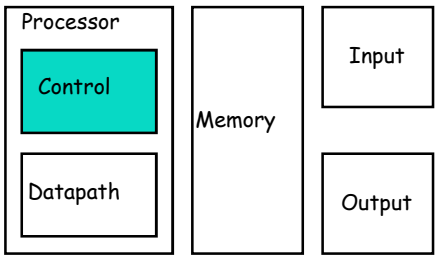
EEM 486: Computer Architecture

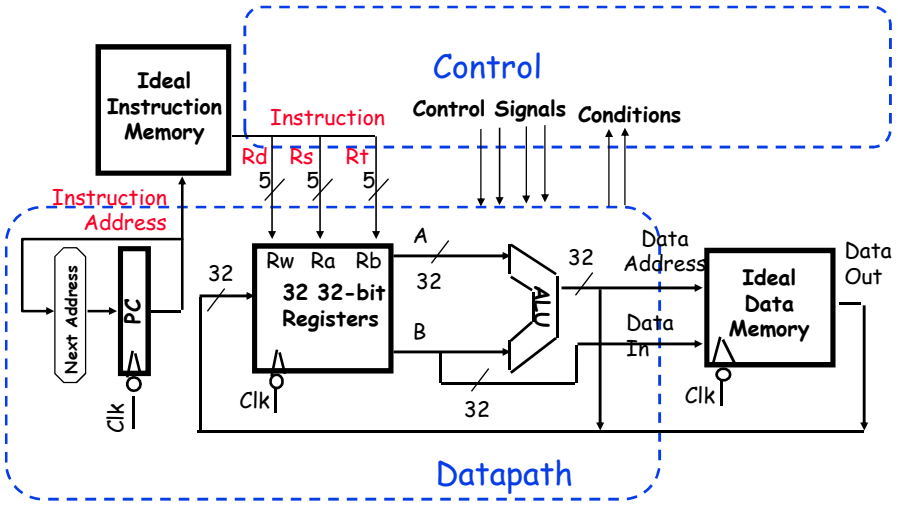
Lecture 3

Designing Single Cycle Control

The Big Picture: Where are We Now?

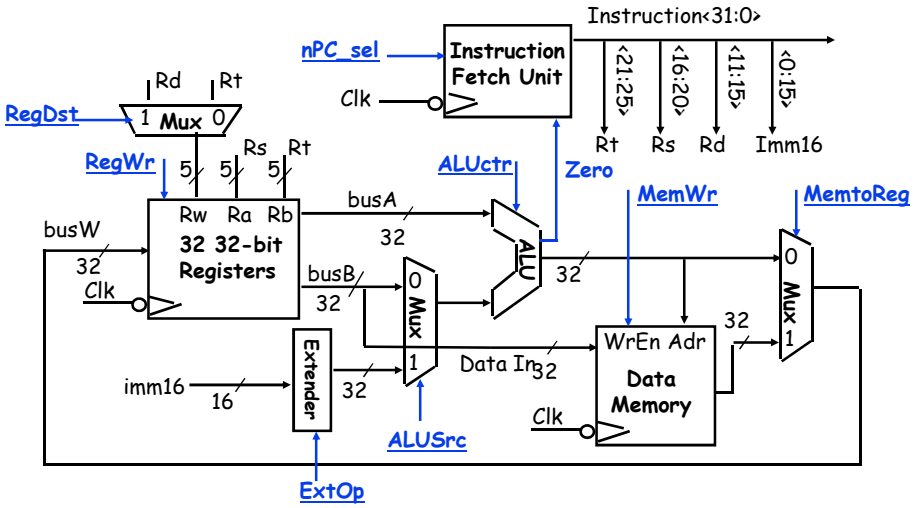


An Abstract View of the Implementation



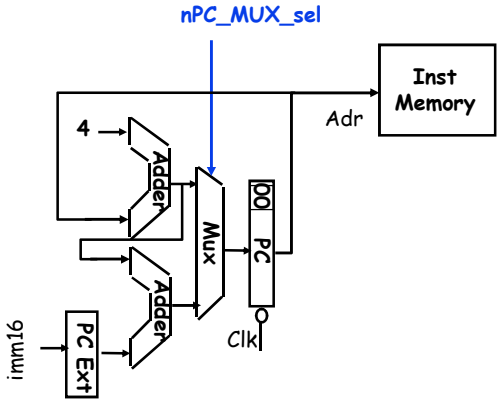
Recap: A Single Cycle Datapath

- We have everything except control signals (underline)



