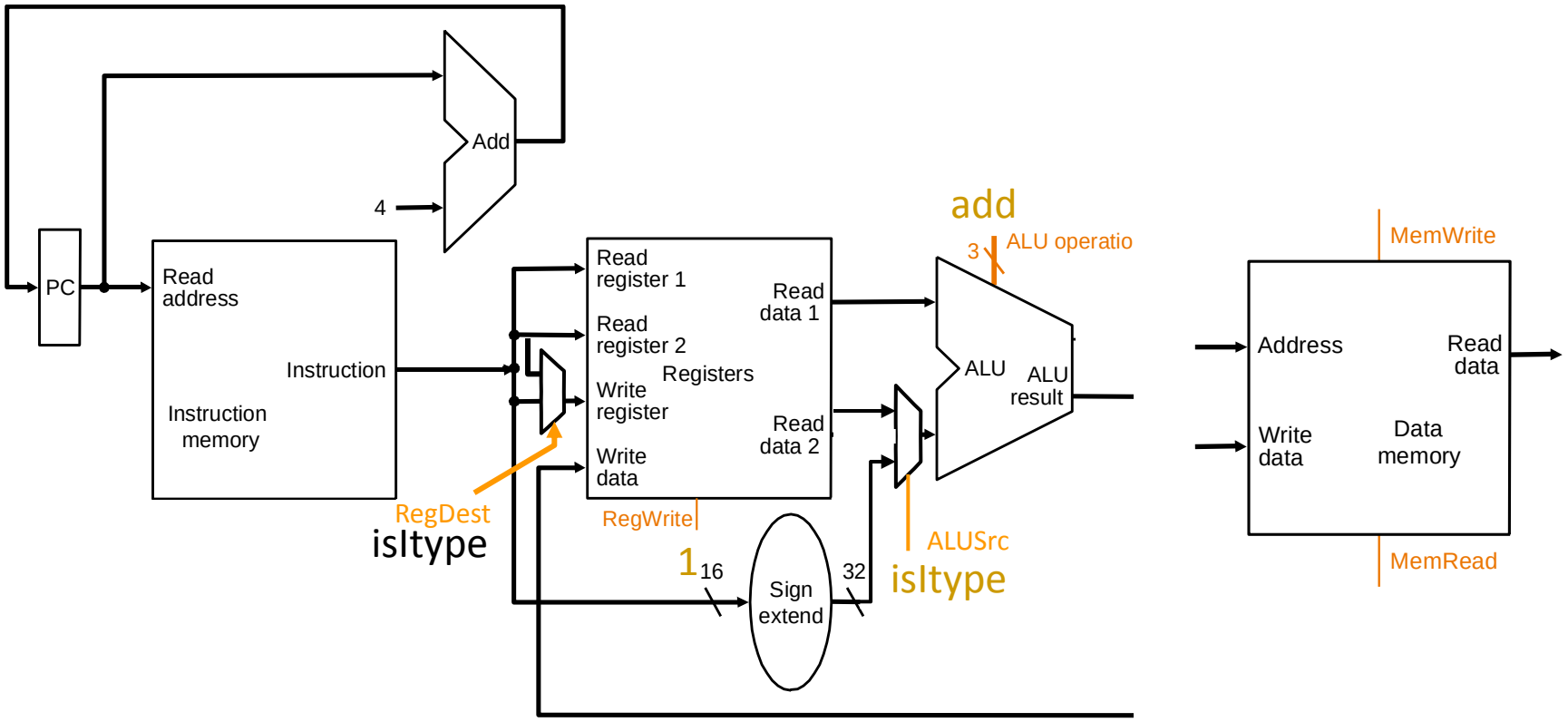
  if MEM[PC]==LW rt $offset_{16}$ (base)

  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

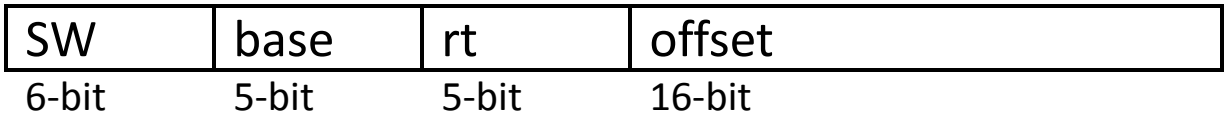
Combinational state update logic

Store Instructions

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

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

- Machine encoding



I-type

- Semantics

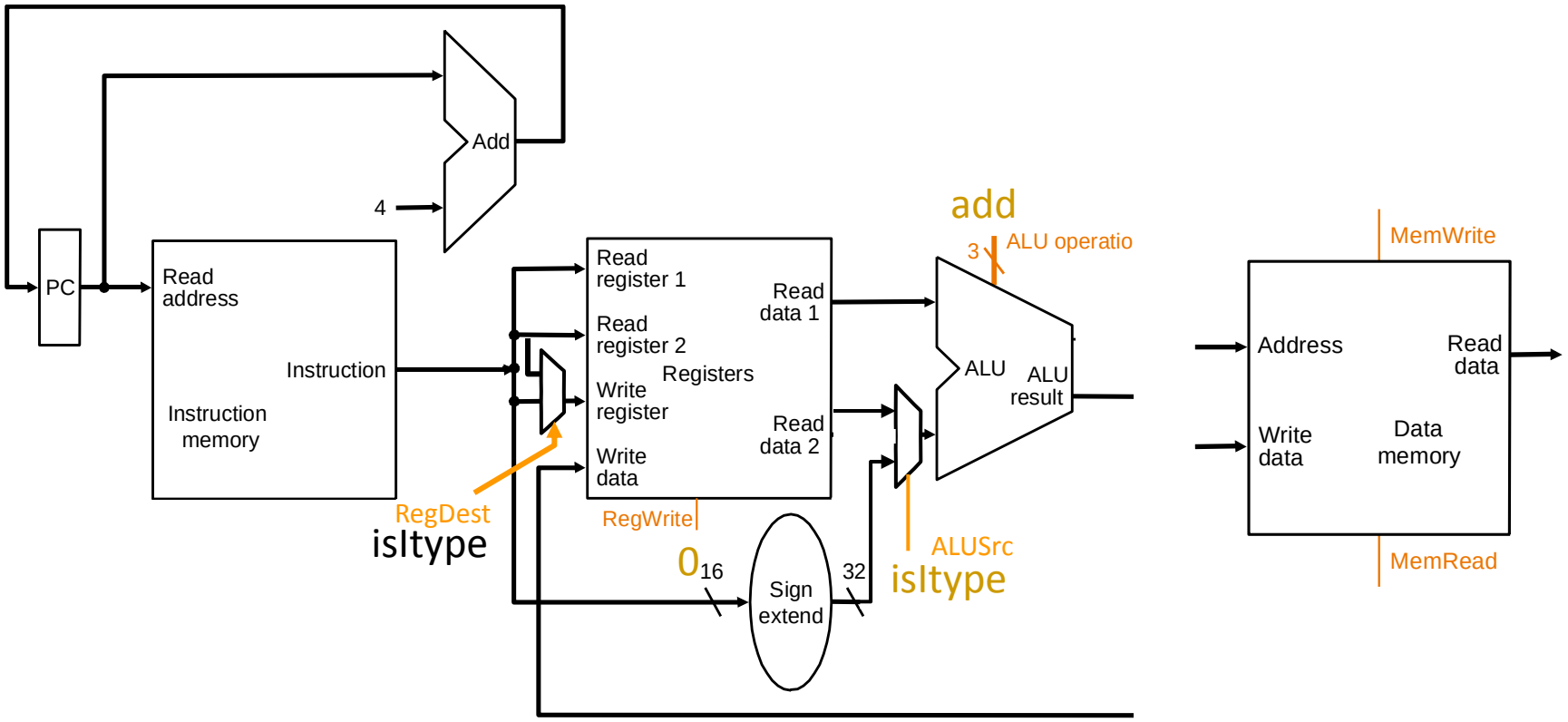
  if MEM[PC]==SW rt $offset_{16}$ (base)

  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]

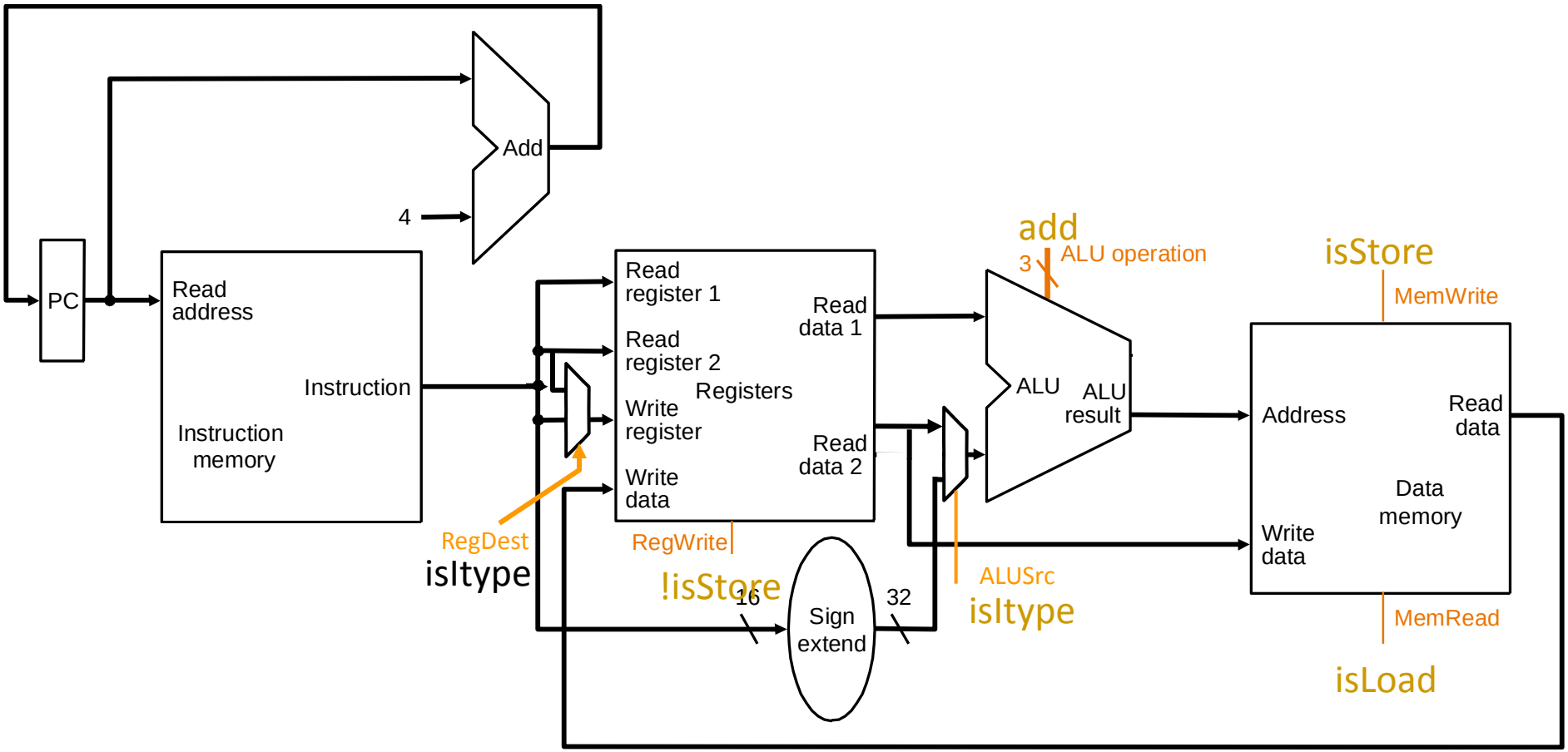
MEM[translate(EA)] ← GPR[rt]

PC ← PC + 4

IF	ID	EX	MEM	WB

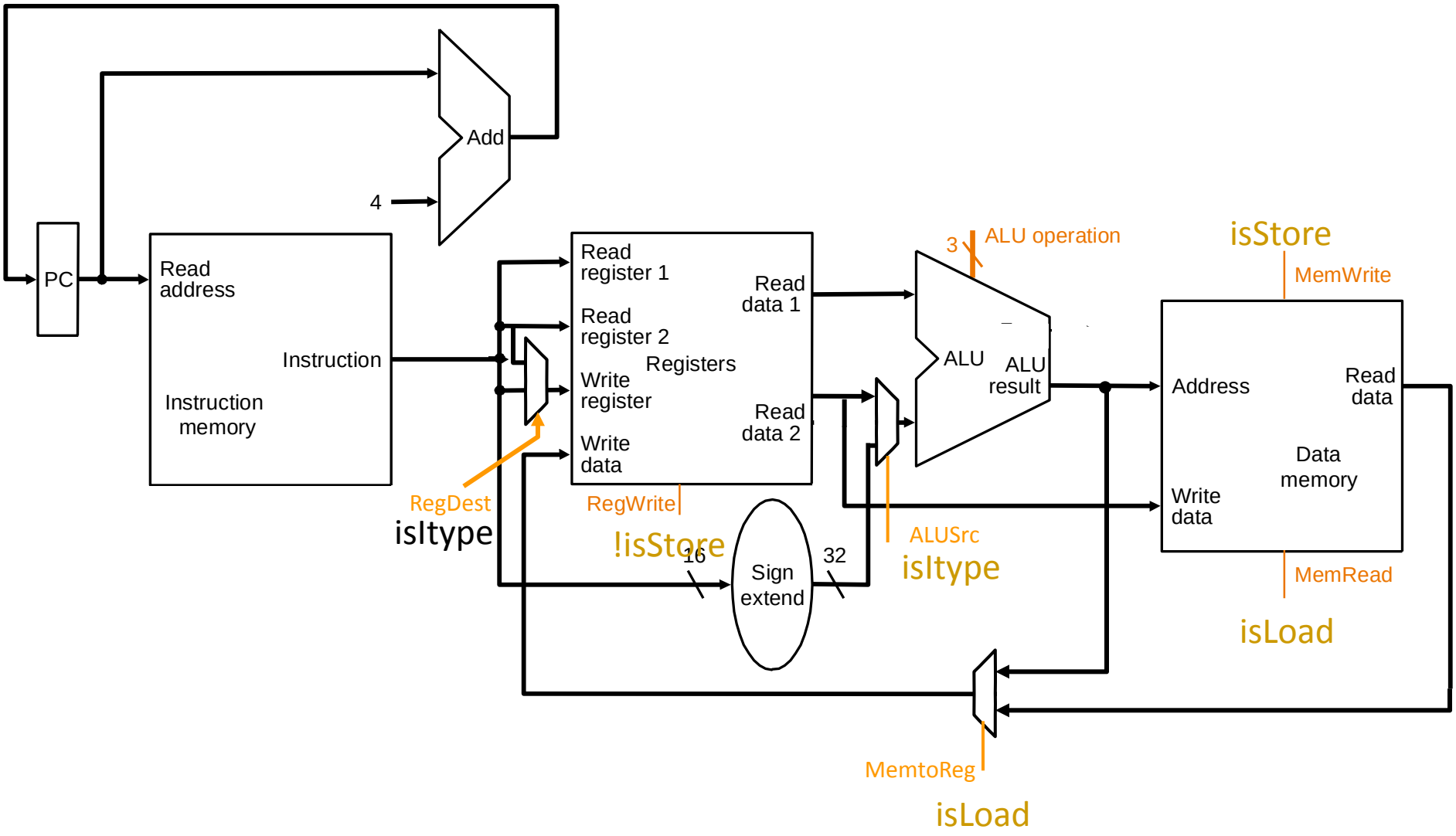
Combinational state update logic

Load-Store Datapath



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

Datapath for Non-Control-Flow Insts.



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

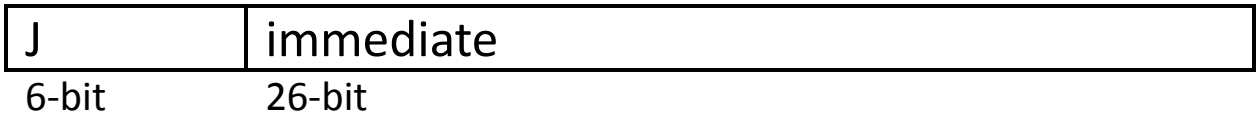
Single-Cycle Datapath for *Control Flow Instructions*

Unconditional Jump Instructions

- Assembly

  J immediate$_{26}$

- Machine encoding



J-type

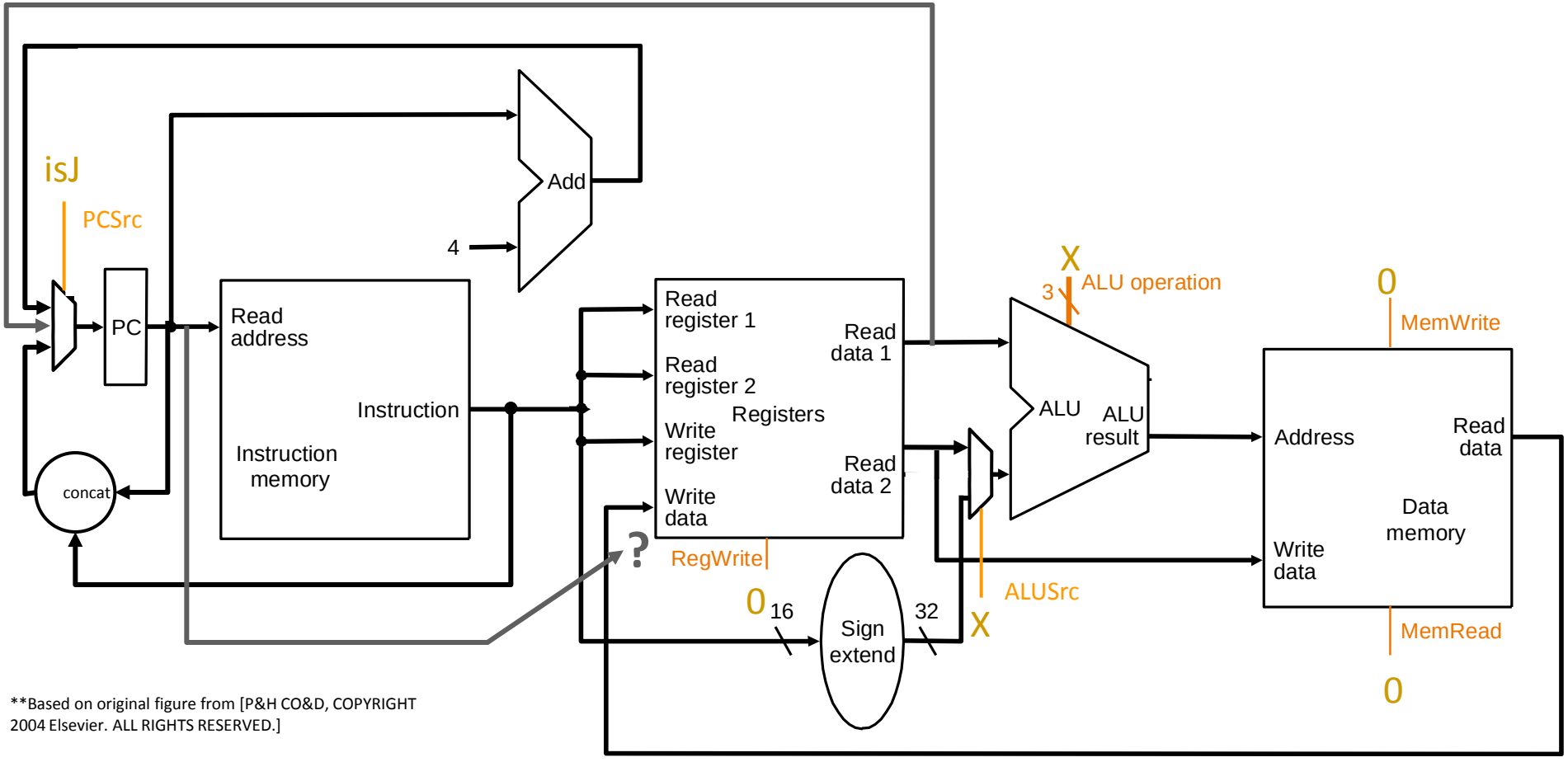
- Semantics

  if MEM[PC]==J immediate$_{26}$

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

  PC ← target

Unconditional Jump Datapath



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

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

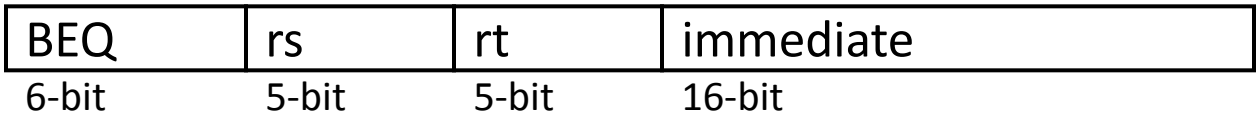
What about JR, JAL, JALR?

