Digital Design & Computer Arch.

Lecture 12: Microarchitecture Fundamentals II

Prof. Onur Mutlu

ETH Zürich
Spring 2021
15 April 2021

Readings

- Last time and today
  - Introduction to microarchitecture and single-cycle microarchitecture
    - H&H, Chapter 7.1-7.3
    - P&P, Appendices A and C
  - Multi-cycle microarchitecture
    - H&H, Chapter 7.4
    - P&P, Appendices A and C

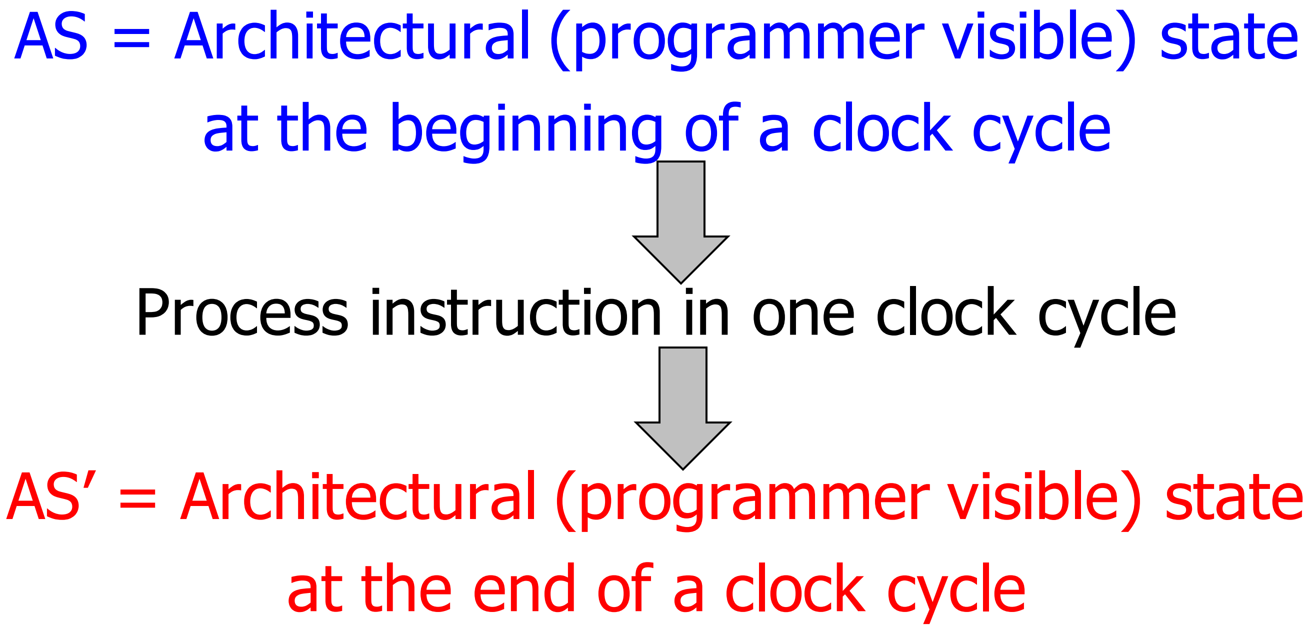
- Tomorrow and next week
  - Pipelining
    - H&H, Chapter 7.5
  - Pipelining Issues
    - H&H, Chapter 7.8.1-7.8.3

Agenda for Today & Next Few Lectures

- Instruction Set Architectures (ISA): LC-3 and MIPS
- Assembly programming: LC-3 and MIPS
- Microarchitecture (principles & single-cycle uarch)
- Multi-cycle microarchitecture
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, …
- Out-of-Order Execution

Recall: 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*

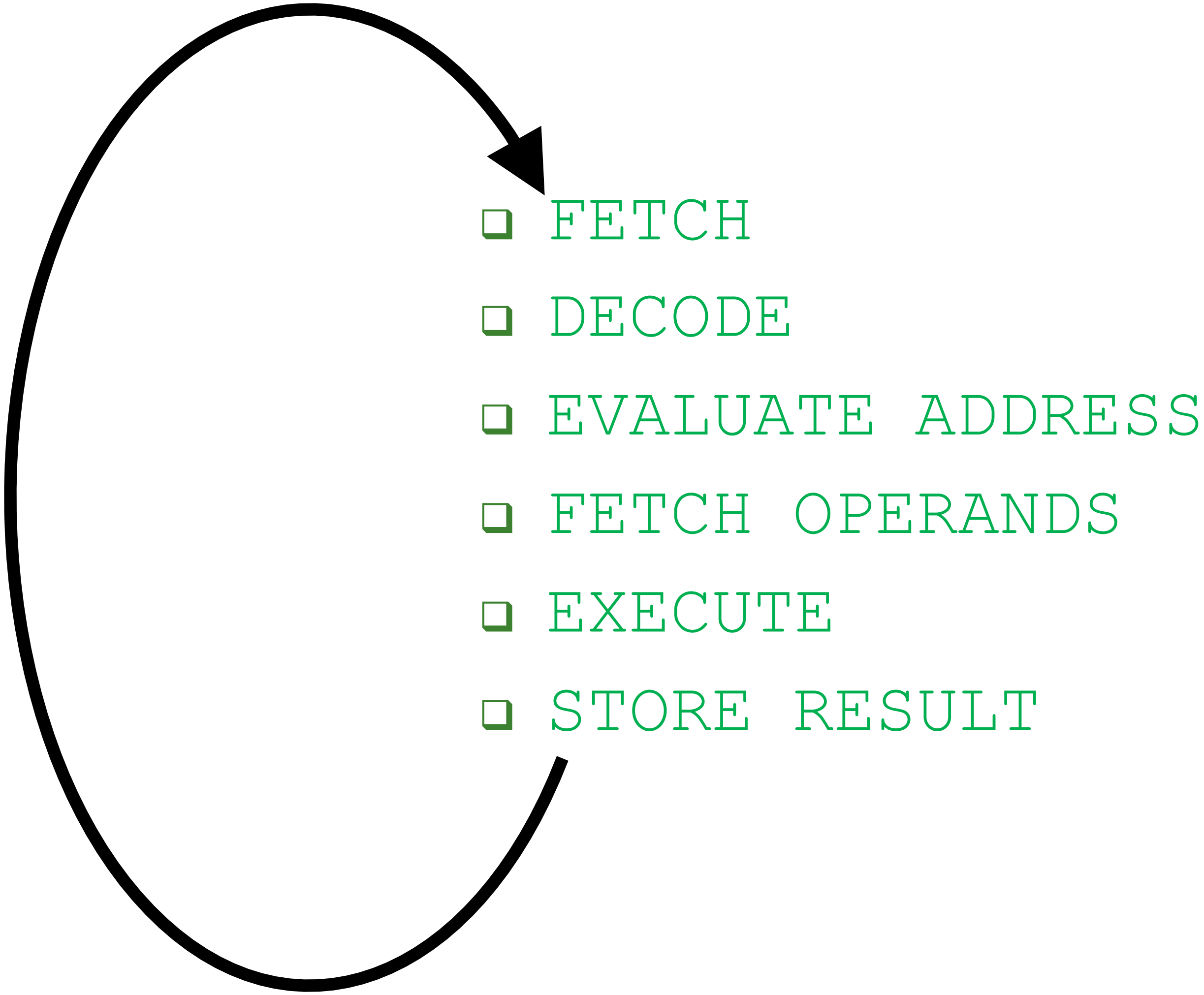


AS = Architectural (programmer visible) state at the beginning of a clock cycle

Process instruction in one clock cycle

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

Recall: The Instruction Processing "Cycle"



- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

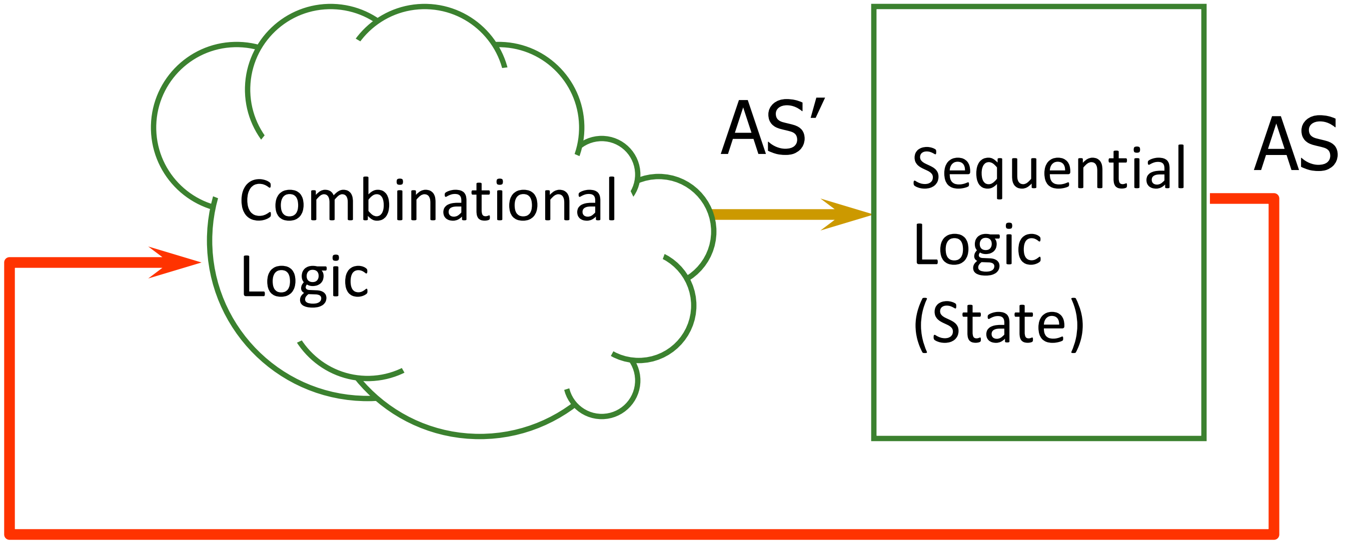
Instruction Processing "Cycle" vs. Machine Clock Cycle

- 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

Recall: Single-Cycle Machine

- Single-cycle machine



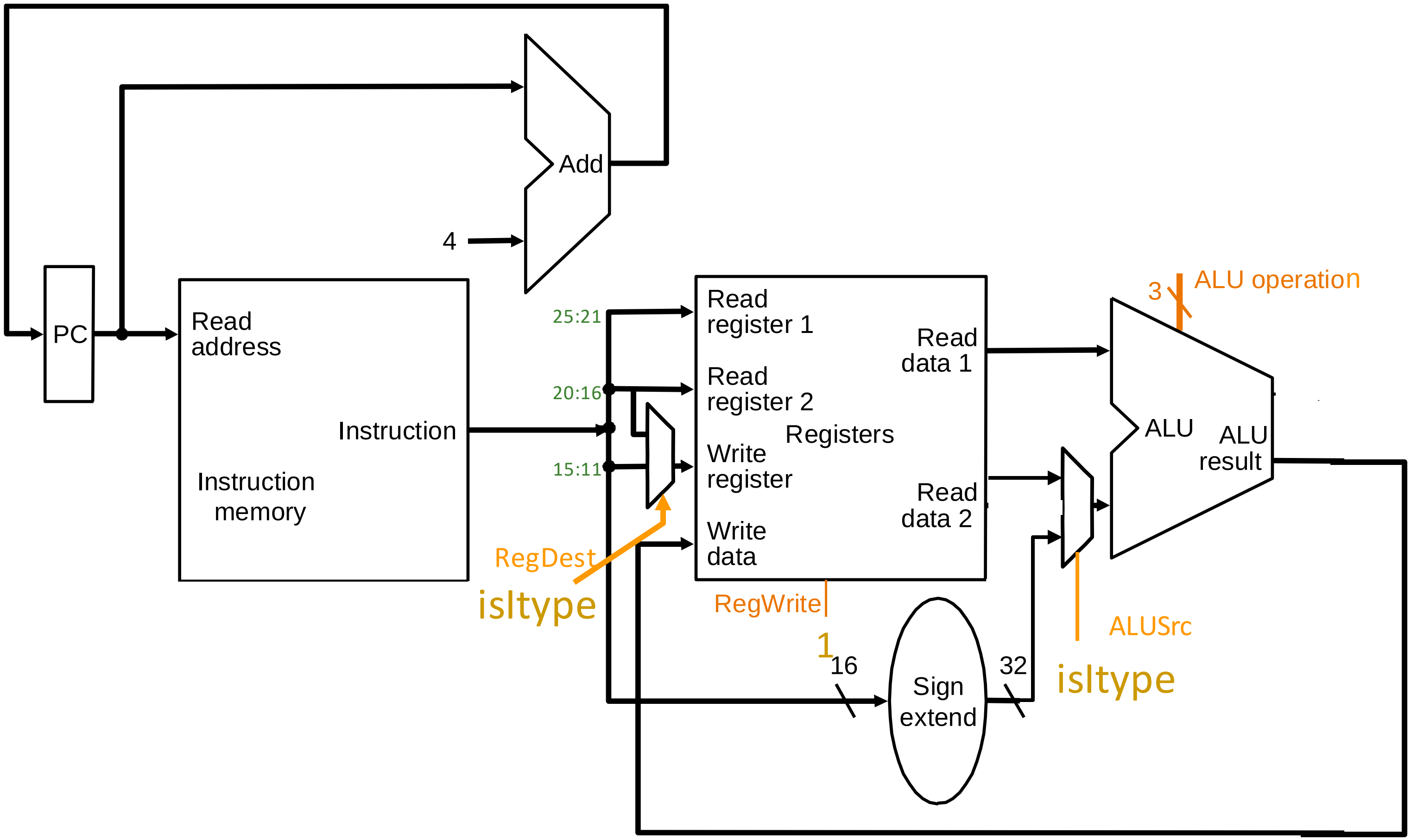
AS: Architectural State

Recall: Datapath and Control Logic

- 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

Single-Cycle Datapath for *Arithmetic and Logical Instructions*

Datapath for R- and I-Type ALU Insts.



IF | ID | EX | MEM | WB

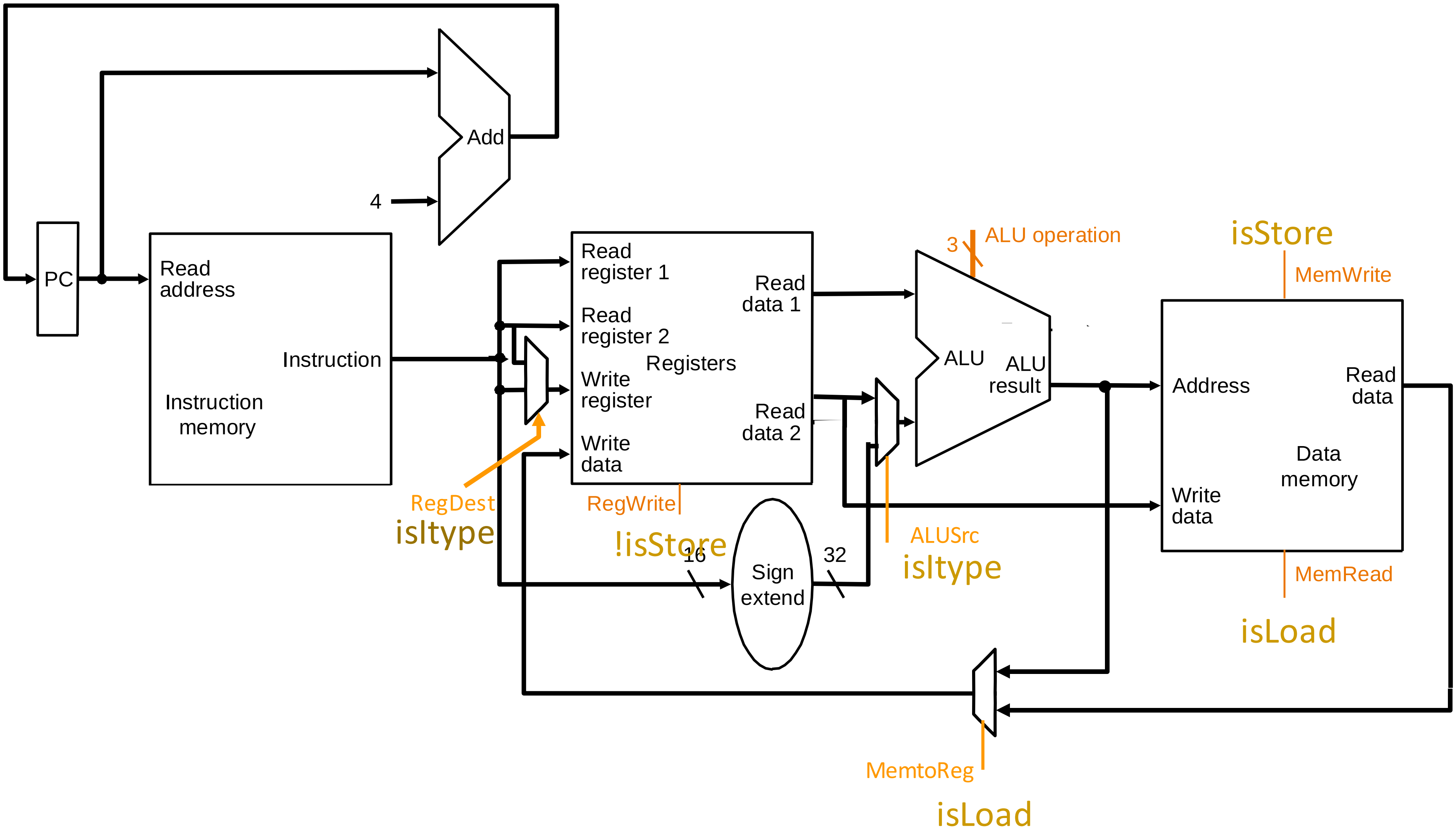
```
if MEM[PC] == ADDI rt rs immediate
    GPR[rt] ← GPR[rs] + sign-extend (immediate)
    PC ← PC + 4
```

Combinational state update logic

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

Single-Cycle Datapath for *Data Movement Instructions*

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*

Jump Instruction

- Unconditional branch or jump

```
j   target
```

j (2)	immediate
6 bits	26 bits

J-Type

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

- Semantics

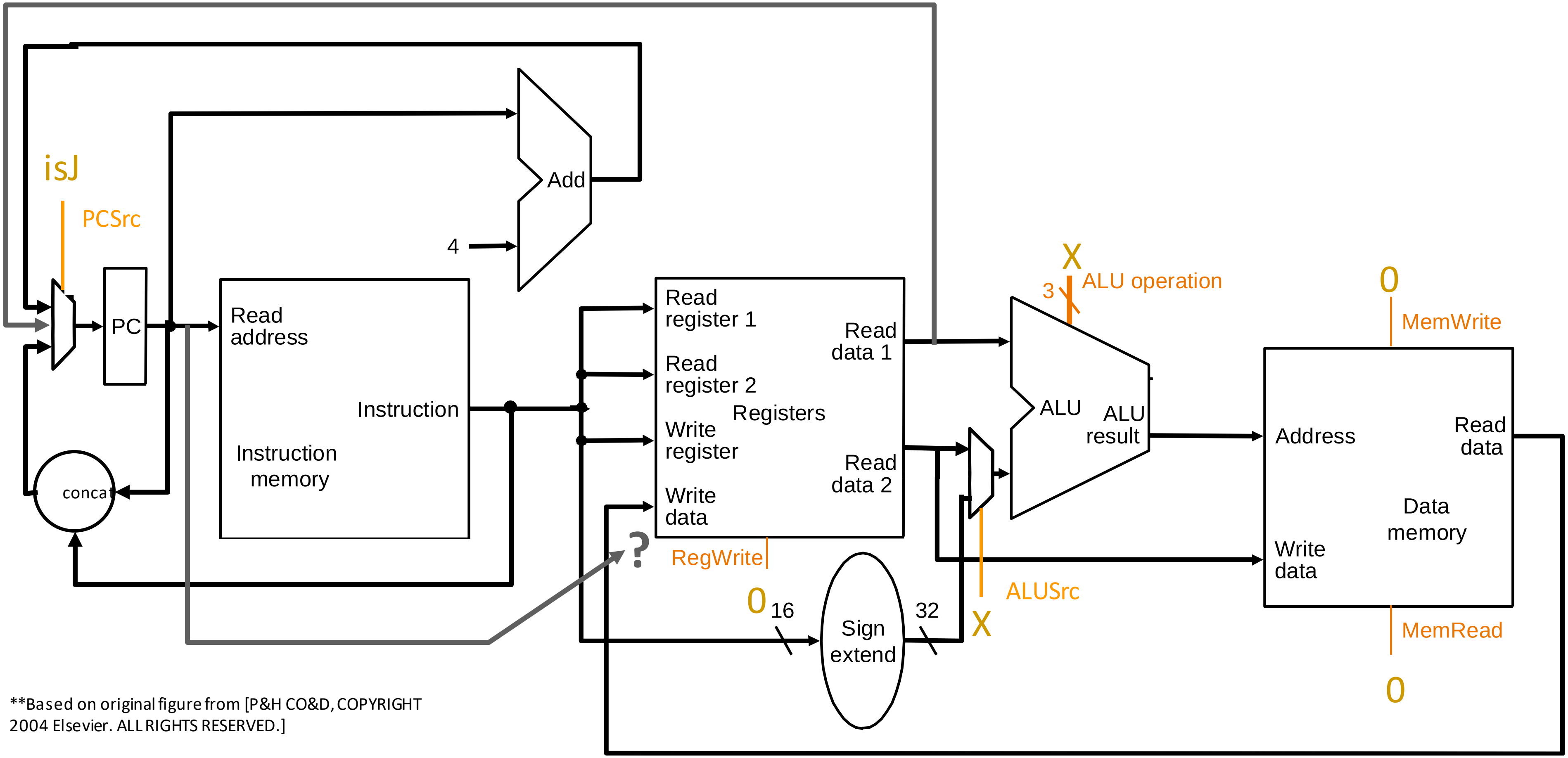
if MEM[PC]== j immediate_{26}

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

    PC ← target

† This is the incremented PC

Unconditional Jump Datapath



**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?

Other Jumps in MIPS

- jal: jump and link (function calls)
  - Semantics

```
if MEM[PC]== jal immediate$_{26}$
    $ra ← PC + 4
    target = { PC[†][31:28], immediate$_{26}$, 2’b00 }
    PC ← target
```

- jr: jump register
  - Semantics

```
if MEM[PC]== jr rs
    PC ← GPR(rs)
```

- jalr: jump and link register
  - Semantics

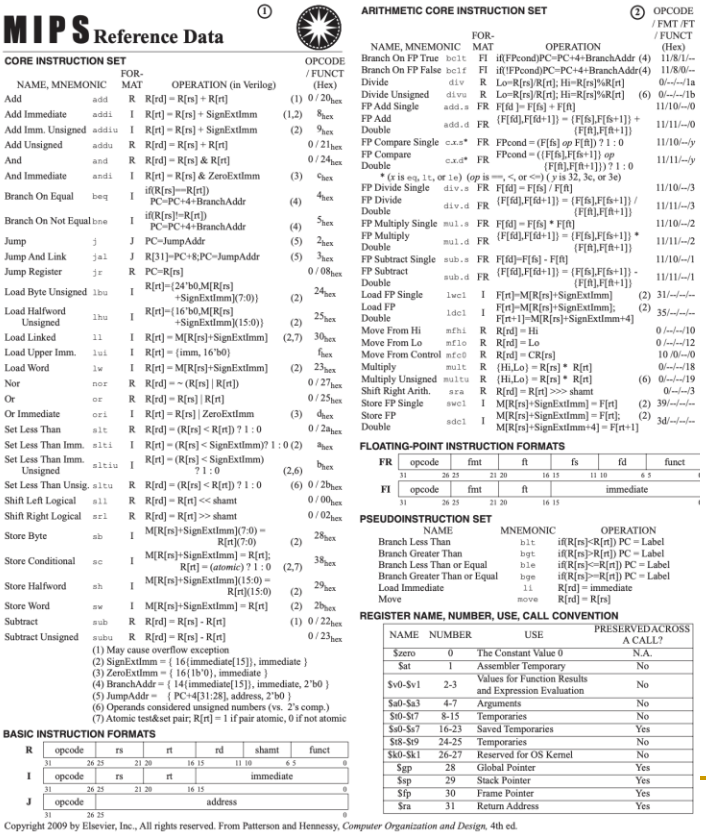
```
if MEM[PC]== jalr rs
    $ra ← PC + 4
    PC ← GPR(rs)
```

[†] This is the incremented PC

Aside: MIPS Cheat Sheet

- https://safari.ethz.ch/digitaltechnik/spring2021/lib/exe/fetch.php?media=mips_reference_data.pdf

- On the course website



MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

	31–26	25–21	20–16	15–11	10–6	5–0
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	address				

Copyright 2009 by Elsevier, Inc., All rights reserved. From Patterson and Hennessy, *Computer Organization and Design*, 4th ed.

ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.*x*.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.*x*.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)

FLOATING-POINT INSTRUCTION FORMATS

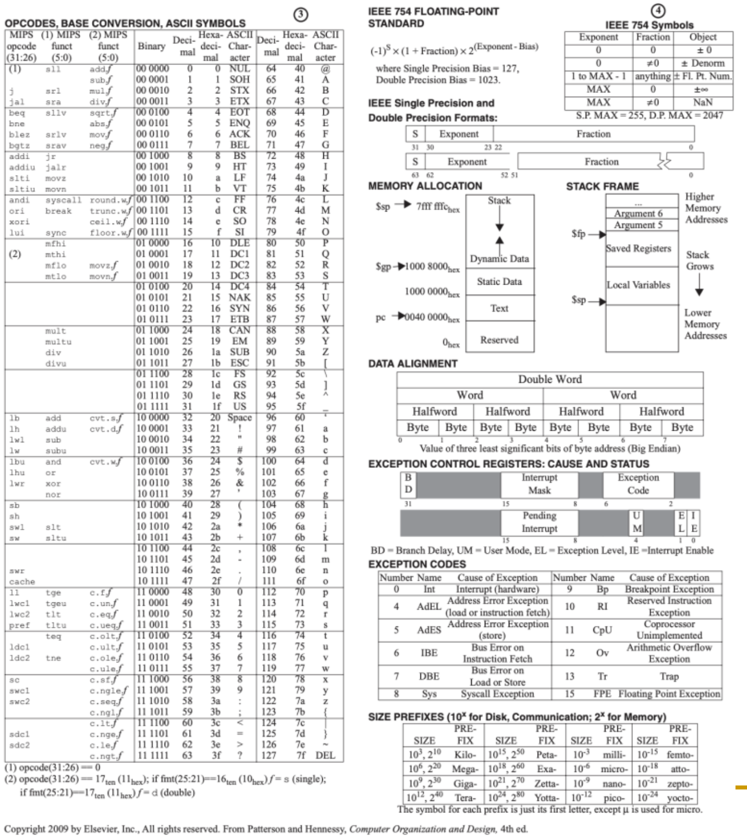
	31–26	25–21	20–16	15–11	10–6	5–0
FR	opcode	fmt	ft	fs	fd	funct
FI	opcode	fmt	ft	immediate		

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes



OPCODES, BASE CONVERSION, ASCII SYMBOLS ③

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.*f*	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.*f*	00 1101	13	d	CR	77	4d	M
xori		ceil.w.*f*	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.*f*	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.*f*	01 0010	18	12	DC2	82	52	R
	mtlo	movn.*f*	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	multu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	`
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.*f*	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	'	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.f.*f*	11 0000	48	30	0	112	70	p
lwc1	tgeu	c.un.*f*	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.*f*	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.*f*	11 0011	51	33	3	115	73	s
	teq	c.olt.*f*	11 0100	52	34	4	116	74	t
ldc1		c.ult.*f*	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.*f*	11 0110	54	36	6	118	76	v
		c.ule.*f*	11 0111	55	37	7	119	77	w
sc		c.sf.*f*	11 1000	56	38	8	120	78	x
swc1		c.ngle.*f*	11 1001	57	39	9	121	79	y
swc2		c.seq.*f*	11 1010	58	3a	:	122	7a	z
		c.ngl.*f*	11 1011	59	3b	;	123	7b	{
		c.lt.*f*	11 1100	60	3c	<	124	7c	|
sdc1		c.nge.*f*	11 1101	61	3d	=	125	7d	}
sdc2		c.le.*f*	11 1110	62	3e	>	126	7e	~
		c.ngt.*f*	11 1111	63	3f	?	127	7f	DEL

(1) opcode(31:26) == 0
(2) opcode(31:26) == 17_{ten} (11_{hex}); if fmt(25:21)==16_{ten} (10_{hex}) *f* = s (single); if fmt(25:21)==17_{ten} (11_{hex}) *f* = d (double)

Copyright 2009 by Elsevier, Inc., All rights reserved. From Patterson and Hennessy, *Computer Organization and Design*, 4th ed.

IEEE 754 FLOATING-POINT STANDARD ④

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

where Single Precision Bias = 127, Double Precision Bias = 1023.

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

IEEE Single Precision and Double Precision Formats:

S (31)	Exponent (30–23)	Fraction (22–0)
S (63)	Exponent (62–52)	Fraction (51–0)

MEMORY ALLOCATION

$sp → 7fff $fffc_{hex}$	Stack ↓ ↑ Dynamic Data
$gp → 1000 8000_{hex}	Static Data
1000 0000_{hex}	Text
pc → 0040 0000_{hex}	Reserved
0_{hex}	

STACK FRAME

	...	Higher Memory Addresses
	Argument 6	
	Argument 5	
$fp →	Saved Registers	Stack Grows ↓
	Local Variables	
$sp →		Lower Memory Addresses

DATA ALIGNMENT

Double Word							
Word				Word			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7

Value of three least significant bits of byte address (Big Endian)

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS

31		15–8		6–2	
BD		Interrupt Mask		Exception Code	

	15–8		4		1	0
	Pending Interrupt		UM		EL	IE

BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX
$10^3, 2^{10}$	Kilo-	$10^{15}, 2^{50}$	Peta-	10^{-3}	milli-	10^{-15}	femto-
$10^6, 2^{20}$	Mega-	$10^{18}, 2^{60}$	Exa-	10^{-6}	micro-	10^{-18}	atto-
$10^9, 2^{30}$	Giga-	$10^{21}, 2^{70}$	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
$10^{12}, 2^{40}$	Tera-	$10^{24}, 2^{80}$	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

The symbol for each prefix is just its first letter, except µ is used for micro.

Copyright 2009 by Elsevier, Inc., All rights reserved. From Patterson and Hennessy, *Computer Organization and Design*, 4th ed.

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

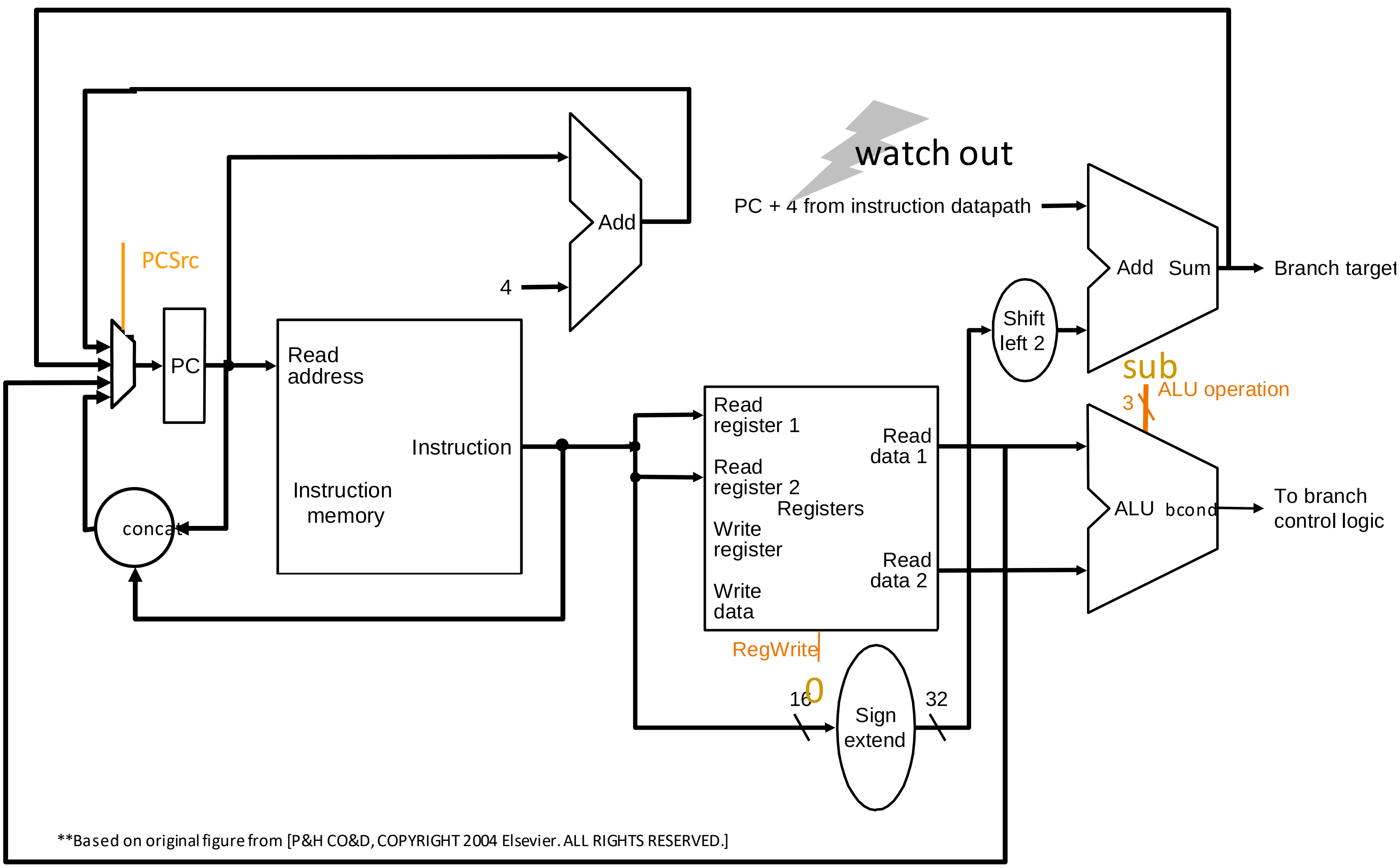
- Semantics (assuming no branch delay slot)

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

  - Variations: beq, bne, blez, bgtz

[†] This is the incremented PC

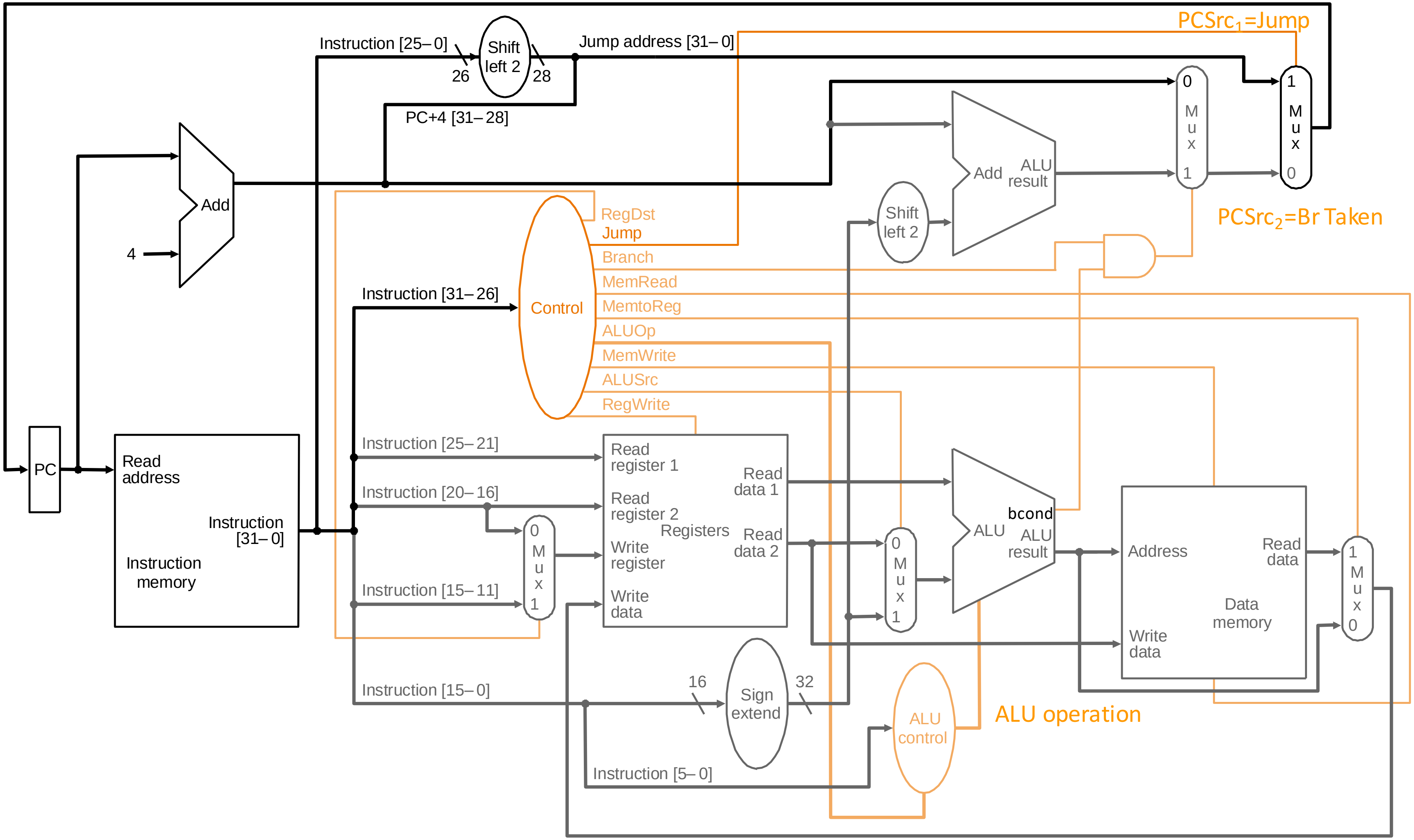
Conditional Branch Datapath (for you to finish)



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

How to uphold the delayed branch semantics?

Putting It All Together



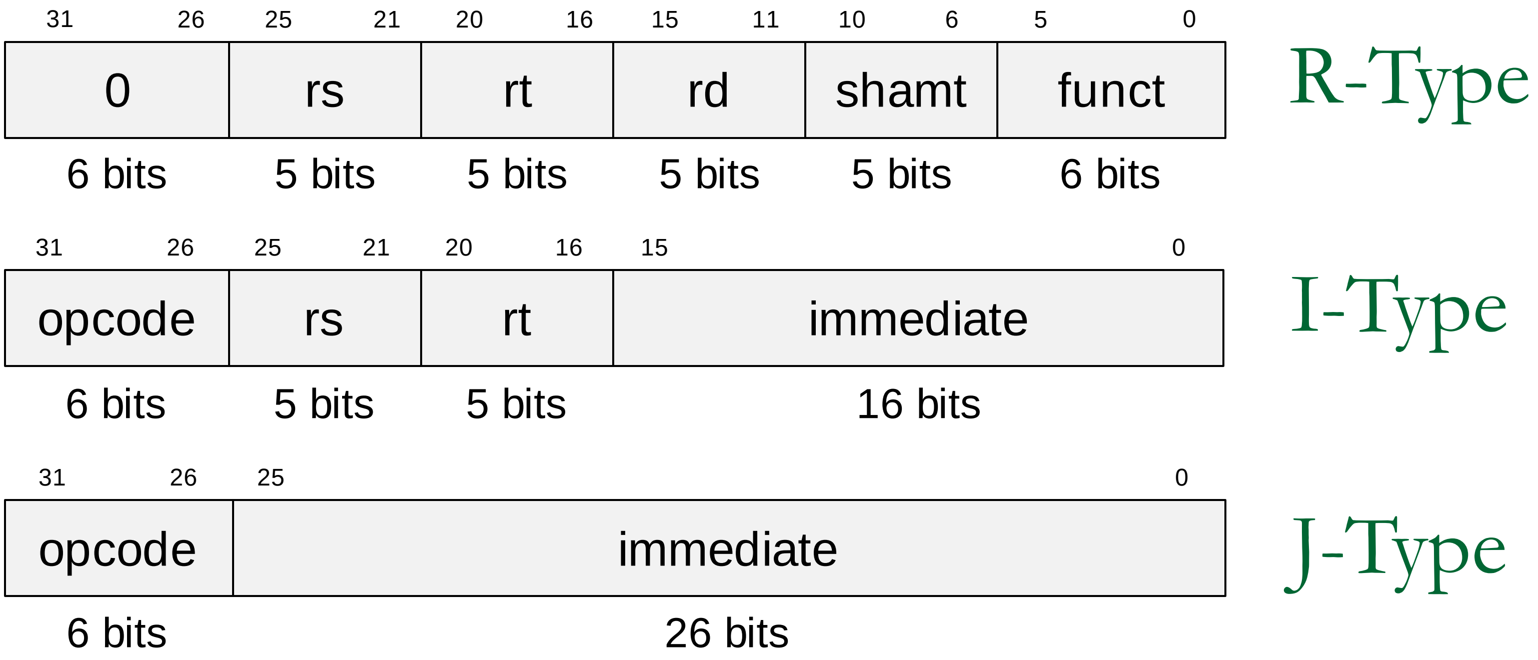
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted

Single-Cycle Control Logic

Single-Cycle Hardwired Control

- As combinational function of Inst=MEM[PC]



R-Type

31 26	25 21	20 16	15 11	10 6	5 0
0	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Type

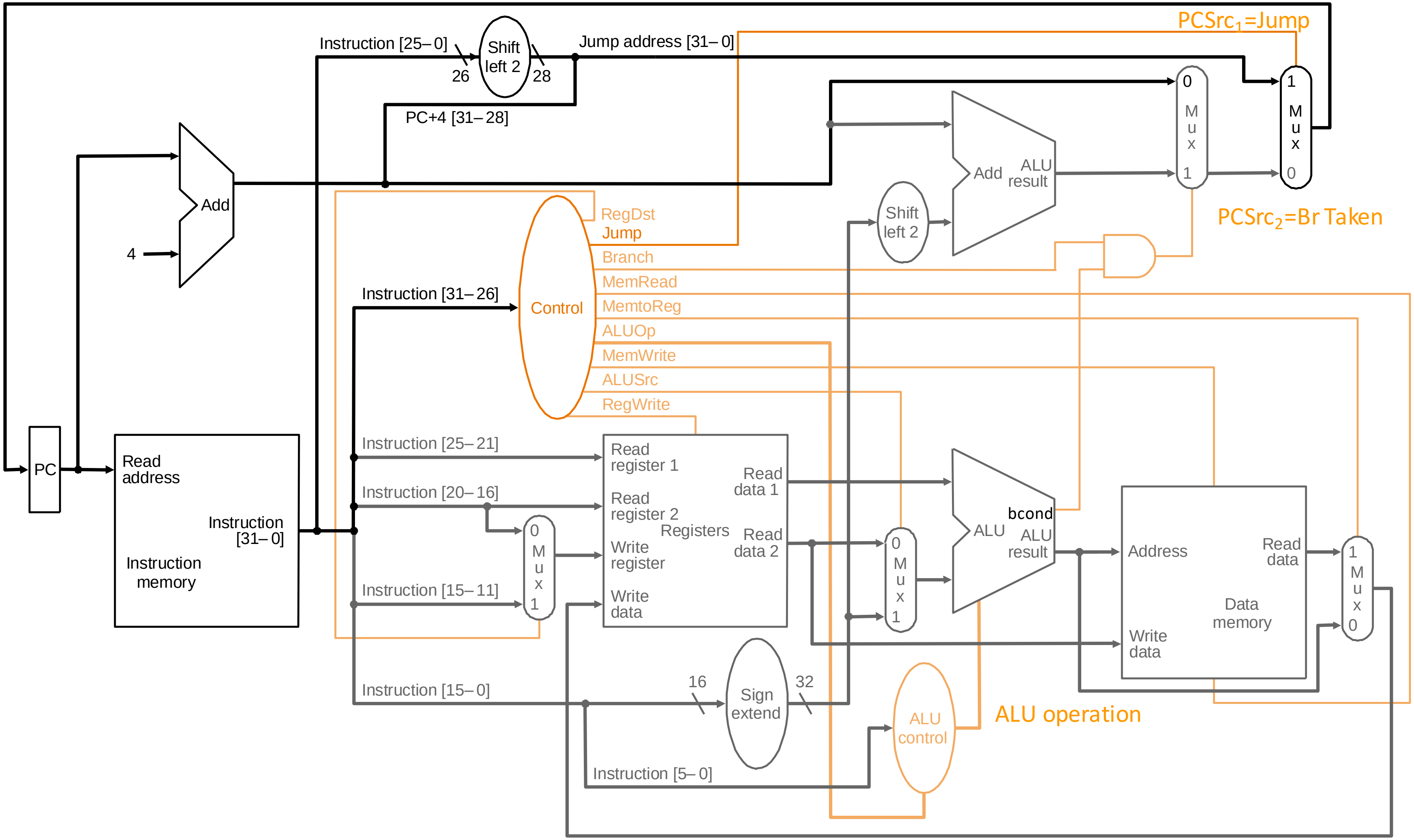
31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

J-Type

31 26	25 0
opcode	immediate
6 bits	26 bits

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

Generate Control Signals (in Orange Color)



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

JAL, JR, JALR omitted

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))

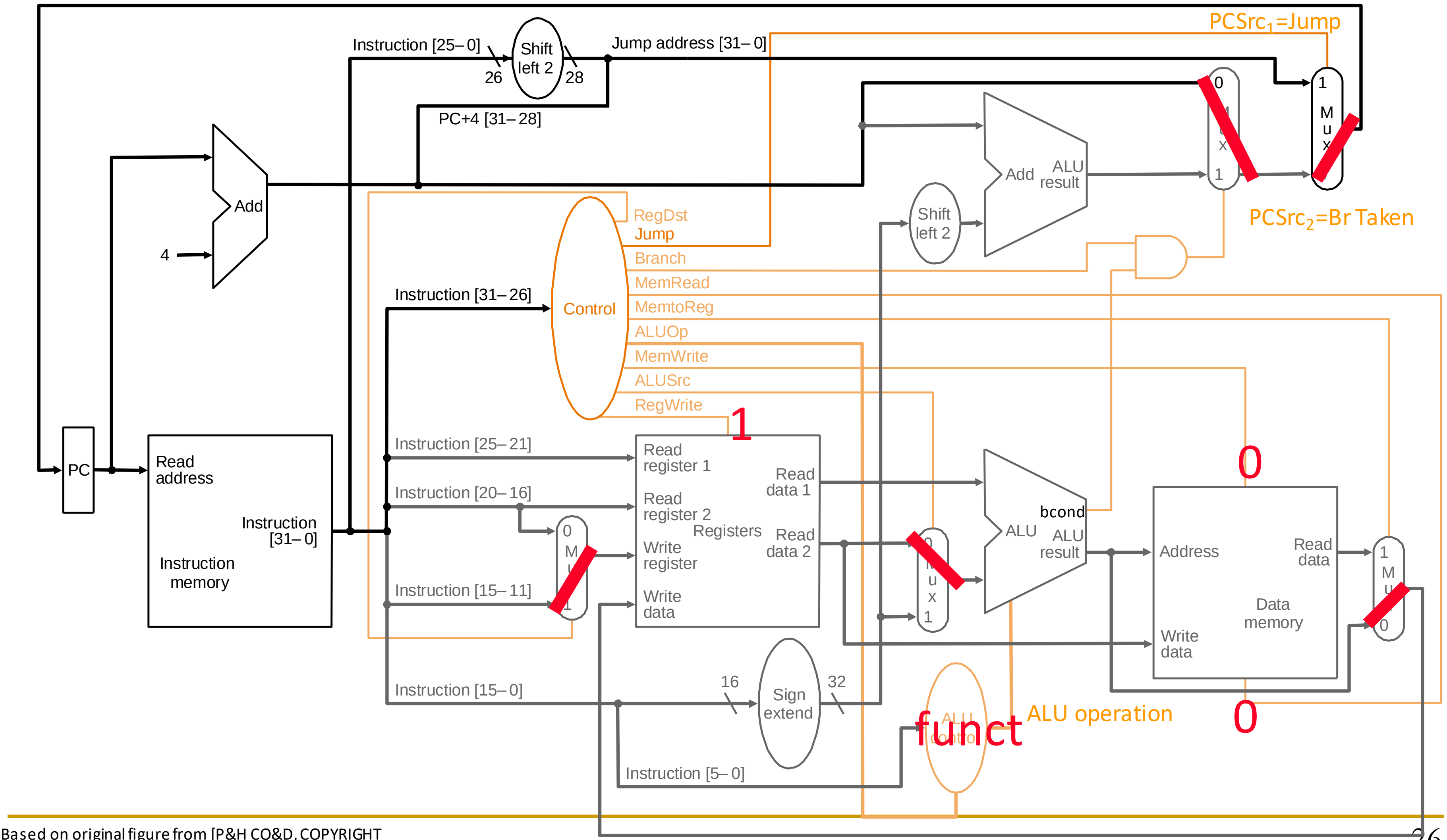
JAL and JALR require additional RegDest and MemtoReg options

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"

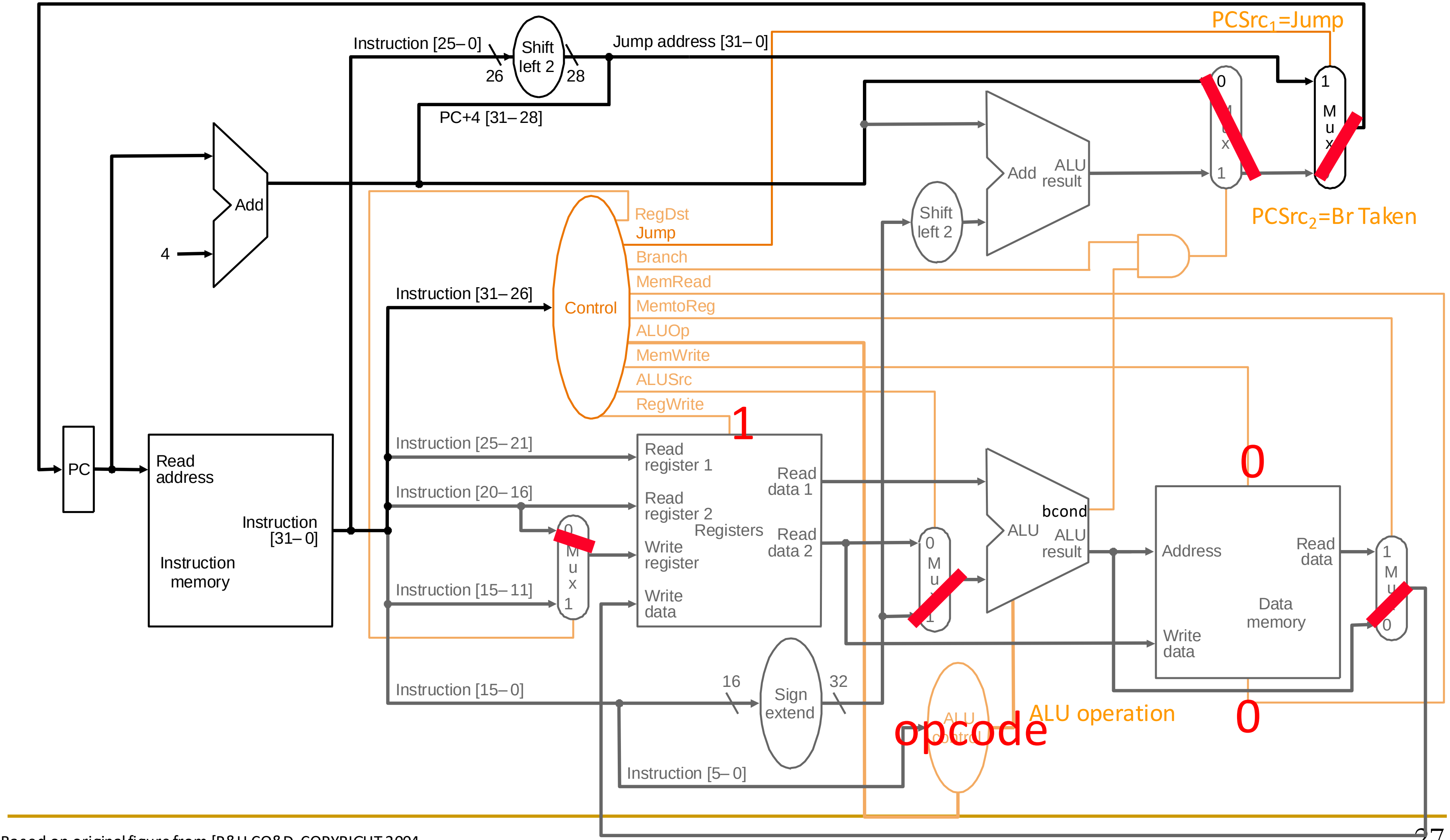
JR and JALR require additional PCSrc options

R-Type ALU



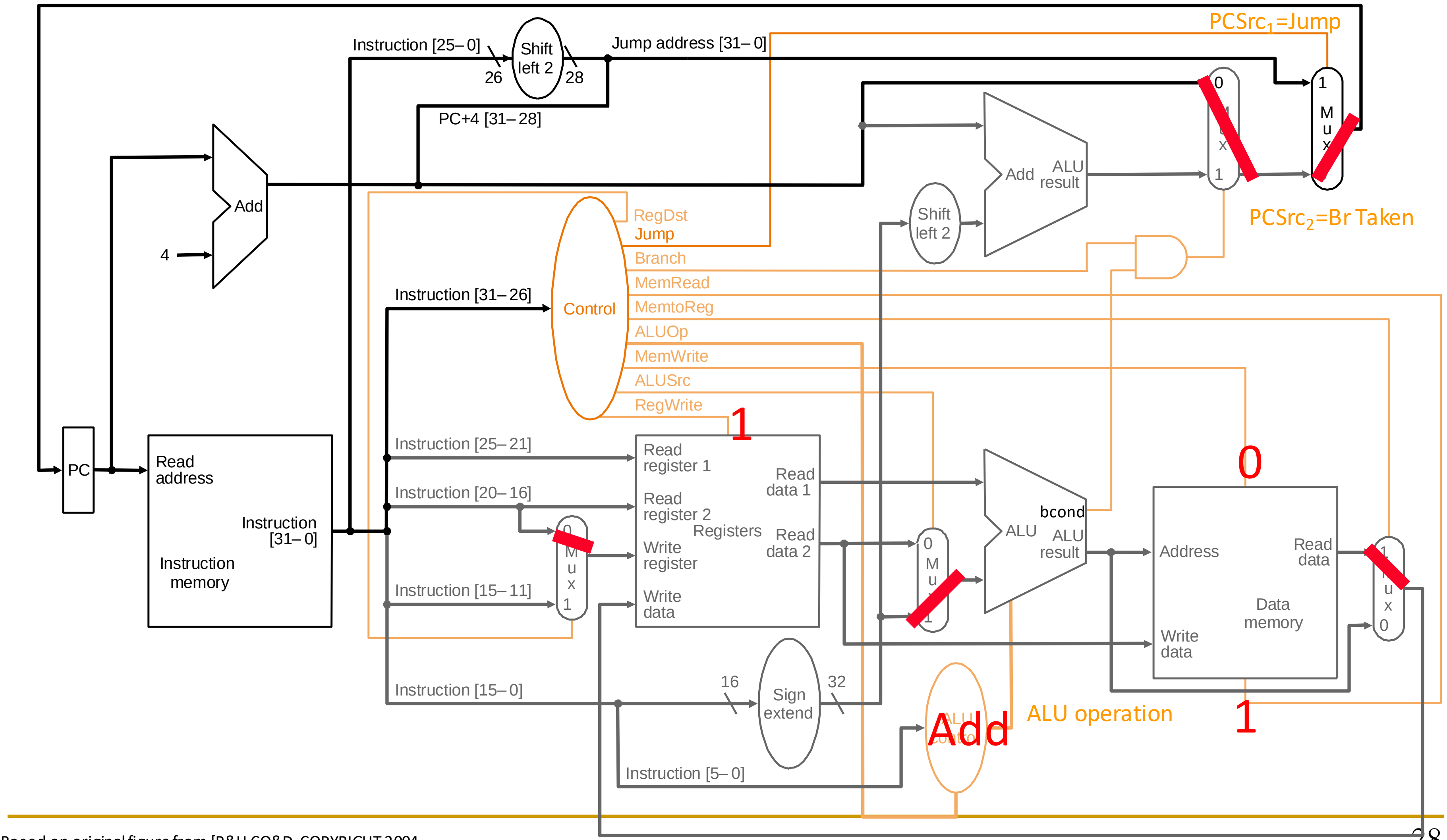
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