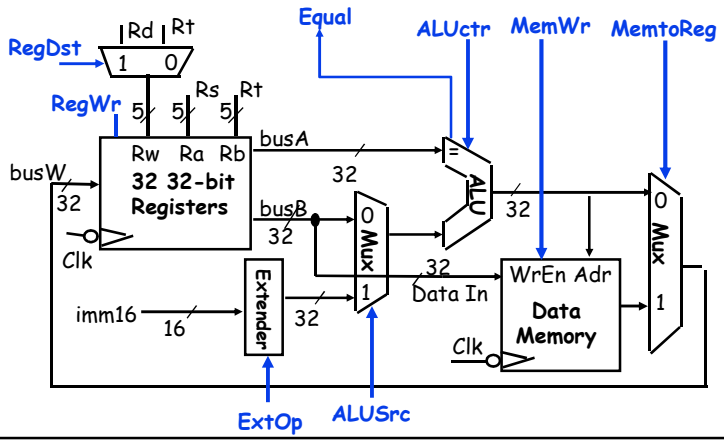
Recap: Meaning of the Control Signals

- nPC_MUX_sel: 0 ⇒ PC <- PC + 4
  1 ⇒ PC <- PC + 4 + SignExt(Im16) || 00

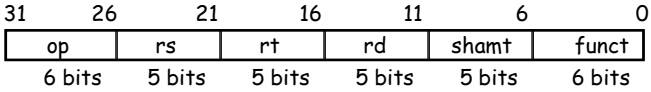


Recap: Meaning of the Control Signals

- ExtOp: "zero", "sign"
- ALUsrc: 0 ⇒ regB; 1 ⇒ immed
- ALUctr: "add", "sub", "or"
- MemWr: 1 ⇒ write memory
- MemtoReg: 0 ⇒ ALU; 1 ⇒ Mem
- RegDst: 0 ⇒ "rt"; 1 ⇒ "rd"
- RegWr: 1 ⇒ write register



RTL: The Add Instruction



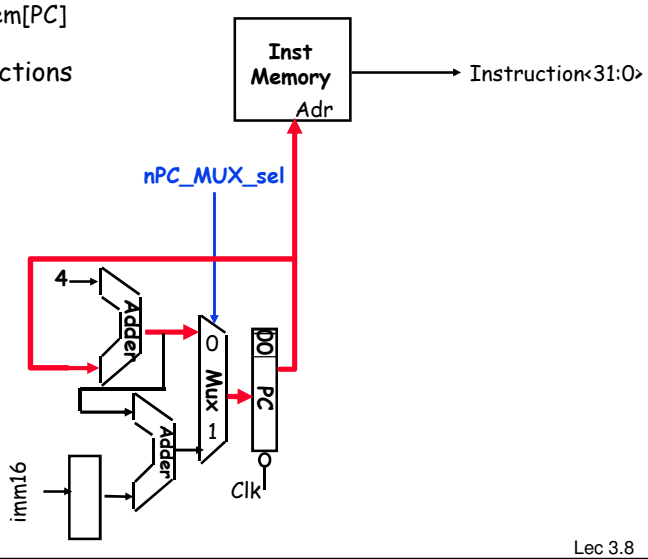
- add rd, rs, rt
  - mem[PC] — Fetch the instruction from memory
  - R[rd] <- R[rs] + R[rt] — The actual operation
  - PC <- PC + 4 — Calculate the next instruction's address

Instruction Fetch Unit at the Beginning of Add

- Fetch the instruction from Instruction memory:

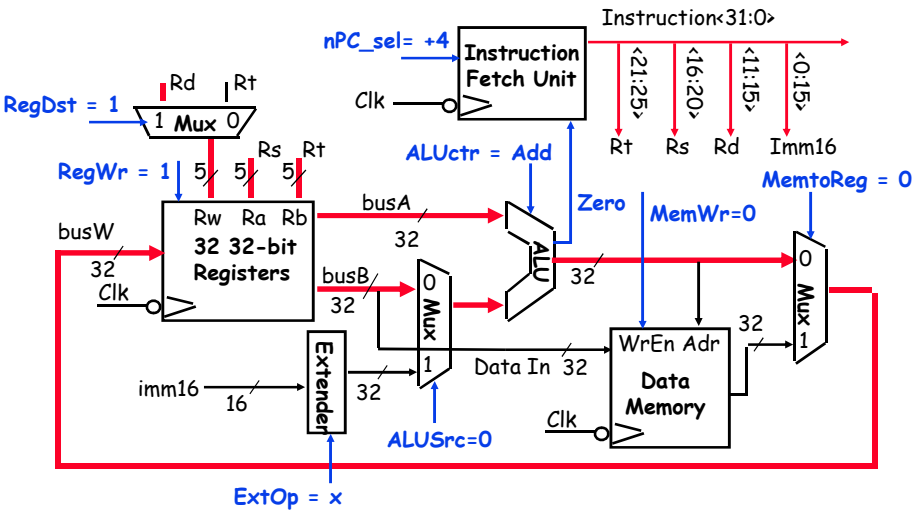
  Instruction <- mem[PC]

