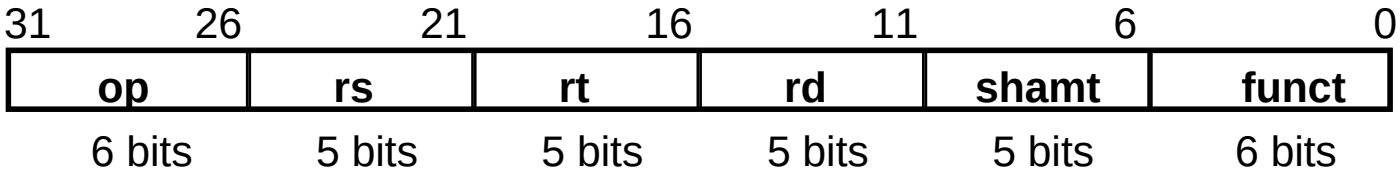
Computer Architecture
Ch. 5-2: Designing Single Cycle Control

Spring, 2005

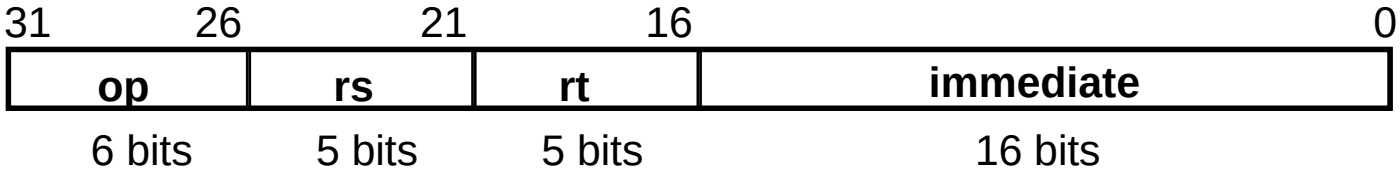
Sao-Jie Chen (csj@cc.ee.ntu.edu.tw)

Recap: The MIPS Instruction Formats

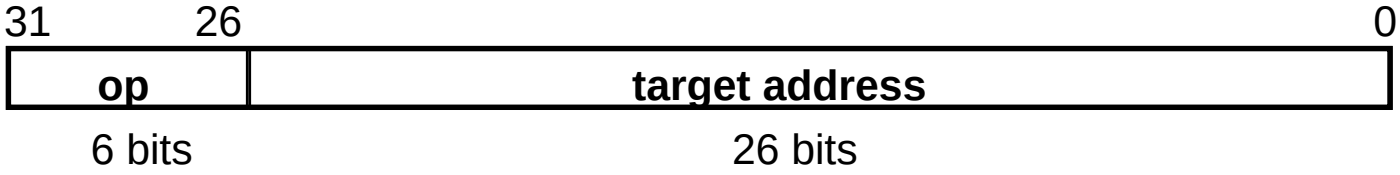
- All MIPS instructions are 32 bits long. The three instruction formats:
  - R-type



  - I-type



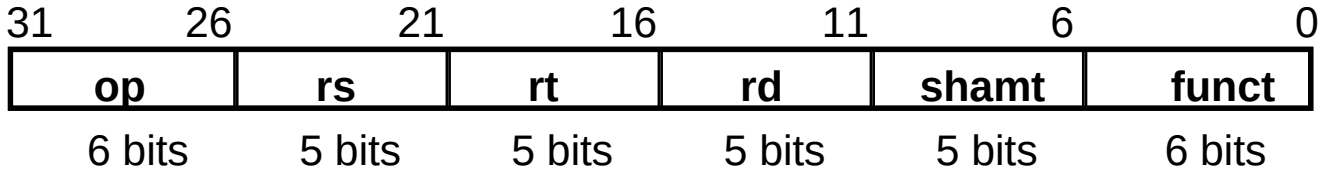
  - J-type

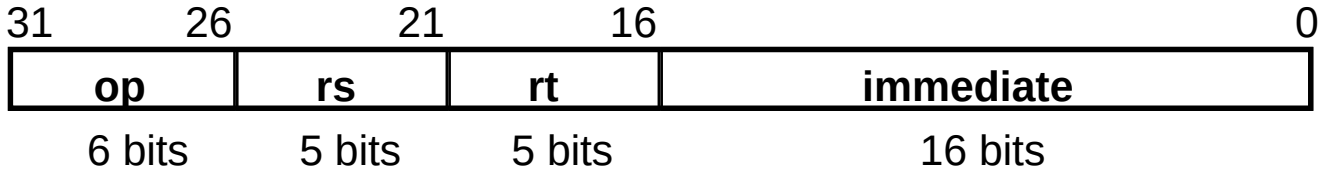


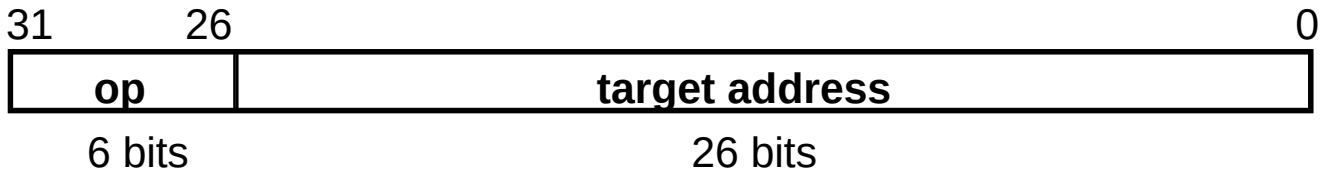
- The different fields are:
  - op: operation of the instruction
  - rs, rt, rd: the source and destination registers specifier
  - shamt: shift amount
  - funct: selects the variant of the operation in the “op” field
  - address / immediate: address offset or immediate value
  - target address: target address of the jump instruction

Recap: The MIPS Subset

- **ADD and subtract**
  - **add rd, rs, rt**
  - **sub rd, rs, rt**
- **OR Imm:**
  - **ori rt, rs, imm16**
- **LOAD and STORE**
  - **lw rt, rs, imm16**
  - **sw rt, rs, imm16**
- **BRANCH:**
  - **beq rs, rt, imm16**
- **JUMP:**
  - **j target**

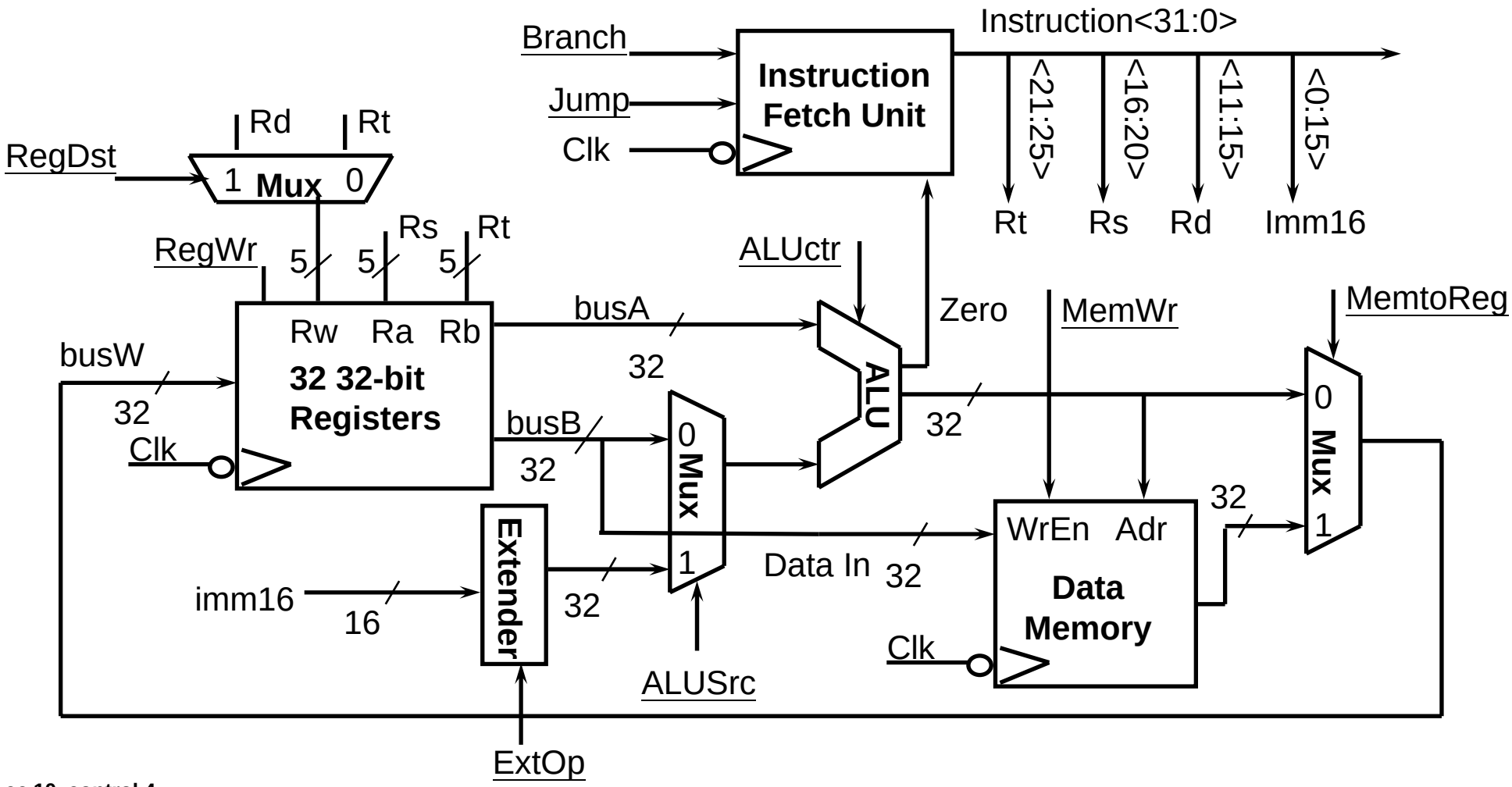






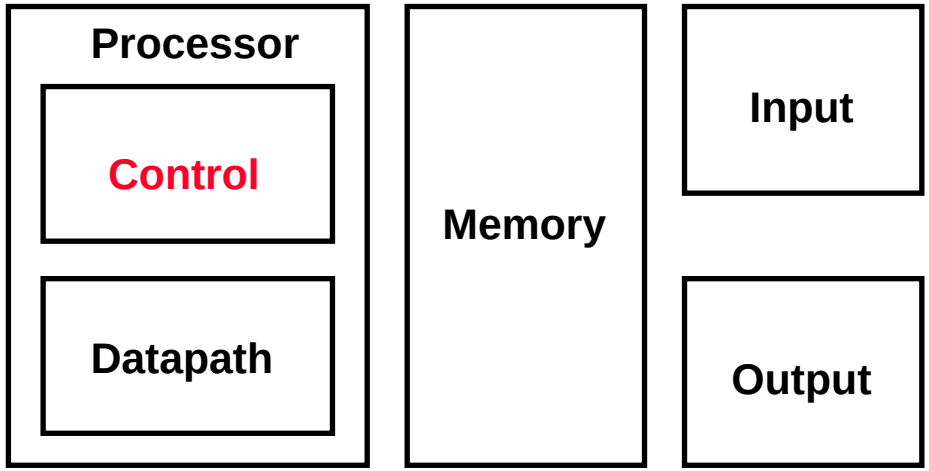
Recap: A Single Cycle Datapath

- We have everything except control signals (underline)
  - Today's lecture will show you how to generate the control signals



