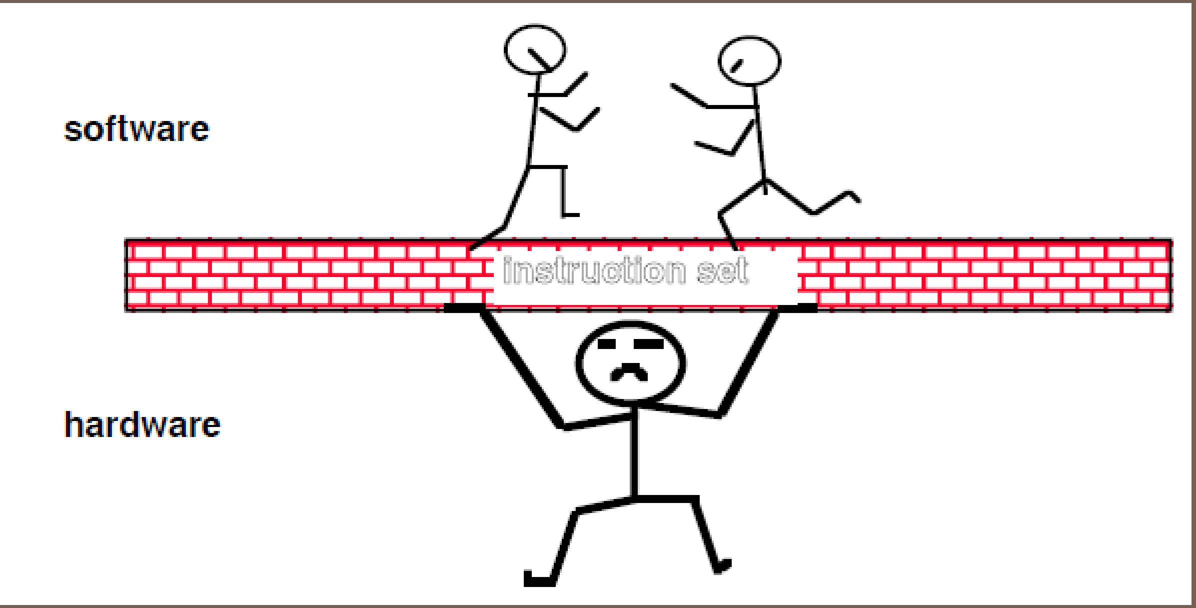


COMPUTER SYSTEMS ORGANIZATION

Acknowledgment: Almost all of these slides are based on Dave Patterson's CS152 Lecture Slides at UC, Berkeyley

MIPS CPU Instructions

Three Types of CPU Instructions

- R-type
- I-Type
- J-Type

R-Type Instructions

- R-type Instruction Format

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	sa	function
6	5	5	5	5	6

R-Type Instructions: ADD

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000
6	5	5	5	5	6

Format: `ADD rd, rs, rt` **MIPS32**

- Format: ADD rd, rs, rt
- R[rd] = R[rs] + R[rt]
- 32-bit 2's Complement Addition
- Destination register will not be modified if integer overflow exceptions occurs.

R-Type Instructions: SUB

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUB 100010
6	5	5	5	5	6

- Format: SUB rd, rs, rt
- R[rd] = R[rs] - R[rt]
- 32-bit signed subtraction
- Destination register will not be modified if integer overflow exceptions occurs.

R-Type Instructions: AND

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	AND 100100
6	5	5	5	5	6

Format: `AND rd, rs, rt` **MIPS32**

- Format: AND rd, rs, rt
- R[rd] = R[rs] & R[rt]

R-Type Instructions: OR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	OR 100101
6	5	5	5	5	6

Format: `OR rd, rs, rt` **MIPS32**

- Format: OR rd, rs, rt
- R[rd] = R[rs] | R[rt]

R-Type Instructions: SLT

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	SLT 101010
6	5	5	5	5	6

Format: `SLT rd, rs, rt` **MIPS32**

- Format: SLT rd, rs, rt
- R[rd] = R[rs] < R[rt] ? 1:0
- Signed comparision

There are many other R-type instructions like ADDU, NOR, XOR etc.

R-Type Instructions: SLL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SLL 000000
6	5	5	5	5	6

Format: `SLL rd, rt, sa` **MIPS32**

- Format: SLL rd, rt, sa
- R[rd] = R[rt] << sa

Note: In our processor design we do not implement shift instructions.

I-type Instructions

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

I-type Instructions

31 26	25 21	20 16	15 0
ADDI 001000	rs	rt	immediate
6	5	5	16

Format: `ADDI rt, rs, immediate` **MIPS32**

- Format: ADDI rt, rs, immediate
- R[rt] = R[rs] + sign_extend(immediate)
- immediate is 16-bit signed immediate
- 32-bit 2'complement addition
- Destination register will not be updated if integer overflow exception occurs

I-type Instructions

31 … 26	25 … 21	20 … 16	15 … 0
ANDI 001100	rs	rt	immediate
6	5	5	16

Format: `ANDI rt, rs, immediate` **MIPS32**

- Format: ANDI rt, rs, immediate
- R[rt] = R[rs] & zero_extend(immediate)

I-type Instructions: ORI

31 — 26	25 — 21	20 — 16	15 — 0
ORI 001101	rs	rt	immediate
6	5	5	16

Format: `ORI rt, rs, immediate` **MIPS32**

- Format: ORI rt, rs, immediate
- R[rt] = R[rs] | zero_extend(immediate)

I-type Instructions: SLTI

31 26	25 21	20 16	15 0
SLTI 001010	rs	rt	immediate
6	5	5	16

Format: `SLTI rt, rs, immediate` **MIPS32**

- Format: SLTI rt, rs, immediate
- R[rt] = R[rs] < sign_extend(immediate) ? 1:0

I-type Instructions: LW

31 26	25 21	20 16	15 0
LW 100011	base	rt	offset
6	5	5	16

Format: `LW rt, offset(base)` **MIPS32**

- Format: LW rt, offset(base)
- vdddr = sign_extend(offset) + R[base]
- R[rt] = Mem[vaddr]
- If vaddr is now word-aligned, an exception will be raised.

I-type Instructions: SW

31 26	25 21	20 16	15 0
SW 101011	base	rt	offset
6	5	5	16

Format: `SW rt, offset(base)` **MIPS32**

- Format: SW rt, offset(base)
- vdddr = sign_extend(offset) + R[base]
- Mem[vaddr] = R[rt]
- If vaddr is now word-aligned, an exception will be raised.

I-type Instructions: SLTI

31 26	25 21	20 16	15 0
SLTI 001010	rs	rt	immediate
6	5	5	16

Format: `SLTI rt, rs, immediate` **MIPS32**

- Format: SLTI rt, rs, immediate
- R[rt] = R[rs] < sign_extend(immediate) ? 1:0

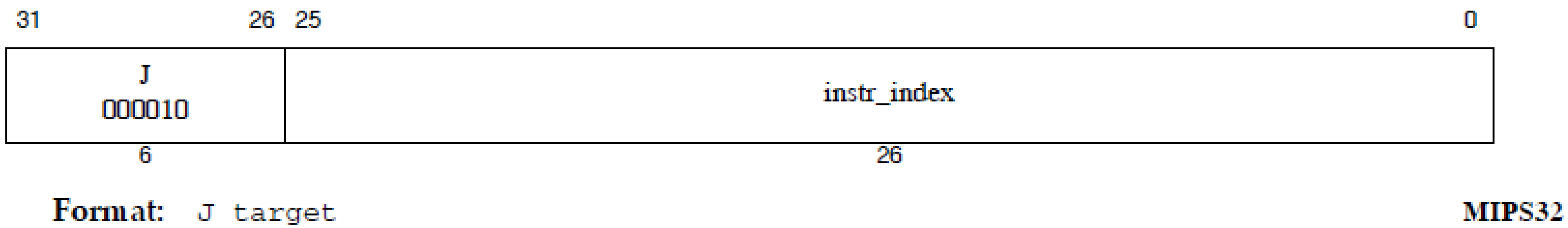
I-Type Instructions: BEQ

31 … 26	25 … 21	20 … 16	15 … 0
BEQ 000100	rs	rt	offset
6	5	5	16

Format: `BEQ rs, rt, offset` **MIPS32**

- Format: BEQ rs, rt, offset
- If R[rs] == R[rt] then
  - PC = addr_of_branch + 4 + sign_extend(offset << 2)

