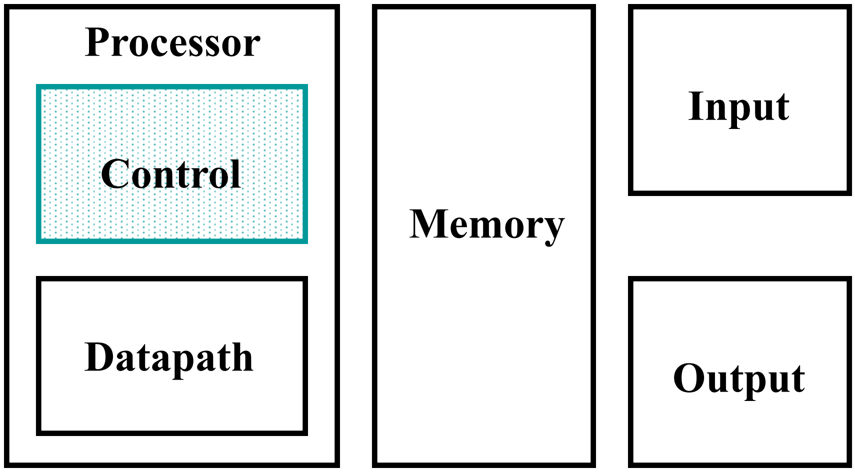
CSL7070: Computer Architecture
Lecture 8, 24th February 2023

Dip Sankar Banerjee

॥ त्वं ज्ञानमयो विज्ञानमयोऽसि ॥

Indian Institute of Technology, Jodhpur
January-April 2023

Control Design



- Next: Designing the Control for the Single Cycle Datapath

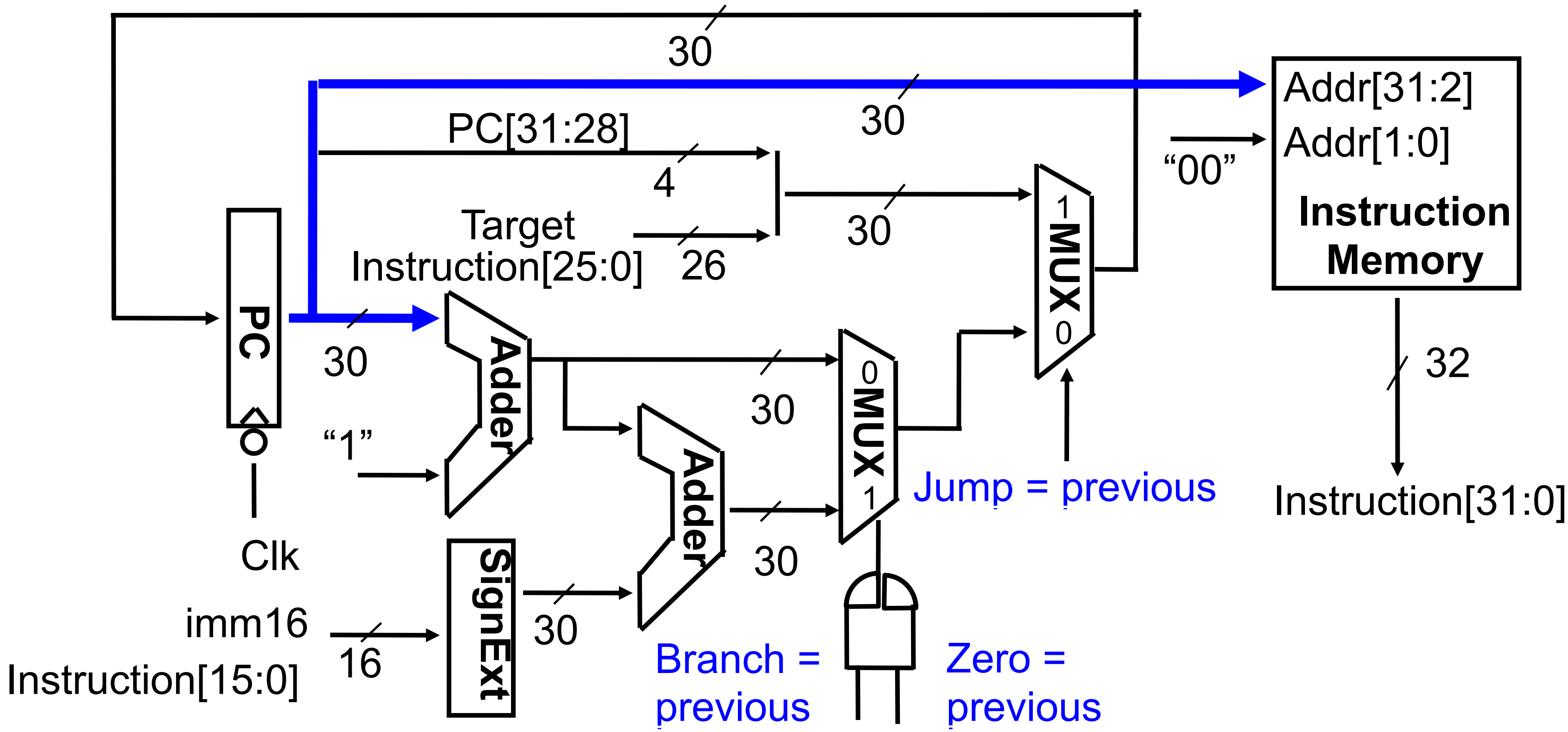
Adding Control

- Analyze datapath and RTLs for control
  - Identify control points for pieces of the datapath
    - Instruction Fetch Unit
    - Integer function units
    - Memory
  - Categorize type of control signal
    - Flow of data through multiplexors
    - Writes of state information
  - Derive control signal values for each instruction
- Design and implement control with logic/PLA/ROM (for single cycle & pipelined)

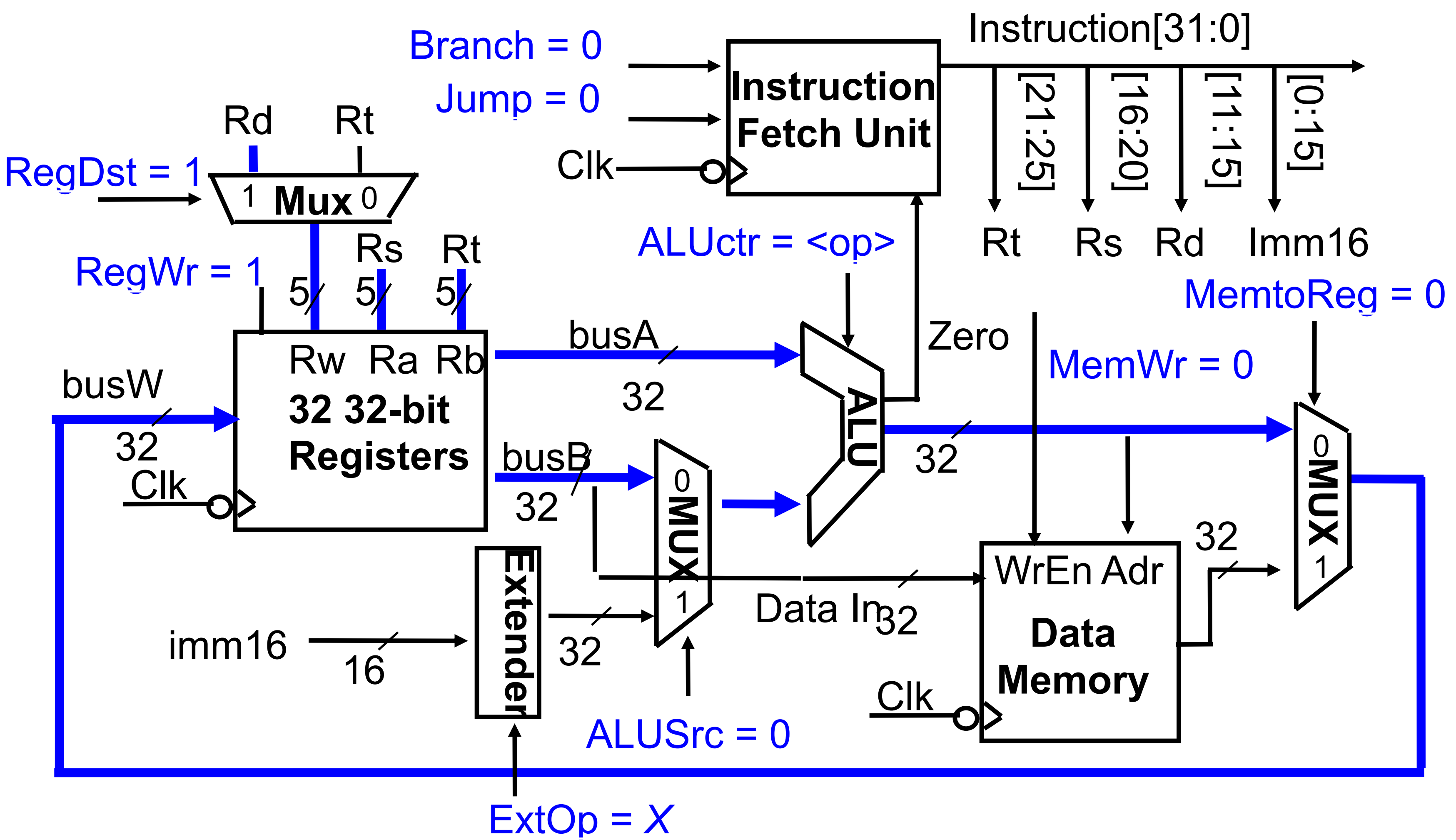
Instruction Fetch (first part)