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)

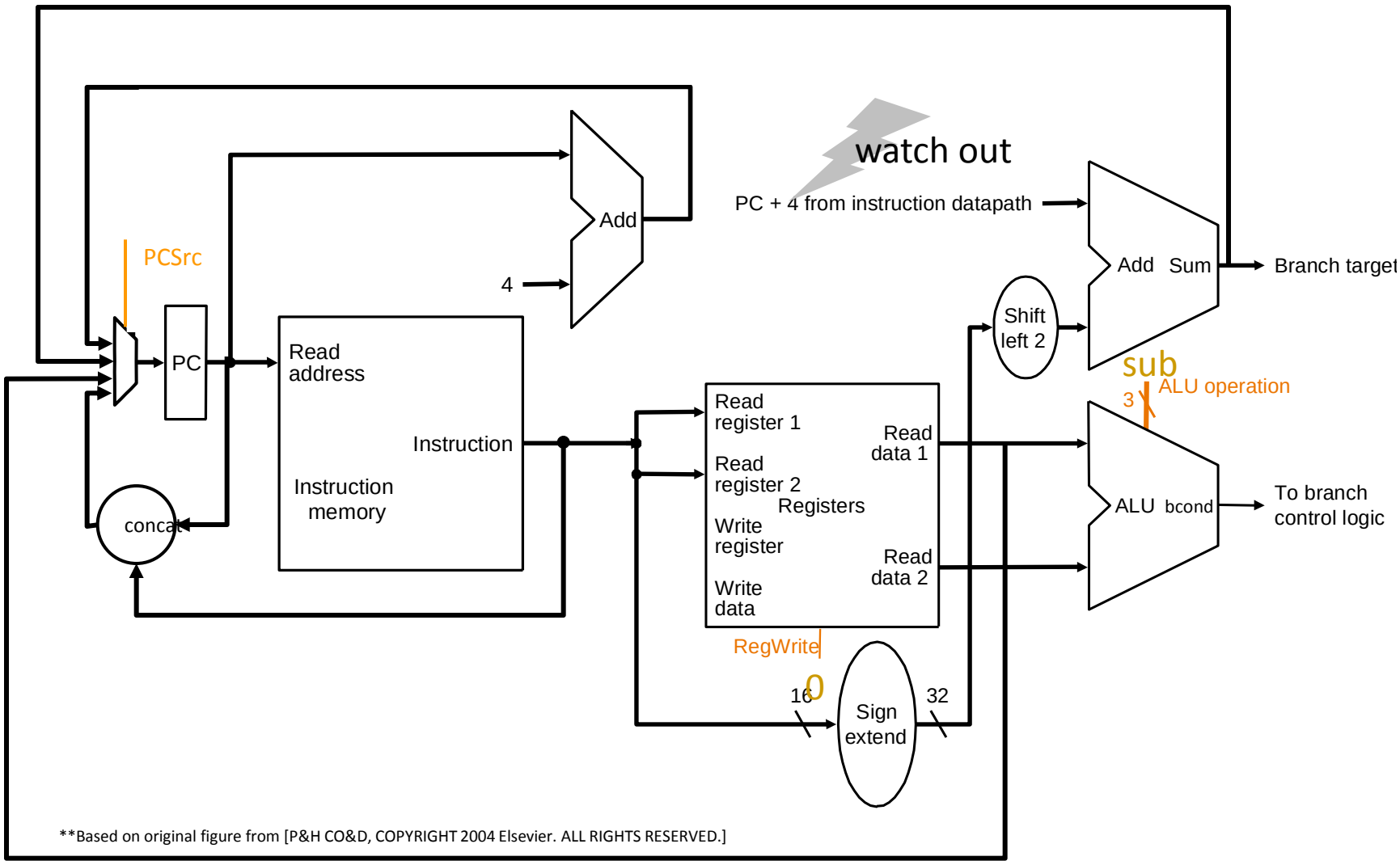
  if MEM[PC]==BEQ rs rt $immediate_{16}$

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

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

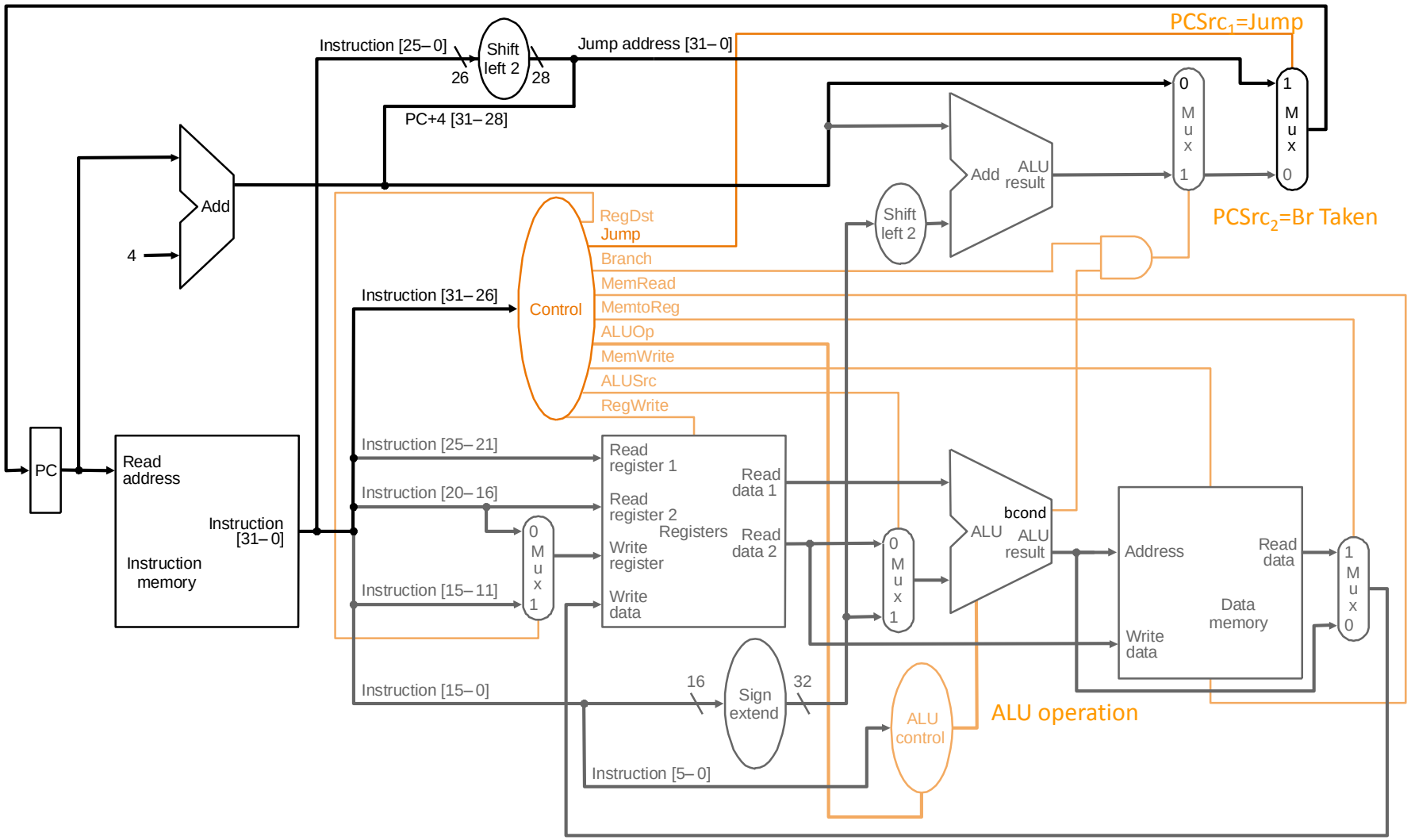
  else PC ← PC + 4

Conditional Branch Datapath (For You to Fix)



How to uphold the delayed branch semantics?

Putting It All Together



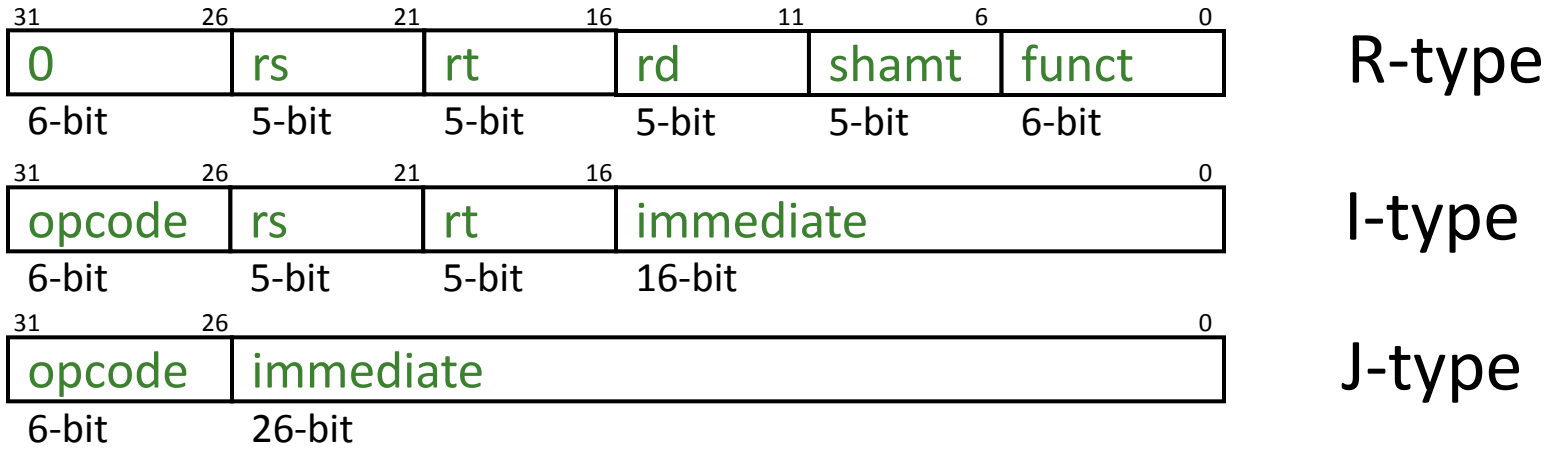
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted

Single-Cycle Control Logic

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
  - J, JR, JAL, JALR

Single-Bit Control Signals

	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 load to GPR wr. port	opcode==LW
RegWrite	GPR write disabled	GPR write enabled	(opcode!=SW) && (opcode!=Bxx) && (opcode!=J) && (opcode!=JR))

JAL and JALR require additional RegDest and MemtoReg options

Single-Bit Control Signals

	When De-asserted	When asserted	Equation
MemRead	Memory read disabled	Memory read port return 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"

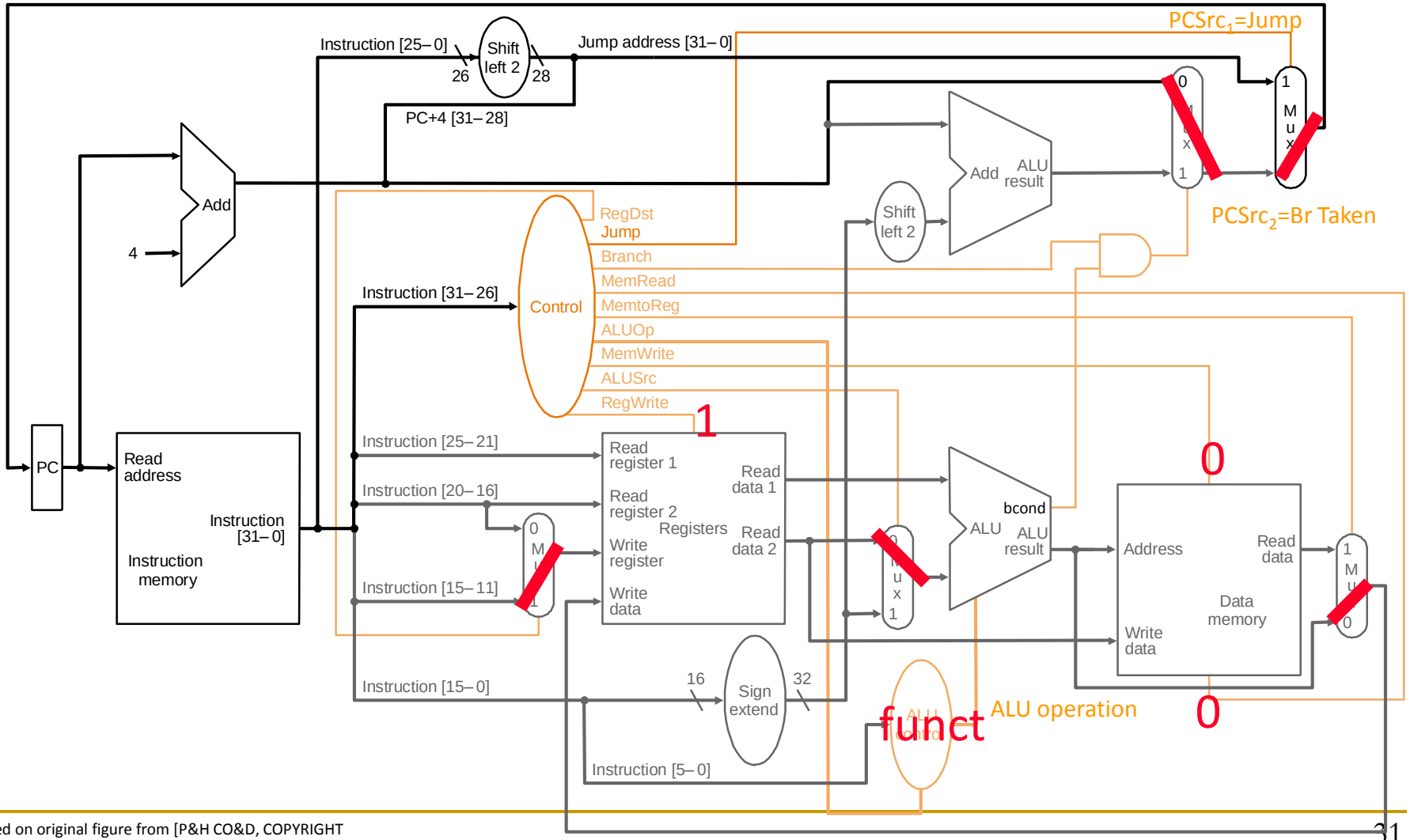
JR and JALR require additional PCSrc options

ALU Control

- case opcode
  - '0' ⇒ select operation according to funct
  - 'ALUi' ⇒ selection operation according to opcode
  - 'LW' ⇒ select addition
  - 'SW' ⇒ select addition
  - 'Bxx' ⇒ select bcond generation function
  - __ ⇒ don't care

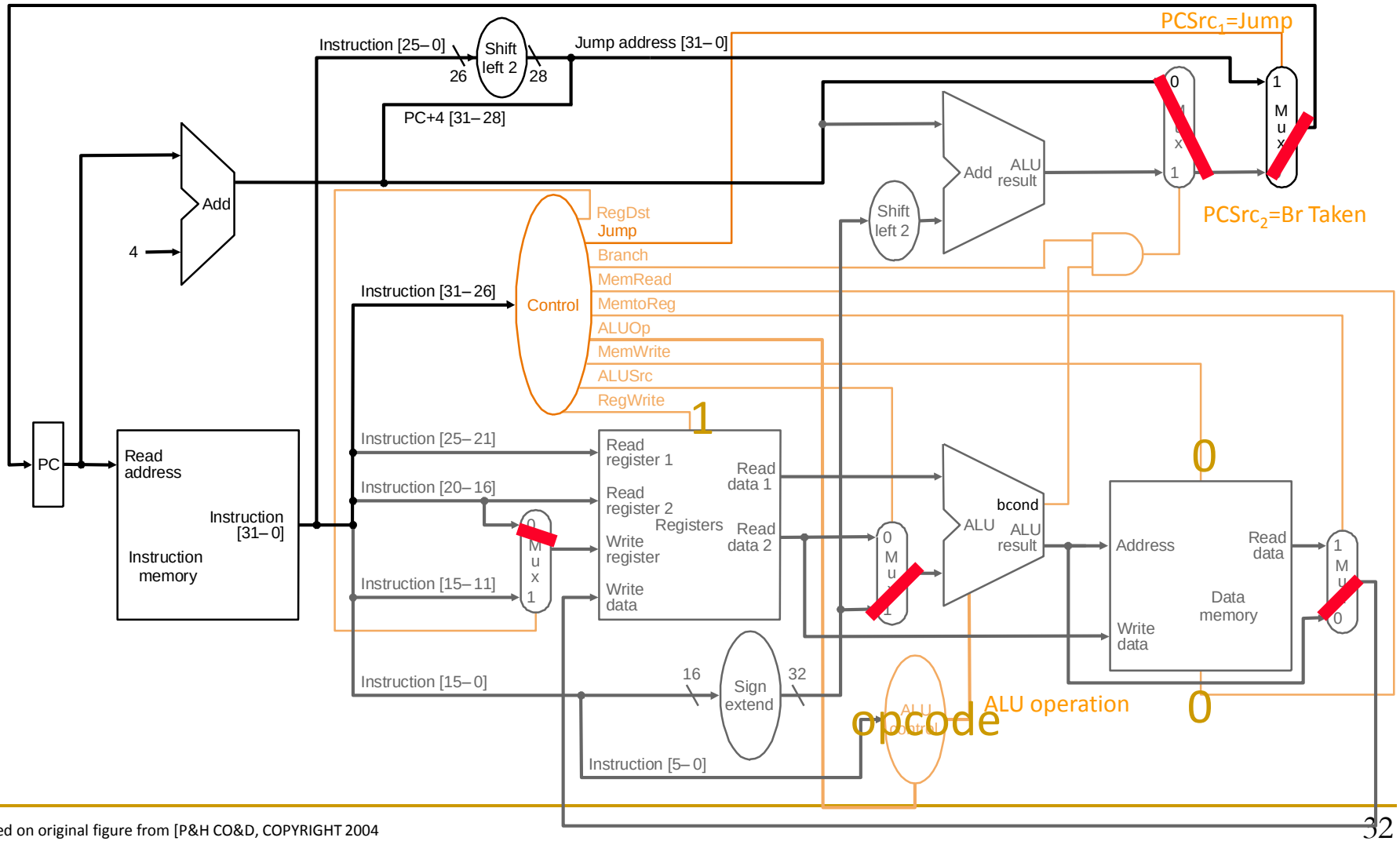
- Example ALU operations
  - ADD, SUB, AND, OR, XOR, NOR, etc.
  - bcond on equal, not equal, LE zero, GT zero, etc.

R-Type ALU



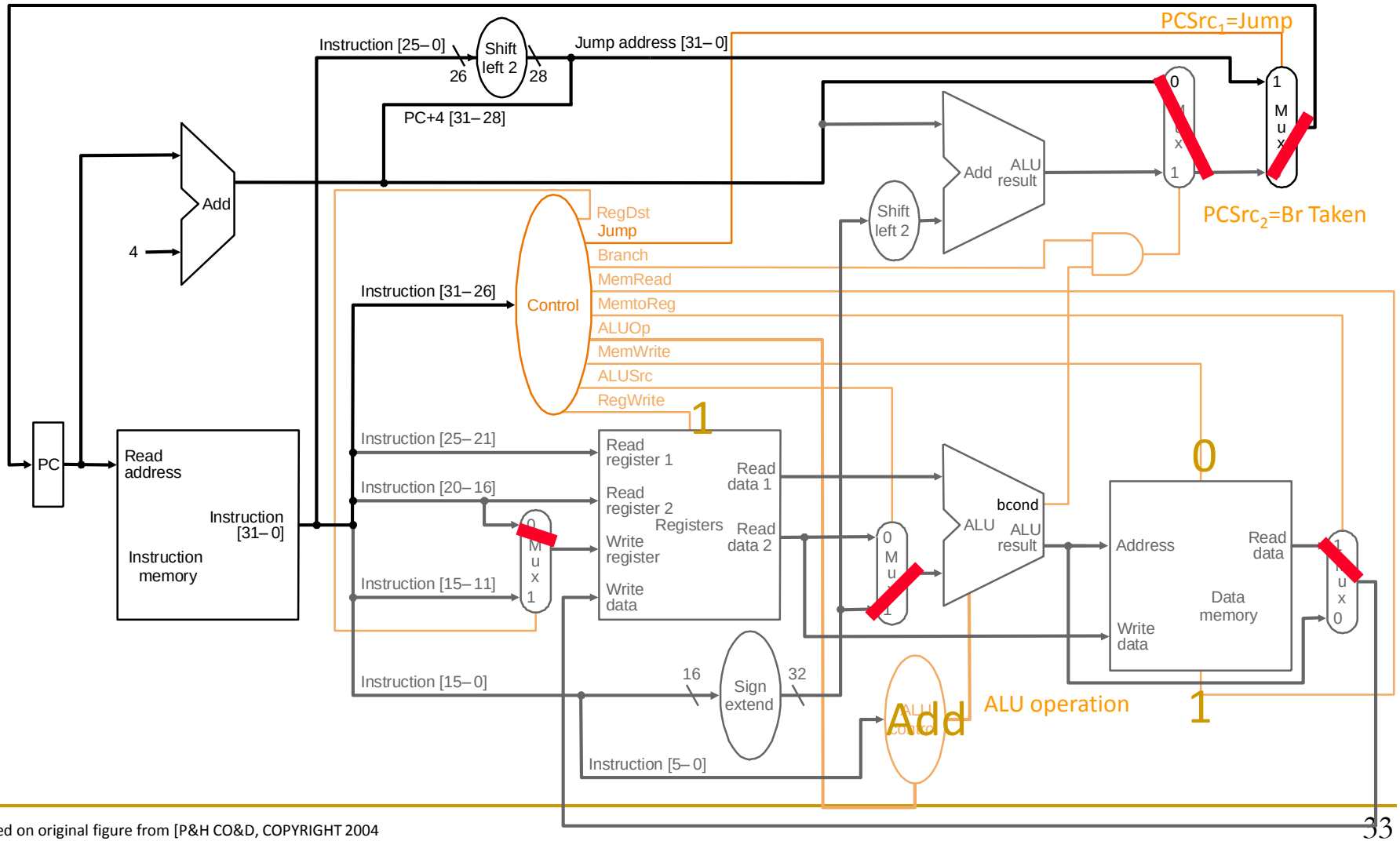
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

I-Type ALU



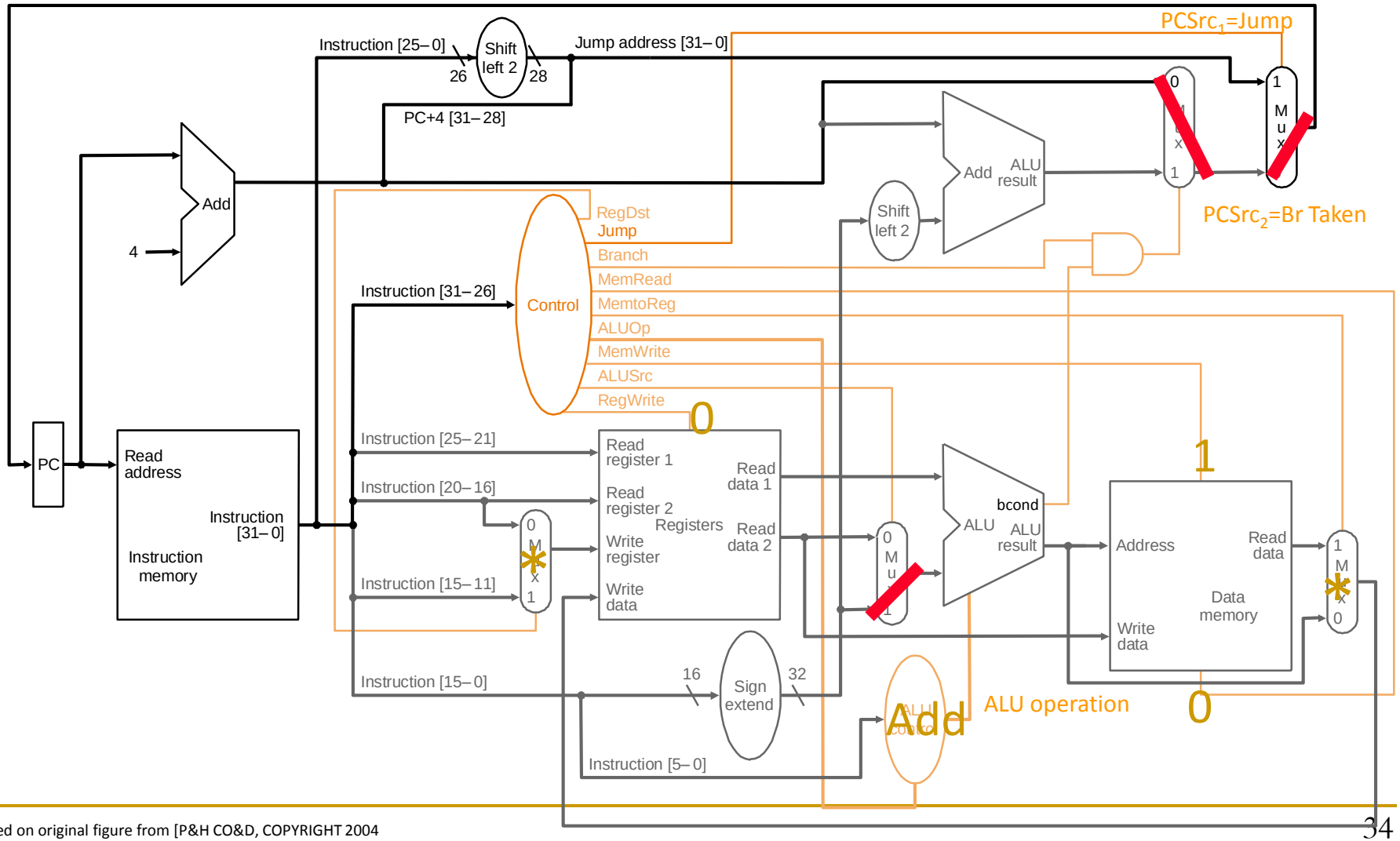
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