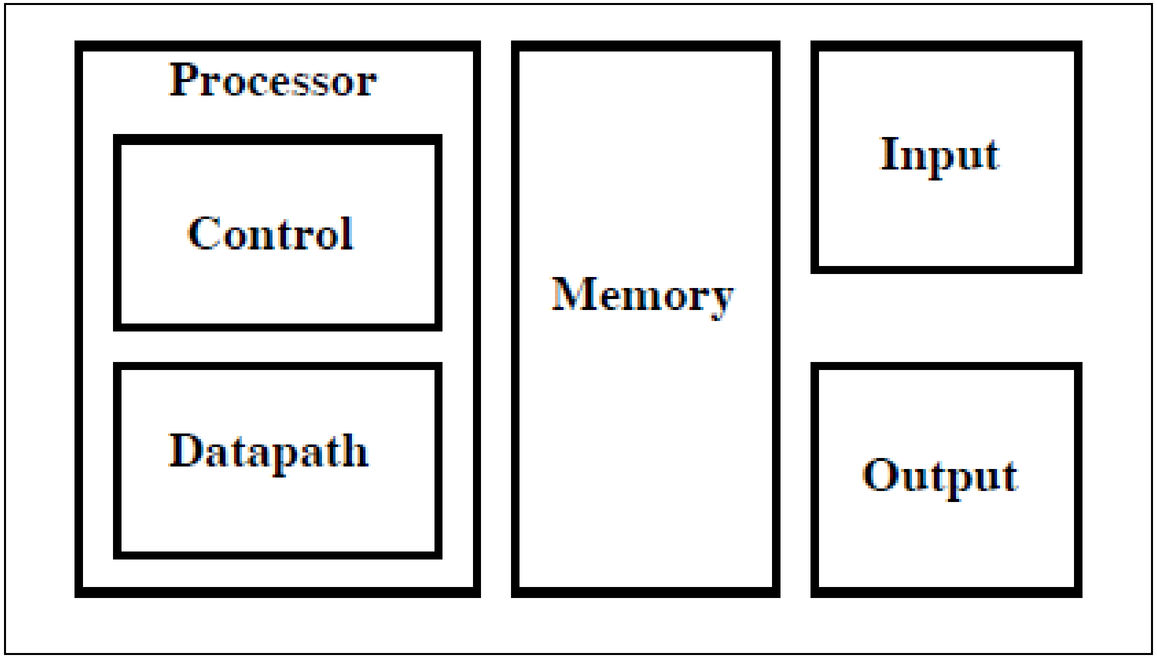
MIPS J-Type Instructions



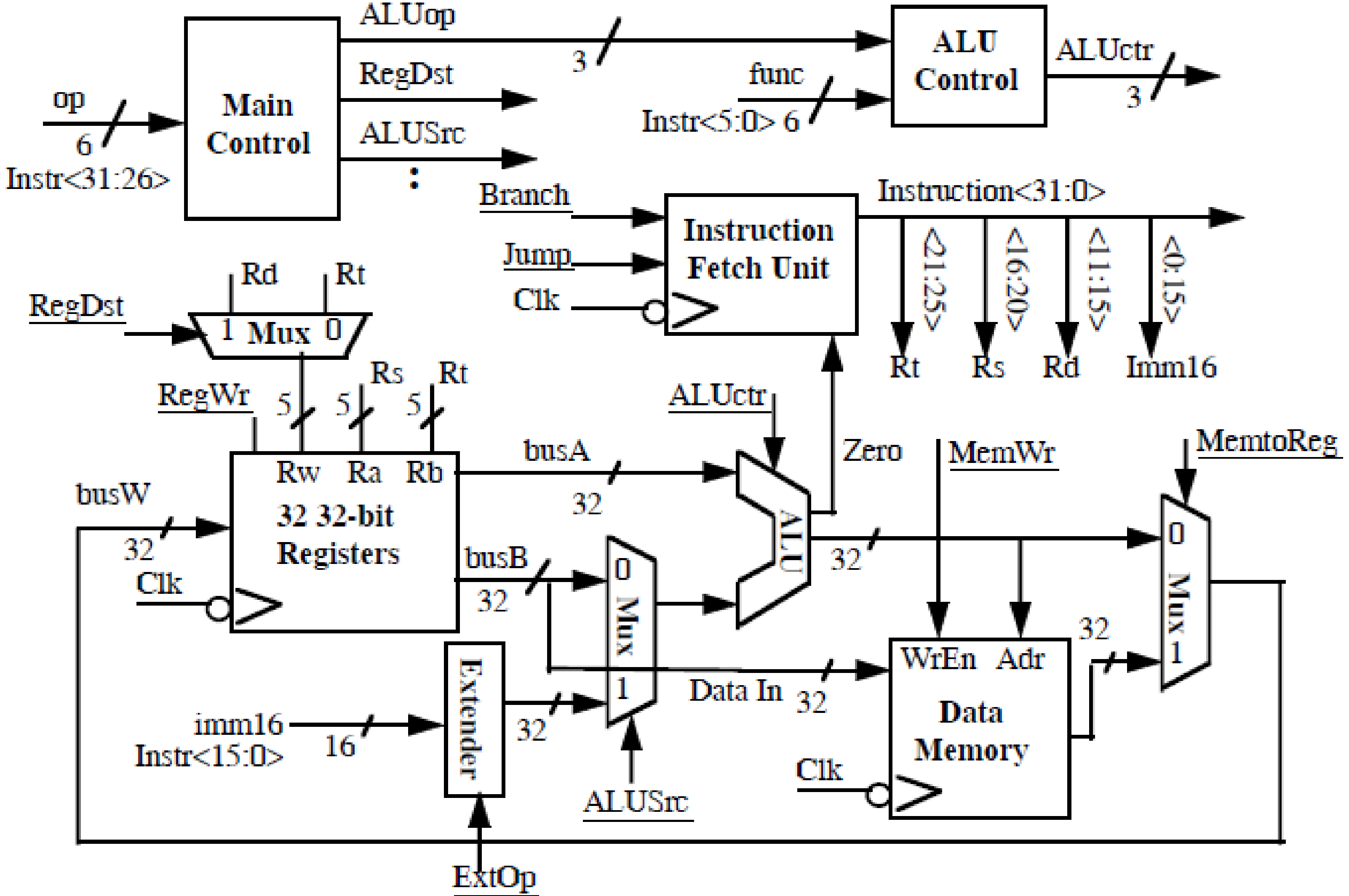
- Format: J target
- Target Address
  - Lower 28 bits: instr_index||00
  - Upper Four Bits: Bits 31, 30, 29, 28 of the address of the Jump Instruction.

The Big Picture: Where are We Now?

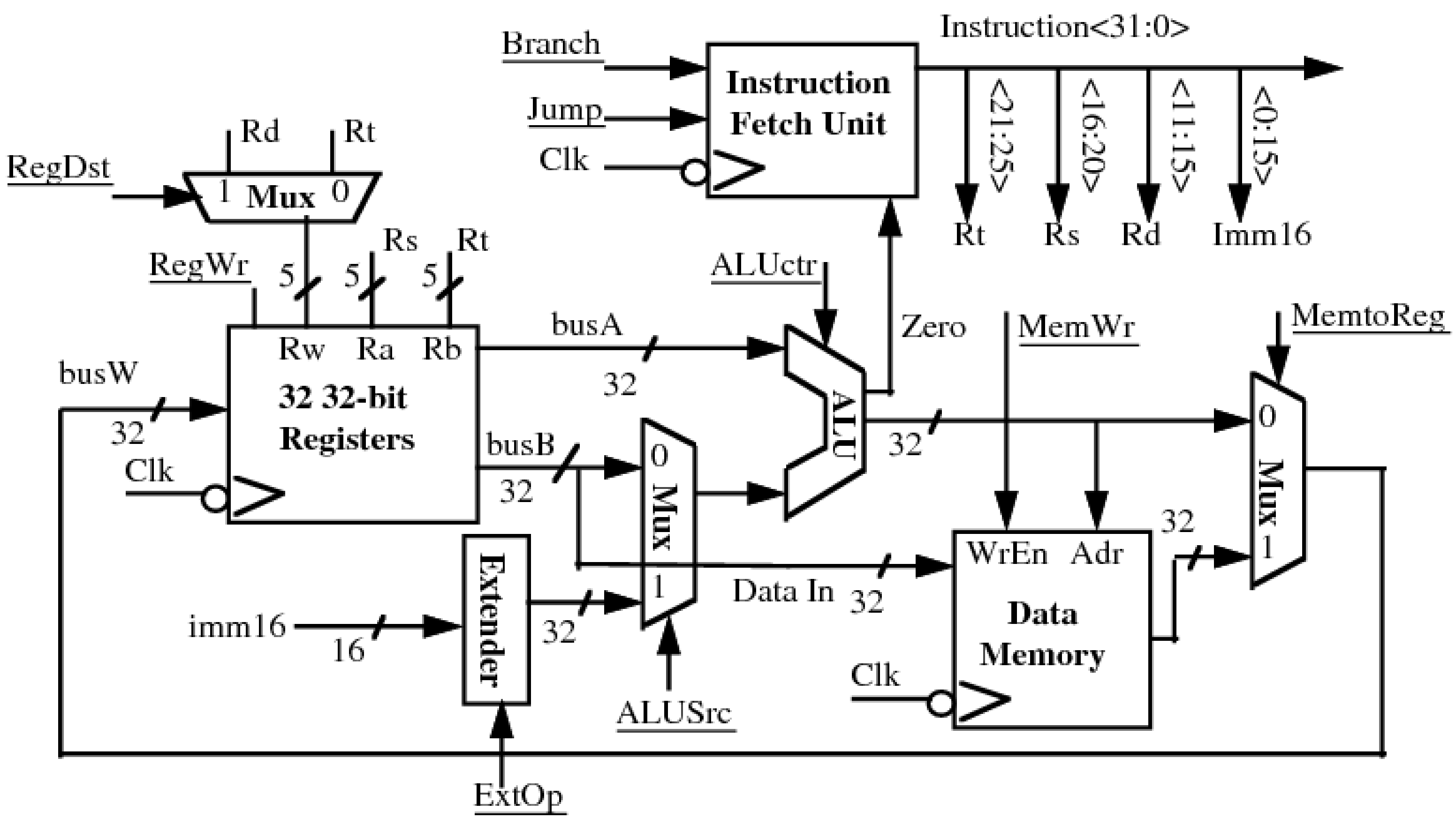
- Five Classic Components of a Computer



MIPS Processor: Control Path + Data Path



Data Path

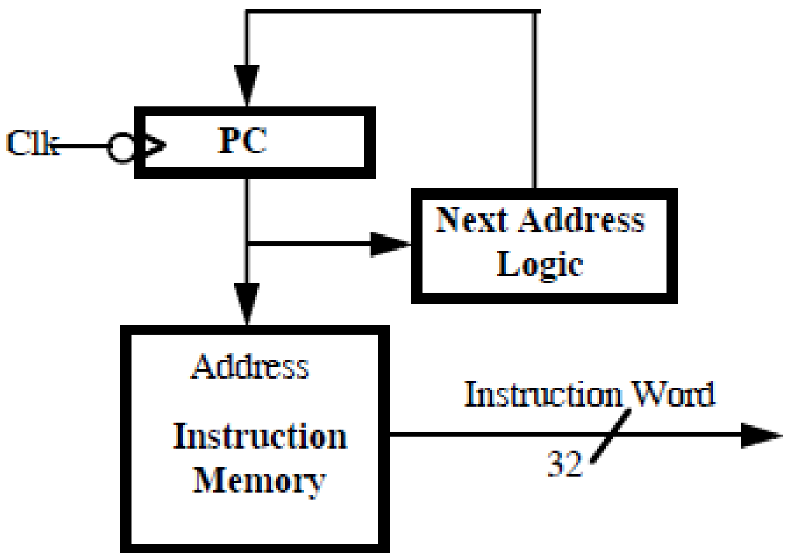


Instruction<31:0>
Branch
Jump
Clk
Instruction Fetch Unit
<21:25>
<16:20>
<11:15>
<0:15>
Rt
Rs
Rd
Imm16
RegDst
Rd
Rt
1 Mux 0
RegWr
5
5
5
Rs
Rt
Rw Ra Rb
32 32-bit Registers
busW
32
Clk
busA
32
busB
32
ALUctr
Zero
ALU
32
MemWr
MemtoReg
0
Mux
1
32
0
Mux
1
WrEn Adr
Data In 32
Data Memory
Clk
imm16
16
Extender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

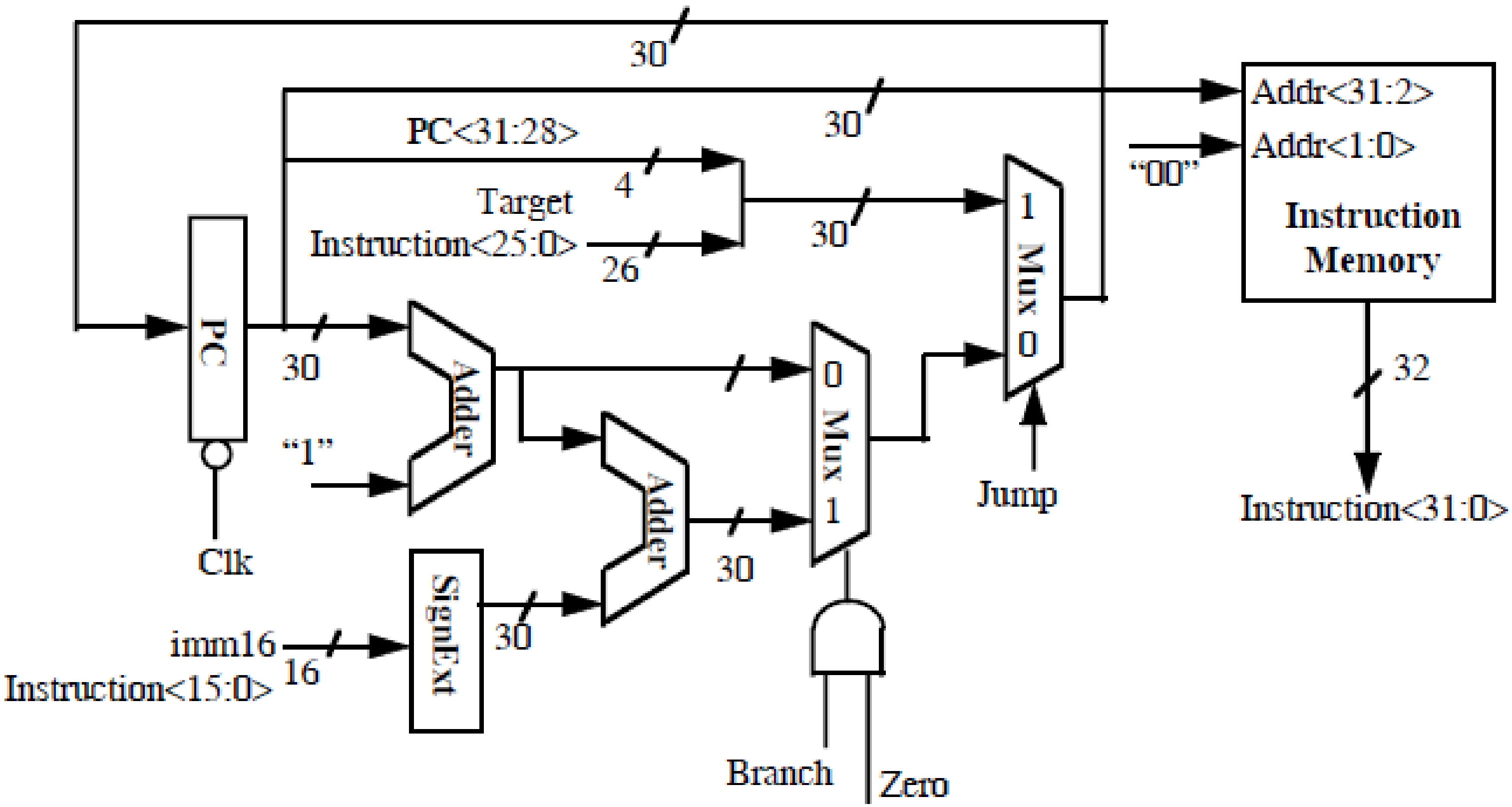
Overview of Instruction Fetch Unit

At a falling clock edge what happens:

- PC gets updated at the falling clock edge
- Fetch the Instruction from the address pointed to by PC
- Pass the PC through the next address logic
- Next value of the PC
  - Sequential Code
    - nextPC = PC + 4
  - Branch and Jump
    - nextPC = "something else"



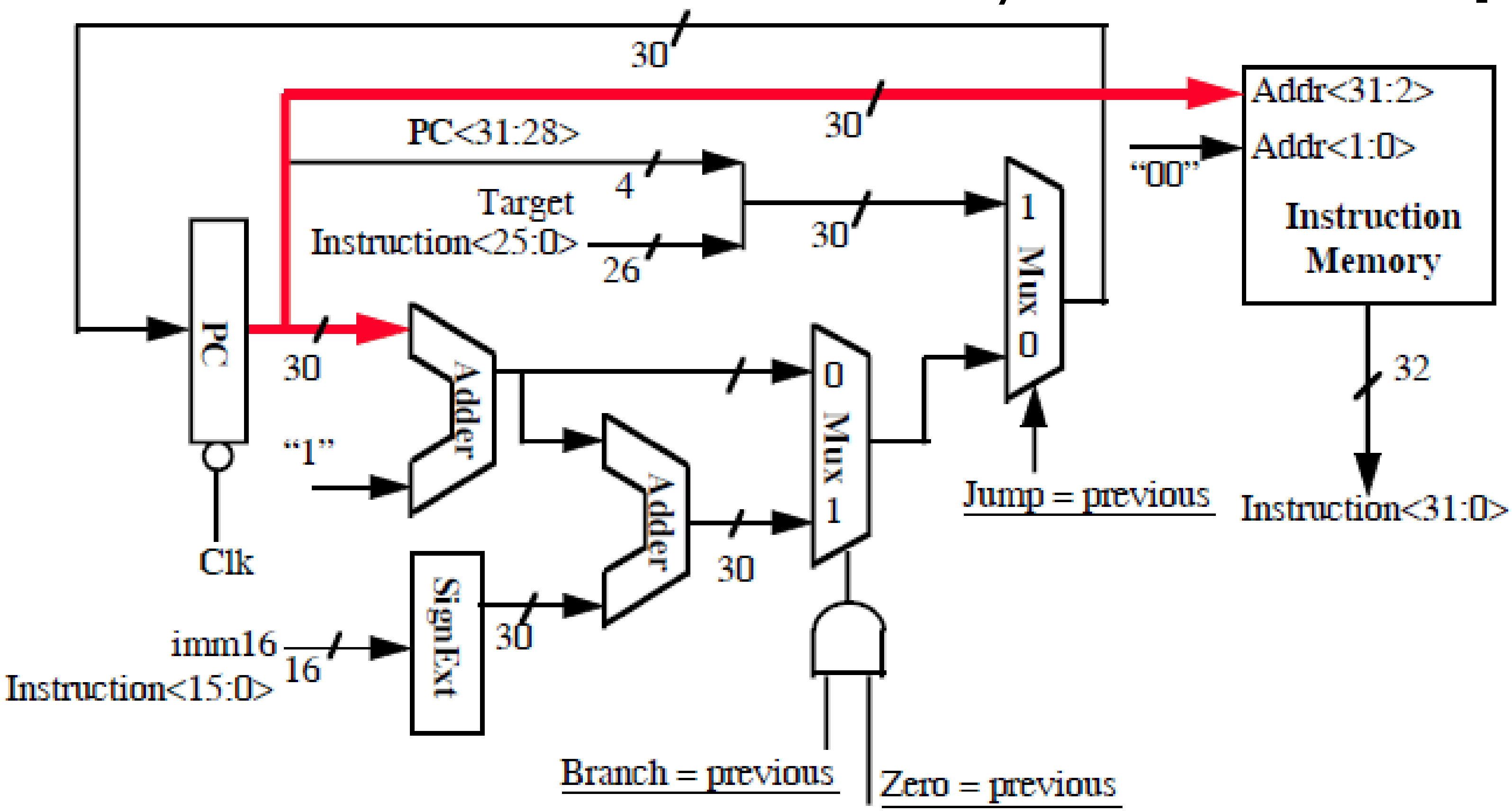
Instruction Fetch Unit



Instruction Fetch Unit at the Beginning of Add / Subtract

The followin two steps are the same for all the instructions.

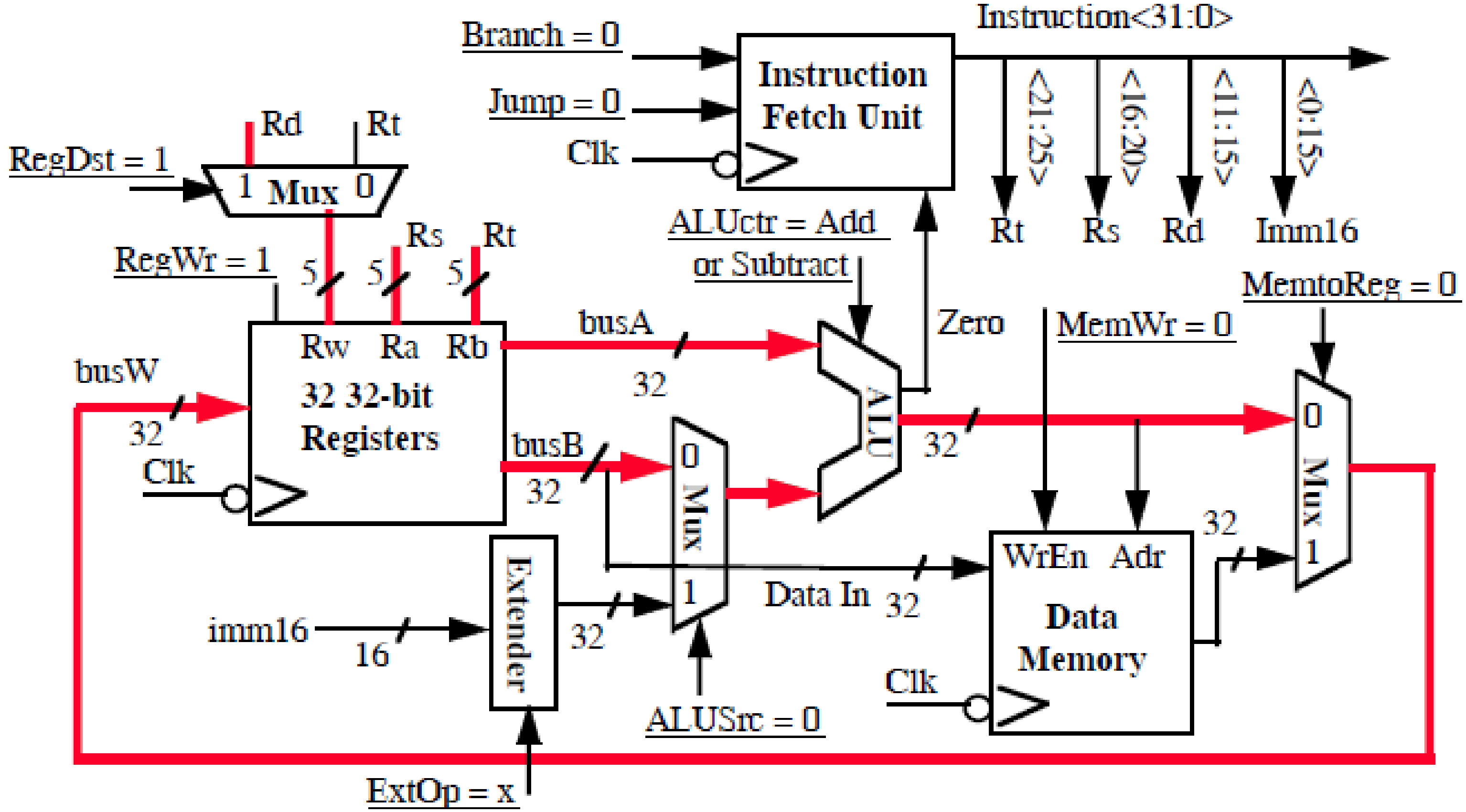
1. PC = nextPC
2. Fetch the instruction from Instruction memory: Instruction = mem[PC]