The Big Picture: Where are We Now?

- The Five Classic Components of a Computer

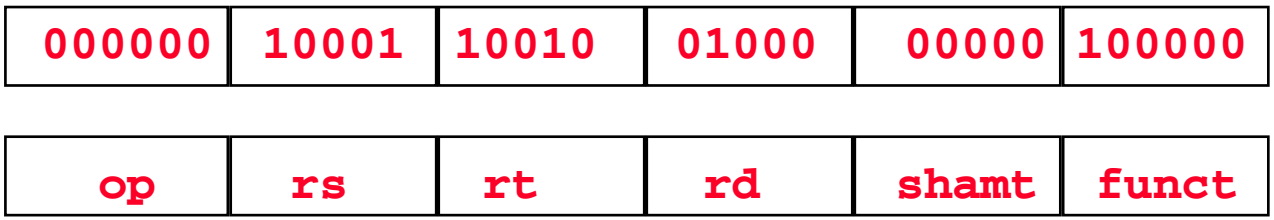


- Today's Topic: Designing the Control for the Single Cycle Datapath

Control

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
- Example:

add $8, $17, $18 Instruction Format:



- ALU's operation based on instruction type and function code

Control

- e.g., what should the ALU do with this instruction
- Example: lw $1, 100($2)

35	2	1	100

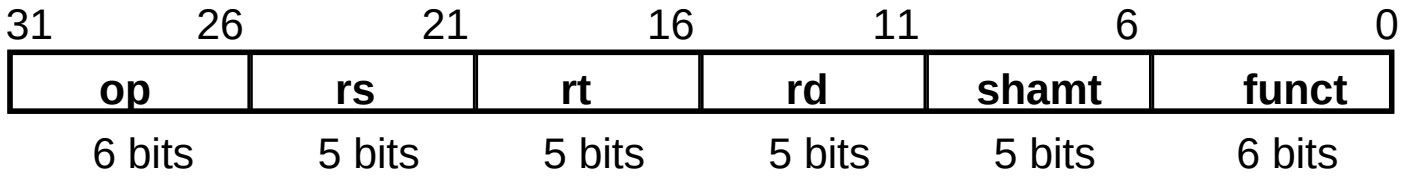
op	rs	rt	16 bit offset

- ALU control input

```
0000   AND
0001   OR
0010   add
0110   subtract
0111   set-on-less-than
1100   NOR
```

- Why is the code for subtract 0110 and not 0011?

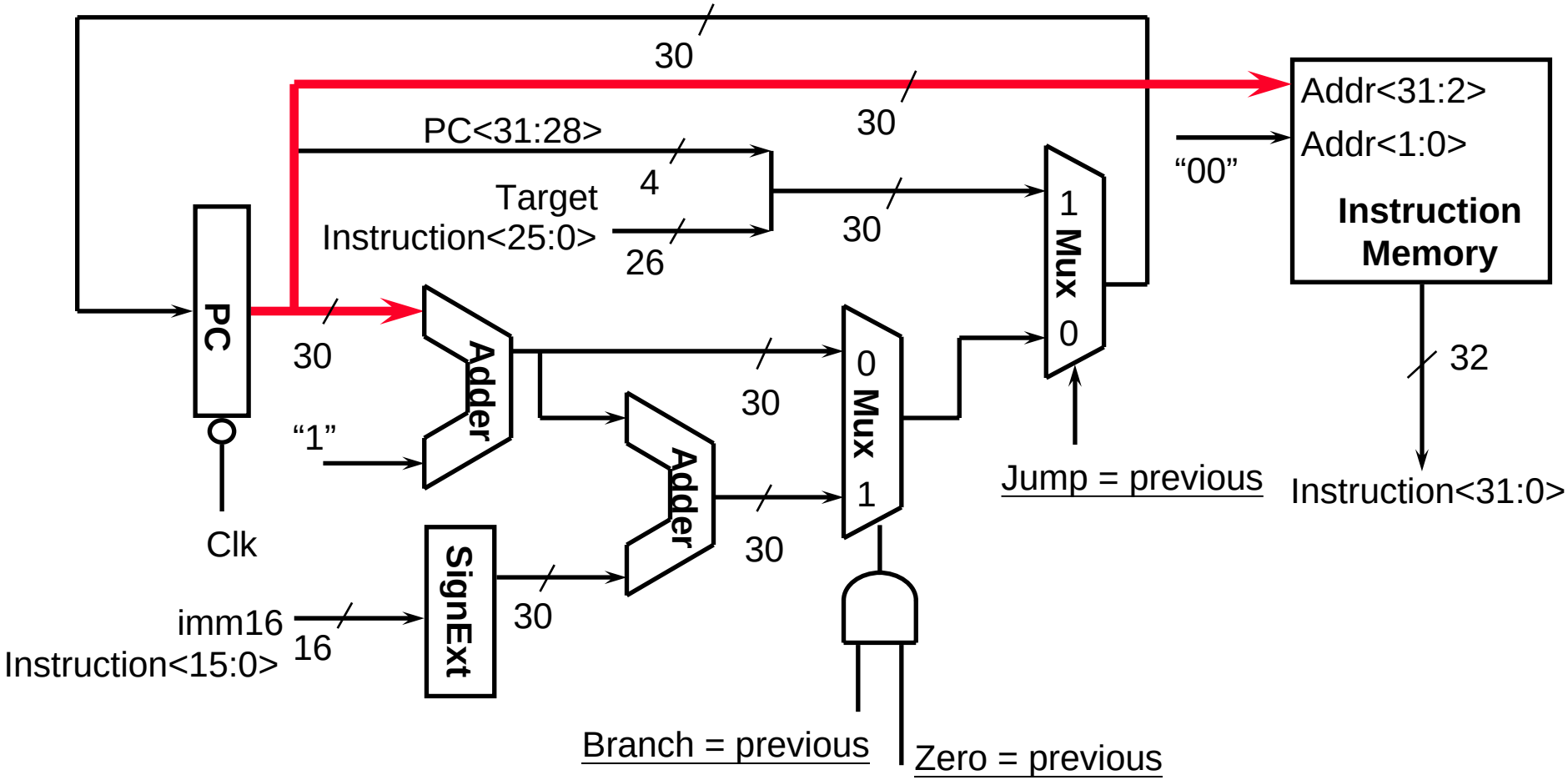
RTL: The ADD Instruction



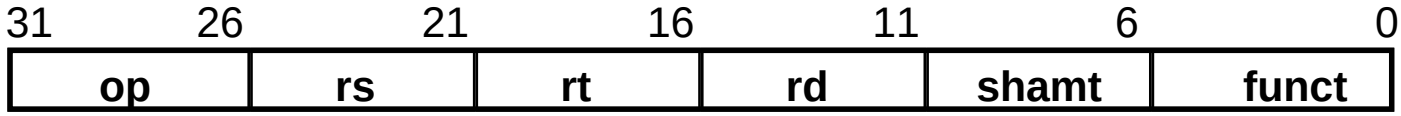
- **add rd, rs, rt**
  - **mem[PC]** — **Fetch the instruction from memory**
  - **R[rd] <- R[rs] + R[rt]** — **The actual operation**
  - **PC <- PC + 4** — **Calculate the next instruction's address**

Instruction Fetch Unit at the Beginning of Add / Subtract

- **Fetch the instruction from Instruction memory: Instruction <- mem[PC]**
- **This is the same for all instructions**