LW



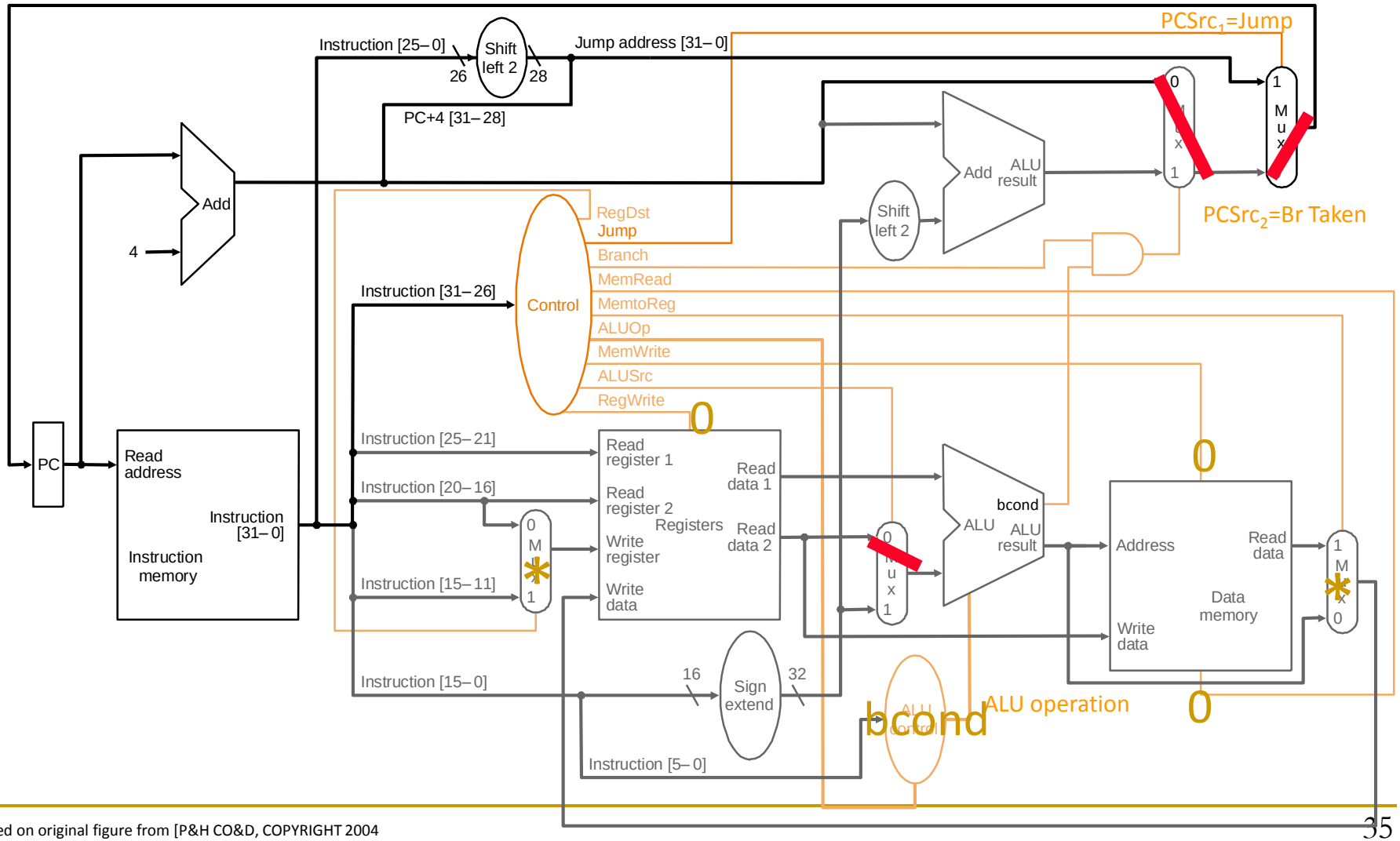
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

SW



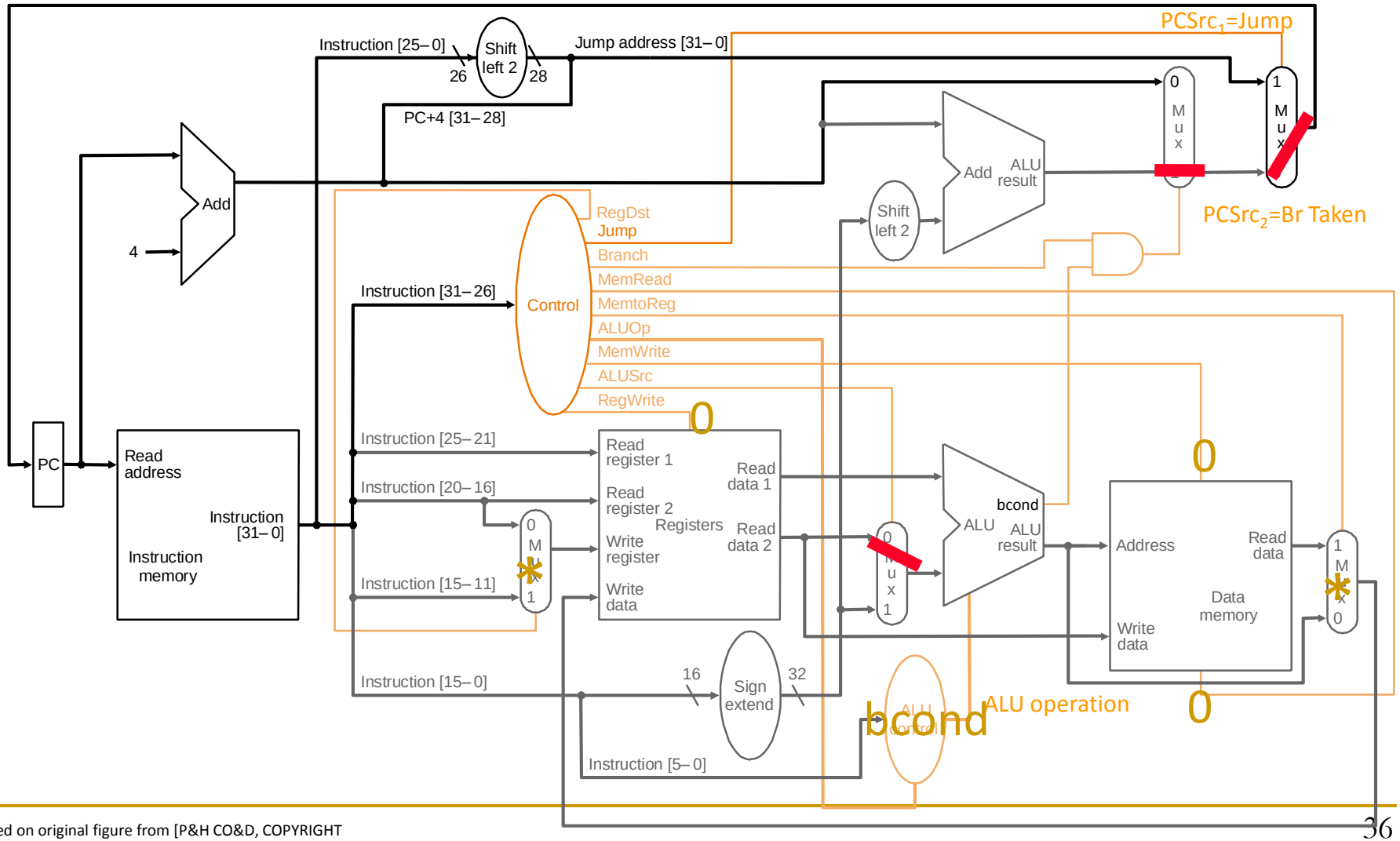
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Branch Not Taken



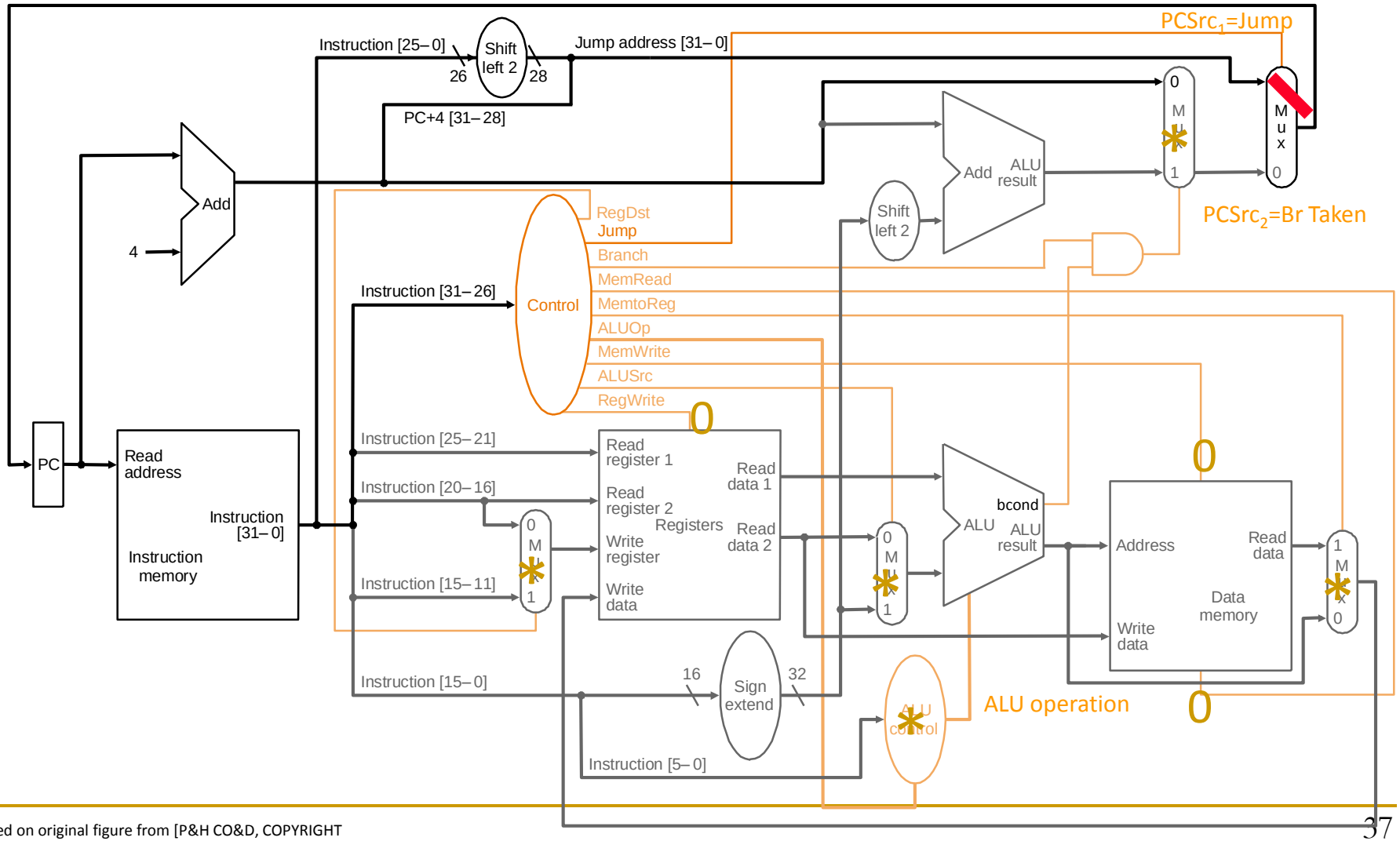
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Branch Taken



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

Jump



What is in That Control Box?

- Combinational Logic → Hardwired Control
  - Idea: Control signals generated combinationally based on instruction
  - Necessary in a single-cycle microarchitecture…

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

Evaluating the Single-Cycle Microarchitecture

A Single-Cycle Microarchitecture

- Is *this* a good idea/design?
- When is this a good design?
- When is this a bad design?
- How can we design a better microarchitecture?

A Single-Cycle Microarchitecture: Analysis

- Every instruction takes 1 cycle to execute
  - CPI (Cycles per instruction) is strictly 1

- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute

- Clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction
  - Critical path of the design is determined by the processing time of the slowest instruction

What is the Slowest Instruction to Process?

- Let's go back to the basics

- All six phases of the instruction processing cycle take a *single machine clock cycle* to complete
  - Fetch
  - Decode
  - Evaluate Address
  - Fetch Operands
  - Execute
  - Store Result

1. Instruction fetch (IF)
2. Instruction decode and register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

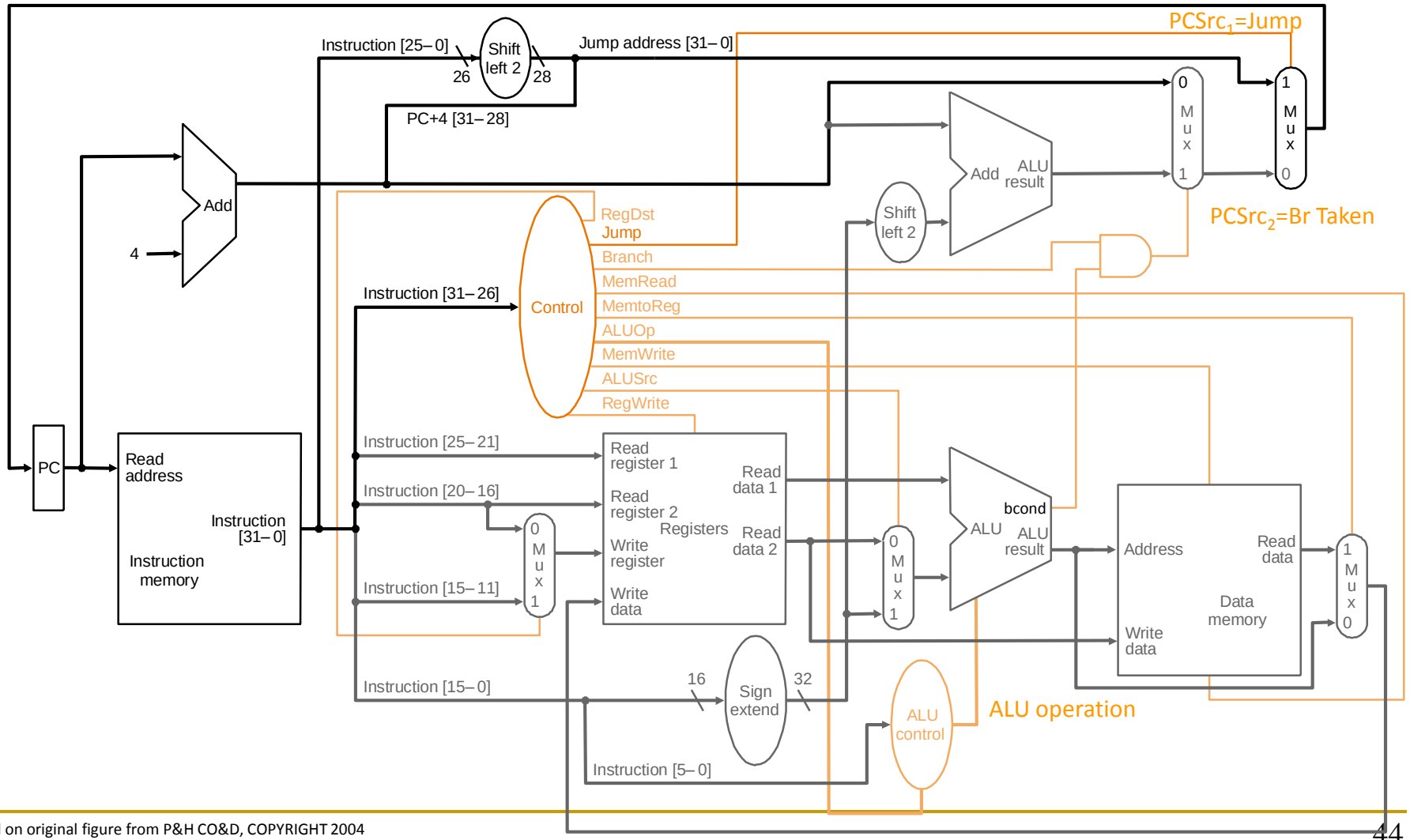
- Do each of the above phases take the same time (latency) for all instructions?

Single-Cycle Datapath Analysis

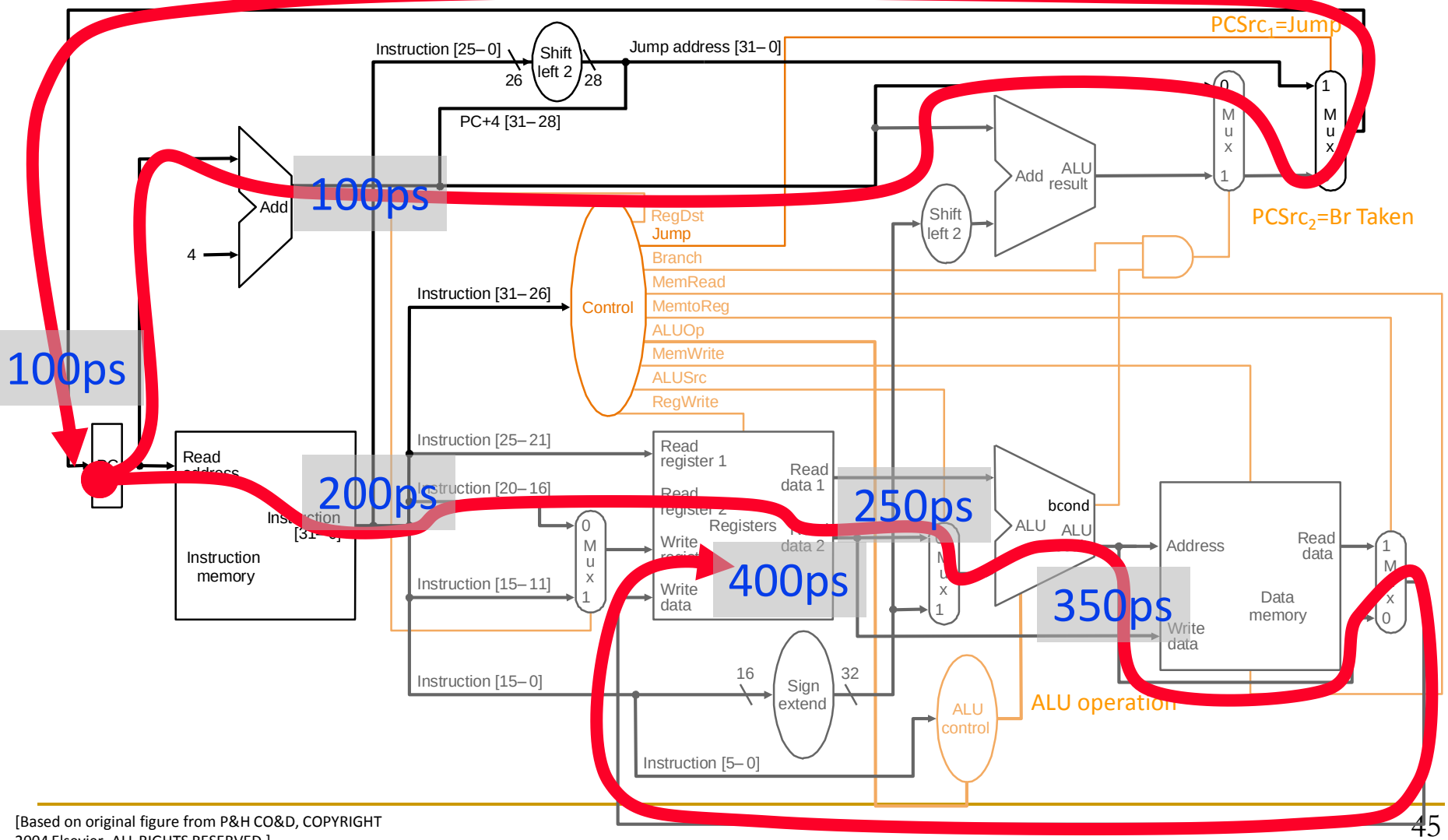
- Assume
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