- Same for all instructions



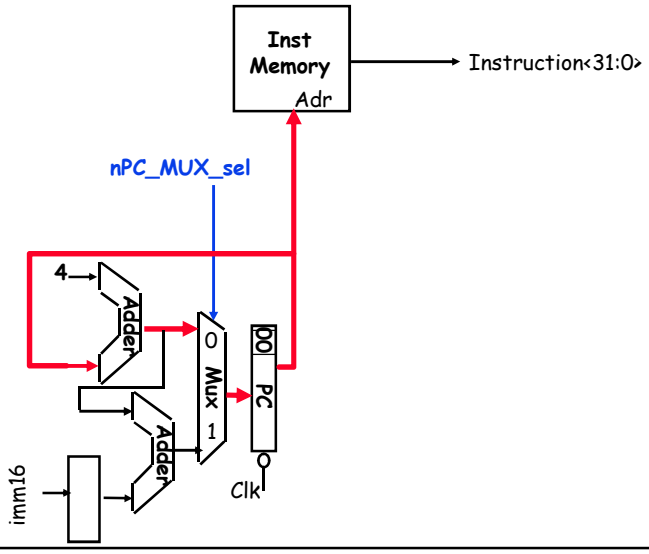
The Single Cycle Datapath During Add

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



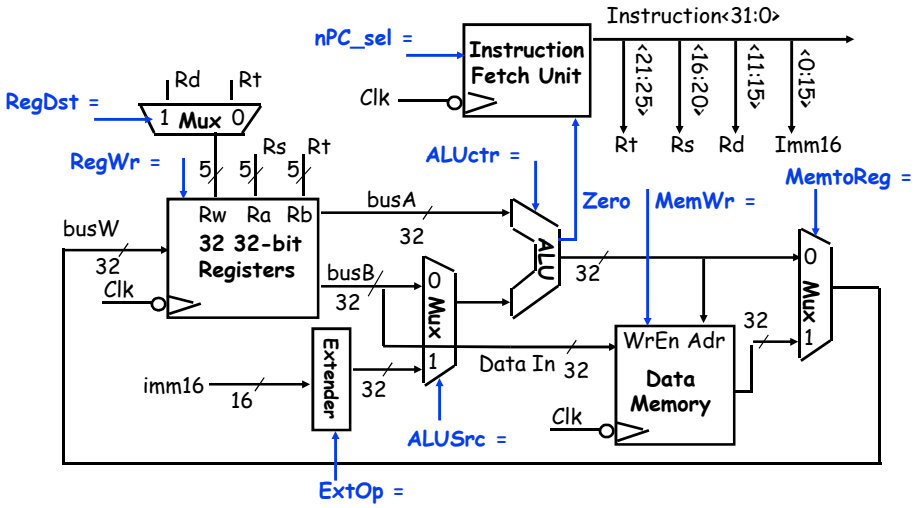
Instruction Fetch Unit at the End of Add