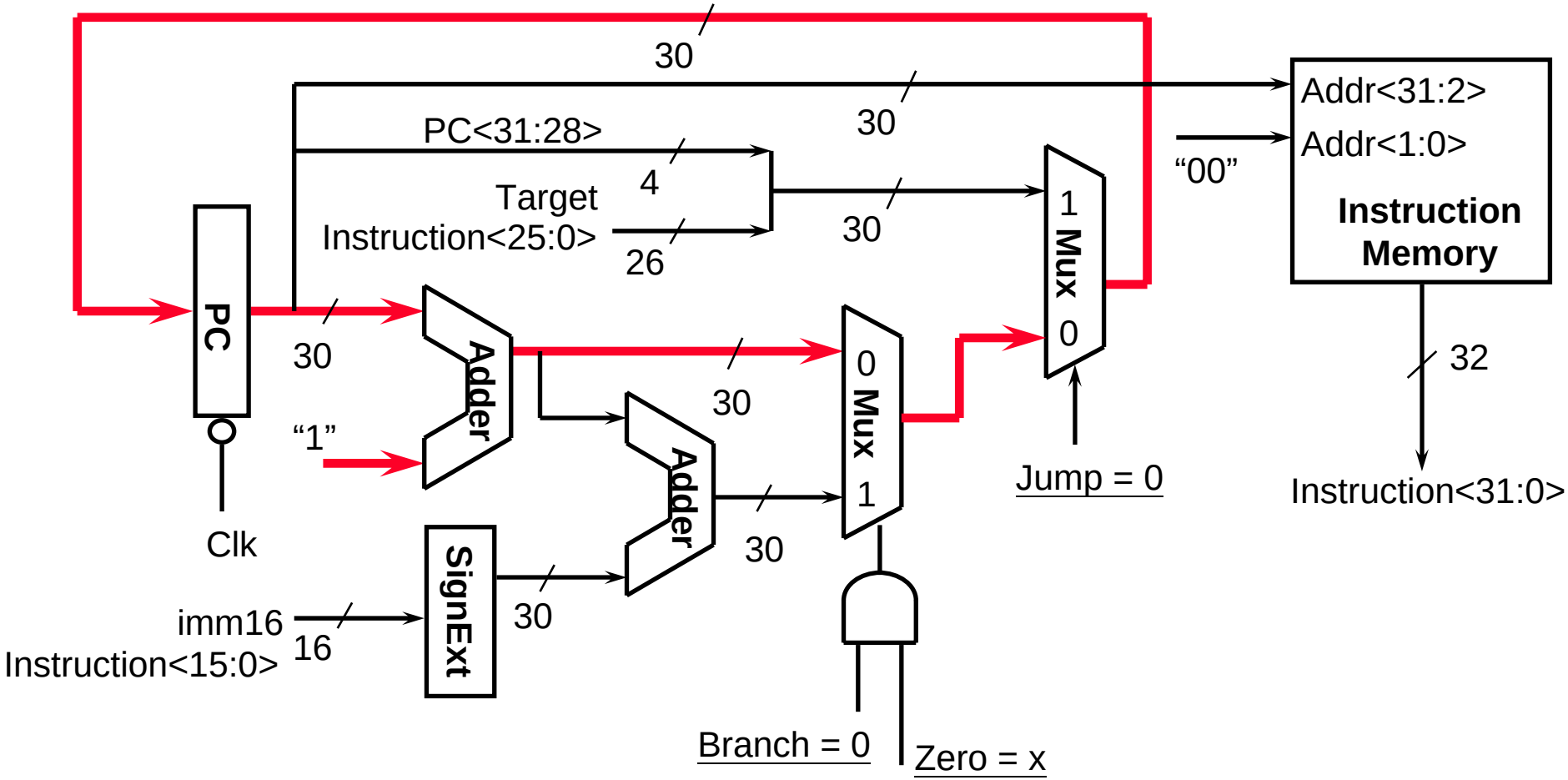
The Single Cycle Datapath during Add and Subtract



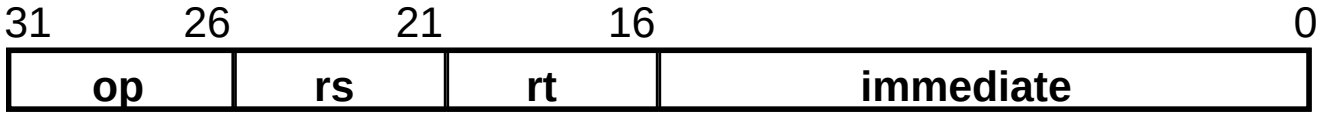
- R[rd] <- R[rs] + / - R[rt]

Instruction Fetch Unit at the End of Add and Subtract

- **PC <- PC + 4**
  - **This is the same for all instructions except: Branch and Jump**

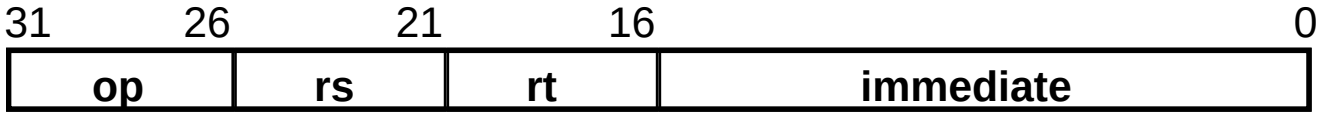


The Single Cycle Datapath during Or Immediate



- R[rt] <- R[rs] or ZeroExt(imm16)

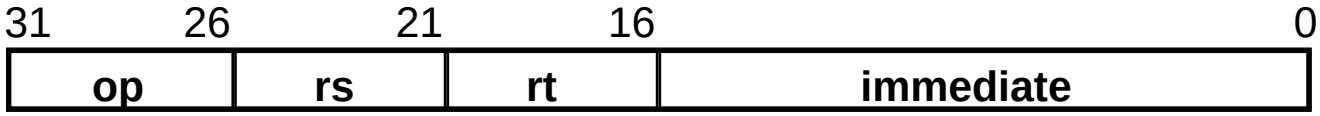
The Single Cycle Datapath during Load



- R[rt] <- Data Memory [R[rs] + SignExt(imm16)]

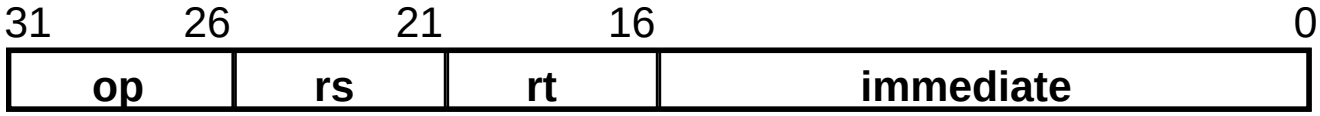
Branch = 0
Jump = 0
Clk
Instruction Fetch Unit
Instruction<31:0>
<21:25>
<16:20>
<11:15>
<0:15>
Rt
Rs
Rd
Imm16
RegDst = 0
Rd
Rt
1 Mux 0
RegWr = 1
5
5
5
Rs
Rt
busW
32
Clk
Rw Ra Rb
32 32-bit Registers
busA
32
busB
32
ALUctr = Add
ALU
Zero
MemWr = 0
32
MemtoReg = 1
0 Mux 1
0 Mux 1
imm16
16
Extender
32
ALUSrc = 1
Data In
32
WrEn Adr
Data Memory
Clk
32
ExtOp = 1

The Single Cycle Datapath during Store

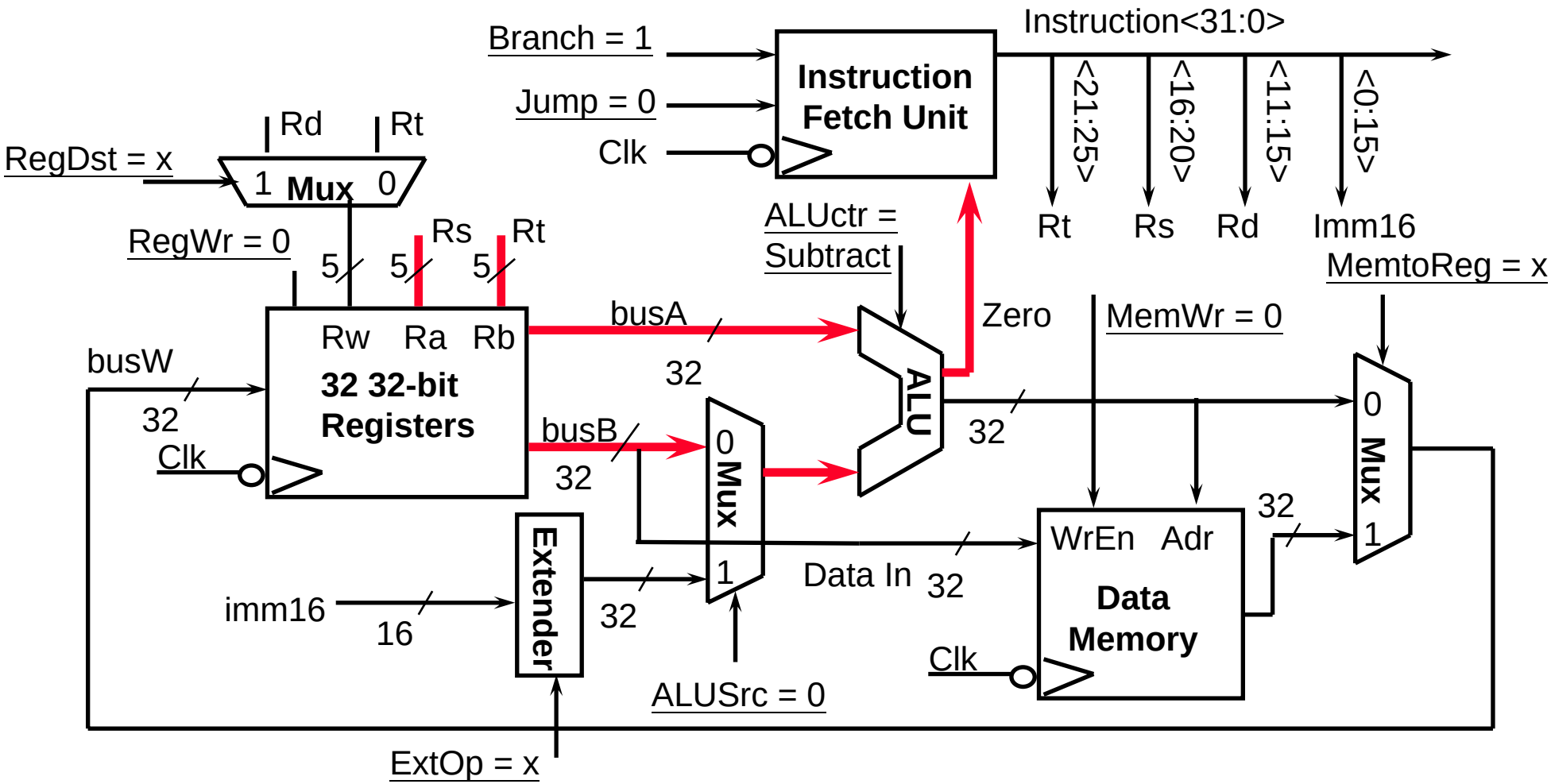


- Data Memory [R[rs] + SignExt(imm16)] <- R[rt]