Let's Find the Critical Path

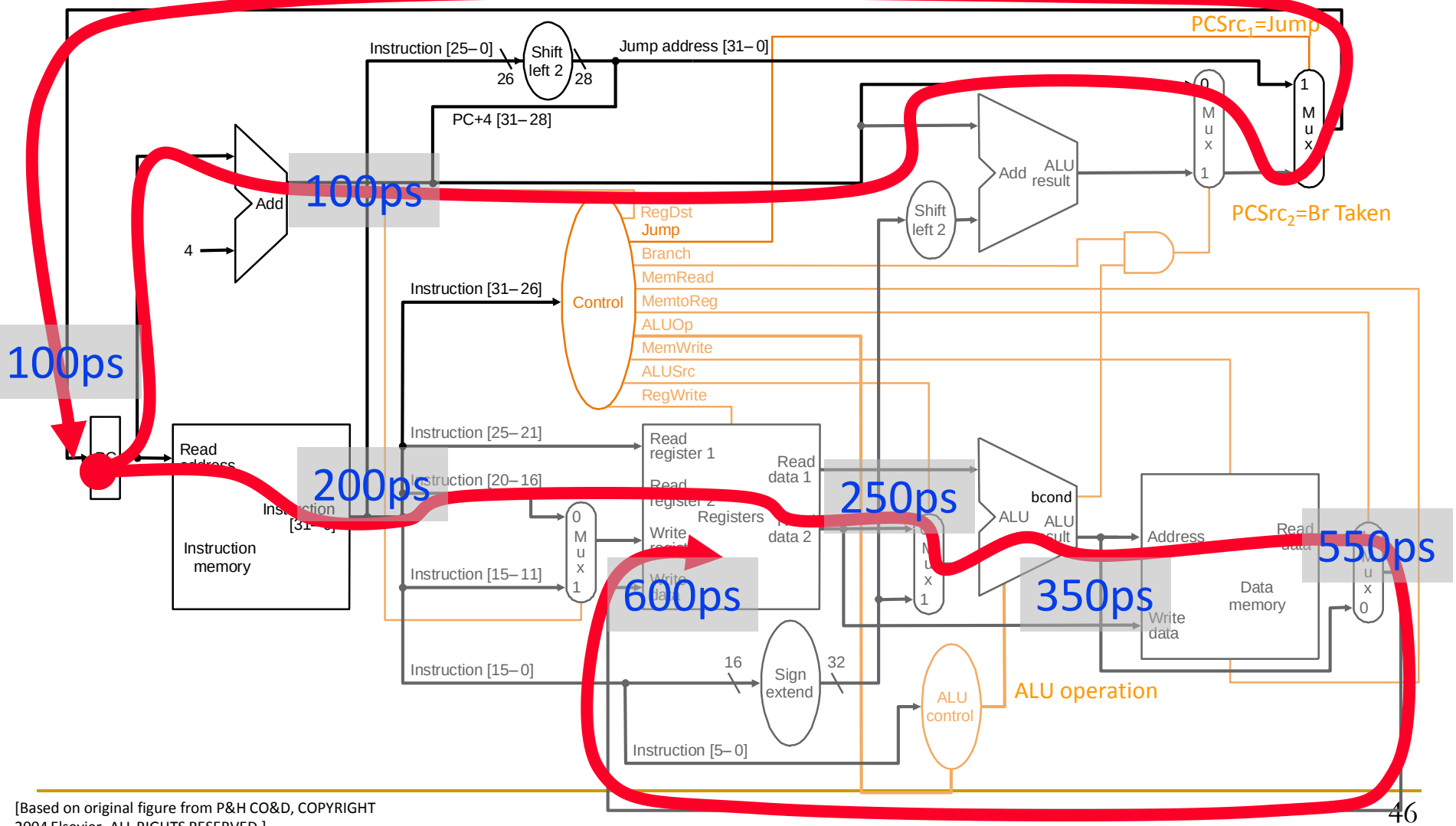


R-Type and I-Type ALU



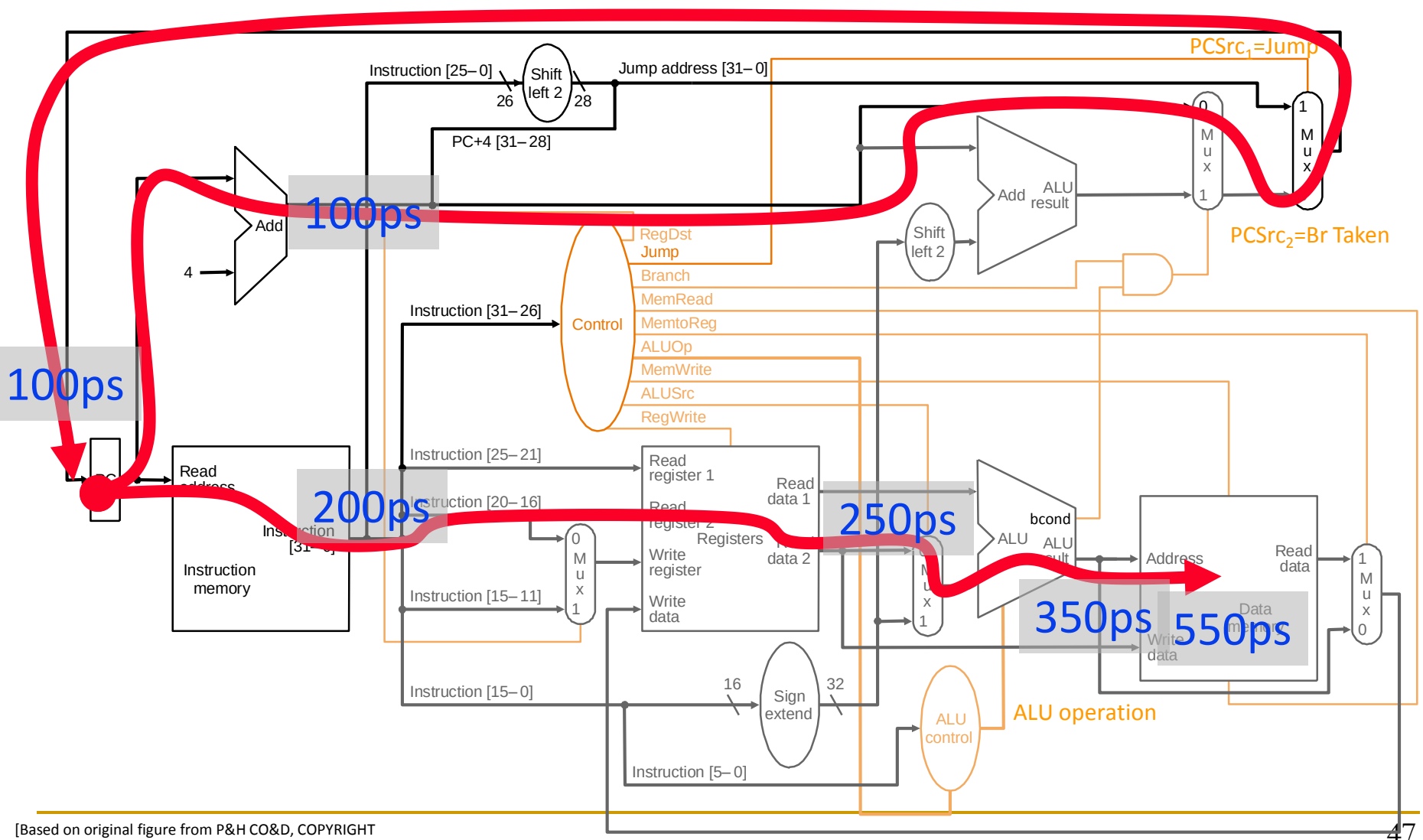
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

LW

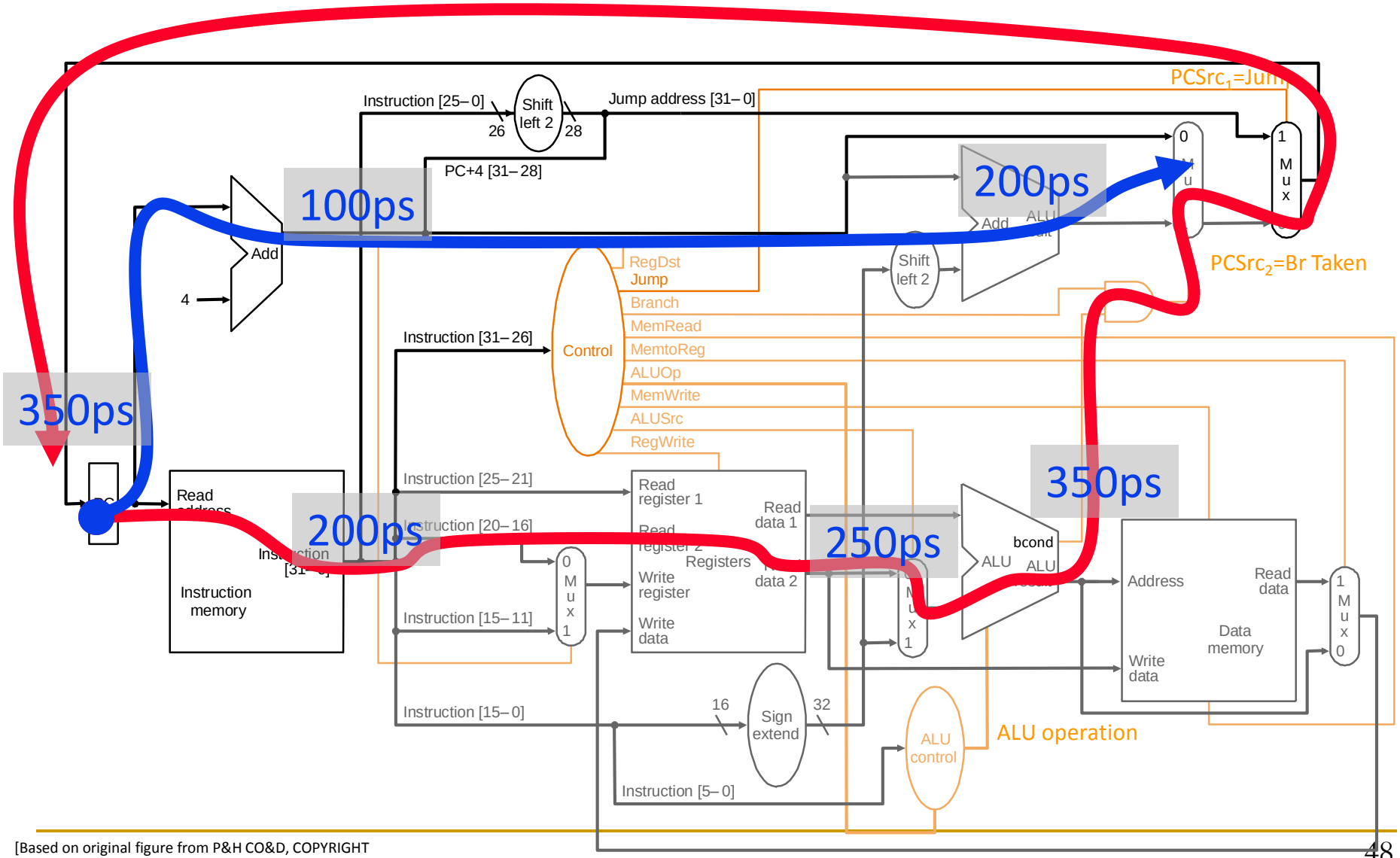


[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

SW

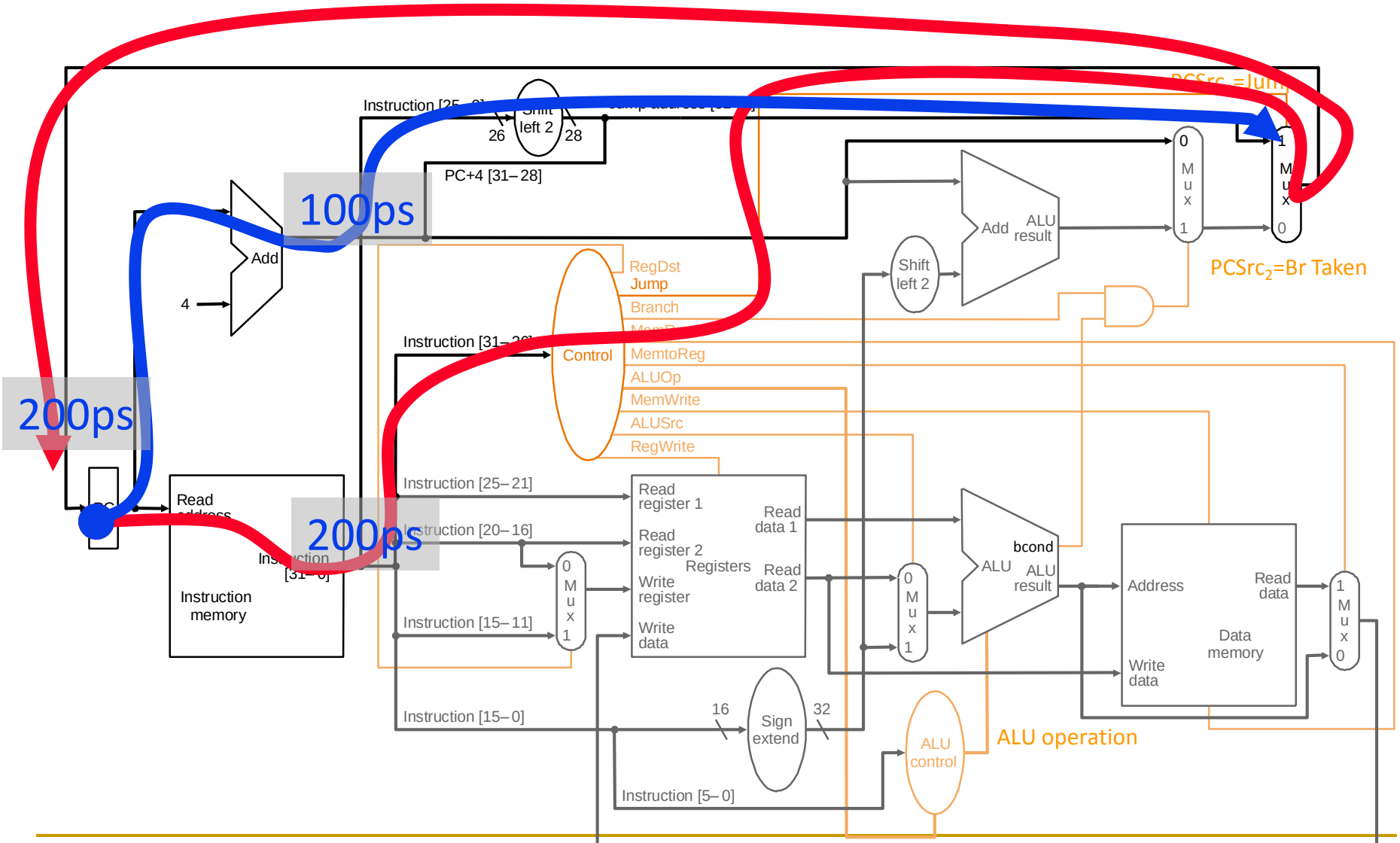


Branch Taken



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

Jump



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

What About Control Logic?

- How does that affect the critical path?

- Food for thought for you:
  - Can control logic be on the critical path?
  - A note on CDC 5600: control store access too long…

What is the Slowest Instruction to Process?

- Memory is not magic
- What if memory *sometimes* takes 100ms to access?
- Does it make sense to have a simple register to register add or jump to take {100ms+all else to do a memory operation}?
- And, what if you need to access memory more than once to process an instruction?
  - Which instructions need this?
  - Do you provide multiple ports to memory?

Single Cycle uArch: Complexity

- Contrived
  - All instructions run as slow as the slowest instruction
- Inefficient
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle
- Not necessarily the simplest way to implement an ISA
  - Single-cycle implementation of REP MOVS, INDEX, POLY?
- Not easy to optimize/improve performance
  - Optimizing the common case does not work (e.g. common instructions)
  - Need to optimize the worst case all the time

Microarchitecture Design Principles

- Critical path design
  - Find the maximum combinational logic delay and decrease it
- Bread and butter (common case) design
  - Spend time and resources on where it matters
    - i.e., improve what the machine is really designed to do
  - Common case vs. uncommon case
- Balanced design
  - Balance instruction/data flow through hardware components
  - Balance the hardware needed to accomplish the work
- *How does a single-cycle microarchitecture fare in light of these principles?*

Multi-Cycle Microarchitectures

Multi-Cycle Microarchitectures

- Goal: Let each instruction take (close to) only as much time it really needs

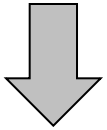
- Idea
  - Determine clock cycle time independently of instruction processing time
  - Each instruction takes as many clock cycles as it needs to take
    - Multiple state transitions per instruction
    - The states followed by each instruction is different

Remember: The "Process instruction" Step

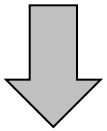
- ISA specifies abstractly what A' should be, given an instruction and A
  - It defines an abstract finite state machine where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no "intermediate states" between A and 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: AS → AS' (transform A to A' in a single clock cycle)
    - Choice 2: AS → AS+MS1 → AS+MS2 → AS+MS3 → AS' (take multiple clock cycles to transform AS to AS')

Multi-Cycle Microarchitecture

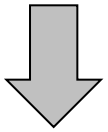
AS = Architectural (programmer visible) state
at the beginning of an instruction



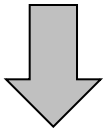
Step 1: Process part of instruction in one clock cycle



Step 2: Process part of instruction in the next clock cycle



...



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

Benefits of Multi-Cycle Design

- Critical path design
  - Can keep reducing the critical path independently of the worst-case processing time of any instruction
- Bread and butter (common case) design
  - Can optimize the number of states it takes to execute "important" instructions that make up much of the execution time
- Balanced design
  - No need to provide more capability or resources than really needed
    - An instruction that needs resource X multiple times does not require multiple X's to be implemented
    - Leads to more efficient hardware: Can reuse hardware components needed multiple times for an instruction

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

Now, we have two degrees of freedom to optimize independently

An Aside: CPI vs. Frequency

- CPI vs. Clock cycle time

- At odds with each other
  - Reducing one increases the other for a single instruction
  - Why?

- Average CPI can be amortized/reduced via concurrent processing of multiple instructions
  - The same cycle is devoted to processing multiple instructions
  - Example: Pipelining, superscalar execution

A Multi-Cycle Microarchitecture
A Closer Look

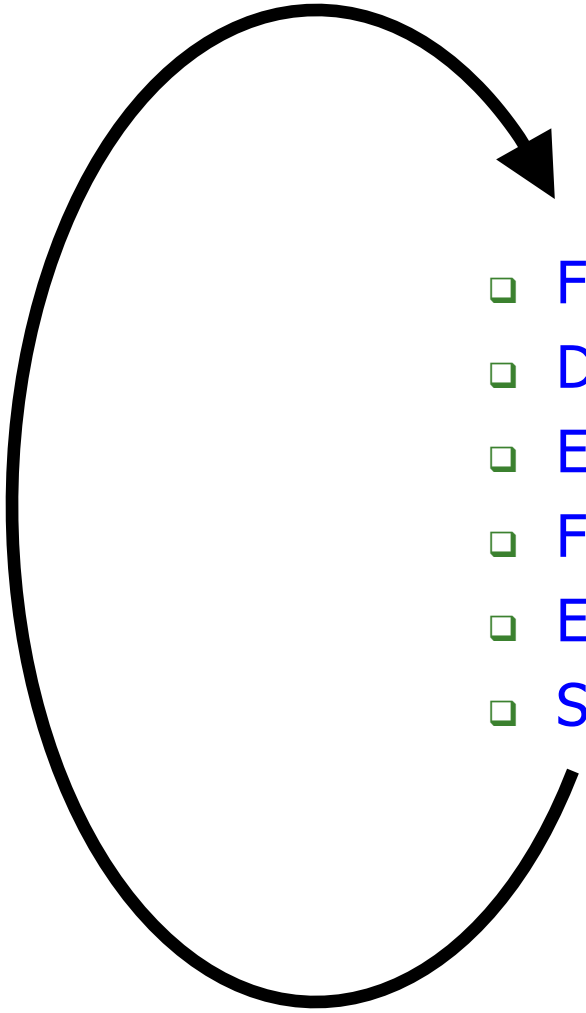
How Do We Implement This?

- Maurice Wilkes, "The Best Way to Design an Automatic Calculating Machine," Manchester Univ. Computer Inaugural Conf., 1951.

- The concept of microcoded/microprogrammed machines

- Realization
  - One can implement the "process instruction" step as a finite state machine that sequences between states and eventually returns back to the "fetch instruction" state
  - A state is defined by the control signals asserted in it
  - Control signals for the next state determined in current state

The Instruction Processing Cycle



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

A Basic Multi-Cycle Microarchitecture

- Instruction processing cycle divided into "states"
  - A stage in the instruction processing cycle can take multiple states
- A multi-cycle microarchitecture sequences from state to state to process an instruction
  - The behavior of the machine in a state is completely determined by control signals in that state
- The behavior of the entire processor is specified fully by a *finite state machine*
- In a state (clock cycle), control signals control
  - How the datapath should process the data
  - How to generate the control signals for the next clock cycle

Microprogrammed Control Terminology

- Control signals associated with the current state
  - Microinstruction
- Act of transitioning from one state to another
  - Determining the next state and the microinstruction for the next state
  - Microsequencing
- Control store stores control signals for every possible state
  - Store for microinstructions for the entire FSM
- Microsequencer determines which set of control signals will be used in the next clock cycle (i.e., next state)

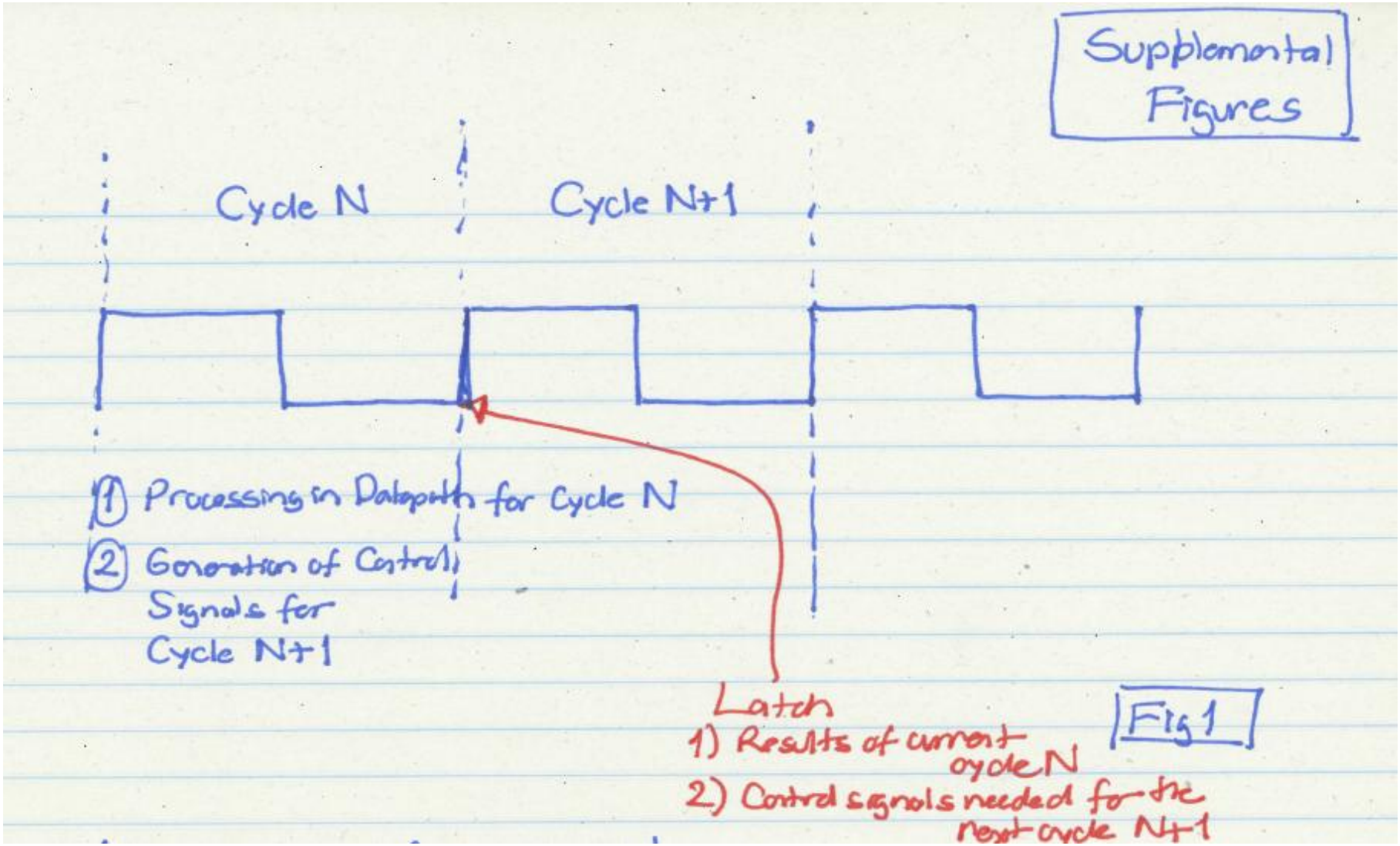
What Happens In A Clock Cycle?