I-Type ALU



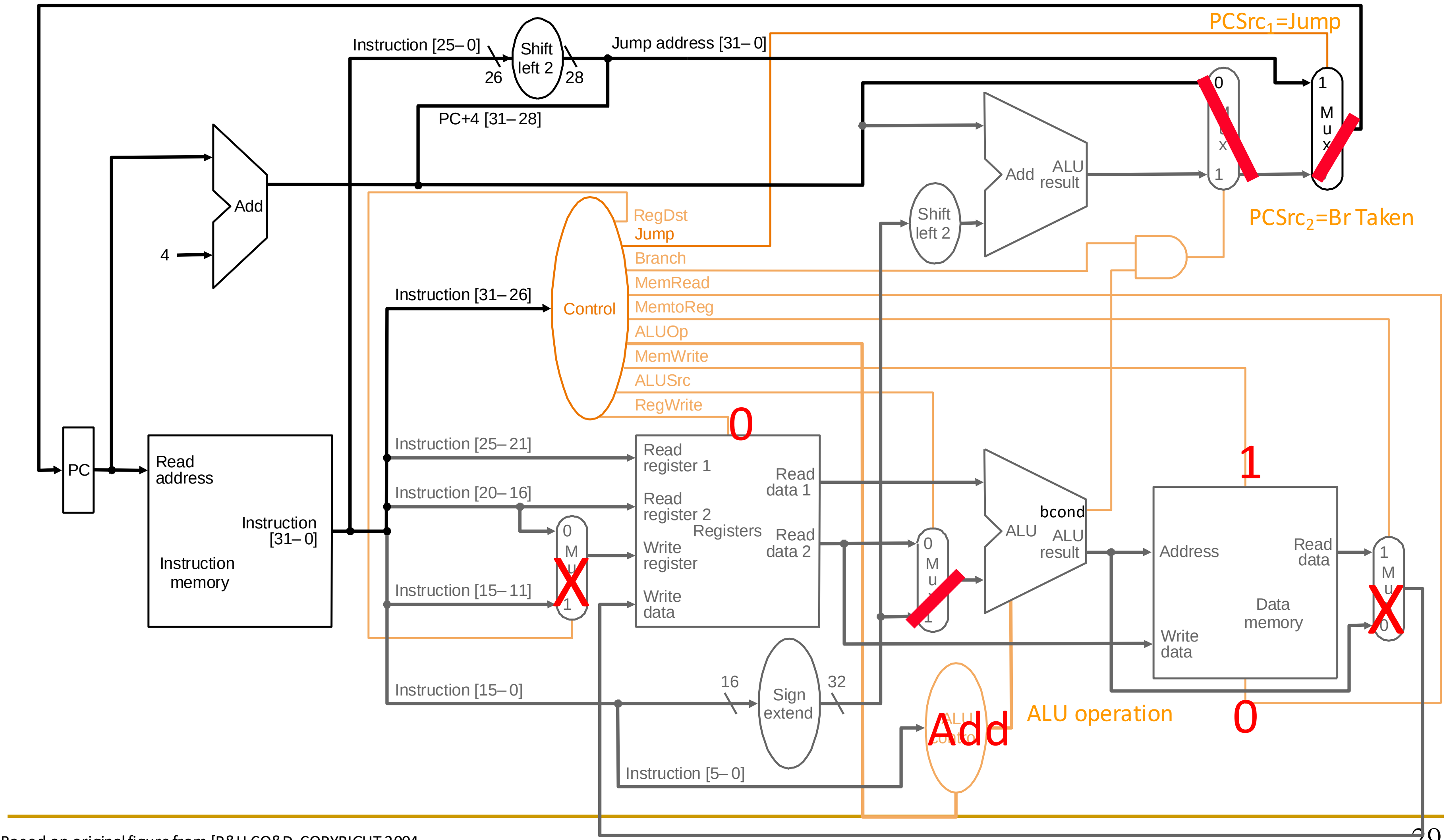
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

LW



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

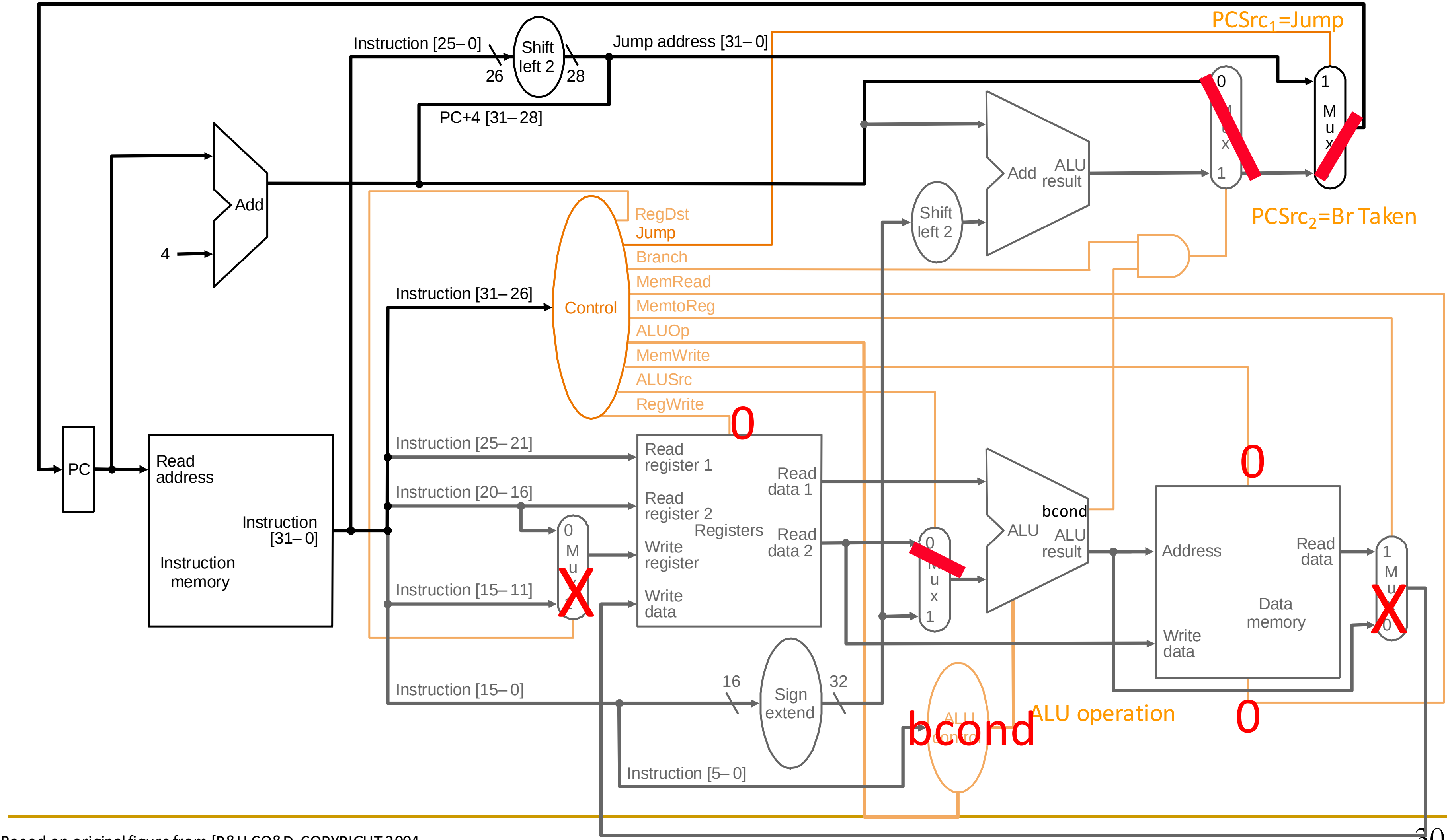
SW



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

Branch (Not Taken)

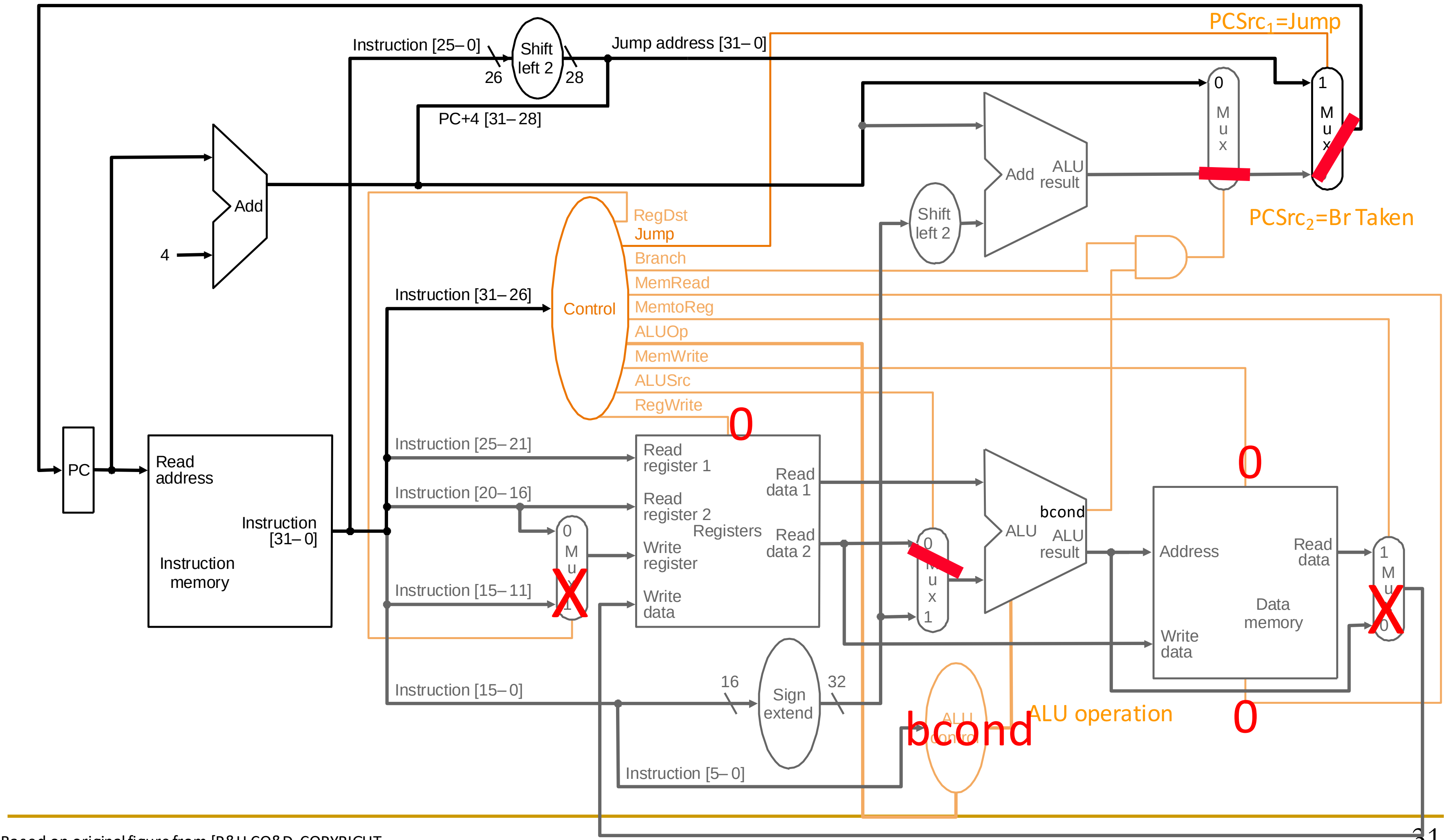
Some control signals are dependent on the processing of data



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

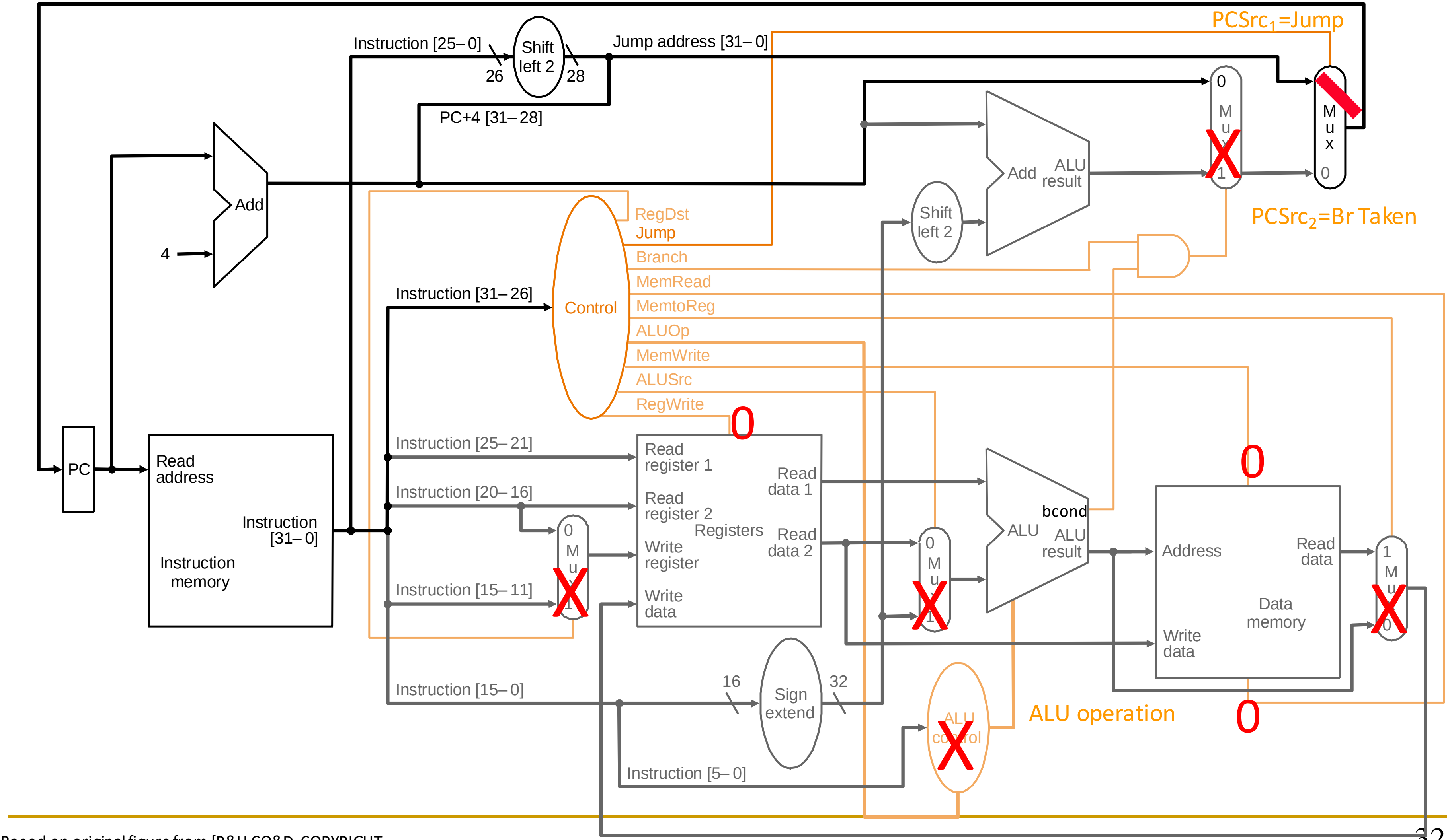
Branch (Taken)

Some control signals are dependent on the processing of data



**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.]

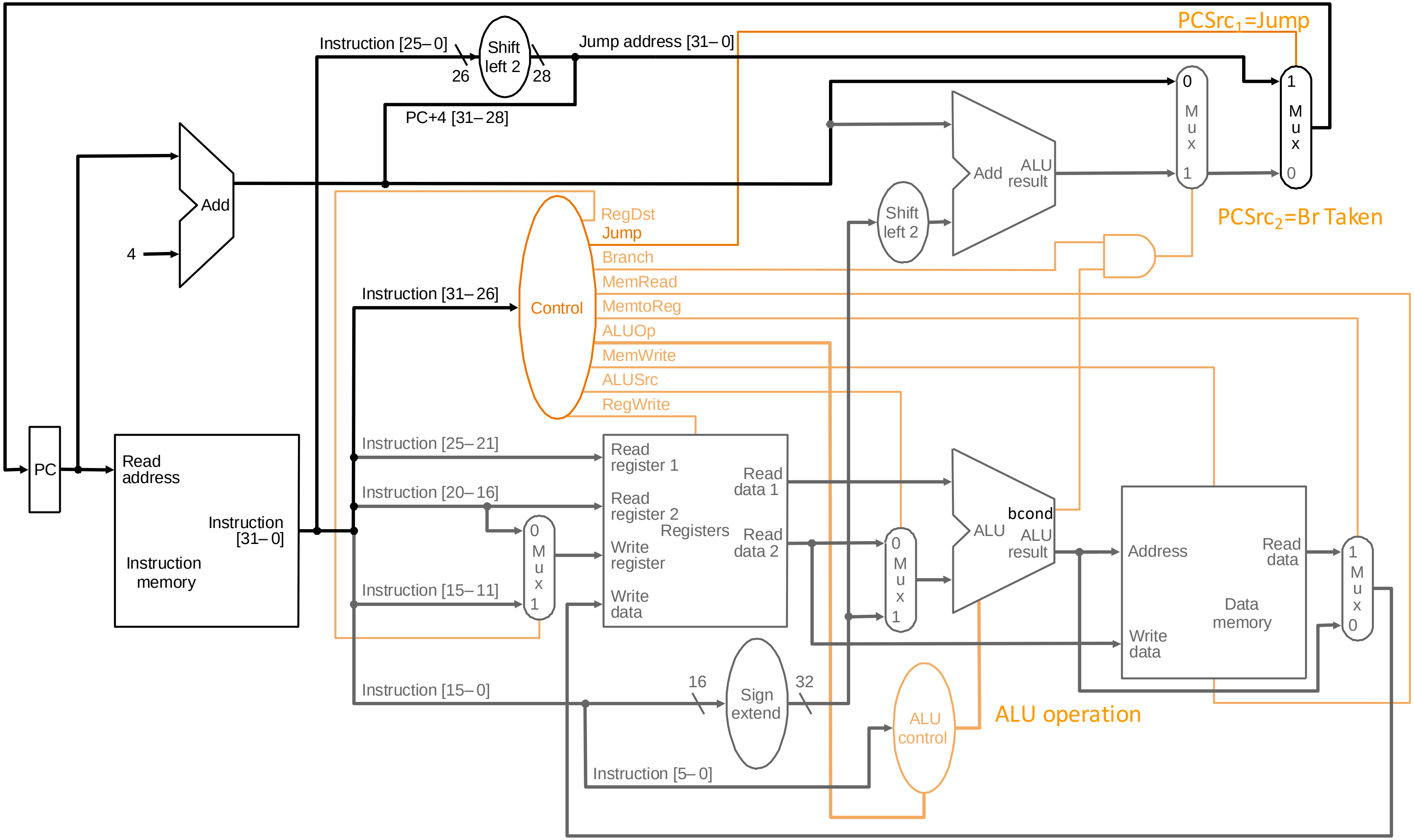
What is in That Control Box?

- 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 latency of each structure + how much on the critical path control signal generation is, etc.

Review: Complete Single-Cycle Processor



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

JAL, JR, JALR omitted

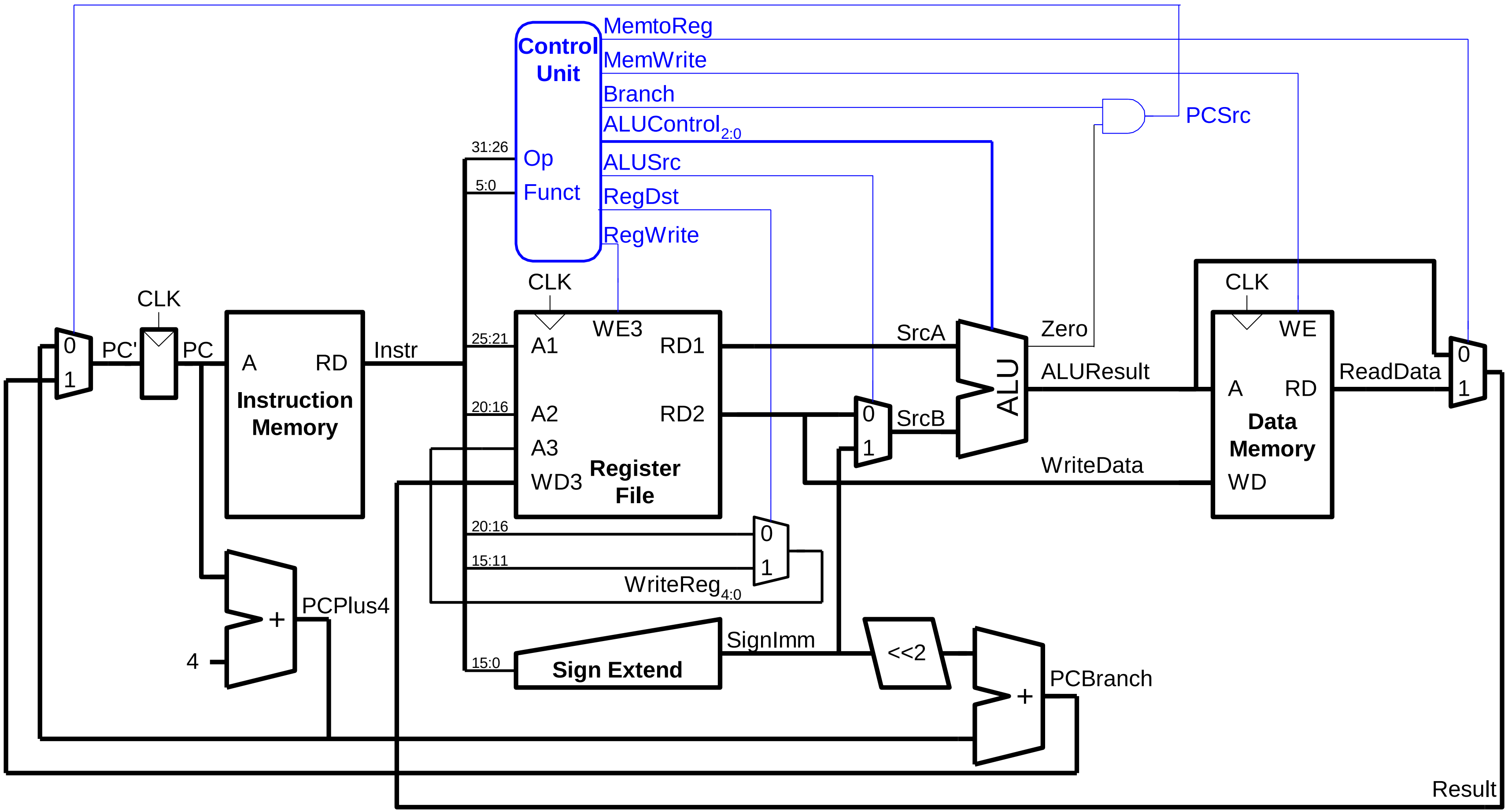
Another Single-Cycle MIPS Processor (from H&H)

See backup slides to reinforce the concepts we have covered.

They are to complement your reading:

H&H, Chapter 7.1-7.3, 7.6

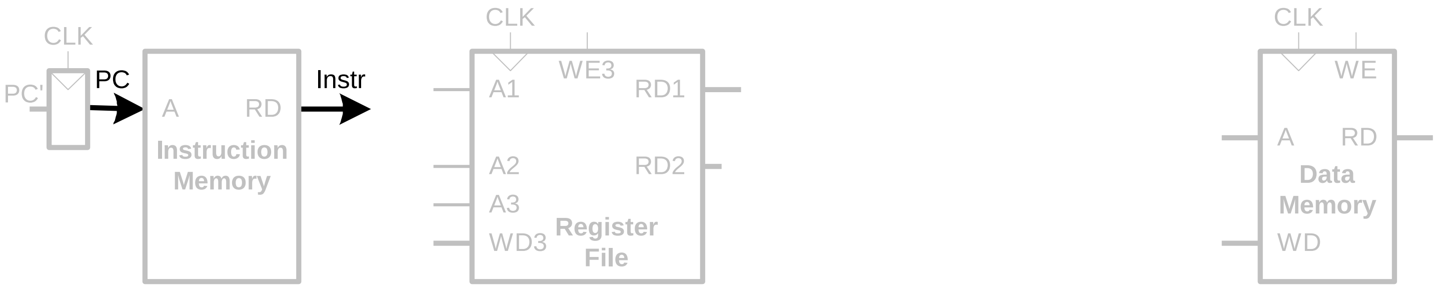
Another Complete Single-Cycle Processor



Single-cycle processor. Harris and Harris, Chapter 7.3.

Example: Single-Cycle Datapath: `lw` fetch

- *STEP 1:* Fetch instruction



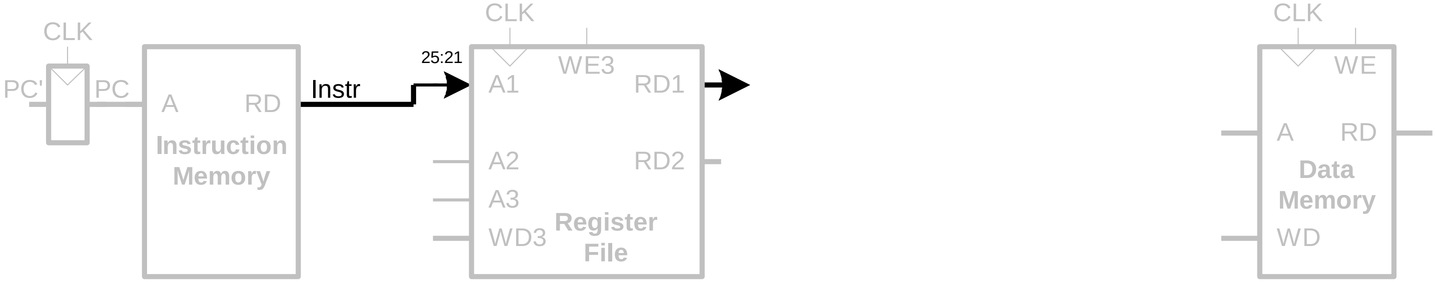
```
lw $s3, 1($0)  # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` register read

- *STEP 2:* **Read source operands from register file**



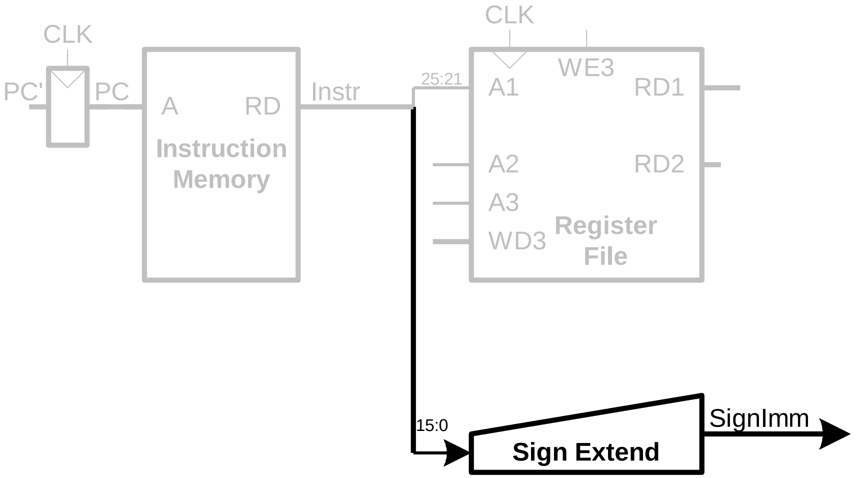
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` immediate

- ***STEP 3:*** **Sign-extend the immediate**



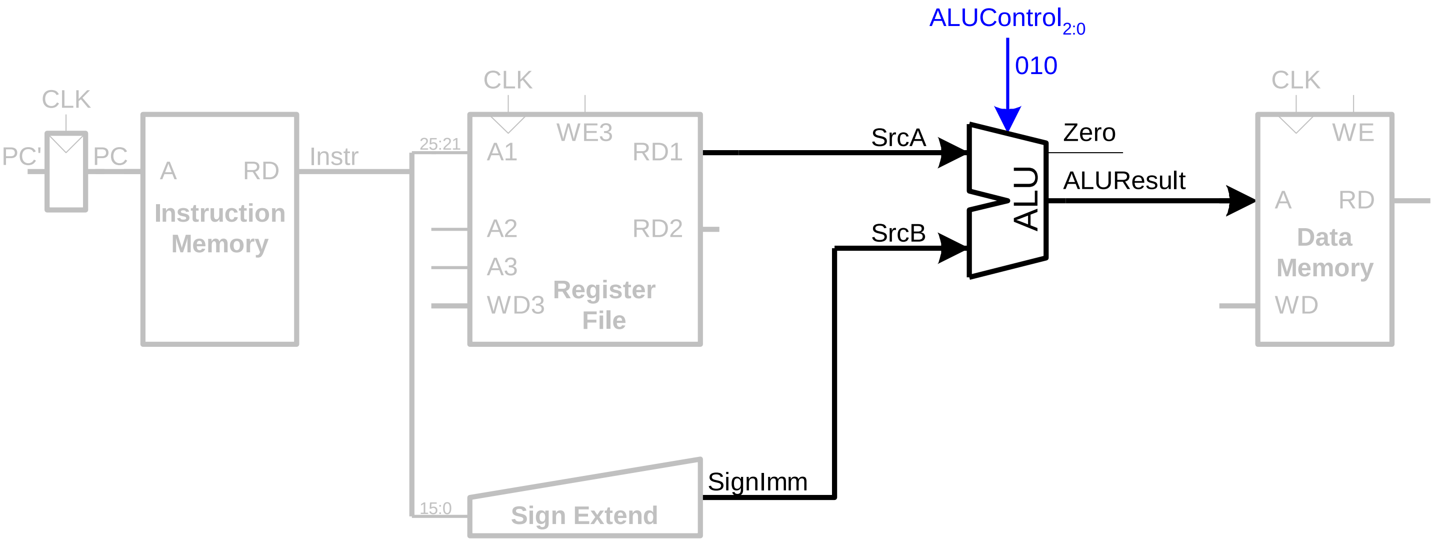
```
lw $s3, 1($0)  # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` address

- *STEP 4:* **Compute the memory address**



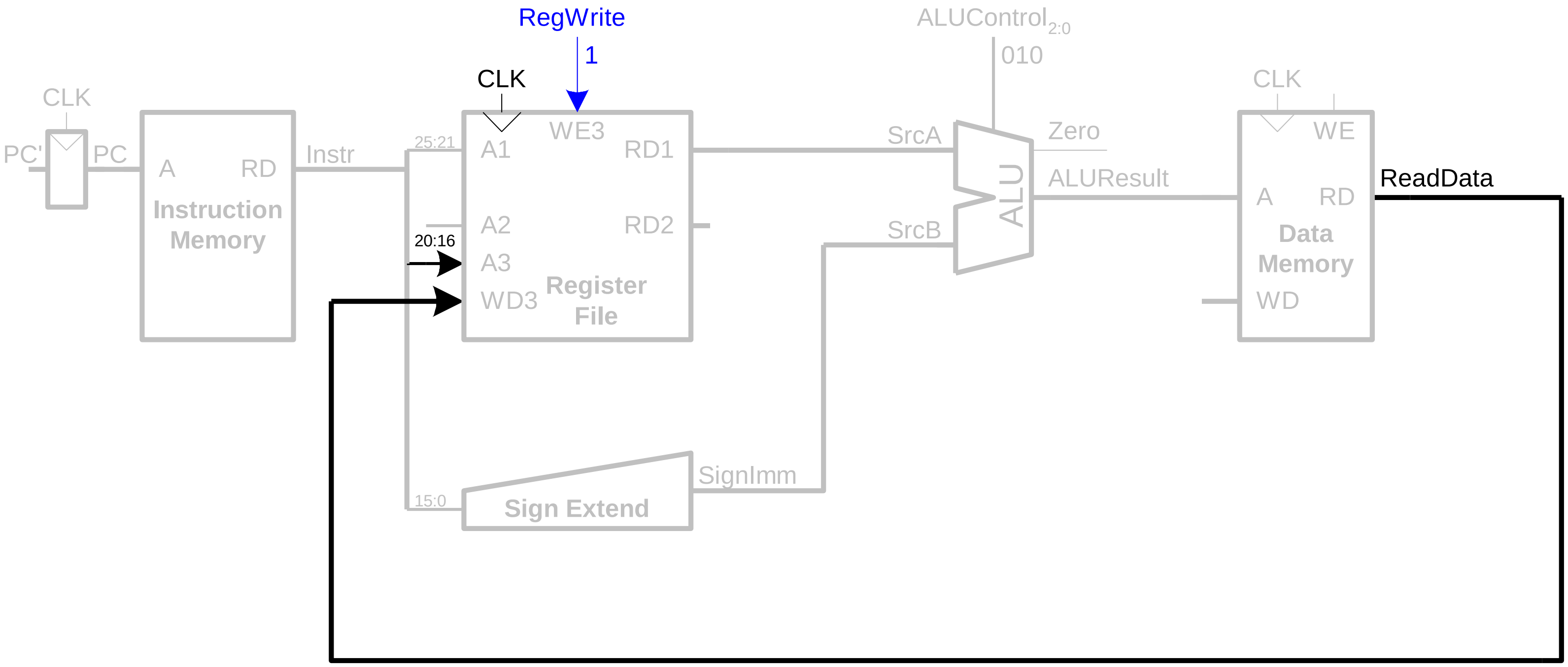
```
lw $s3, 1($0)  # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` memory read

- ***STEP 5:*** **Read from memory and write back to register file**



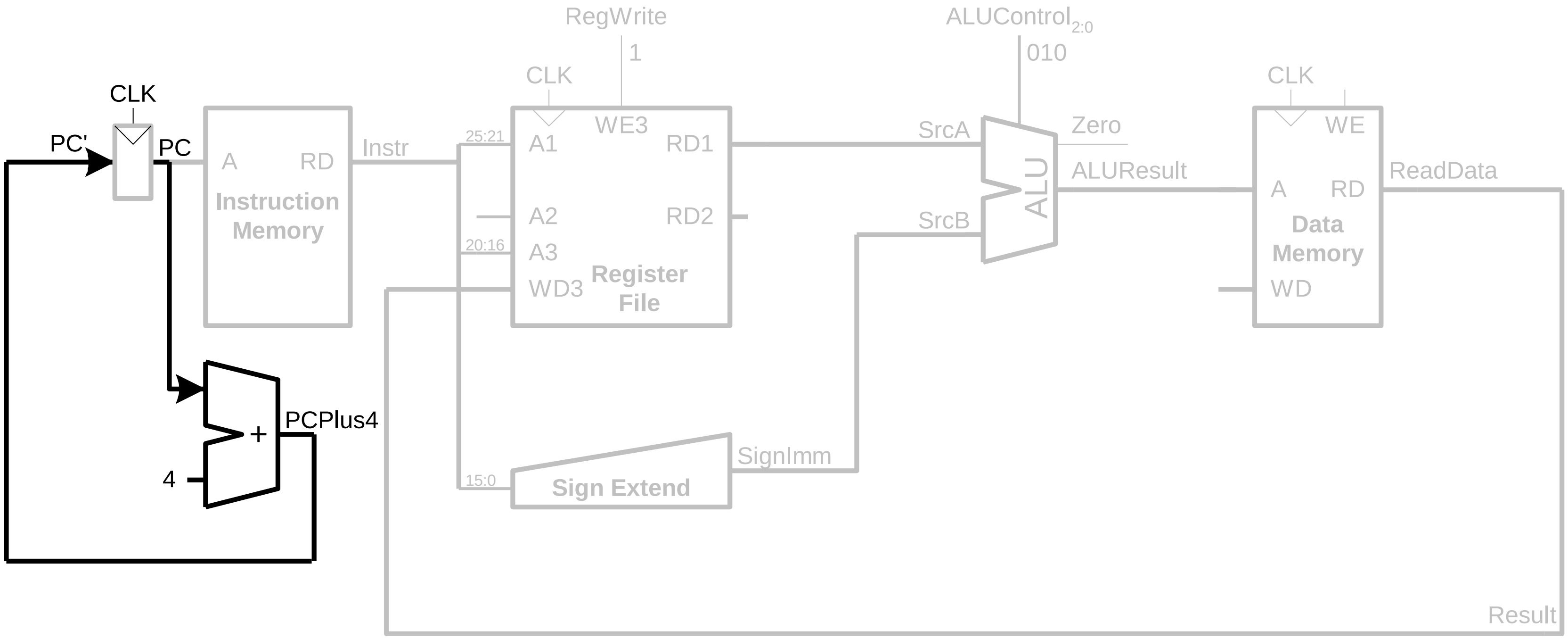
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` PC increment

- ***STEP 6:*** **Determine address of next instruction**



```
lw $s3, 1($0)  # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

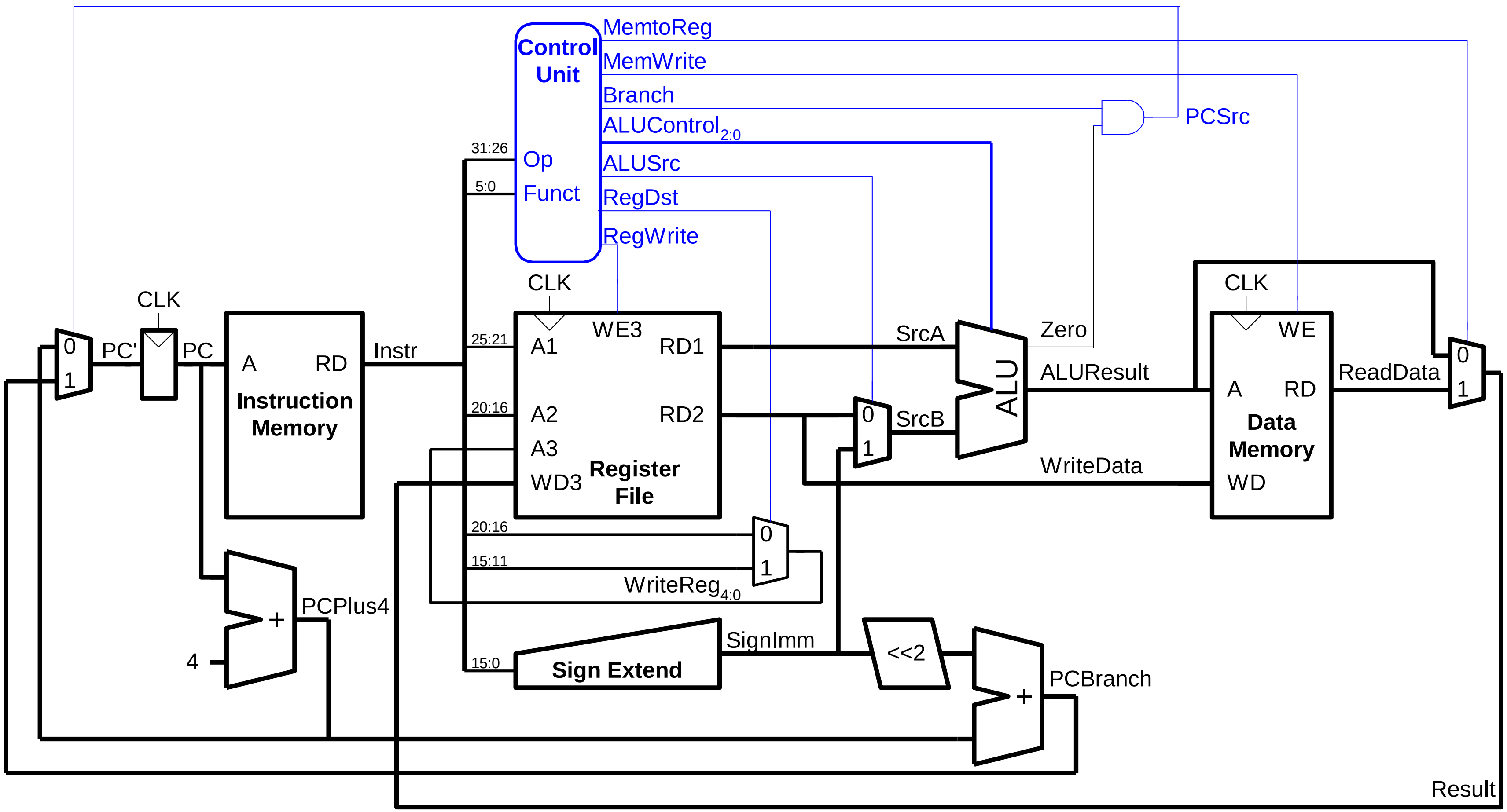
Similarly, We Need to Design the Control Unit

- 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

Single-cycle processor. Harris and Harris, Chapter 7.3.

Another Complete Single-Cycle Processor (H&H)

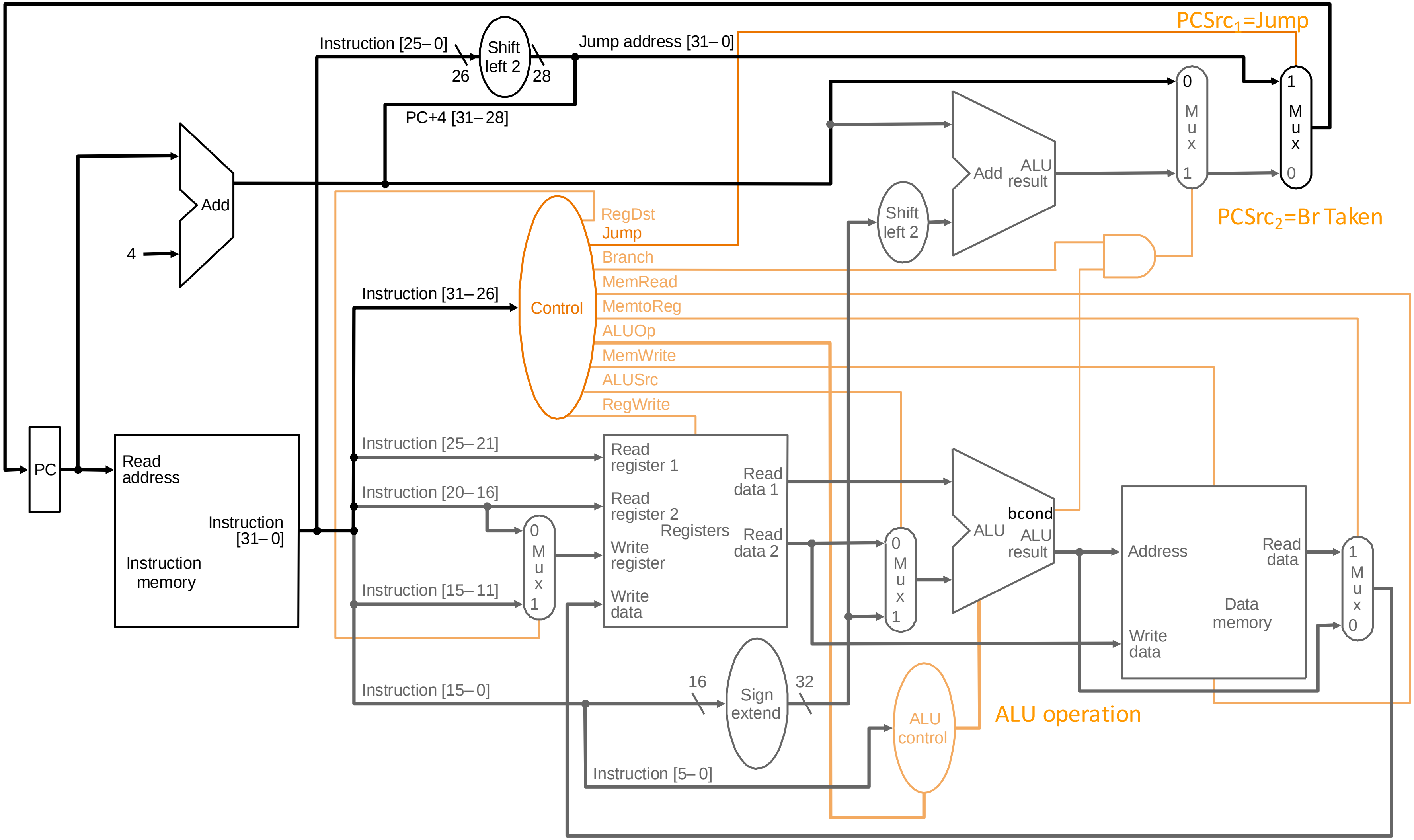


Your Reading Assignment

- Please read the Lecture Slides & the Backup Slides

- Please do your readings from the H&H Book
  - H&H, Chapter 7.1-7.3, 7.6

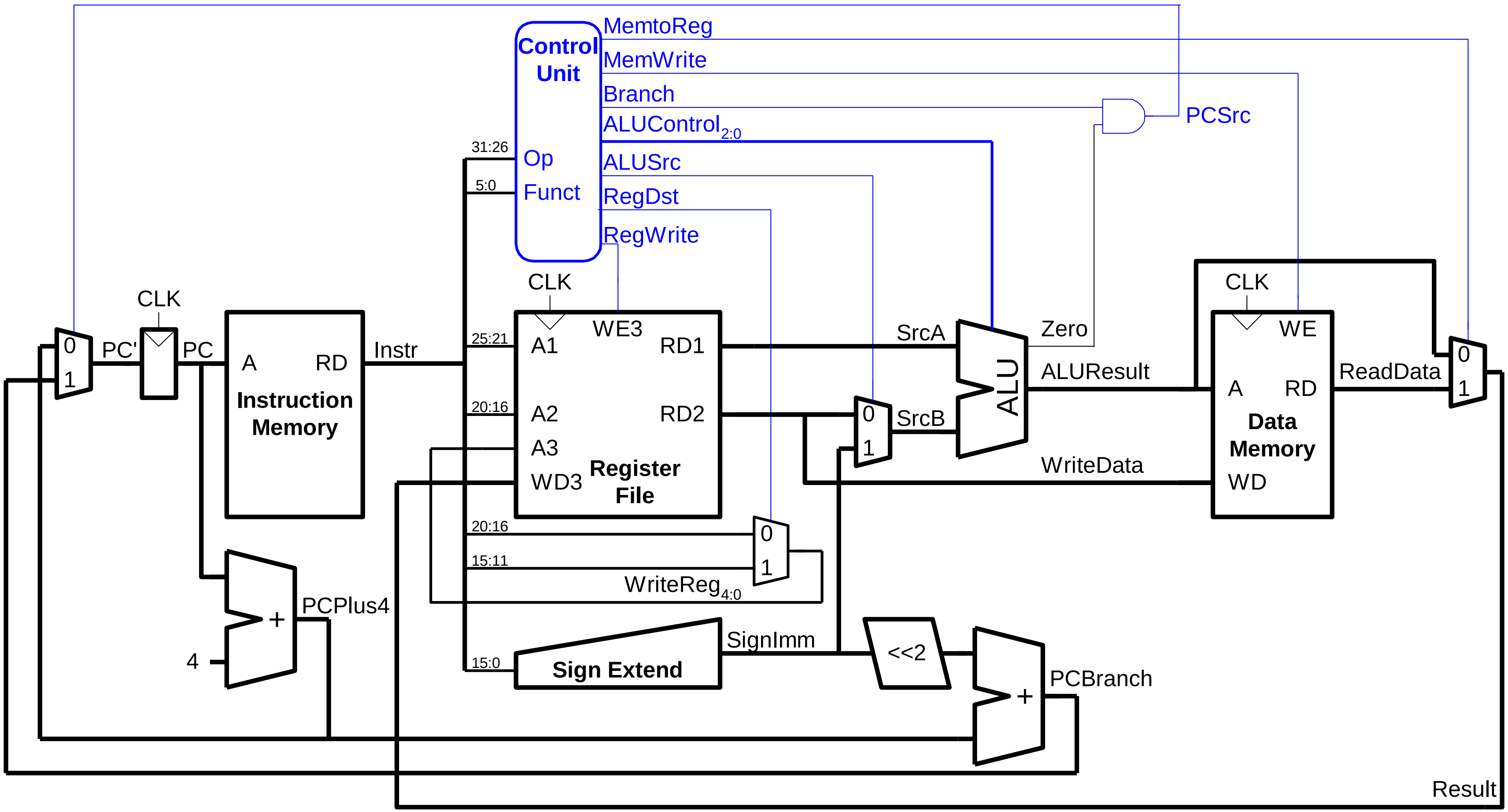
Single-Cycle Uarch I (We Developed in Lectures)



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

JAL, JR, JALR omitted

Single-Cycle Uarch II (In Your Readings)



Single-cycle processor. Harris and Harris, Chapter 7.3.

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?

Performance Analysis Basics

Recall: Performance Analysis Basics

- Execution time of a single instruction
  - **{CPI} x {clock cycle time}**
    - CPI: Number of cycles it takes to execute an 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}**

Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed

Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed

- **How fast are my instructions?**
  - Instructions are realized on the hardware
  - Each instruction can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***

Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed

- **How fast are my instructions?**
  - Instructions are realized on the hardware
  - Each instruction can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***

- **How long is one clock cycle?**
  - The critical path determines how much time one cycle requires = ***clock period***
  - 1/clock period = ***clock frequency*** = how many cycles can be done each second

Processor Performance

- **As a general formula**
  - Our program consists of executing **N** instructions
  - Our processor needs **CPI** cycles (on average) for each instruction
  - The clock frequency of the processor is **f**

    → the clock period is therefore **T**=1/f

Processor Performance

- **As a general formula**
  - Our program consists of executing **N** instructions
  - Our processor needs **CPI** cycles (on average) for each instruction
  - The clock frequency of the processor is **f**

    → the clock period is therefore **T**=1/f

- **Our program executes in**

N x CPI x (1/f) =

N x CPI x T seconds

Performance Analysis of Our Single-Cycle Design

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?

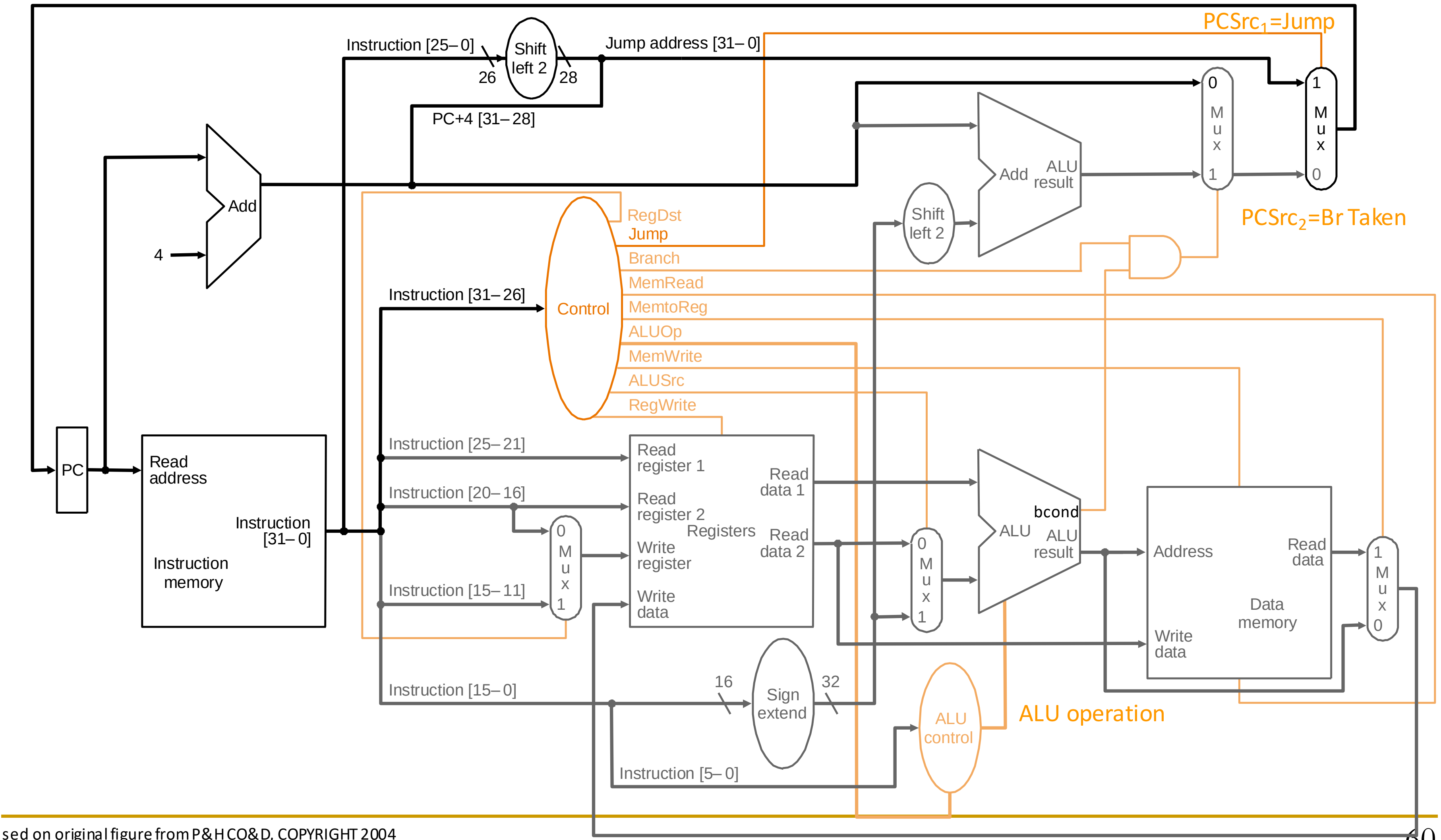
- Let’s go back to the basics

- 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?