The Single Cycle Datapath during Add and Subtract

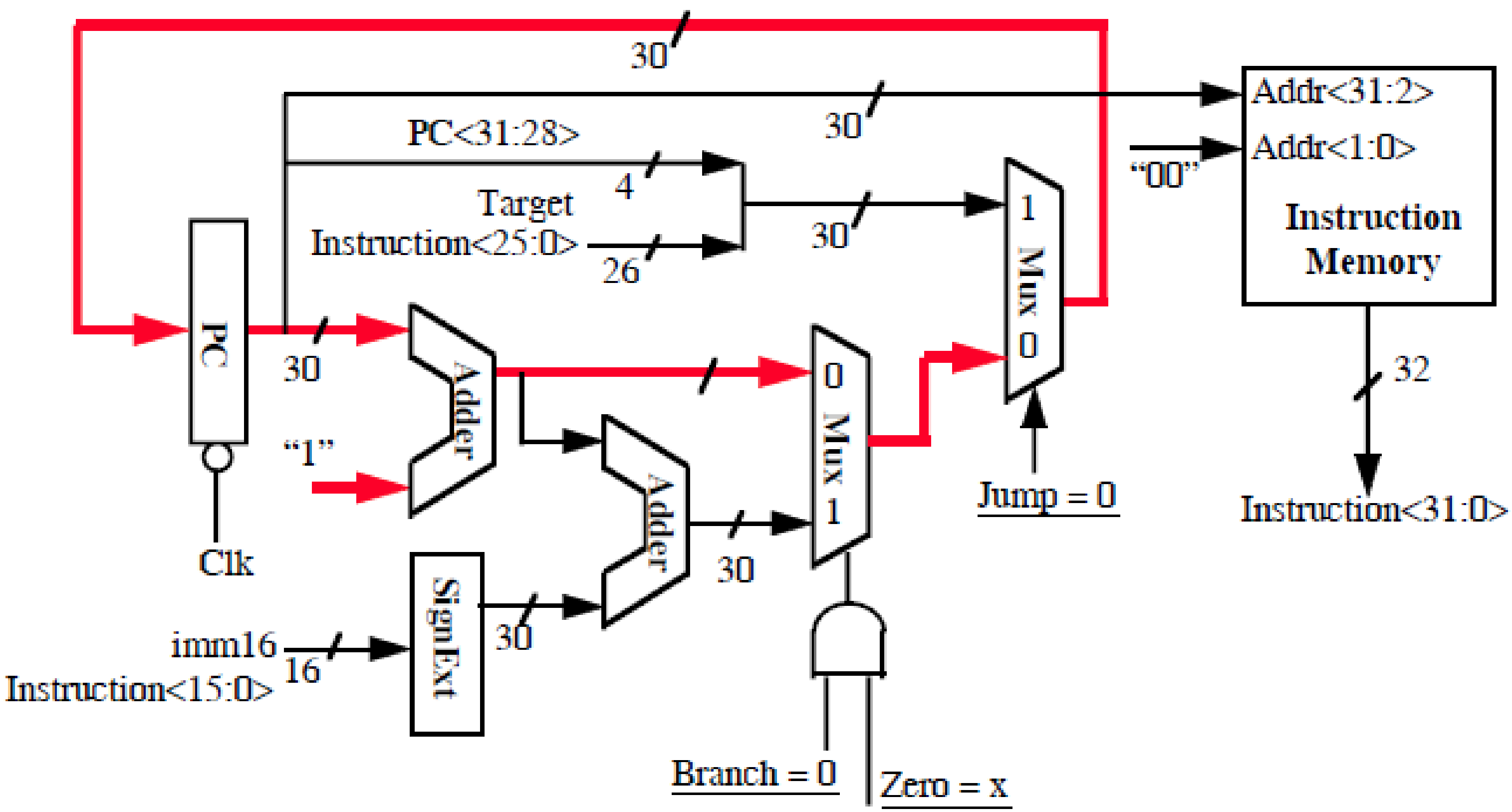
31	26	21	16	11	6	0
op	rs	rt	rd	shamt	funct	