- The control signals (microinstruction) for the current state control
  - Processing in the data path
  - Generation of control signals (microinstruction) for the next cycle
  - See Supplemental Figure 1 (next slide)

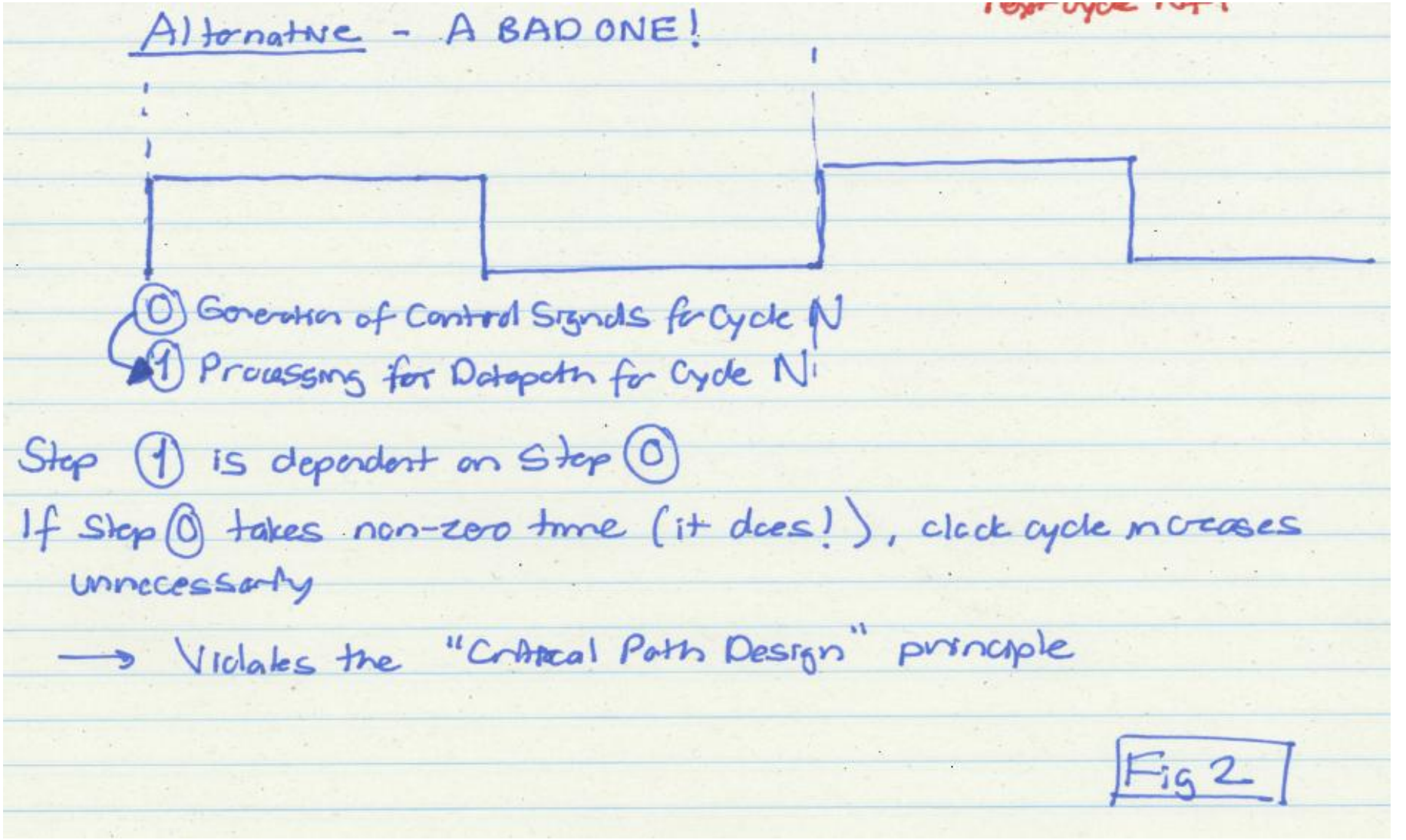
- Datapath and microsequencer operate concurrently

- Question: why not generate control signals for the current cycle in the current cycle?
  - This will lengthen the clock cycle
  - Why would it lengthen the clock cycle?
  - See Supplemental Figure 2

A Clock Cycle



A Bad Clock Cycle!



A Simple LC-3b Control and Datapath

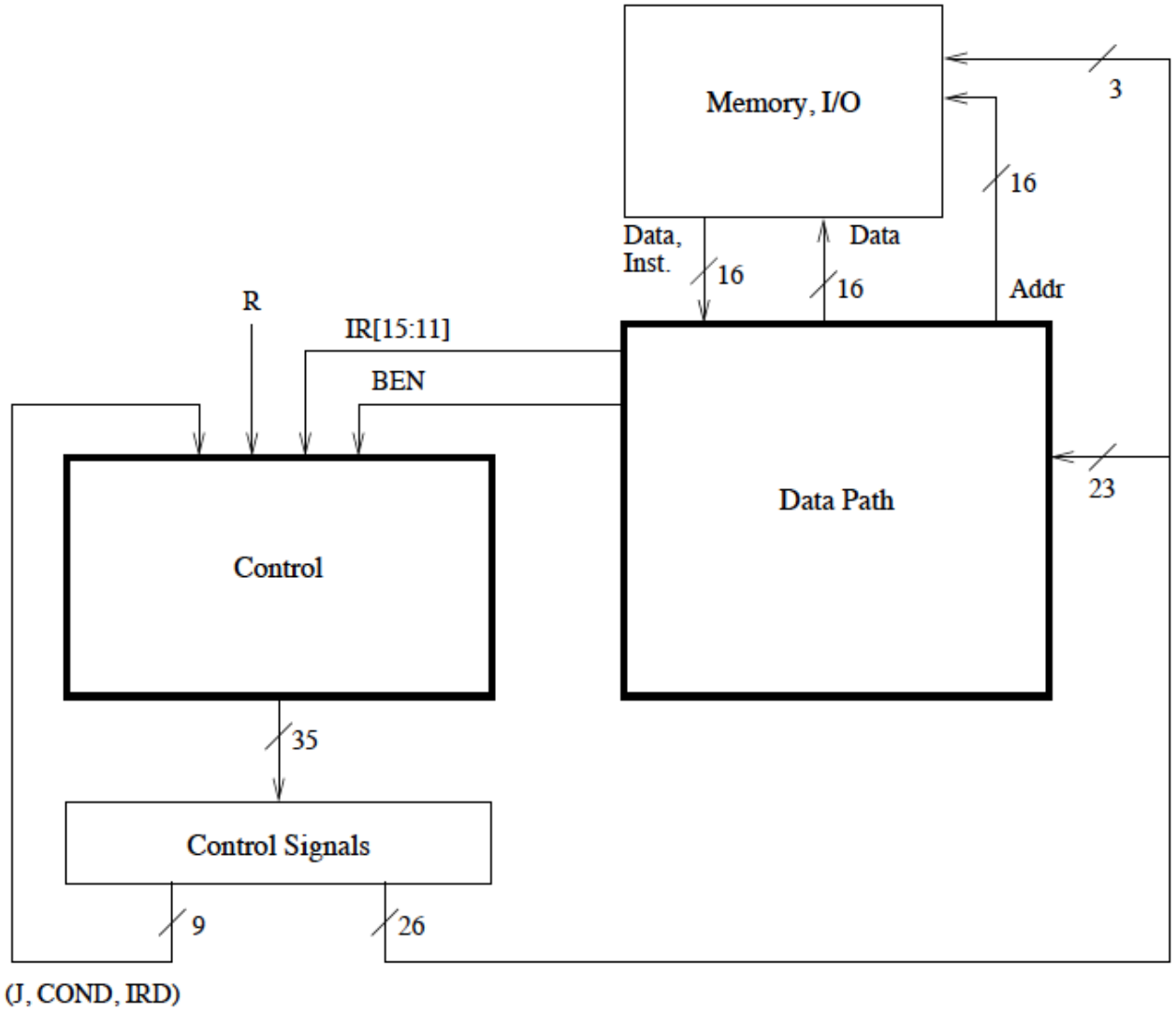


Figure C.1: Microarchitecture of the LC-3b, major components

What Determines Next-State Control Signals?

- What is happening in the current clock cycle
  - See the 9 control signals coming from "Control" block
    - What are these for?
- The instruction that is being executed
  - IR[15:11] coming from the Data Path
- Whether the condition of a branch is met, if the instruction being processed is a branch
  - BEN bit coming from the datapath
- Whether the memory operation is completing in the current cycle, if one is in progress
  - R bit coming from memory

A Simple LC-3b Control and Datapath

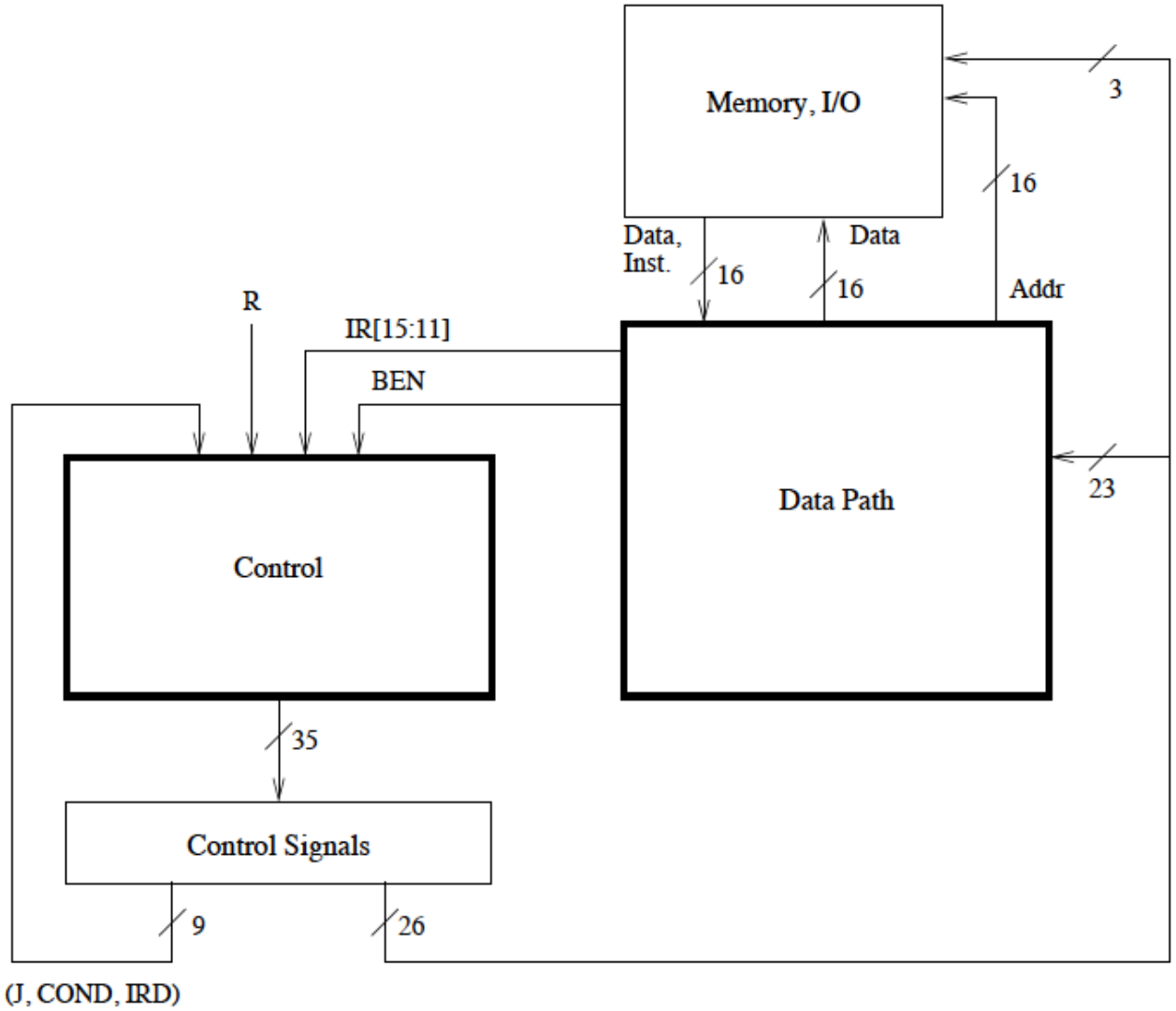


Figure C.1: Microarchitecture of the LC-3b, major components

The State Machine for Multi-Cycle Processing

- The behavior of the LC-3b uarch is completely determined by
  - the 35 control signals and
  - additional 7 bits that go into the control logic from the datapath

- 35 control signals completely describe the state of the control structure

- We can completely describe the behavior of the LC-3b as a state machine, i.e. a directed graph of
  - Nodes (one corresponding to each state)
  - Arcs (showing flow from each state to the next state(s))

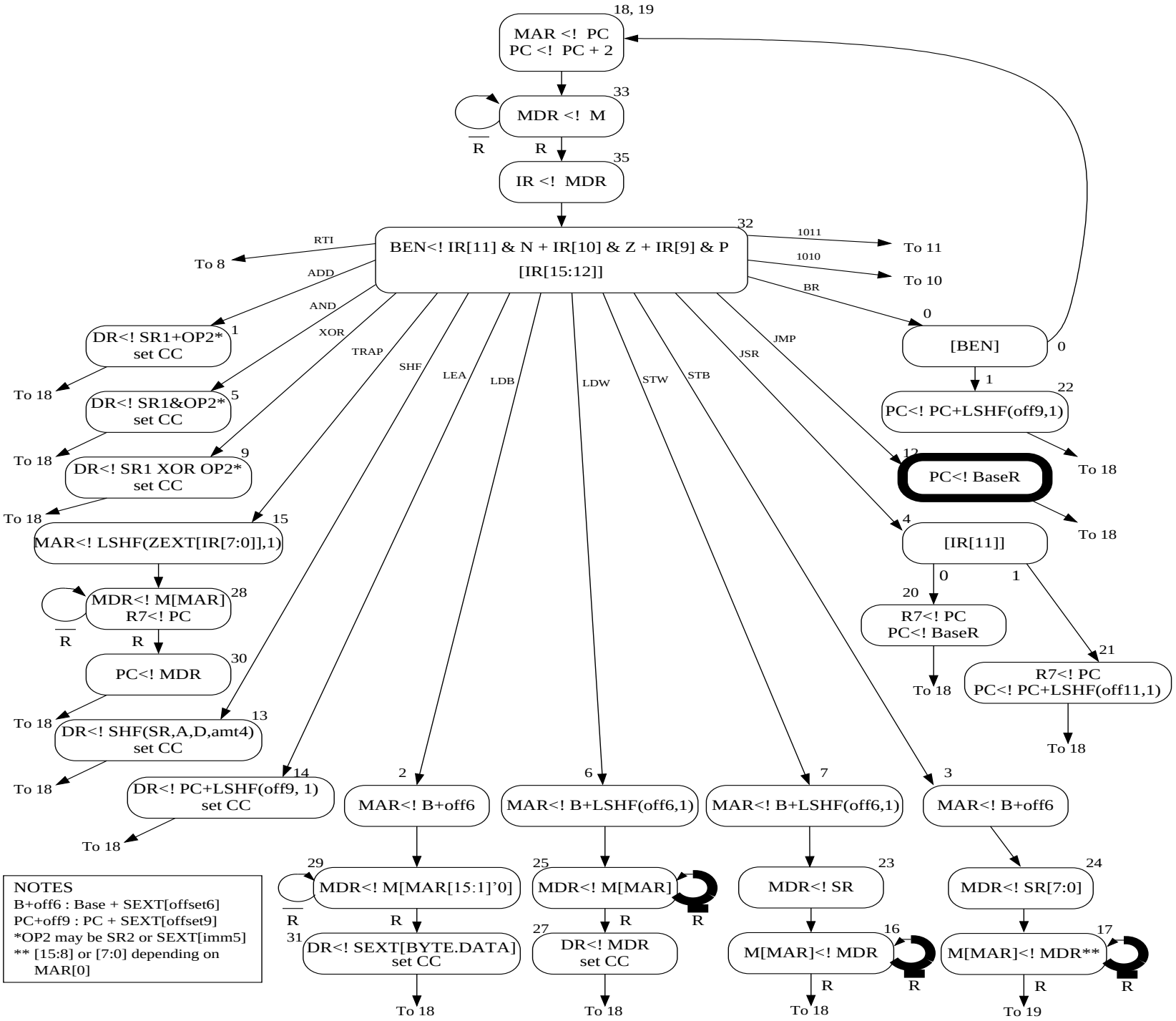
An LC-3b State Machine

- Patt and Patel, App C, Figure C.2

- Each state must be uniquely specified
  - Done by means of *state variables*

- 31 distinct states in this LC-3b state machine
  - Encoded with 6 state variables

- Examples
  - State 18,19 correspond to the beginning of the instruction processing cycle
  - Fetch phase: state 18, 19 → state 33 → state 35
  - Decode phase: state 32


18, 19
MAR <! PC
PC <! PC + 2
33
MDR <! M
R
R
35
IR <! MDR
32
BEN<! IR[11] & N + IR[10] & Z + IR[9] & P
[IR[15:12]]
RTI
To 8
ADD
AND
XOR
TRAP
SHF
LEA
LDB
LDW
STW
STB
JSR
JMP
1011
To 11
1010
To 10
BR
1
DR<! SR1+OP2*
set CC
To 18
5
DR<! SR1&OP2*
set CC
To 18
9
DR<! SR1 XOR OP2*
set CC
To 18
15
MAR<! LSHF(ZEXT[IR[7:0]],1)
28
MDR<! M[MAR]
R7<! PC
R
R
30
PC<! MDR
To 18
13
DR<! SHF(SR,A,D,amt4)
set CC
To 18
14
DR<! PC+LSHF(off9, 1)
set CC
To 18
0
[BEN]
0
1
22
PC<! PC+LSHF(off9,1)
To 18
12
PC<! BaseR
To 18
4
[IR[11]]
0
1
20
R7<! PC
PC<! BaseR
To 18
21
R7<! PC
PC<! PC+LSHF(off11,1)
To 18
2
MAR<! B+off6
29
MDR<! M[MAR[15:1]'0]
R
R
31
DR<! SEXT[BYTE.DATA]
set CC
To 18
6
MAR<! B+LSHF(off6,1)
25
MDR<! M[MAR]
R
R
27
DR<! MDR
set CC
To 18
7
MAR<! B+LSHF(off6,1)
23
MDR<! SR
16
M[MAR]<! MDR
R
R
To 18
3
MAR<! B+off6
24
MDR<! SR[7:0]
17
M[MAR]<! MDR**
R
R
To 19
NOTES
B+off6 : Base + SEXT[offset6]
PC+off9 : PC + SEXT[offset9]
*OP2 may be SR2 or SEXT[imm5]
** [15:8] or [7:0] depending on
MAR[0]

LC-3b State Machine: Some Questions

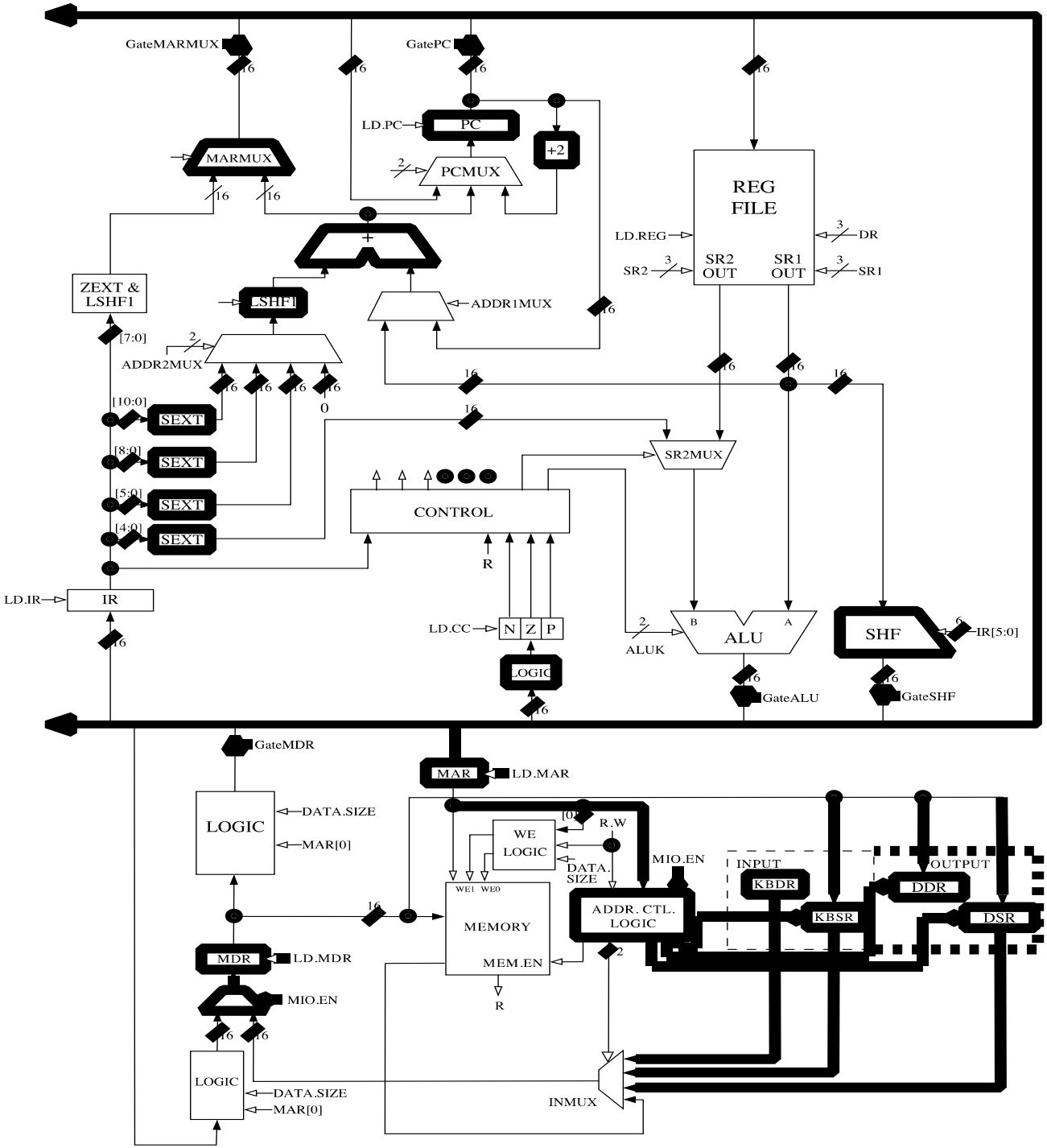
- How many cycles does the fastest instruction take?
- How many cycles does the slowest instruction take?
- Why does the BR take as long as it takes in the FSM?
- What determines the clock cycle?
- Is this a Mealy machine or a Moore machine?