Let's Find the Critical Path



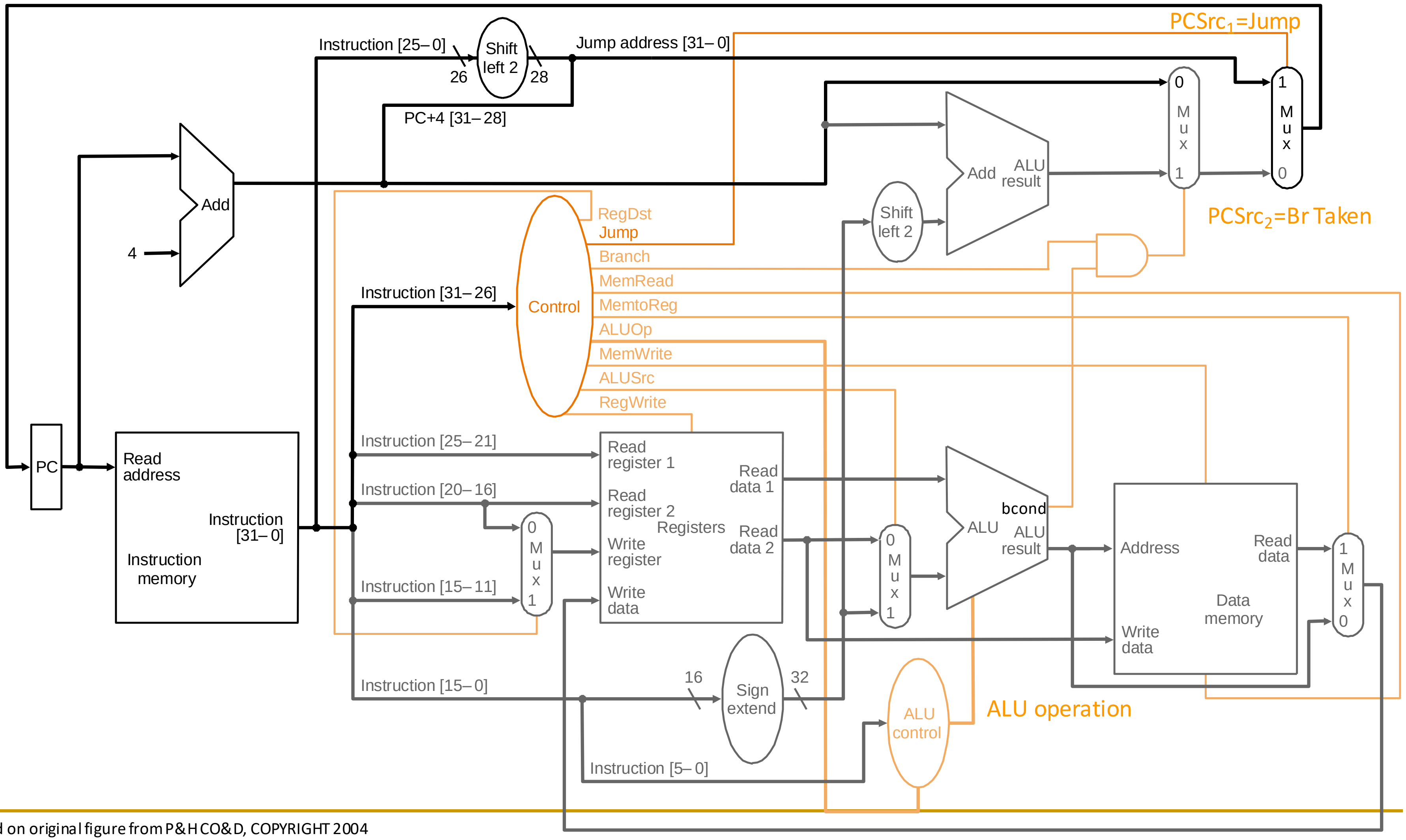
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Example Single-Cycle Datapath Analysis

- Assume (for the design in the previous slide)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

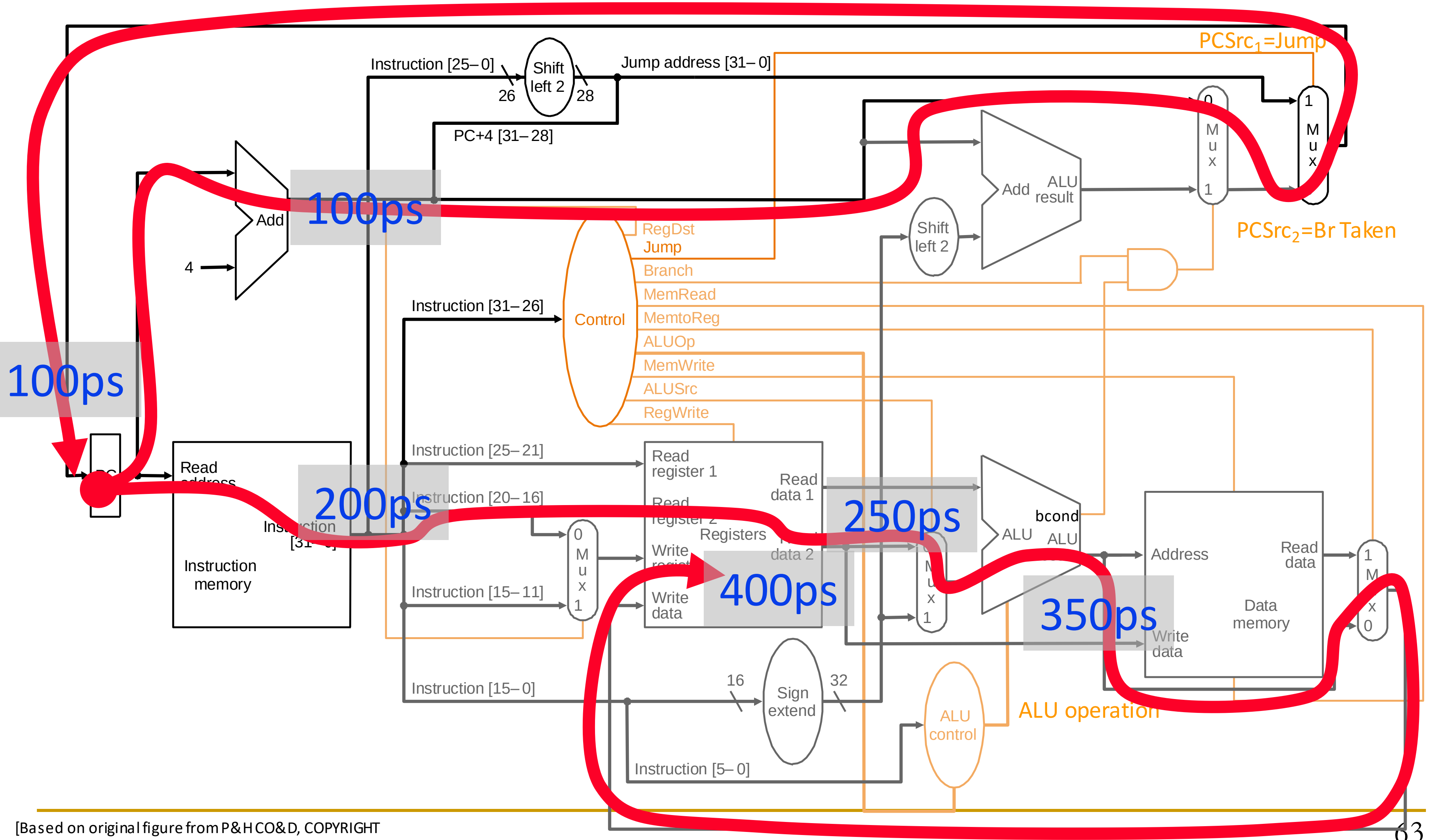
steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

Let’s Find the Critical Path



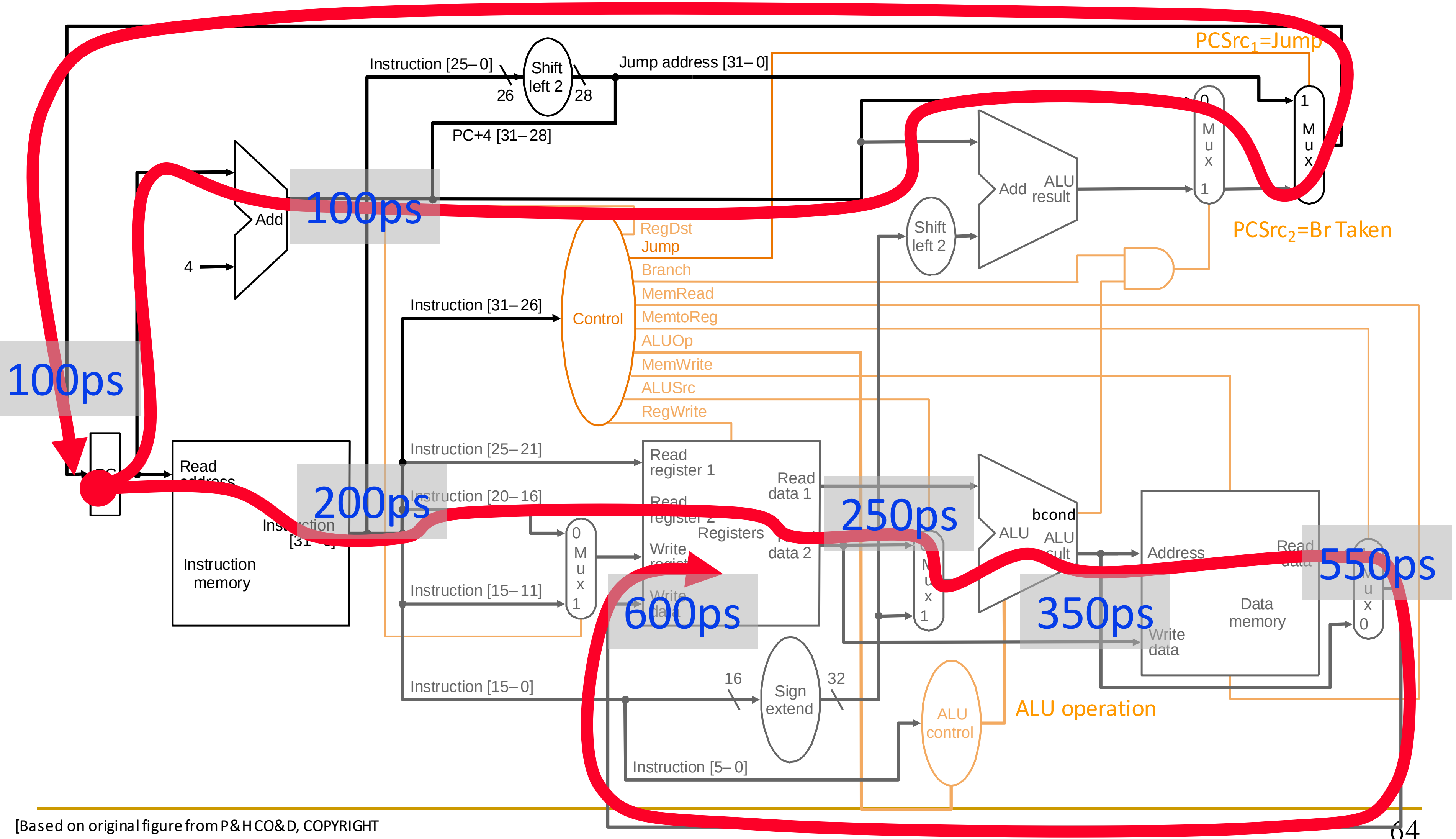
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

R-Type and I-Type ALU



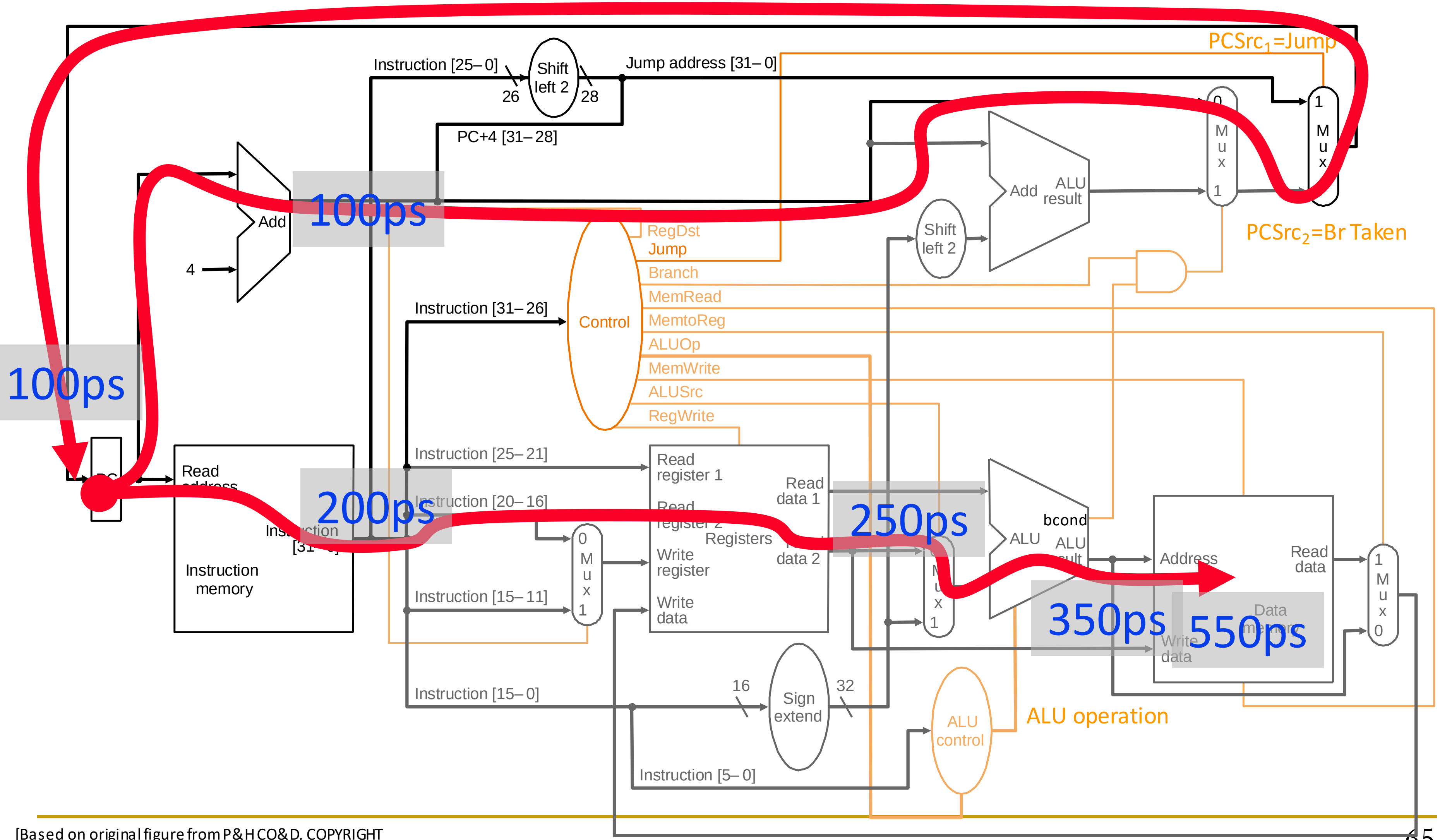
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

LW



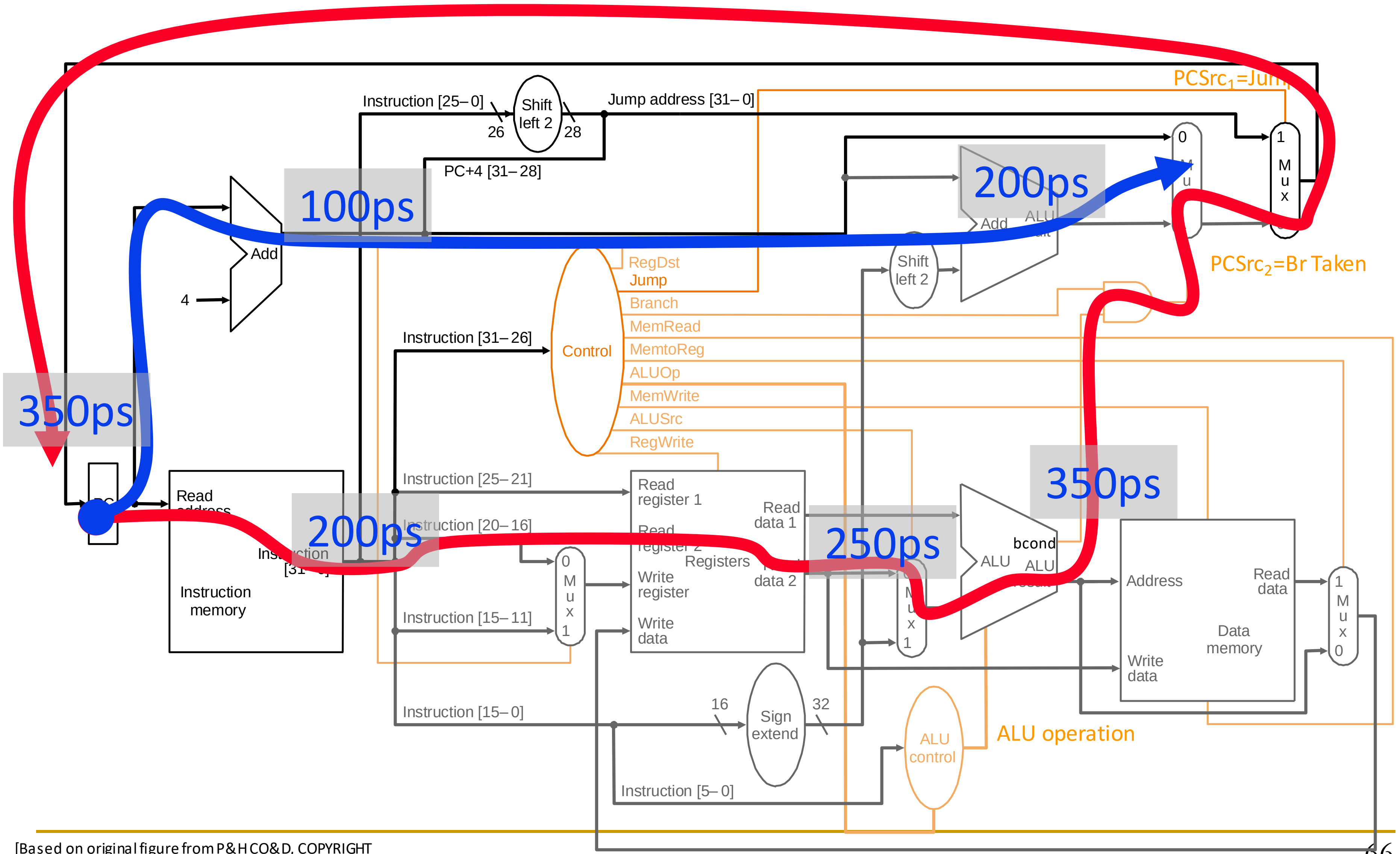
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

SW



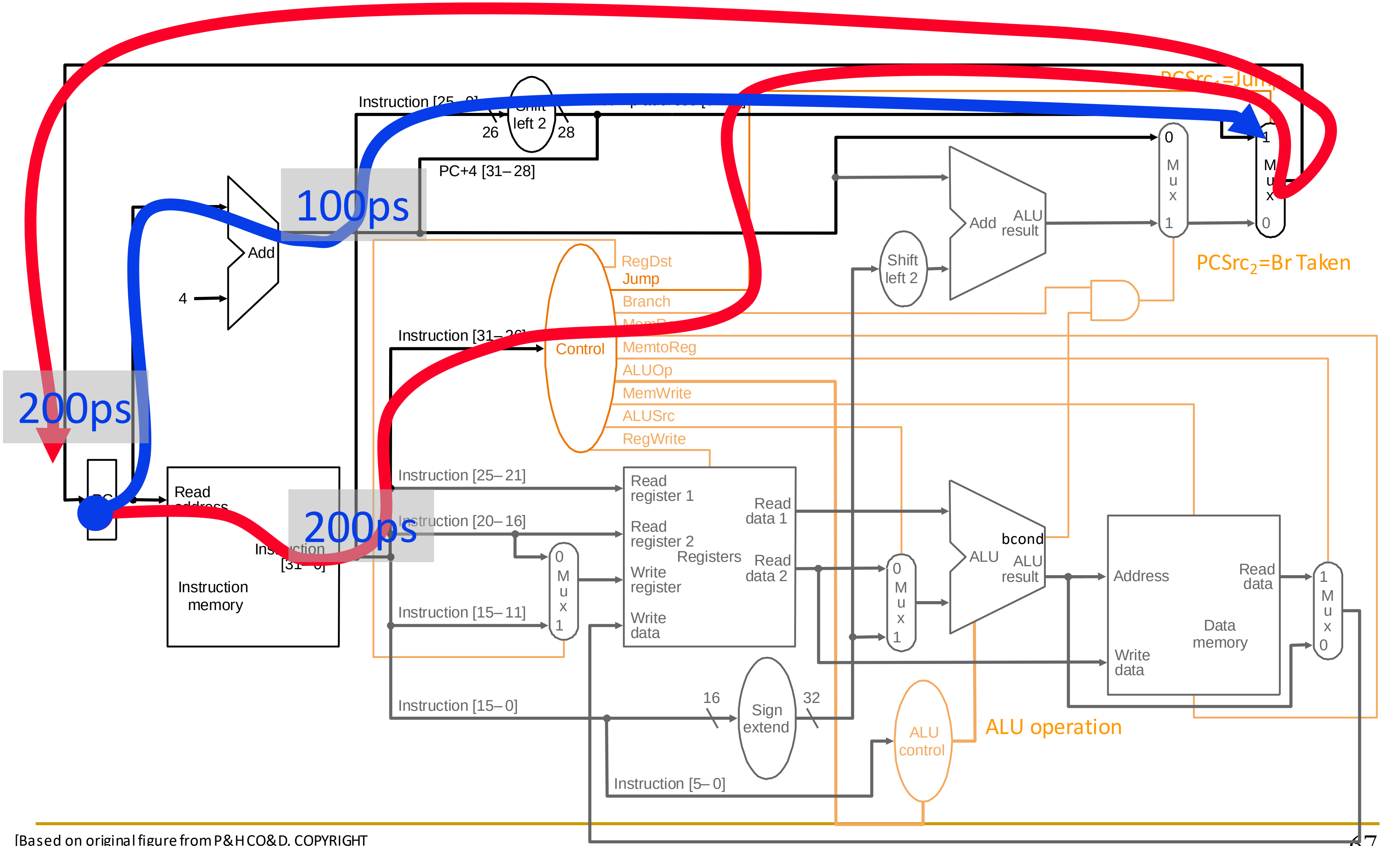
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Branch Taken



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

Jump



[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?
  - Historical example:
    - CDC 5600: control store access too long…

What is the Slowest Instruction to Process?

- Real world: **Memory is slow (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 (x86) or INDEX (VAX)?

- Not easy to optimize/improve performance
  - Optimizing the common case (frequent instructions) does not work
  - Need to optimize the worst case all the time

(Micro)architecture Design Principles

- Critical path design
  - Find and decrease the maximum combinational logic delay
  - Break a path into multiple cycles if it takes too long

- Bread and butter (common case) design
  - Spend time and resources on where it matters most
    - 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
  - Design to eliminate bottlenecks: balance the hardware for the work

Single-Cycle Design vs. Design Principles

- Critical path design

- Bread and butter (common case) design

- Balanced design

How does a single-cycle microarchitecture fare with respect to these principles?

Aside: System Design Principles

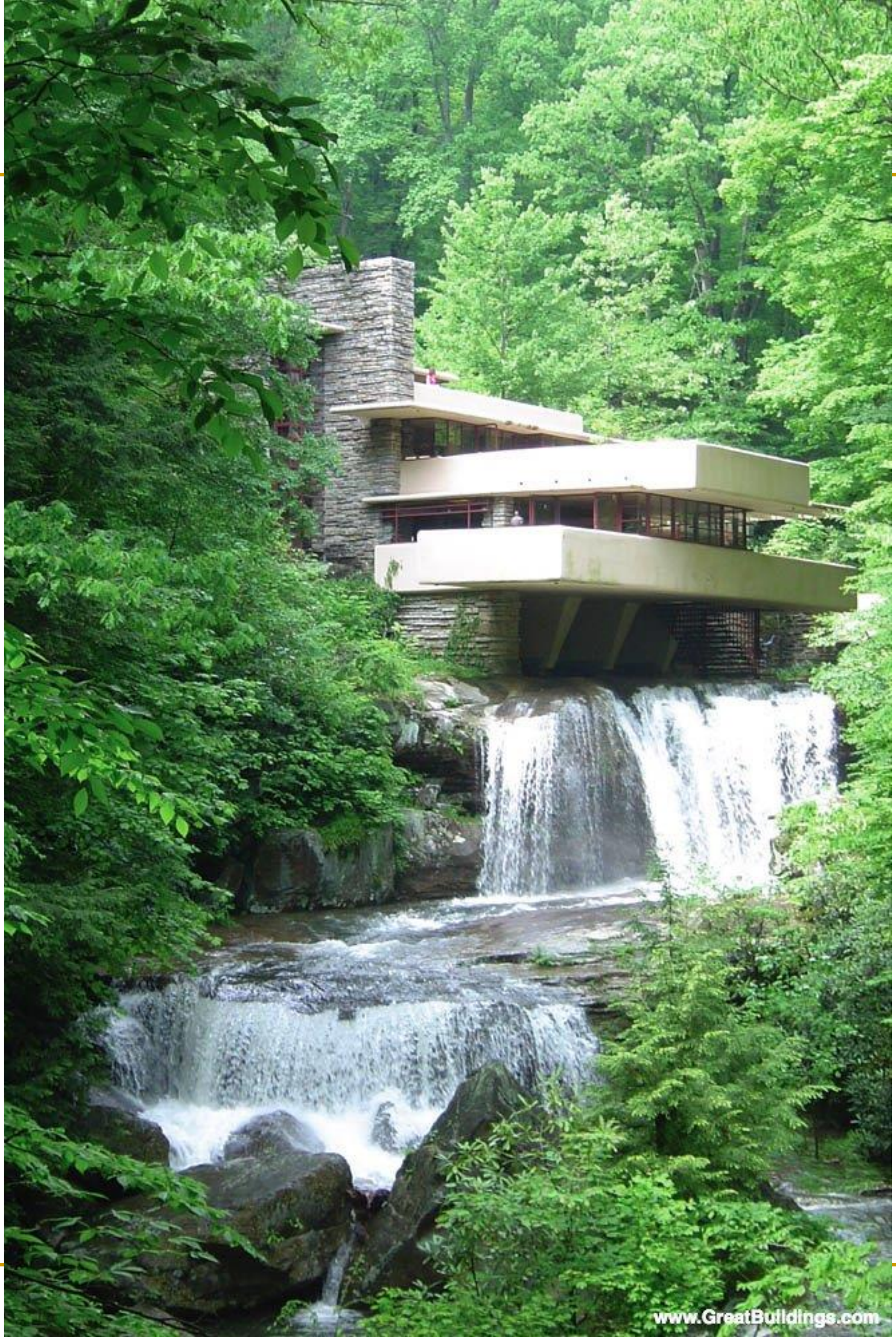
- When designing computer systems/architectures, it is important to follow good principles
  - Actually, this is true for *any* system design
    - Real architectures, buildings, bridges, …
    - Good consumer products
    - Mechanisms for security/safety-critical systems
    - …

- Remember: "principled design" from our second lecture
  - Frank Lloyd Wright: "architecture […] based upon principle, and not upon precedent"

Aside: From Lecture 2

- "architecture […] based upon principle, and not upon precedent"

This



That

Recall: Takeaways

- It all starts from the basic building blocks and design principles

- And, knowledge of how to use, apply, enhance them

- Underlying technology might change (e.g., steel vs. wood)
  - but methods of taking advantage of technology bear resemblance
  - methods used for design depend on the principles employed

Aside: System Design Principles

- We will continue to cover key principles in this course
- Here are some references where you can learn more

- Yale Patt, "Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution," Proc. of IEEE, 2001. (Levels of transformation, design point, etc)

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966. (Flynn's Bottleneck → Balanced design)

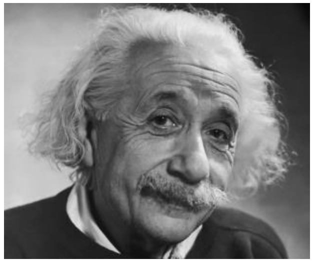
- Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS Conference, April 1967. (Amdahl's Law → Common-case design)

- Butler W. Lampson, "Hints for Computer System Design," ACM Operating Systems Review, 1983.

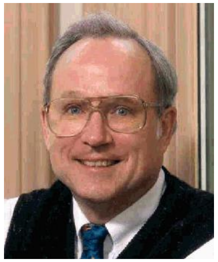
A Key System Design Principle

- Keep it simple

- "Everything should be made as simple as possible, but no simpler."
  - Albert Einstein



- And, keep it low cost: "An engineer is a person who can do for a dime what any fool can do for a dollar."



- For more, see:
  - Butler W. Lampson, "Hints for Computer System Design," ACM Operating Systems Review, 1983.
  - http://research.microsoft.com/pubs/68221/acrobat.pdf

Can We Do Better?

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

Recall: 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')

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

Downsides of Multi-Cycle Design

- Need to store the intermediate results at the end of each clock cycle
  - Hardware overhead for registers
  - Register setup/hold overhead paid multiple times for an instruction

Remember: 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

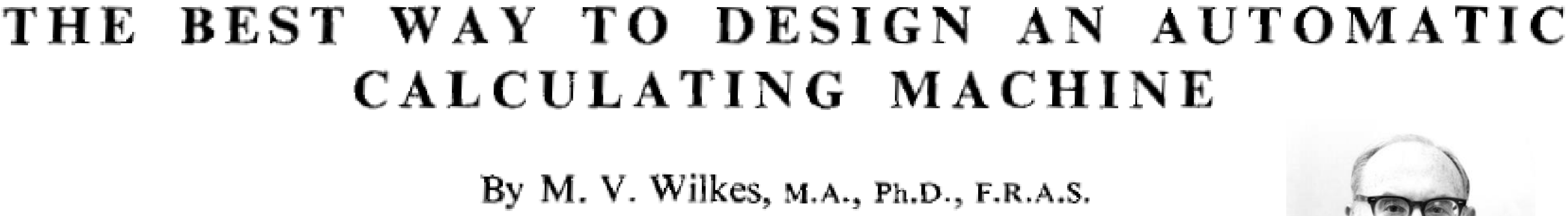
In multi-cycle, we have two degrees of freedom to optimize independently

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 BEST WAY TO DESIGN AN AUTOMATIC CALCULATING MACHINE

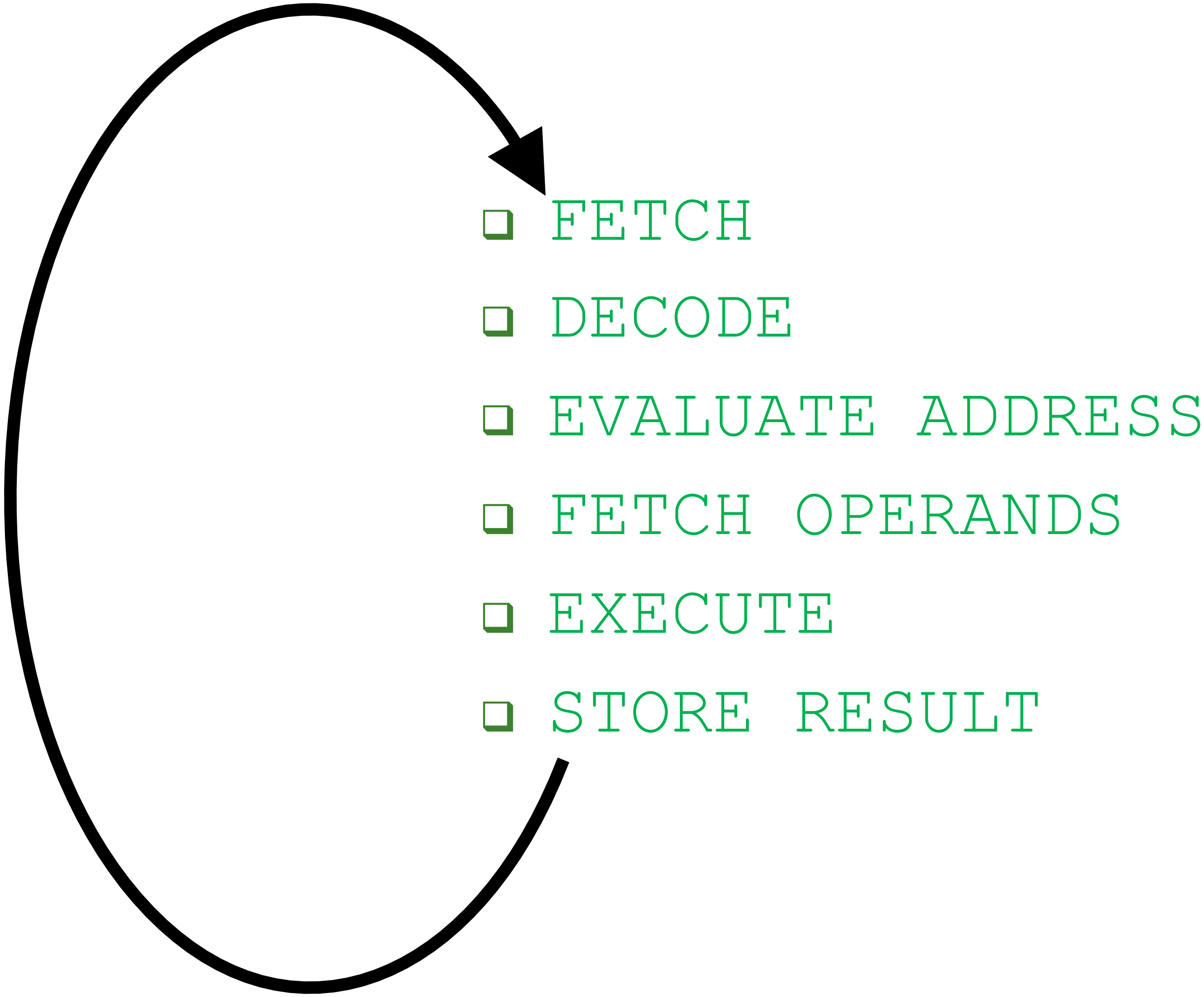
By M. V. Wilkes, M.A., Ph.D., F.R.A.S.

- An elegant implementation:
  - The concept of microcoded/microprogrammed machines

Multi-Cycle Microarchitectures

- Key Idea for 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 are determined in current state

Recall: 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

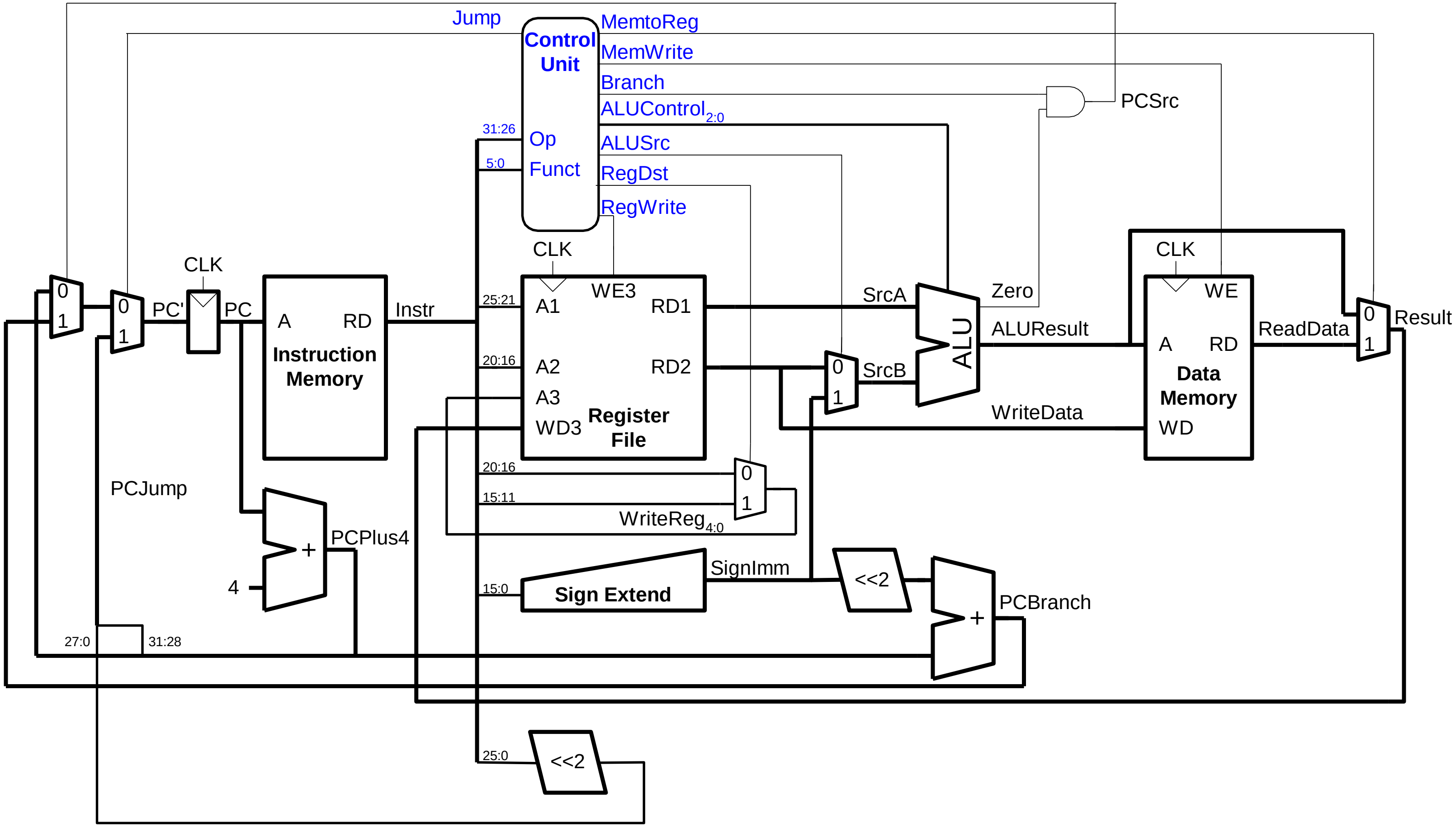
- A multi-cycle microarchitecture sequences from state to state to process an instruction
  - The behavior of the machine in a state is completely determined by control signals in that state

- The behavior of the entire processor is specified fully by a *finite state machine*

- In a state (clock cycle), control signals control two things:
  - How the datapath should process the data
  - How to generate the control signals for the (next) clock cycle

One Example Multi-Cycle Microarchitecture

Remember: Single-Cycle MIPS Processor



Multi-Cycle MIPS Processor

- **Single-cycle microarchitecture:**
  - cycle time limited by longest instruction (`lw`) → low clock frequency
  - three adders/ALUs and two memories → high hardware cost

- **Multi-cycle microarchitecture:**
  - + higher clock frequency
  - + simpler instructions take few clock cycles
  - + reuse expensive hardware across multiple cycles
  - - sequencing overhead paid many times
  - - hardware overhead for storing intermediate results

- **Multi-cycle requires the same design steps as single cycle:**
  - datapath
  - control logic

What Do We Want To Optimize?

- **Single-cycle microarchitecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (lower cost)

What Do We Want To Optimize?

- **Single-cycle microarchitecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (lower cost)

- **Single-cycle microarchitecture needs three adders**
  - ALU, PC, Branch address calculation
  - We want to use only one ALU for all operations (lower cost)

What Do We Want To Optimize?

- **Single-cycle microarchitecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (lower cost)

- **Single-cycle microarchitecture needs three adders**
  - ALU, PC, Branch address calculation
  - We want to use only one ALU for all operations (lower cost)

- **Single-cycle microarchitecture: each instruction takes one cycle**
  - The slowest instruction slows down every single instruction
  - We want to determine clock cycle time independently of instruction processing time
    - Divide each instruction into multiple clock cycles
    - Simpler instructions can be very fast (compared to the slowest)

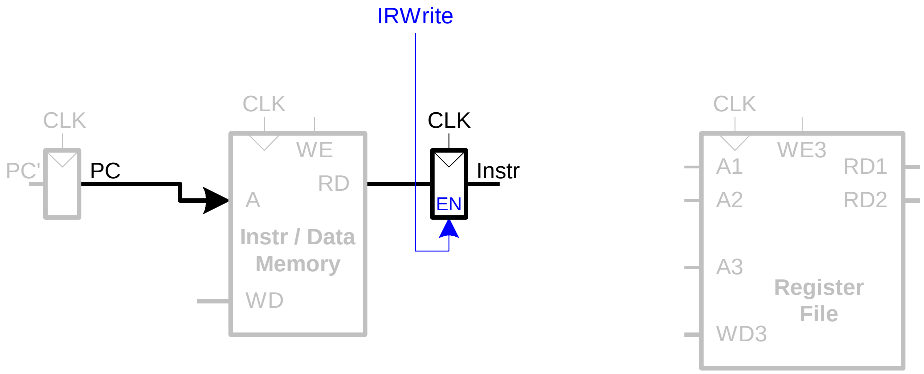
Let's Construct the Multi-Cycle Datapath

Consider the lw Instruction

- **For an instruction such as: `lw $t0, 0x20($t1)`**
- **We need to:**
  - Read the instruction from memory
  - Then read **`$t1`** from register array
  - Add the immediate value (**`0x20`**) to calculate the memory address
  - Read the content of this address
  - Write to the register **`$t0`** this content

Multi-Cycle Datapath: Instruction Fetch

- **We will consider lw, but fetch is the same for all instructions**
  - STEP 1: Fetch instruction

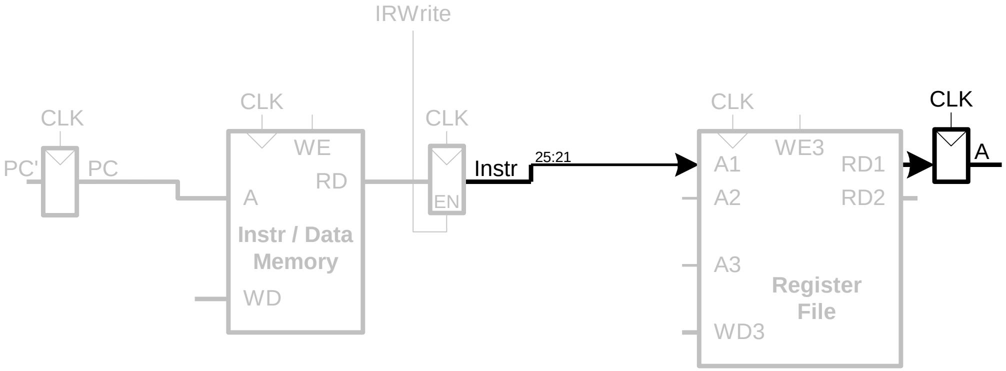


read from the memory location [rs]+imm to location [rt]

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

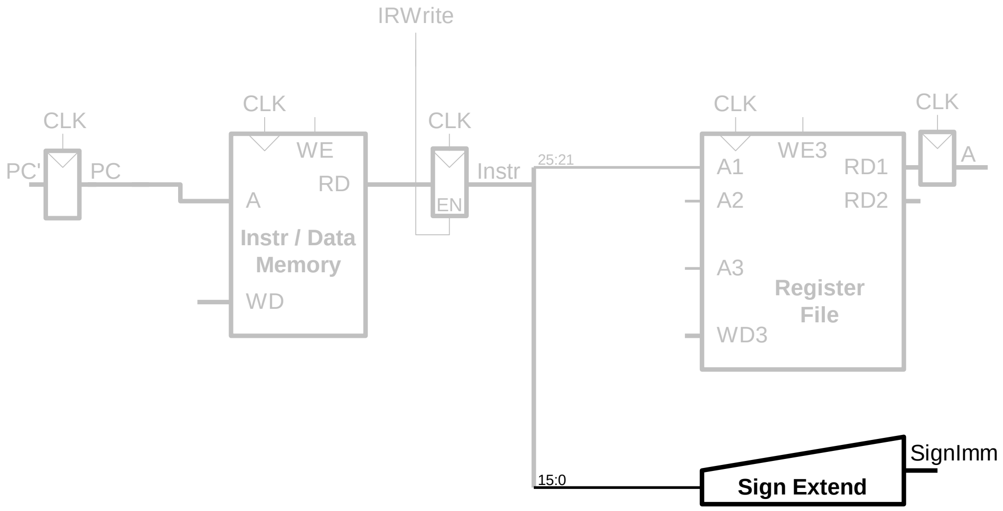
Multi-Cycle Datapath: `lw` register read



I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

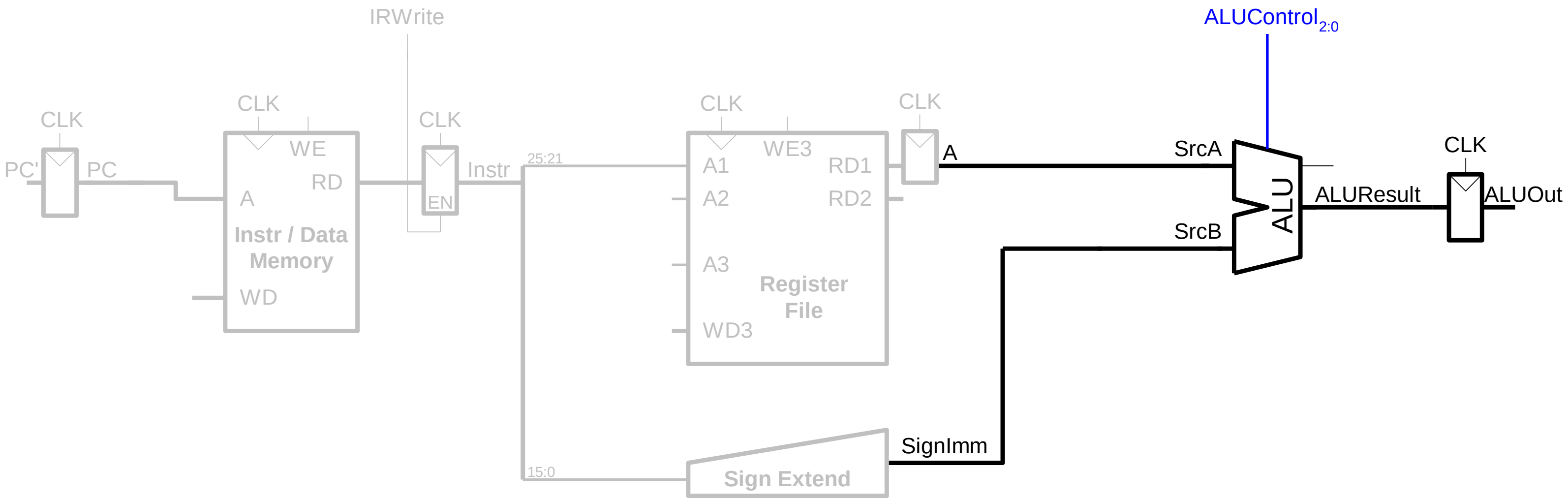
Multi-Cycle Datapath: `lw` immediate



I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

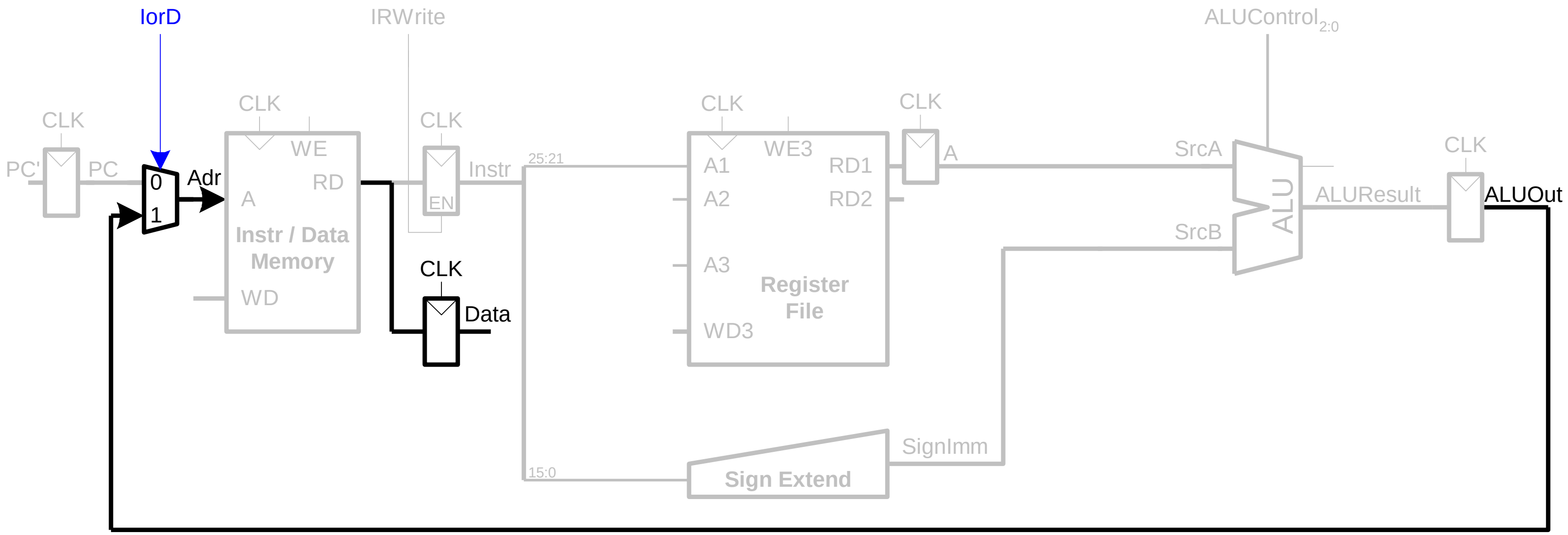
Multi-Cycle Datapath: `lw` address



I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

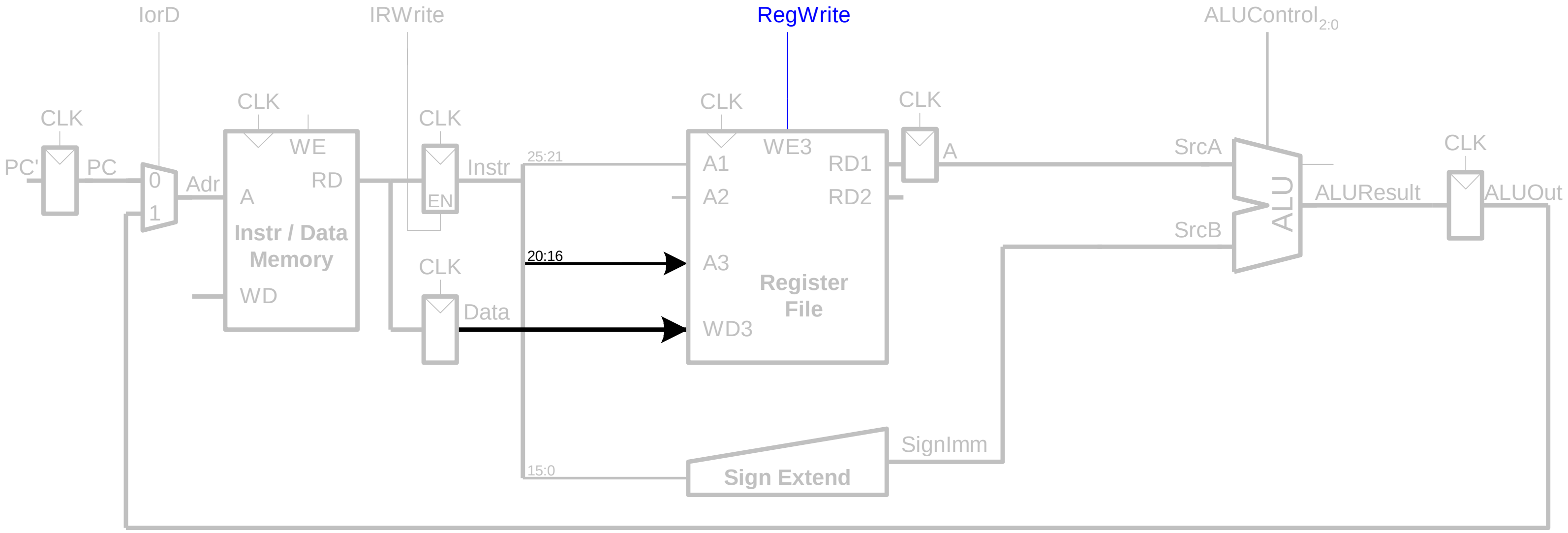
Multi-Cycle Datapath: `lw` memory read



I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

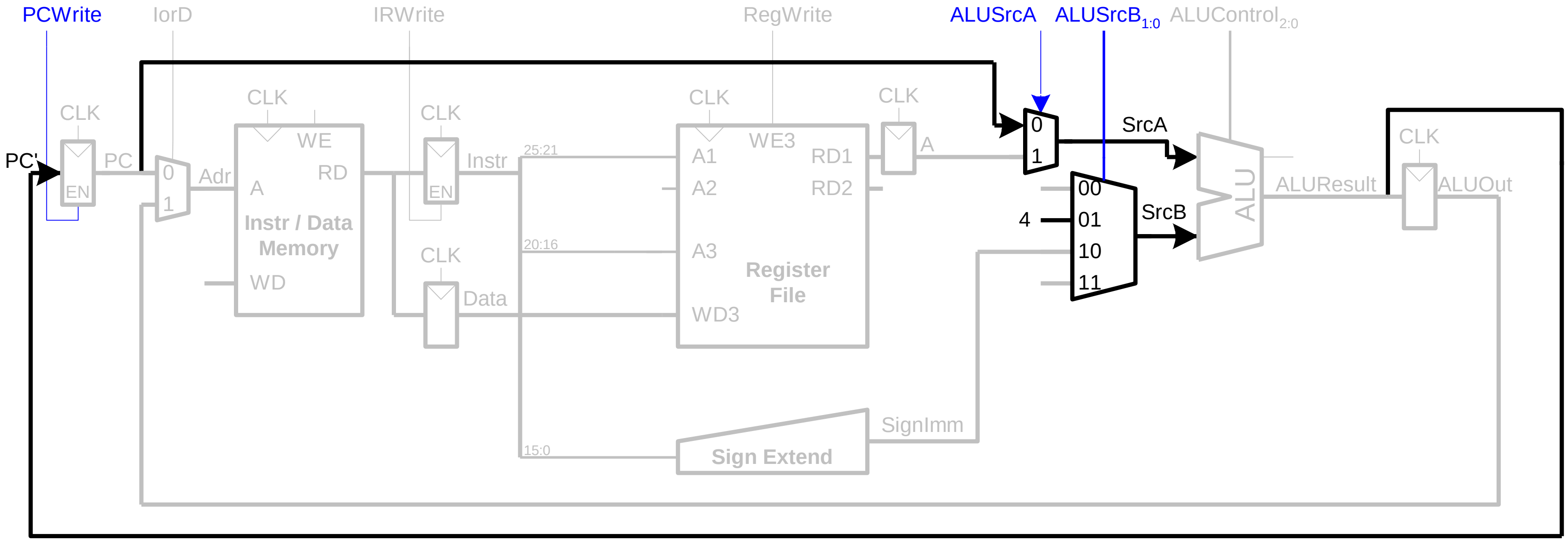
Multi-Cycle Datapath: `lw` write register



I-Type

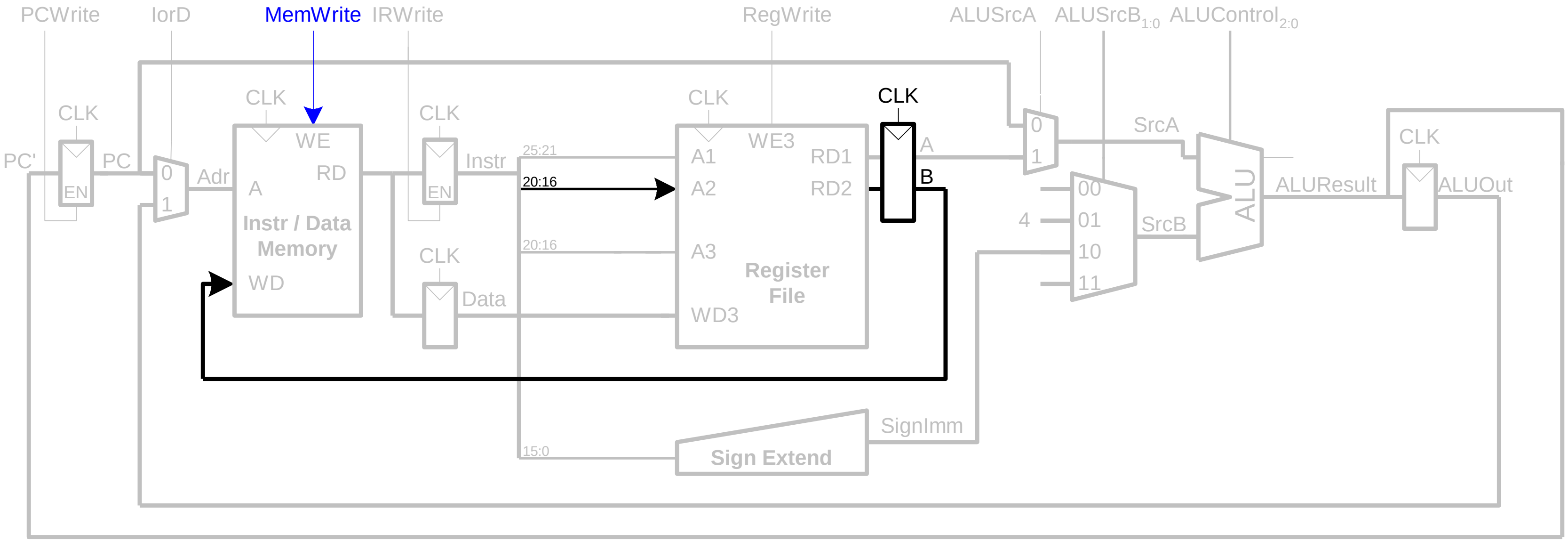
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Multi-Cycle Datapath: increment PC