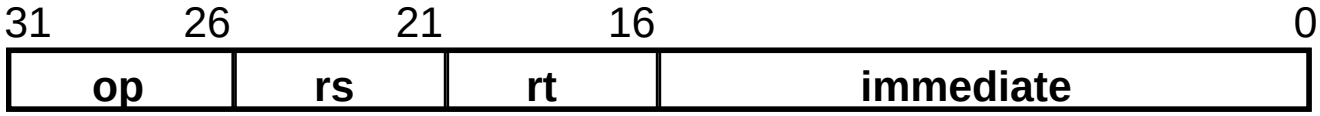
The Single Cycle Datapath during Branch



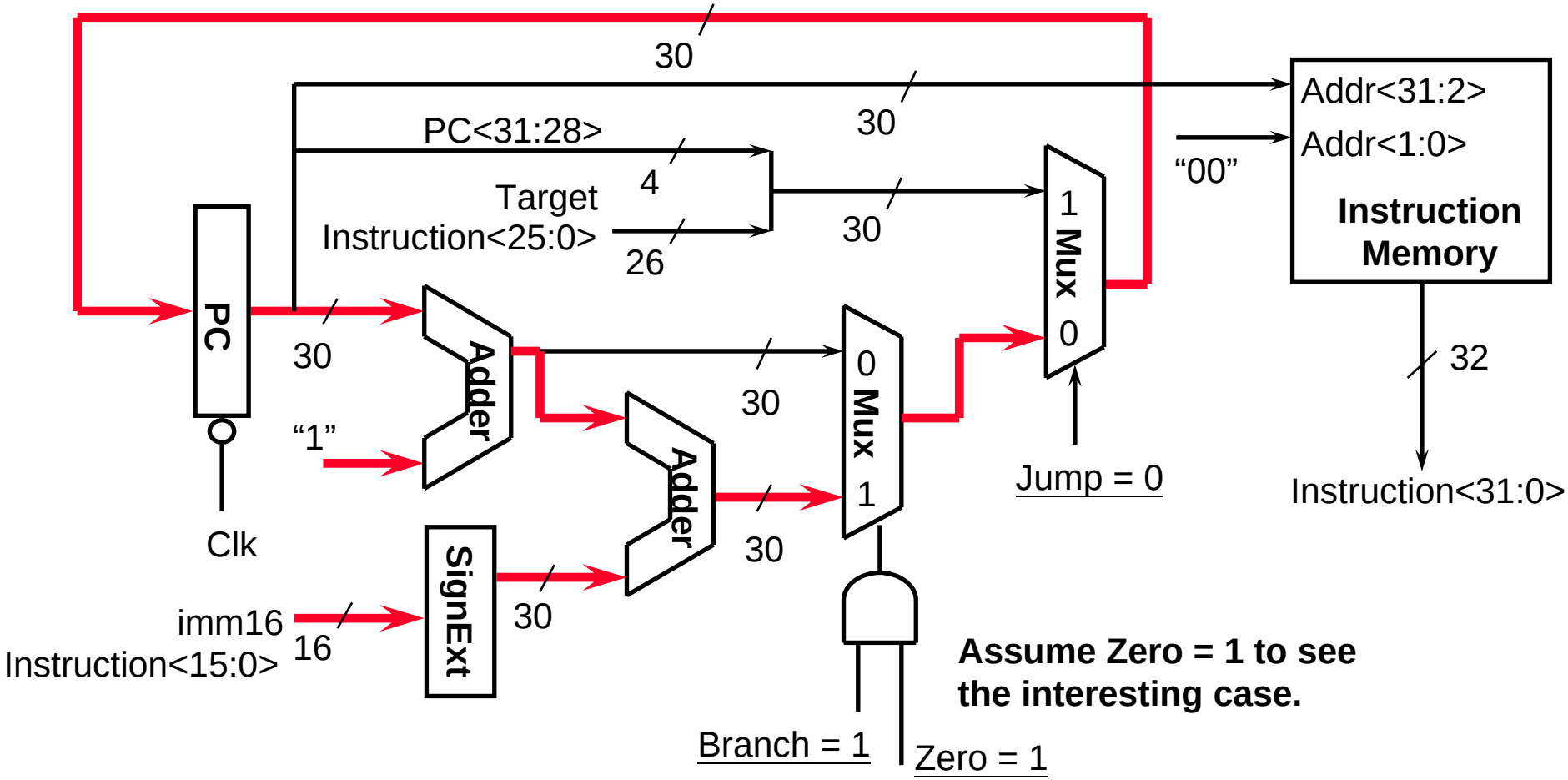
- if (R[rs] - R[rt] == 0) then Zero <- 1; else Zero <- 0



