Computer System Architecture

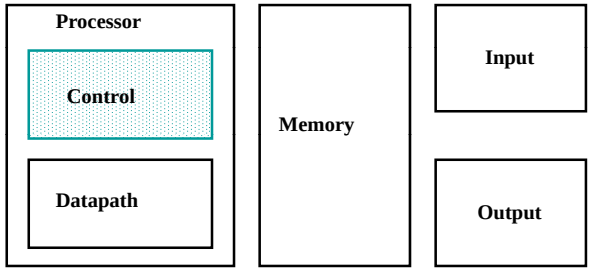
Processor Part II

Chalermek Intanagonwiwat

Slides courtesy of John Hennessy and David Patterson

Where are We Now?

- The Five Classic Components of a Computer



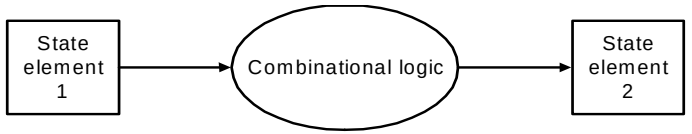
- Today's Topic: Designing the Control for the Single Cycle Datapath

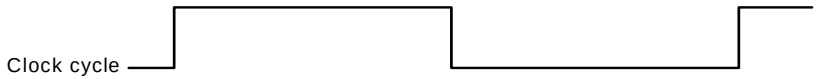
Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce "right answer" right away
  - we use write signals along with clock to determine when to write

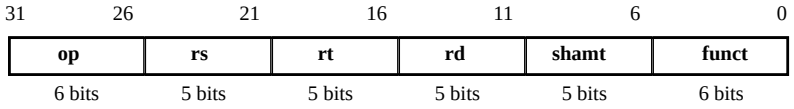
Our Simple Control Structure (cont.)

- Cycle time determined by length of the longest path



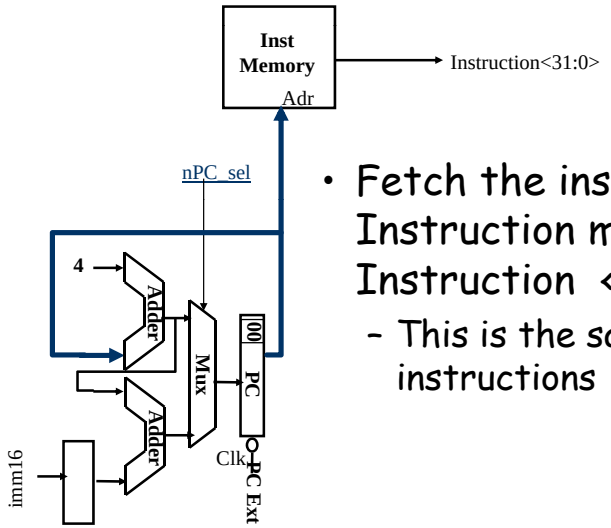


RTL: The Add Instruction



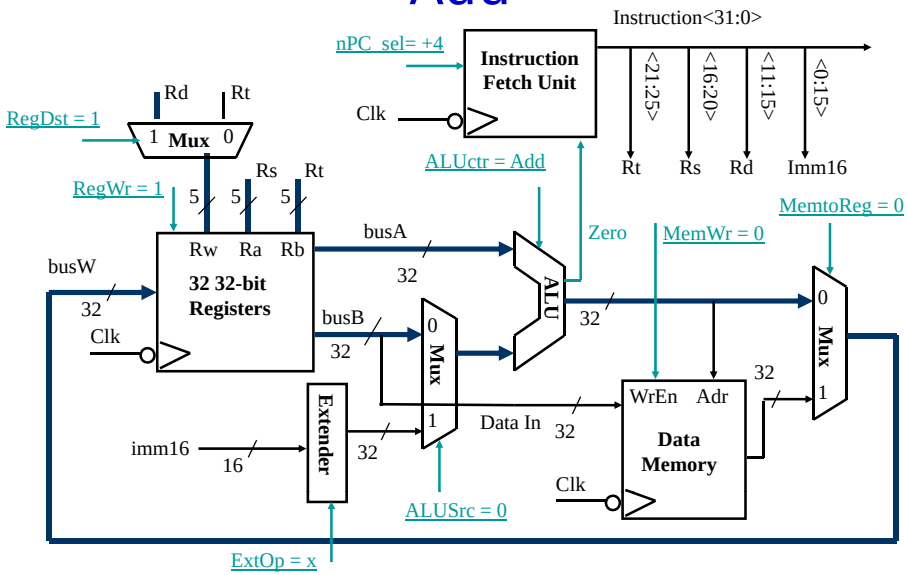
- add rd, rs, rt
  - mem[PC] Fetch the instruction from memory
  - R[rd] <- R[rs] + R[rt] The actual operation
  - PC <- PC + 4 Calculate the next instruction's address