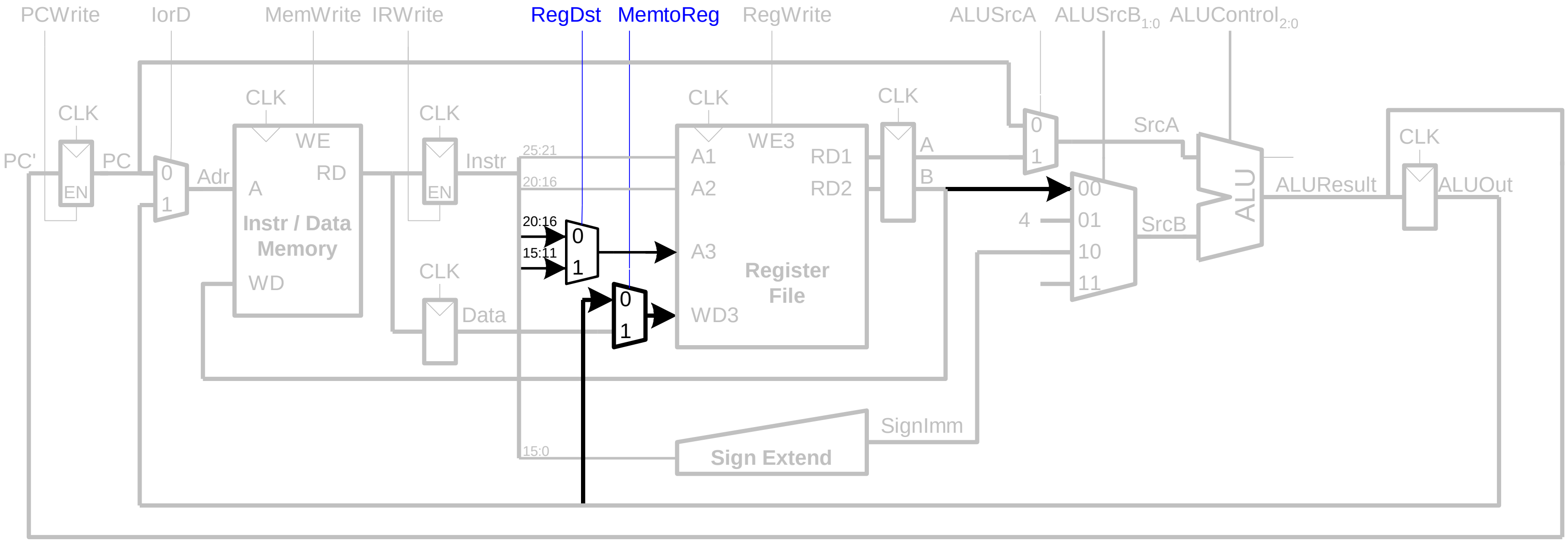
Multi-Cycle Datapath: sw

- **Write data in rt to memory**



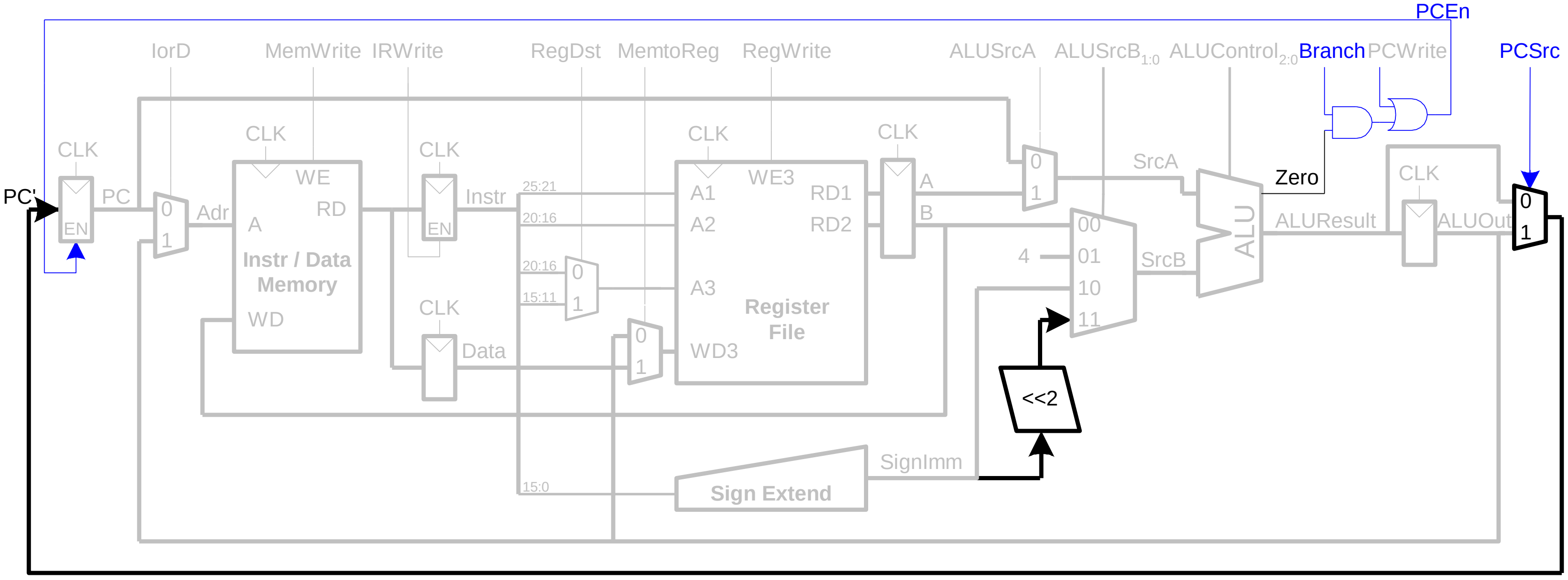
Multi-Cycle Datapath: R-type Instructions

- **Read from rs and rt**
  - Write ALUResult to register file
  - Write to rd (instead of rt)

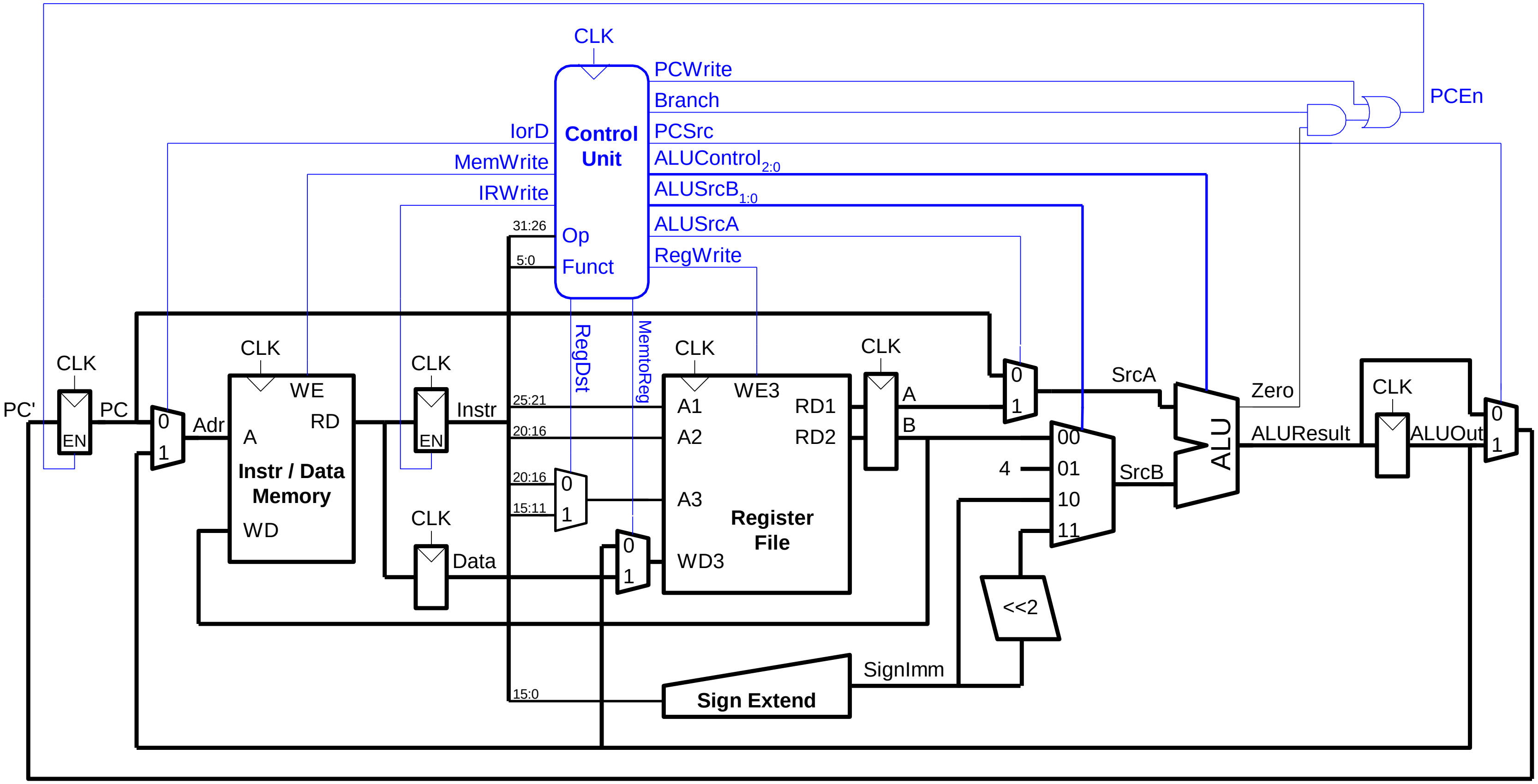


Multi-Cycle Datapath: beq

- **Determine whether values in rs and rt are equal**
  - Calculate branch target address:
    Target Address = (sign-extended immediate << 2) + (PC+4)

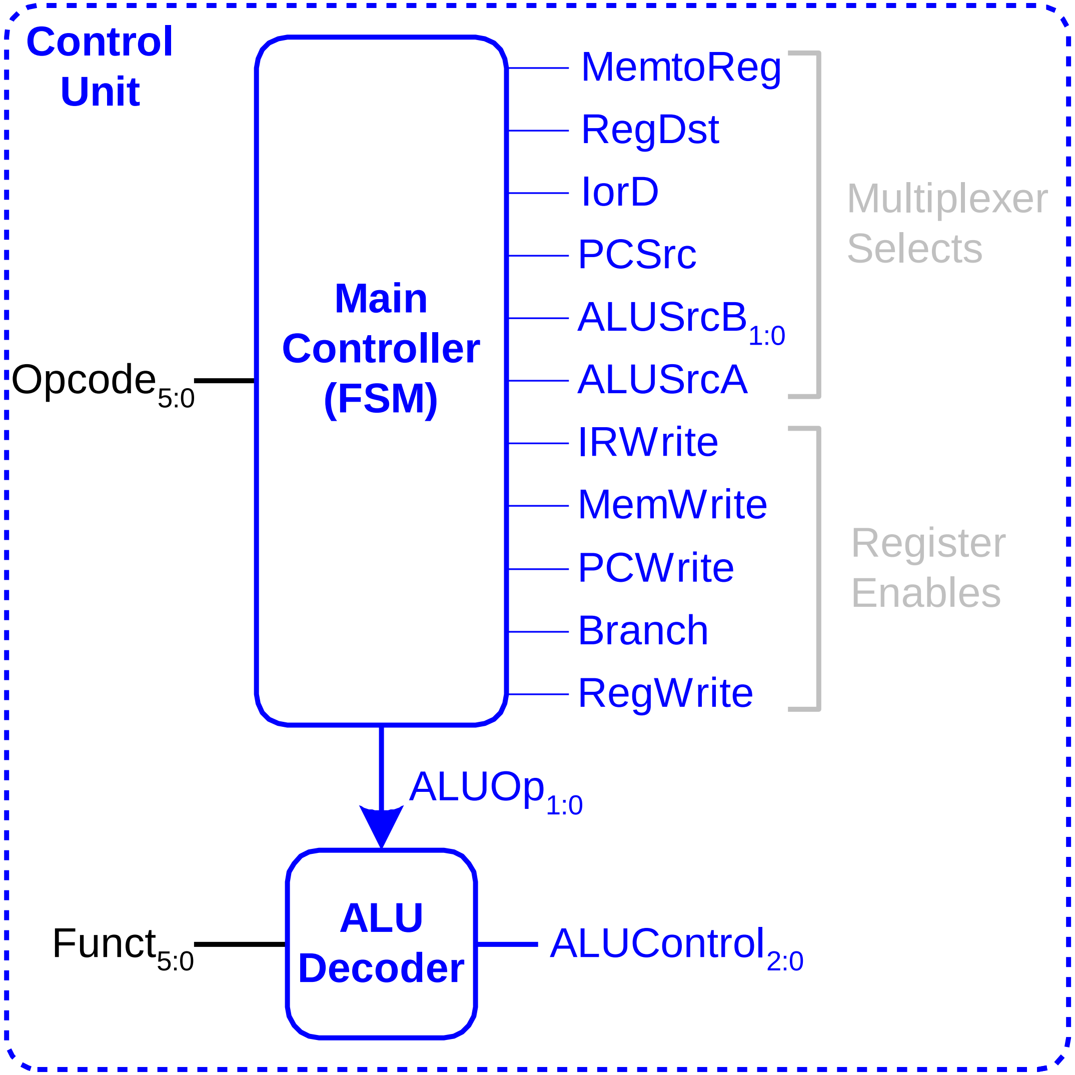


Complete Multi-Cycle Processor

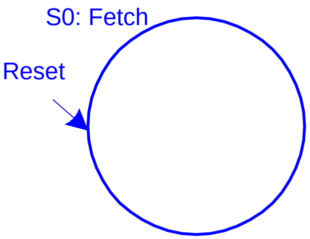


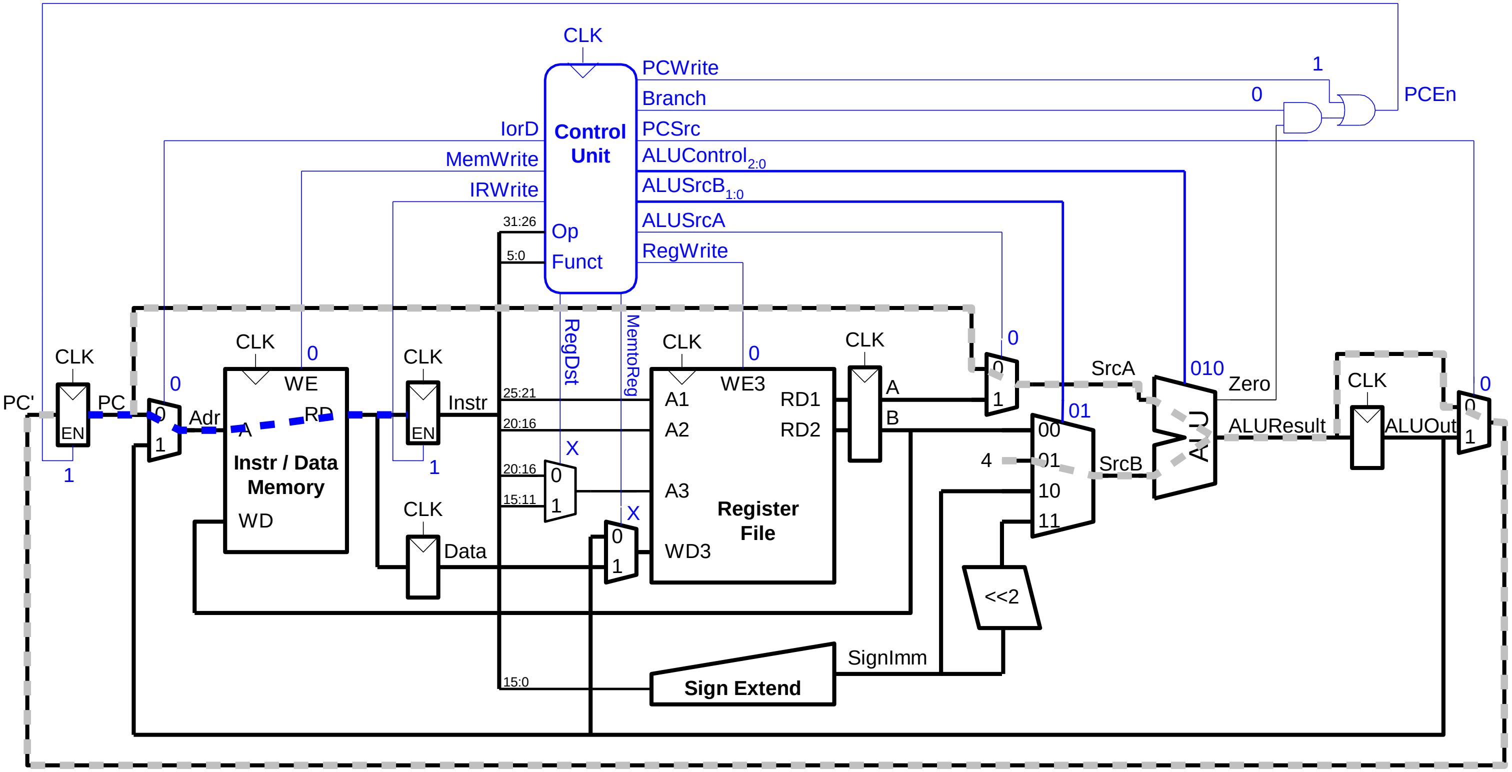
Let’s Construct the Multi-Cycle Control Logic

Control Unit

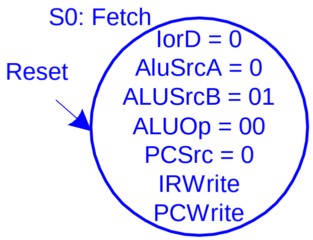


Main Controller FSM: Fetch





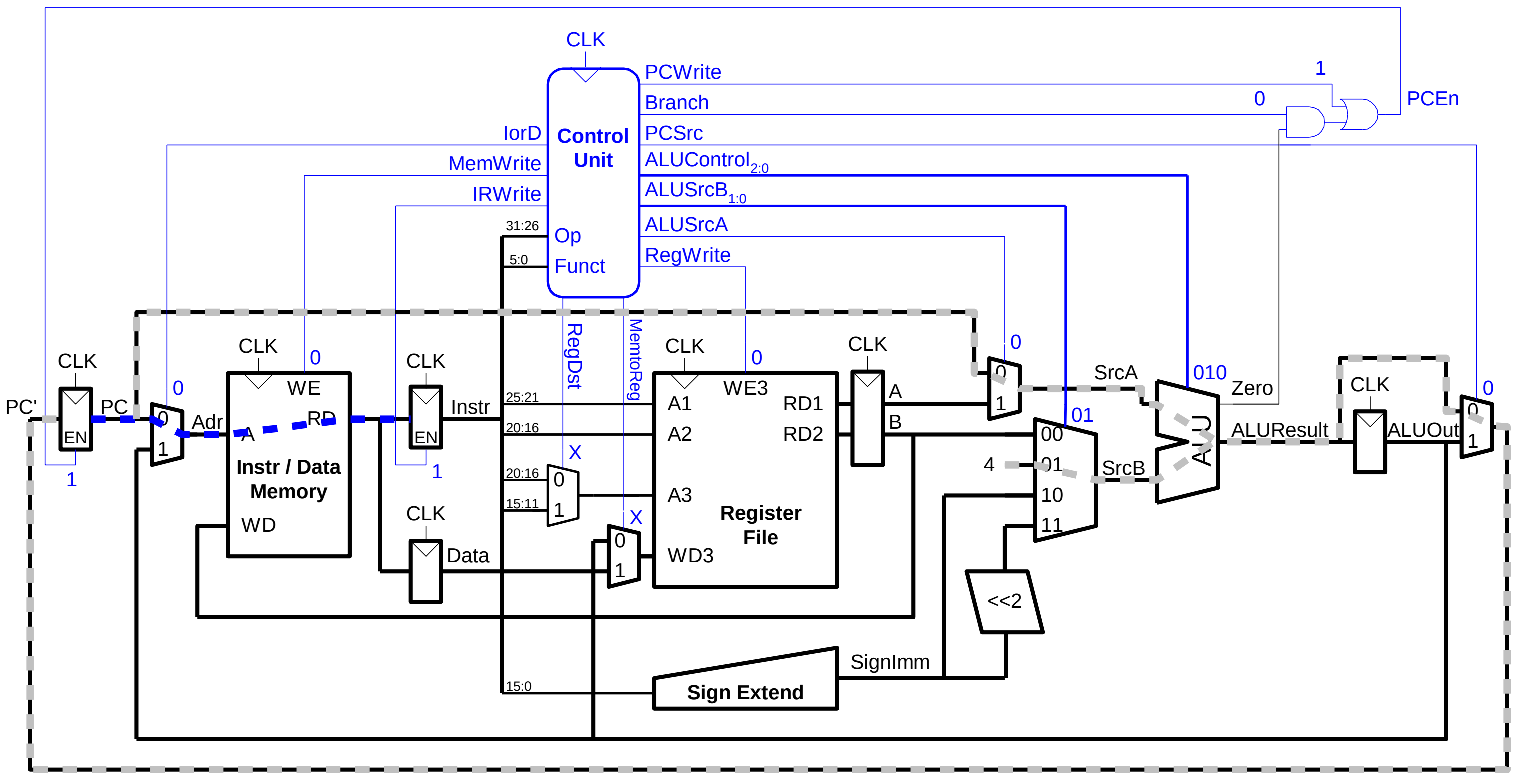
Main Controller FSM: Fetch



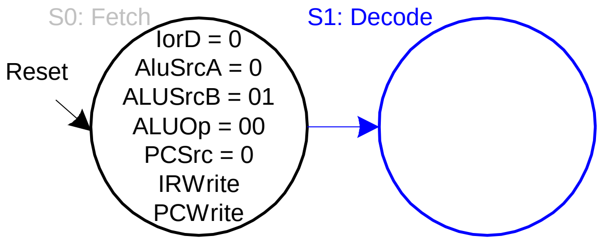
S0: Fetch

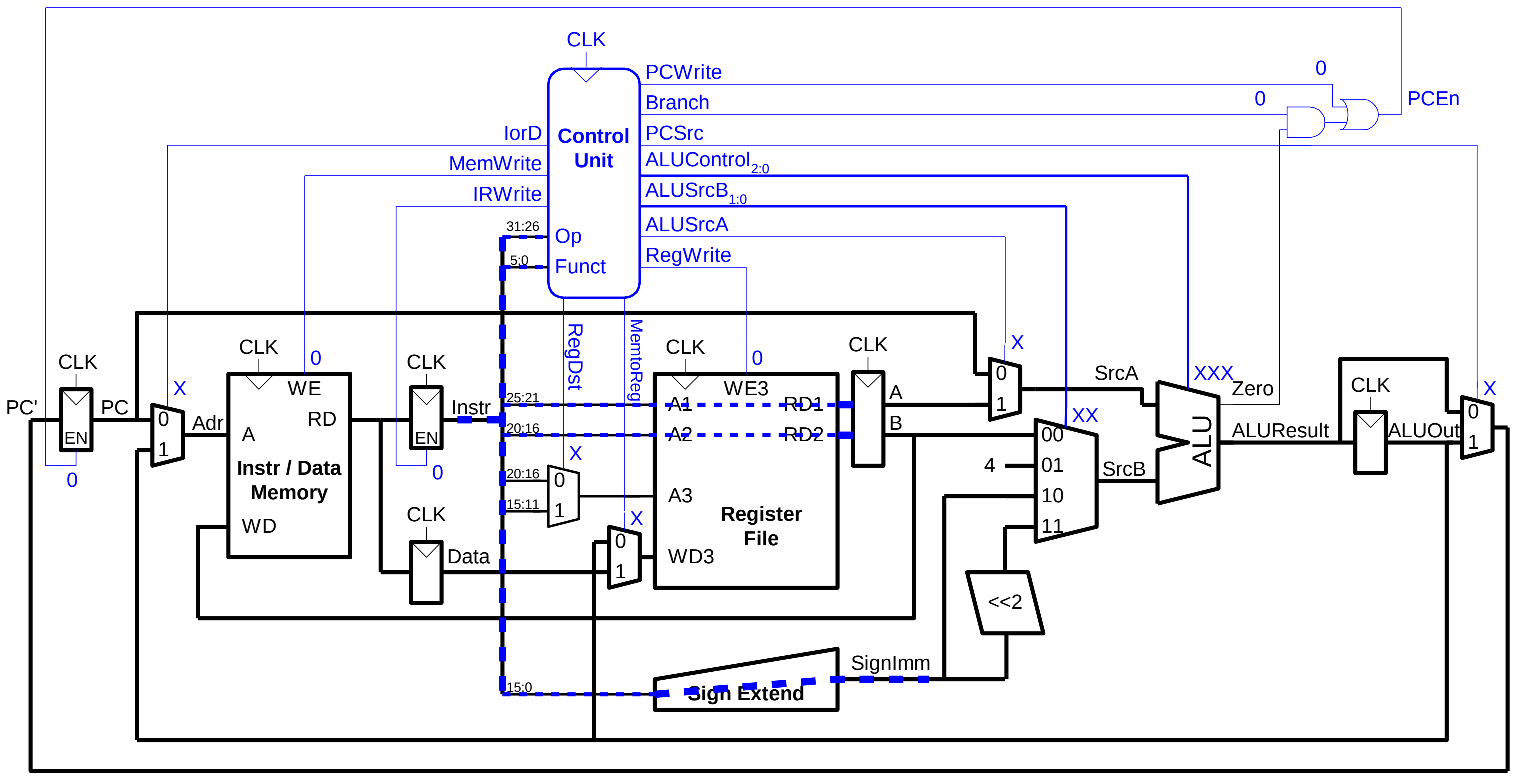
Reset

IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

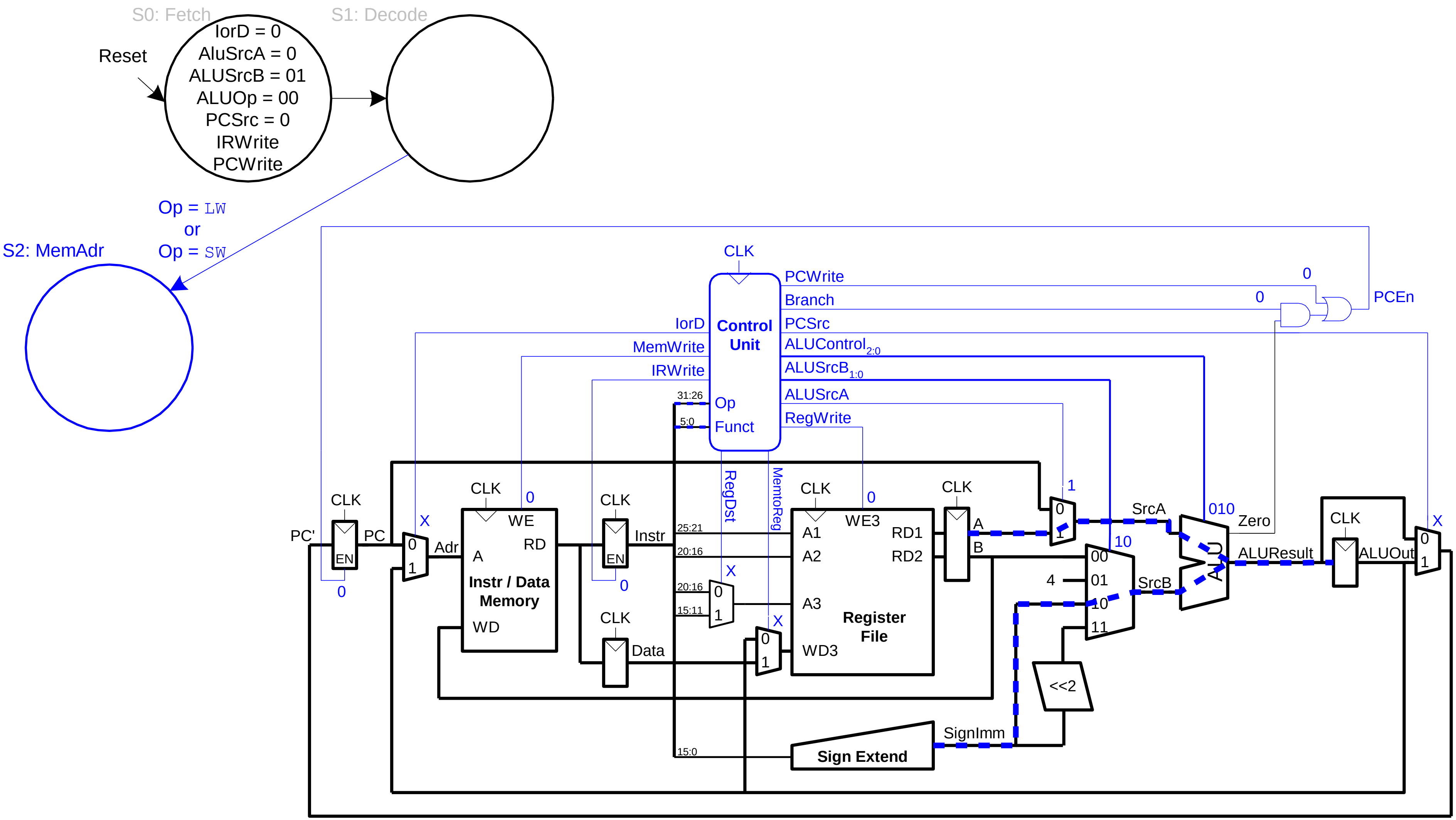


Main Controller FSM: Decode

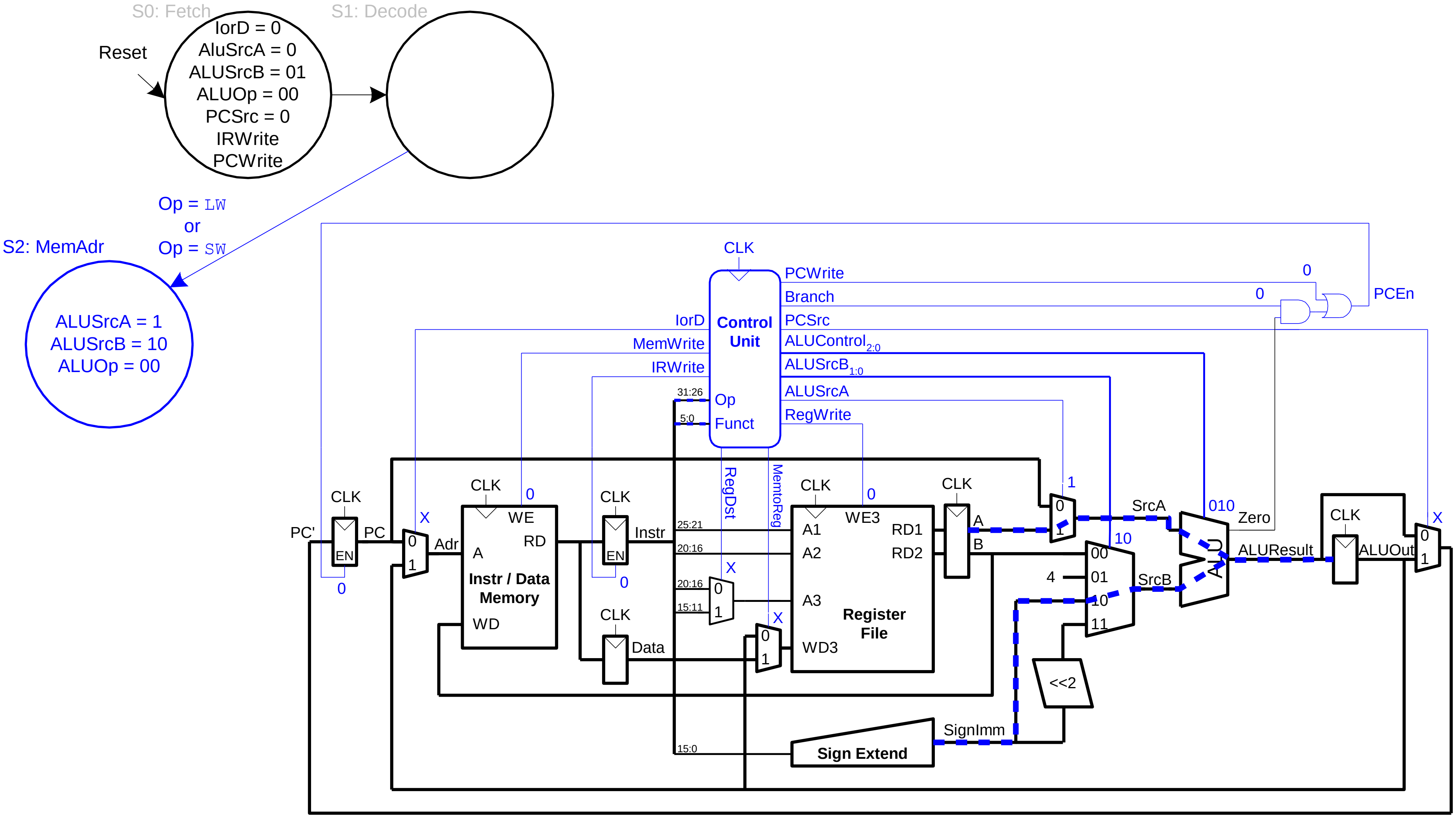




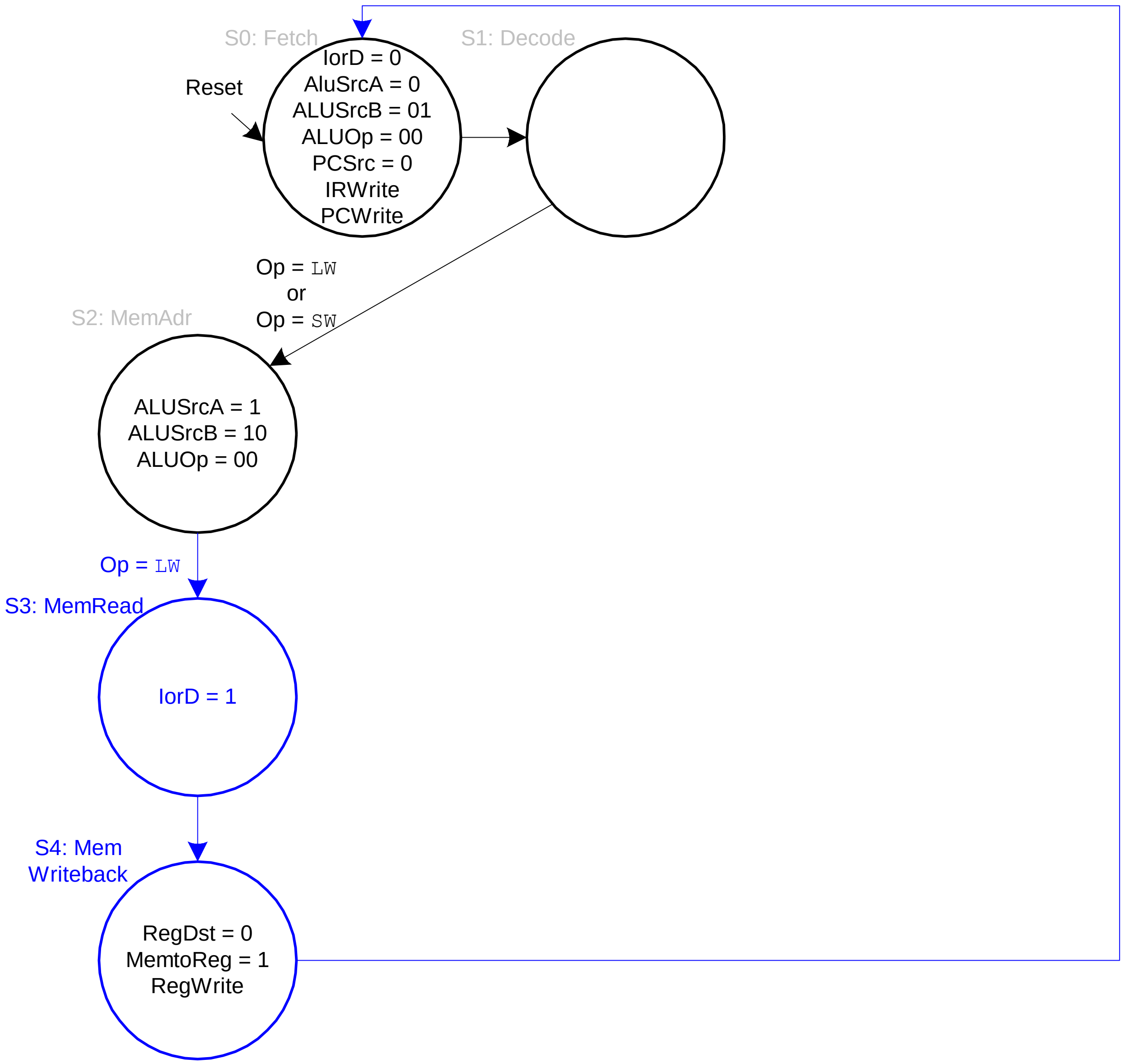
Main Controller FSM: Address Calculation



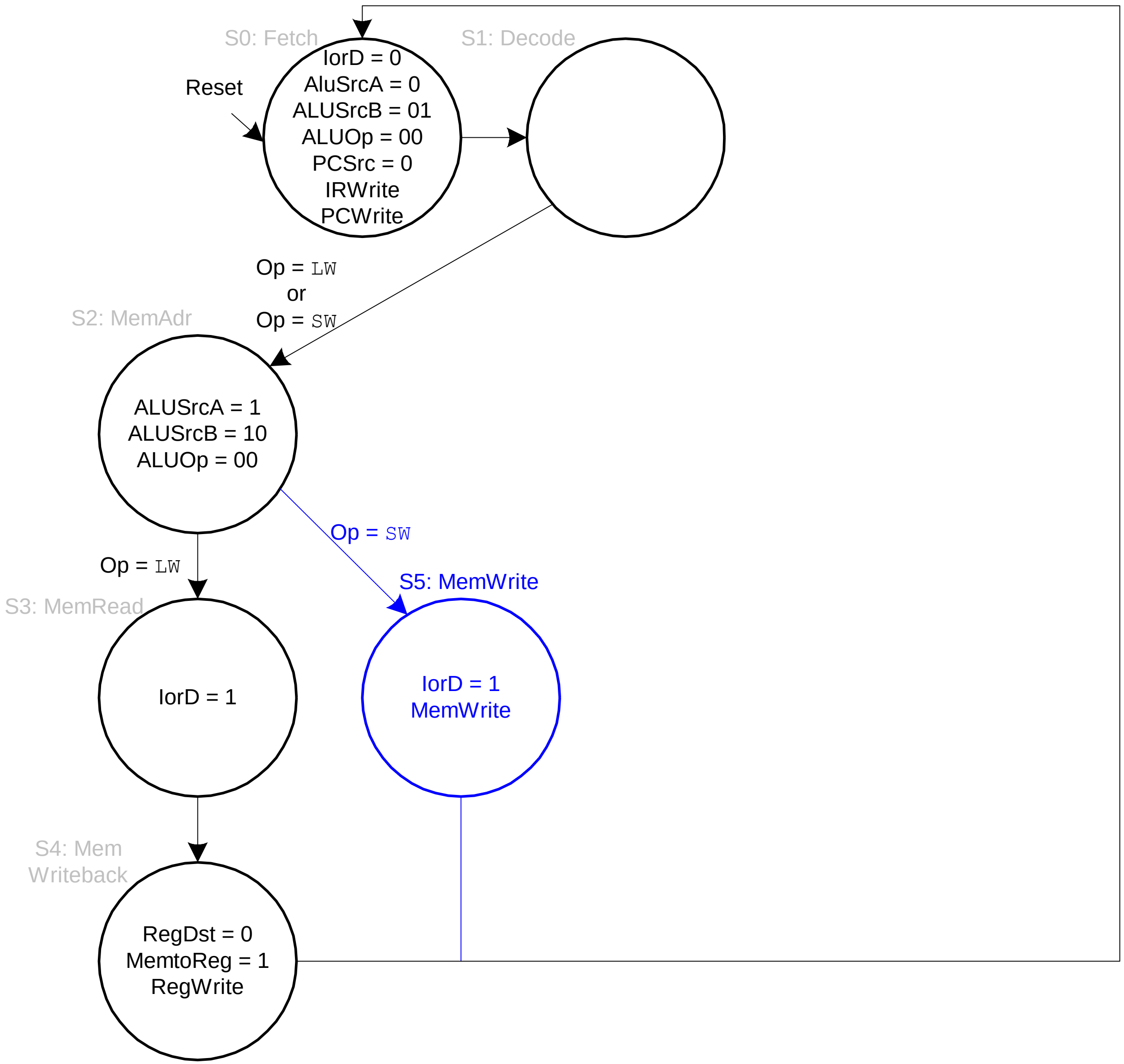
Main Controller FSM: Address Calculation