Instruction Fetch Unit at the End of Branch



- if (Zero == 1) then PC = PC + 4 + SignExt(imm16)*4; else PC = PC + 4

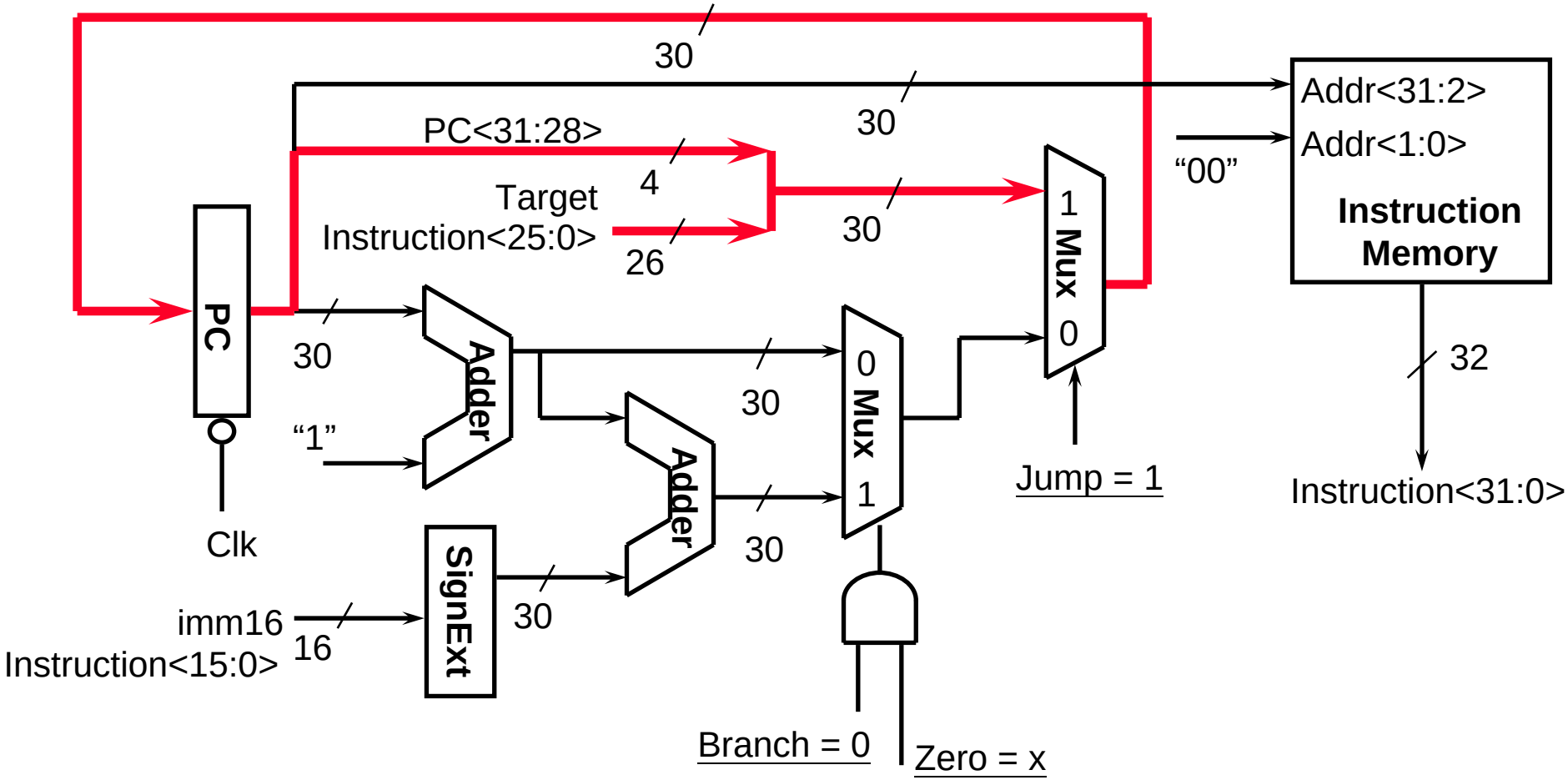


The Single Cycle Datapath during Jump

- **Nothing to do!** **Make sure control signals are set correctly!**

Instruction Fetch Unit at the End of Jump

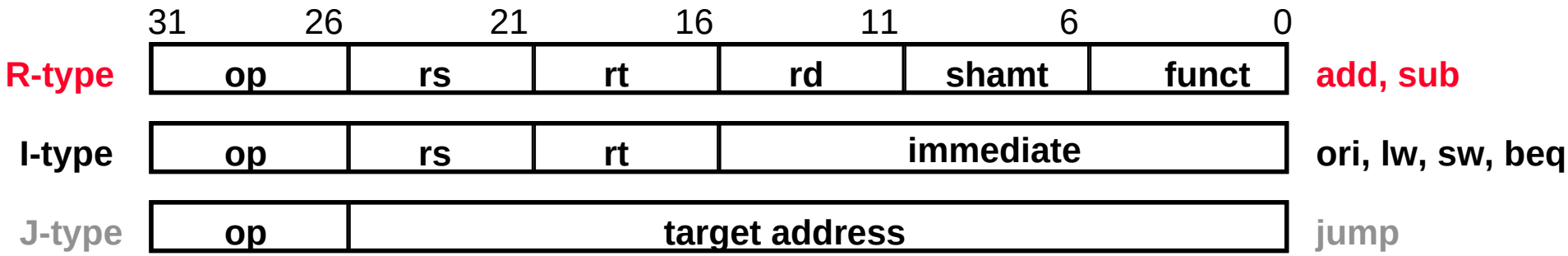
- PC <- PC<31:28> concat target<25:0>



A Summary of the Control Signals

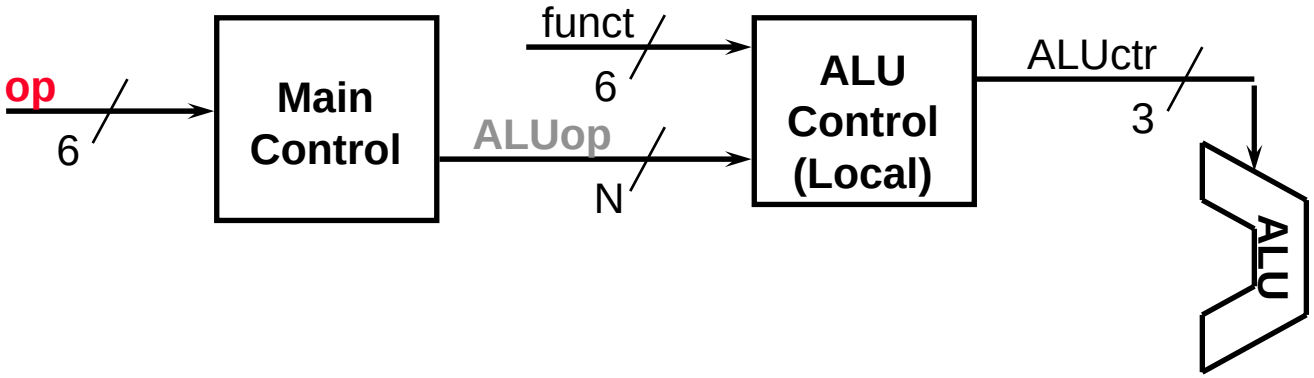
See Appendix B

	funct	10 0000	10 0010	We Don't Care :-)				
	op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
		add	sub	ori	lw	sw	beq	jump
RegDst		1	1	0	0	x	x	x
ALUSrc		0	0	1	1	1	0	x
MemtoReg		0	0	0	1	x	x	x
RegWrite		1	1	1	1	0	0	0
MemWrite		0	0	0	0	1	0	0
Branch		0	0	0	0	0	1	0
Jump		0	0	0	0	0	0	1
ExtOp		x	x	0	1	1	x	x
ALUctr<2:0>		Add	Subtract	Or	Add	Add	Subtract	xxx

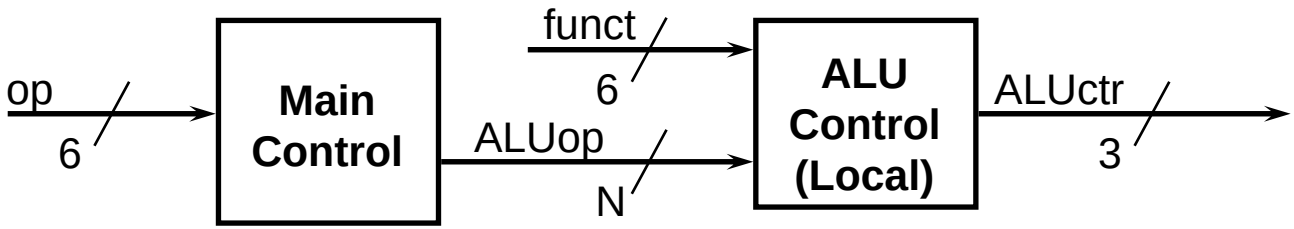


The Concept of Local Decoding

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx



The Encoding of ALUop

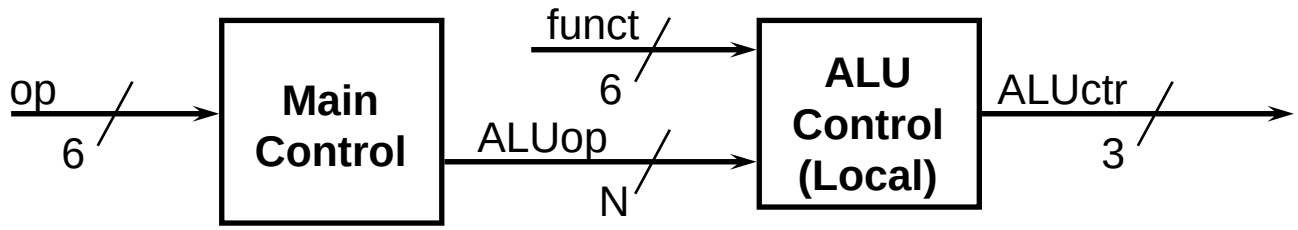


- **In this exercise, ALUop has to be 2 bits wide to represent:**
  - **(1) "R-type" instructions**
  - **"I-type" instructions that require the ALU to perform:**
    - ⇒ **(2) Or, (3) Add, and (4) Subtract**

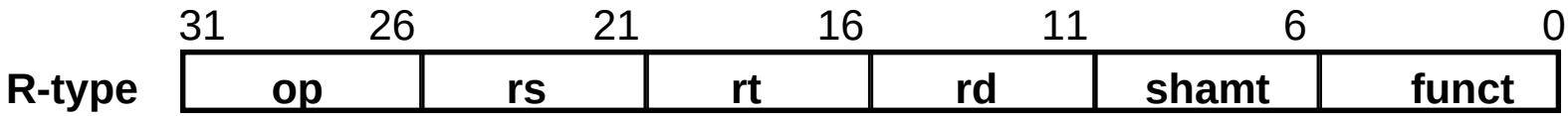
- **To implement the full MIPS ISA, ALUop has to be 3 bits to represent:**
  - **(1) "R-type" instructions**
  - **"I-type" instructions that require the ALU to perform:**
    - ⇒ **(2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)**

	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

The Decoding of the "Funct" Field

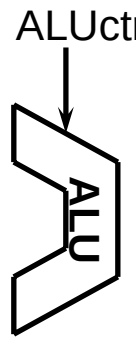


	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx



funct<5:0>	Instruction Operation
10 0000	add
10 0010	subtract
10 0100	and
10 0101	or
10 1010	set-on-less-than

Recall ALU Homework (also P. 353 text):



ALUctr<2:0>	ALU Operation
000	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

The Truth Table for ALUctr

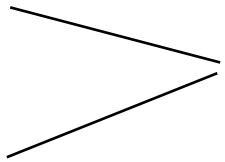
ALUop (Symbolic)	R-type	ori	lw	sw	beq
ALUop<2:0>	“R-type”	Or	Add	Add	Subtract
	1 00	0 10	0 00	0 00	0 01

funct<3:0>	Instruction Op.
0000	add
0010	subtract
0100	and
0101	or
1010	set-on-less-than

ALUop			funct				ALU Operation	ALUctr		
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	x	x	x	x	Add	0	1	0
0	x	1	x	x	x	x	Subtract	1	1	0
0	1	x	x	x	x	x	Or	0	0	1
1	x	x	0	0	0	0	Add	0	1	0
1	x	x	0	0	1	0	Subtract	1	1	0
1	x	x	0	1	0	0	And	0	0	0
1	x	x	0	1	0	1	Or	0	0	1
1	x	x	1	0	1	0	Set on <	1	1	1

Control

- **Must describe hardware to compute 3-bit ALU conrol input**
  - **given instruction type**
    00 = lw, sw
    01 = beq,
    11 = arithmetic
  - **function code for arithmetic**



ALUOp
computed from instruction type

- **Describe it using a truth table (can turn into gates):**

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

The Logic Equation for ALUctr<2>

ALUop			funct				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<2>
0	x	1	x	x	x	x	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

This makes funct<3> a don't care

- ALUctr<2> = !ALUop<2> & ALUop<0> +
  ALUop<2> & !funct<2> & funct<1> & !funct<0>

The Logic Equation for ALUctr<1>

ALUop			funct				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<1>
0	0	0	x	x	x	x	1
0	x	1	x	x	x	x	1
1	x	x	0	0	0	0	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

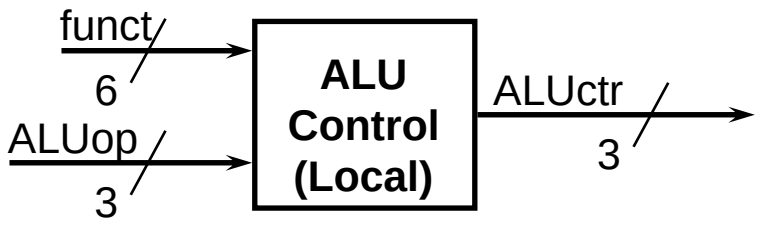
- **ALUctr<1> = !ALUop<2> & !ALUop<1> +**
  ALUop<2> & !funct<2> & !funct<0>

The Logic Equation for ALUctr<0>

ALUop			funct				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<0>
0	1	x	x	x	x	x	1
1	x	x	0	1	0	1	1
1	x	x	1	0	1	0	1

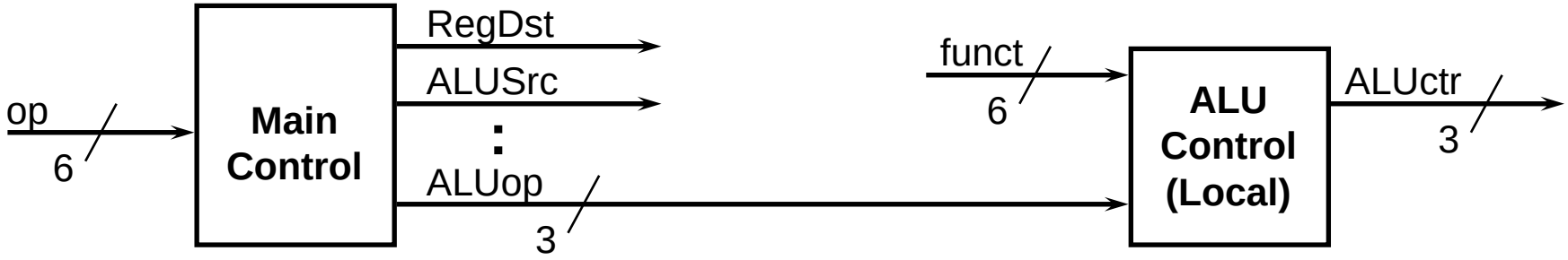
- **ALUctr<0> = !ALUop<2> & ALUop<1>**
  + ALUop<2> & !funct<3> & funct<2> & !funct<1> & funct<0>
  + ALUop<2> & funct<3> & !funct<2> & funct<1> & !funct<0>