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

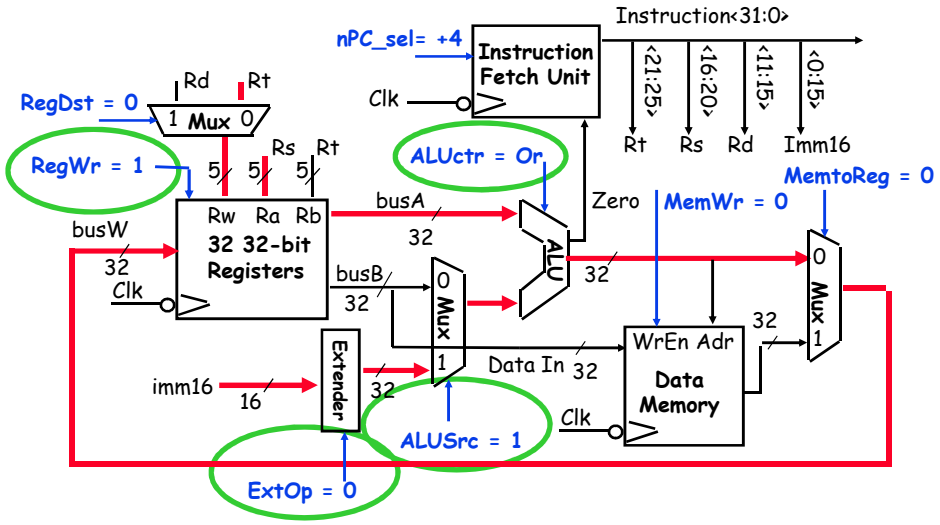


The Single Cycle Datapath During Or Immediate

- R[rt] <- R[rs] or ZeroExt[Imm16]