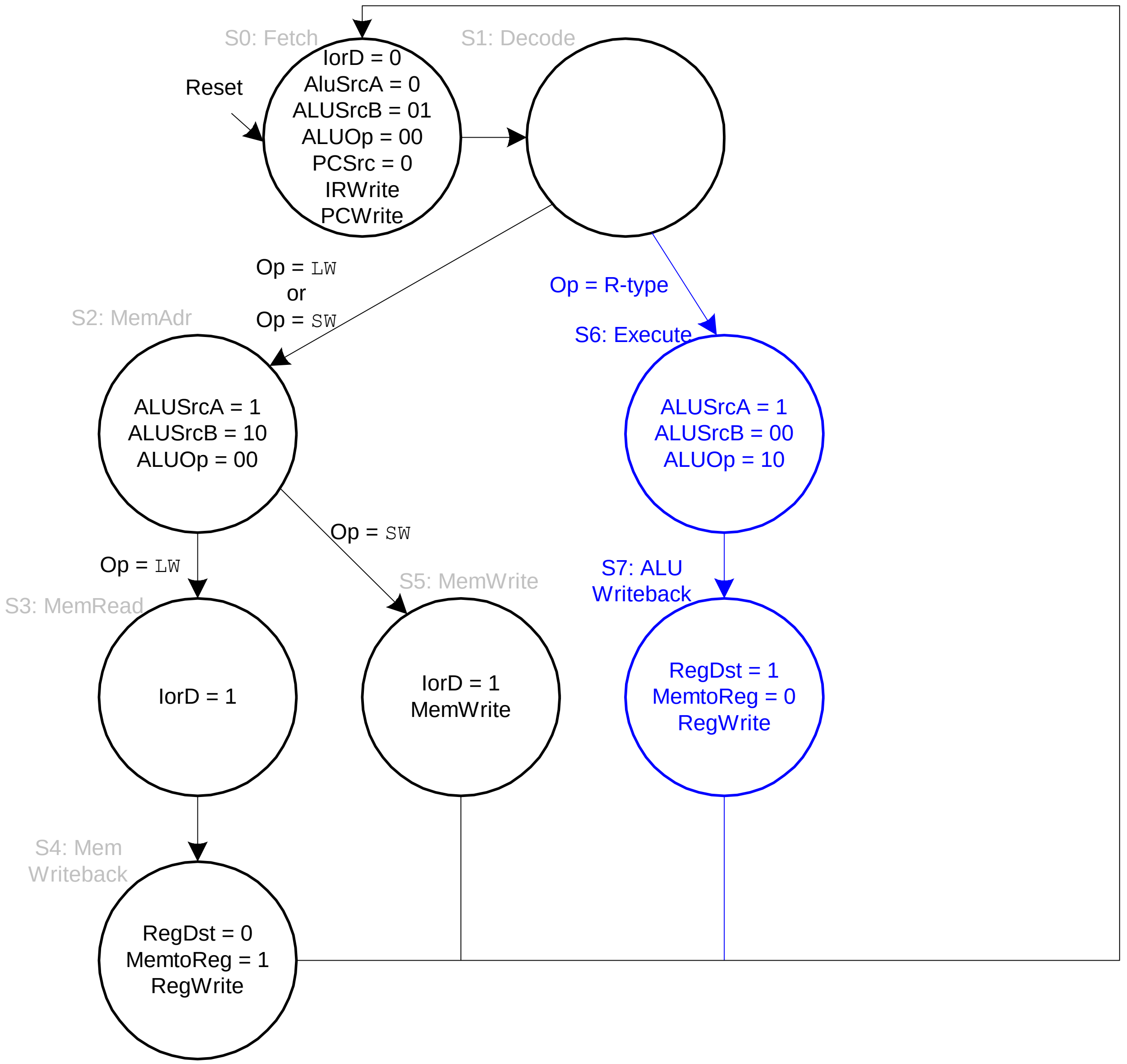
Main Controller FSM: `lw`



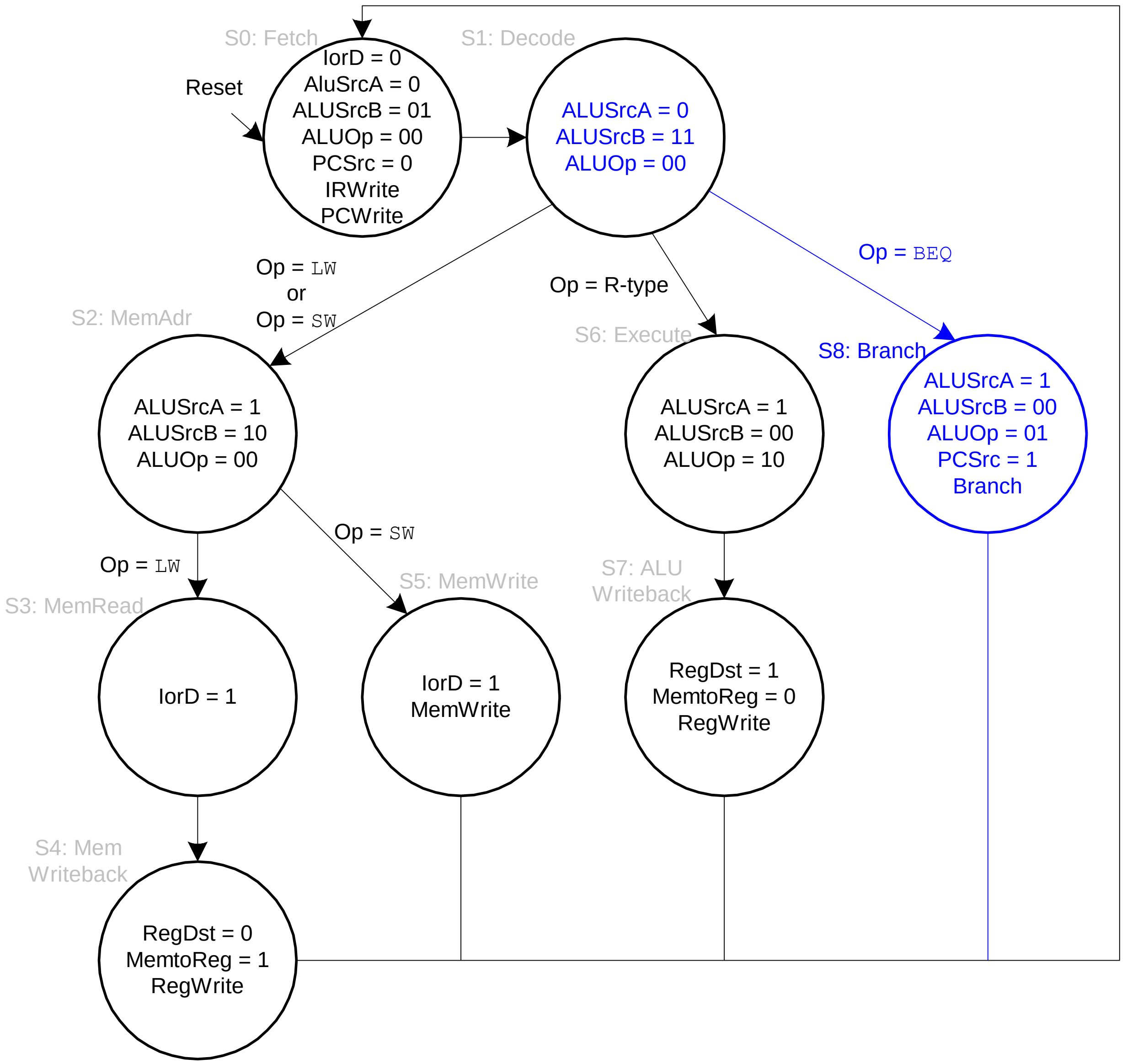
Main Controller FSM: sw



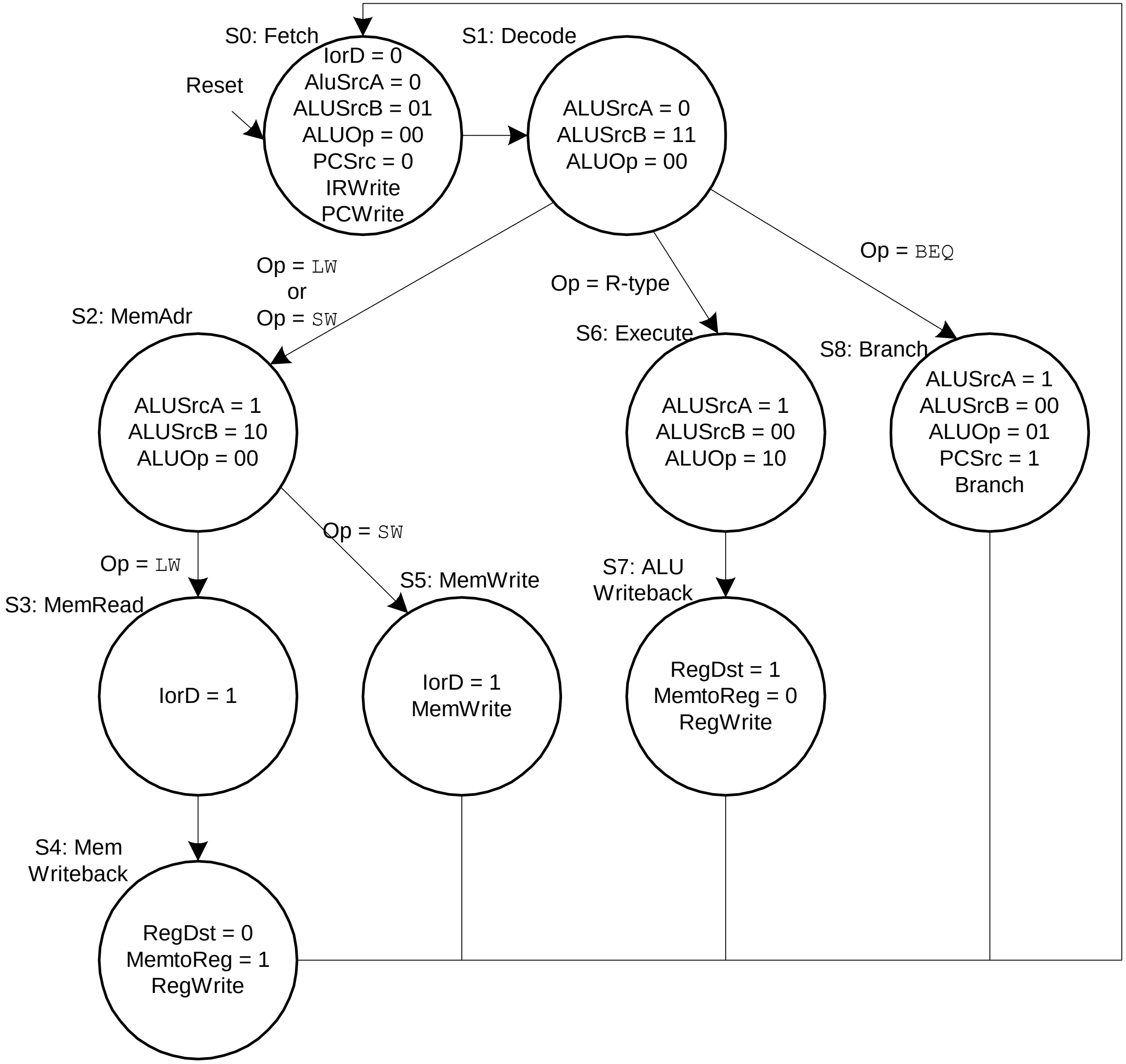
Main Controller FSM: R-Type



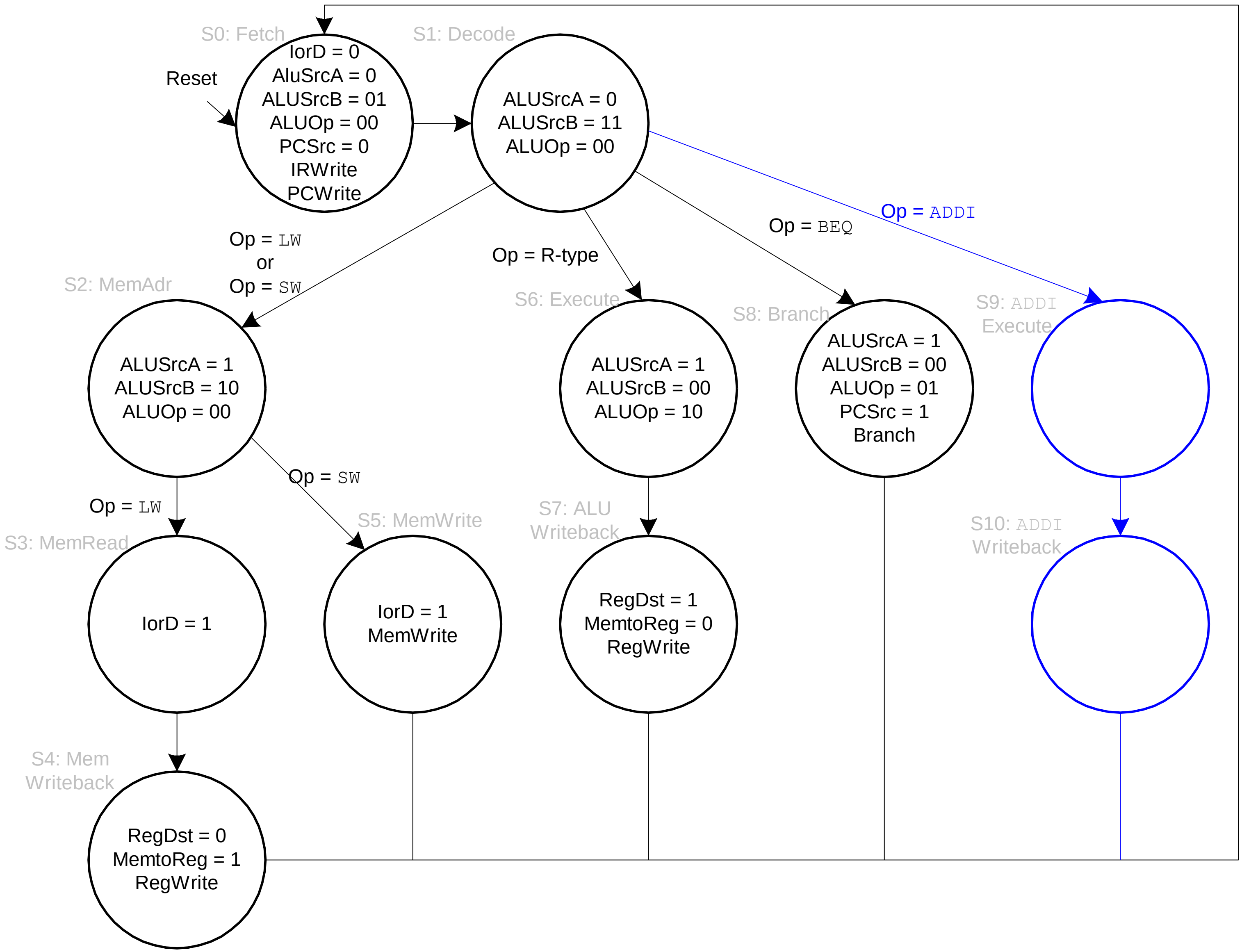
Main Controller FSM: beq



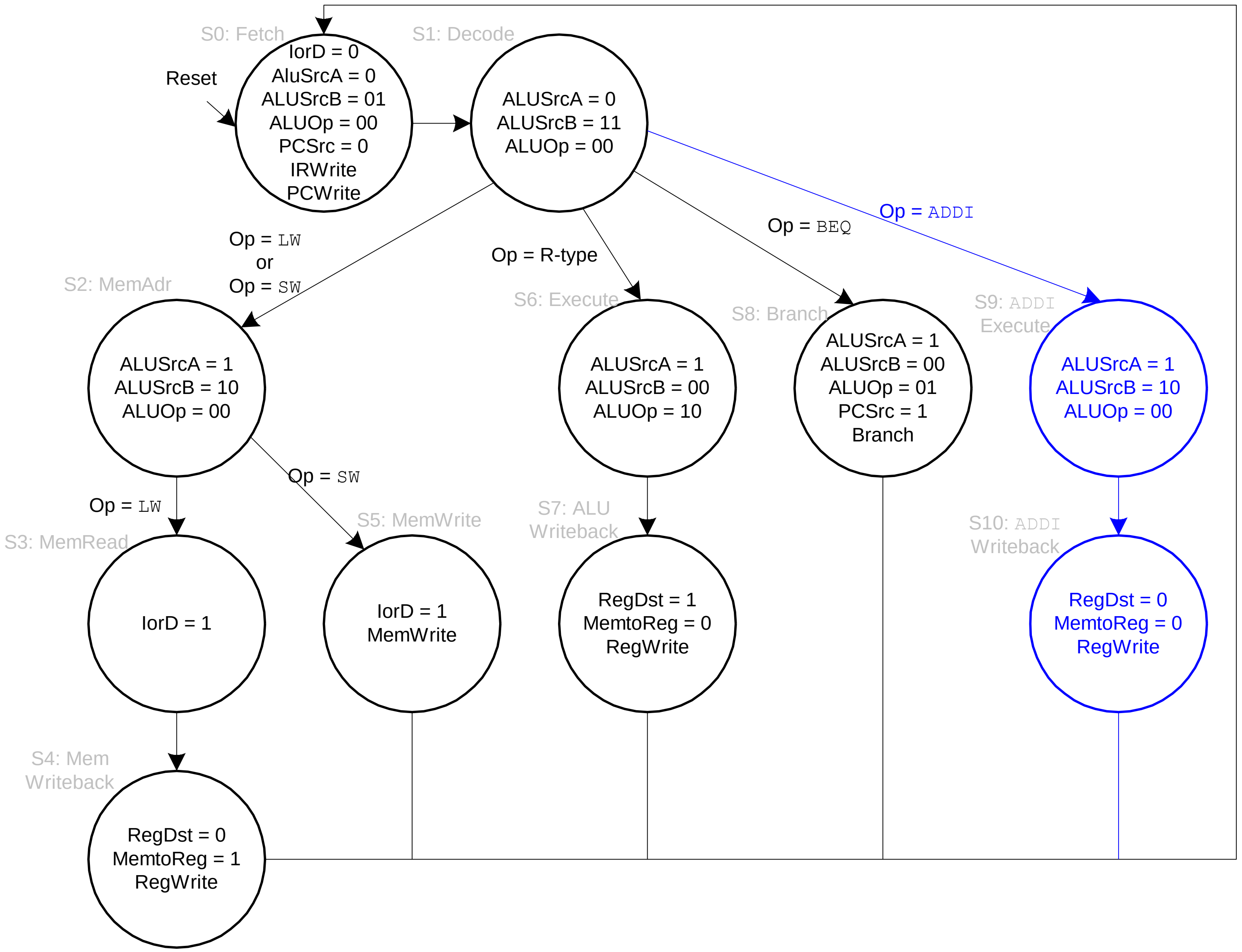
Complete Multi-Cycle Controller FSM



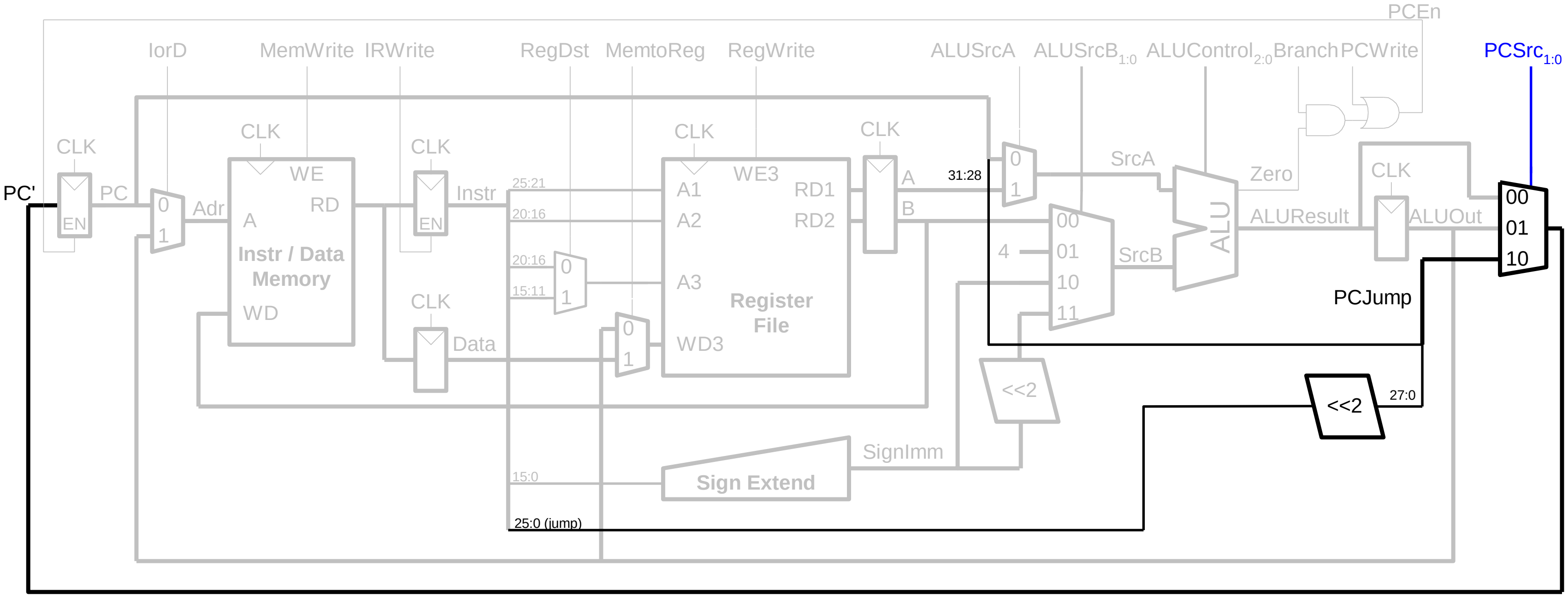
Main Controller FSM: addi



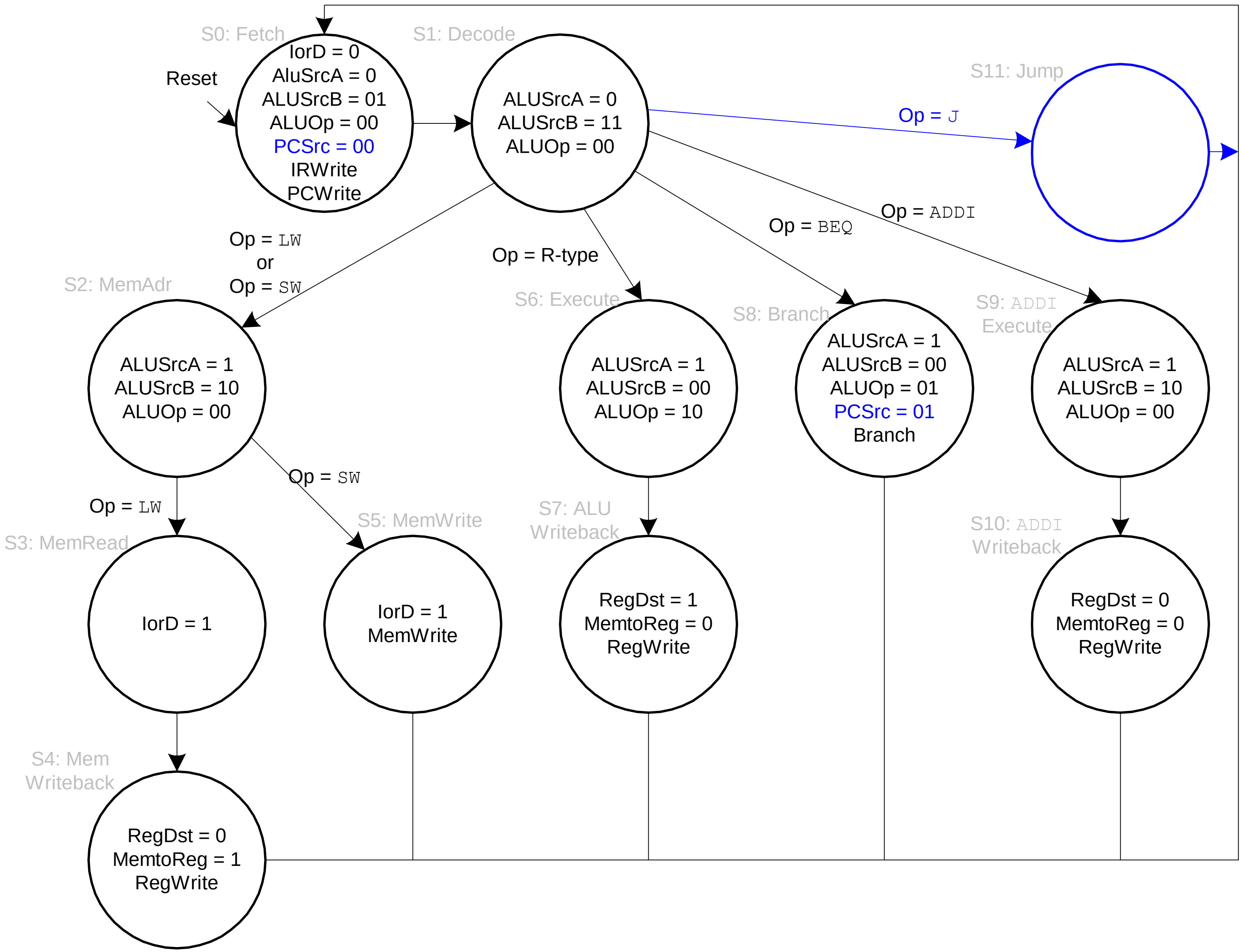
Main Controller FSM: addi



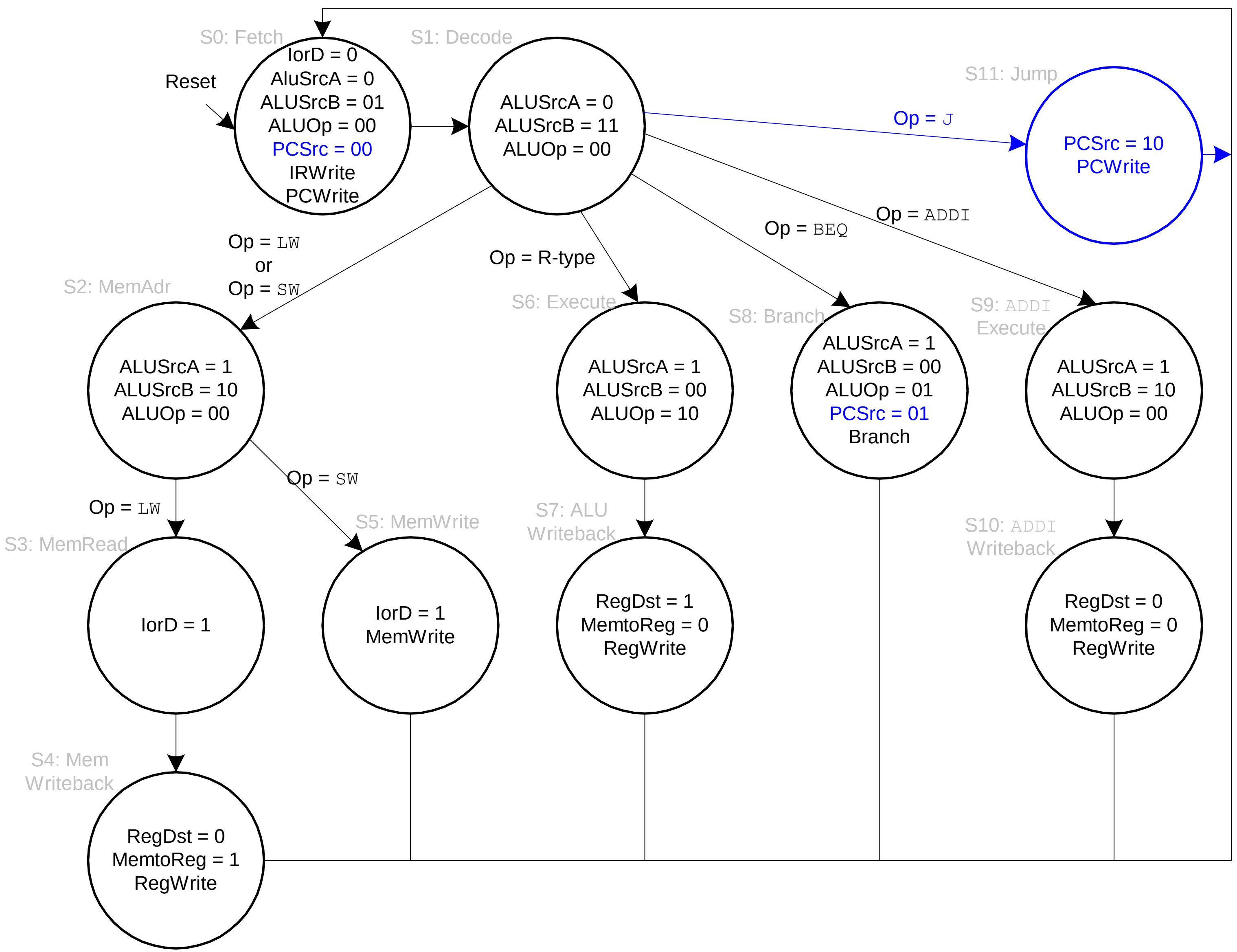
Extended Functionality: j



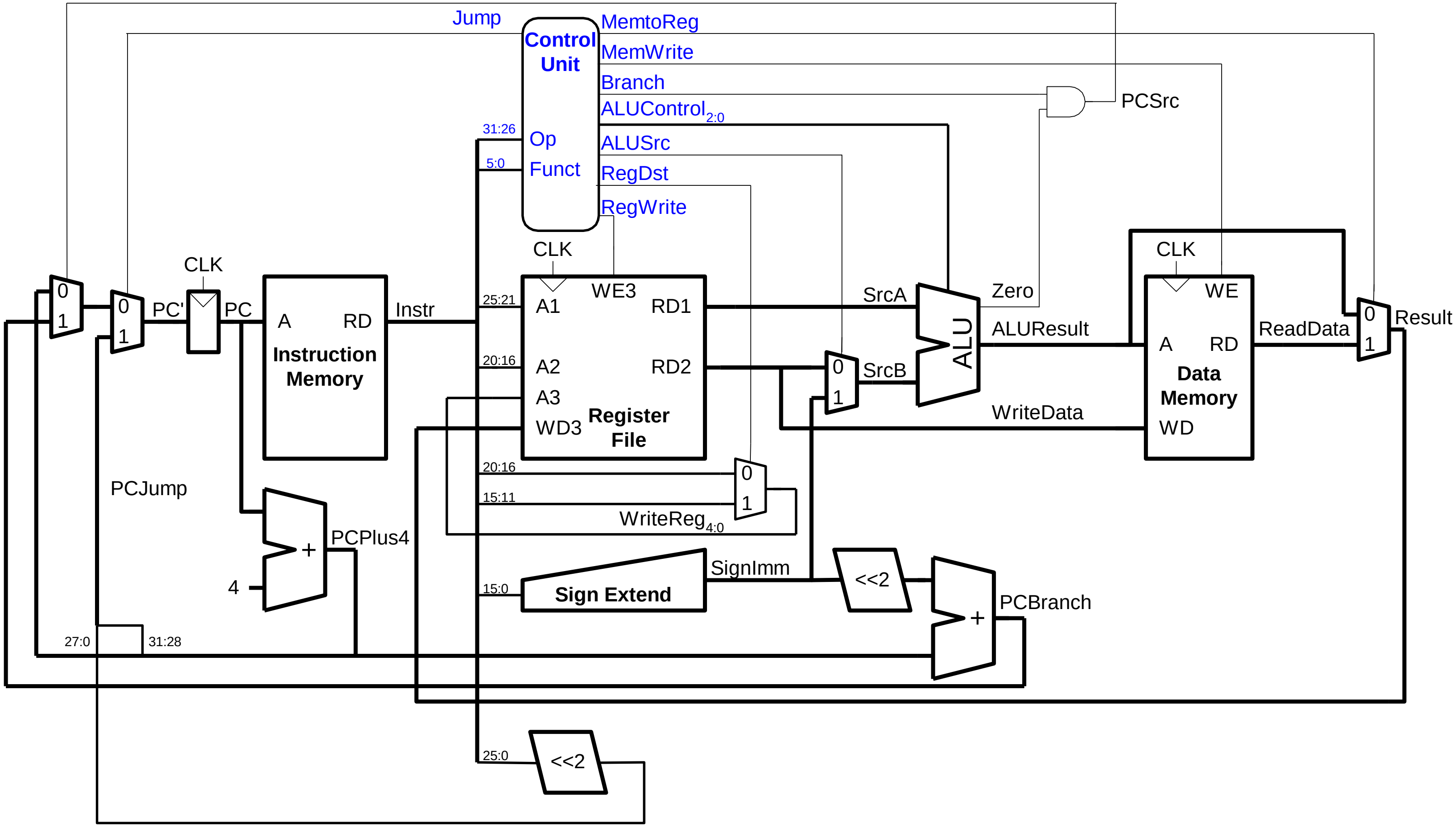
Control FSM: j



Control FSM: j

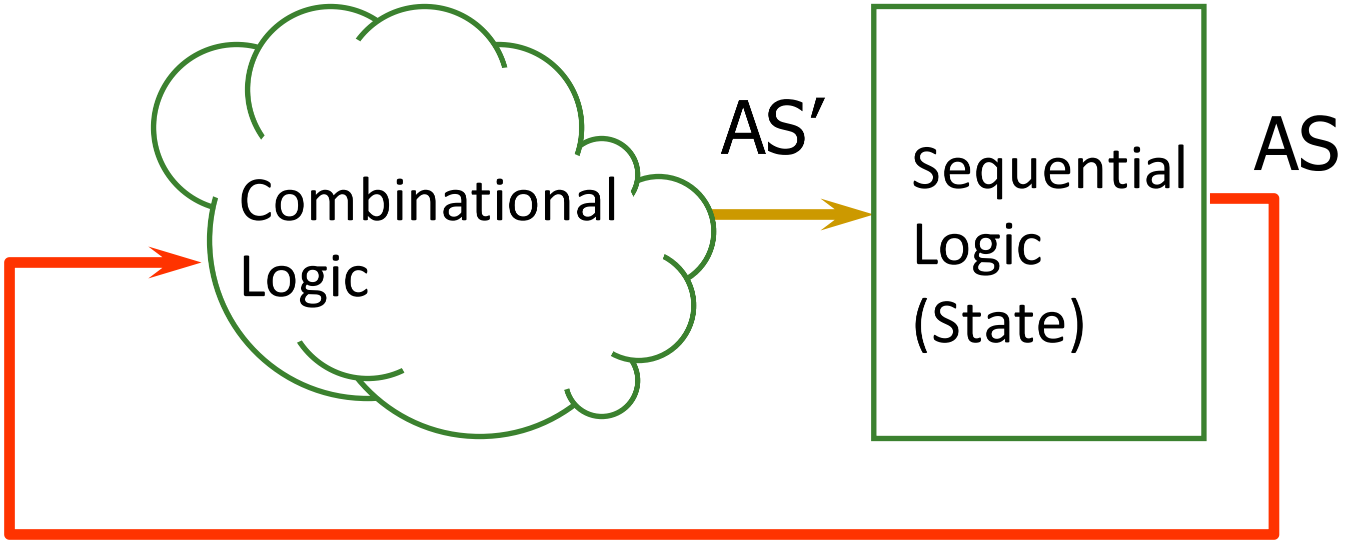


Review: Single-Cycle MIPS Processor



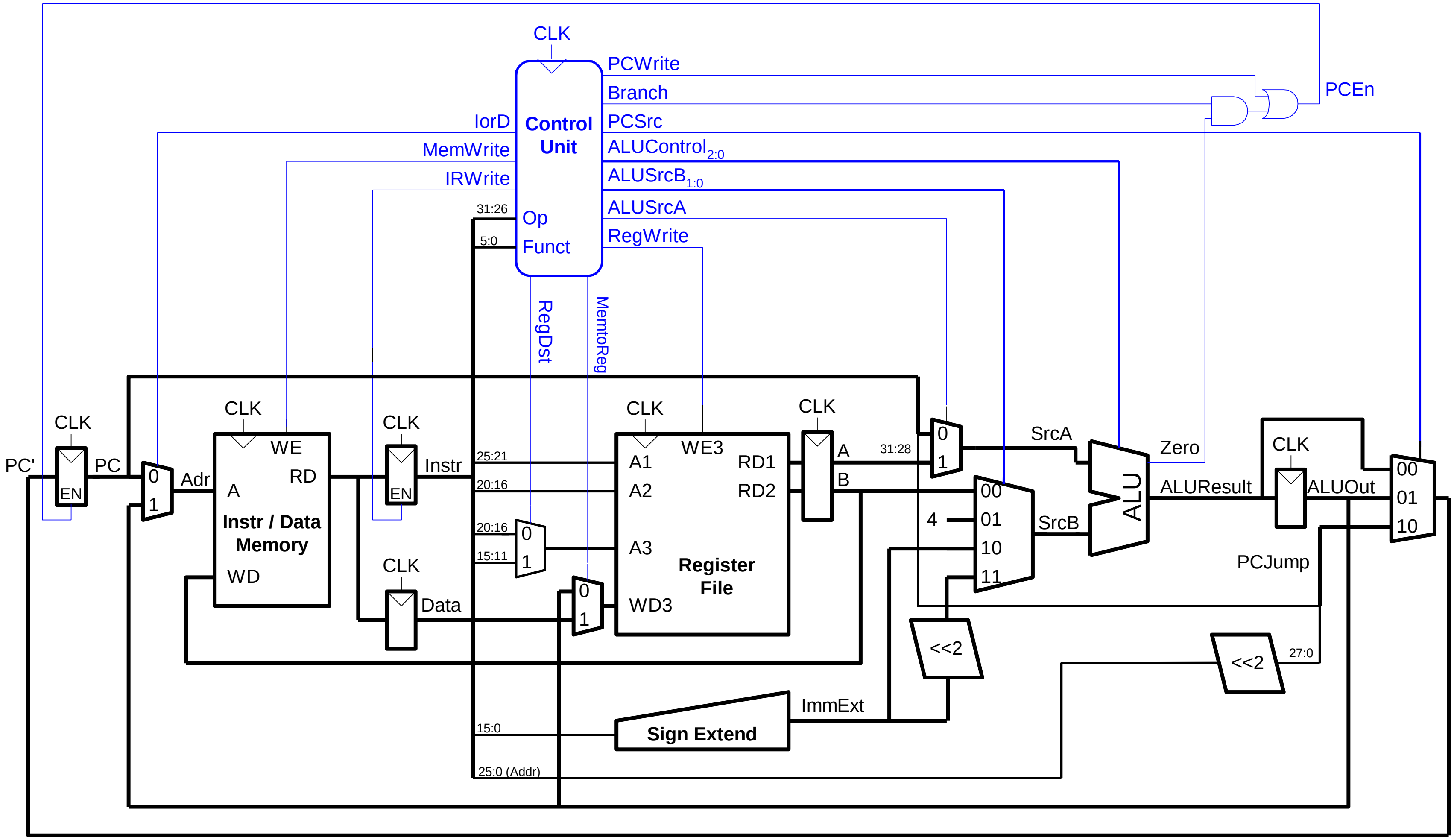
Review: Single-Cycle MIPS FSM

- Single-cycle machine

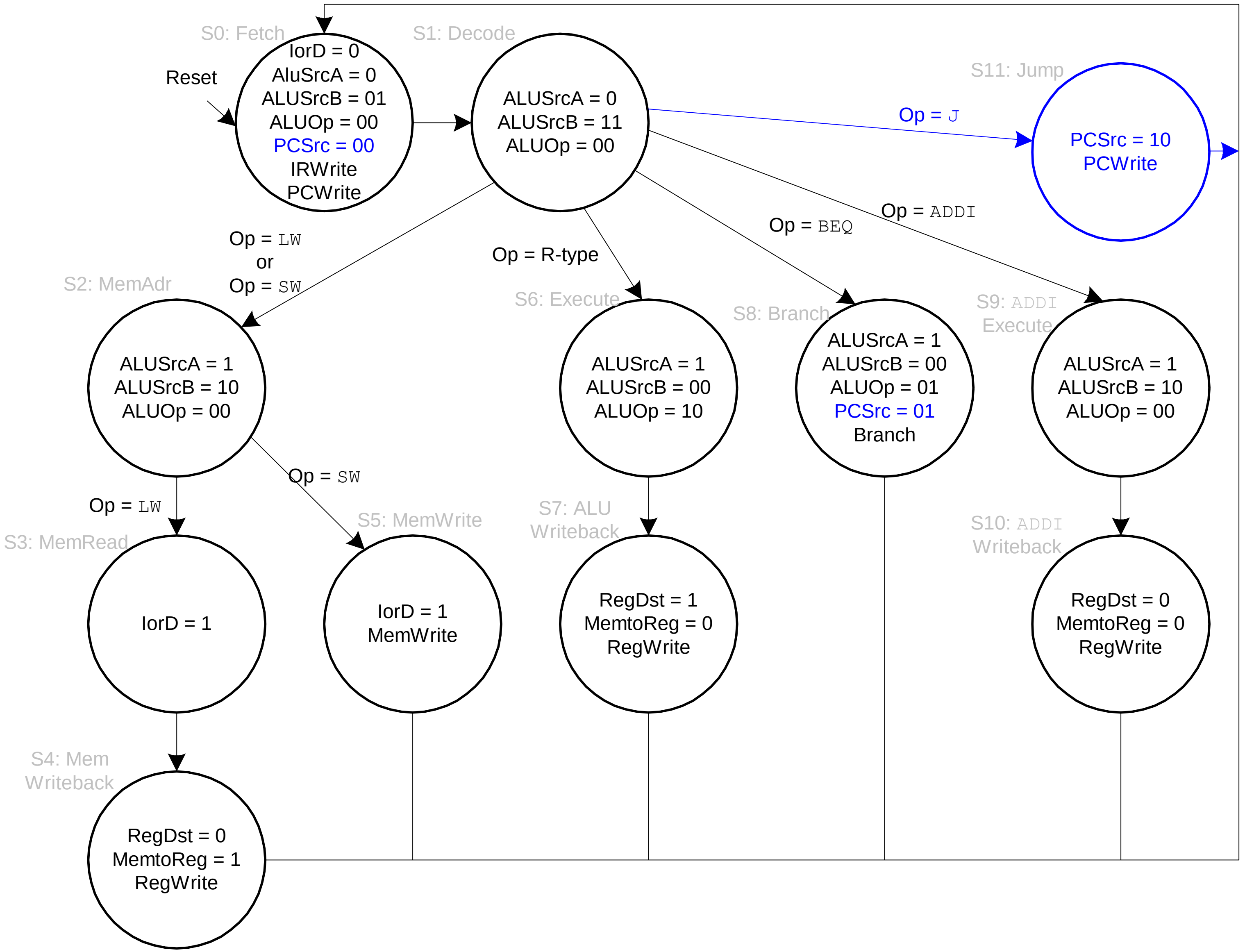


AS: Architectural State

Review: Multi-Cycle MIPS Processor



Review: Multi-Cycle MIPS FSM



What is the shortcoming of this design?

What does this design assume about memory?

What If Memory Takes > One Cycle?

- Stay in the same "memory access" state until memory returns the data
- "Memory Ready?" bit is an input to the control logic that determines the next state

Another Example: **Microprogrammed** Multi-Cycle Microarchitecture

Recall: How Do We Implement This?

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

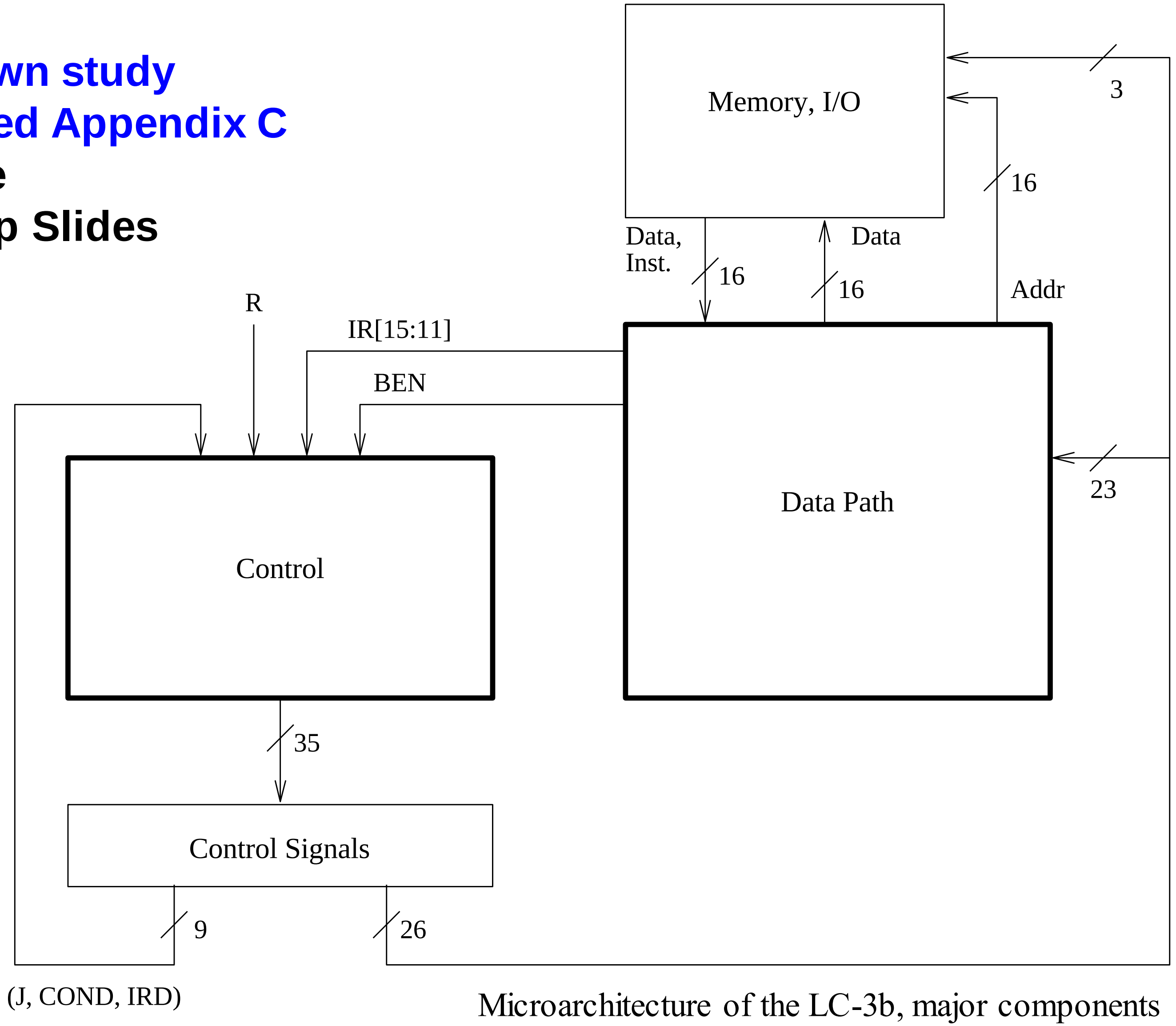
THE BEST WAY TO DESIGN AN AUTOMATIC CALCULATING MACHINE

By M. V. Wilkes, M.A., Ph.D., F.R.A.S.

- An elegant implementation:
  - The concept of microcoded/microprogrammed machines

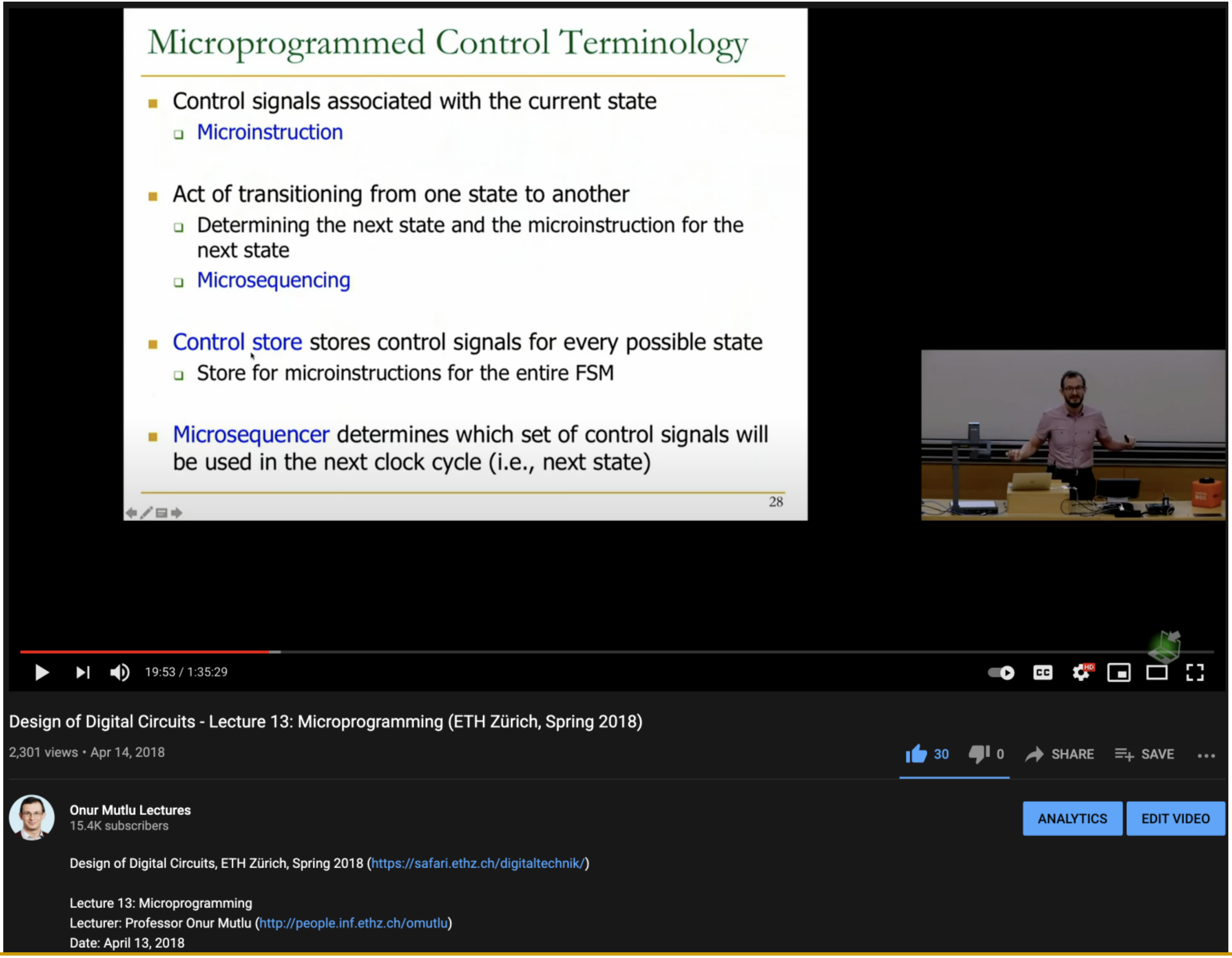
Example uProgrammed Control & Datapath

For your own study
P&P Revised Appendix C
On website
+ In Backup Slides



Microarchitecture of the LC-3b, major components

For More on Microprogrammed Designs



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)

28

Design of Digital Circuits - Lecture 13: Microprogramming (ETH Zürich, Spring 2018)

2,301 views • Apr 14, 2018

Onur Mutlu Lectures
15.4K subscribers

Design of Digital Circuits, ETH Zürich, Spring 2018 (https://safari.ethz.ch/digitaltechnik/)

Lecture 13: Microprogramming
Lecturer: Professor Onur Mutlu (http://people.inf.ethz.ch/omutlu)
Date: April 13, 2018

https://www.youtube.com/onurmutlulectures

Detailed Lectures on Microprogramming

- Design of Digital Circuits, Spring 2018, Lecture 13
  - Microprogramming (ETH Zürich, Spring 2018)
  - https://www.youtube.com/watch?v=u4GhShuBP3Y&list=PL5Q2soXY2Zi_QedyPWtRmFUJ2F8DdYP7l&index=13

- Computer Architecture, Spring 2013, Lecture 7
  - Microprogramming (CMU, Spring 2013)
  - https://www.youtube.com/watch?v=_igvSl5h8cs&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=7

Digital Design & Computer Arch.

Lecture 12: Microarchitecture Fundamentals II

Prof. Onur Mutlu

ETH Zürich

Spring 2021

15 April 2021

We did not cover the following slides. They are for your benefit.

Backup Slides on Single-Cycle Uarch for Your Own Study

Please study these to reinforce the concepts
we covered in lectures.

Please do the readings together with these slides:
H&H, Chapter 7.1-7.3, 7.6

Another Single-Cycle MIPS Processor (from H&H)

These are slides for your own study.
They are to complement your reading
H&H, Chapter 7.1-7.3, 7.6

What to do with the Program Counter?

- The PC needs to be incremented by 4 during each cycle (for the time being).

- Initial PC value (after reset) is `0x00400000`

```
reg [31:0] PC_p, PC_n;      // Present and next state of PC

// […]

  assign PC_n <= PC_p + 4;                   // Increment by 4;

  always @ (posedge clk, negedge rst)
    begin
       if (rst == ‘0’) PC_p <= 32’h00400000; // default
       else            PC_p <= PC_n;         // when clk
    end
```

We Need a Register File

- **Store 32 registers, each 32-bit**
  - 2^5 == 32, we need 5 bits to address each
- **Every R-type instruction uses 3 register**
  - Two for reading (RS, RT)
  - One for writing (RD)
- **We need a special memory with:**
  - 2 read ports (address x2, data out x2)
  - 1 write port (address, data in)

Register File

```
input [4:0]   a_rs, a_rt, a_rd;
input [31:0]  di_rd;
input         we_rd;
output [31:0] do_rs, do_rt;

  reg [31:0] R_arr [31:0]; // Array that stores regs

  // Circuit description
  assign do_rs = R_arr[a_rs];          // Read RS


  assign do_rt = R_arr[a_rt];          // Read RT


  always @ (posedge clk)
      if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```

Register File

```
input [4:0]   a_rs, a_rt, a_rd;
input [31:0]  di_rd;
input         we_rd;
output [31:0] do_rs, do_rt;

  reg [31:0] R_arr [31:0]; // Array that stores regs

  // Circuit description; add the trick with $0
  assign do_rs = (a_rs != 5’b00000)?   // is address 0?
                  R_arr[a_rs] : 0;     // Read RS or 0

  assign do_rt = (a_rt != 5’b00000)?   // is address 0?
                  R_arr[a_rt] : 0;     // Read RT or 0

  always @ (posedge clk)
      if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```

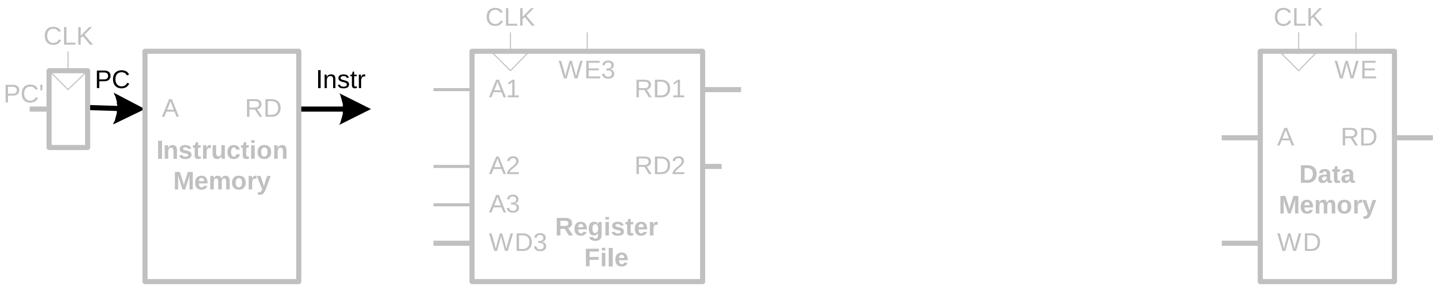
Data Memory Example

- Will be used to store the bulk of data

```
input [15:0]  addr; // Only 16 bits in this example
input [31:0]  di;
input         we;
output [31:0] do;

  reg [31:0] M_arr [0:65535];          // Array for Memory

  // Circuit description
  assign do = M_arr[addr];             // Read memory

  always @ (posedge clk)
      if (we) M_arr[addr] <= di;       // write memory
```

Single-Cycle Datapath: `lw` fetch

- *STEP 1:* **Fetch instruction**



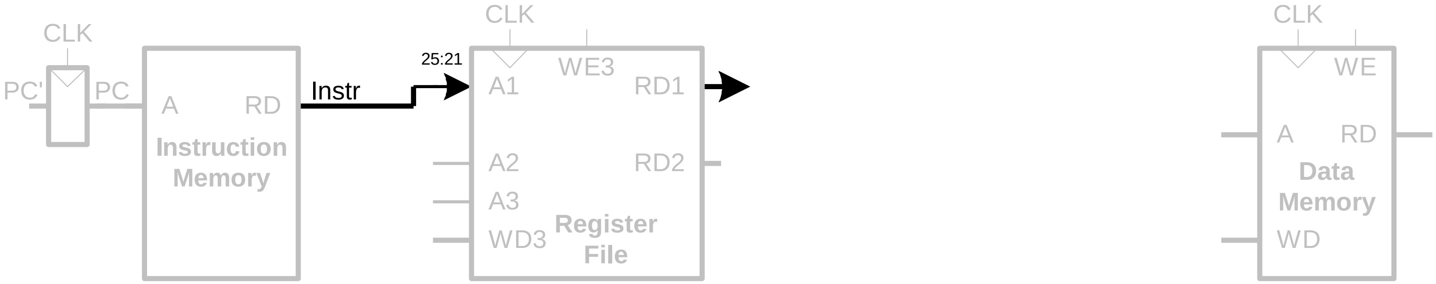
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` register read

- *STEP 2:* **Read source operands from register file**



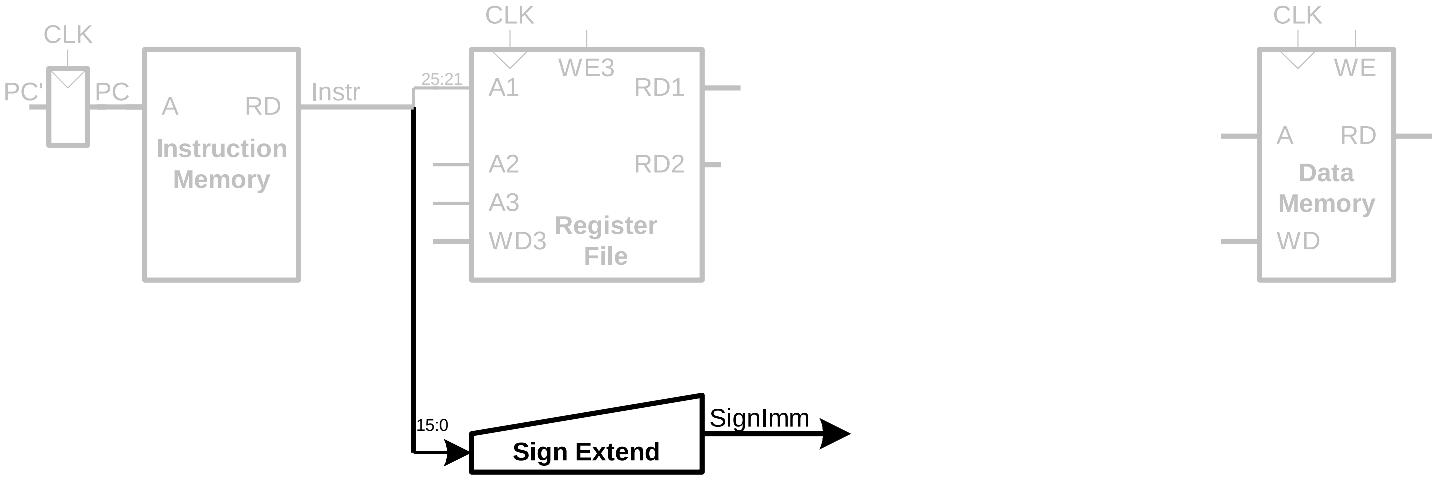
```
lw $s3, 1($0)  # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` immediate

- *STEP 3:* Sign-extend the immediate



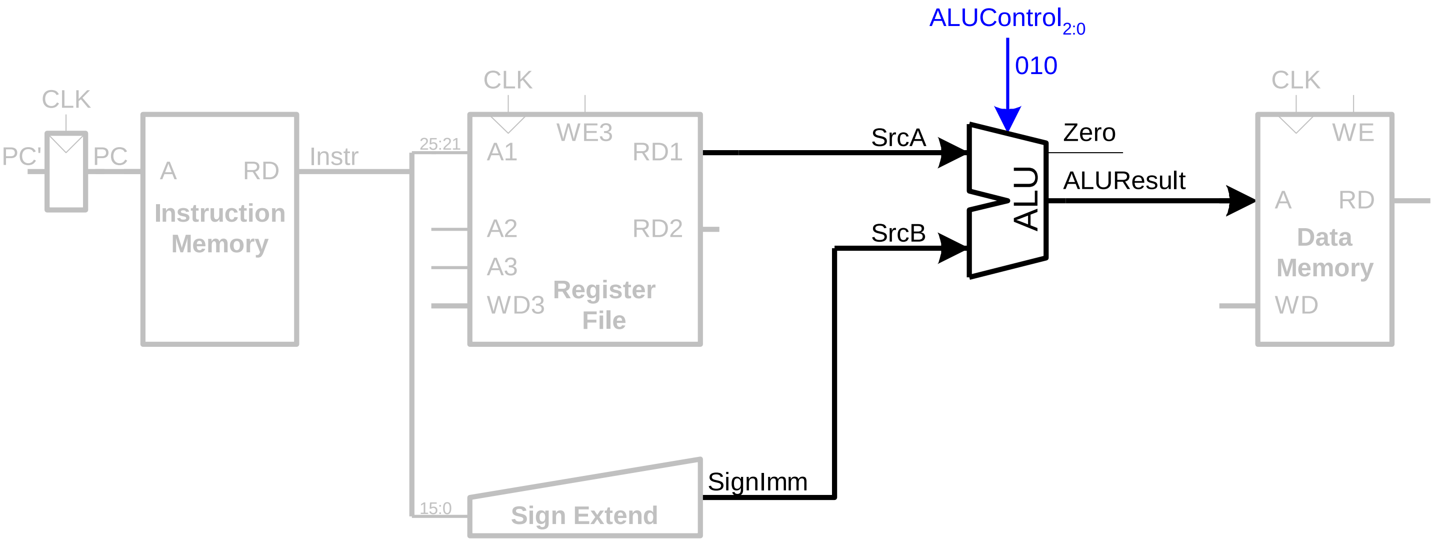
```
lw $s3, 1($0)  # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` address