- Always fetch next instruction

`Mem[PC];`

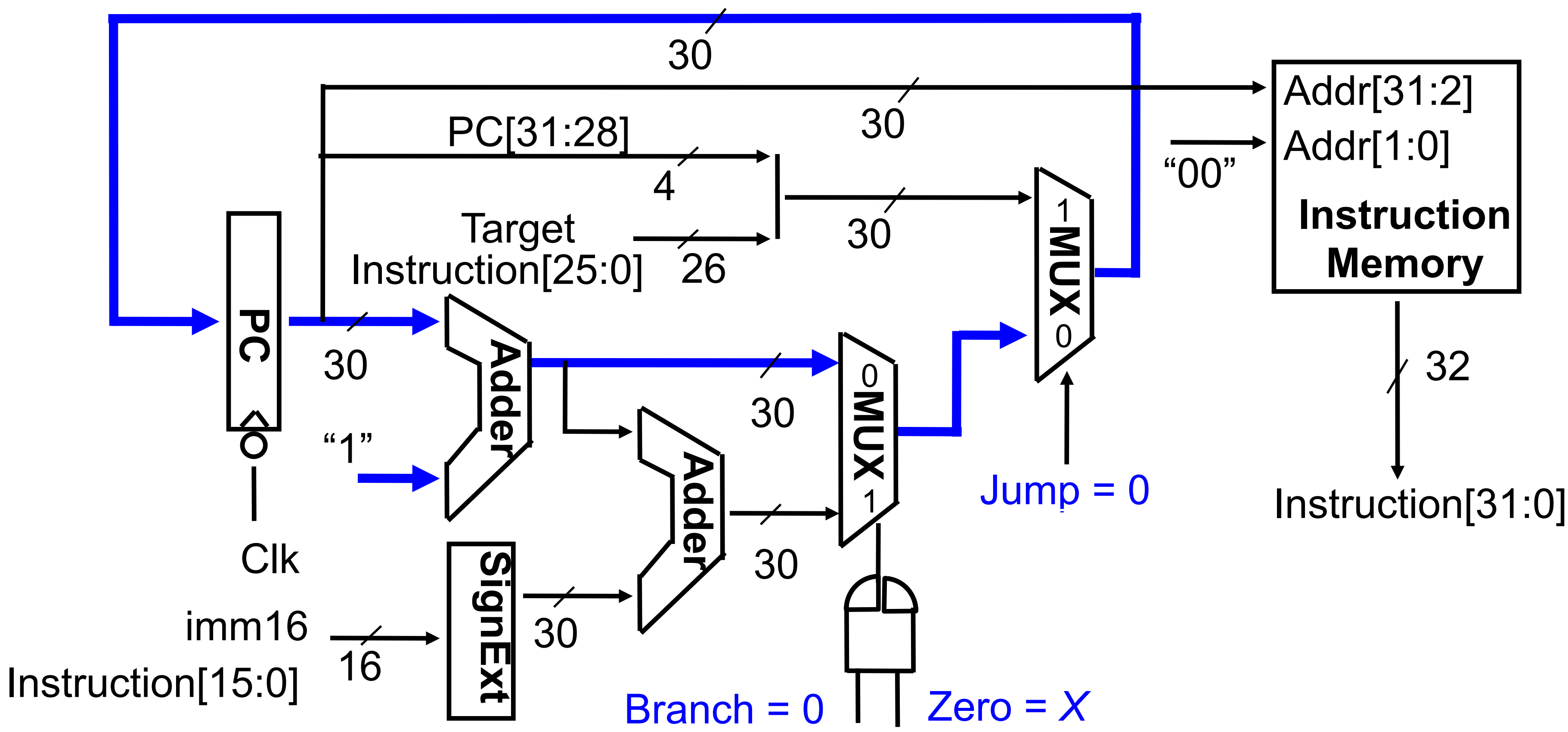


Control for Arithmetic

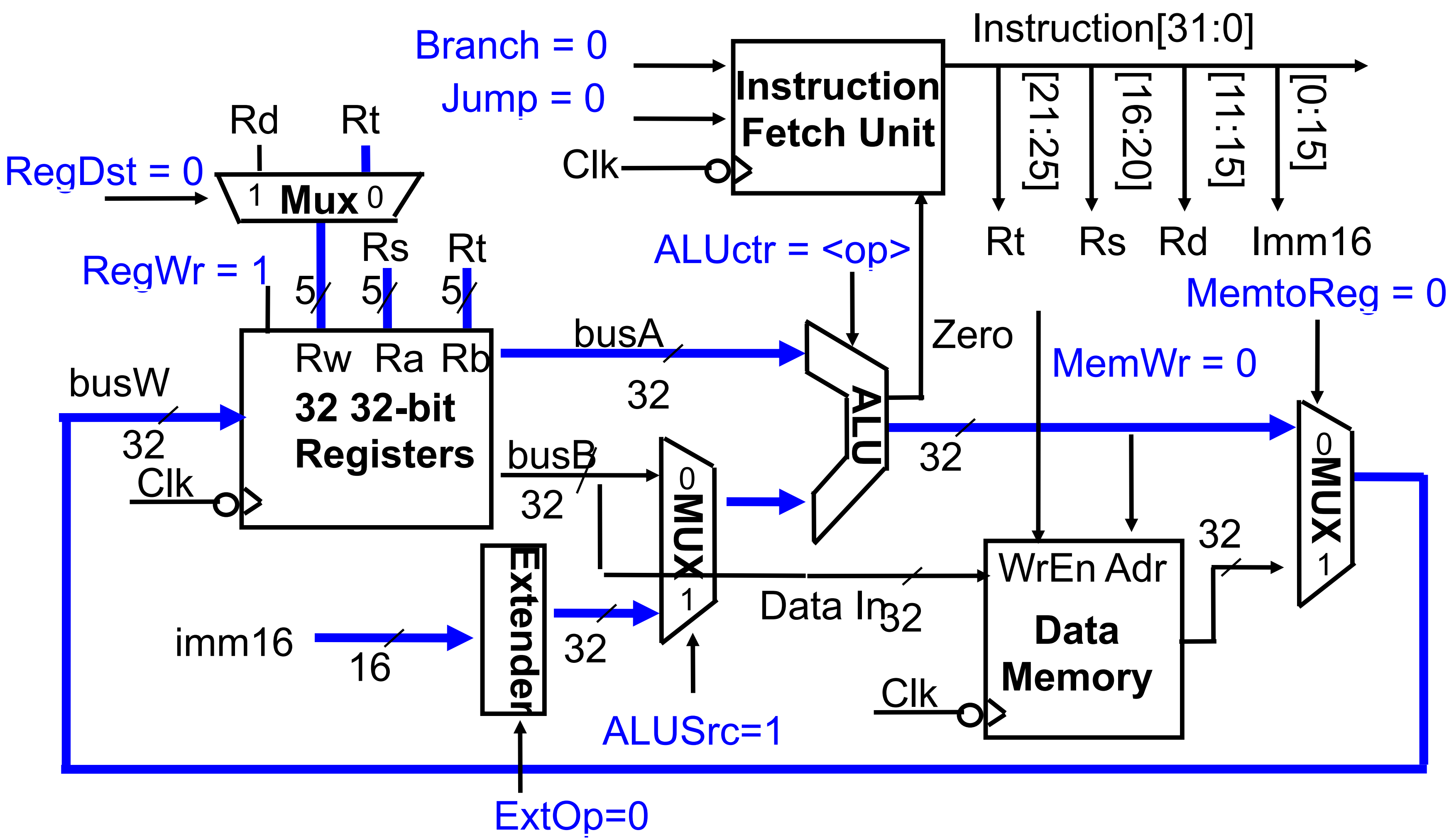


Instruction Fetch at End