The ALU Control Block



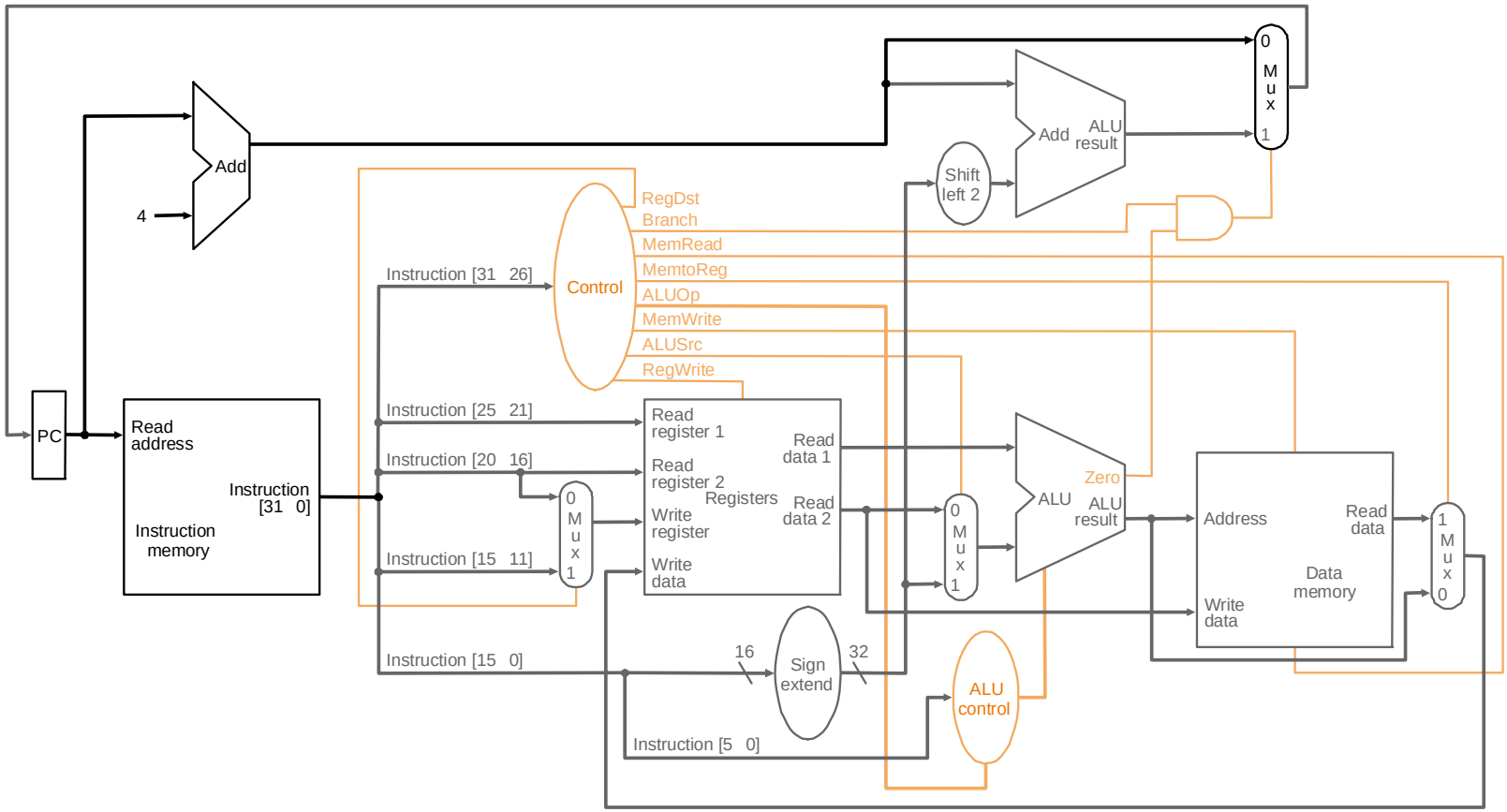
- **ALUctr<2> = !ALUop<2> & ALUop<0> +**
  ALUop<2> & !funct<2> & funct<1> & !funct<0>
- **ALUctr<1> = !ALUop<2> & !ALUop<1> +**
  ALUop<2> & !funct<2> & !funct<0>
- **ALUctr<0> = !ALUop<2> & ALUop<1>**
  + ALUop<2> & !funct<3> & funct<2> & !funct<1> & funct<0>
  + ALUop<2> & funct<3> & !funct<2> & funct<1> & !funct<0>

The "Truth Table" for the Main Control



op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x

Control



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

The "Truth Table" for RegWrite

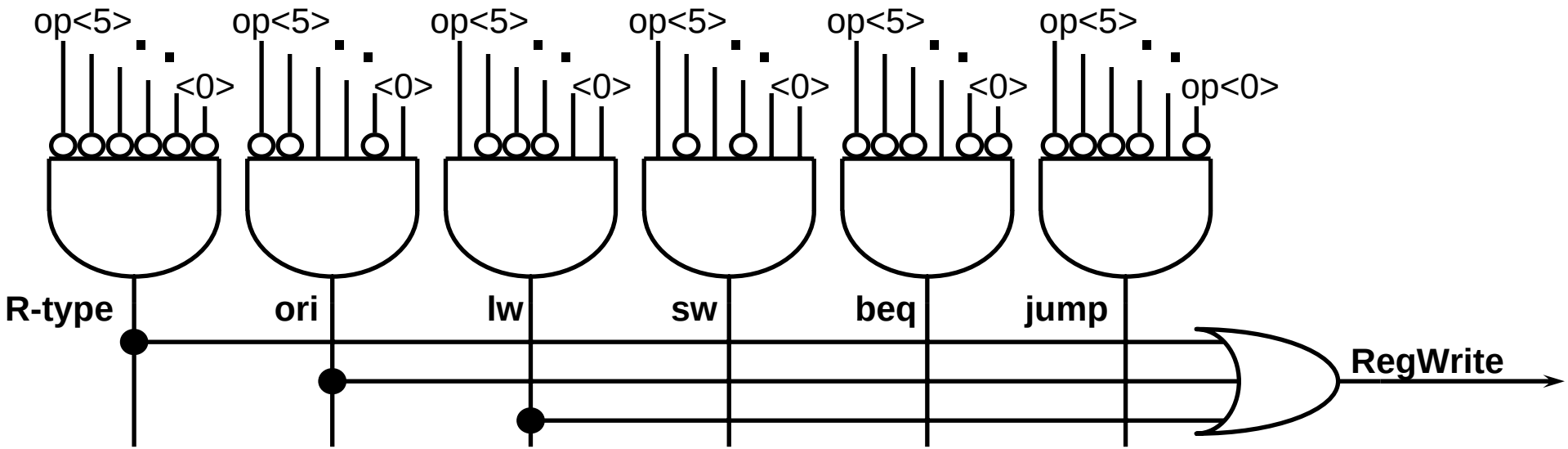
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegWrite	1	1	1	x	x	x

- **RegWrite = R-type + ori + lw**

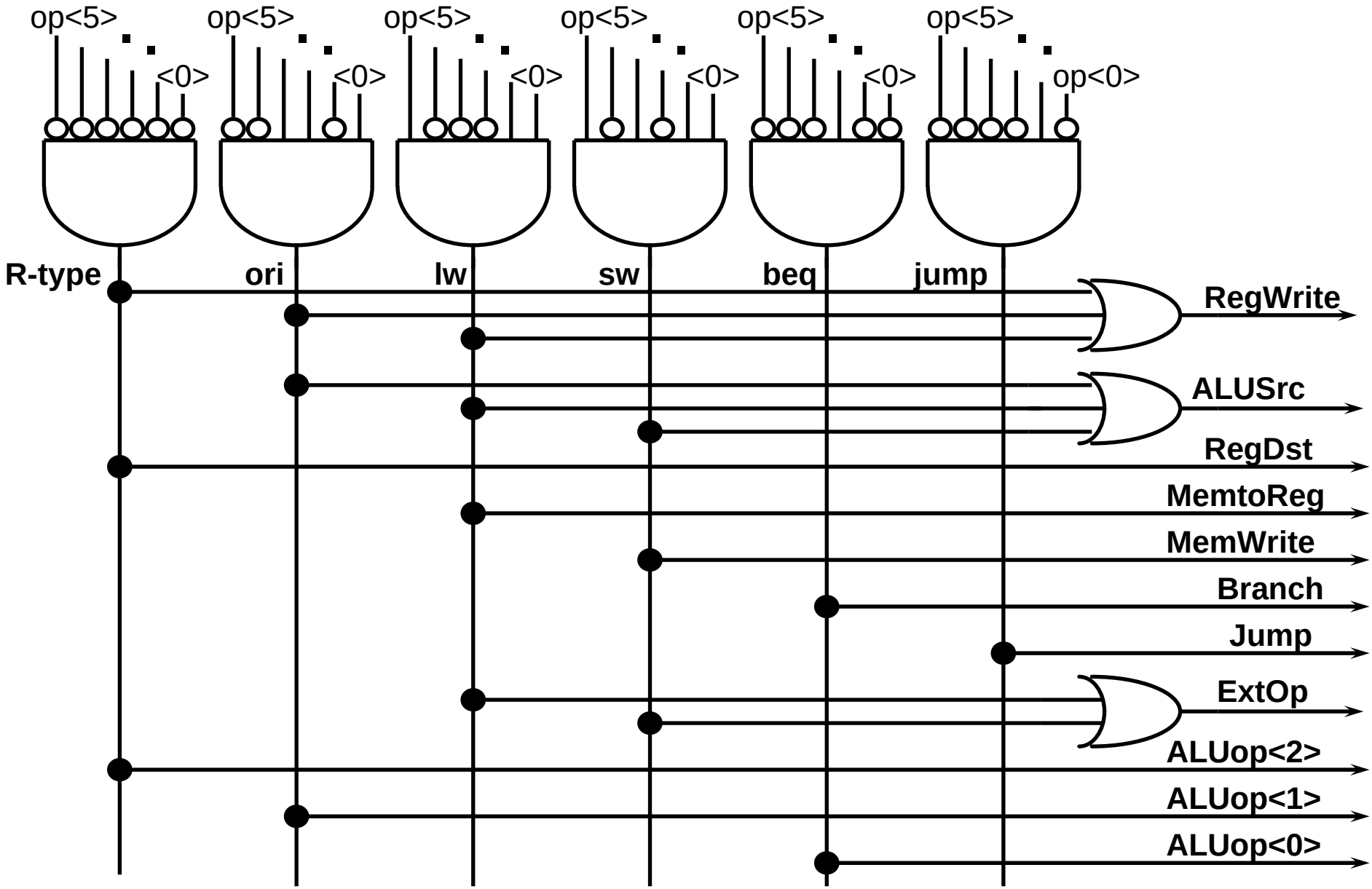
= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> **(R-type)**