- ***STEP 4:*** **Compute the memory address**



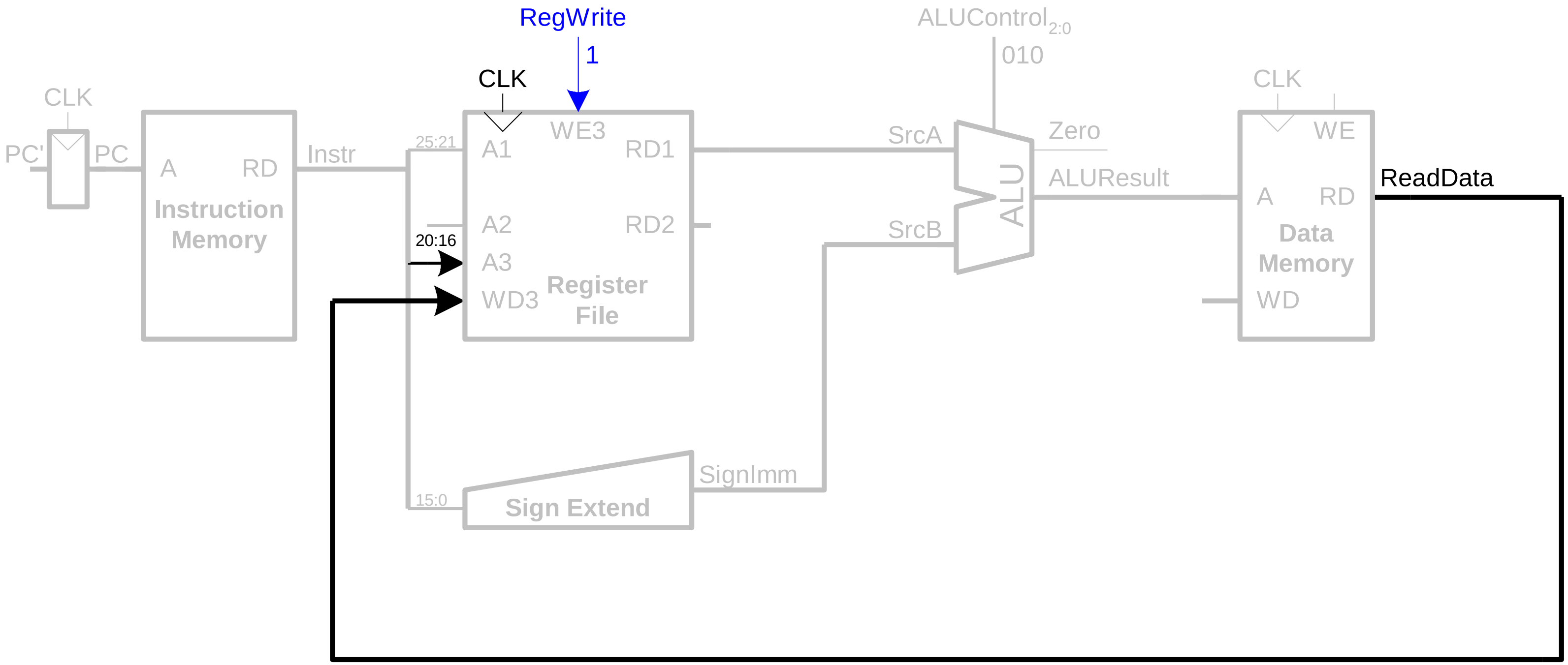
```
lw $s3, 1($0)  # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: lw memory read

- *STEP 5:* **Read from memory and write back to register file**



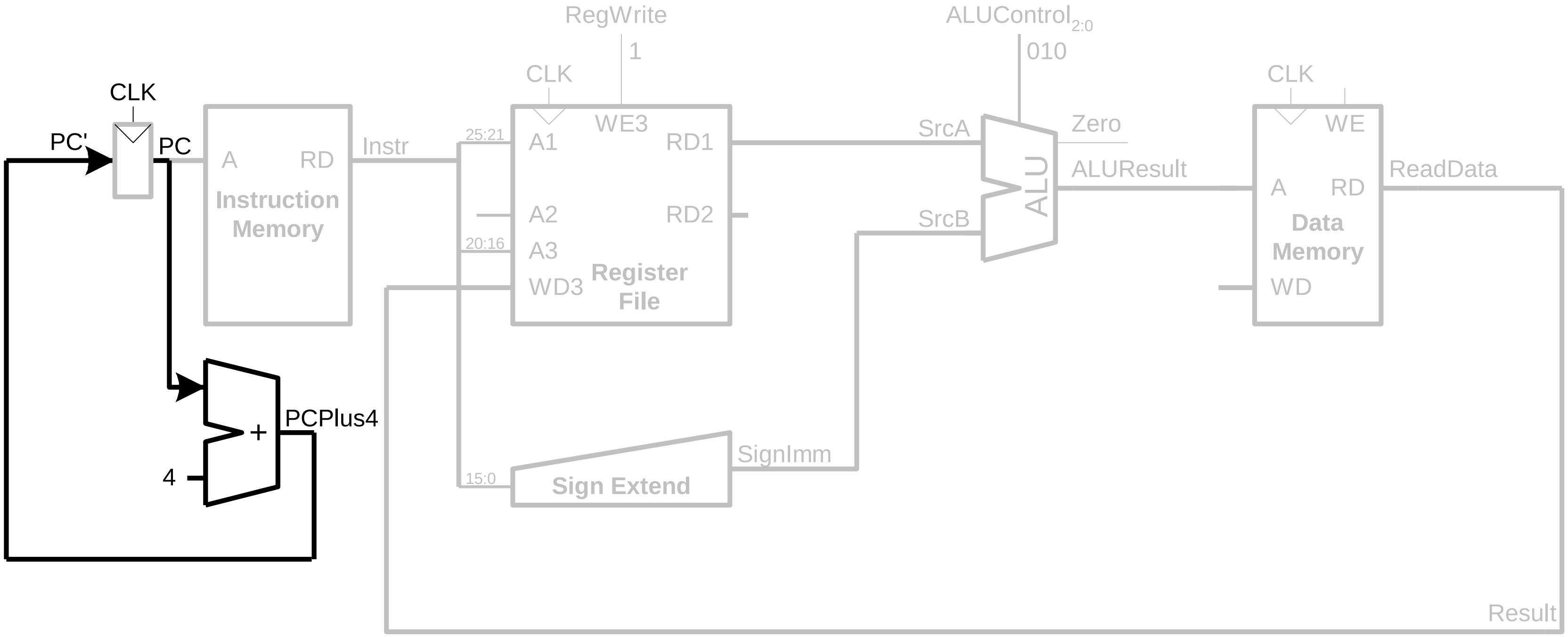
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `lw` PC increment

- *STEP 6:* **Determine address of next instruction**



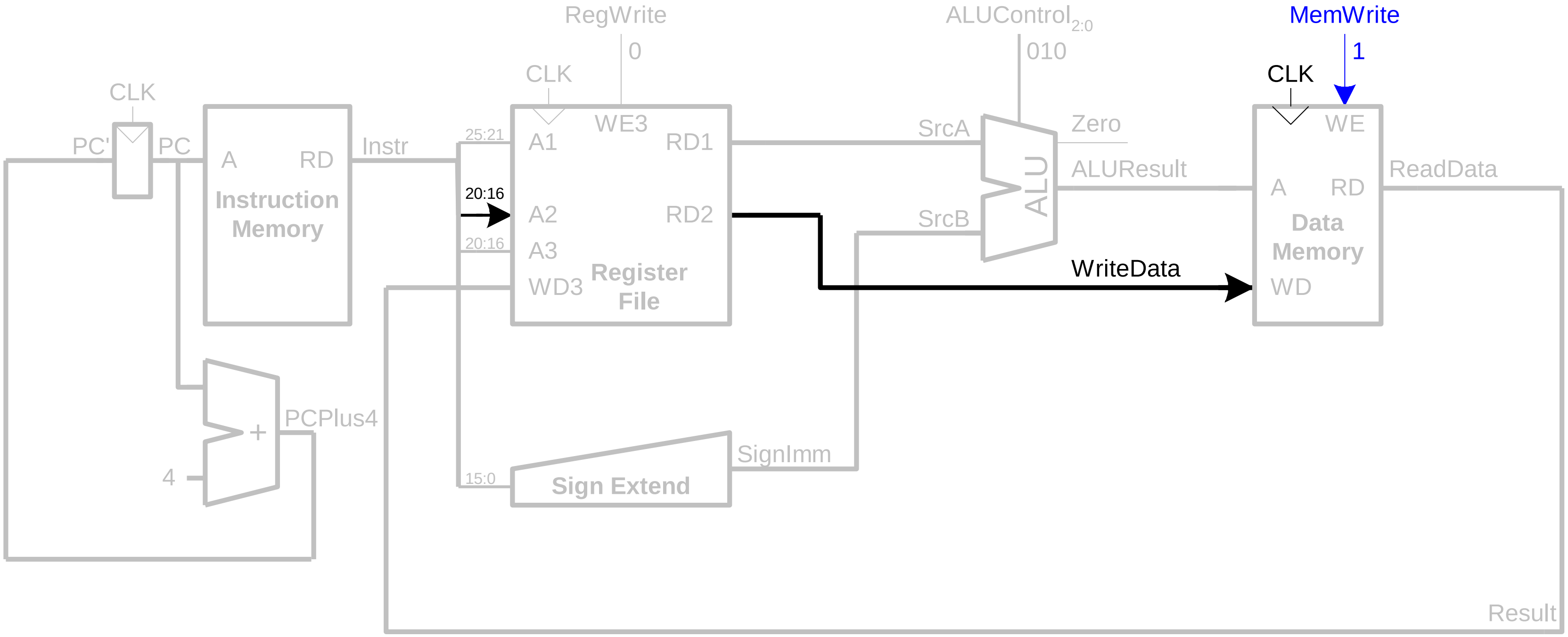
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: `sw`

- **Write data in `rt` to memory**



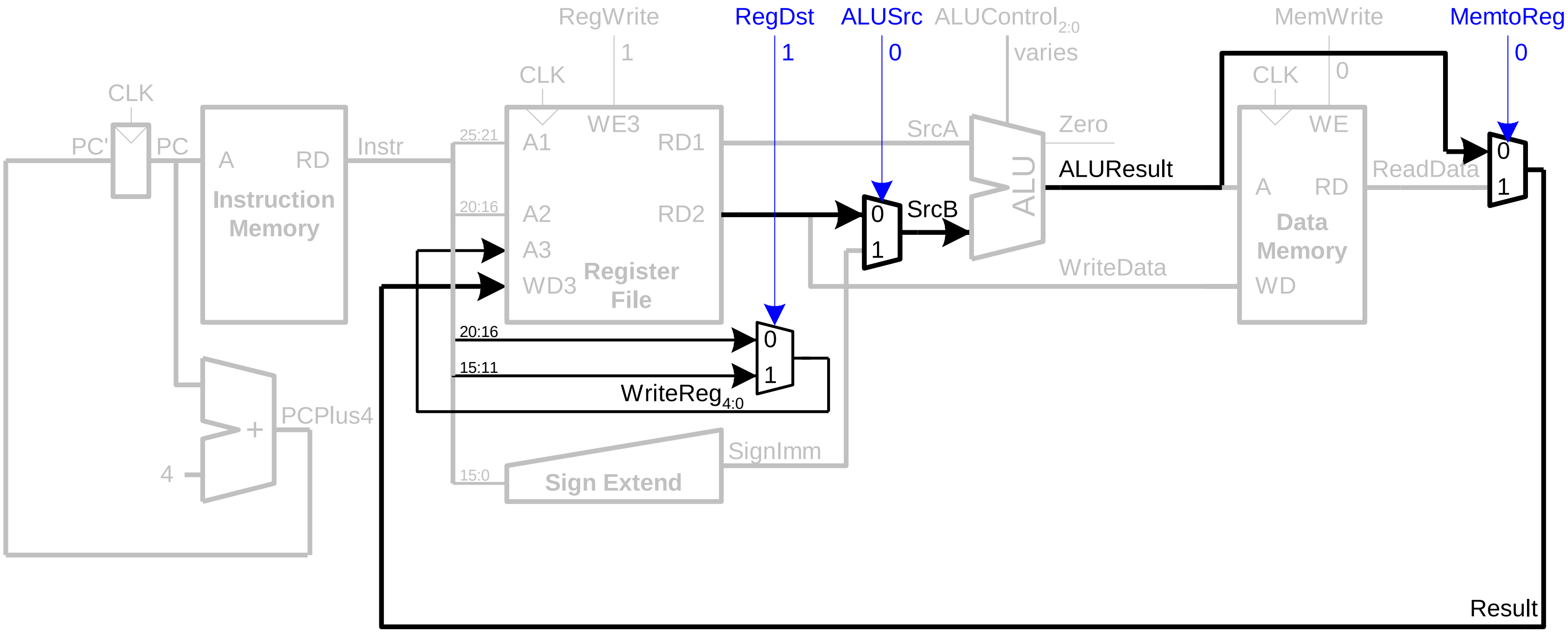
```
sw $t7, 44($0)  # write t7 into memory address 44
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: R-type Instructions

- **Read from rs and rt, write ALUResult to register file**

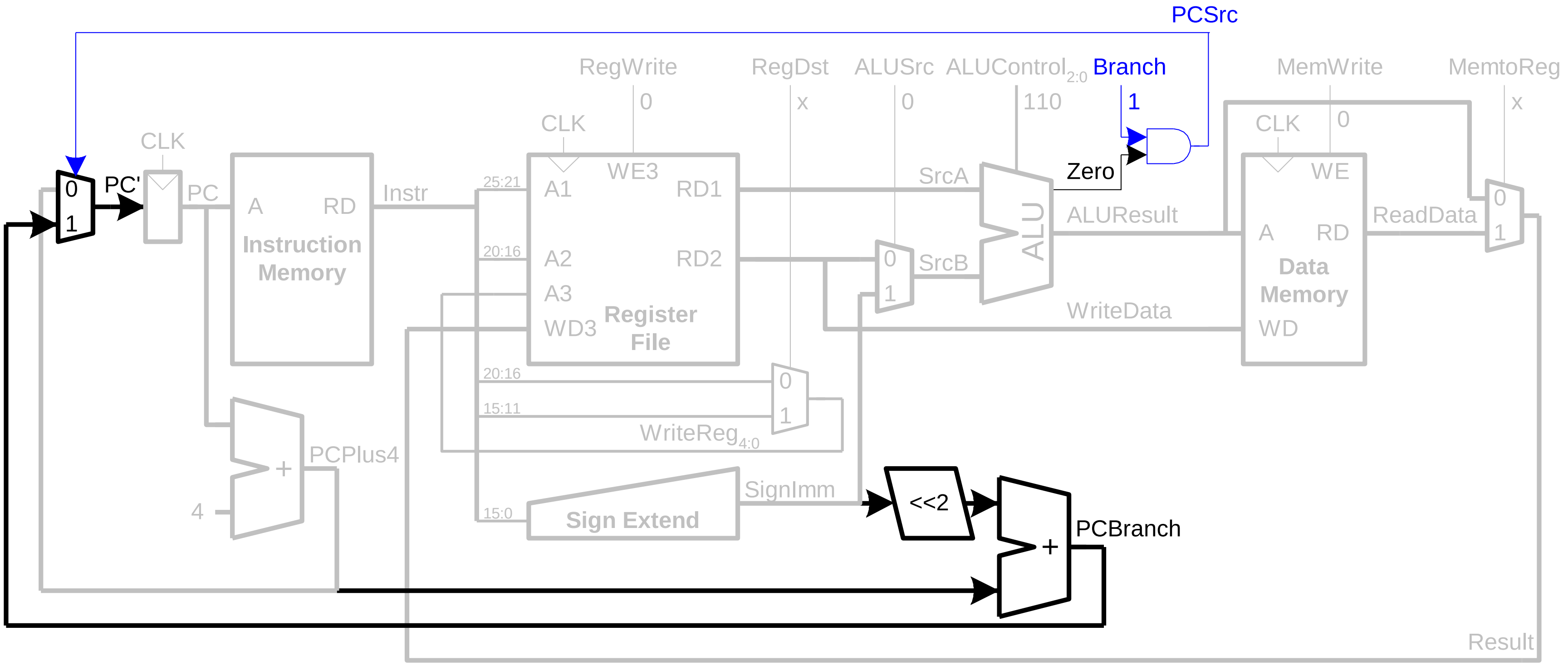


```
add t, b, c  # t = b + c
```

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Single-Cycle Datapath: beq

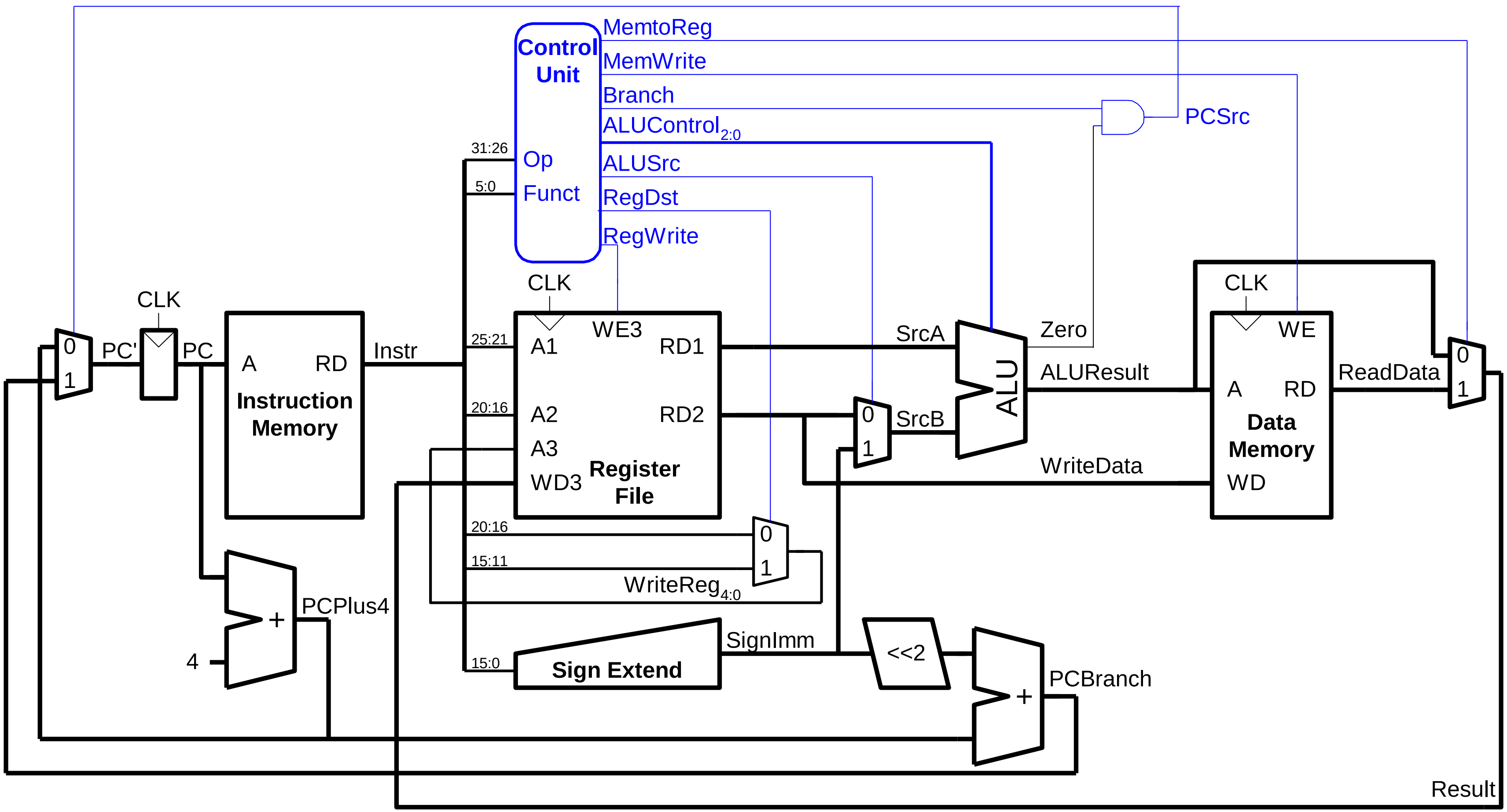


```
beq  $s0, $s1, target  # branch is taken
```

- **Determine whether values in rs and rt are equal**

  Calculate BTA = (sign-extended immediate << 2) + (PC+4)

Complete Single-Cycle Processor

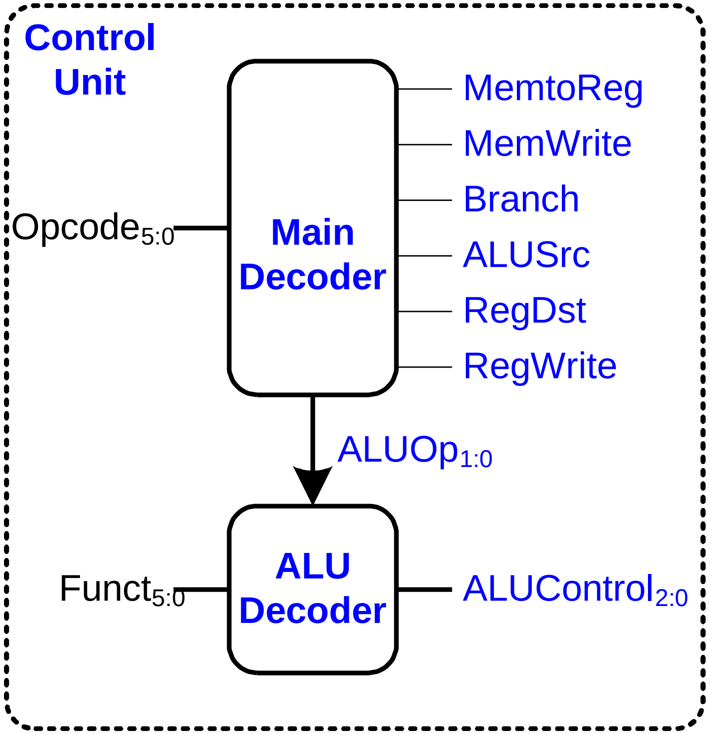


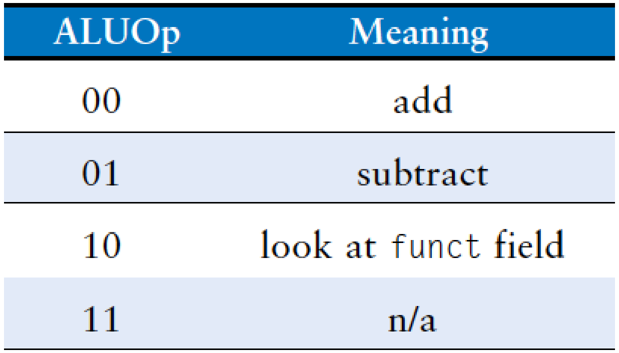
Our MIPS Datapath has Several Options

- **ALU inputs**
  - Either RT or Immediate ***(MUX)***
- **Write Address of Register File**
  - Either RD or RT ***(MUX)***
- **Write Data In of Register File**
  - Either ALU out or Data Memory Out ***(MUX)***
- **Write enable of Register File**
  - Not always a register write ***(MUX)***
- **Write enable of Memory**
  - Only when writing to memory (sw) ***(MUX)***

All these options are our control signals

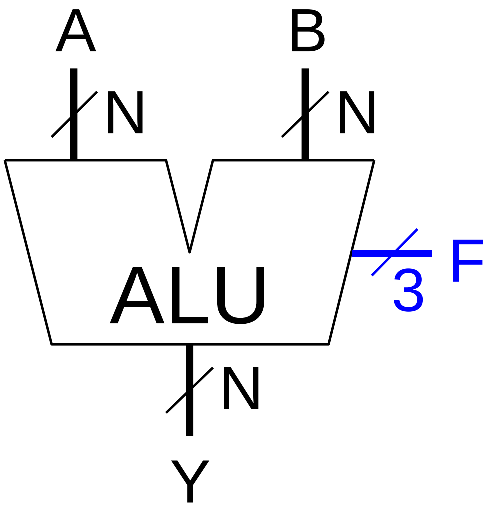
Control Unit





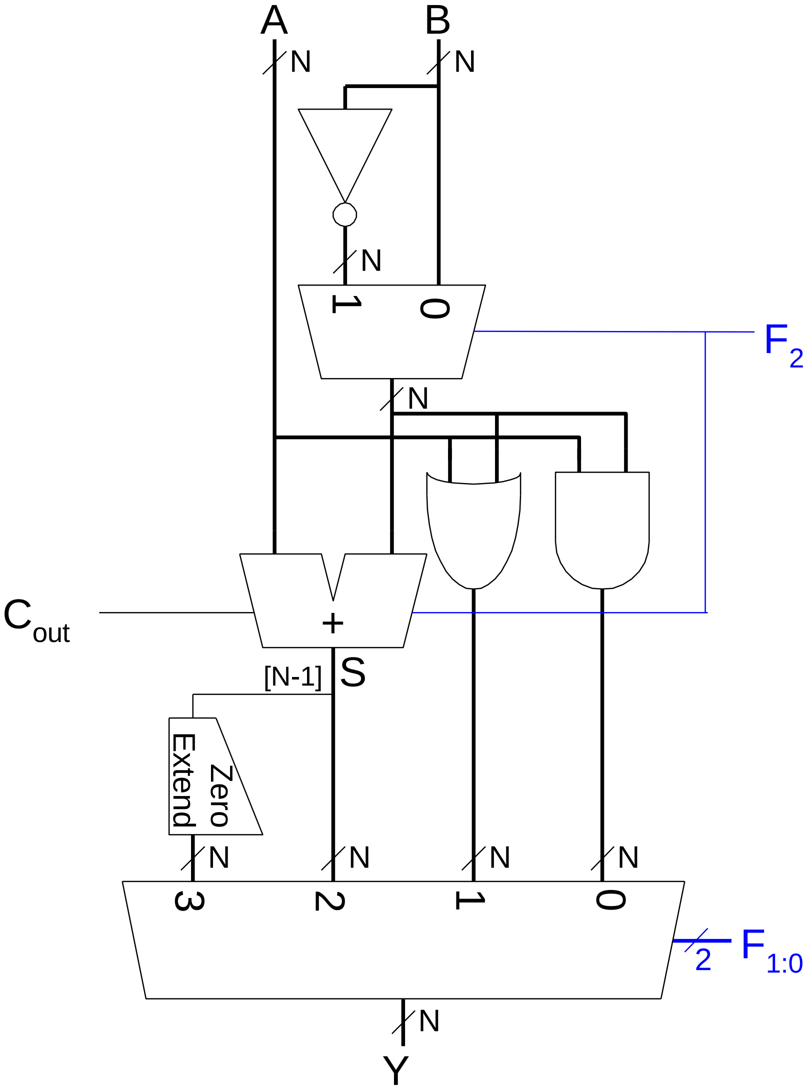
ALUOp	Meaning
00	add
01	subtract
10	look at funct field
11	n/a

ALU Does the Real Work in a Processor



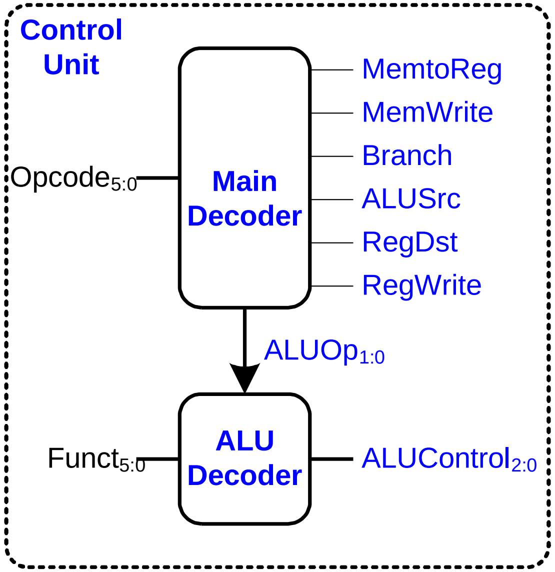
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

ALU Internals



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

Control Unit: ALU Decoder



ALUOp$_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

ALUOp$_{1:0}$	Funct	ALUControl$_{2:0}$
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (add)	010 (Add)
1X	100010 (sub)	110 (Subtract)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

Let us Develop our Control Table

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

Let us Develop our Control Table

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	funct

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

Let us Develop our Control Table

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	funct
`lw`	100011	1	0	1	0	1	add

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

Let us Develop our Control Table

Instruction	Op$_{5:0}$	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	funct
lw	100011	1	0	1	0	1	add
sw	101011	0	X	1	1	X	add

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

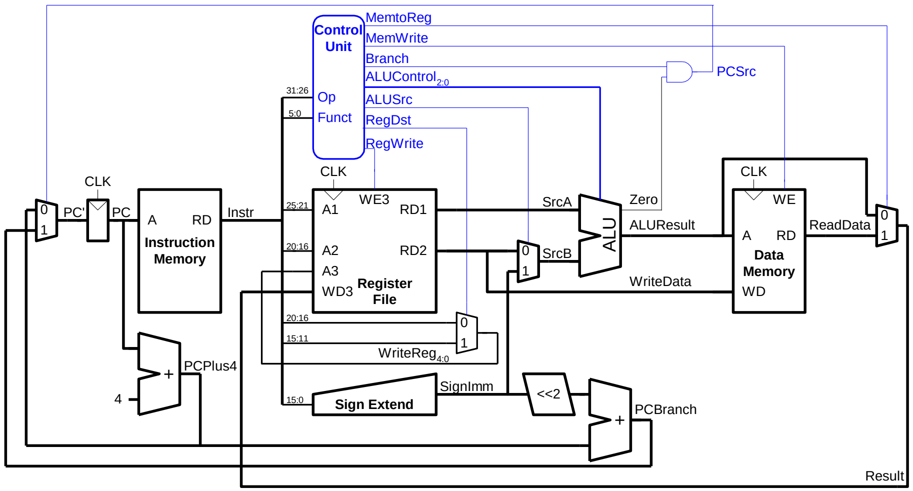
More Control Signals

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	funct
lw	100011	1	0	1	0	0	1	add
sw	101011	0	X	1	0	1	X	add
beq	000100	0	X	0	1	0	X	sub

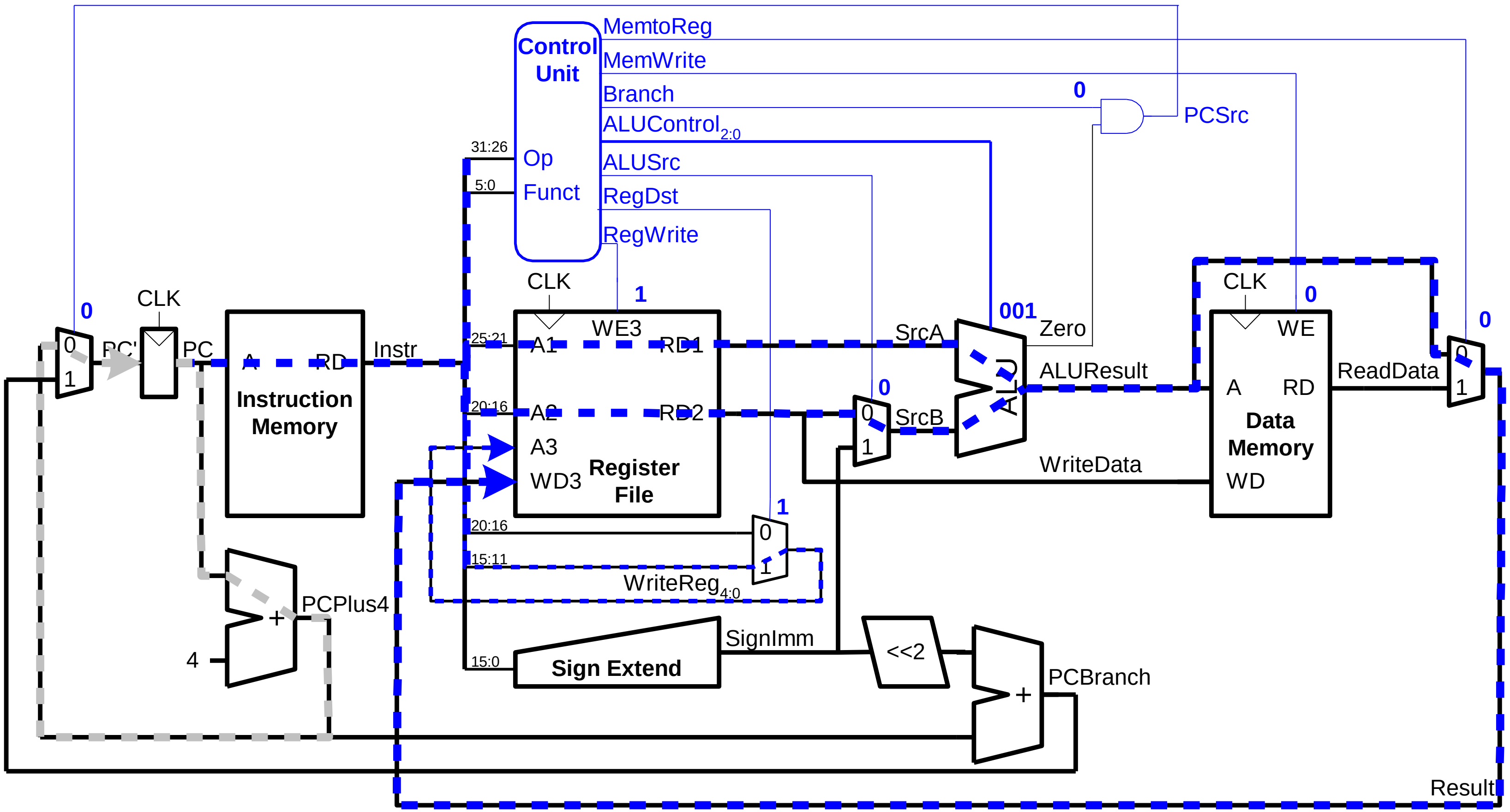
- **New Control Signal**
  - ***Branch*:** Are we jumping or not ?

Control Unit: Main Decoder

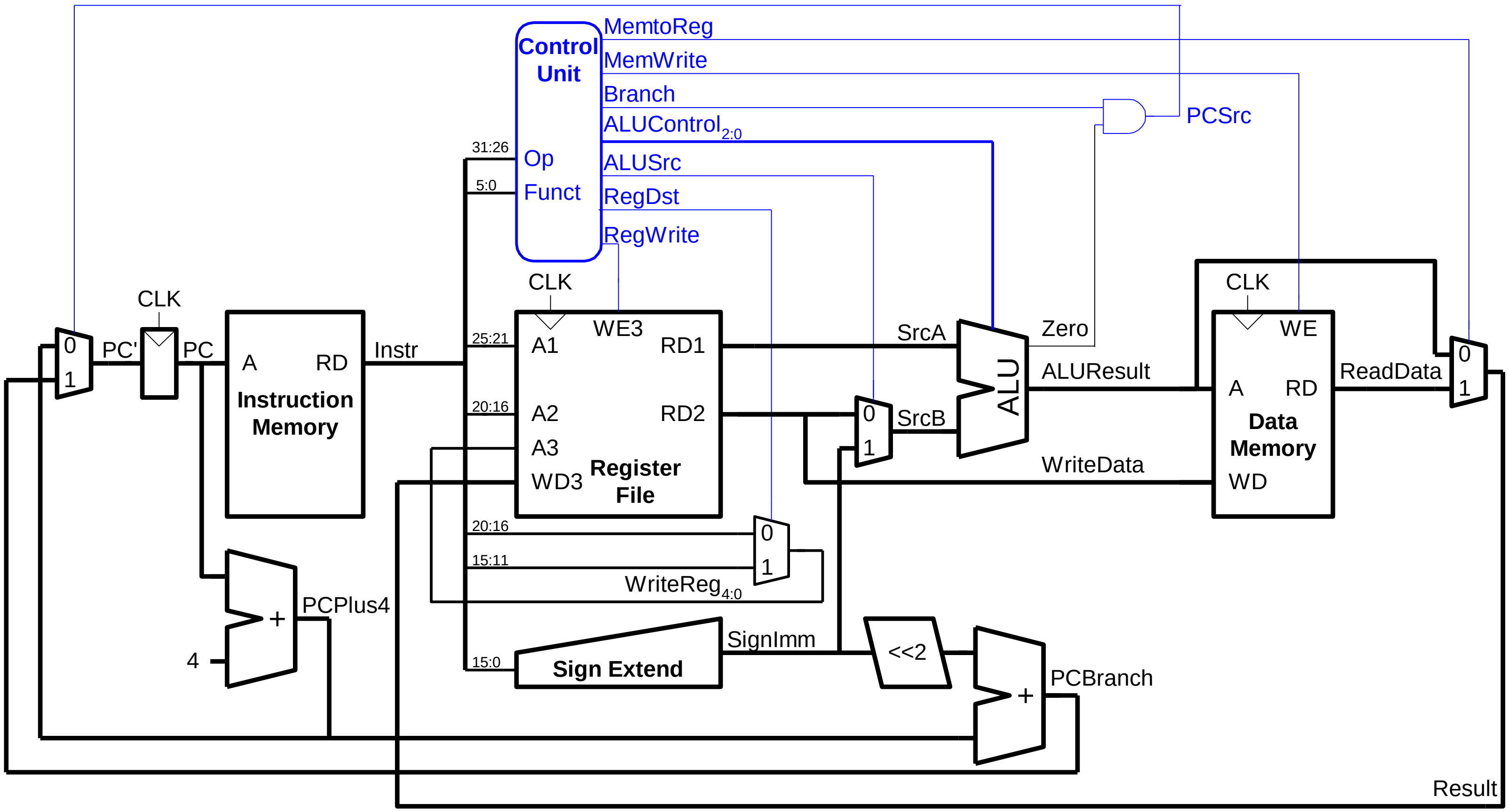
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01



Single-Cycle Datapath Example: or



Extended Functionality: addi

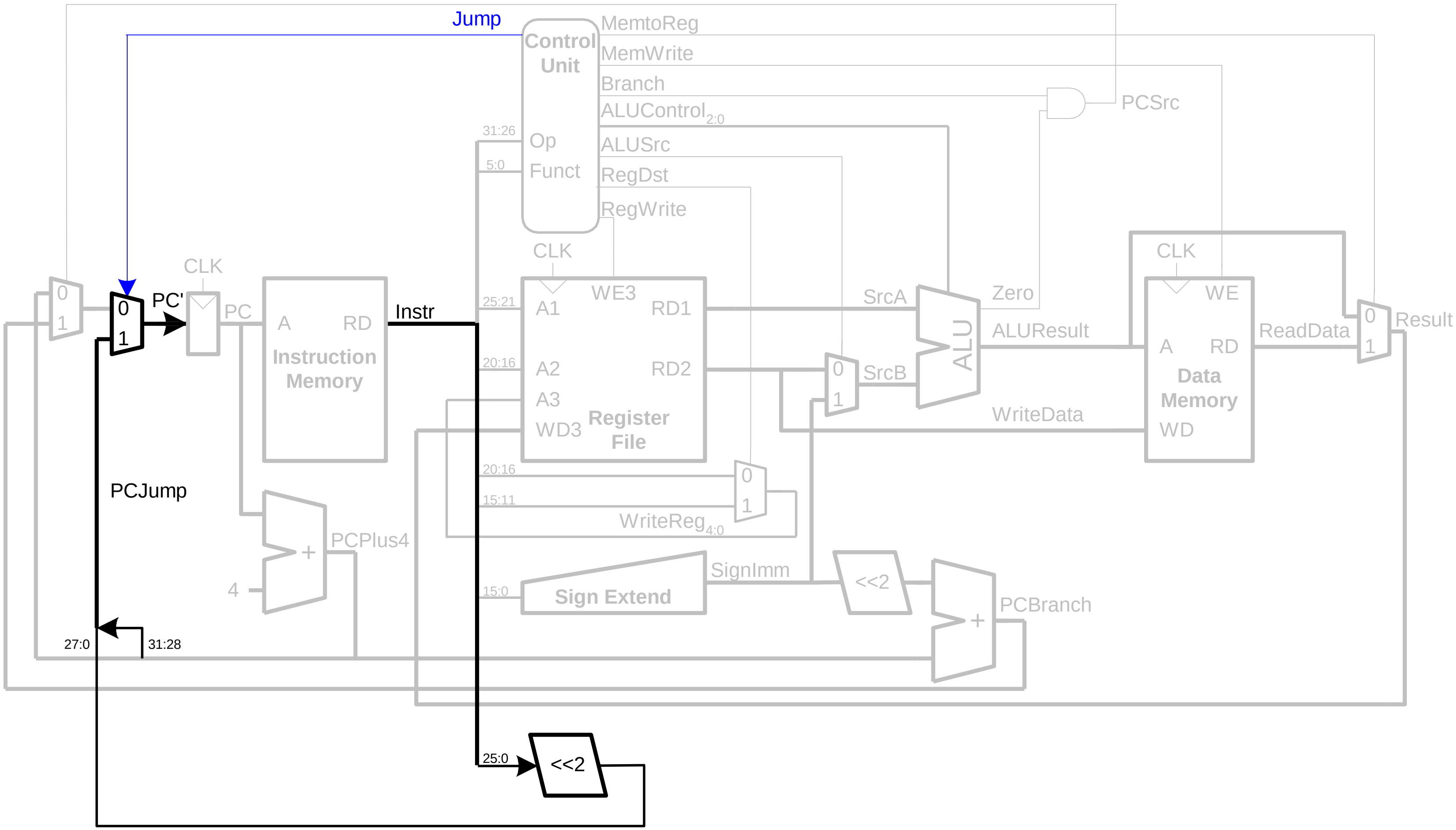


- No change to datapath

Control Unit: addi

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

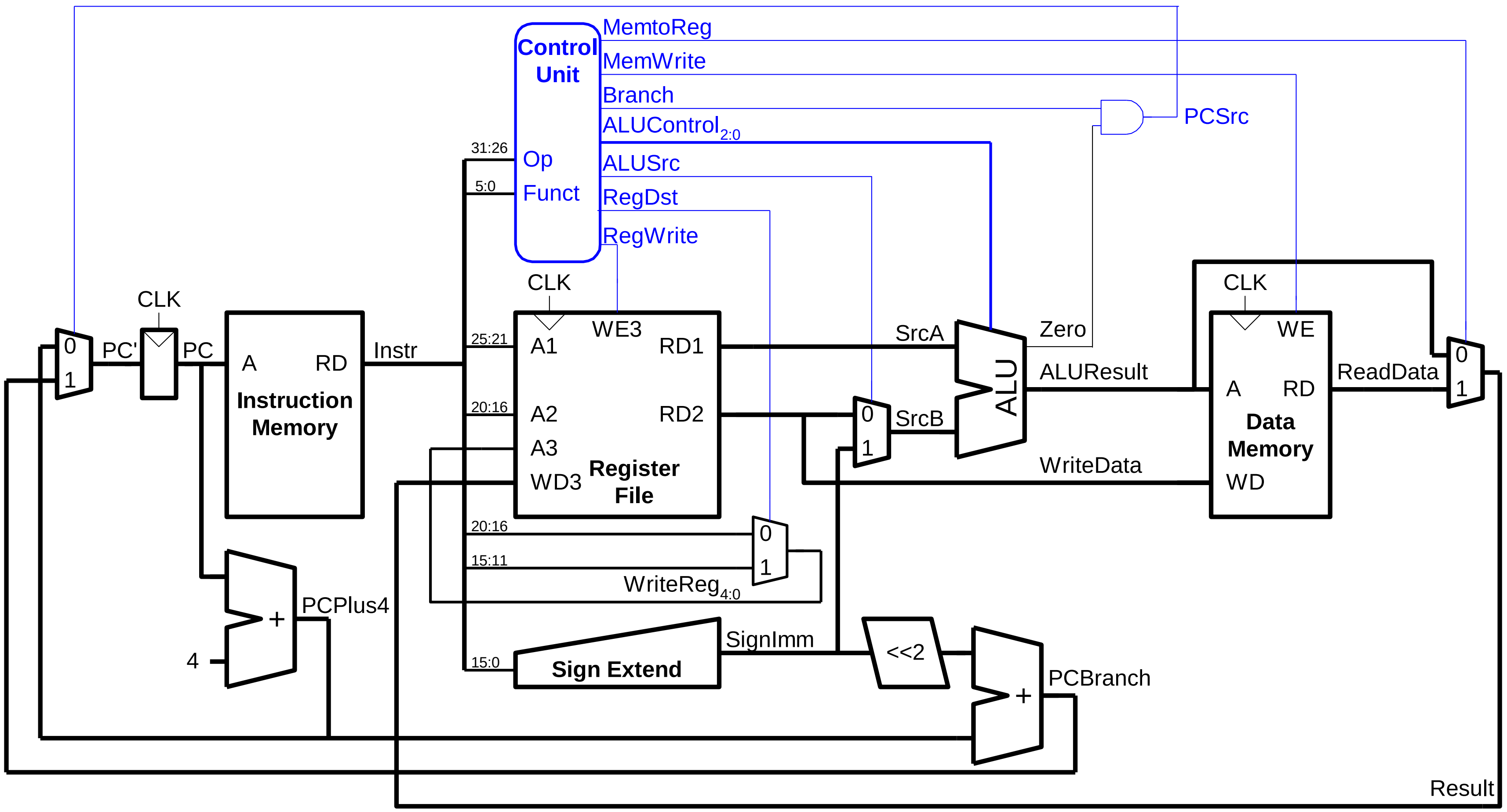
Extended Functionality: j



Control Unit: Main Decoder

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
j	000100	0	X	X	X	0	X	XX	1

Review: Complete Single-Cycle Processor (H&H)



A Bit More on Performance Analysis

How can I Make the Program Run Faster?

N x CPI x (1/f)

How can I Make the Program Run Faster?

N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC – complex instruction sets)
  - Use better compilers

How can I Make the Program Run Faster?

N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC – complex instruction sets)
  - Use better compilers
- **Use fewer cycles to perform each instruction**
  - Simpler instructions (RISC – reduced instruction sets)
  - Use multiple units/ALUs/cores in parallel

How can I Make the Program Run Faster?

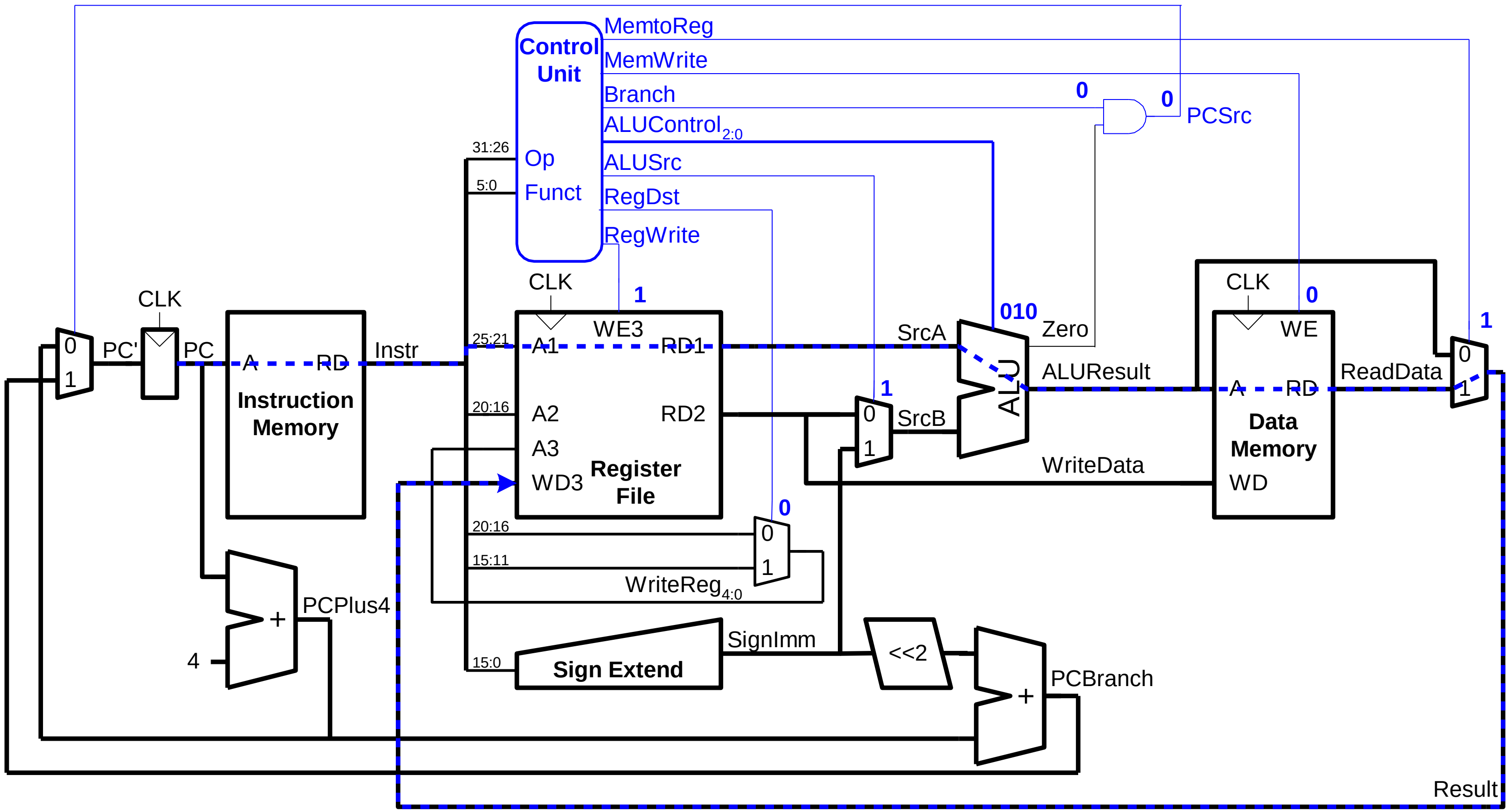
N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC – complex instruction sets)
  - Use better compilers
- **Use fewer cycles to perform each instruction**
  - Simpler instructions (RISC – reduced instruction sets)
  - Use multiple units/ALUs/cores in parallel
- **Increase the clock frequency**
  - Find a ‘newer’ technology to manufacture
  - Redesign time critical components
  - Adopt pipelining

Performance Analysis of Single-Cycle vs. Multi-Cycle Designs

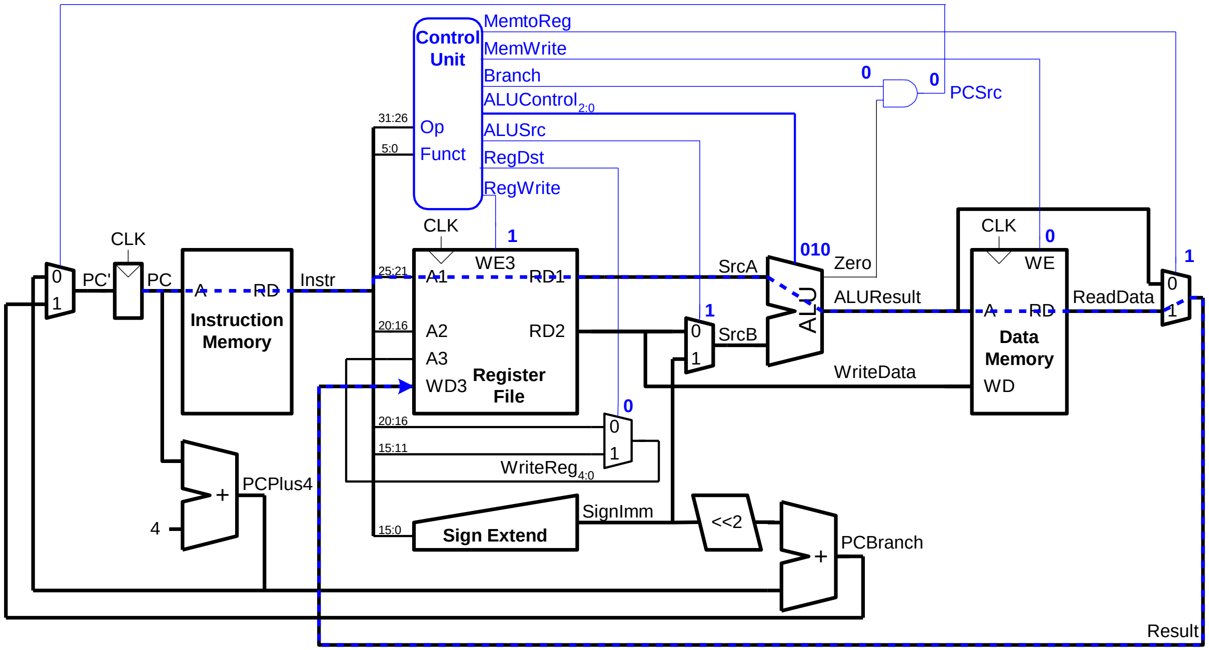
Single-Cycle Performance

- T_C is limited by the critical path (`lw`)



Single-Cycle Performance

- Single-cycle critical path:
  - $\mathbf{T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}}$
- In most implementations, limiting paths are:
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$



Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

T_c =

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$$\begin{aligned} T_c &= t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}} \\ &= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps} \\ &= 925 \text{ ps} \end{aligned}$$

Single-Cycle Performance Example

- Example:

  For a program with 100 billion instructions executing on a single-cycle MIPS processor:

Single-Cycle Performance Example

- Example:

  For a program with 100 billion instructions executing on a single-cycle MIPS processor:

Execution Time

$$= \#\ \text{instructions} \times \text{CPI} \times T_c$$

$$= (100 \times 10^{9})(1)(925 \times 10^{-12}\ \text{s})$$

$$= 92.5\ \text{seconds}$$

Multi-Cycle Performance: CPI

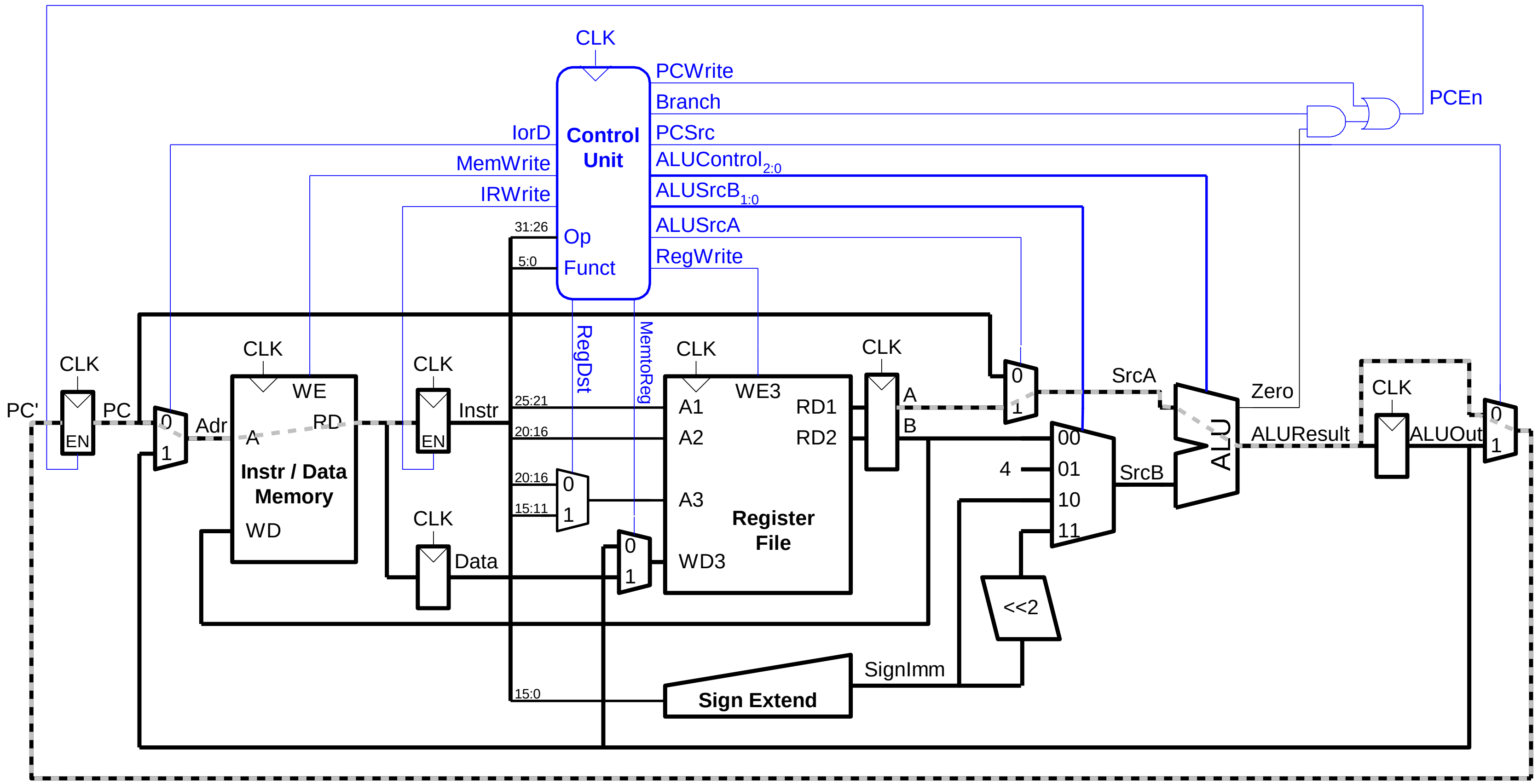
- Instructions take different number of cycles:
  - 3 cycles: `beq, j`
  - 4 cycles: `R-Type, sw, addi`
  - 5 cycles: `lw` **Realistic?**
- CPI is weighted average, e.g. SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

- *Average CPI* = (0.11 + 0.02) 3 +(0.52 + 0.10) 4 +(0.25) 5
  = 4.12

Multi-Cycle Performance: Cycle Time

- Multi-cycle critical path:

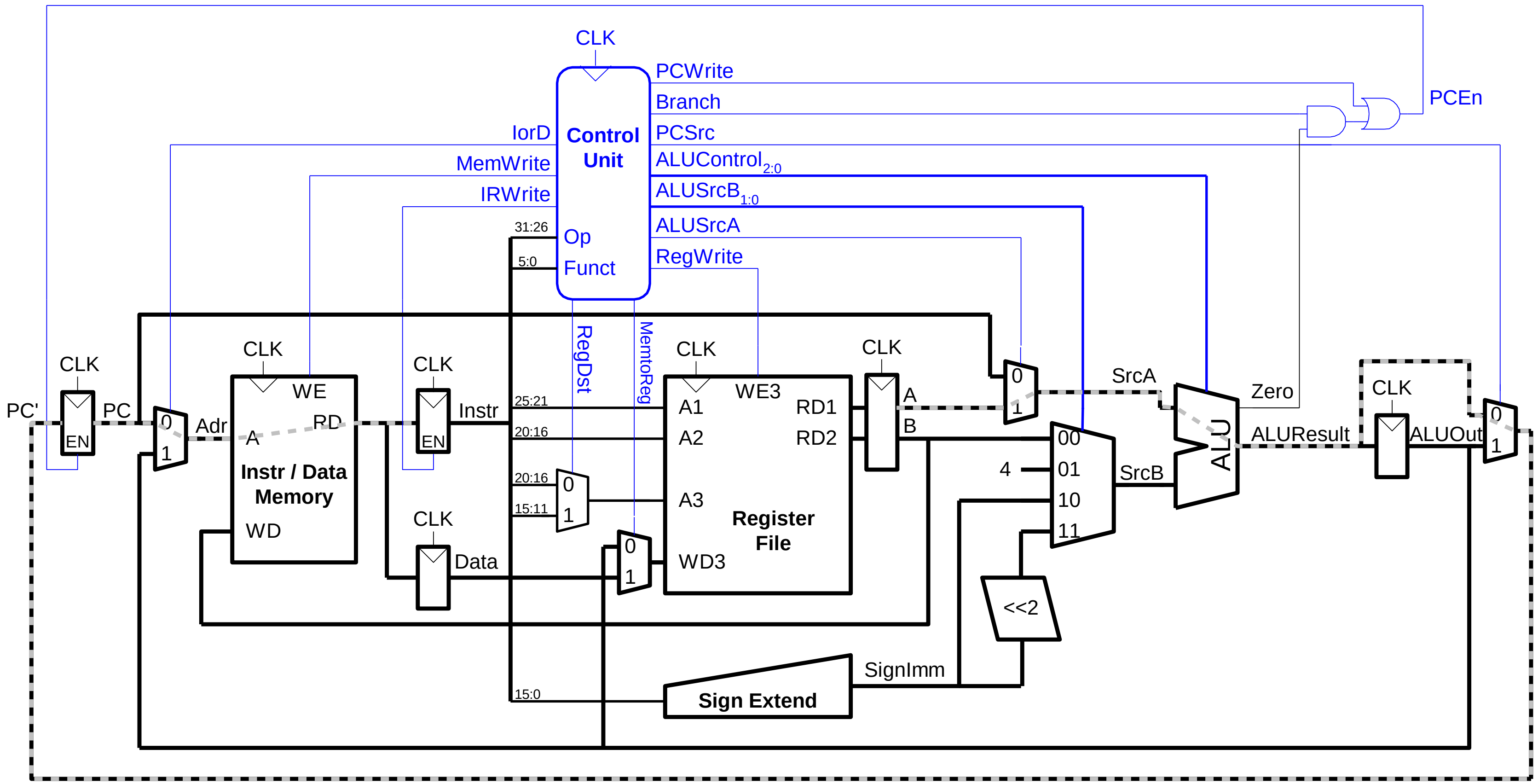
  $T_c =$



Multi-Cycle Performance: Cycle Time

- Multi-cycle critical path:

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$



Multi-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

T_c =

Multi-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

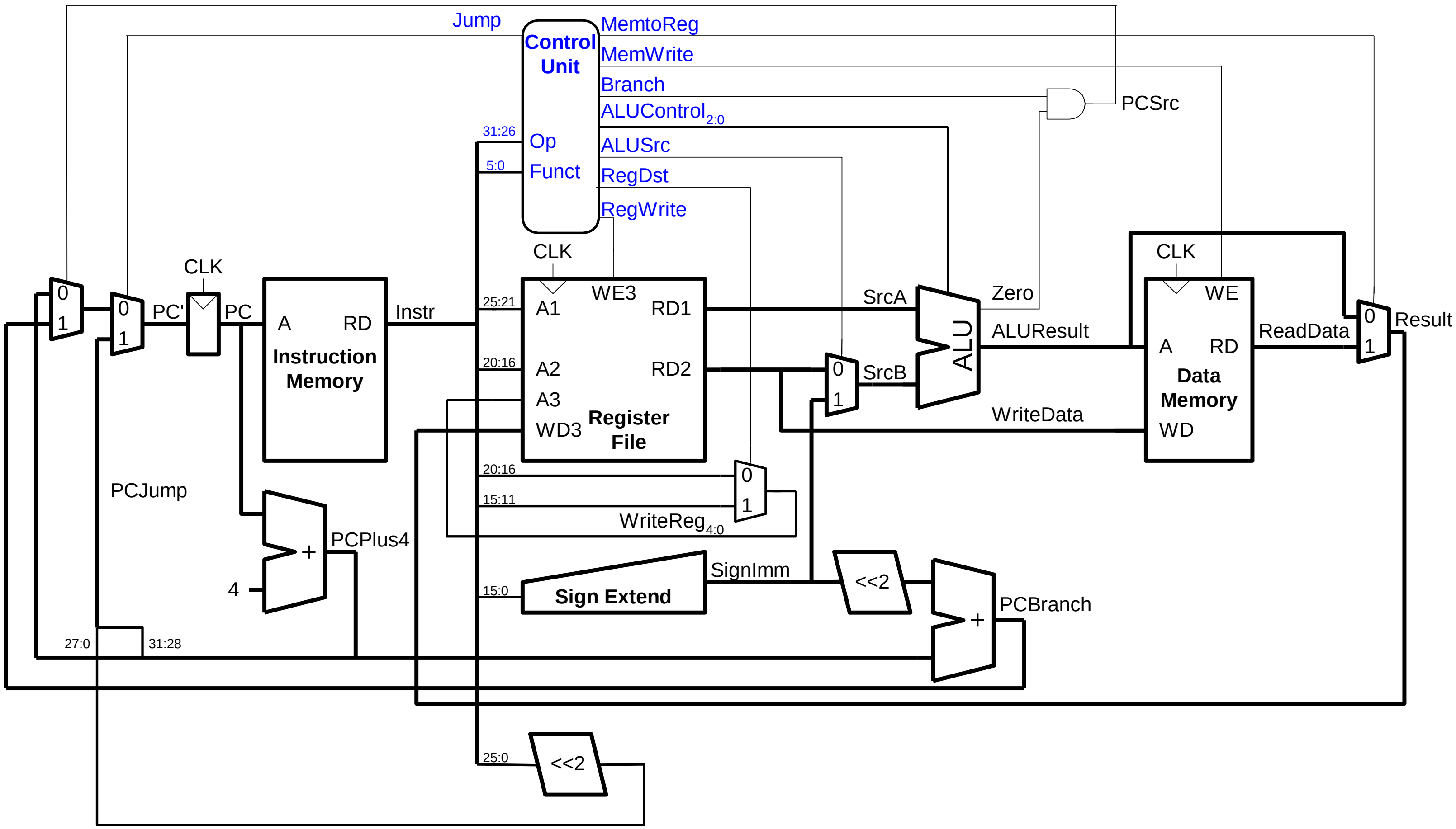
$$
\begin{aligned}
T_c &= t_{pcq\_PC} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup} \\
&= [30 + 25 + 250 + 20]\ \text{ps} \\
&= 325\ \text{ps}
\end{aligned}
$$

Multi-Cycle Performance Example

- For a program with 100 billion instructions executing on a multi-cycle MIPS processor
  - CPI = 4.12
  - T_c = 325 ps
- *Execution Time* = (# instructions) × CPI × T_c
  = $(100 \times 10^{9})(4.12)(325 \times 10^{-12})$
  = 133.9 seconds
- This is slower than the single-cycle processor (92.5 seconds). Why?