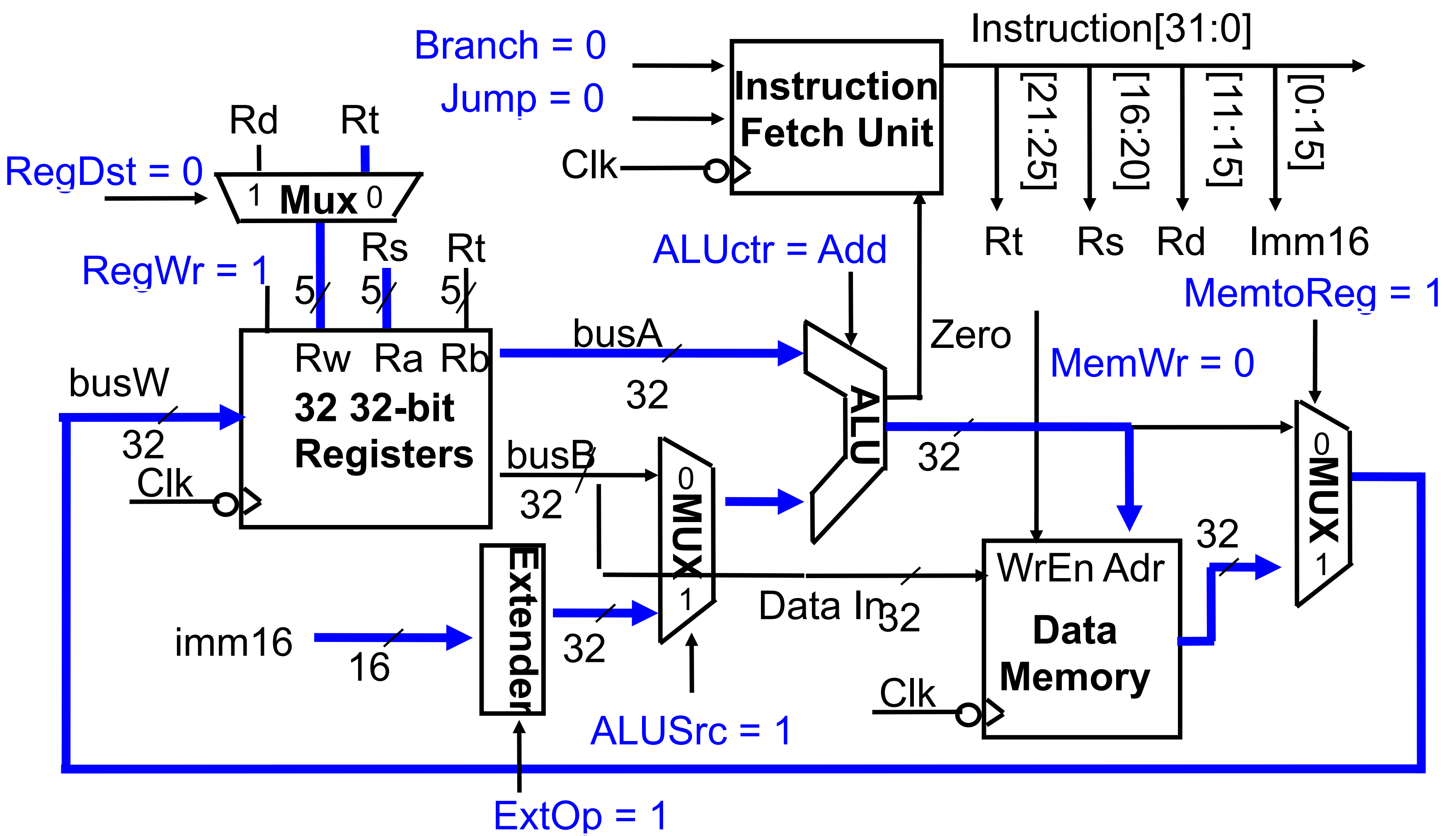
- Increment PC: `PC = PC+4;` (for all but Branch/Jump)



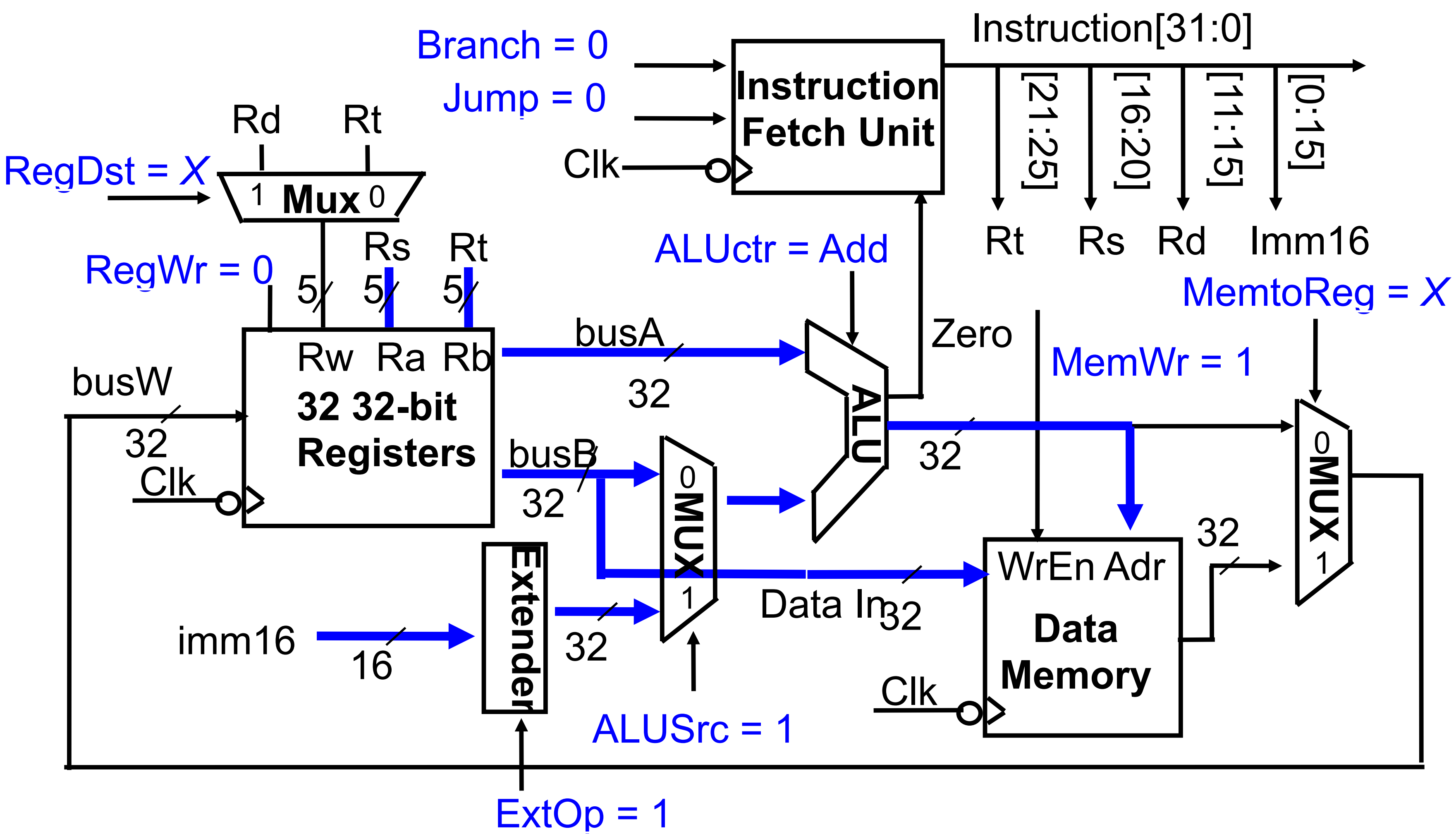
Control for Immediate (`ori`)