+ !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> **(ori)**

+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> **(lw)**

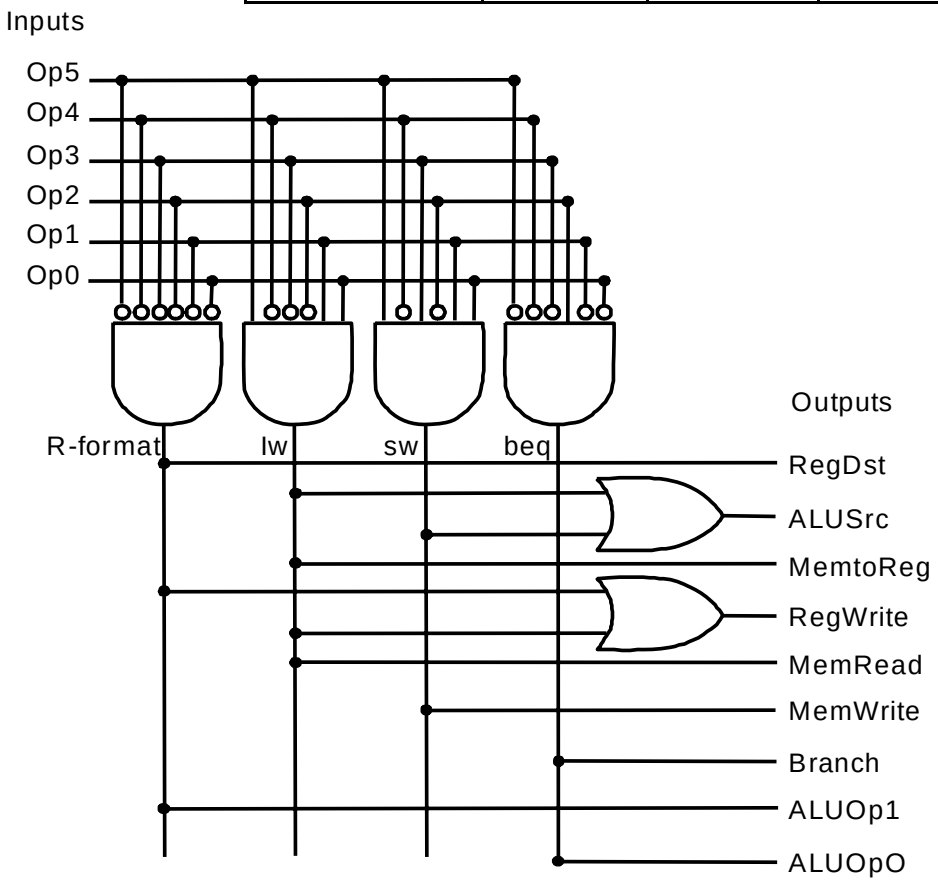


PLA Implementation of the Main Control



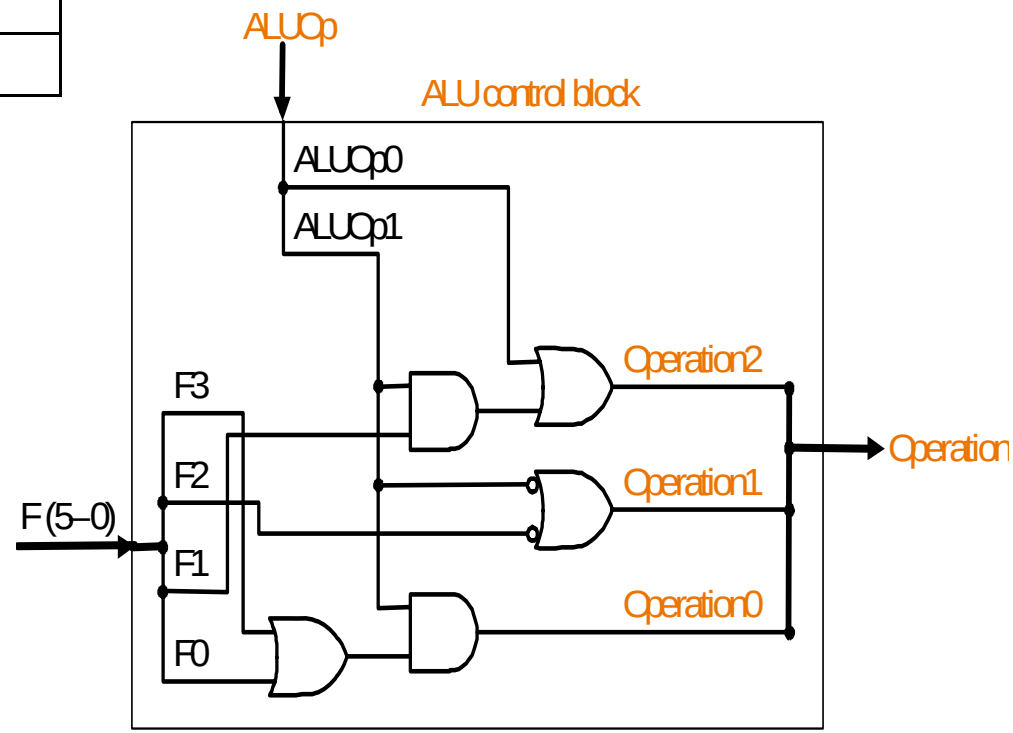
Building Control Path from True Table

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

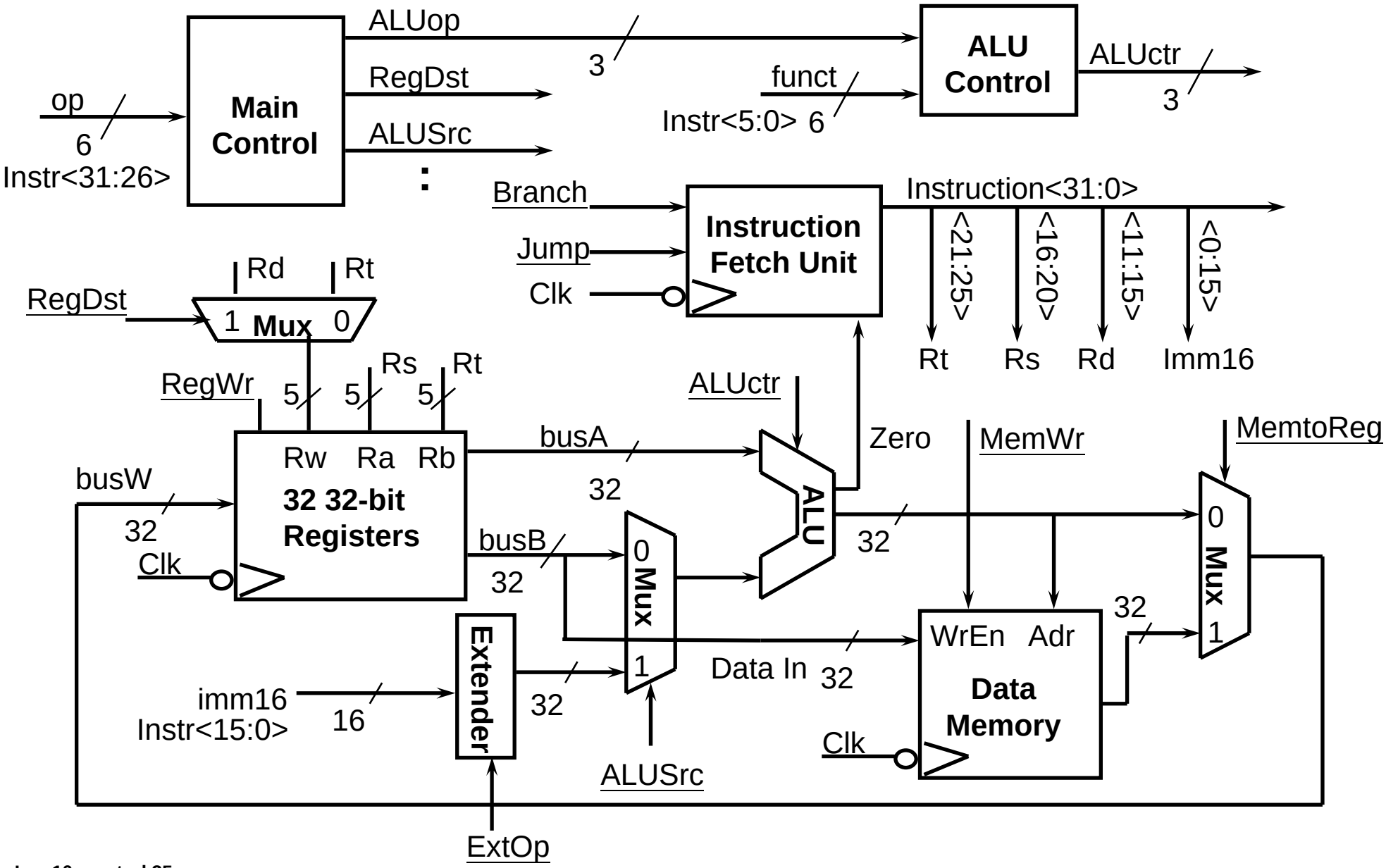


Building Control Path from True Table

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

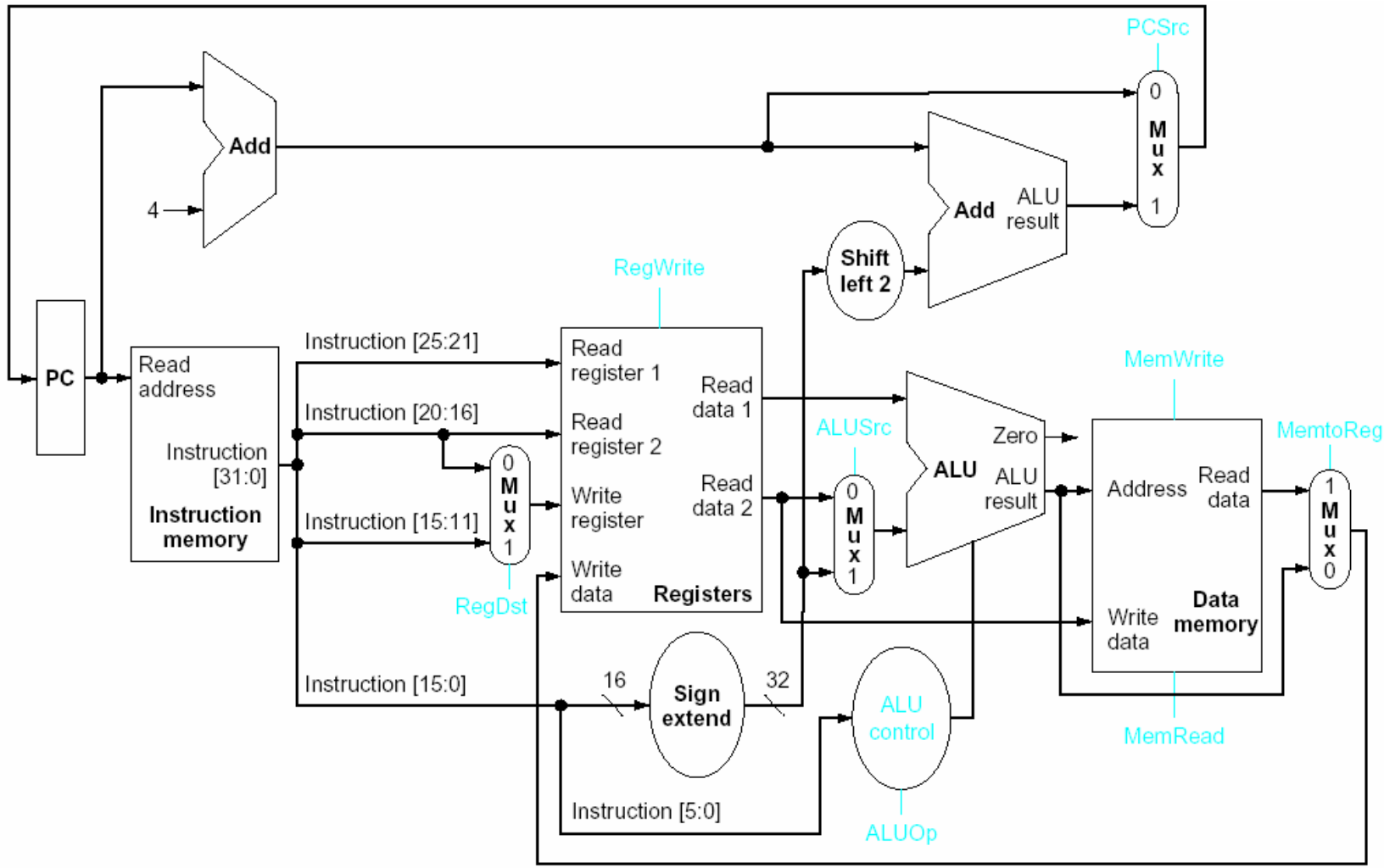


Putting it All Together: A Single Cycle Processor



Single Cycle Implementation

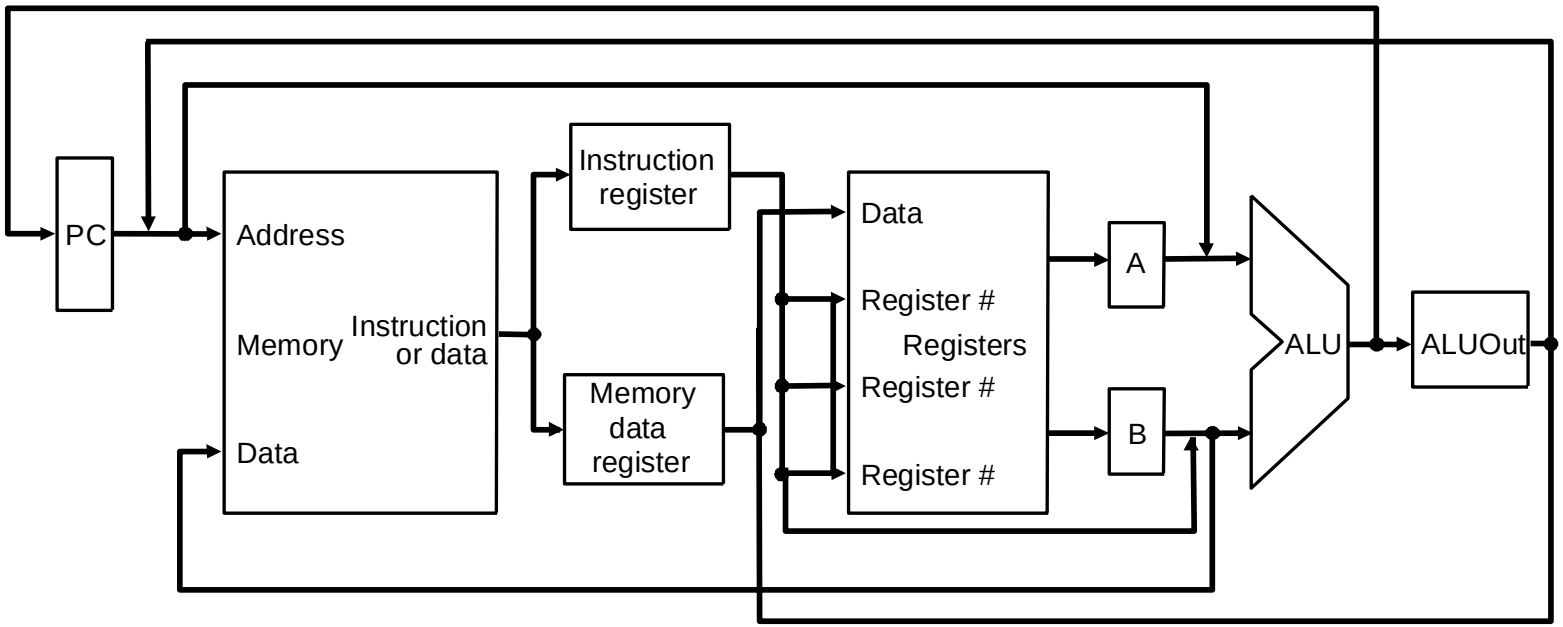
- Calculate cycle time assuming negligible delays except:
  - memory (200ps), ALU and adders (100ps), register file access (50ps)



Where we are headed

- **Single Cycle Problems:**
  - **what if we had a more complicated instruction like floating point?**
  - **wasteful of area**

- **One Solution:**
  - **use a “smaller” cycle time**
  - **have different instructions take different numbers of cycles**
  - **a “multicycle” datapath:**



How is this Different from a Real MIPS Processor?

- **The effect of load in a real MIPS Processor is delayed:**
  - ⇒ lw $1, 100($2) // Load Register R1
  - ⇒ add $3, $1, $0 // Move "old" R1 into R3
  - ⇒ add $4, $1, $0 // Move "new" R1 into R4
  - **The effect of load in our single cycle processor is NOT delayed**