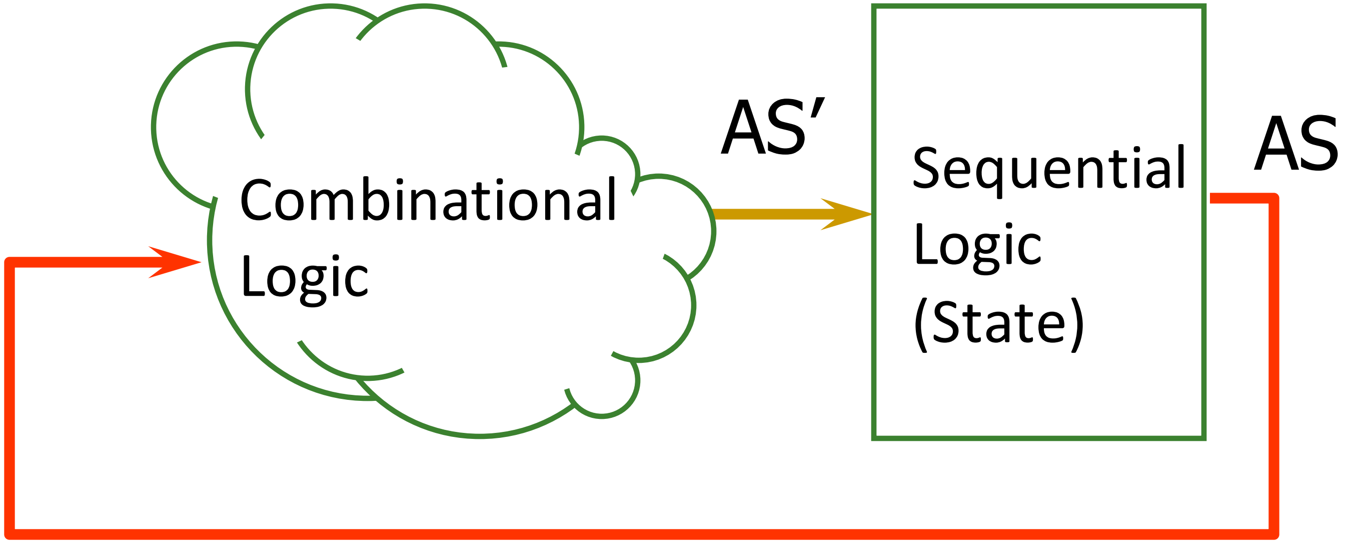
- Did we break the stages in a balanced manner?
- Overhead of register setup/hold paid many times
- How would the results change with different assumptions on memory latency and instruction mix?

Review: Single-Cycle MIPS Processor



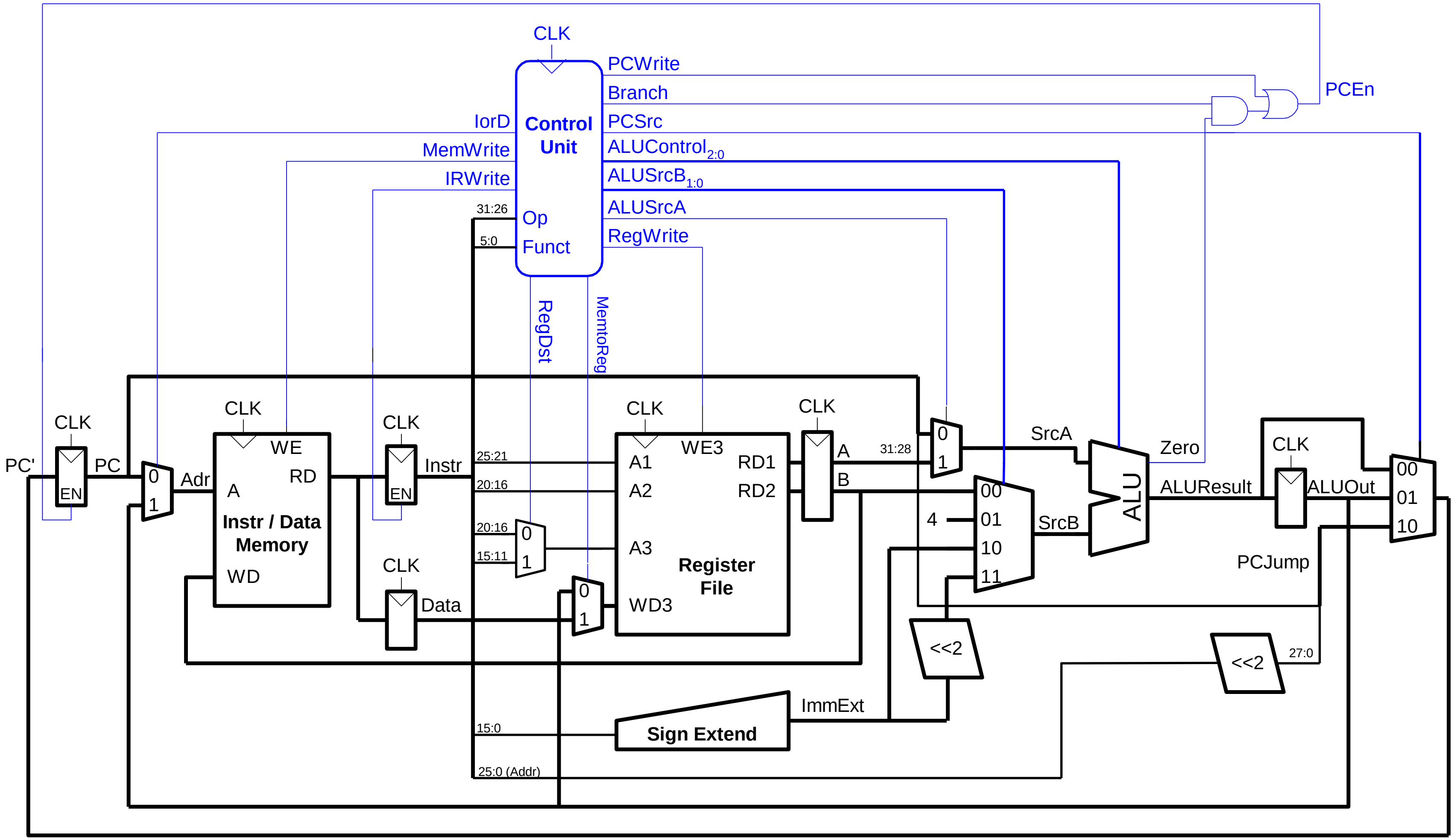
Review: Single-Cycle MIPS FSM

- Single-cycle machine

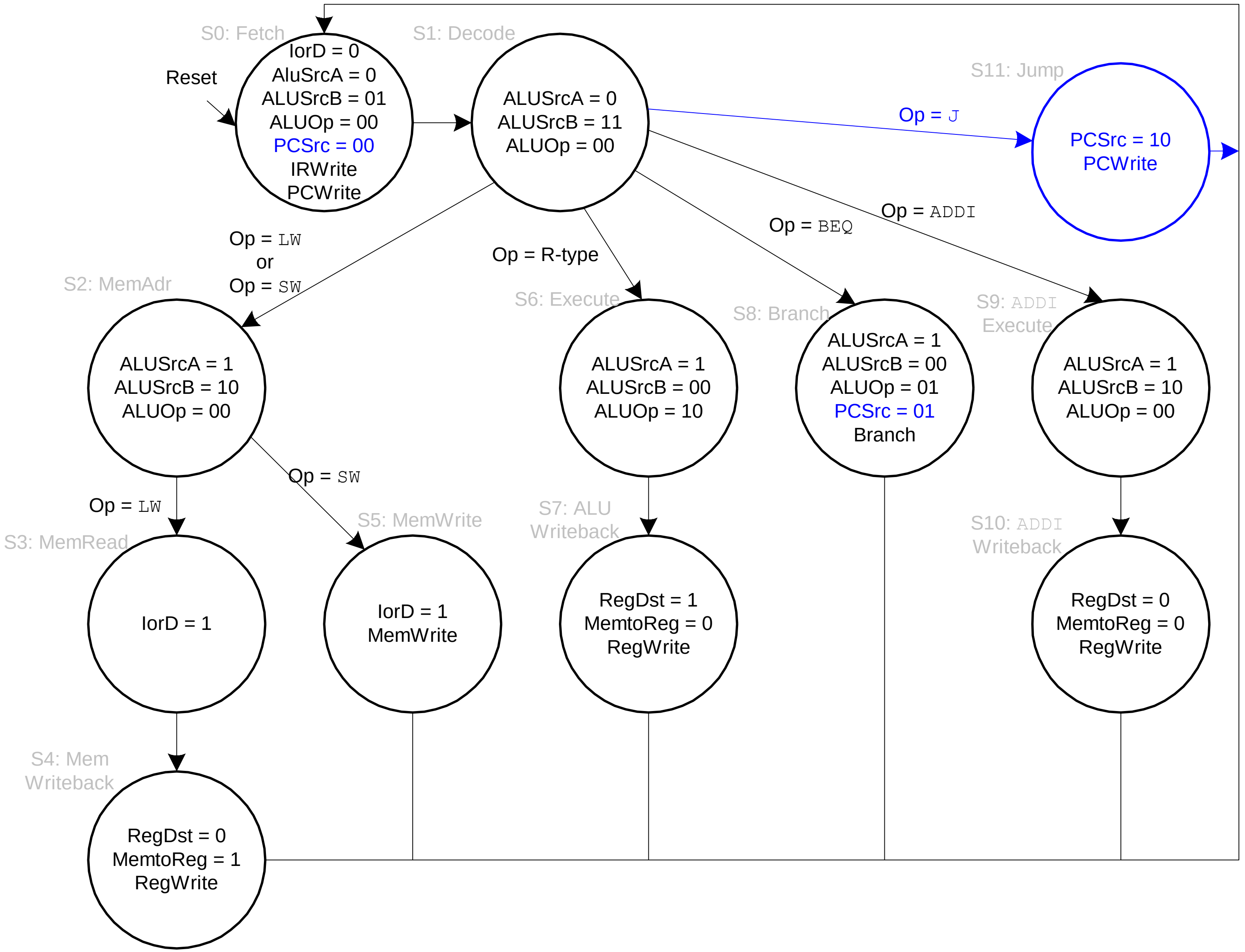


AS: Architectural State

Review: Multi-Cycle MIPS Processor



Review: Multi-Cycle MIPS FSM



What is the shortcoming of this design?

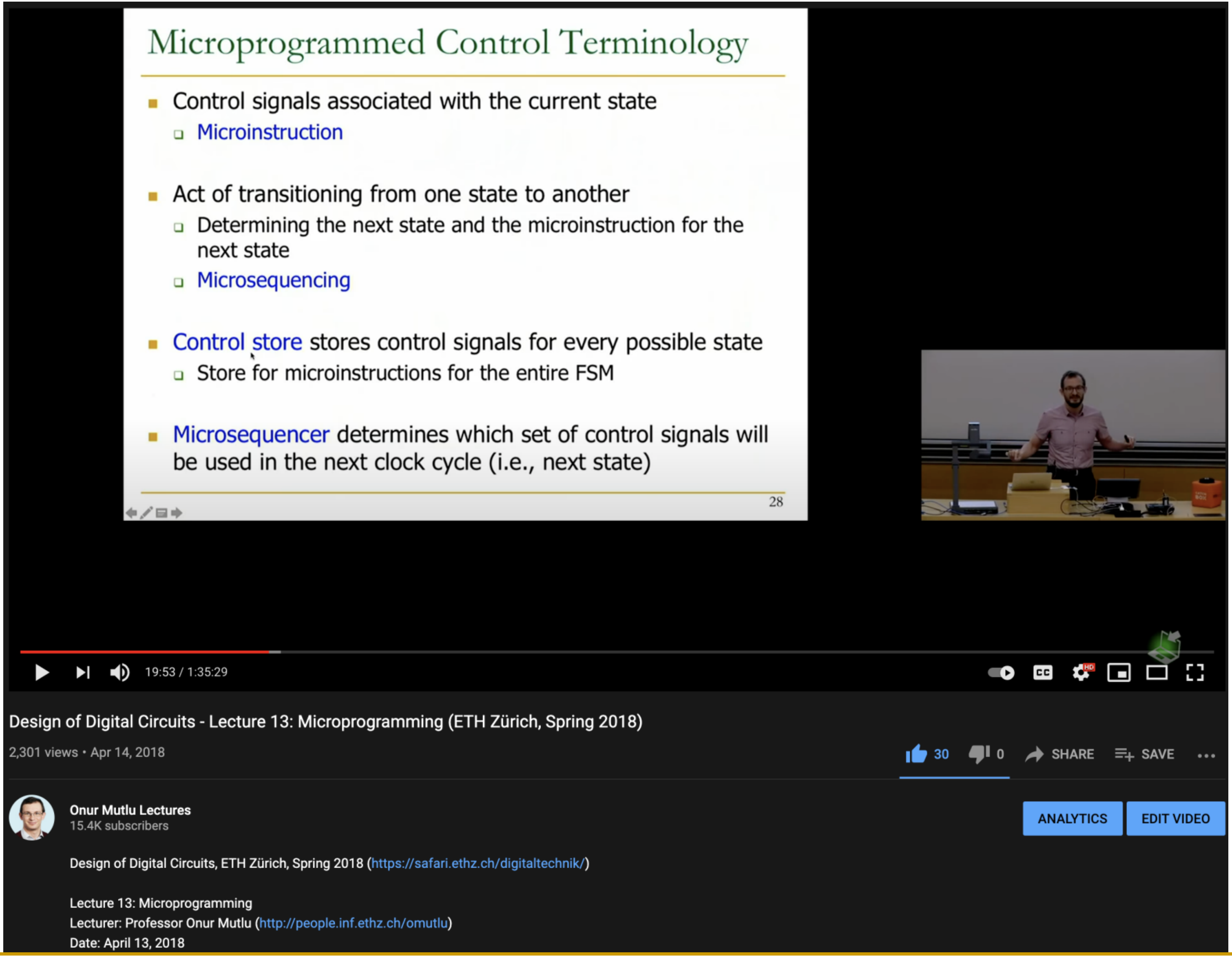
What does this design assume about memory?

What If Memory Takes > One Cycle?

- Stay in the same “memory access” state until memory returns the data
- “Memory Ready?” bit is an input to the control logic that determines the next state

Backup Slides on **Microprogrammed** Multi-Cycle Microarchitectures

These Slides Are Covered in A Past Lecture



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)

28

Design of Digital Circuits - Lecture 13: Microprogramming (ETH Zürich, Spring 2018)

2,301 views • Apr 14, 2018

Onur Mutlu Lectures
15.4K subscribers

Design of Digital Circuits, ETH Zürich, Spring 2018 (https://safari.ethz.ch/digitaltechnik/)

Lecture 13: Microprogramming
Lecturer: Professor Onur Mutlu (http://people.inf.ethz.ch/omutlu)
Date: April 13, 2018

https://www.youtube.com/onurmutlulectures

Lectures on Microprogrammed Designs

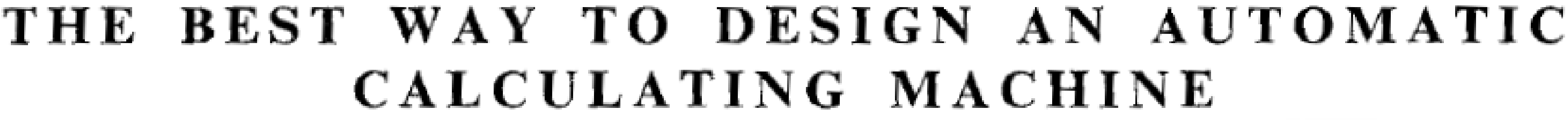
- Design of Digital Circuits, Spring 2018, Lecture 13
  - Microprogramming (ETH Zürich, Spring 2018)
  - https://www.youtube.com/watch?v=u4GhShuBP3Y&list=PL5Q2soXY2Zi_QedyPWtRmFUJ2F8DdYP7l&index=13

- Computer Architecture, Spring 2013, Lecture 7
  - Microprogramming (CMU, Spring 2013)
  - https://www.youtube.com/watch?v=_igvSl5h8cs&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=7

Another Example: **Microprogrammed** Multi-Cycle Microarchitecture

How Do We Implement This?

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



THE BEST WAY TO DESIGN AN AUTOMATIC CALCULATING MACHINE

By M. V. Wilkes, M.A., Ph.D., F.R.A.S.

- An elegant implementation:
  - The concept of microcoded/microprogrammed machines

Recall: 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 two things:
  - How the datapath should process the data
  - How to generate the control signals for the (next) clock cycle

Microprogrammed Control Terminology

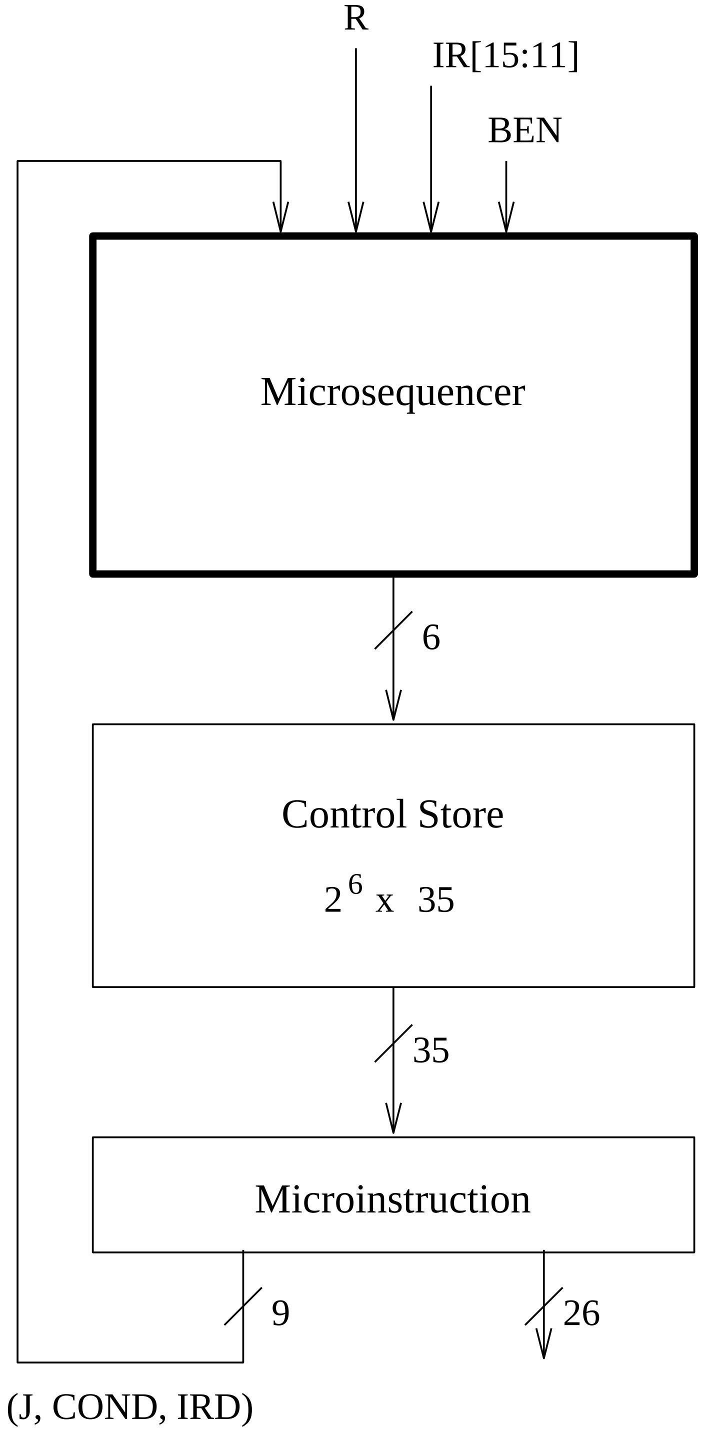
- Control signals associated with the current state
  - Microinstruction

- 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)

Example Control Structure



Simple Design
of the Control Structure

What Happens In A Clock Cycle?

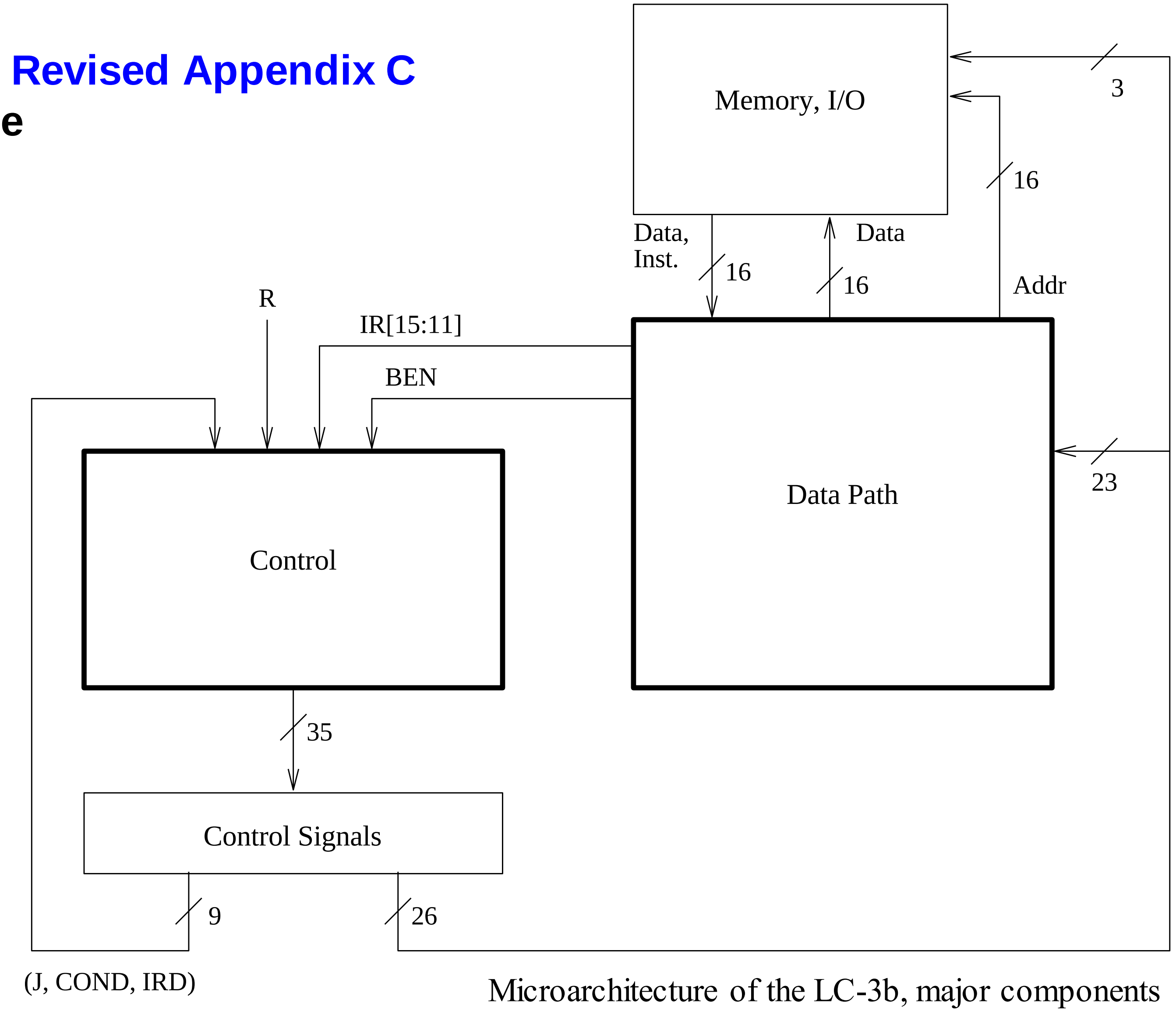
- The control signals (microinstruction) for the current state control two things:
  - Processing in the data path
  - Generation of control signals (microinstruction) for the next cycle
  - *See Supplemental Figure 1 (next-next slide)*

- Datapath and microsequencer operate concurrently

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

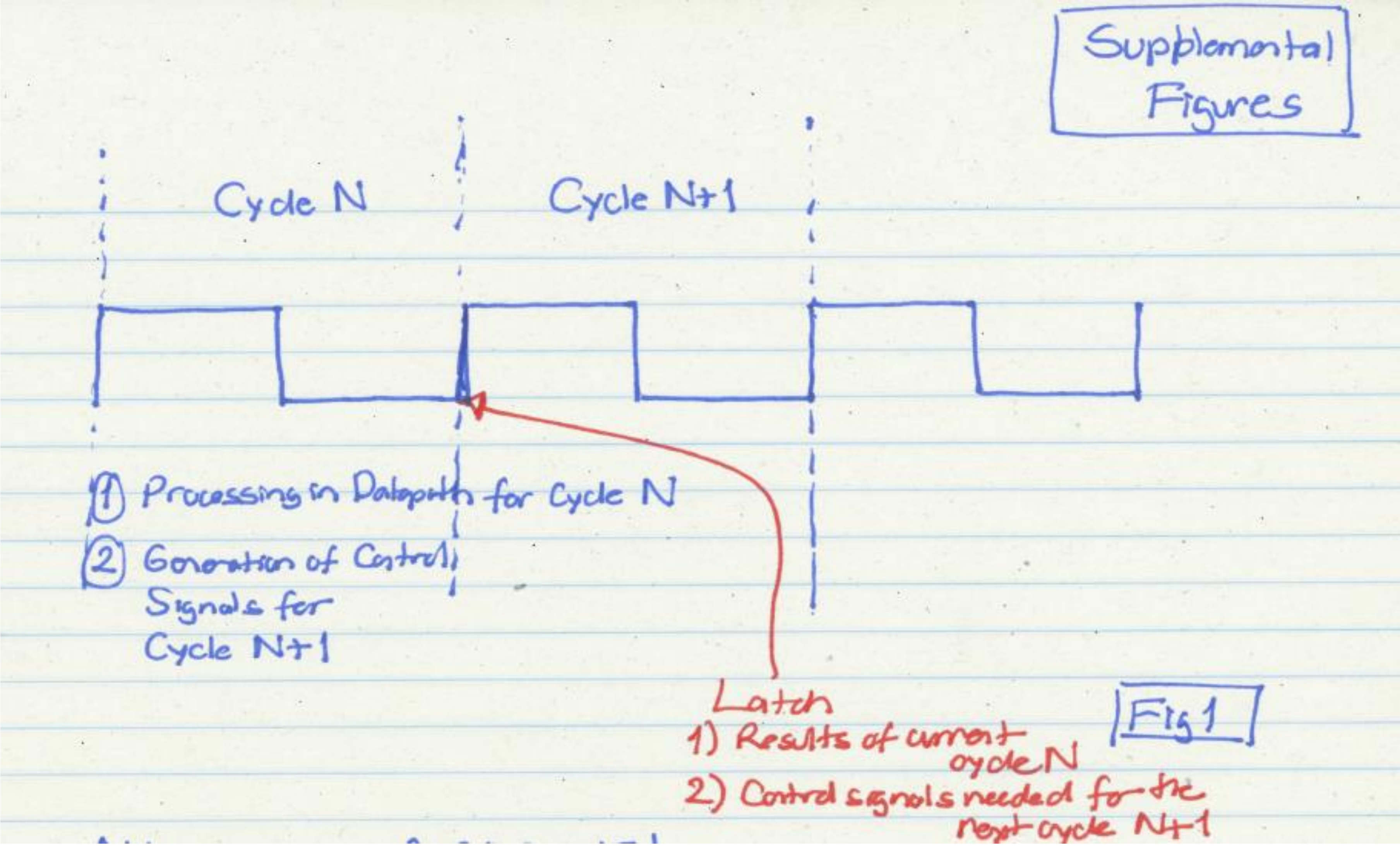
Example uProgrammed Control & Datapath

Read P&P Revised Appendix C
On website



Microarchitecture of the LC-3b, major components

A Clock Cycle

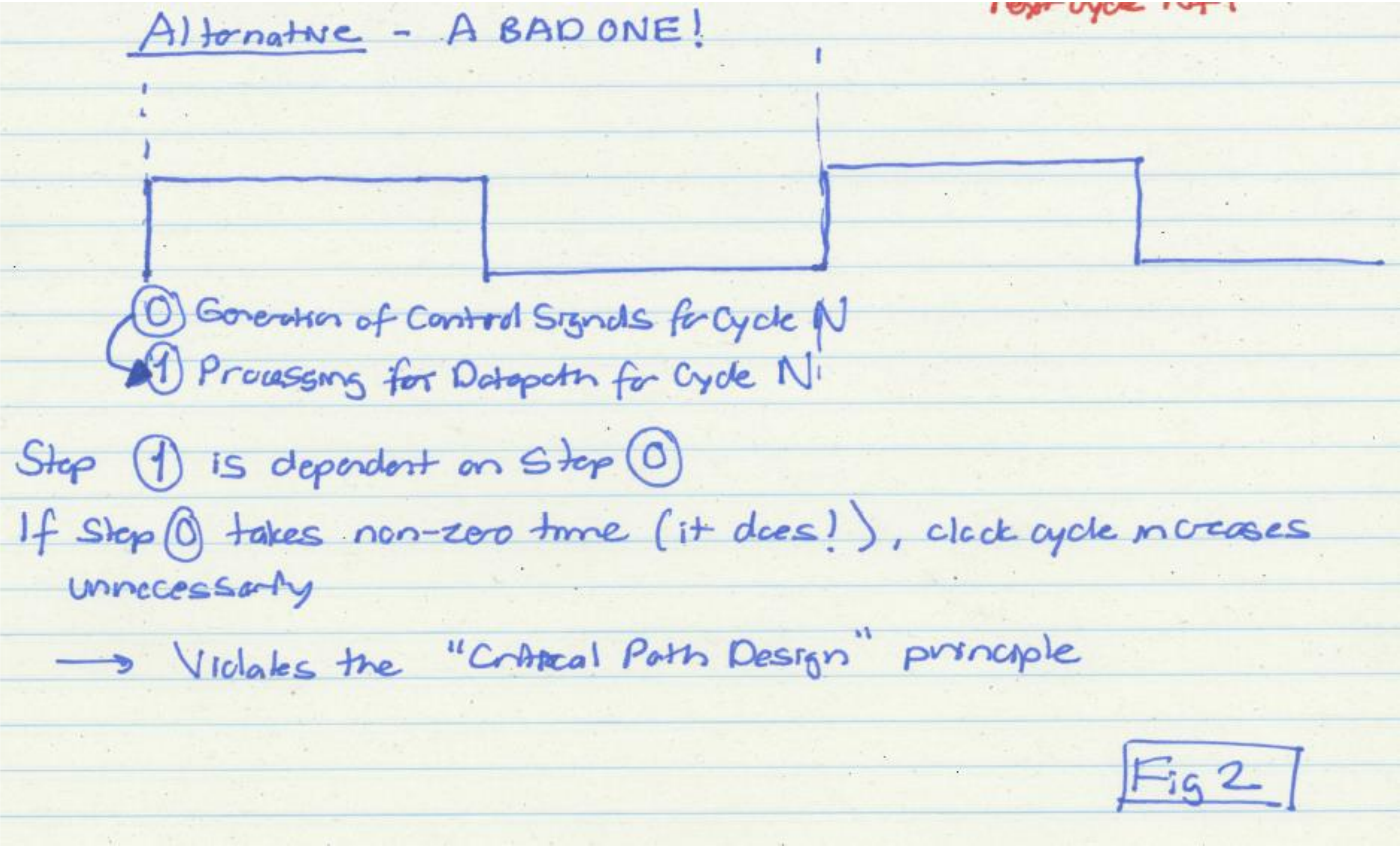


1) Processing in Datapath for Cycle N

2) Generation of Control Signals for Cycle N+1

Latch
1) Results of current cycle N
2) Control signals needed for the next cycle N+1

A Bad Clock Cycle!



Next Cycle N+1

Alternative - A BAD ONE!

⓪ Generation of Control Signals for Cycle N

① Processing for Datapath for Cycle N

Step ① is dependent on Step ⓪

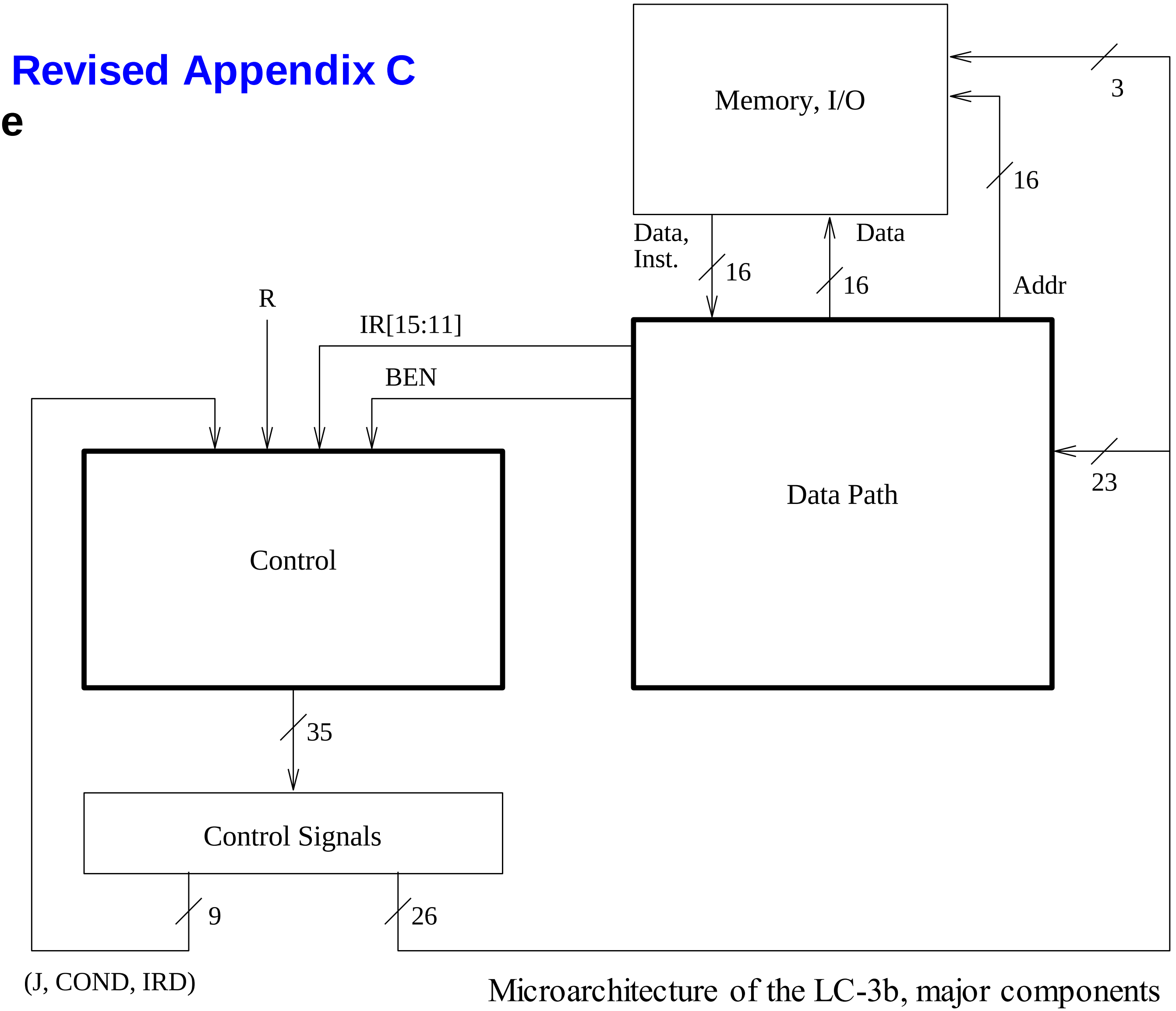
If Step ⓪ takes non-zero time (it does!), clock cycle increases unnecessarily

→ Violates the "Critical Path Design" principle

Fig 2

A Simple LC-3b Control and Datapath

Read P&P Revised Appendix C
On website

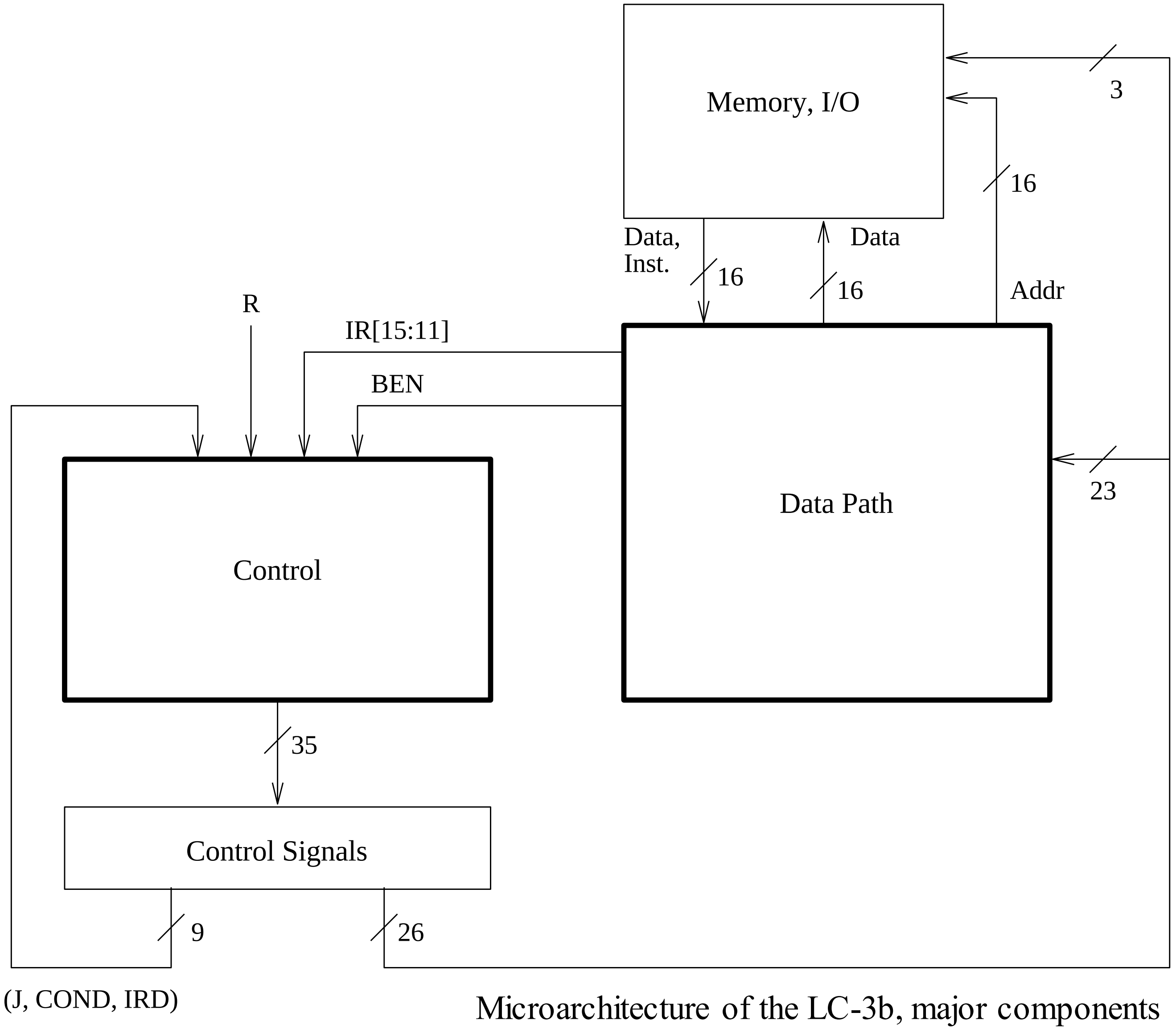


Microarchitecture of the LC-3b, major components

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



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))

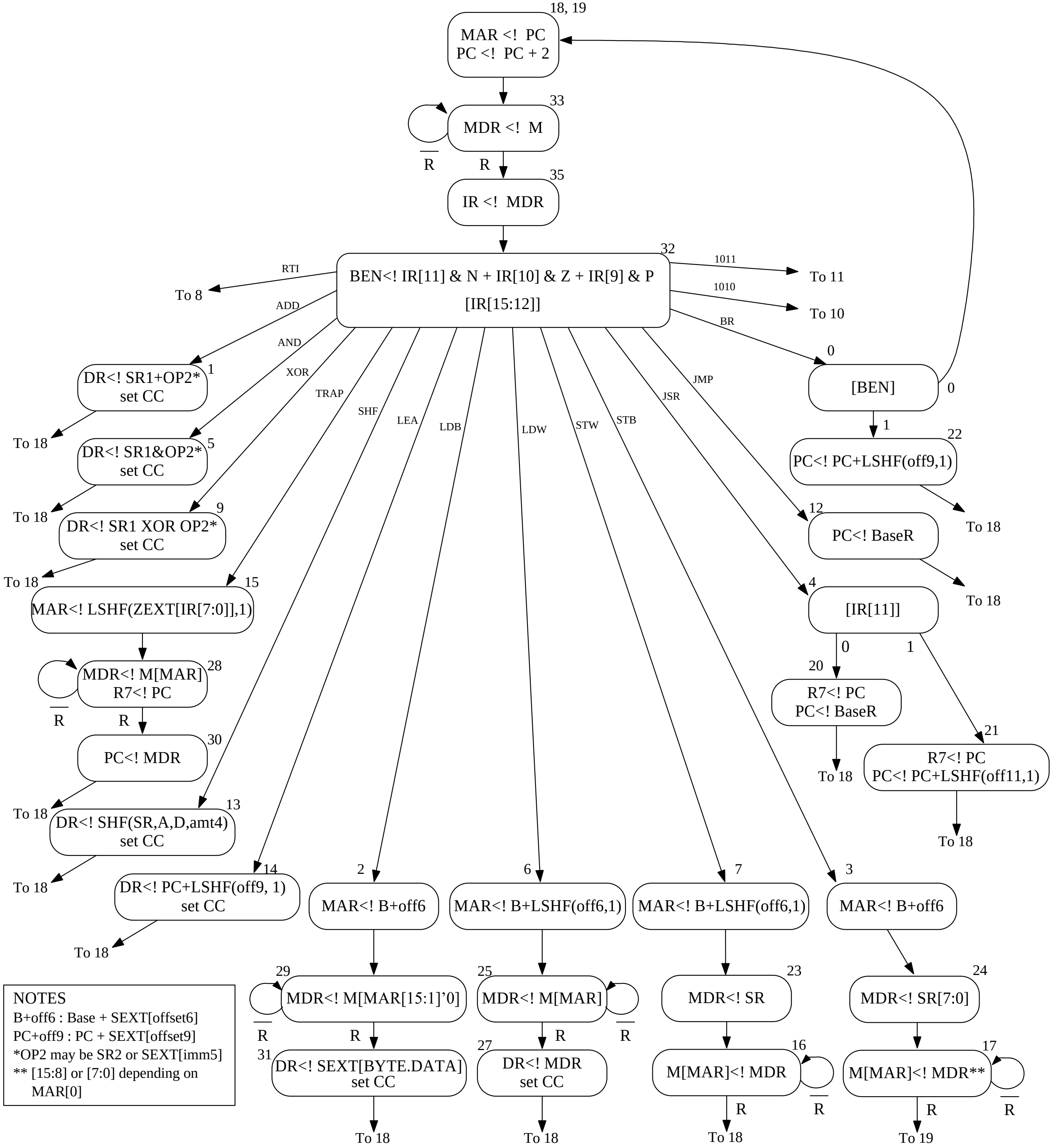
An LC-3b State Machine

- Patt and Patel, Revised Appendix 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
R̄
R

35
IR <! MDR

32
BEN<! IR[11] & N + IR[10] & Z + IR[9] & P
[IR[15:12]]

RTI
To 8
1011
To 11
1010
To 10
BR
ADD
AND
XOR
TRAP
SHF
LEA
LDB
LDW
STW
STB
JSR
JMP

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
R̄
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

2
MAR<! B+off6

29
MDR<! M[MAR[15:1]'0]
R̄
R

31
DR<! SEXT[BYTE.DATA]
set CC
To 18

6
MAR<! B+LSHF(off6,1)

25
MDR<! M[MAR]
R
R̄

27
DR<! MDR
set CC
To 18

7
MAR<! B+LSHF(off6,1)

23
MDR<! SR

16
M[MAR]<! MDR
R
R̄
To 18

3
MAR<! B+off6

24
MDR<! SR[7:0]

17
M[MAR]<! MDR**
R
R̄
To 19

0
[BEN]
0
1

22
PC<! PC+LSHF(off9,1)
To 18

12
PC<! BaseR
To 18

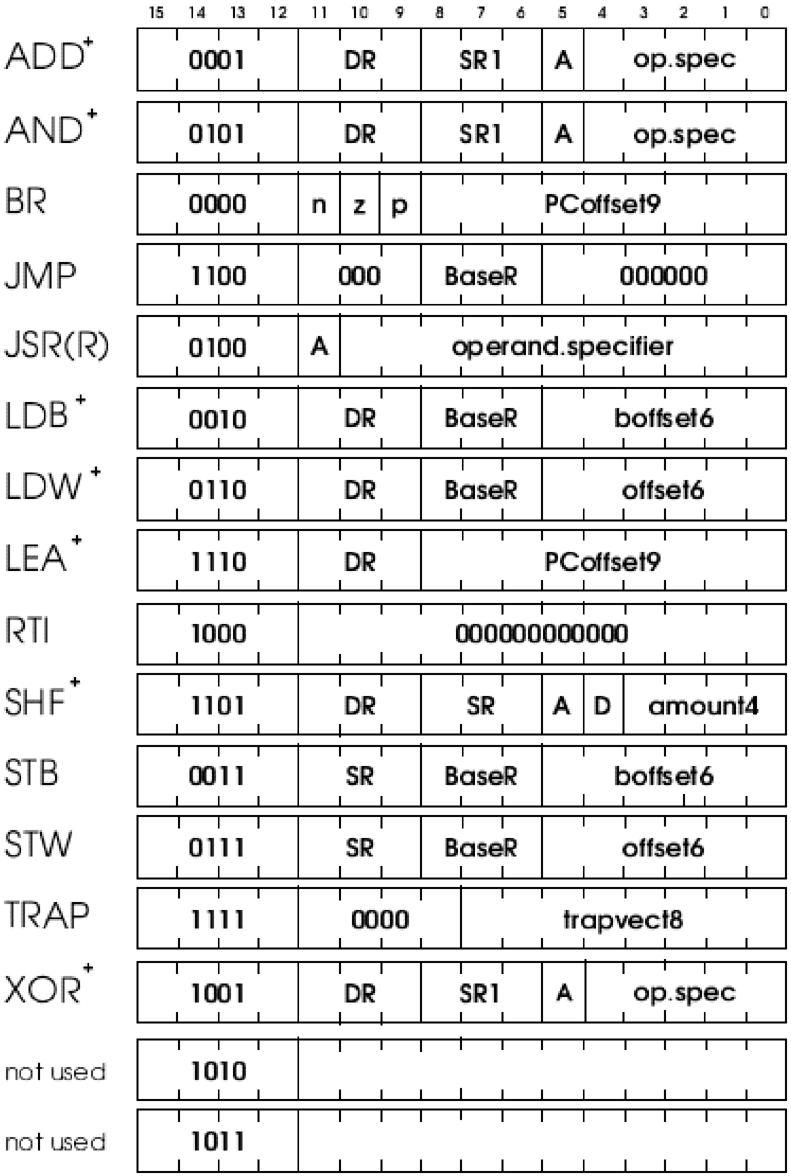
4
[IR[11]]
0
1

20
R7<! PC
PC<! BaseR
To 18

21
R7<! PC
PC<! PC+LSHF(off11,1)
To 18

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]

The FSM Implements the LC-3b ISA



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD^{+}	0001				DR			SR1			A	op.spec				
AND^{+}	0101				DR			SR1			A	op.spec				
BR	0000				n	z	p	PCoffset9								
JMP	1100				000			BaseR			000000					
JSR(R)	0100				A	operand.specifier										
LDB^{+}	0010				DR			BaseR			boffset6					
LDW^{+}	0110				DR			BaseR			offset6					
LEA^{+}	1110				DR			PCoffset9								
RTI	1000				000000000000											
SHF^{+}	1101				DR			SR			A	D	amount4			
STB	0011				SR			BaseR			boffset6					
STW	0111				SR			BaseR			offset6					
TRAP	1111				0000				trapvect8							
XOR^{+}	1001				DR			SR1			A	op.spec				
not used	1010															
not used	1011															

- P&P Appendix A (revised):
  - https://safari.ethz.ch/digitaltechnik/spring2018/lib/exe/fetch.php?media=pp-appendixa.pdf

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?

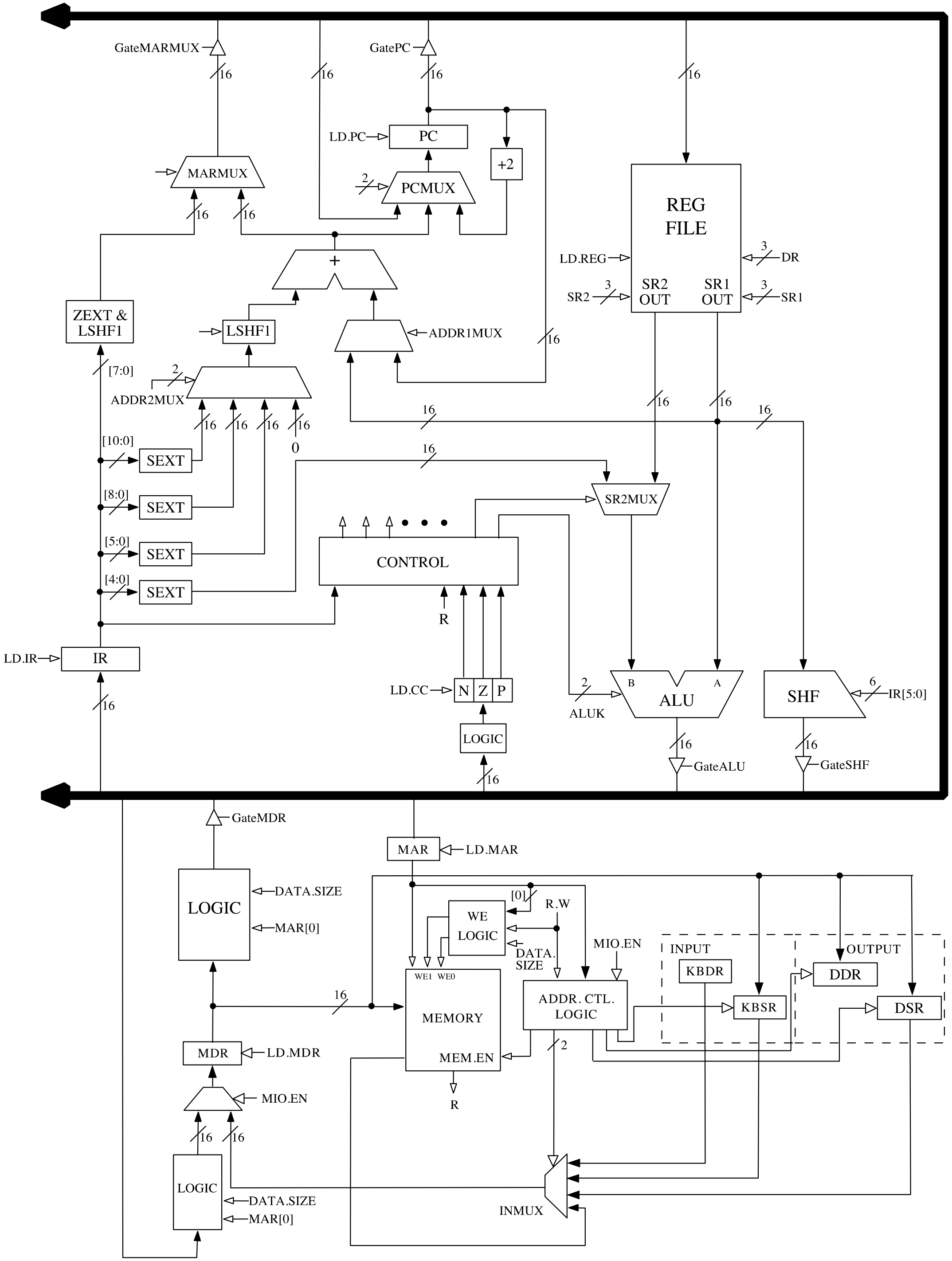
- What determines the clock cycle time?