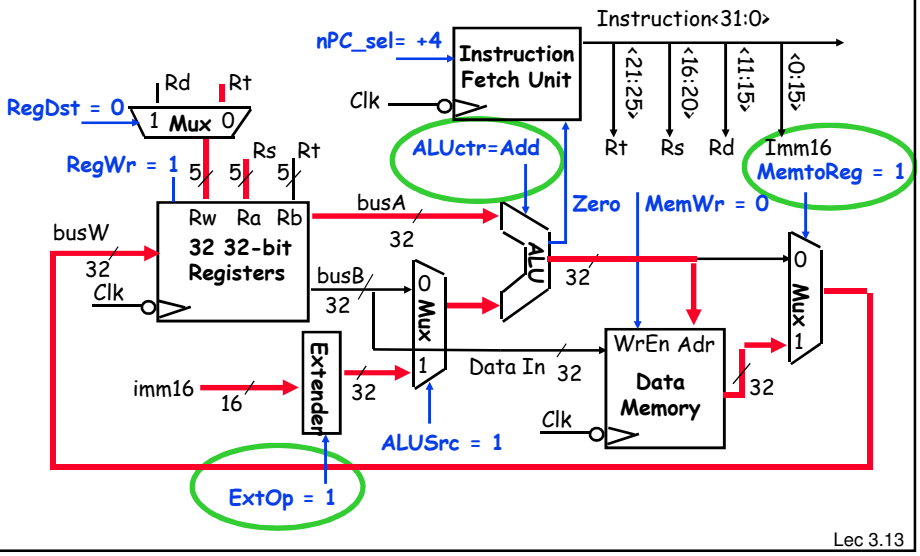


The Single Cycle Datapath During Or Immediate



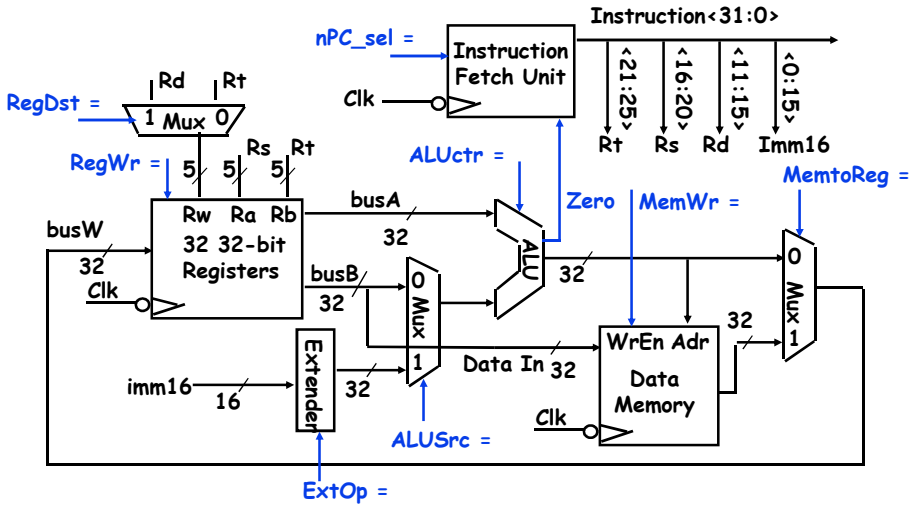
The Single Cycle Datapath During Load

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

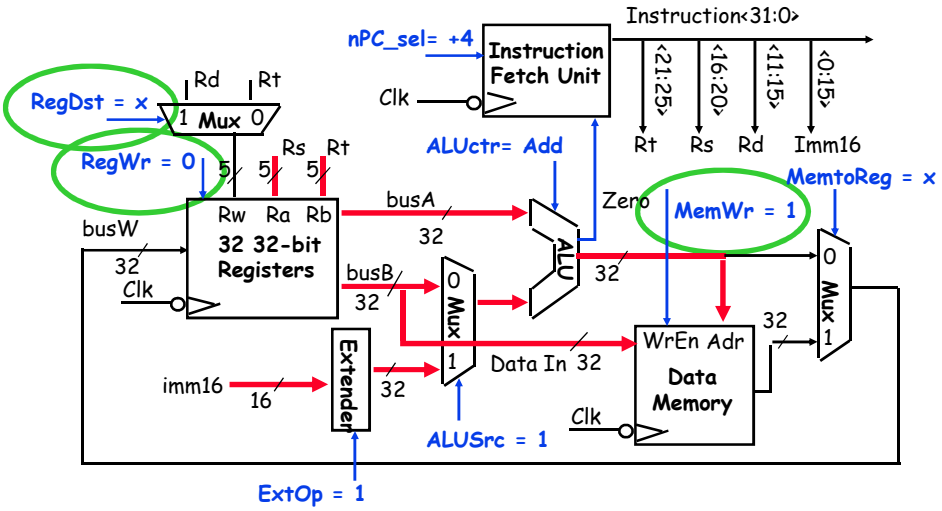


The Single Cycle Datapath During Store

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

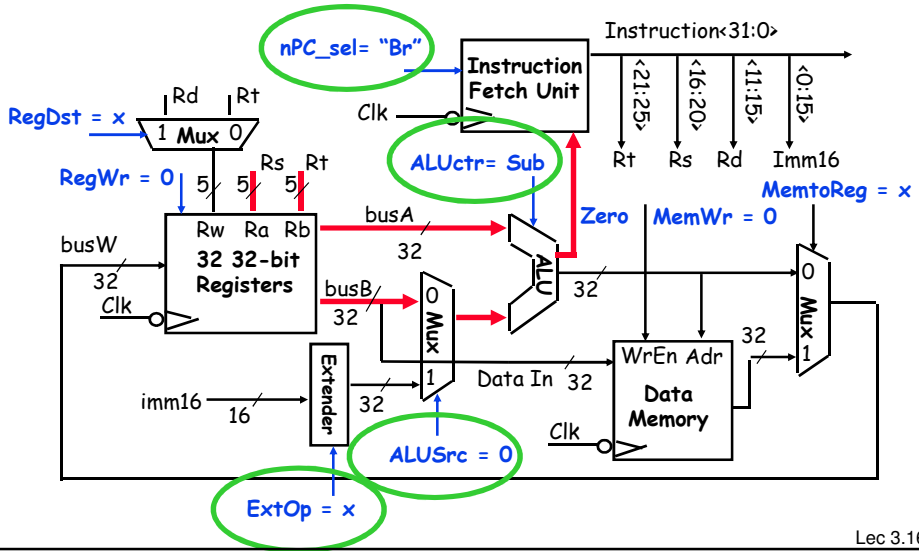


The Single Cycle Datapath During Store

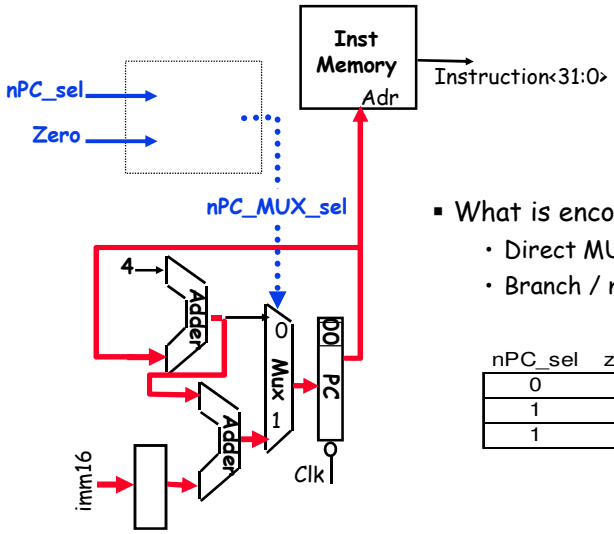


The Single Cycle Datapath During Branch

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



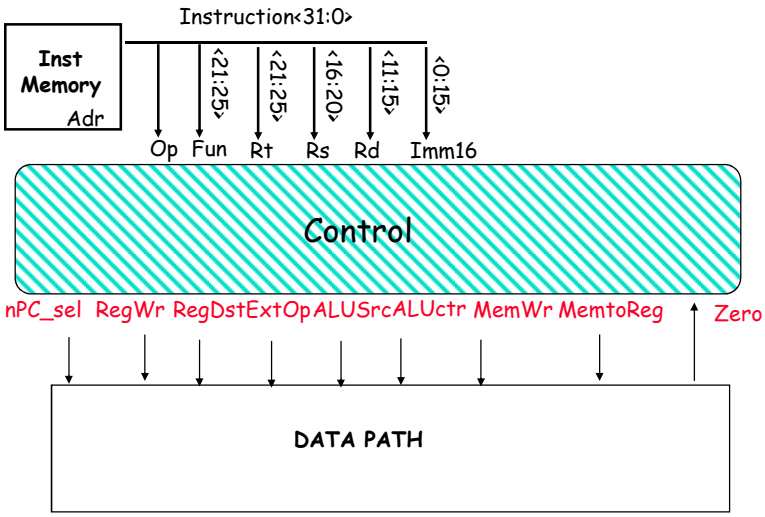
Instruction Fetch Unit at the End of Branch



- What is encoding of nPC_sel?
  - Direct MUX select?
  - Branch / not branch

nPC_sel	zero?	MUX
0	x	0
1	0	0
1	1	1

Step 4: Given Datapath: RTL -> Control



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"	
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[rt];	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"	

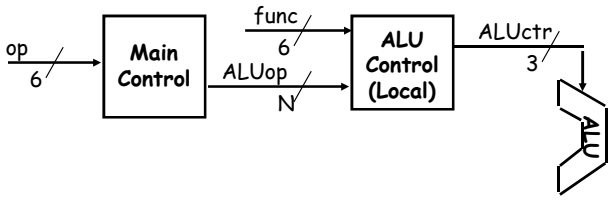
Summary of the Control Signals

See Appendix A

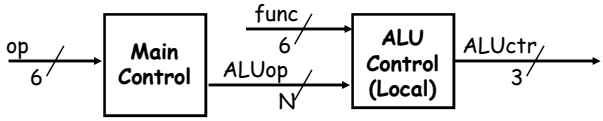
		add	sub	ori	lw	sw	beq
	func	10 0000	10 0010	We Don't Care :-)			
	op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100
RegDst		1	1	0	0	x	x
ALUSrc		0	0	1	1	1	0
MemtoReg		0	0	0	1	x	x
RegWrite		1	1	1	1	0	0
MemWrite		0	0	0	0	1	0
nPCsel		0	0	0	0	0	1
ExtOp		x	x	0	1	1	x
ALUctr<2:0>		Add	Sub	Or	Add	Add	Sub

Concept of Local Decoding