    - ⇒ lw $1, 100($2) // Load Register R1
    - ⇒ add $3, $1, $0 // Move "new" R1 into R3

- **The effect of branch and jump in a real MIPS Processor is delayed:**
  - ⇒ **Instruction Address: 0x00** j 1000
  - ⇒ **Instruction Address: 0x04** add $1, $2, $3
  - ⇒ **Instruction Address: 0x1000** sub $1, $2, $3
  - **Branch and jump in our single cycle processor is NOT delayed**
    - ⇒ **Instruction Address: 0x00** j 1000
    - ⇒ **Instruction Address: 0x1000** sub $1, $2, $3

Drawback of this Single Cycle Processor

- Long cycle time:
  - Cycle time must be long enough for the load instruction:

    PC's Clock -to-Q +

    Instruction Memory Access Time +

    Register File Access Time +

    ALU Delay (address calculation) +

    Data Memory Access Time +

    Register File Setup Time +

    Clock Skew

- Cycle time is much longer than needed for all other instructions

Where to get more information?

- **Chapter 5 of your text book:**
  - **David Patterson and John Hennessy, *Computer Organization & Design: The Hardware / Software Interface*, 3rd Ed., ©2005, Morgan Kaufman Publishers.**
- **One of the best PhD thesis on processor design:**
  - **Manolis Katevenis, *Reduced Instruction Set Computer Architecture for VLSI*, Ph.D. Dissertation, EECS, U C Berkeley, 1982.**
- **For a reference on the MIPS architecture:**
  - **Gerry Kane, *MIPS RISC Architecture*, ©1992, Prentice Hall.**