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

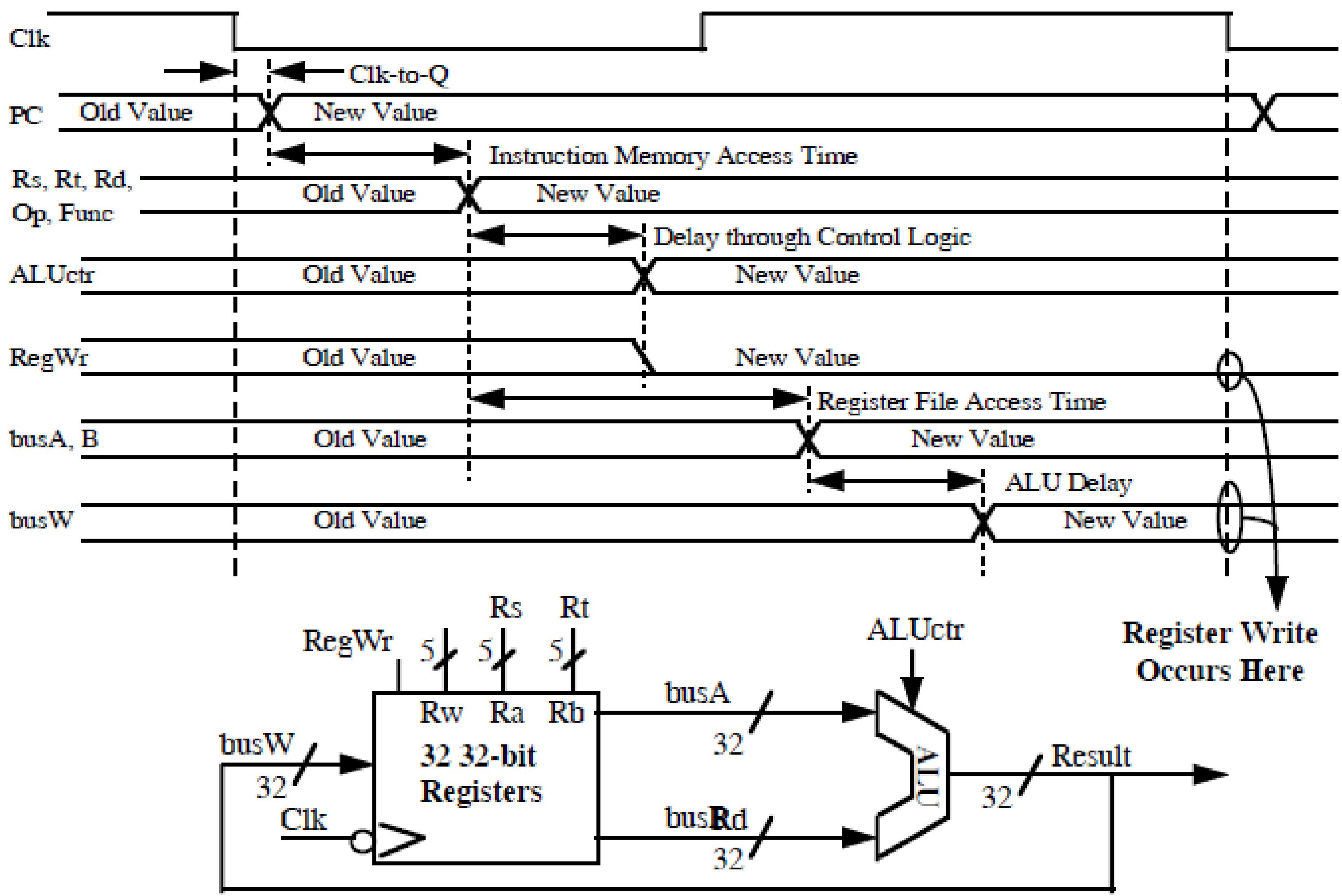


Instruction Fetch Unit at the End of Add and Subtract

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



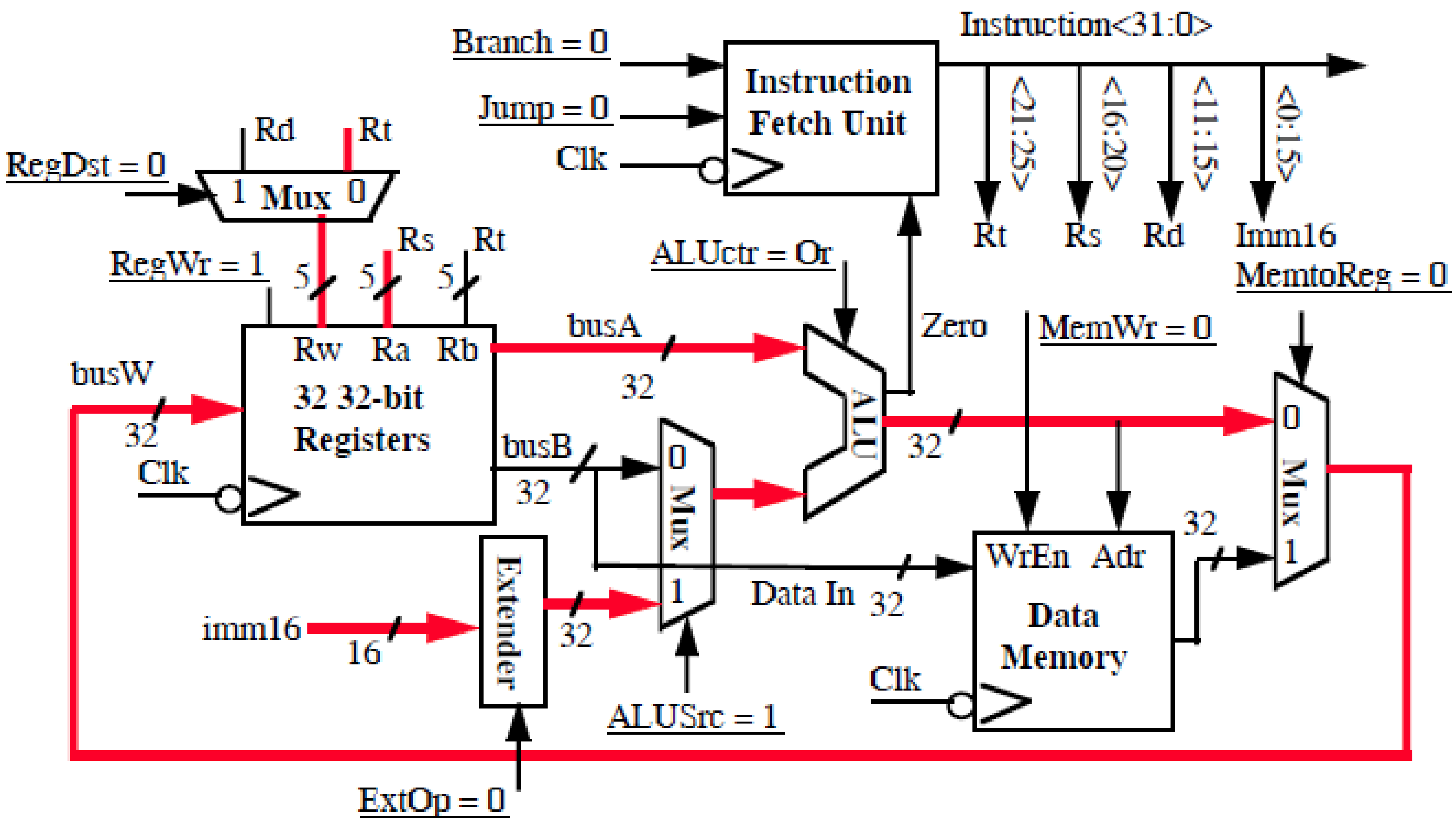
Register – Register Timing



The Single Cycle Datapath during Or Immediate

31 26	26 21	21 16	16 0
op	rs	rt	immediate

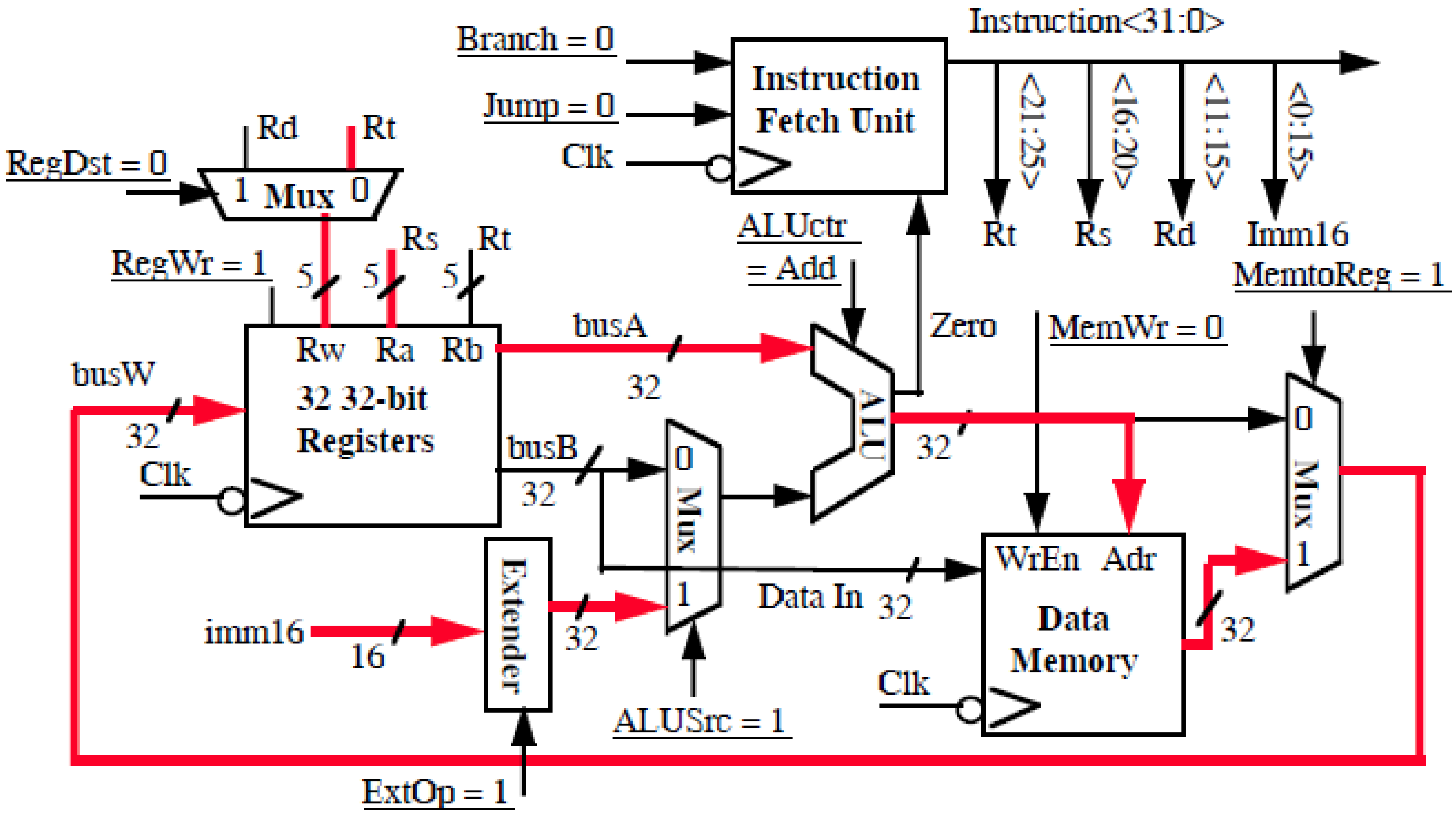
° R[rt] <- R[rs] or ZeroExt[Imm16]



The Single Cycle Datapath during Load

31 26	21	16	0
op	rs	rt	immediate

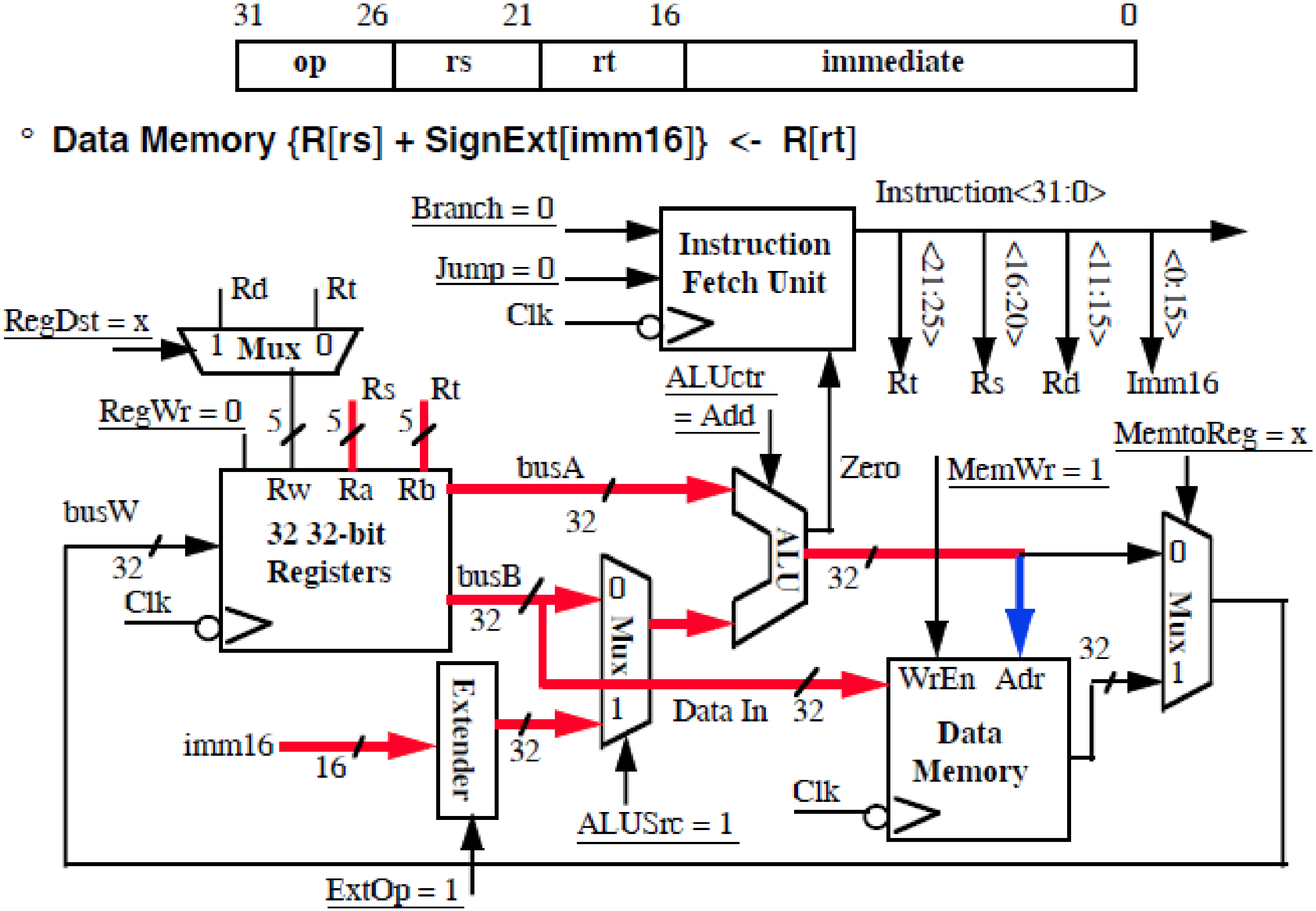
- R[rt] <- Data Memory {R[rs] + SignExt[imm16]}



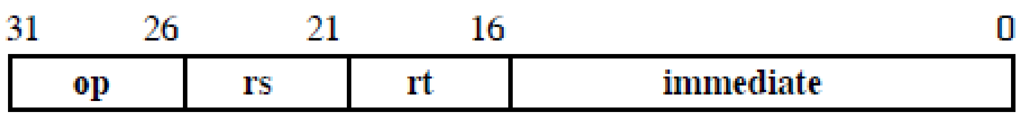
The Single Cycle Datapath during Store

31 op (26)	rs (21)	rt (16)	immediate (0)

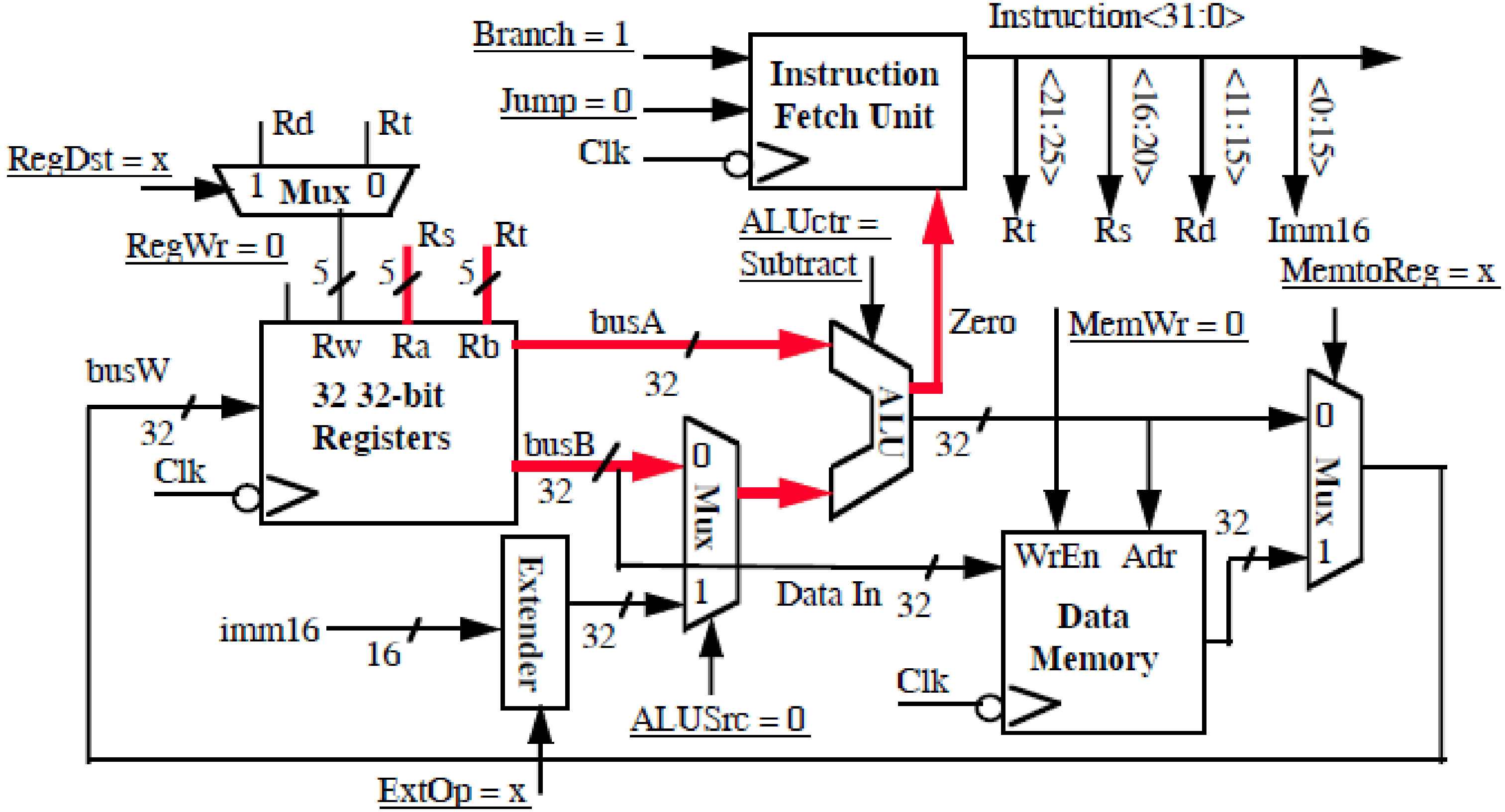
- Data Memory {R[rs] + SignExt[imm16]} <- R[rt]