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



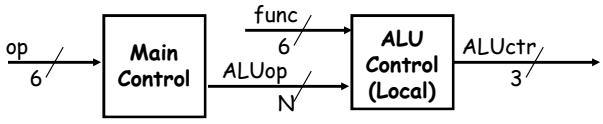
Encoding of ALUop



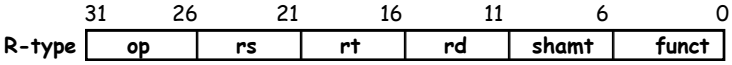
- In this exercise, ALUop has to be 2 bits wide to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract
- To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, (4) Subtract, (5) And, and (6) Xor

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

Decoding of the "func" Field



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



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	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

Truth Table for ALUctr

ALUop (Symbolic)	R-type	ori	lw	sw	beq
ALUop<2:0>	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-thar

ALUop bit<2>	ALUop bit<1>	ALUop bit<0>	func bit<3>	func bit<2>	func bit<1>	func bit<0>	ALU Operation	ALUctr bit<2>	ALUctr bit<1>	ALUctr 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

Logic Equation for ALUctr<2>

ALUop			func				
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 func<3> a don't care

ALUctr<2> = !ALUop<2> & ALUop<0> +

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

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> & ALUop<0> +

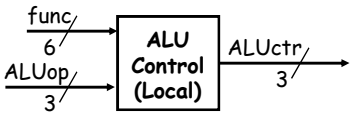
ALUop<2> & !func<2> & !func<0>

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>

ALU Control Block



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

ALUctr<1> = !ALUop<2> & !ALUop<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>