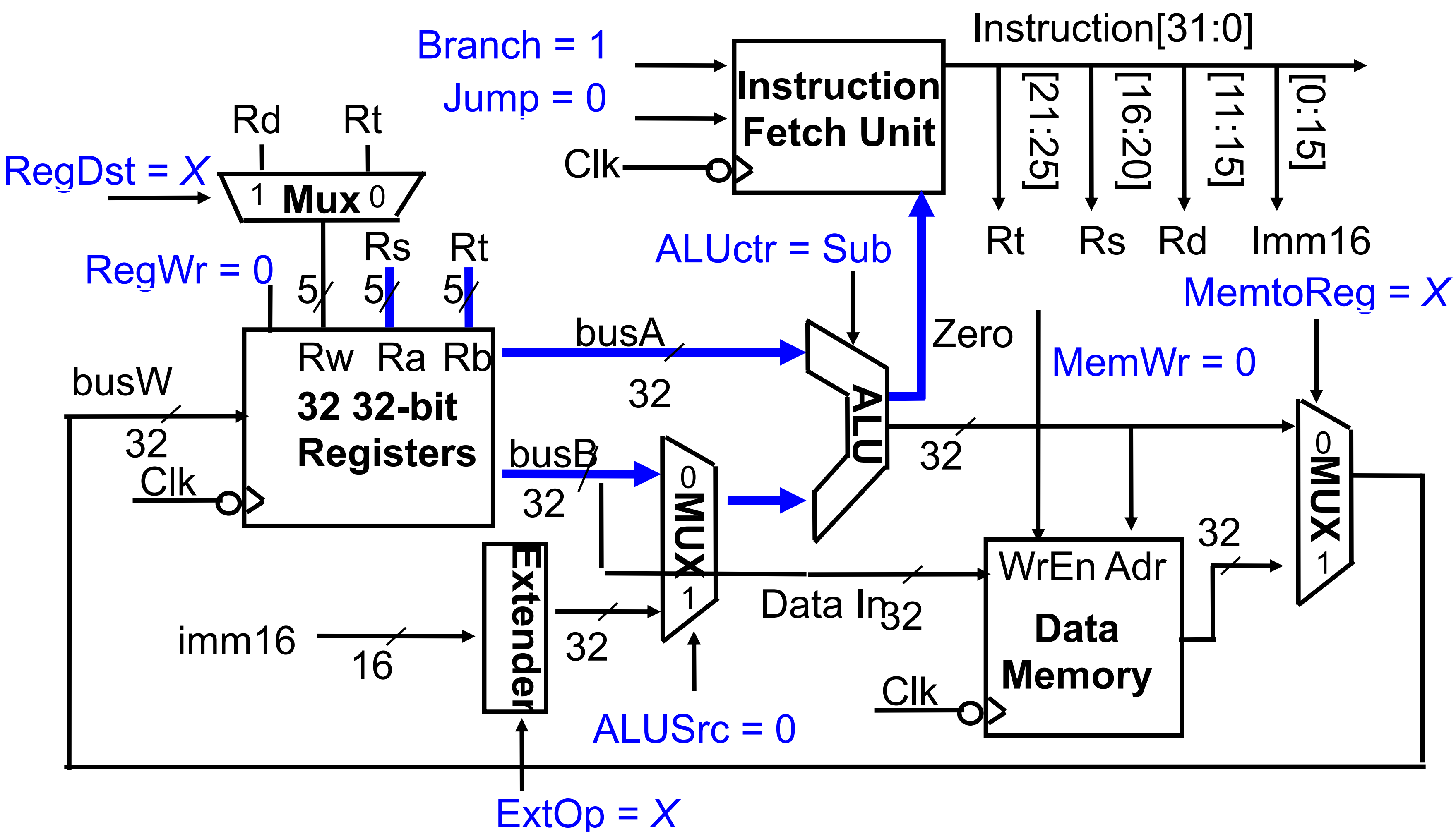
Control for Load (`lw`)



Control for Store (`sw`)

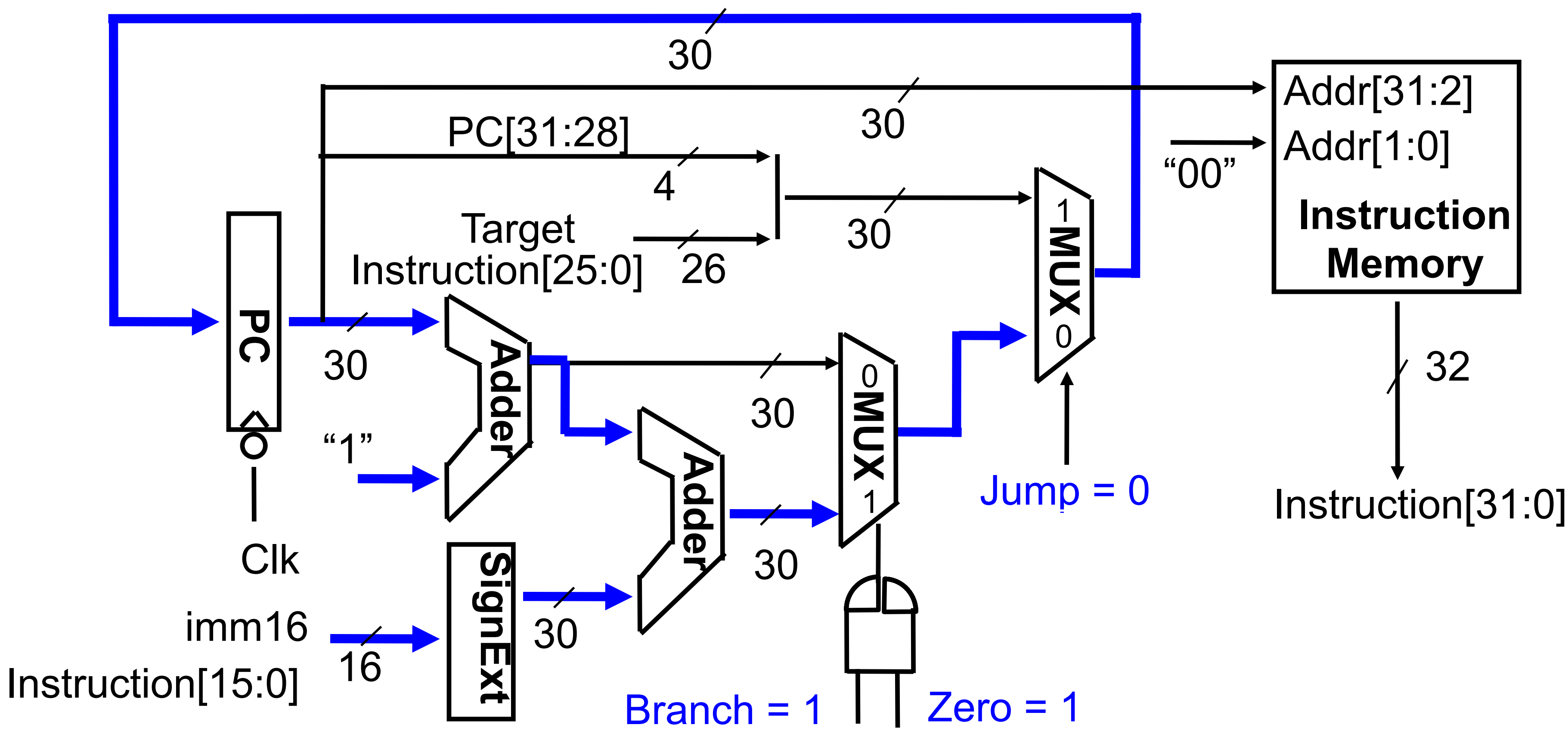


Control for Branch (beq)