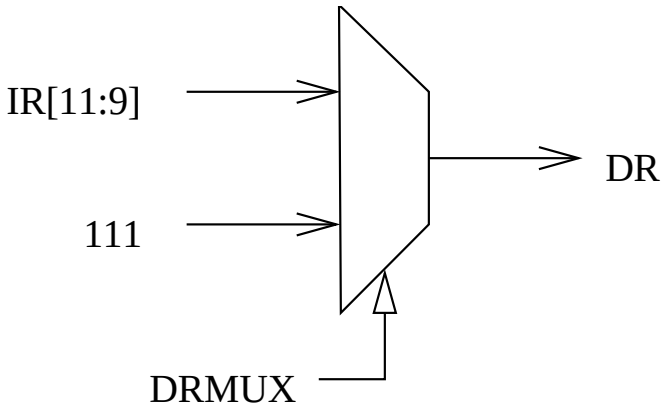
LC-3b Datapath

- Patt and Patel, App C, Figure C.3

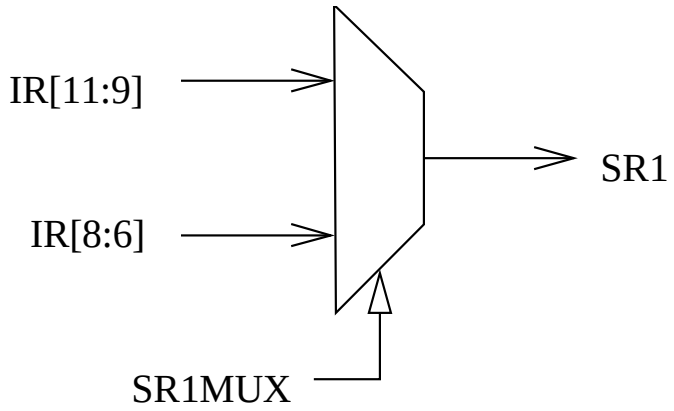
- Single-bus datapath design
  - At any point only one value can be "gated" on the bus (i.e., can be driving the bus)
  - Advantage: Low hardware cost: one bus
  - Disadvantage: Reduced concurrency – if instruction needs the bus twice for two different things, these need to happen in different states

- Control signals (26 of them) determine what happens in the datapath in one clock cycle
  - Patt and Patel, App C, Table C.1

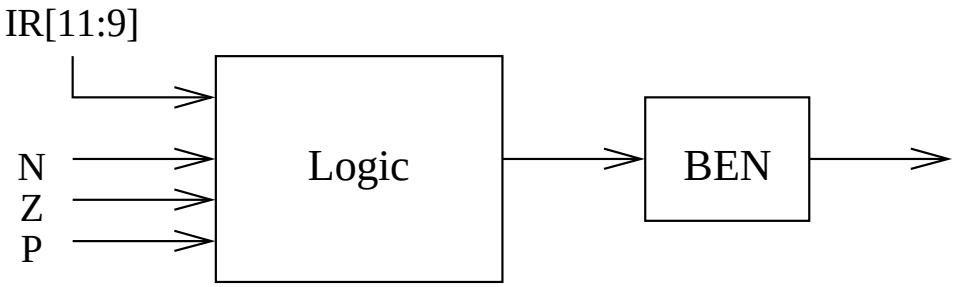

GateMARMUX
GatePC
LD.PC
PC
+2
MARMUX
PCMUX
REG FILE
LD.REG
DR
SR2
SR2 OUT
SR1 OUT
SR1
ZEXT & LSHF1
LSHF1
ADDR1MUX
ADDR2MUX
[7:0]
[10:0]
[8:0]
[5:0]
[4:0]
SEXT
0
SR2MUX
CONTROL
R
LD.IR
IR
LD.CC
N
Z
P
LOGIC
ALUK
B
A
ALU
SHF
IR[5:0]
GateALU
GateSHF
GateMDR
MAR
LD.MAR
LOGIC
DATA.SIZE
MAR[0]
WE LOGIC
R.W
DATA. SIZE
MIO.EN
INPUT
OUTPUT
KBDR
DDR
KBSR
DSR
WE1 WE0
MEMORY
MEM.EN
ADDR. CTL. LOGIC
MDR
LD.MDR
MIO.EN
INMUX



(a)



(b)



(c)

Signal Name	Signal Values	
LD.MAR/1:	NO, LOAD	
LD.MDR/1:	NO, LOAD	
LD.IR/1:	NO, LOAD	
LD.BEN/1:	NO, LOAD	
LD.REG/1:	NO, LOAD	
LD.CC/1:	NO, LOAD	
LD.PC/1:	NO, LOAD	
GatePC/1:	NO, YES	
GateMDR/1:	NO, YES	
GateALU/1:	NO, YES	
GateMARMUX/1:	NO, YES	
GateSHF/1:	NO, YES	
PCMUX/2:	PC+2	;select pc+2
	BUS	;select value from bus
	ADDER	;select output of address adder
DRMUX/1:	11.9	;destination IR[11:9]
	R7	;destination R7
SR1MUX/1:	11.9	;source IR[11:9]
	8.6	;source IR[8:6]
ADDR1MUX/1:	PC, BaseR	
ADDR2MUX/2:	ZERO	;select the value zero
	offset6	;select SEXT[IR[5:0]]
	PCoffset9	;select SEXT[IR[8:0]]
	PCoffset11	;select SEXT[IR[10:0]]
MARMUX/1:	7.0	;select LSHF(ZEXT[IR[7:0]],1)
	ADDER	;select output of address adder
ALUK/2:	ADD, AND, XOR, PASSA	
MIO.EN/1:	NO, YES	
R.W/1:	RD, WR	
DATA.SIZE/1:	BYTE, WORD	
LSHF1/1:	NO, YES	

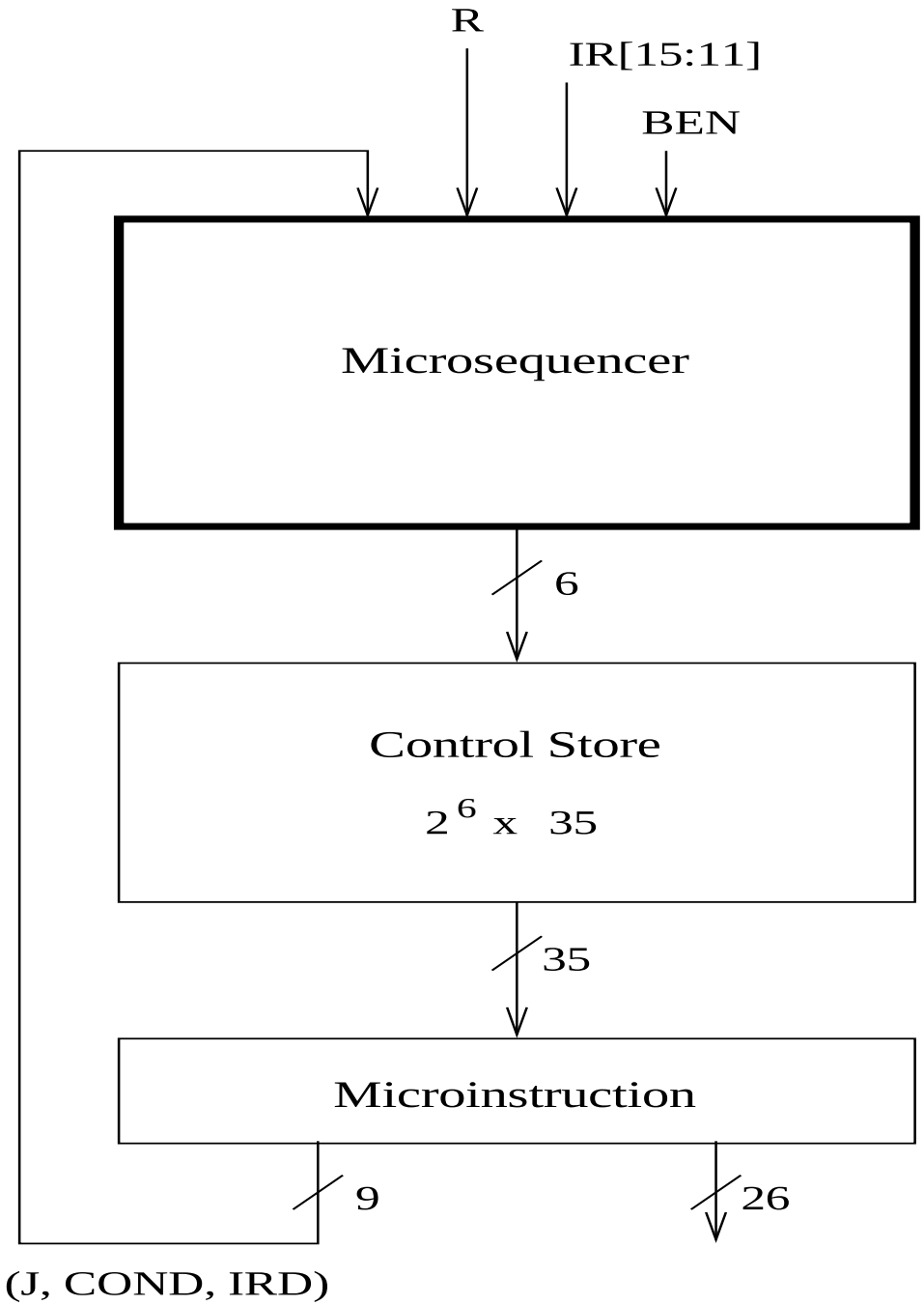
Table C.1: Data path control signals

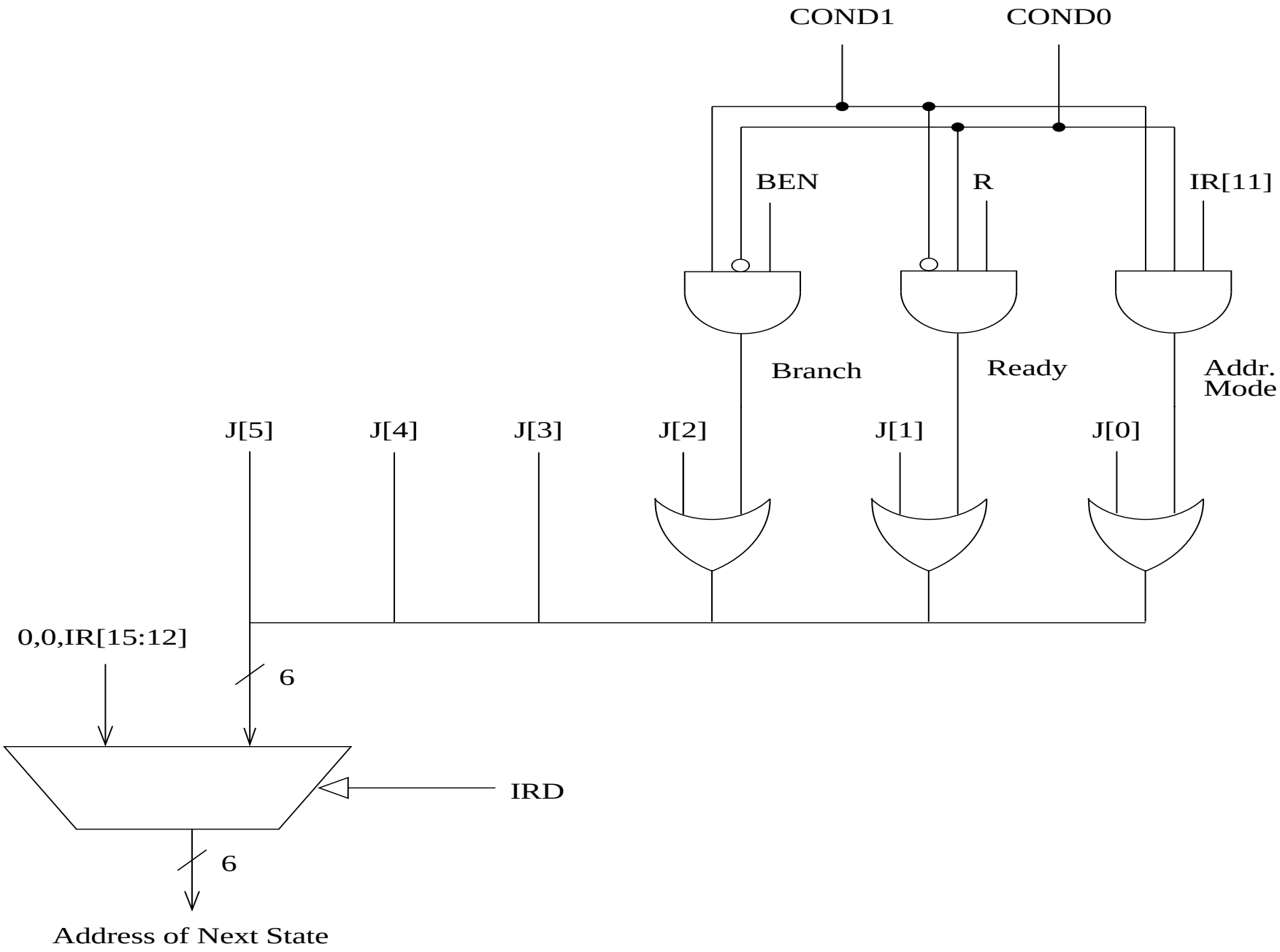
LC-3b Datapath: Some Questions

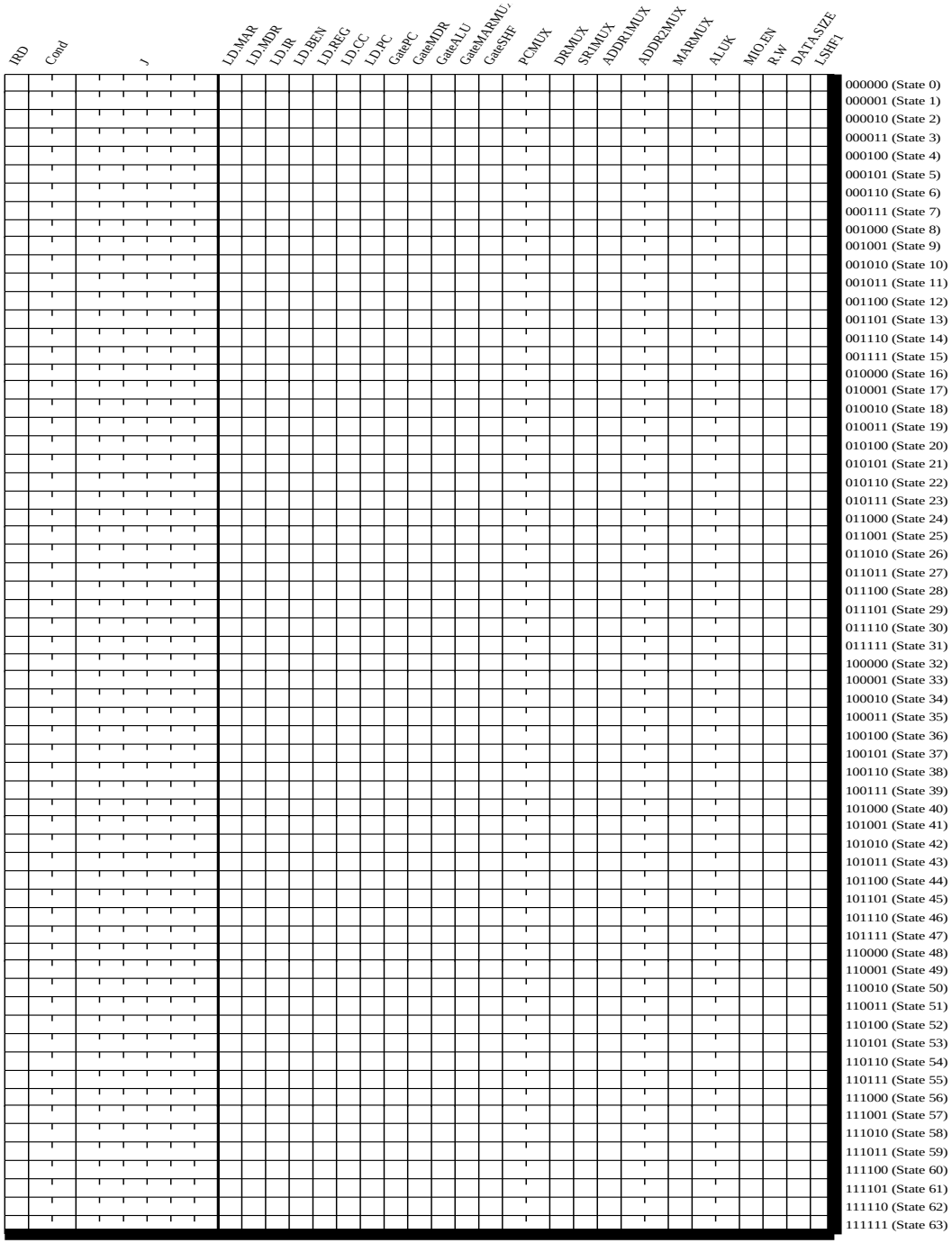
- How does instruction fetch happen in this datapath according to the state machine?
- What is the difference between gating and loading?
- Is this the smallest hardware you can design?