Instruction Fetch Unit at the Beginning of Add

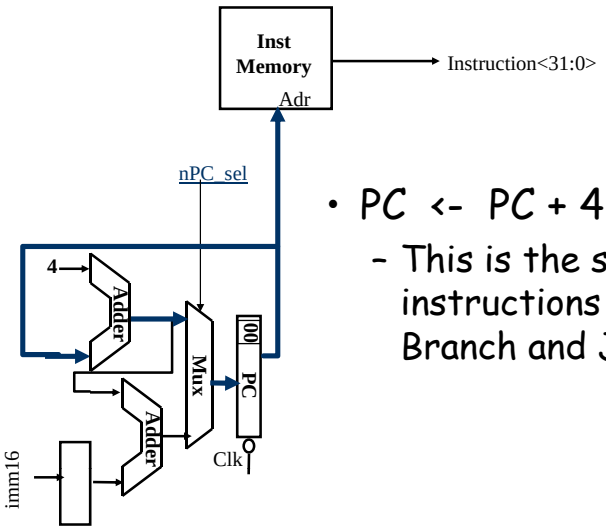


- Fetch the instruction from Instruction memory:
  Instruction <- mem[PC]
  - This is the same for all instructions

The Single Cycle Datapath during Add

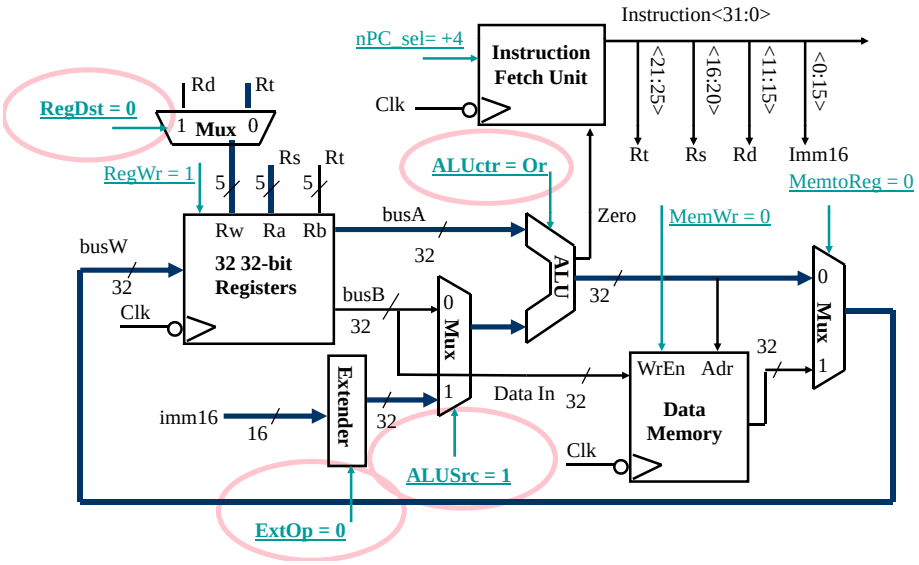