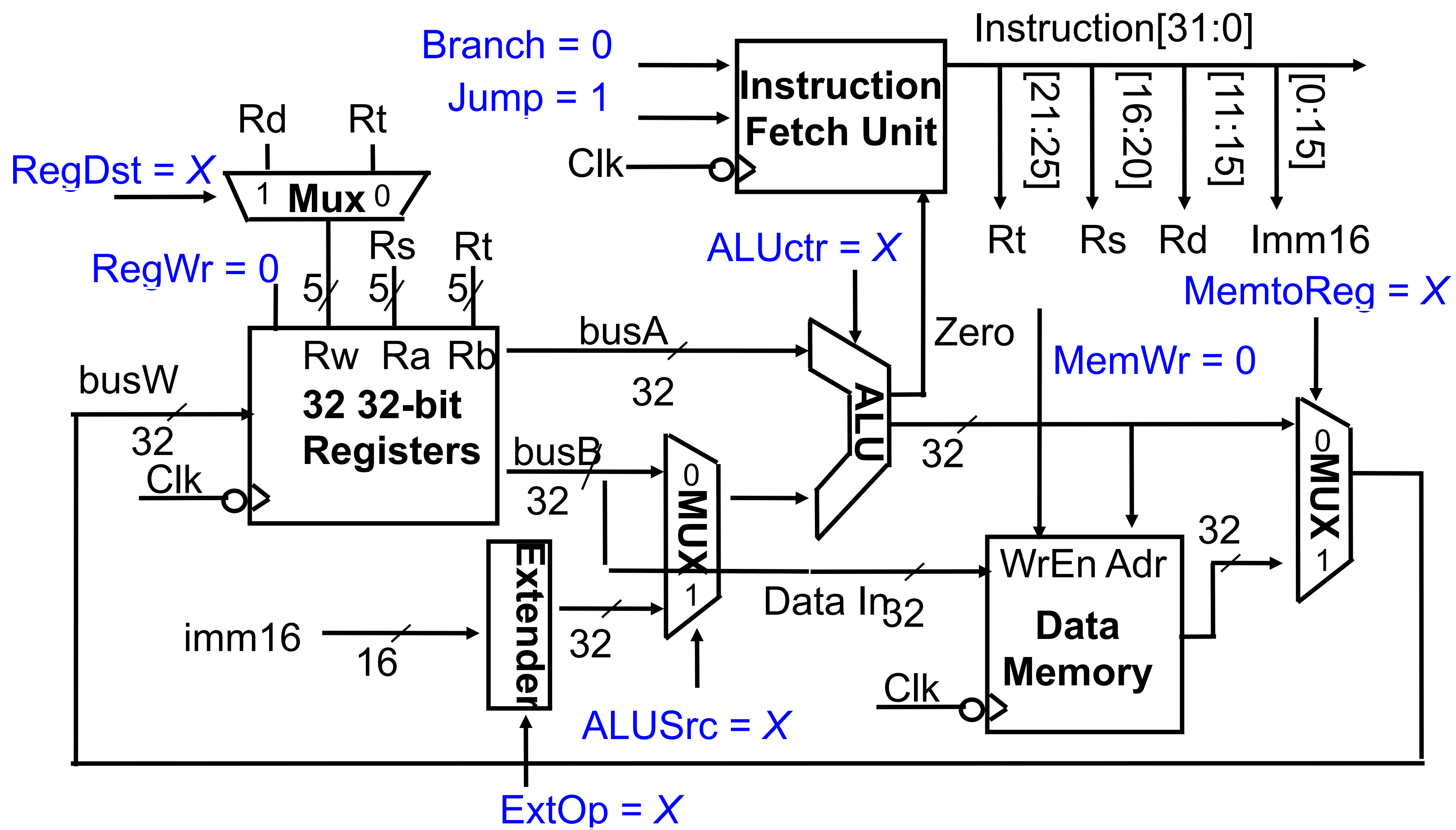


Instruction Fetch (beq)

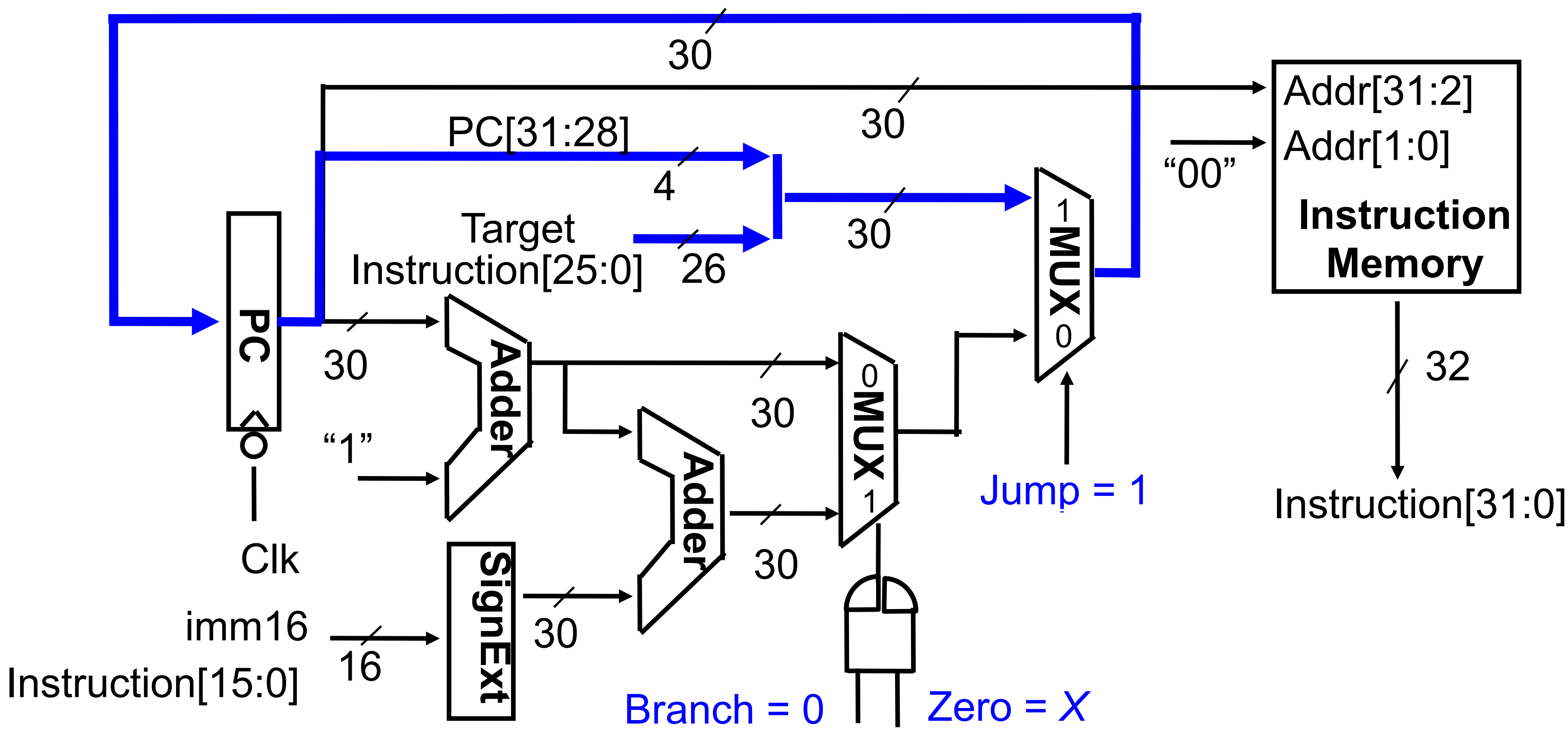
Consider the interesting case where we branch (Zero = 1)