LC-3b Microprogrammed Control Structure

- Patt and Patel, App C, Figure C.4
- Three components:
  - Microinstruction, control store, microsequencer
- Microinstruction: control signals that control the datapath (26 of them) and determine the next state (9 of them)
- Each microinstruction is stored in a unique location in the control store (a special memory structure)
- Unique location: address of the state corresponding to the microinstruction
  - Remember each state corresponds to one microinstruction
- Microsequencer determines the address of the next microinstruction (i.e., next state)


R
IR[15:11]
BEN
Microsequencer
6
Control Store
2 6 x 35
35
Microinstruction
9
26
(J, COND, IRD)


COND1
COND0
BEN
R
IR[11]
Branch
Ready
Addr.
Mode
J[5]
J[4]
J[3]
J[2]
J[1]
J[0]
0,0,IR[15:12]
6
IRD
6
Address of Next State

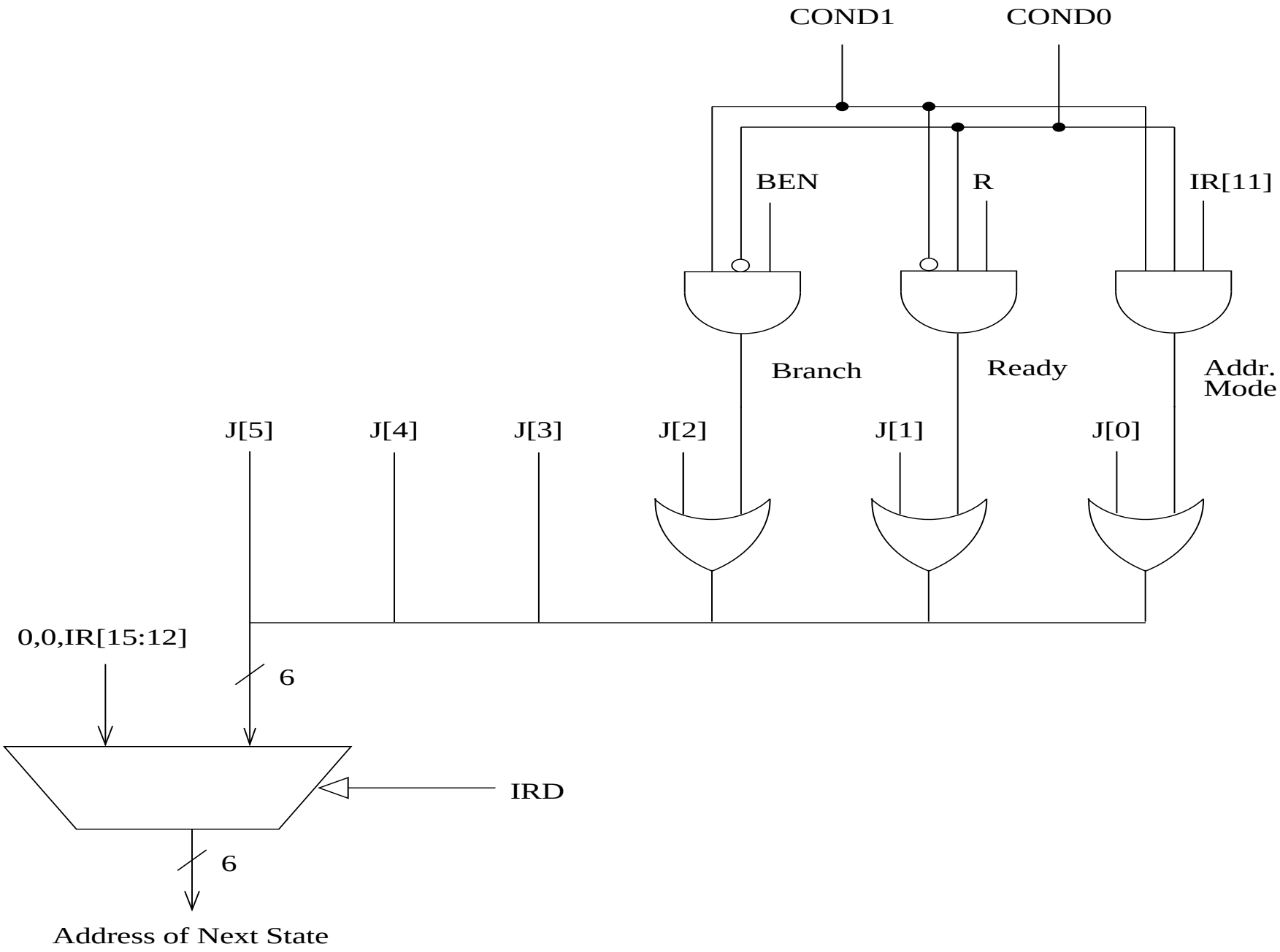

IRD
Cond
J
LD.MAR
LD.MDR
LD.IR
LD.BEN
LD.REG
LD.CC
LD.PC
GatePC
GateMDR
GateALU
GateMARMUX
GateSHF
PCMUX
DRMUX
SR1MUX
ADDR1MUX
ADDR2MUX
MARMUX
ALUK
MIO.EN
R.W
DATA.SIZE
LSHF1
000000 (State 0)
000001 (State 1)
000010 (State 2)
000011 (State 3)
000100 (State 4)
000101 (State 5)
000110 (State 6)
000111 (State 7)
001000 (State 8)
001001 (State 9)
001010 (State 10)
001011 (State 11)
001100 (State 12)
001101 (State 13)
001110 (State 14)
001111 (State 15)
010000 (State 16)
010001 (State 17)
010010 (State 18)
010011 (State 19)
010100 (State 20)
010101 (State 21)
010110 (State 22)
010111 (State 23)
011000 (State 24)
011001 (State 25)
011010 (State 26)
011011 (State 27)
011100 (State 28)
011101 (State 29)
011110 (State 30)
011111 (State 31)
100000 (State 32)
100001 (State 33)
100010 (State 34)
100011 (State 35)
100100 (State 36)
100101 (State 37)
100110 (State 38)
100111 (State 39)
101000 (State 40)
101001 (State 41)
101010 (State 42)
101011 (State 43)
101100 (State 44)
101101 (State 45)
101110 (State 46)
101111 (State 47)
110000 (State 48)
110001 (State 49)
110010 (State 50)
110011 (State 51)
110100 (State 52)
110101 (State 53)
110110 (State 54)
110111 (State 55)
111000 (State 56)
111001 (State 57)
111010 (State 58)
111011 (State 59)
111100 (State 60)
111101 (State 61)
111110 (State 62)
111111 (State 63)

LC-3b Microsequencer

- Patt and Patel, App C, Figure C.5
- The purpose of the microsequencer is to determine the address of the next microinstruction (i.e., next state)
- Next address depends on 9 control signals

Signal Name	Signal Values	
J/6:		
COND/2:	$COND_0$	;Unconditional
	$COND_1$	;Memory Ready
	$COND_2$	;Branch
	$COND_3$	;Addressing Mode
IRD/1:	NO, YES	

Table C.2: Microsequencer control signals


COND1
COND0
BEN
R
IR[11]
Branch
Ready
Addr.
Mode
J[5]
J[4]
J[3]
J[2]
J[1]
J[0]
0,0,IR[15:12]
6
IRD
6
Address of Next State

The Microsequencer: Some Questions

- When is the IRD signal asserted?
- What happens if an illegal instruction is decoded?
- What are condition (COND) bits for?
- How is variable latency memory handled?
- How do you do the state encoding?
  - Minimize number of state variables
  - Start with the 16-way branch
  - Then determine constraint tables and states dependent on COND

An Exercise in Microprogramming

Handouts

- 7 pages of Microprogrammed LC-3b design
- http://www.ece.cmu.edu/~ece447/s13/doku.php?id=manuals
- http://www.ece.cmu.edu/~ece447/s13/lib/exe/fetch.php?media=lc3b-figures.pdf

A Simple LC-3b Control and Datapath

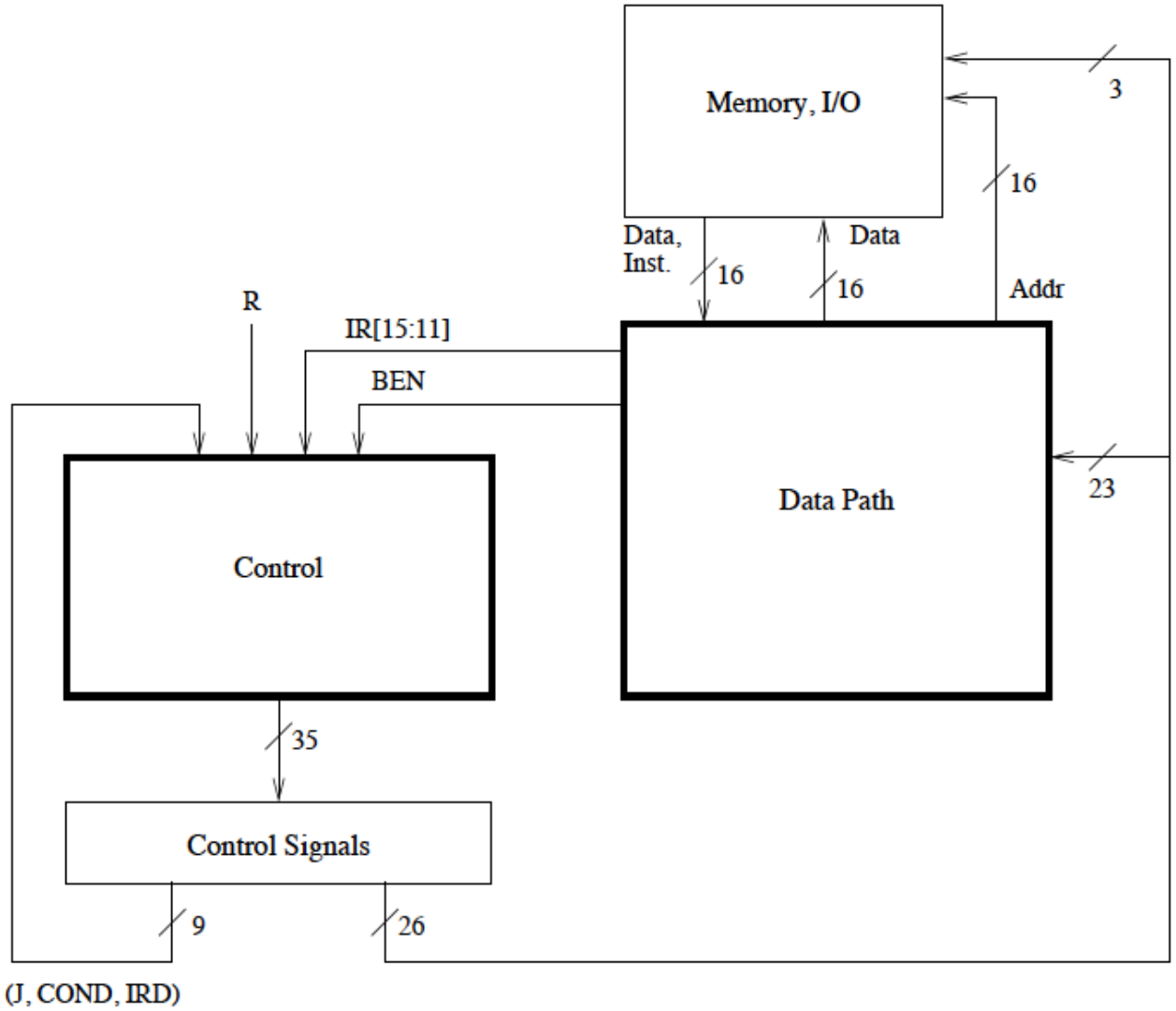
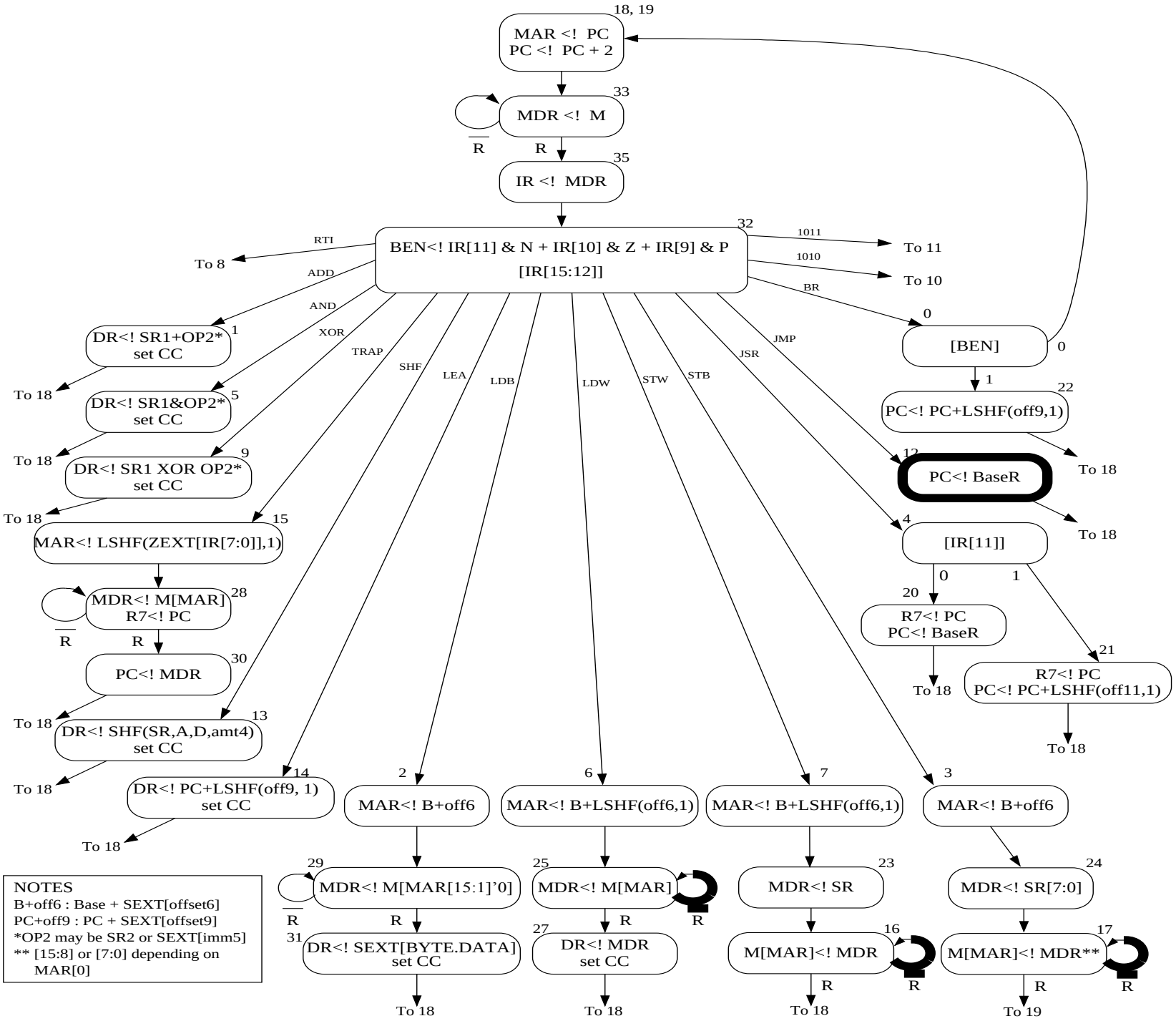
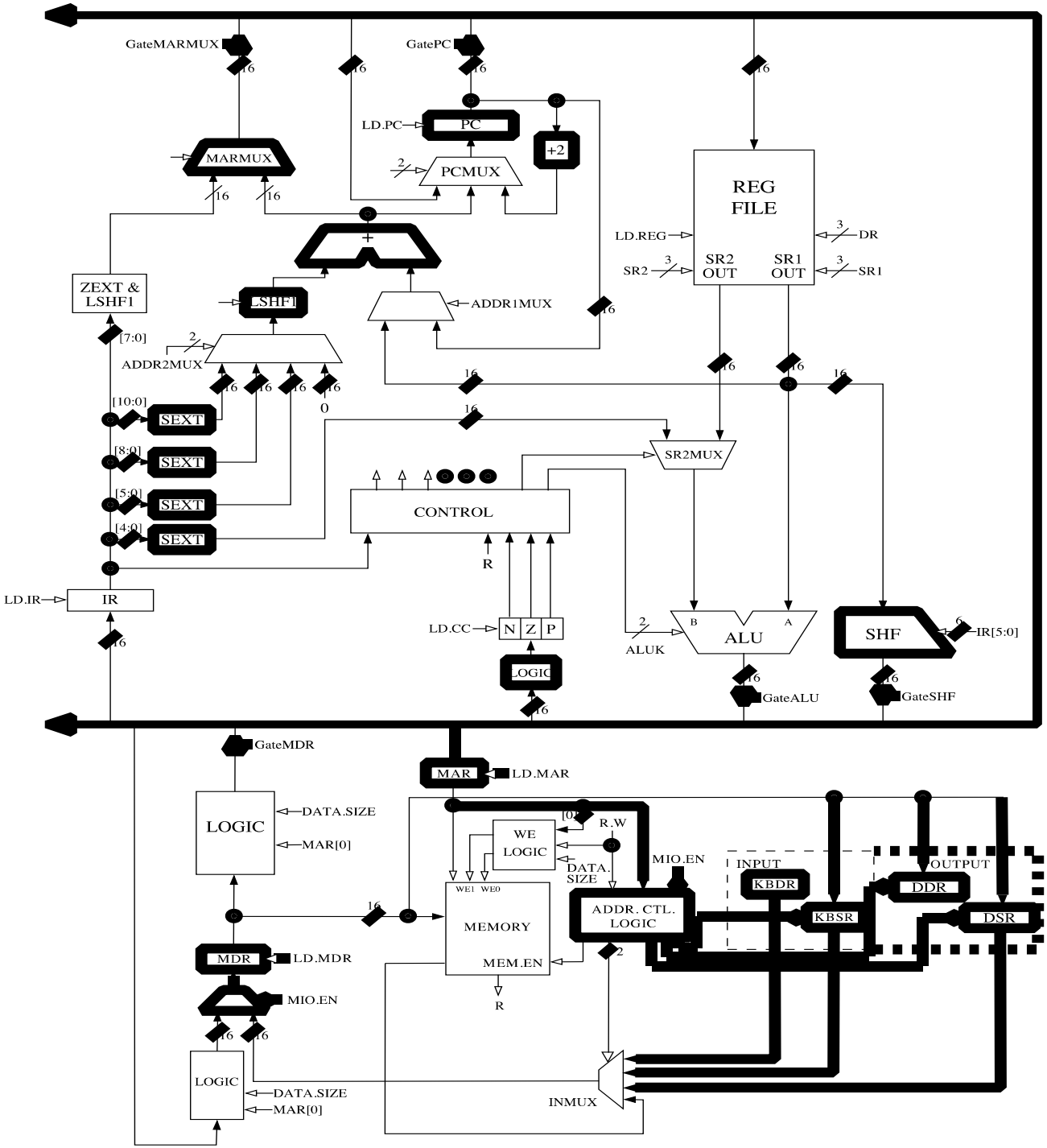
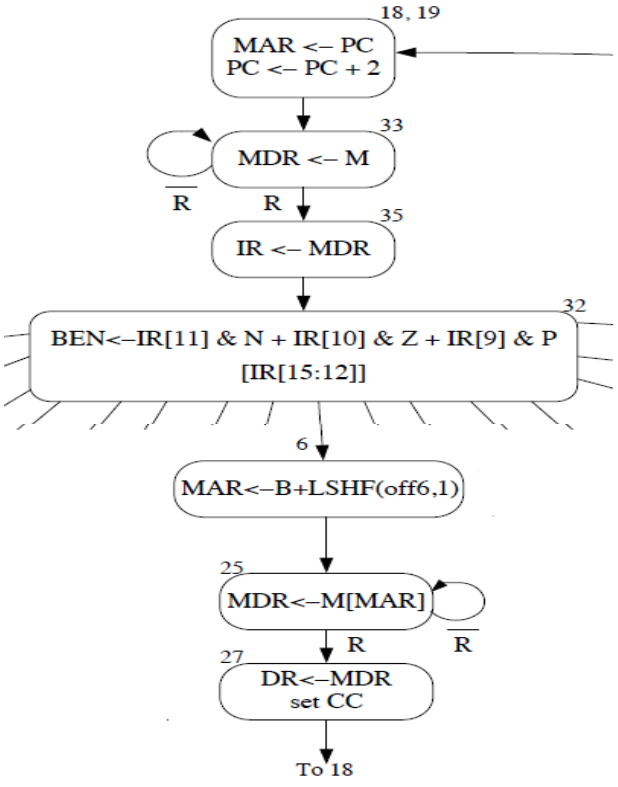


Figure C.1: Microarchitecture of the LC-3b, major components

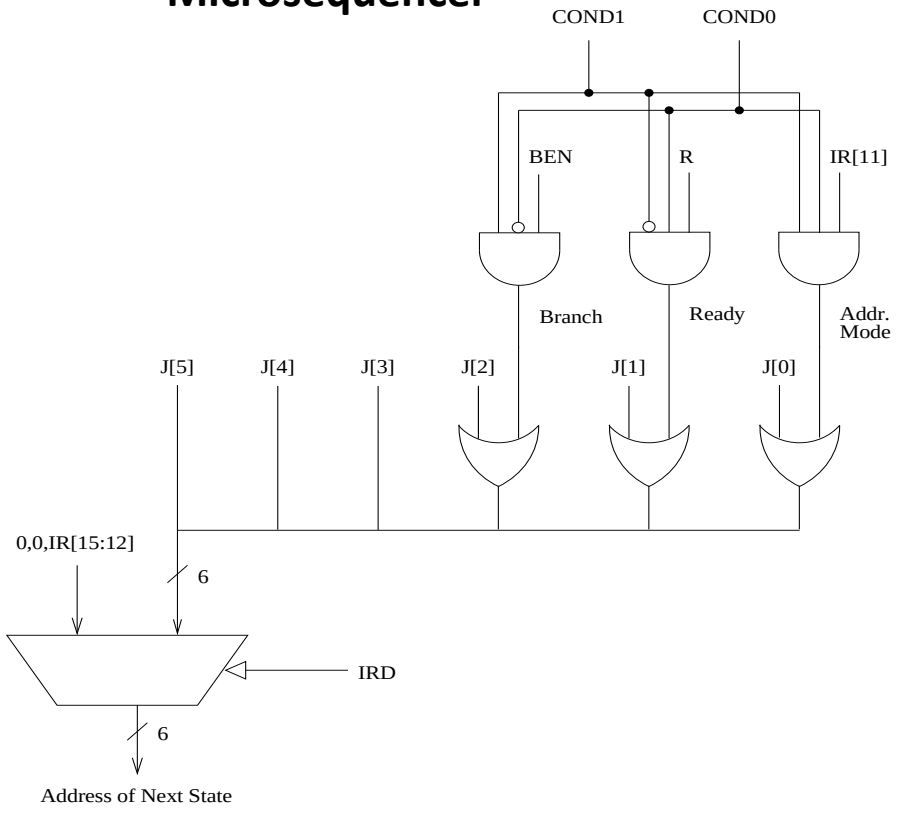

18, 19
MAR <! PC
PC <! PC + 2
33
MDR <! M
R
R
35
IR <! MDR
32
BEN<! IR[11] & N + IR[10] & Z + IR[9] & P
[IR[15:12]]
RTI
To 8
ADD
AND
XOR
TRAP
SHF
LEA
LDB
LDW
STW
STB
JSR
JMP
1011
To 11
1010
To 10
BR
1
DR<! SR1+OP2*
set CC
To 18
5
DR<! SR1&OP2*
set CC
To 18
9
DR<! SR1 XOR OP2*
set CC
To 18
15
MAR<! LSHF(ZEXT[IR[7:0]],1)
28
MDR<! M[MAR]
R7<! PC
R
R
30
PC<! MDR
To 18
13
DR<! SHF(SR,A,D,amt4)
set CC
To 18
14
DR<! PC+LSHF(off9, 1)
set CC
To 18
0
[BEN]
0
1
22
PC<! PC+LSHF(off9,1)
To 18
12
PC<! BaseR
To 18
4
[IR[11]]
0
1
20
R7<! PC
PC<! BaseR
To 18
21
R7<! PC
PC<! PC+LSHF(off11,1)
To 18
2
MAR<! B+off6
29
MDR<! M[MAR[15:1]'0]
R
R
31
DR<! SEXT[BYTE.DATA]
set CC
To 18
6
MAR<! B+LSHF(off6,1)
25
MDR<! M[MAR]
R
R
27
DR<! MDR
set CC
To 18
7
MAR<! B+LSHF(off6,1)
23
MDR<! SR
16
M[MAR]<! MDR
R
R
To 18
3
MAR<! B+off6
24
MDR<! SR[7:0]
17
M[MAR]<! MDR**
R
R
To 19
NOTES
B+off6 : Base + SEXT[offset6]
PC+off9 : PC + SEXT[offset9]
*OP2 may be SR2 or SEXT[imm5]
** [15:8] or [7:0] depending on
MAR[0]


GateMARMUX
GatePC
LD.PC
PC
+2
MARMUX
PCMUX
REG FILE
LD.REG
DR
SR2
SR2 OUT
SR1 OUT
SR1
ZEXT & LSHF1
LSHF1
ADDR1MUX
ADDR2MUX
[7:0]
[10:0]
[8:0]
[5:0]
[4:0]
SEXT
0
SR2MUX
CONTROL
R
LD.IR
IR
LD.CC
N
Z
P
LOGIC
ALUK
B
A
ALU
SHF
IR[5:0]
GateALU
GateSHF
GateMDR
MAR
LD.MAR
LOGIC
DATA.SIZE
MAR[0]
WE LOGIC
R.W
DATA. SIZE
MIO.EN
INPUT
OUTPUT
KBDR
DDR
KBSR
DSR
WE1 WE0
MEMORY
MEM.EN
ADDR. CTL. LOGIC
MDR
LD.MDR
MIO.EN
INMUX