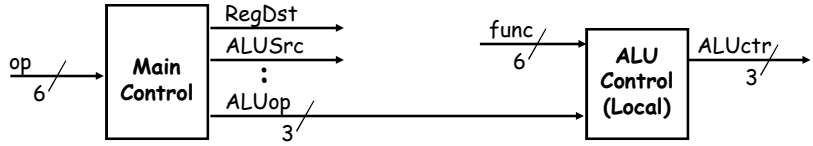
Step 5: Logic For Each Control Signal

- nPC_sel <= if (OP == BEQ) then "Br" else "+4"
- 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"
- ExtOp <= _____
- MemWr <= _____
- MemtoReg <= _____
- RegWr: <= _____
- RegDst: <= _____

Step 5: Logic for Each Control Signal

- nPC_sel <= if (OP == BEQ) then "Br" else "+4"
- 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"
- 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

"Truth Table" for the Main Control



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

"Truth Table" for RegWrite

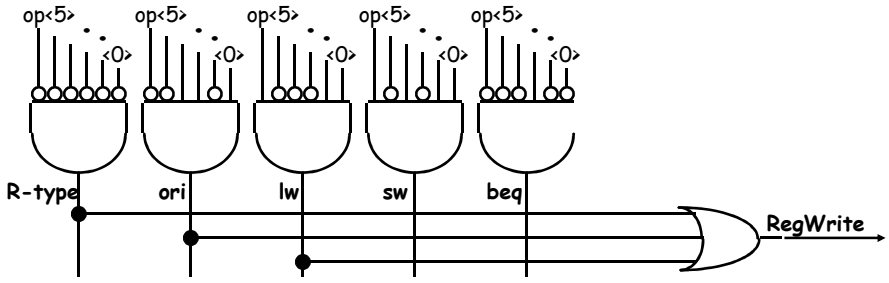
op	00 0000	00 1101	10 0011	10 1011	00 0100
	R-type	ori	lw	sw	beq
RegWrite	1	1	1	0	0

RegWrite = R-type + ori + lw

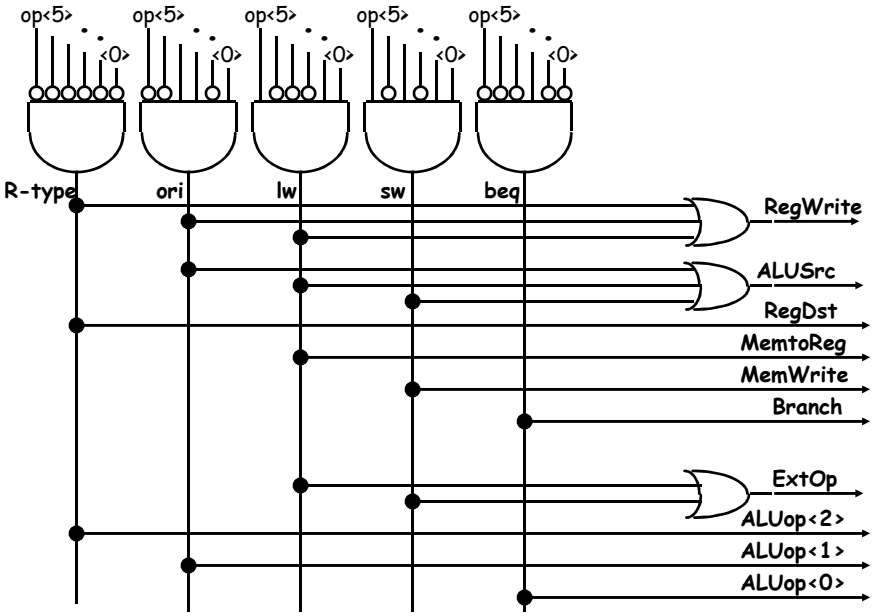
= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)