Instruction Fetch Unit at the End of Add

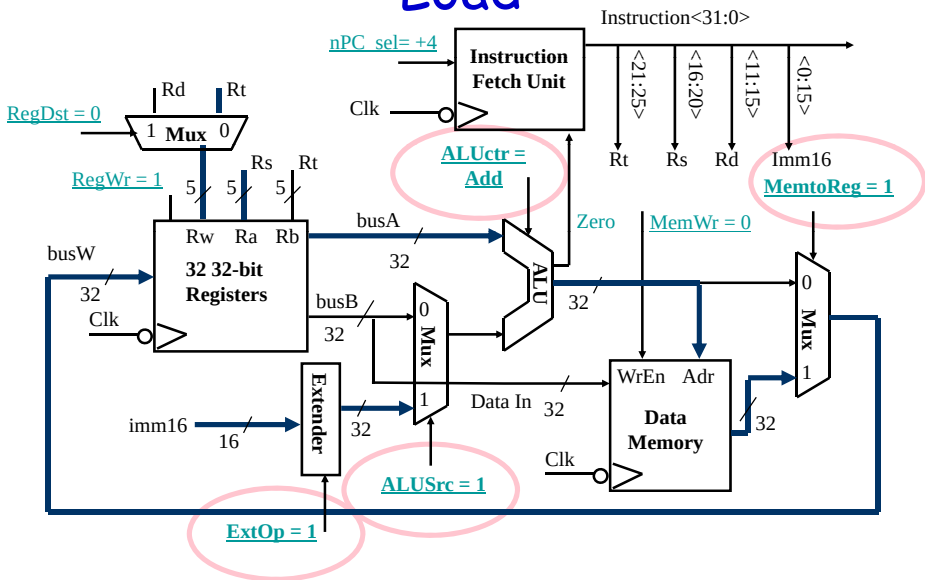


- PC <- PC + 4
  - This is the same for all instructions except: Branch and Jump

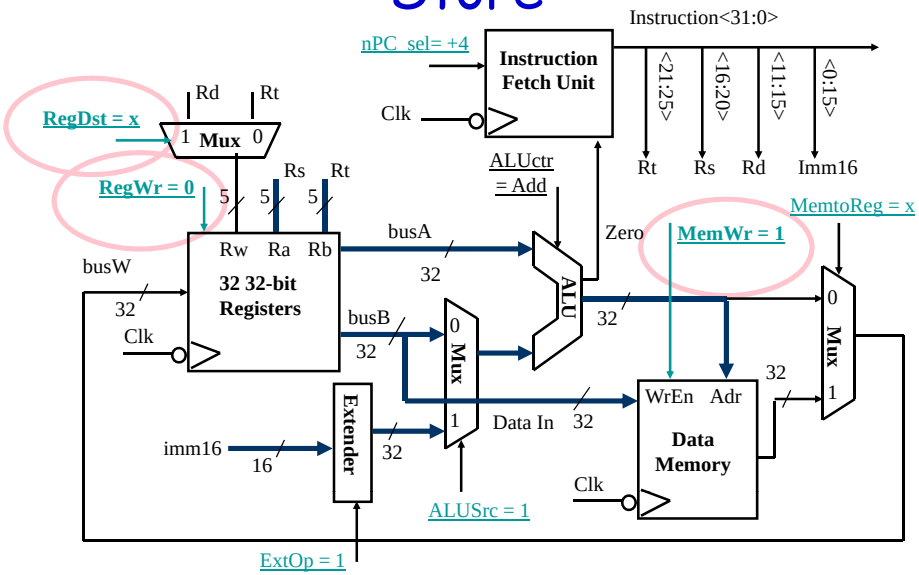
The Single Cycle Datapath during Or Immediate