The Single Cycle Datapath during Branch



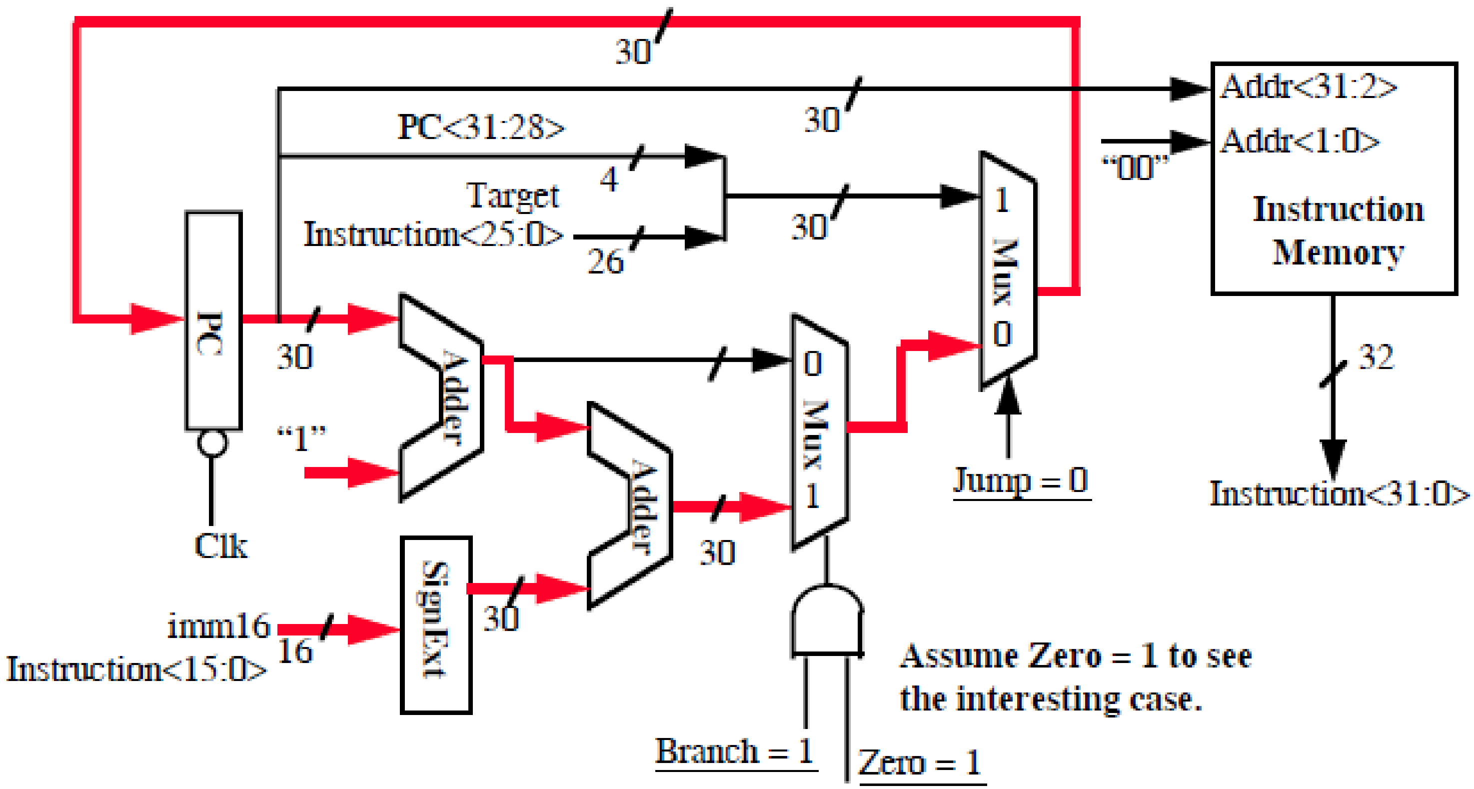
- if (R[rs] - R[rt] == 0) then Zero <- 1 ; else Zero <- 0



Instruction Fetch Unit at the End of Branch

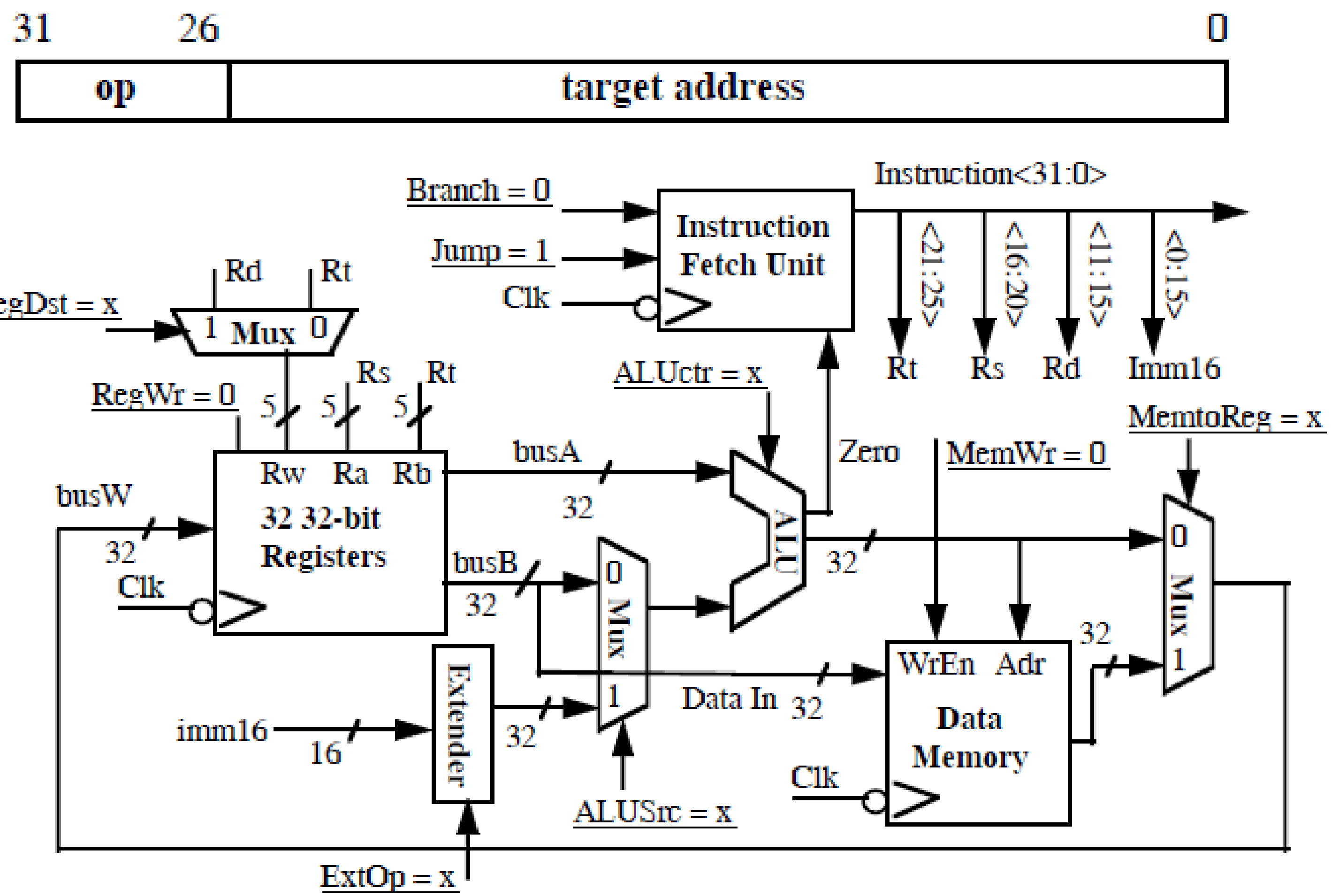
31	26	21	16	0
op	rs	rt	immediate	

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



The Single Cycle Datapath during Jump

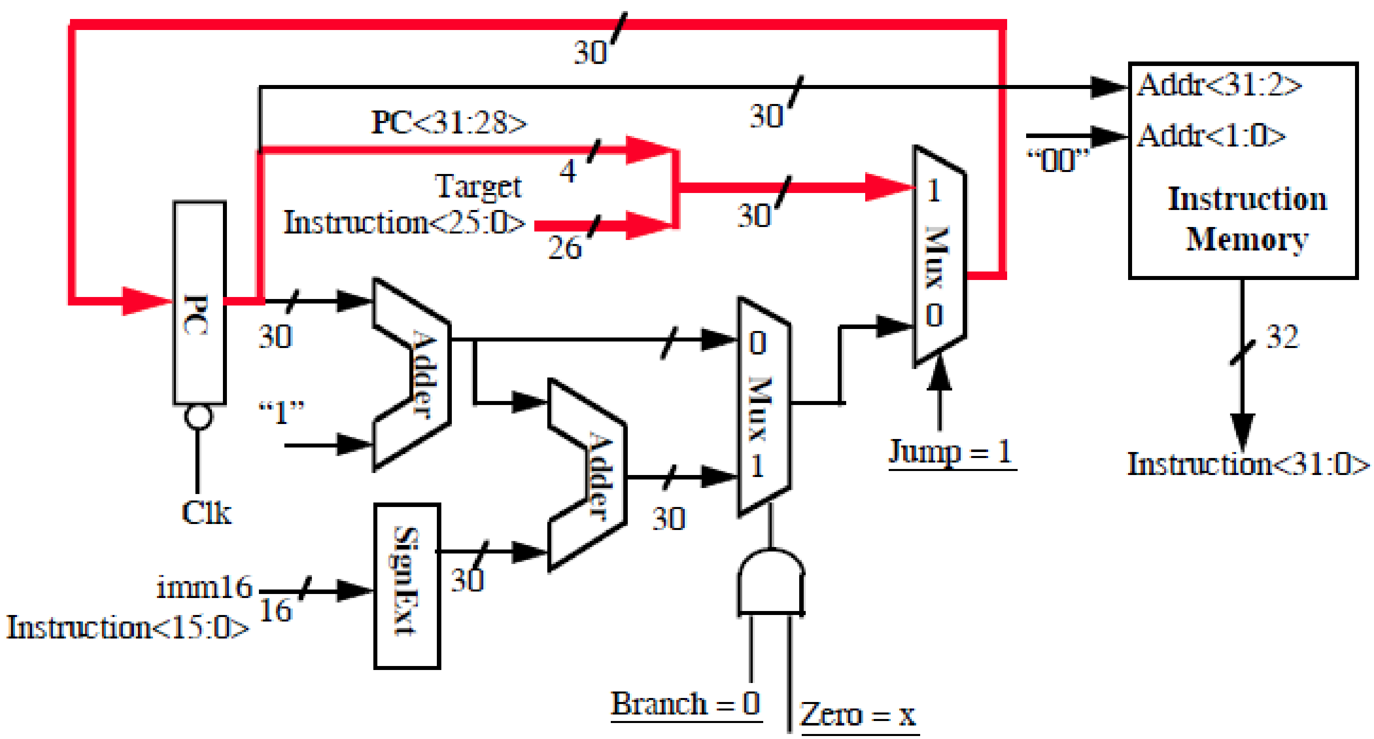
- Nothing to do! Make sure control signals are set correctly!