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

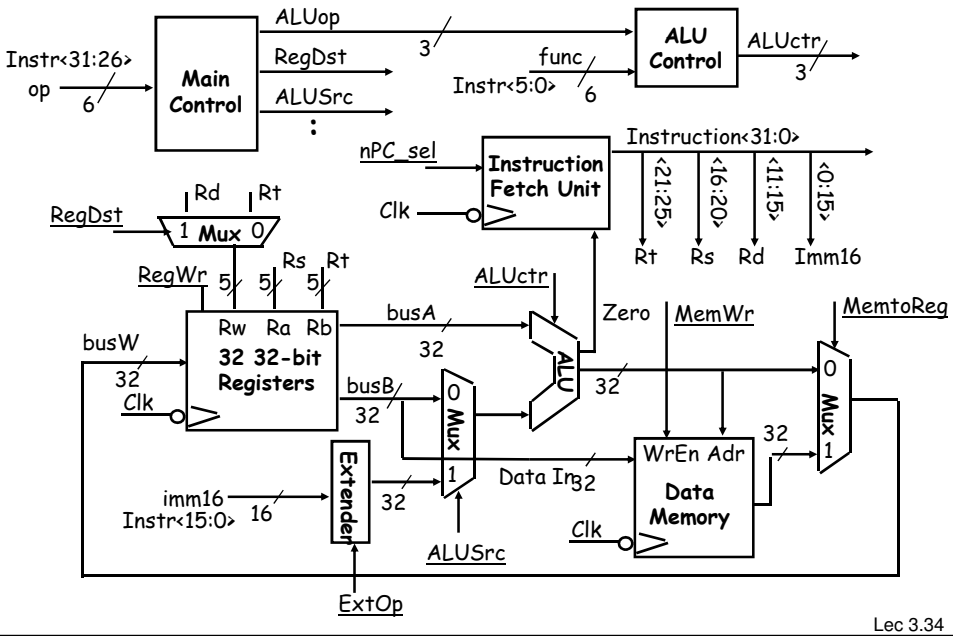
\+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



PLA Implementation of the Main Control

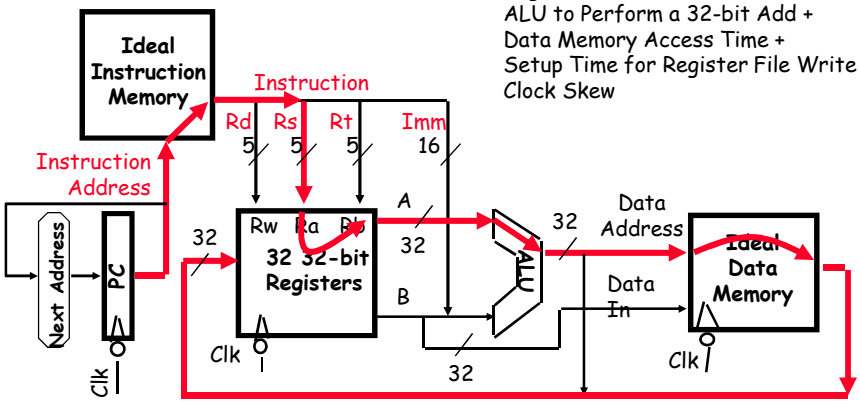


Putting it All Together: A Single Cycle Processor

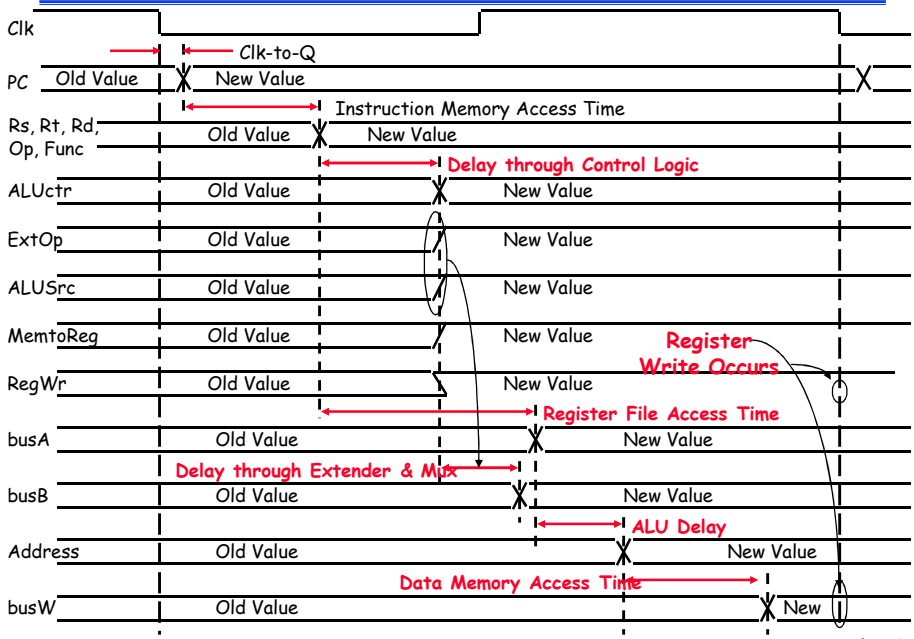


Recap: An Abstract View of the Critical Path (Load)

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



Worst Case Timing (Load)



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 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