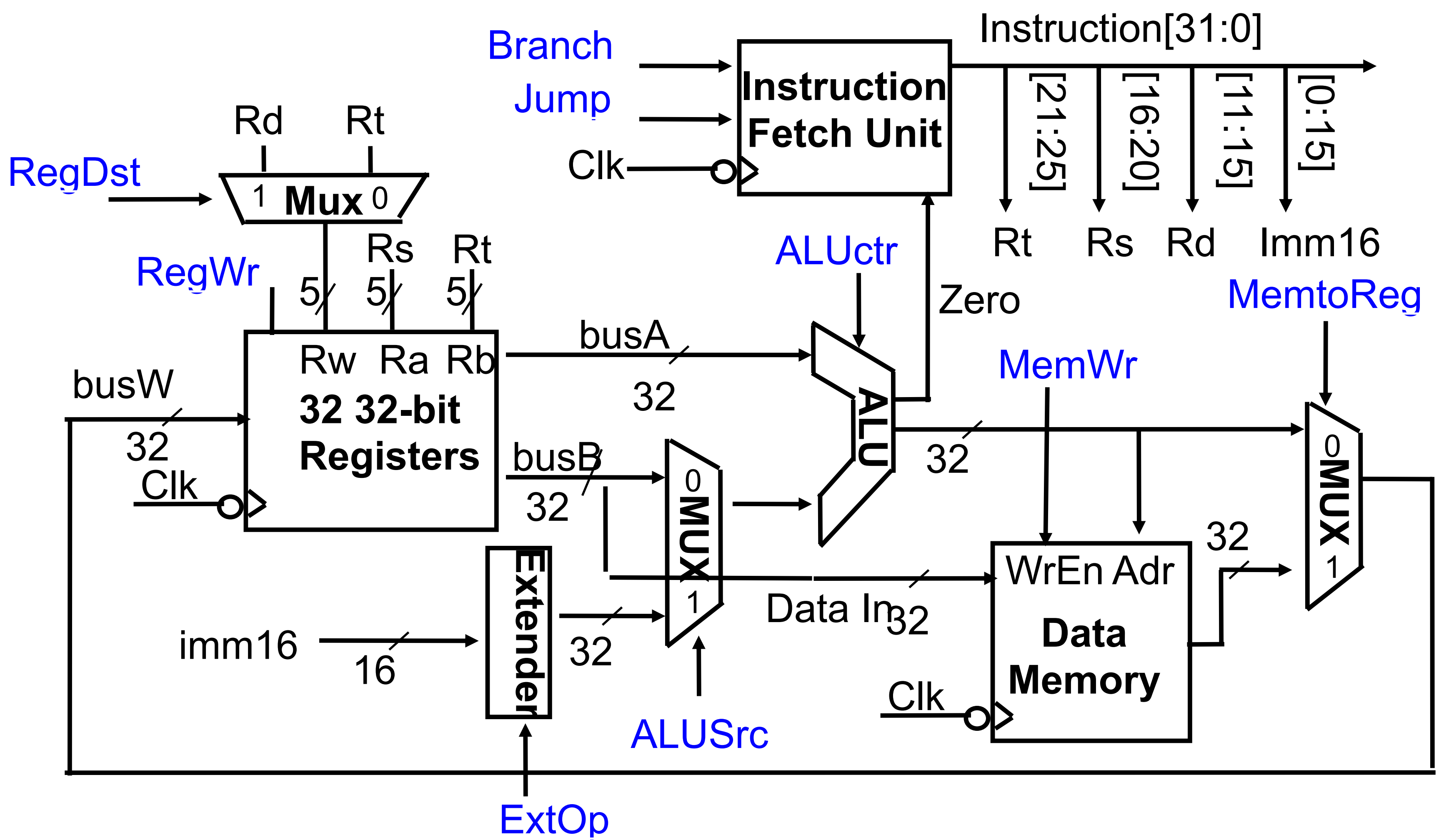
Control for Jump (`j`)



Instruction Fetch (j)



Control Path



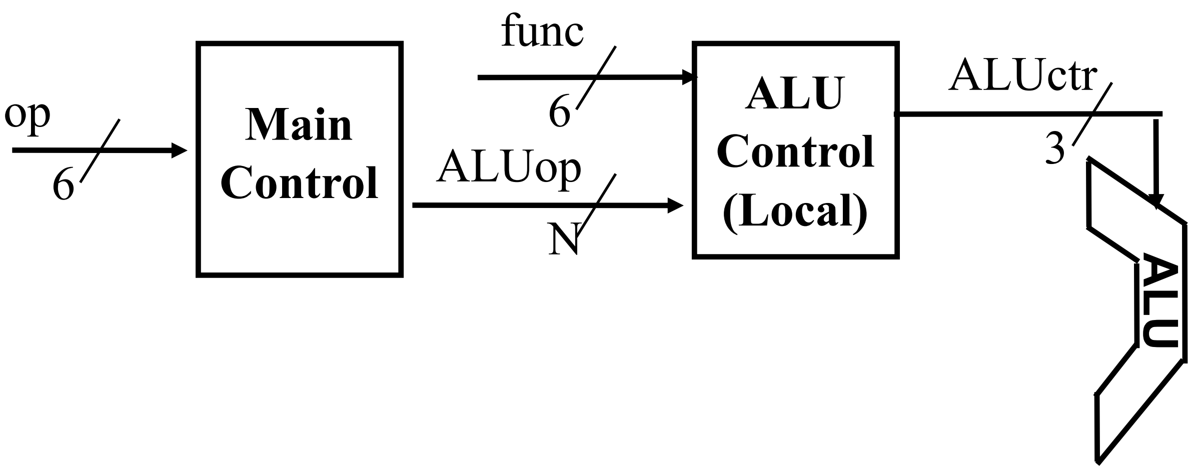
Summary of Control Signals

coding from green card → func, op

	add	sub	ori	lw	sw	beq	jump
func	10 0000	10 0010	**Not Important**				
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	Sub	Or	Add	Add	Sub	xxx

Multilevel Decoding

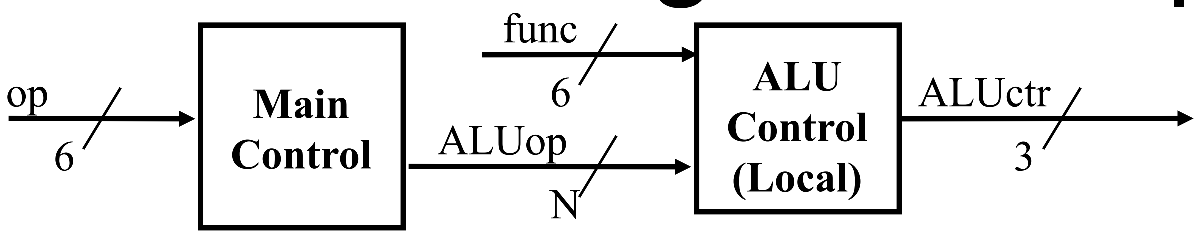
- 12-input control will be very large (2^{12} = 4096)
- To keep decoder size smaller, decode some control lines in each stage
- Since only R-type instructions (with op = 000000) need function field bits, give these to ALU control



Multilevel Decoding: Main Control Table

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

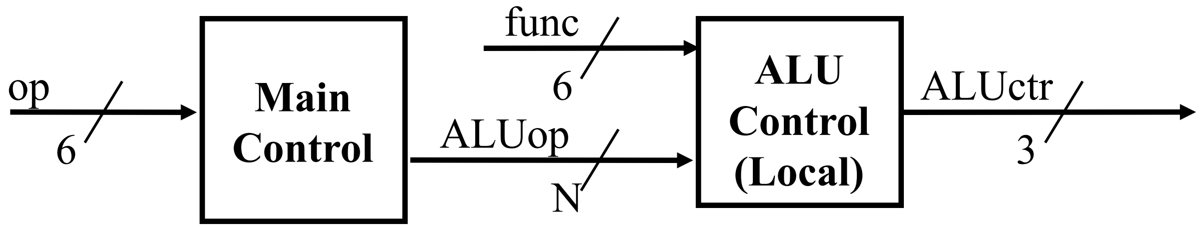
The Encoding of ALUop



- 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 wide to represent:
  - (1) “R-type” instructions
  - “I-type” instructions that require the ALU to perform:
    - (2) Or, (3) Add, (4) Subtract, and (5) And (e.g. 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

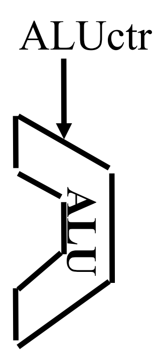
The Decoding of the "func" Field



	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

	31–26	26–21	21–16	16–11	11–6	6–0
R-type	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**

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



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

Truth Tables

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

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

ALUop			func				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

The Logic Equation for ALUctr<2>

ALUop bit<2>	ALUop bit<1>	ALUop bit<0>	func bit<3>	func bit<2>	func bit<1>	func 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 func<3> a don't care

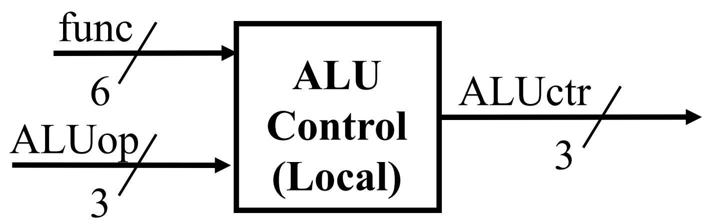
- ALUctr<2> = !ALUop<2> & ALUop<0> +
  ALUop<2> & !func<2> & func<1> & !func<0>