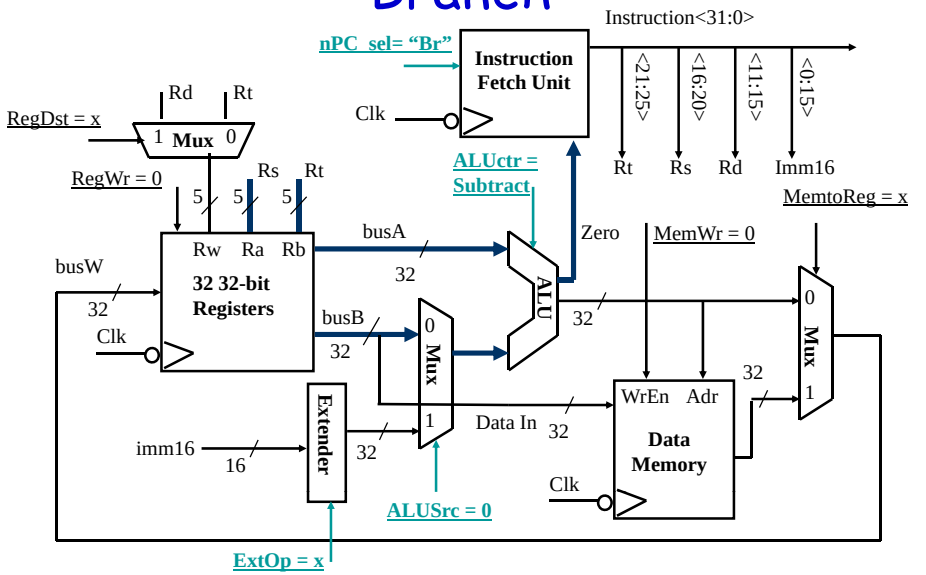
The Single Cycle Datapath during Load



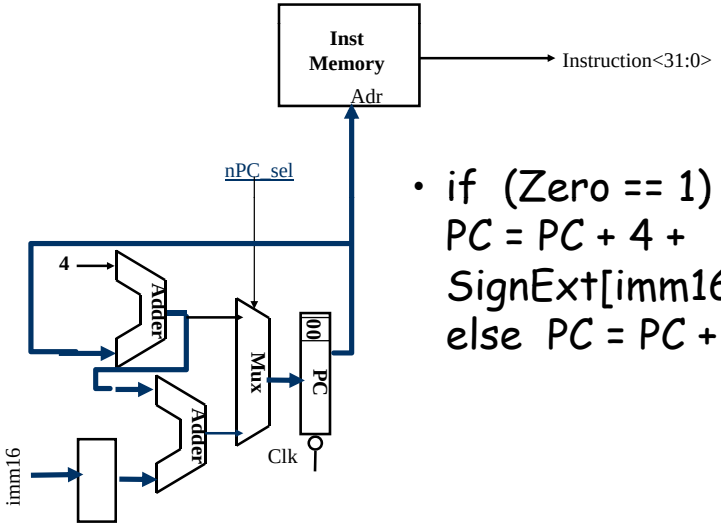
The Single Cycle Datapath during Store



The Single Cycle Datapath during Branch

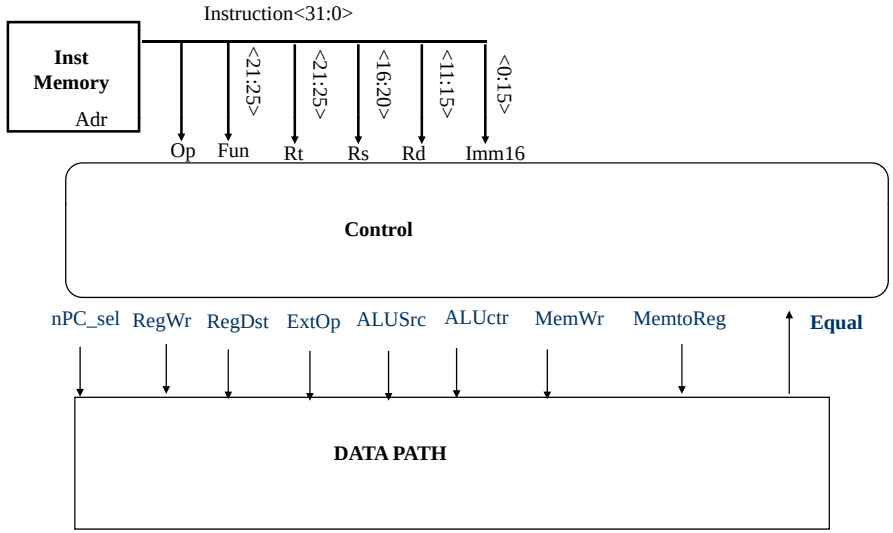


Instruction Fetch Unit at the End of Branch



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

Step 4: Given Datapath: RTL -> Control



A Summary of Control Signals

inst Register Transfer

ADD R[rd] <- R[rs] + R[rt]; PC <- PC + 4

ALUsrc = RegB, ALUctr = "add", RegDst = rd, RegWr, nPC_sel = "+4"

SUB R[rd] <- R[rs] - R[rt]; PC <- PC + 4

ALUsrc = RegB, ALUctr = "sub", RegDst = rd, RegWr, nPC_sel = "+4"

ORi R[rt] <- R[rs] | zero_ext(Imm16); PC <- PC + 4

ALUsrc = Im, Extop = "Z", ALUctr = "or", RegDst = rt, RegWr, nPC_sel = "+4"

A Summary of Control Signals (cont.)

LOAD R[rt] <- MEM[R[rs] + sign_ext(Imm16)];
PC <- PC + 4

ALUsrc = Im, Extop = "Sn", ALUctr = "add", MemtoReg, RegDst = rt, RegWr, nPC_sel = "+4"