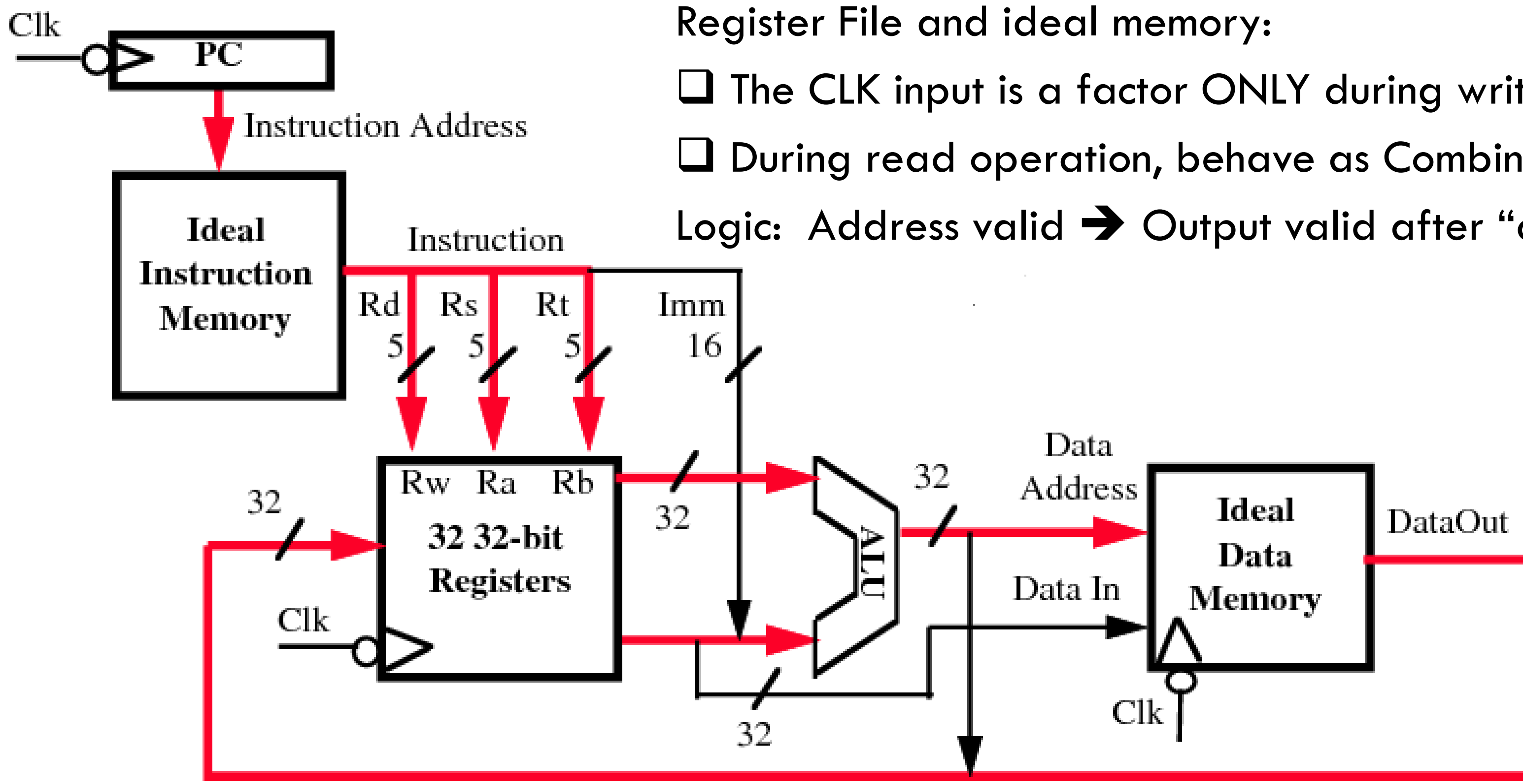
Instruction Fetch Unit at the End of Jump

31	26	0
op	target address	

- PC <- PC<31:29> concat target<25:0> concat "00"



An Abstract View of the Critical Path



Register File and ideal memory:

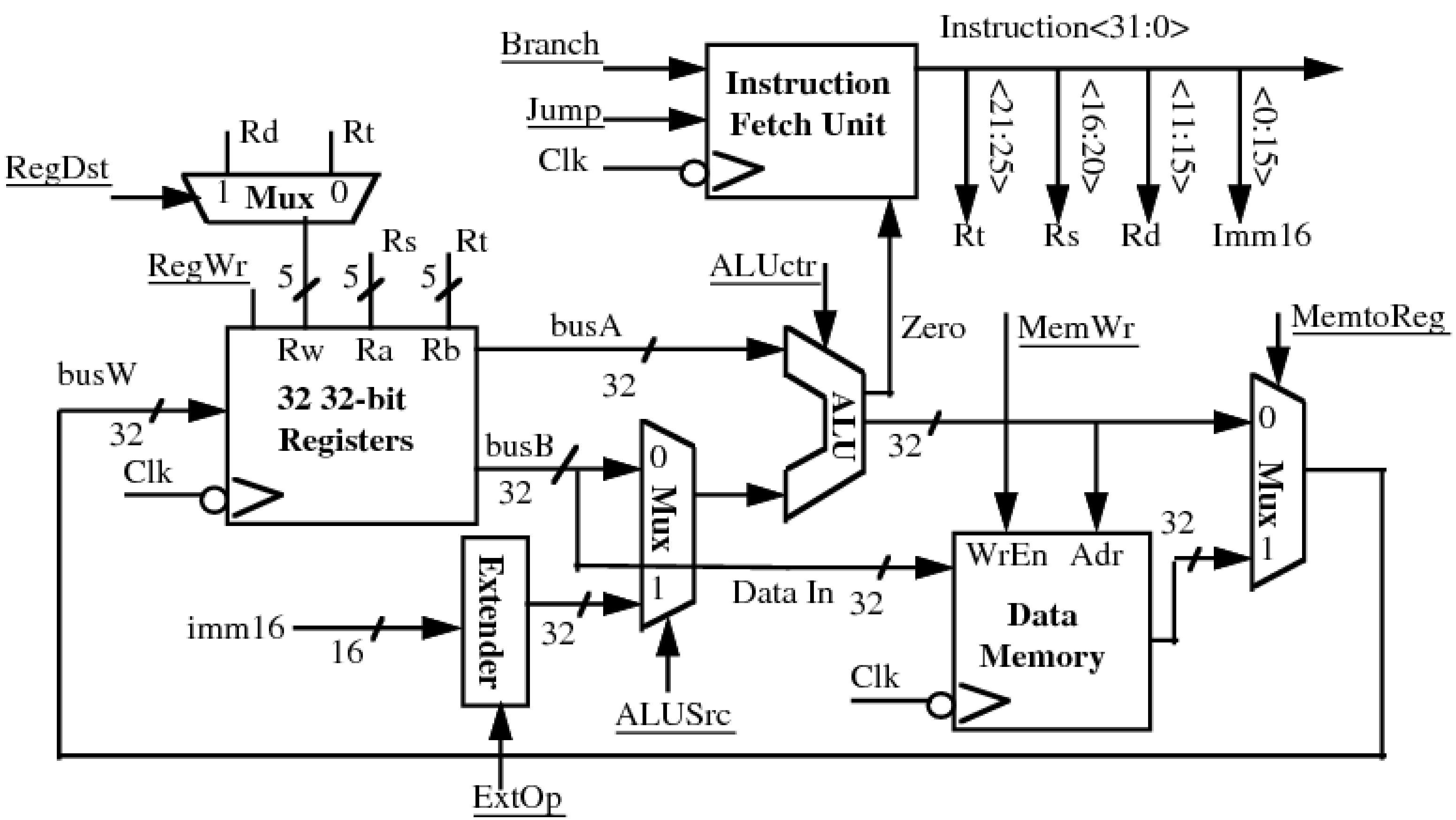
- The CLK input is a factor ONLY during write operation
- During read operation, behave as Combinational

Logic: Address valid ➔ Output valid after "access time"

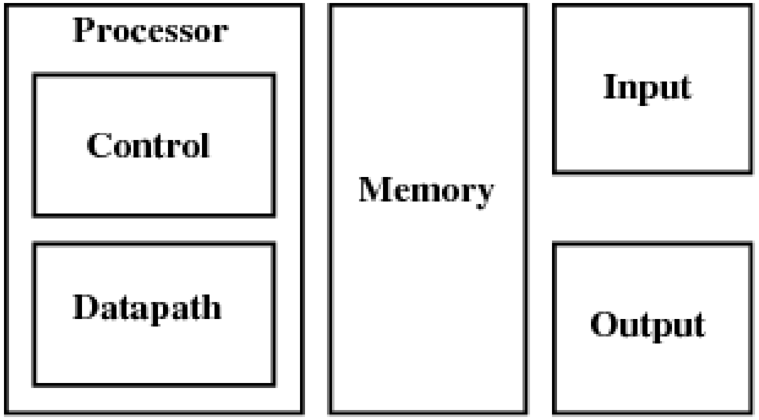
Critical Path (Load Operation) = PC's Clk-to-Q + Instruction Memory's Access Time + Register File's Access Time + ALU to Perform 32-bit Add + Data Memory Access Time + Setup Time for Register File Write

Putting it all together: A Single Cycle Datapath

We have everything except control signals (underline)



The Big Picture: Where are we Now?



- The Five Classic Components of a Computer
- Next Topic: Control Path Design

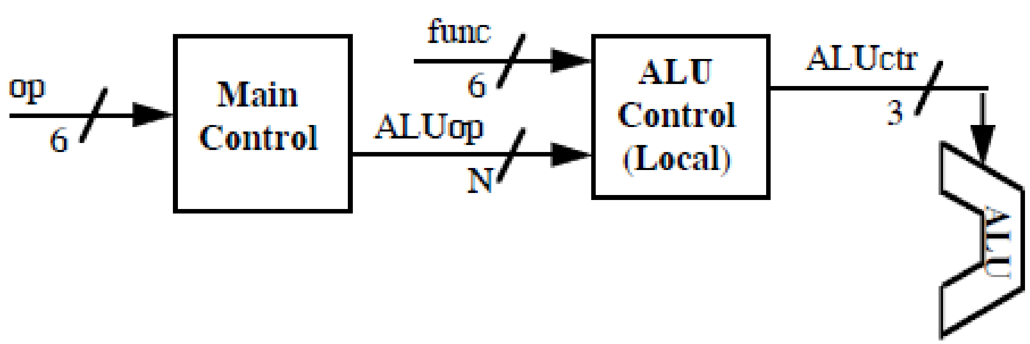
A Summary of Control Signals

func	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	0	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

	31	26	21	16	11	6	0	
R-type	op	rs	rt	rd	shamt	funct		add, sub
I-type	op	rs	rt	immediate				ori, lw, sw, beq
J-type	op	target address						jump

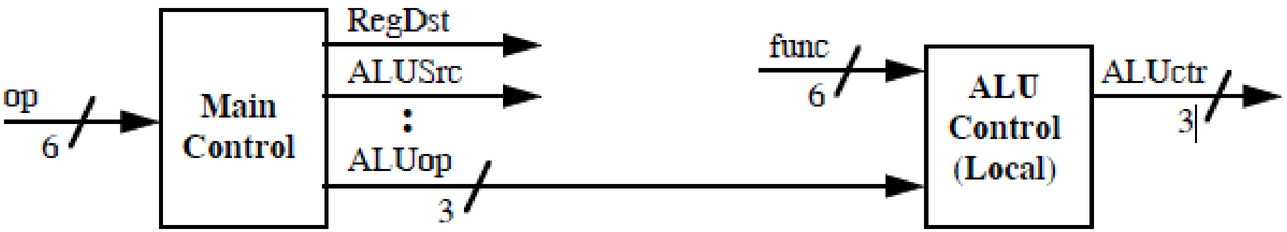
The Concept of Local Decoding

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 “Truth Table” for the Main Control

Key Idea: Two levels of Control logic.