The Logic Equation for ALUctr<1>

ALUop			func				
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

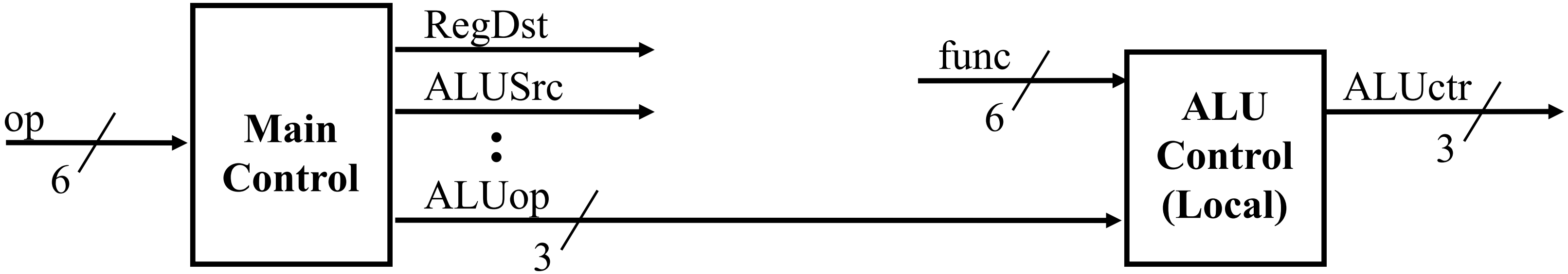
- ALUctr<1> = !ALUop<2> & !ALUop<1> +
  ALUop<2> & !func<2> & !func<0>

The ALU Control Logic



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

Main Control Truth Table



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

Truth Table for RegWrite

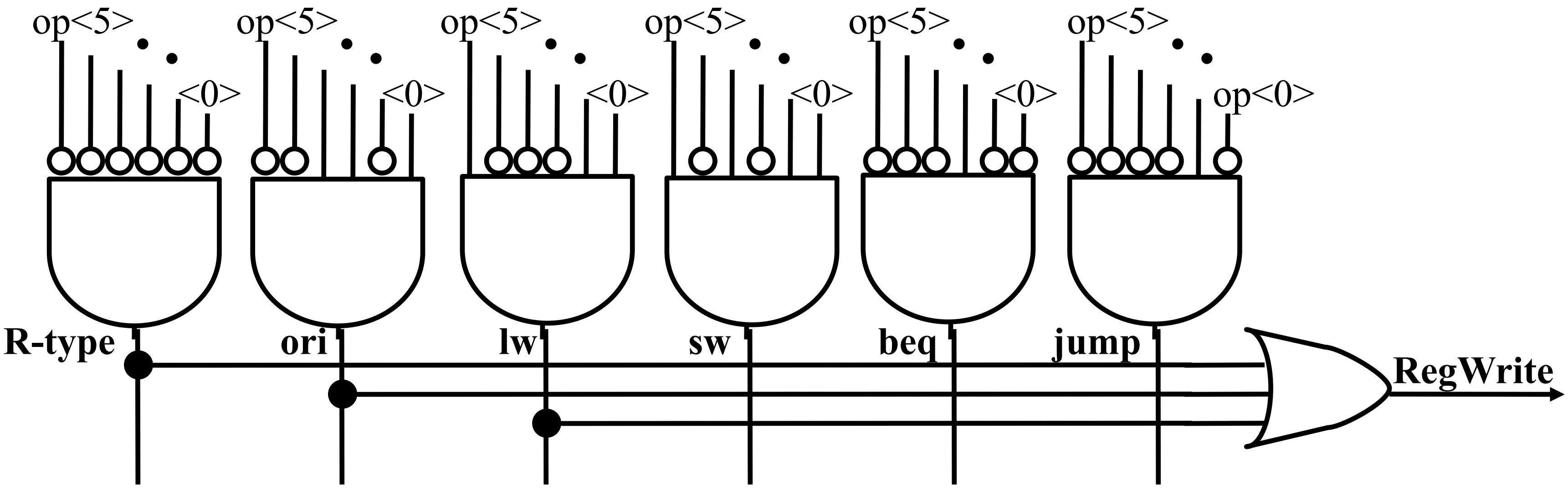
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegWrite	1	1	1	0	0	0

- RegWrite = R-type + ori + lw

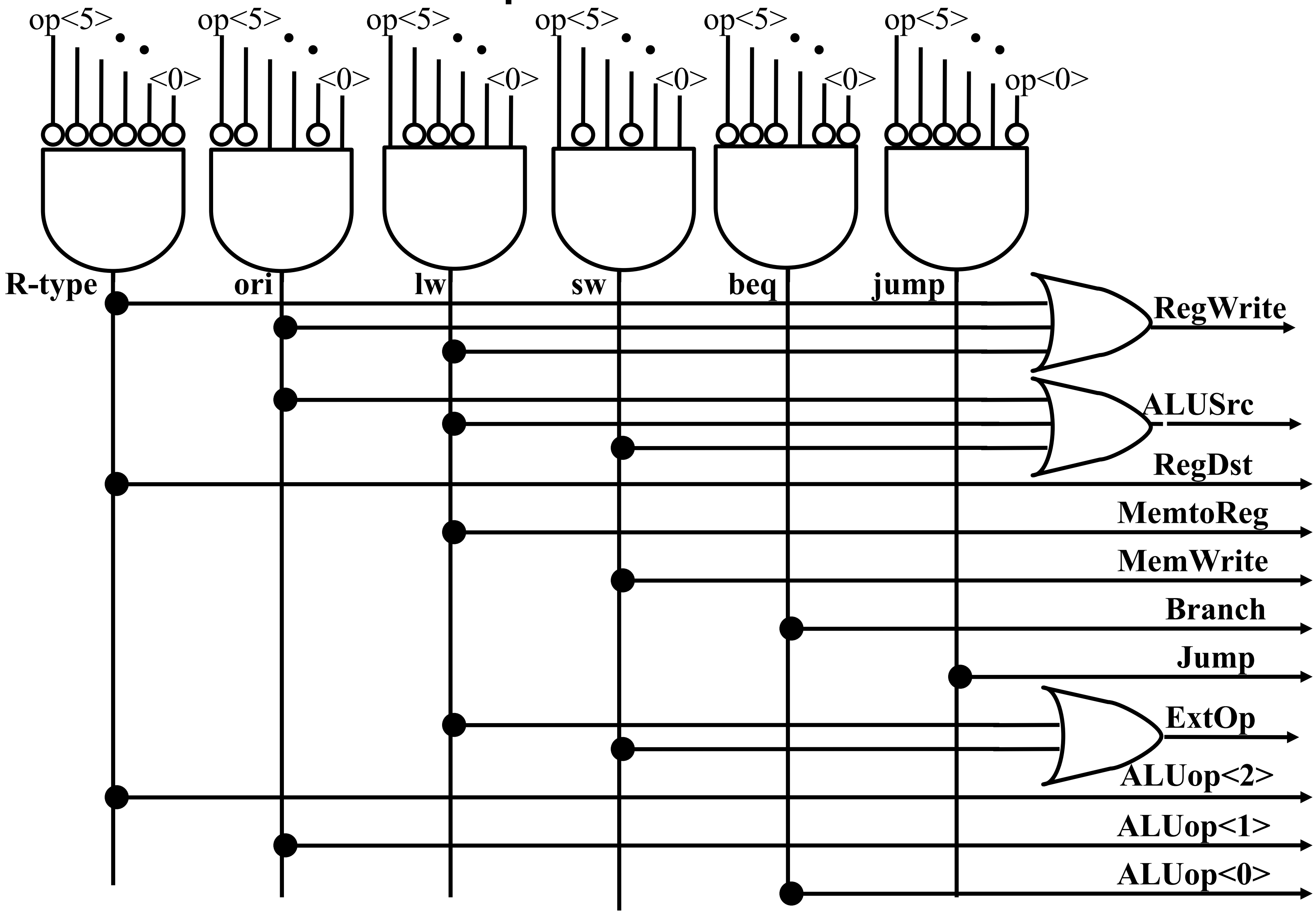
= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)

+ !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)