STORE MEM[R[rs] + sign_ext(Imm16)] <- R[rs];
PC <- PC + 4

ALUsrc = Im, Extop = "Sn", ALUctr = "add", MemWr, nPC_sel = "+4"

BEQ if (R[rs] == R[rt]) then
PC <- PC + sign_ext(Imm16)] || 00 else PC <- PC + 4

nPC_sel = "Br", ALUctr = "sub"

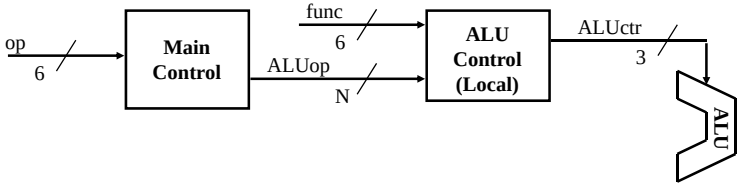
A Summary of the Control Signals (cont.)

See Appendix A

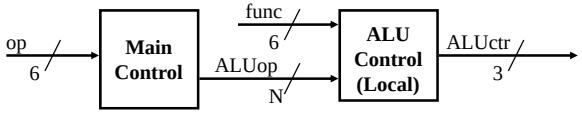
	func	10 0000	10 0010	We Don't Care :-)			
	op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
		add	**sub**	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst		1	1	0	0	x	x	x
ALUSrc		0	0	1	1	1	0	x
MemtoReg		0	0	0	1	x	x	x
RegWrite		1	1	1	1	0	0	0
MemWrite		0	0	0	0	1	0	0
nPCsel		0	0	0	0	0	1	0
Jump		0	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



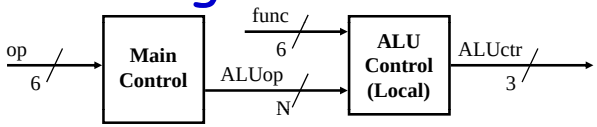
- In this exercise, ALUop has to be 2 bits wide to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract

The Encoding of ALUop (cont.)