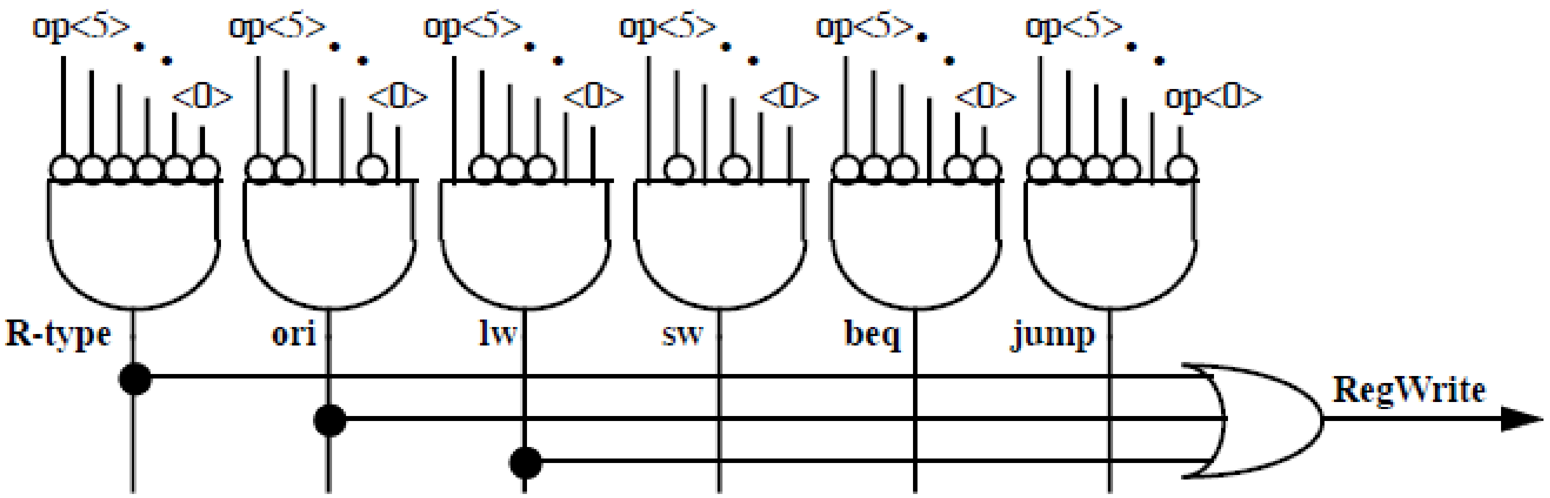


op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx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

Question: Can you write the truth table for the ALU control keeping in mind the ALU we designed in the class?

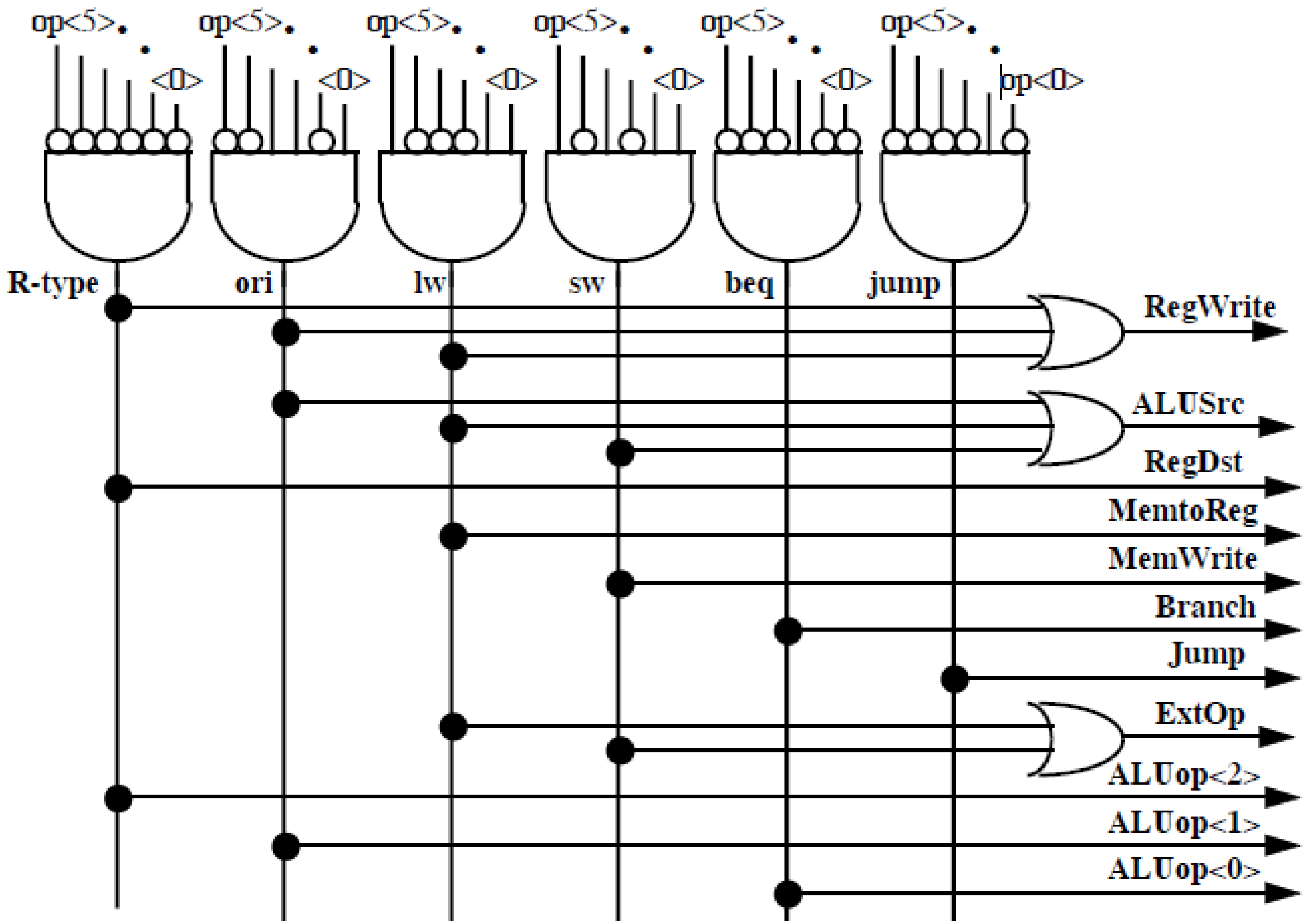
The “Truth Table” for RegWrite

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

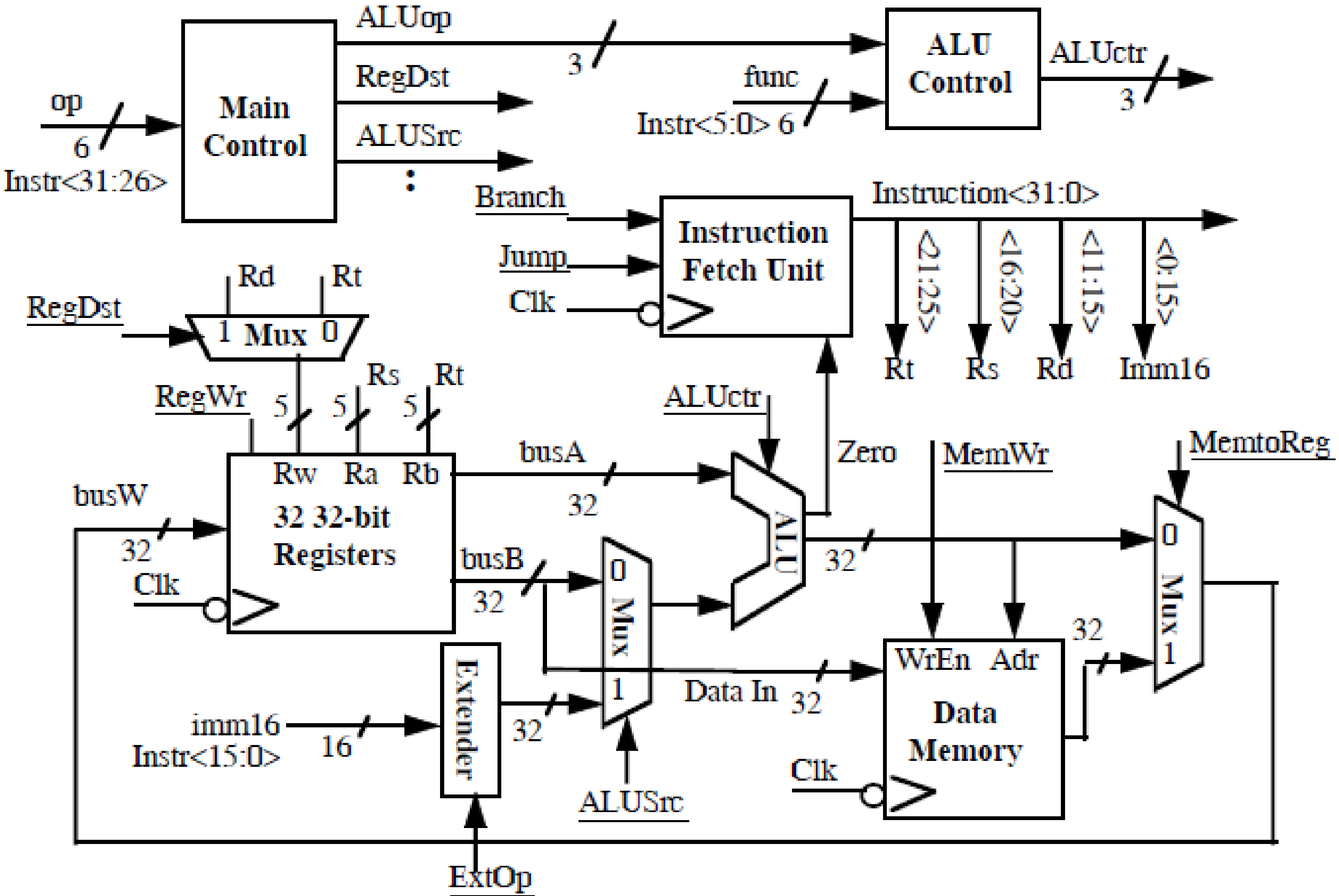


PLA Implementation of Main Control

Hmm! What is PLA?



Putting it All Together: A Single Cycle Processor



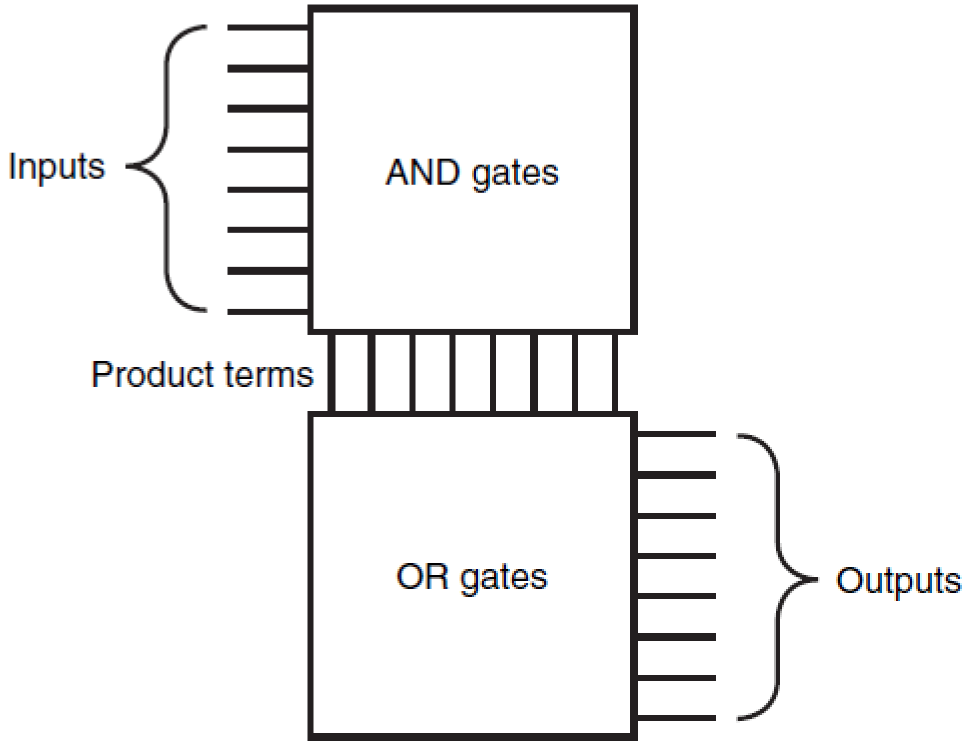
Drawback of this Single Cycle Processor

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

We are assuming Clock Skew is zero

Programmable Logic Arrays

- PLAs can be used to realize combinational circuits



PLAs