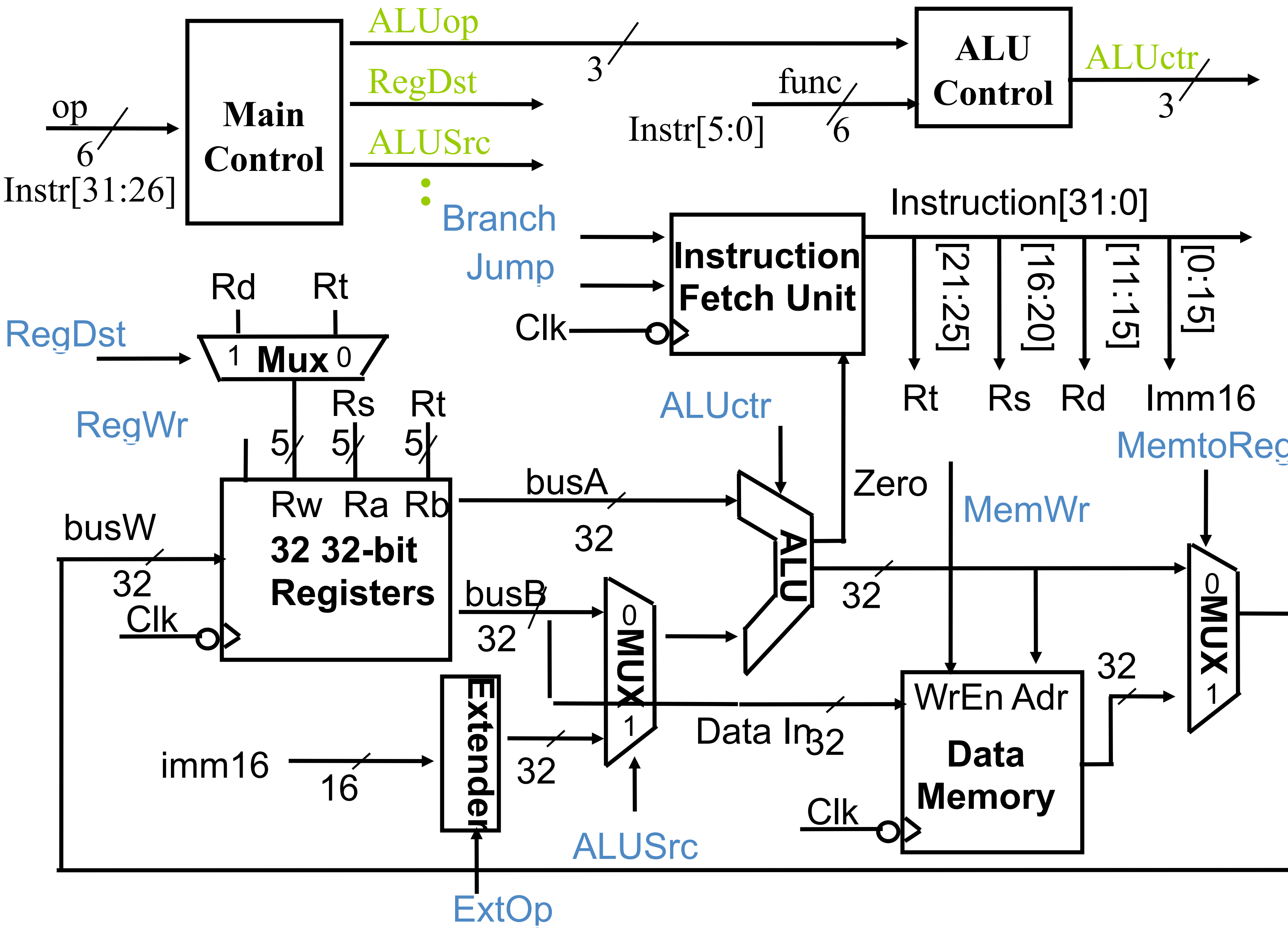
PLA Implementation



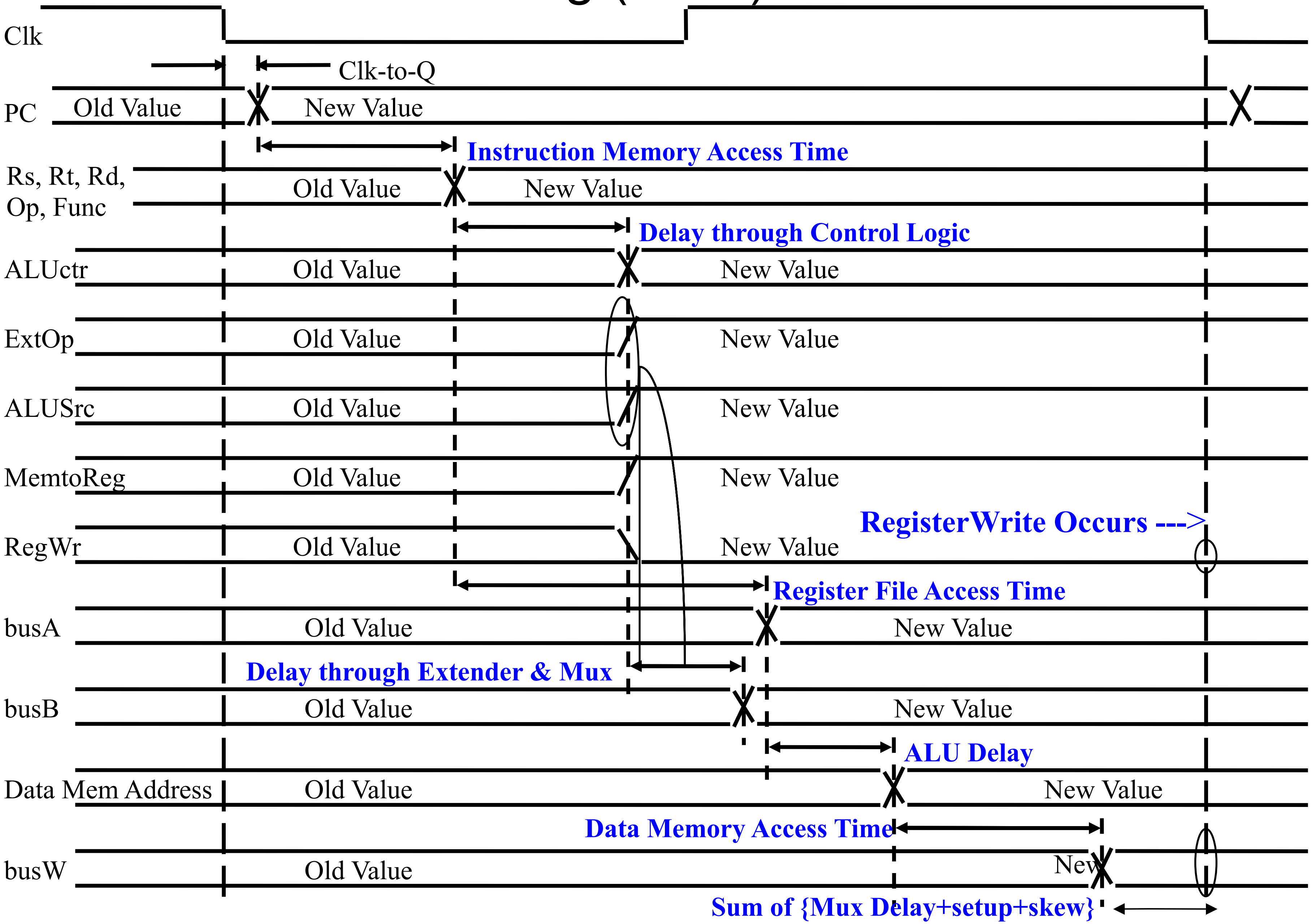
Implementing Control

- Programmable Logic Array (PLA) vs. "Random Logic"
  - Design Changes
    - Validation changes are common
    - PLA is less work to change; area/timing impact is predictable
  - Area
    - Tradeoff depends on complexity of logic (# of gates)
  - Timing and Power
    - Random logic generally better since individual paths can be tuned
- Alternative approach is Read Only Memory (ROM/PROM)
  - Also combinational, but size makes it slow
  - used for microcoded control with more than one state/cycle per instruction

Putting It All Together



Worst Case Timing (Load)



Single Cycle Processor

- Advantages
  - Single cycle per instruction makes logic and clock simple
  - All machines would have a CPI of 1
- Disadvantages
  - Inefficient utilization of memory and functional units since different instructions take different lengths of time
    - Each functional unit is used only once per clock cycle
    - e.g. ALU only computes values a small amount of the time
  - Cycle time is the worst case path → long cycle times!
    - Load instruction
      - **PC CLK-to-Q +**
      - **instruction memory access time +**
      - **register file access time +**
      - **ALU delay +**
      - **data memory access time +**
      - **register file setup time +**
      - **clock skew**
  - All machines would have a CPI of 1, with cycle time set by the longest instruction!

Summary

- Single cycle datapath => CPI=1, CCT => long
- 5 steps to design a processor
  - 1. Analyze instruction set => datapath requirements
  - 2. Select set of datapath components & establish clock methodology
  - 3. Assemble datapath meeting the requirements
  - 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  - 5. Assemble the control logic
- Control is the hard part
- MIPS makes control easier
  - Instructions same size
  - Source registers always in same place
  - Immediates same size, location
  - Operations always on registers/immediates