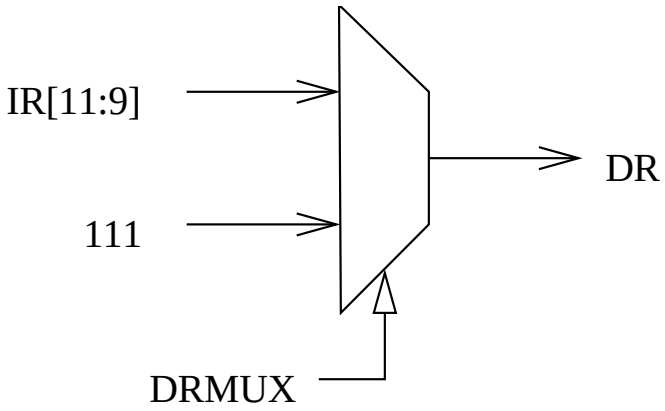
State Machine for LDW



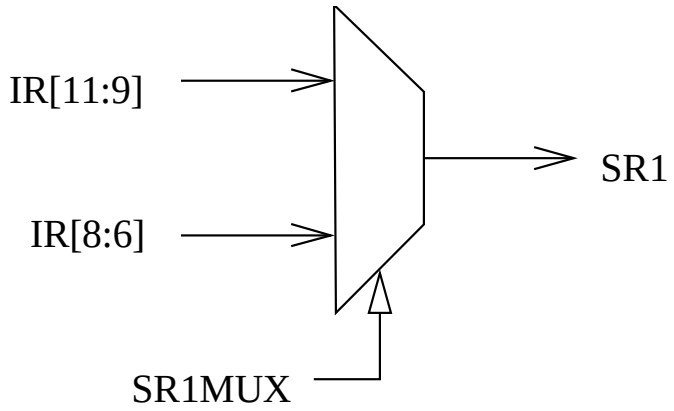
Microsequencer



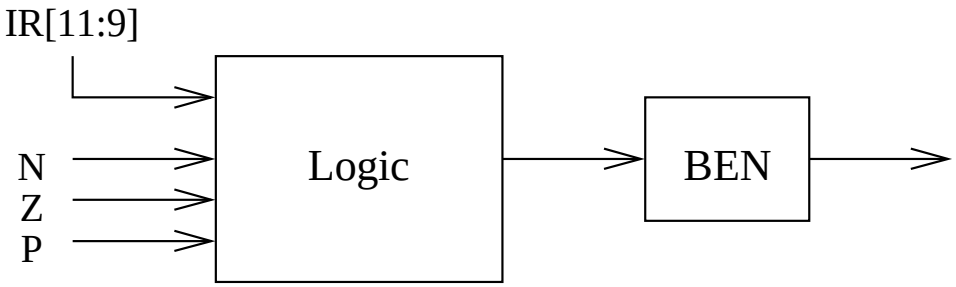
	IRD	Cond	J	LD.MAR	LD.MDR	LD.IR	LD.BEN	LD.REG	LD.CC	LD.PC	GatePC	GateMDR	GateALU	GateMARMUX	GateSHF	PCMUX	DRMUX	SR1MUX	ADDR1MUX	ADDR2MUX	MARMUX	ALUK	MIO.EN	R.W	DATA.SIZE	LSHF1
State 18 (010010)																										
State 33 (100001)																										
State 35 (100011)																										
State 32 (100000)																										
State 6 (000110)																										
State 25 (011001)																										
State 27 (011011)																										



(a)



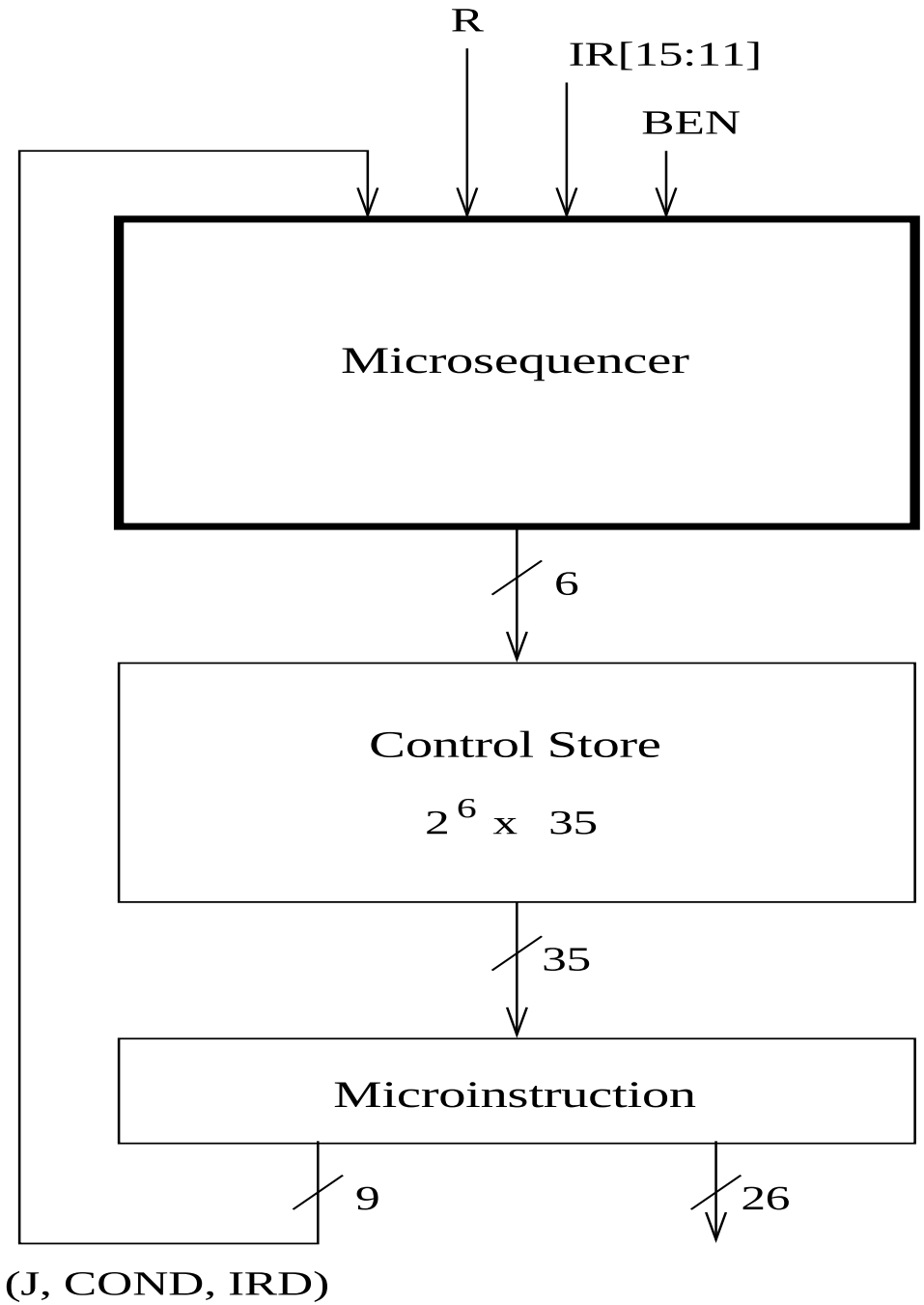
(b)

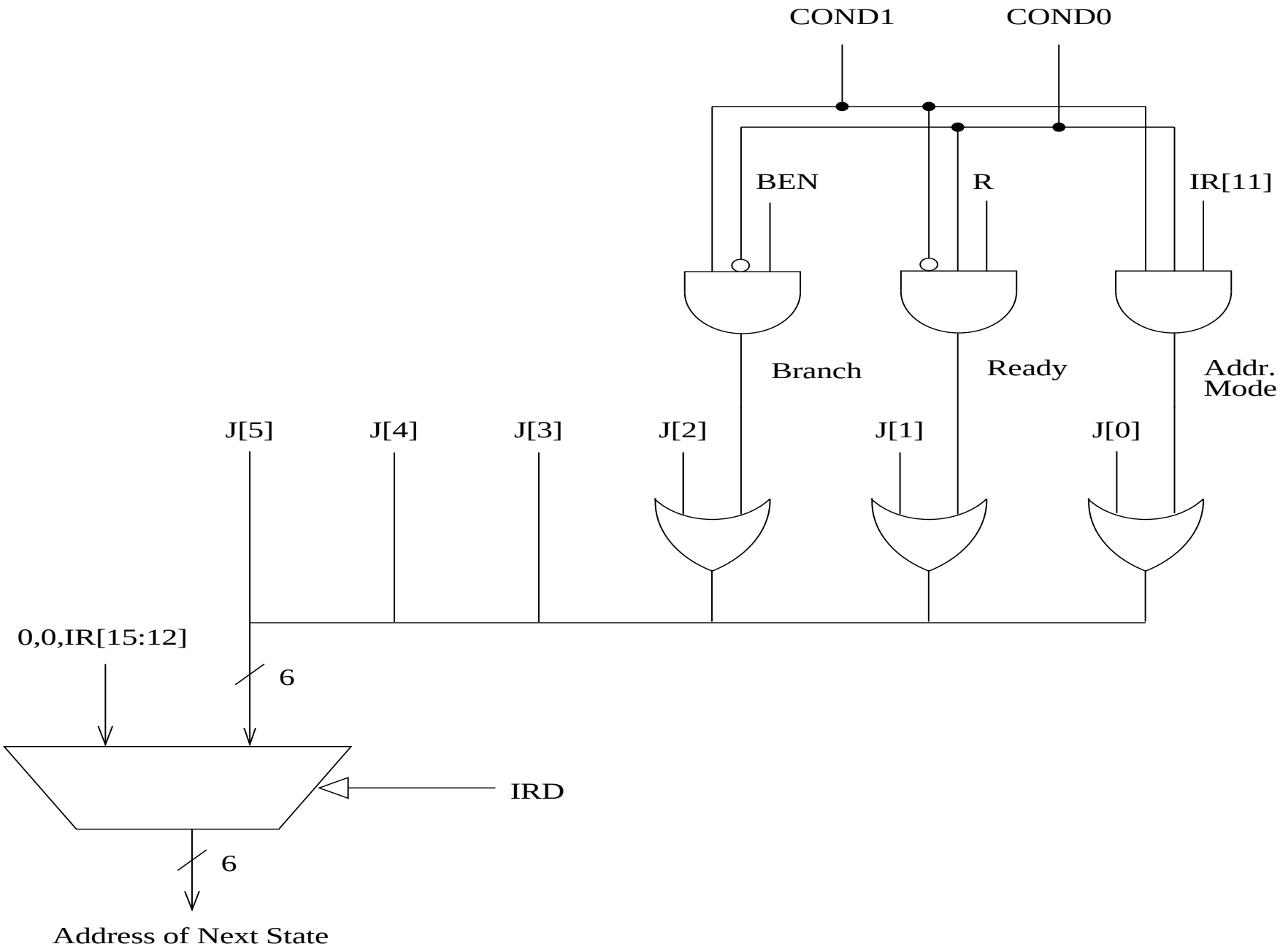


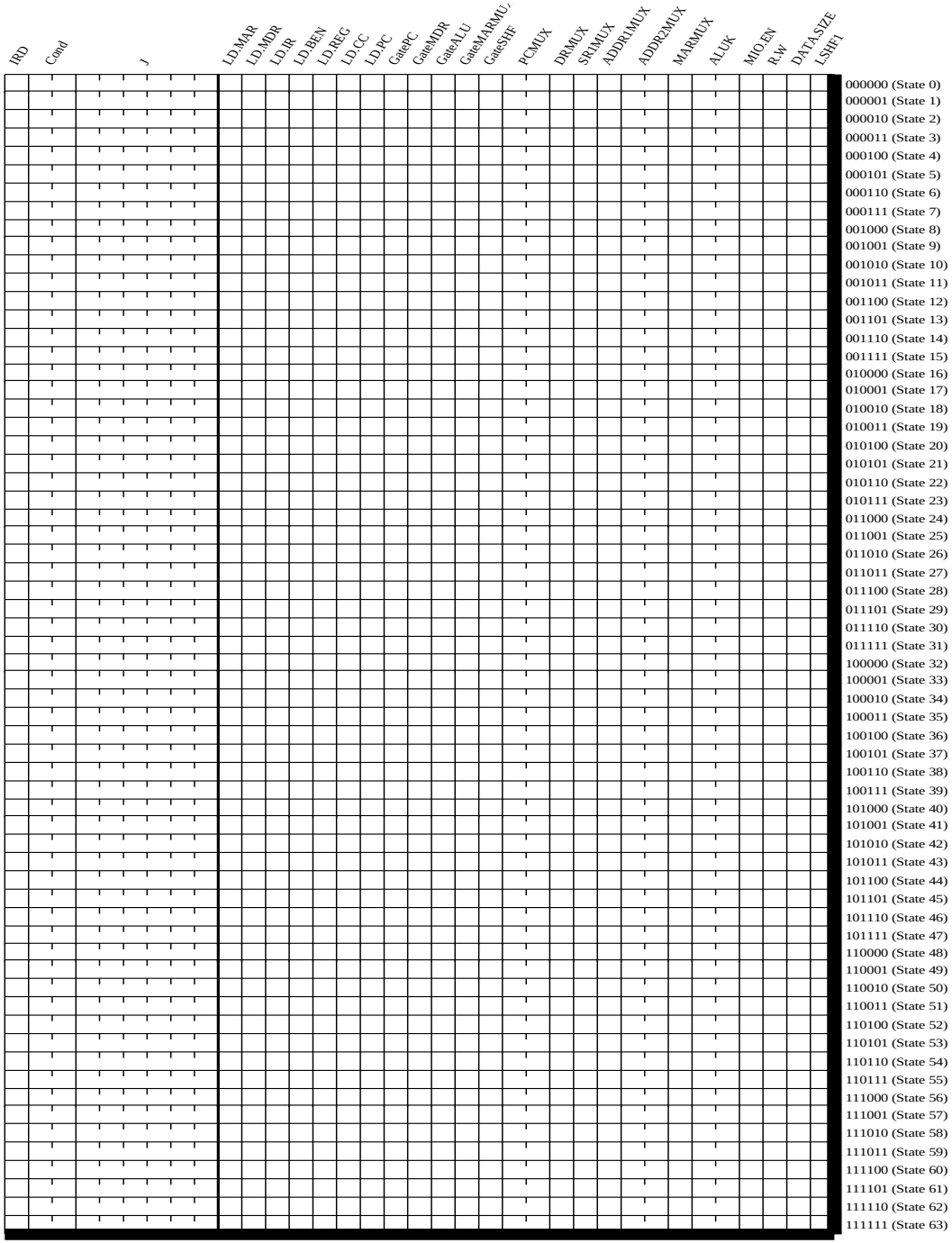
(c)

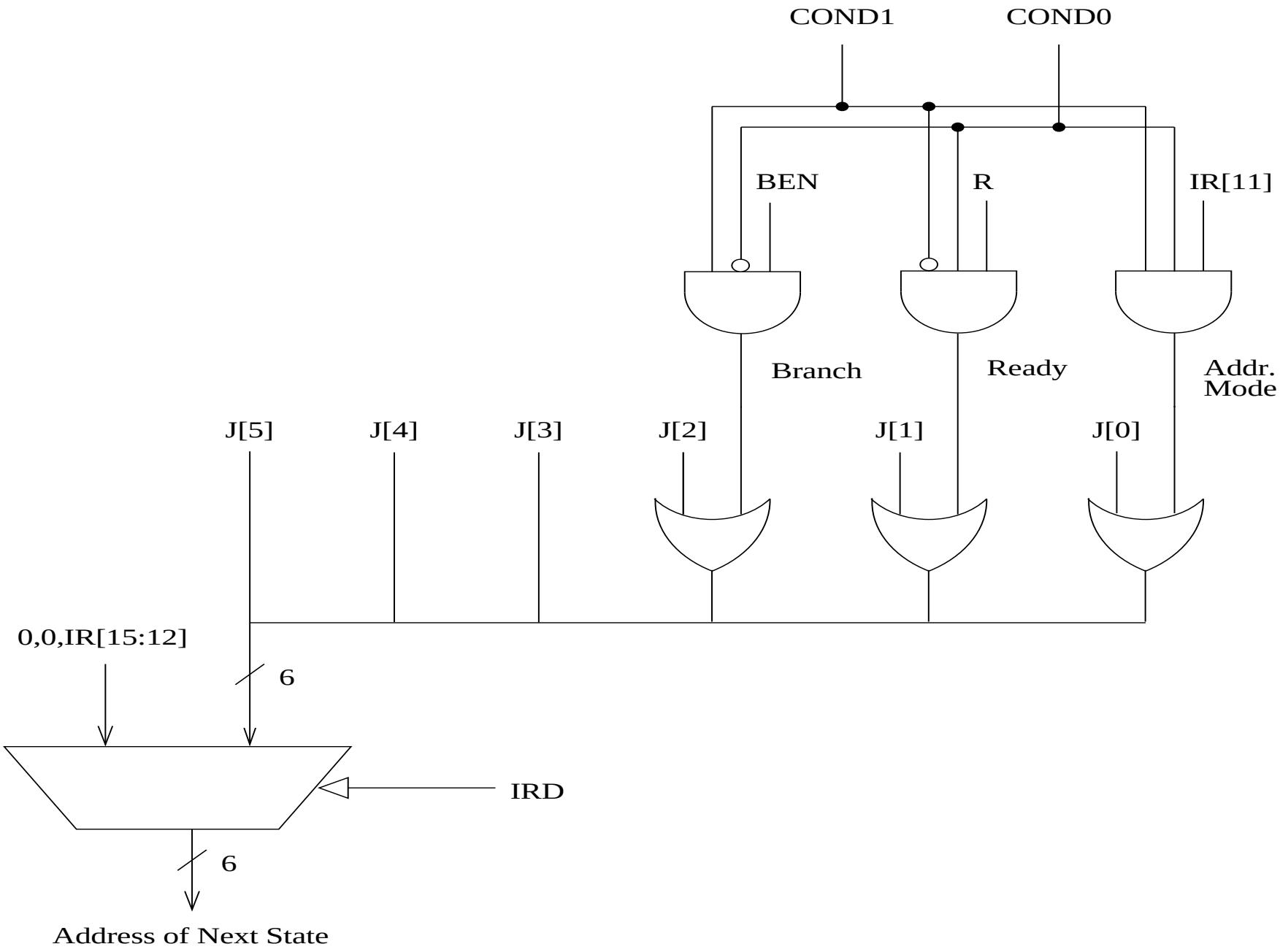
Signal Name	Signal Values	
LD.MAR/1:	NO, LOAD	
LD.MDR/1:	NO, LOAD	
LD.IR/1:	NO, LOAD	
LD.BEN/1:	NO, LOAD	
LD.REG/1:	NO, LOAD	
LD.CC/1:	NO, LOAD	
LD.PC/1:	NO, LOAD	
GatePC/1:	NO, YES	
GateMDR/1:	NO, YES	
GateALU/1:	NO, YES	
GateMARMUX/1:	NO, YES	
GateSHF/1:	NO, YES	
PCMUX/2:	PC+2	;select pc+2
	BUS	;select value from bus
	ADDER	;select output of address adder
DRMUX/1:	11.9	;destination IR[11:9]
	R7	;destination R7
SR1MUX/1:	11.9	;source IR[11:9]
	8.6	;source IR[8:6]
ADDR1MUX/1:	PC, BaseR	
ADDR2MUX/2:	ZERO	;select the value zero
	offset6	;select SEXT[IR[5:0]]
	PCoffset9	;select SEXT[IR[8:0]]
	PCoffset11	;select SEXT[IR[10:0]]
MARMUX/1:	7.0	;select LSHF(ZEXT[IR[7:0]],1)
	ADDER	;select output of address adder
ALUK/2:	ADD, AND, XOR, PASSA	
MIO.EN/1:	NO, YES	
R.W/1:	RD, WR	
DATA.SIZE/1:	BYTE, WORD	
LSHF1/1:	NO, YES	

Table C.1: Data path control signals


R
IR[15:11]
BEN
Microsequencer
6
Control Store
2 6 x 35
35
Microinstruction
9
26
(J, COND, IRD)


COND1
COND0
BEN
R
IR[11]
Branch
Ready
Addr.
Mode
J[5]
J[4]
J[3]
J[2]
J[1]
J[0]
0,0,IR[15:12]
6
IRD
6
Address of Next State


IRD
Cond
J
LD.MAR
LD.MDR
LD.IR
LD.BEN
LD.REG
LD.CC
LD.PC
GatePC
GateMDR
GateALU
GateMARMUX
GateSHF
PCMUX
DRMUX
SR1MUX
ADDR1MUX
ADDR2MUX
MARMUX
ALUK
MIO.EN
R.W
DATA.SIZE
LSHF1
000000 (State 0)
000001 (State 1)
000010 (State 2)
000011 (State 3)
000100 (State 4)
000101 (State 5)
000110 (State 6)
000111 (State 7)
001000 (State 8)
001001 (State 9)
001010 (State 10)
001011 (State 11)
001100 (State 12)
001101 (State 13)
001110 (State 14)
001111 (State 15)
010000 (State 16)
010001 (State 17)
010010 (State 18)
010011 (State 19)
010100 (State 20)
010101 (State 21)
010110 (State 22)
010111 (State 23)
011000 (State 24)
011001 (State 25)
011010 (State 26)
011011 (State 27)
011100 (State 28)
011101 (State 29)
011110 (State 30)
011111 (State 31)
100000 (State 32)
100001 (State 33)
100010 (State 34)
100011 (State 35)
100100 (State 36)
100101 (State 37)
100110 (State 38)
100111 (State 39)
101000 (State 40)
101001 (State 41)
101010 (State 42)
101011 (State 43)
101100 (State 44)
101101 (State 45)
101110 (State 46)
101111 (State 47)
110000 (State 48)
110001 (State 49)
110010 (State 50)
110011 (State 51)
110100 (State 52)
110101 (State 53)
110110 (State 54)
110111 (State 55)
111000 (State 56)
111001 (State 57)
111010 (State 58)
111011 (State 59)
111100 (State 60)
111101 (State 61)
111110 (State 62)
111111 (State 63)


COND1
COND0
BEN
R
IR[11]
Branch
Ready
Addr.
Mode
J[5]
J[4]
J[3]
J[2]
J[1]
J[0]
0,0,IR[15:12]
6
IRD
6
Address of Next State

End of the Exercise in Microprogramming

Homework 2

- You will write the microcode for the entire LC-3b as specified in Appendix C