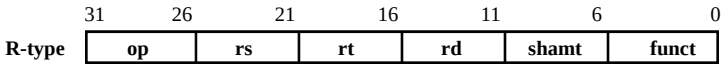
- To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)

	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 "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



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. 286 text):

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

The Truth Table for ALUctr

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			func				ALUctr<2>
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	
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				ALUctr<1>
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	
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

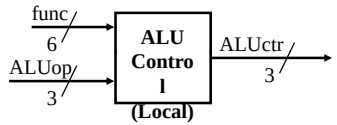
- ALUctr<1> = !ALUop<2> & !ALUop<1> +
  ALUop<2> & !func<2> & !func<0>

The Logic Equation for ALUctr<0>

ALUop			func				
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> & !func<3> & func<2> & !func<1> & func<0>
  + ALUop<2> & func<3> & !func<2> & func<1> & !func<0>

The ALU Control Block



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

Step 5: Logic for each control signal

- nPC_sel <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <= if (OP == "Rtype") then "regB" else "immed"
- ALUctr <= if (OP == "Rtype") then **funct** elseif (OP == ORi) then "OR" elseif (OP == BEQ) then "sub" else "add"

Step 5: Logic for each control signal (cont.)

- ExtOp <= if (OP == ORi) then "zero" else "sign"
- MemWr <= (OP == Store)
- MemtoReg <= (OP == Load)
- RegWr: <= if ((OP == Store) || (OP == BEQ)) then 0 else 1
- RegDst: <= if ((OP == Load) || (OP == ORi)) then 0 else 1

The "Truth Table" for the Main Control

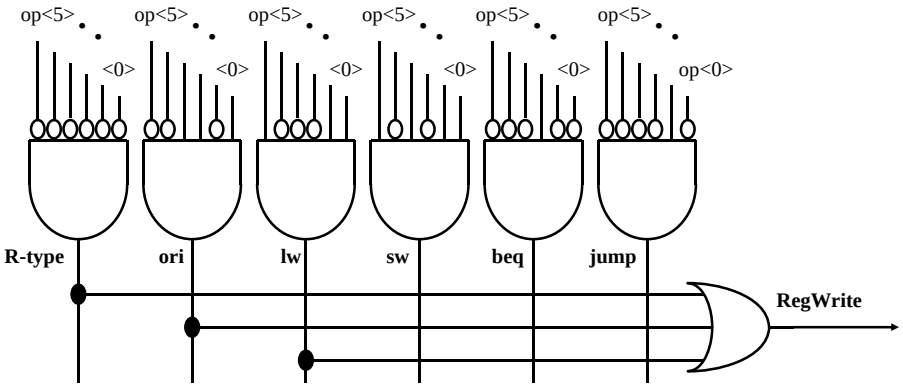
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx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

The "Truth Table" for RegWrite

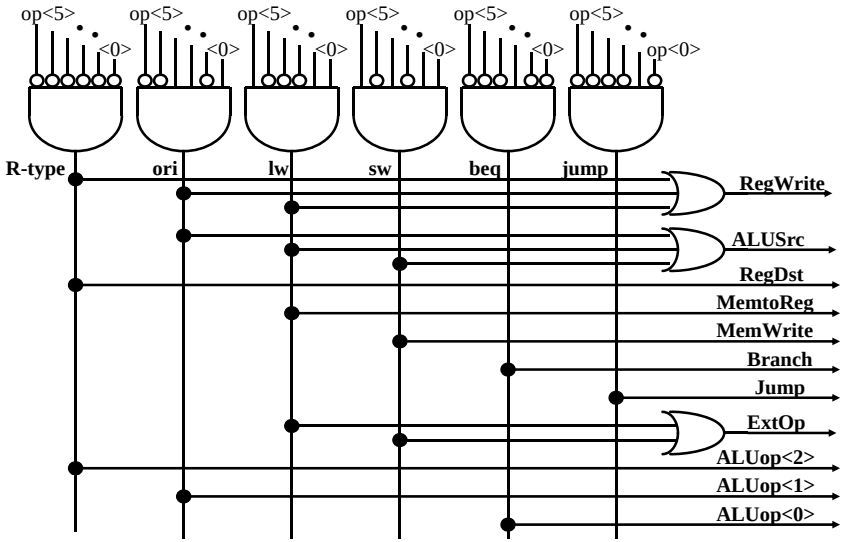
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegWrite	1	1	1	0	0	0