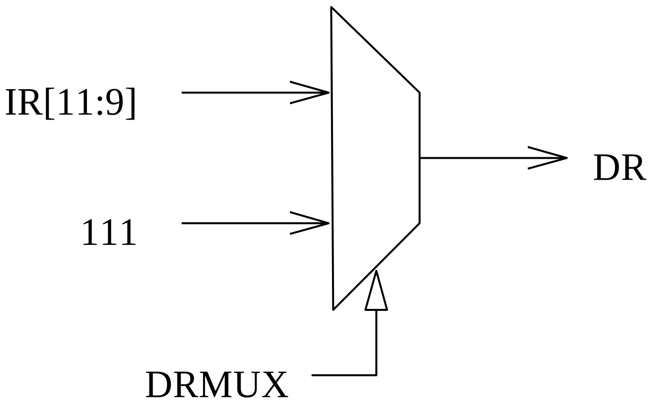
LC-3b Datapath

- Patt and Patel, Revised Appendix C, Figure C.3

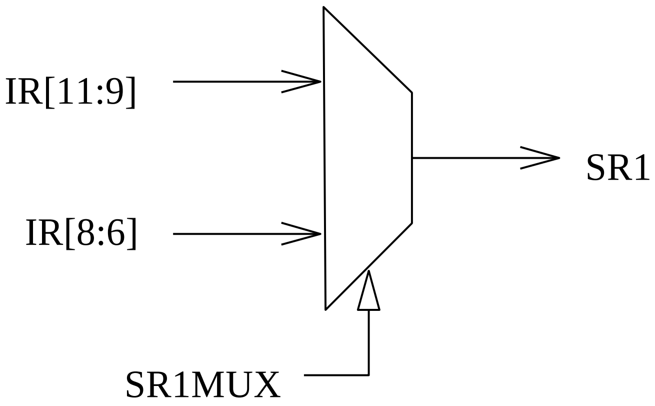
- Single-bus datapath design
  - At any point only one value can be "gated" on the bus (i.e., can be driving the bus)
  - Advantage: Low hardware cost: one bus
  - Disadvantage: Reduced concurrency – if instruction needs the bus twice for two different things, these need to happen in different states

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


GateMARMUX
GatePC
16
16
16
16
LD.PC
PC
+2
MARMUX
2
PCMUX
16
16
REG
FILE
LD.REG
3
DR
SR2
3
SR2
OUT
SR1
OUT
3
SR1
+
ZEXT &
LSHF1
LSHF1
ADDR1MUX
16
[7:0]
2
ADDR2MUX
16
16
16
16
0
16
16
16
[10:0]
SEXT
16
[8:0]
SEXT
SR2MUX
[5:0]
SEXT
CONTROL
[4:0]
SEXT
R
LD.IR
IR
LD.CC
N
Z
P
2
ALUK
B
A
ALU
SHF
6
IR[5:0]
16
LOGIC
16
16
GateALU
GateSHF
16
GateMDR
MAR
LD.MAR
LOGIC
DATA.SIZE
MAR[0]
[0]
R.W
WE
LOGIC
DATA.
SIZE
MIO.EN
INPUT
OUTPUT
KBDR
DDR
WE1
WE0
16
MEMORY
ADDR. CTL.
LOGIC
KBSR
DSR
MDR
LD.MDR
MEM.EN
2
R
MIO.EN
16
16
LOGIC
DATA.SIZE
MAR[0]
INMUX

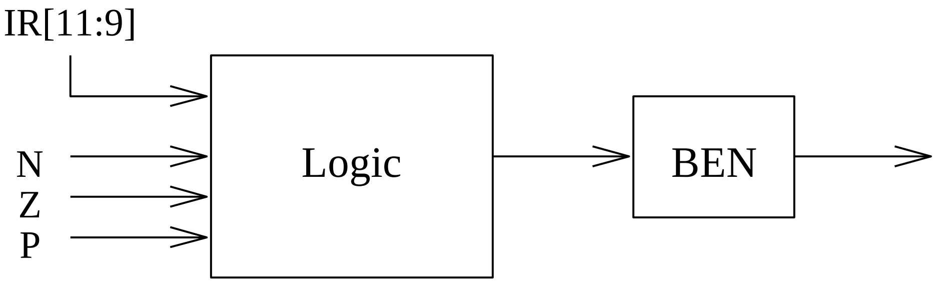


(a)

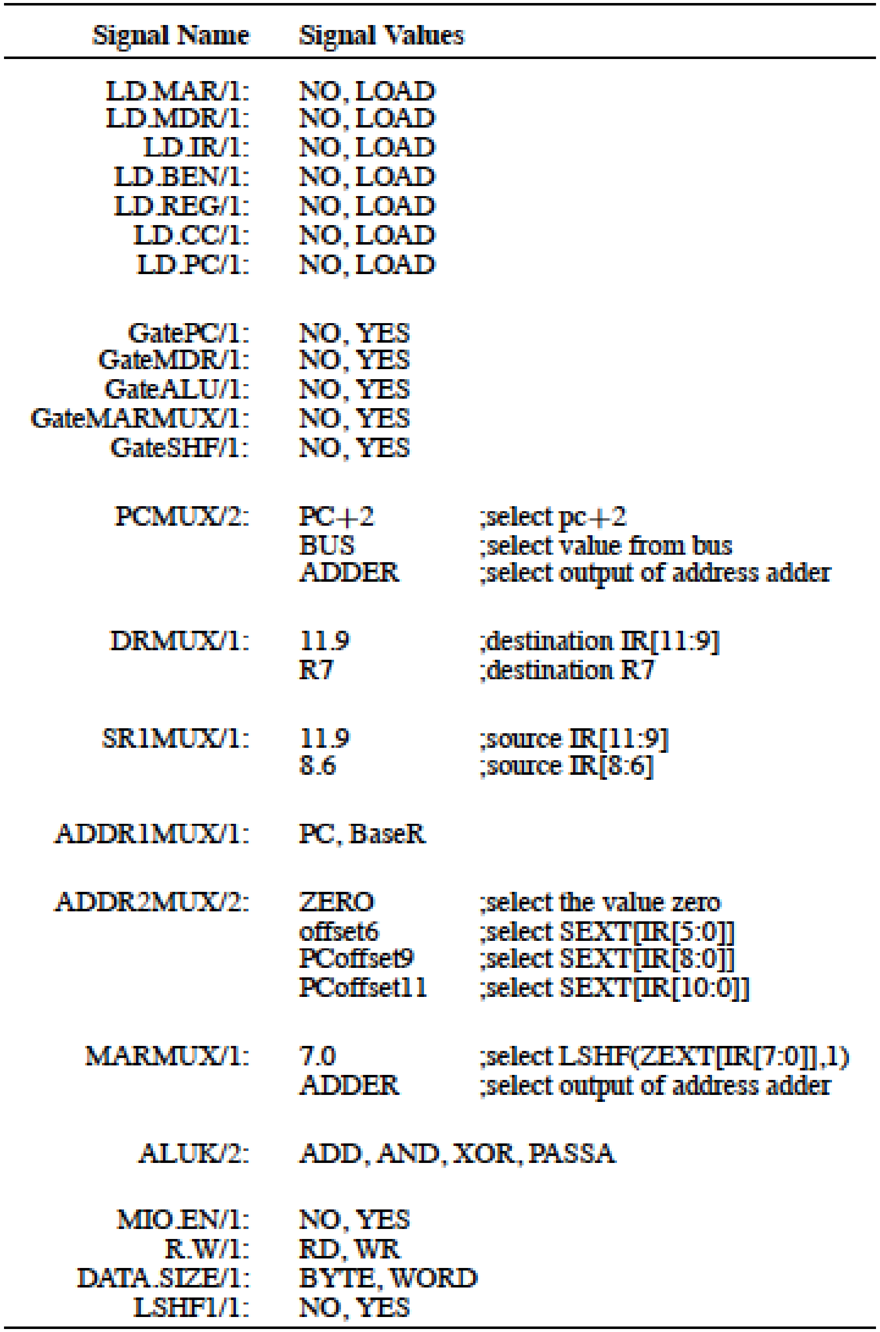


(b)

Remember the MIPS datapath



(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

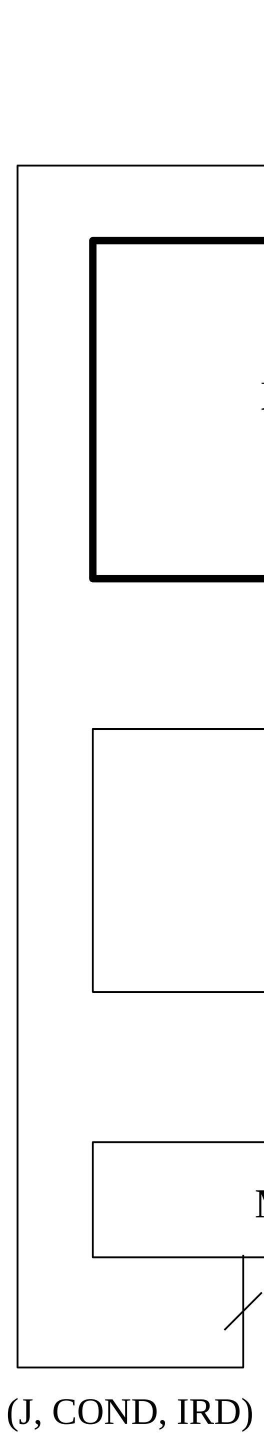
- How does instruction fetch happen in this datapath according to the state machine?

- What is the difference between gating and loading?
  - Gating: Enable/disable an input to be connected to the bus
    - Combinational: during a clock cycle
  - Loading: Enable/disable an input to be written to a register
    - Sequential: e.g., at a clock edge (assume at the end of cycle)

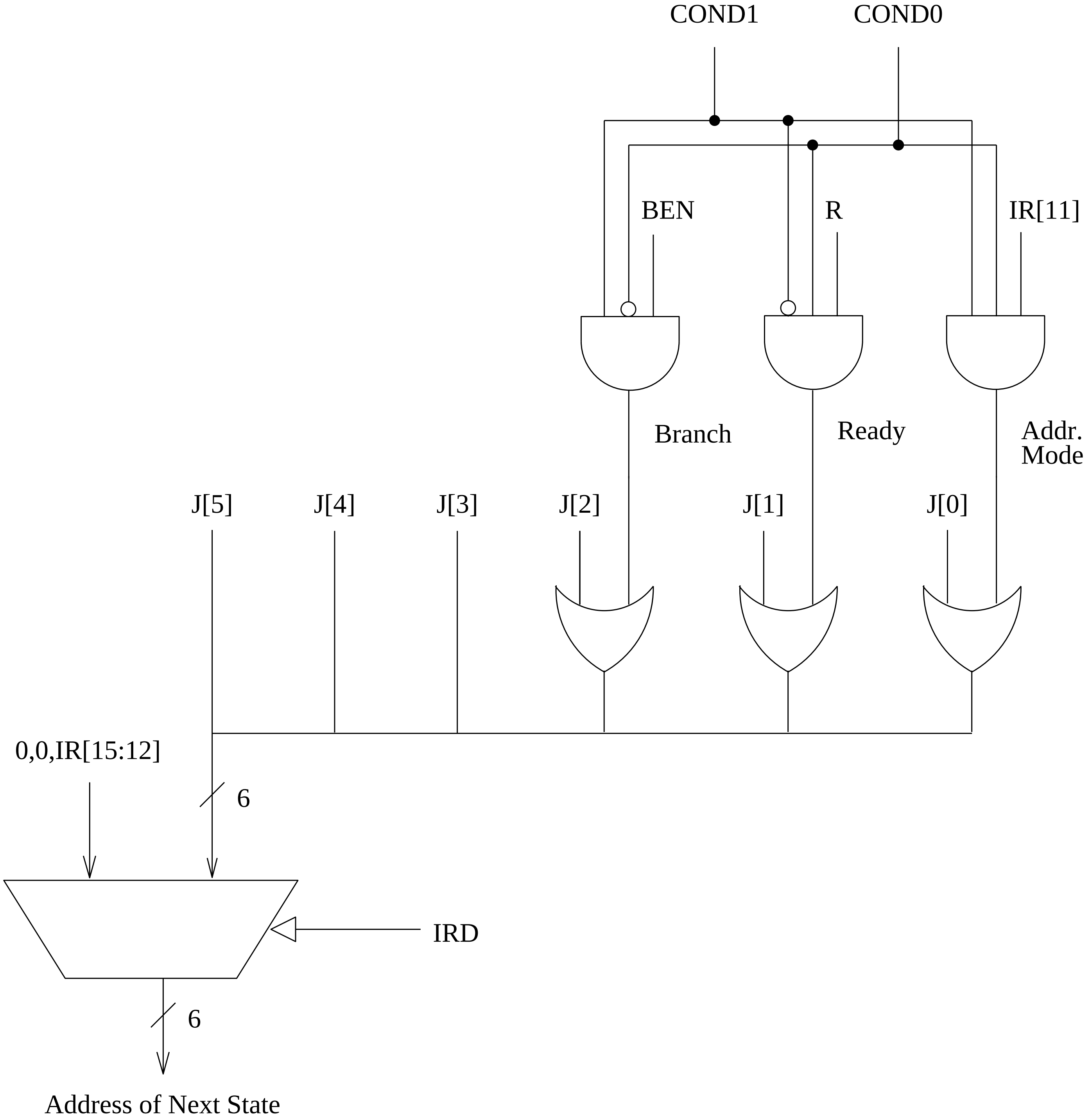
- Is this the smallest hardware you can design?

LC-3b Microprogrammed Control Structure

- Patt and Patel, Appendix C, Figure C.4
- Three components:
  - Microinstruction, control store, microsequencer
- Microinstruction: control signals that control the datapath (26 of them) and help 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)



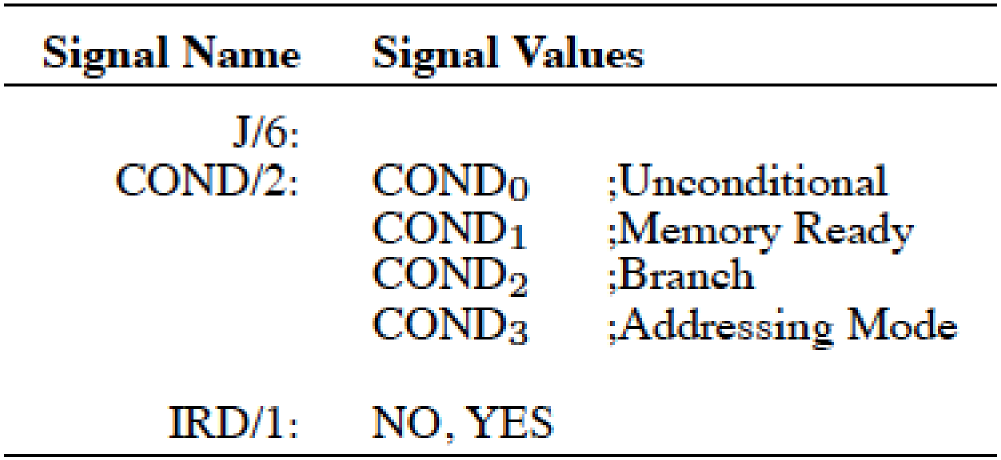
Simple Design of the Control Structure


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)

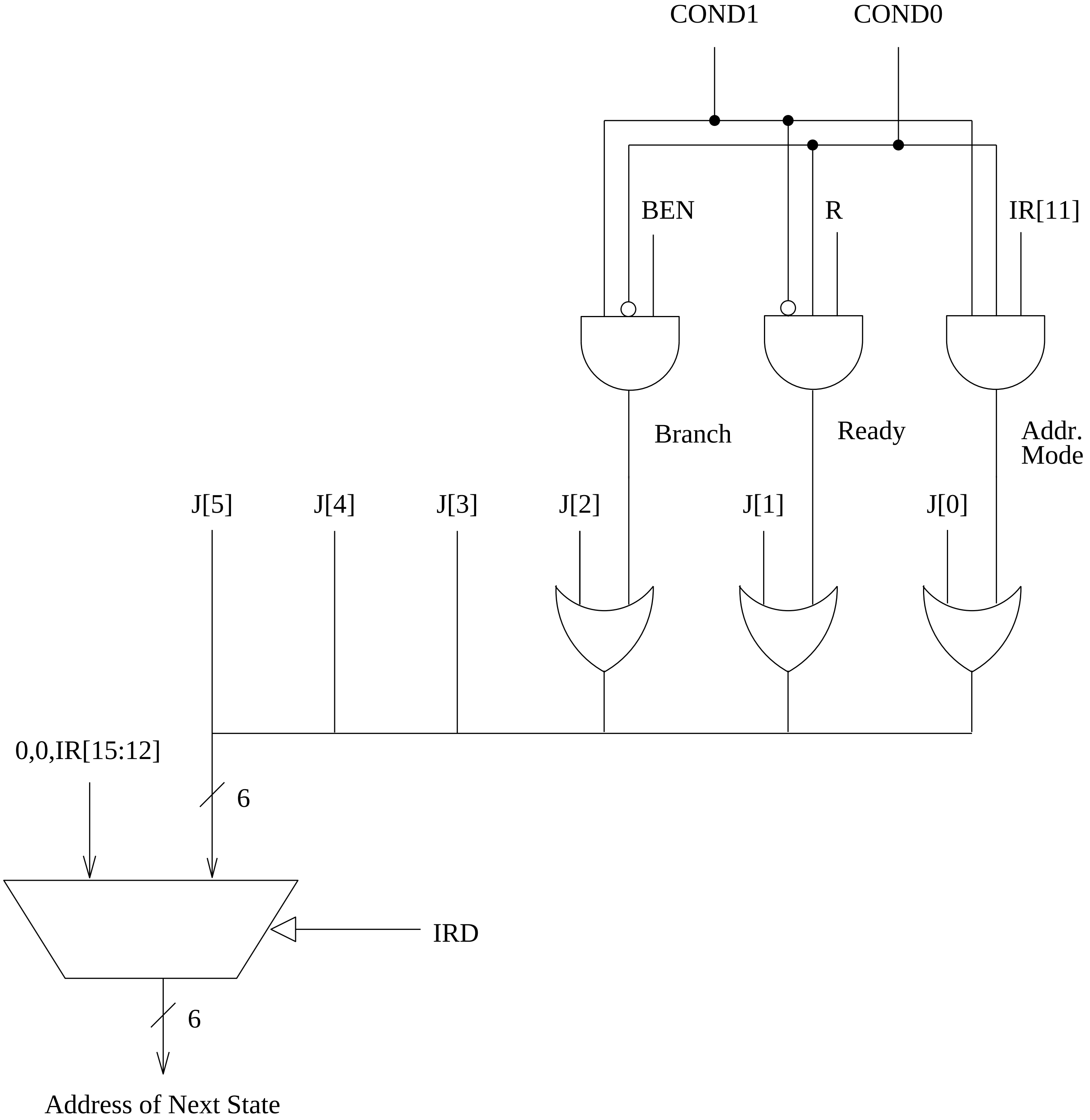
LC-3b Microsequencer

- Patt and Patel, Appendix C, Figure C.5
- The purpose of the microsequencer is to determine the address of the next microinstruction (i.e., next state)
  - Next state could be conditional or unconditional
- Next state address depends on 9 control signals (plus 7 data 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?

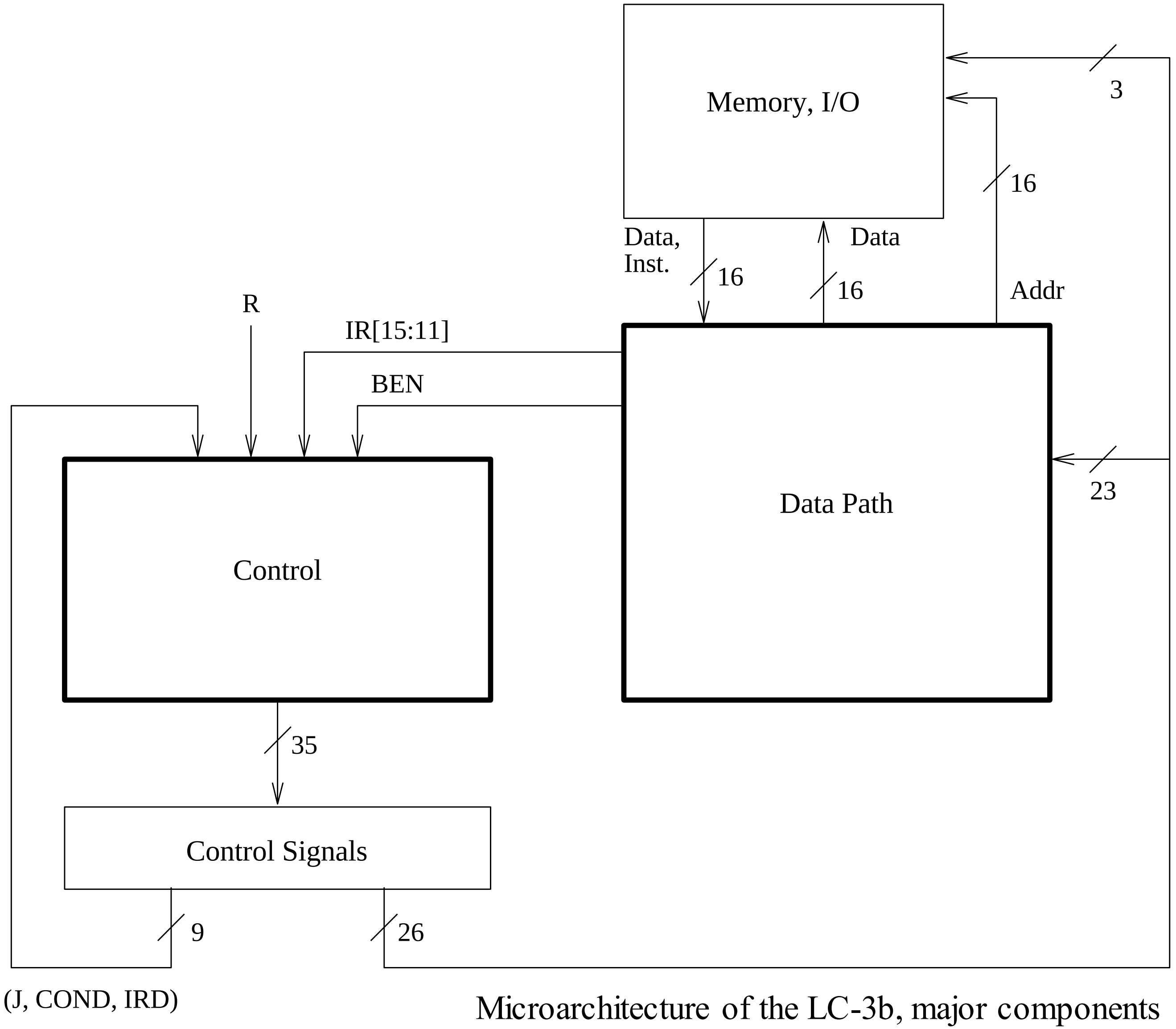
- How do you do the state encoding?
  - Minimize number of state variables (~ control store size)
  - Start with the 16-way branch
  - Then determine constraint tables and states dependent on COND

An Exercise in Microprogramming

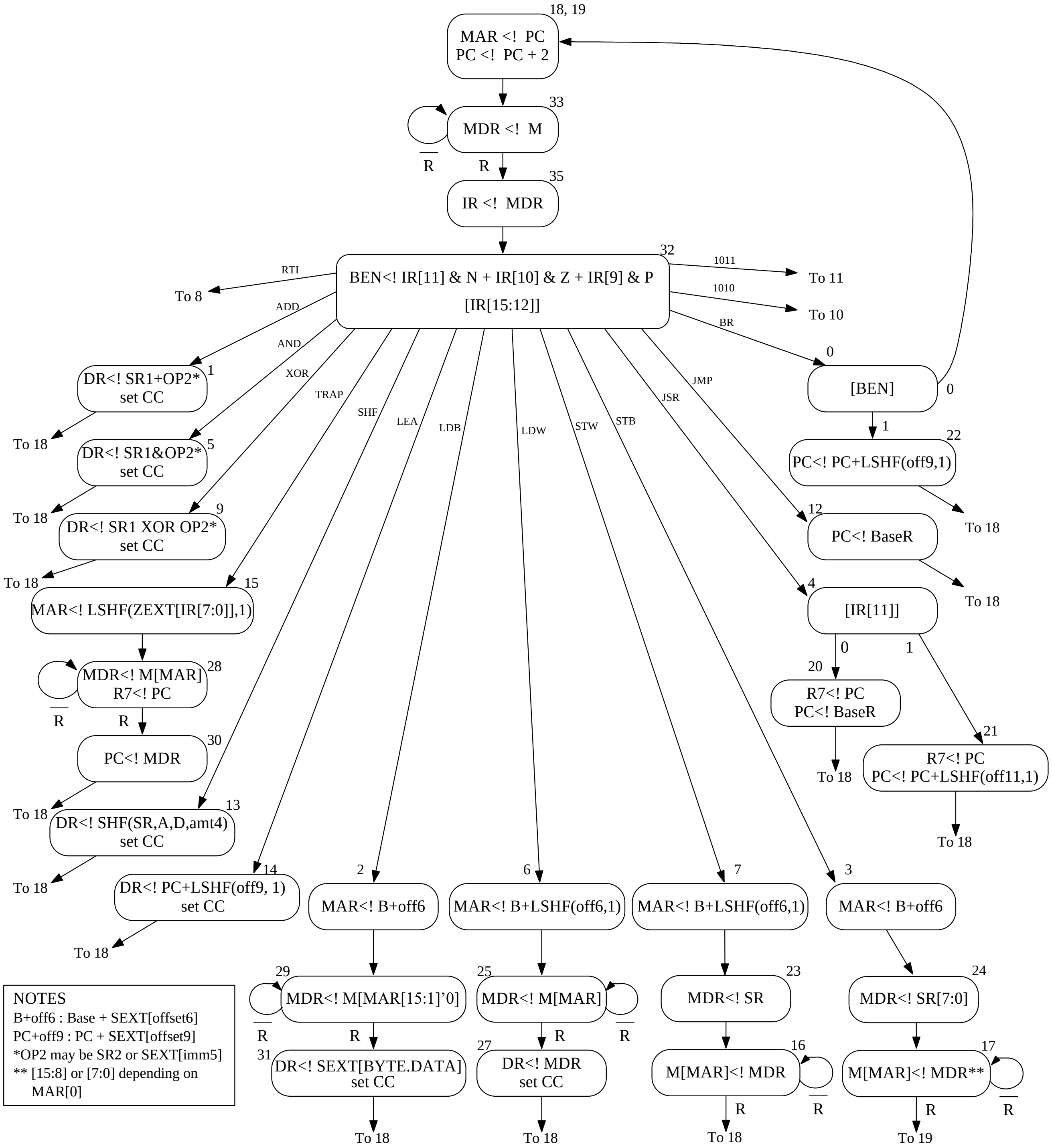
Handouts

- 7 pages of Microprogrammed LC-3b design
- https://safari.ethz.ch/digitaltechnik/spring2018/lib/exe/fetch.php?media=lc3b-figures.pdf

A Simple LC-3b Control and Datapath



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
1011
To 11
1010
To 10
BR
ADD
AND
XOR
TRAP
SHF
LEA
LDB
LDW
STW
STB
JSR
JMP

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

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

0
[BEN]
0
1

22
PC<! PC+LSHF(off9,1)
To 18

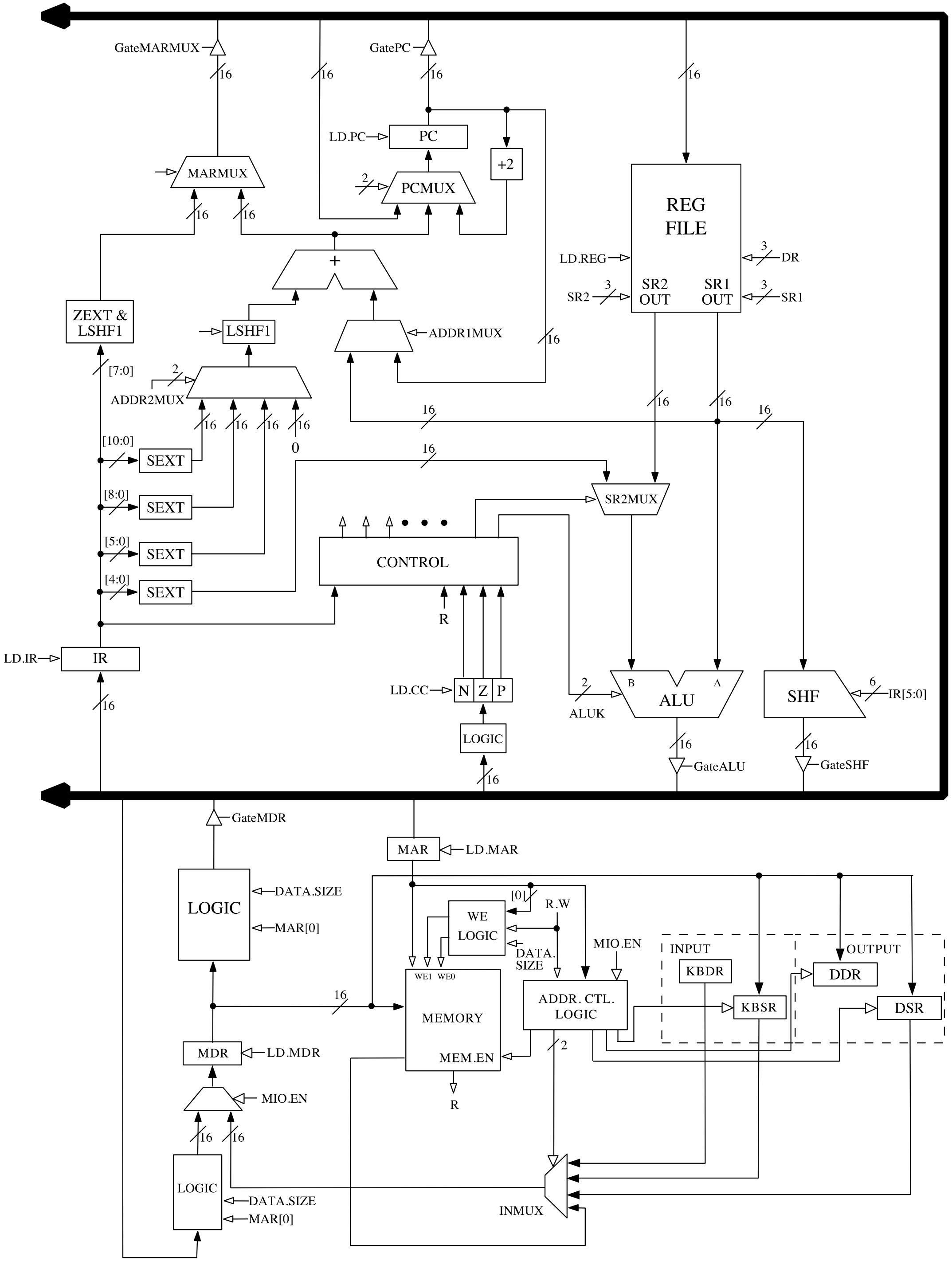
12
PC<! BaseR
To 18

4
[IR[11]]
0
1

20
R7<! PC
PC<! BaseR
To 18

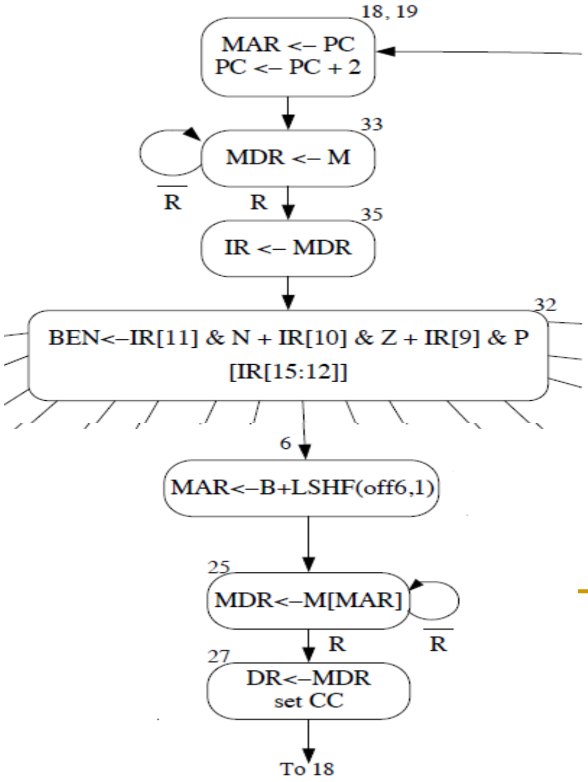
21
R7<! PC
PC<! PC+LSHF(off11,1)
To 18

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]

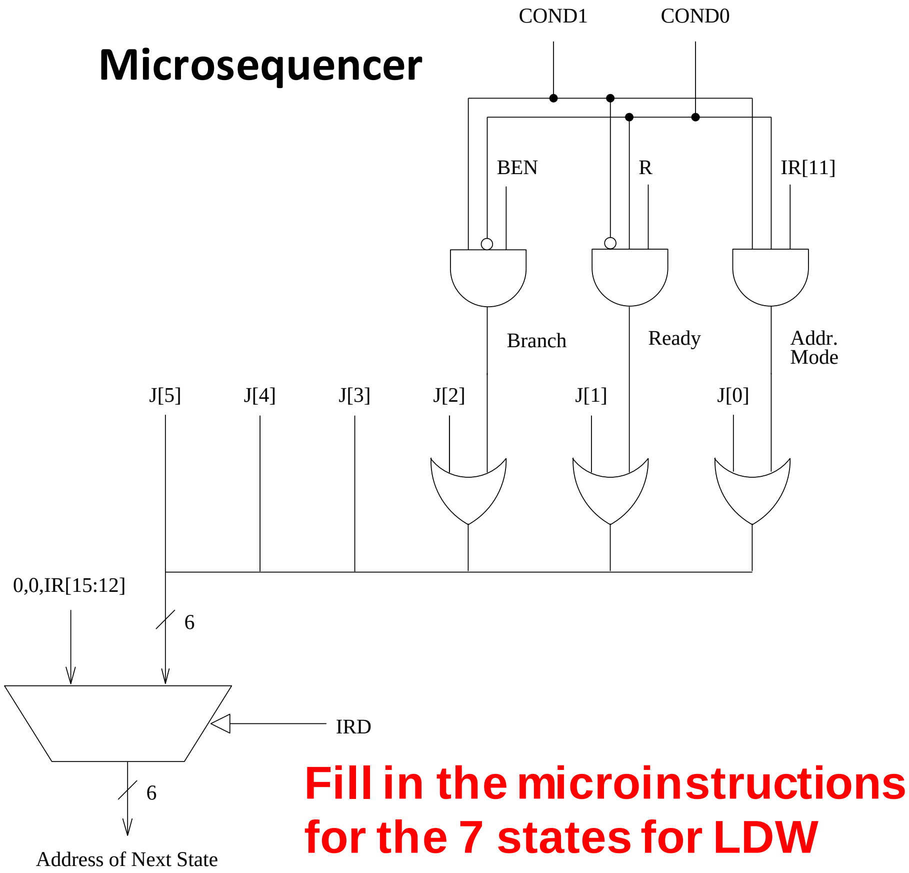


A Simple Datapath Can Become Very Powerful

State Machine for LDW

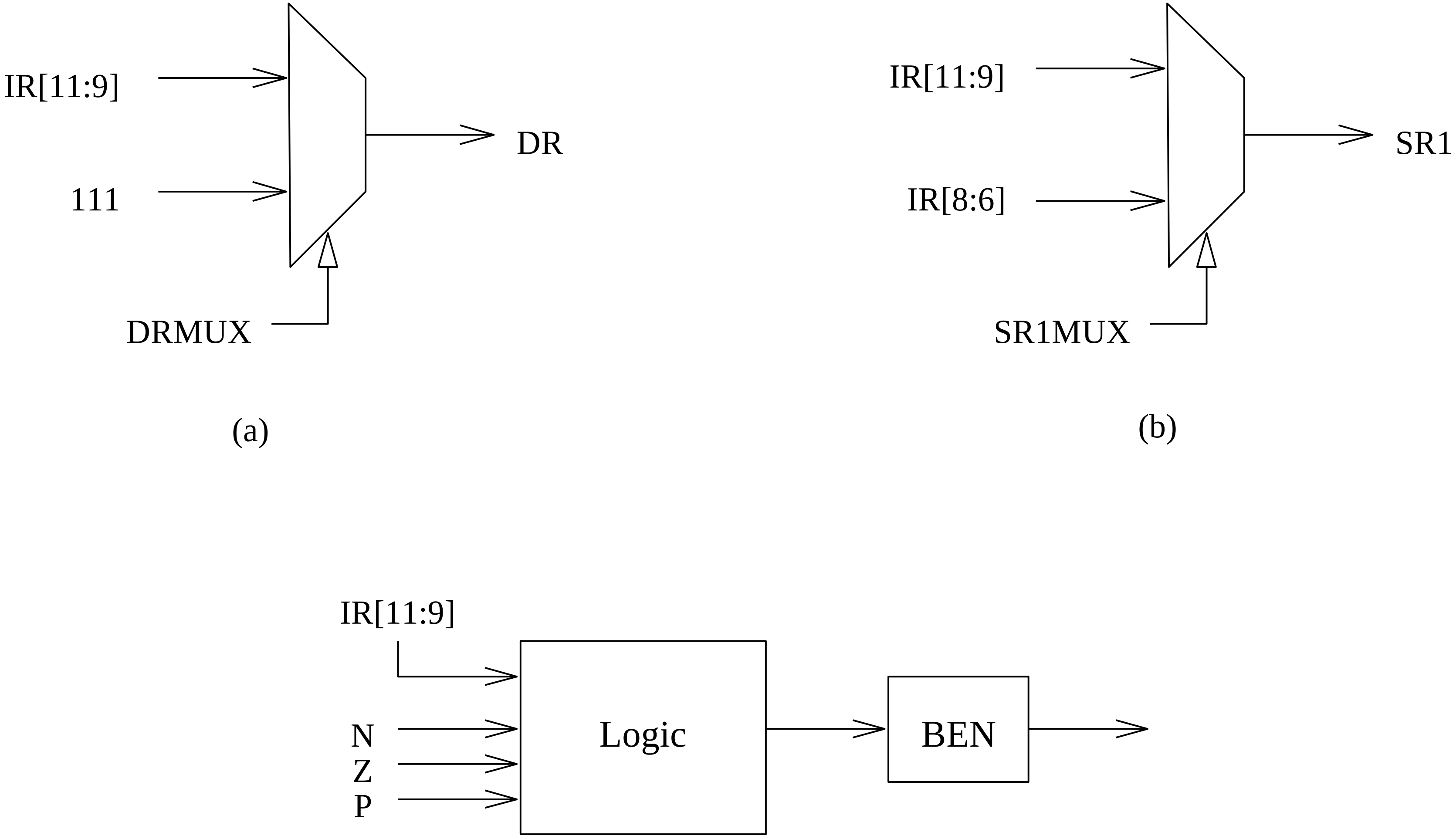


Microsequencer



Fill in the microinstructions for the 7 states for LDW

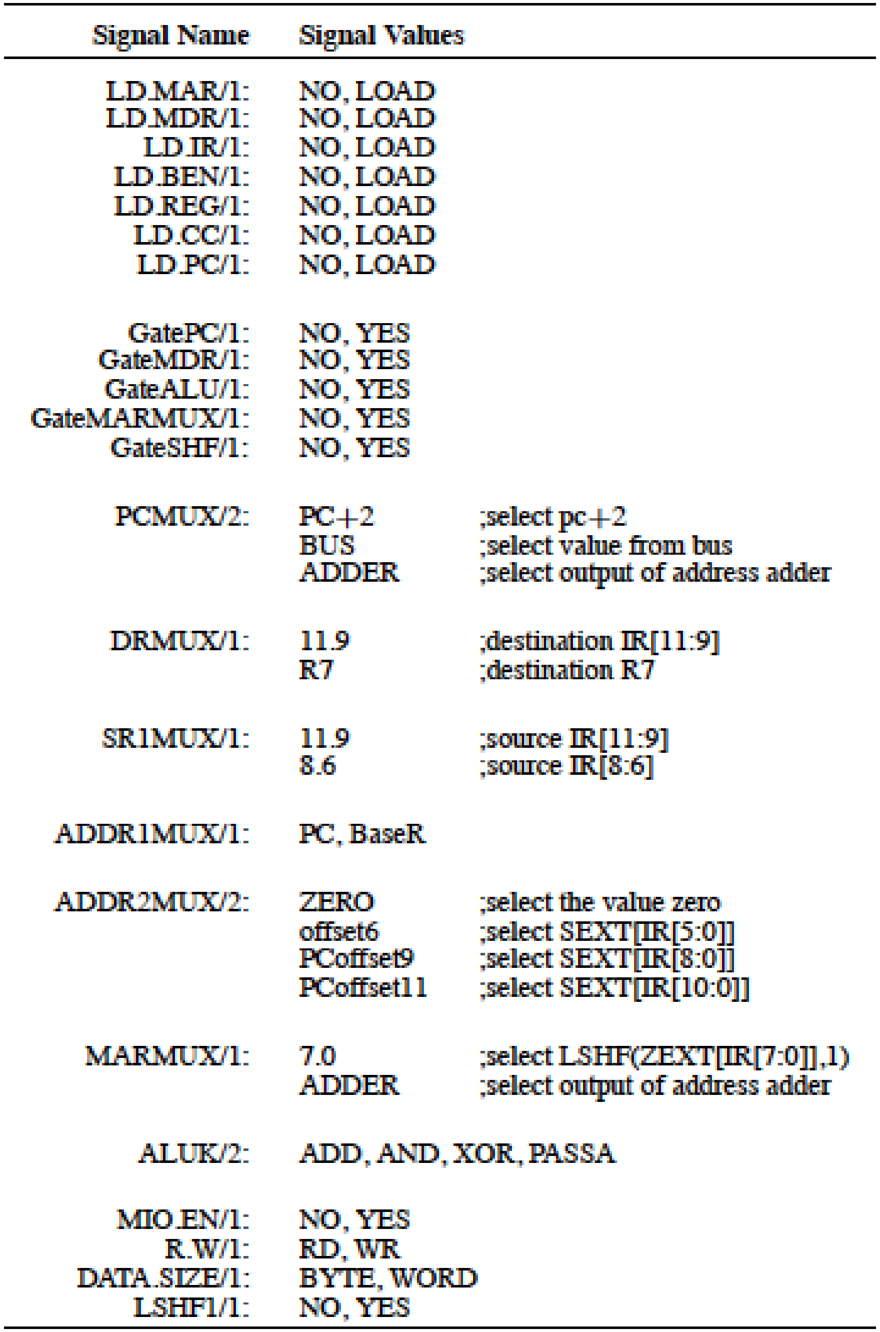
	IRD	Cond	J	LD.MAR	LD.MDR	LD.IR	LD.BEN	LD.REG	LD.CC	LD.PC	GatePC	GateMDR	GateALU	GateMARMUX	GateSHF	PCMUX	DRMUX	SR1MUX	ADDR1MUX	ADDR2MUX	MARMUX	ALUK	MIO.EN	R.W	DATA.SIZE	LSHF1
State 18 (010010)																										
State 33 (100001)																										
State 35 (100011)																				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



(a)

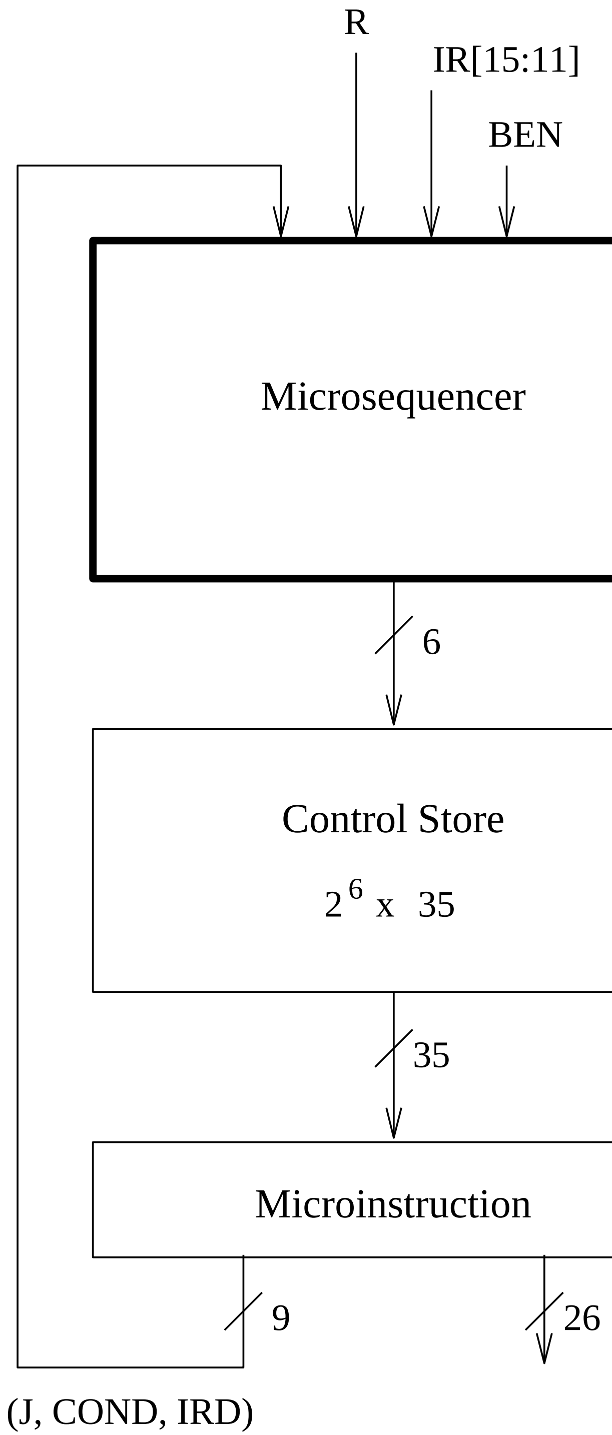
(b)

(c)

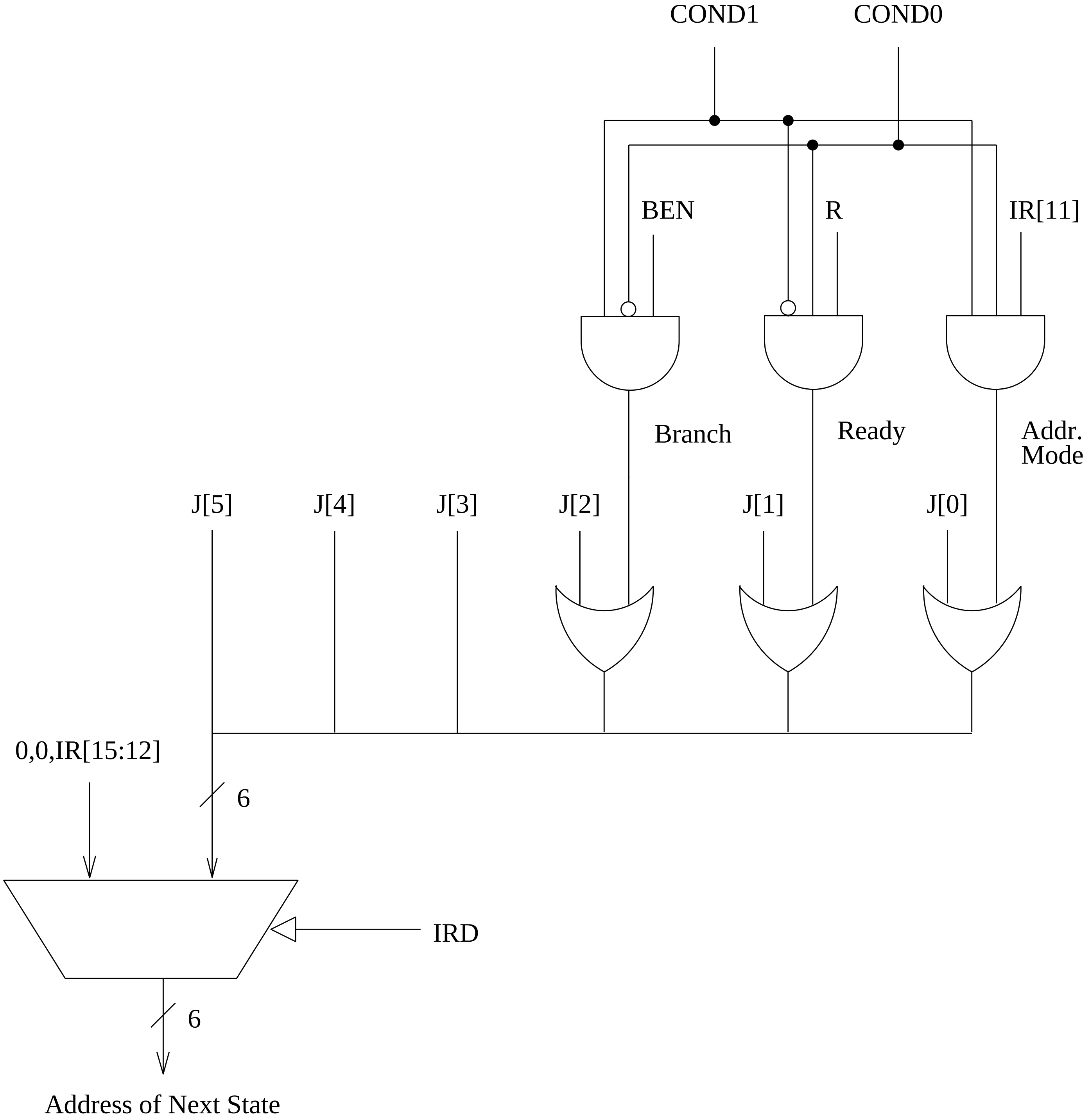


Signal Name	Signal Values	
LD.MAR/1:	NO, LOAD	
LD.MDR/1:	NO, LOAD	
LD.IR/1:	NO, LOAD	
LD.BEN/1:	NO, LOAD	
LD.REG/1:	NO, LOAD	
LD.CC/1:	NO, LOAD	
LD.PC/1:	NO, LOAD	
GatePC/1:	NO, YES	
GateMDR/1:	NO, YES	
GateALU/1:	NO, YES	
GateMARMUX/1:	NO, YES	
GateSHF/1:	NO, YES	
PCMUX/2:	PC+2	;select pc+2
	BUS	;select 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



Simple Design
of the Control Structure


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)

End of the Exercise in Microprogramming

Variable-Latency Memory

- The ready signal (R) enables memory read/write to execute correctly
  - Example: transition from state 33 to state 35 is controlled by the R bit asserted by memory when memory data is available

- Could we have done this in a single-cycle microarchitecture?

- What did we assume about memory and registers in a single-cycle microarchitecture?

The Microsequencer: Advanced Questions

- What happens if the machine is interrupted?

- What if an instruction generates an exception?

- How can you implement a complex instruction using this control structure?
  - Think REP MOVS instruction in x86
    - string copy of N elements starting from address A to address B

The Power of Abstraction

- The concept of a control store of microinstructions enables the hardware designer with a new abstraction: microprogramming

- The designer can translate any desired operation to a sequence of microinstructions
- All the designer needs to provide is
  - The sequence of microinstructions needed to implement the desired operation
  - The ability for the control logic to correctly sequence through the microinstructions
  - Any additional datapath elements and control signals needed (no need if the operation can be "translated" into existing control signals)

Let's Do Some More Microprogramming

- Implement REP MOVS in the LC-3b microarchitecture

- What changes, if any, do you make to the
  - state machine?
  - datapath?
  - control store?
  - microsequencer?

- Show all changes and microinstructions
- Optional HW Assignment

x86 REP MOVS (String Copy) Instruction

REP MOVS (DEST SRC)

```
IF AddressSize = 16
    THEN
        Use CX for CountReg;
    ELSE IF AddressSize = 64 and REX.W used
        THEN Use RCX for CountReg; FI;
    ELSE
        Use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
    DO
        Service pending interrupts (if any);
        Execute associated string instruction;
        CountReg ← (CountReg – 1);
        IF CountReg = 0
            THEN exit WHILE loop; FI;
        IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
        or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
            THEN exit WHILE loop; FI;
    OD;
```

```
DEST ← SRC;
IF (Byte move)
    THEN IF DF = 0
        THEN
            (R|E)SI ← (R|E)SI + 1;
            (R|E)DI ← (R|E)DI + 1;
        ELSE
            (R|E)SI ← (R|E)SI – 1;
            (R|E)DI ← (R|E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 2;
            (R|E)DI ← (R|E)DI + 2;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 2;
            (R|E)DI ← (R|E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 4;
            (R|E)DI ← (R|E)DI + 4;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 4;
            (R|E)DI ← (R|E)DI – 4;
        FI;
    ELSE IF (Quadword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 8;
            (R|E)DI ← (R|E)DI + 8;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 8;
            (R|E)DI ← (R|E)DI – 8;
        FI;
FI;
```

How many instructions does this take in MIPS ISA?

How many microinstructions does this take to add to the LC-3b microarchitecture?

Aside: Alignment Correction in Memory

- Unaligned accesses

- LC-3b has byte load and byte store instructions that move data not aligned at the word-address boundary
  - Convenience to the programmer/compiler

- How does the hardware ensure this works correctly?
  - Take a look at state 29 for LDB
  - States 24 and 17 for STB
  - Additional logic to handle unaligned accesses

- P&P, Revised Appendix C.5

Aside: Memory Mapped I/O

- Address control logic determines whether the specified address of LDW and STW are to memory or I/O devices
- Correspondingly enables memory or I/O devices and sets up muxes
- An instance where the final control signals of some datapath elements (e.g., MEM.EN or INMUX/2) **cannot** be stored in the control store
  - These signals are dependent on memory address
- P&P, Revised Appendix C.6

Advantages of Microprogrammed Control

- Allows a very simple design to do powerful computation by controlling the datapath (using a sequencer)
  - High-level ISA translated into microcode (sequence of u-instructions)
  - Microcode (u-code) enables a minimal datapath to emulate an ISA
  - Microinstructions can be thought of as a user-invisible ISA (u-ISA)

- Enables easy extensibility of the ISA
  - Can support a new instruction by changing the microcode
  - Can support complex instructions as a sequence of simple microinstructions (e.g., REP MOVS, INC [MEM])

- Enables update of machine behavior
  - A buggy implementation of an instruction can be fixed by changing the microcode in the field
    - Easier if datapath provides ability to do the same thing in different ways

Update of Machine Behavior

- The ability to update/patch microcode in the field (after a processor is shipped) enables
  - Ability to add new instructions without changing the processor!
  - Ability to "fix" buggy hardware implementations

- Examples
  - IBM 370 Model 145: microcode stored in main memory, can be updated after a reboot
  - IBM System z: Similar to 370/145.
    - Heller and Farrell, "Millicode in an IBM zSeries processor," IBM JR&D, May/Jul 2004.
  - B1700 microcode can be updated while the processor is running
    - User-microprogrammable machine!
    - Wilner, "Microprogramming environment on the Burroughs B1700", CompCon 1972.

Multi-Cycle vs. Single-Cycle uArch

- Advantages
- Disadvantages
- For you to fill in