- 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)

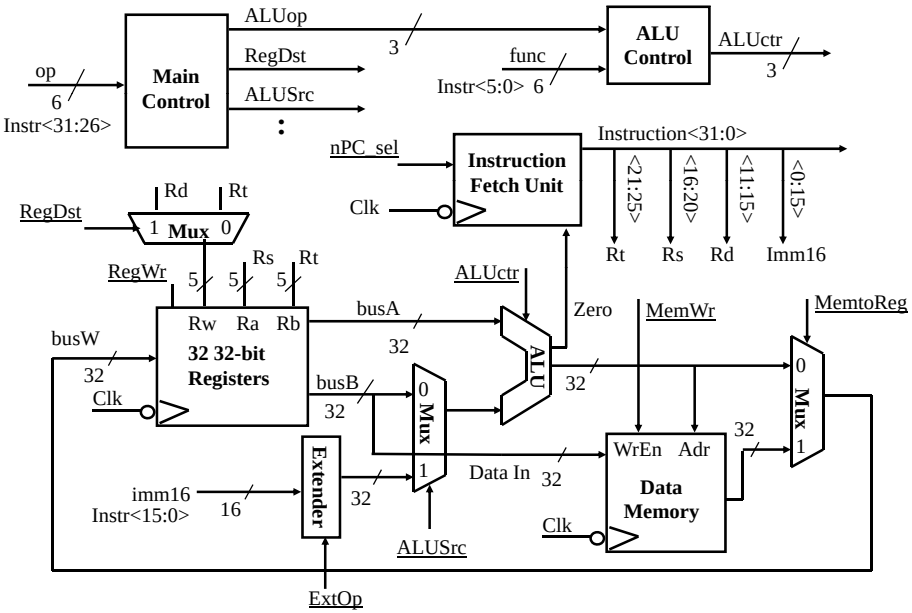
The "Truth Table" for RegWrite (cont.)



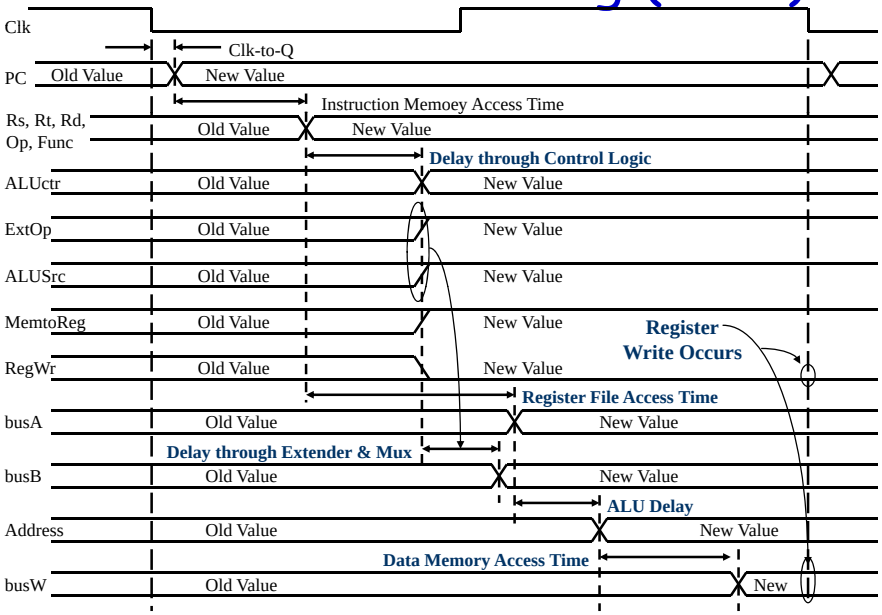
PLA Implementation of the Main Control



A Single Cycle Processor



Worst Case Timing (Load)



Drawback of Single Cycle CPU

- 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 for load is much longer than needed for all other instructions

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**