L08: Single Cycle Datapath & Control

Adapted from:

CS152: Computer Architecture and Engineering
Dave Patterson (www.cs.berkeley.edu/~pattrsn)

The Main Point: How to design a Processor?

1. **Analyse instruction requirements**
2. **Choose components**
3. **Assemble datapath meeting requirements**
4. **Determine control settings**
5. **Design the control**

Today we consider a single cycle design
- **Easy**
- **Long cycle time**

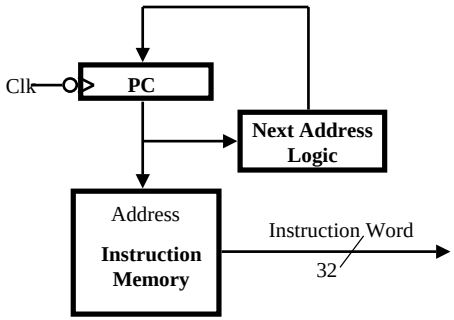
Later we will look at faster implementations
- **Multi-cycle**
- **Pipelined**

3a: Overview of the Instruction Fetch Unit

Common RTL operations:

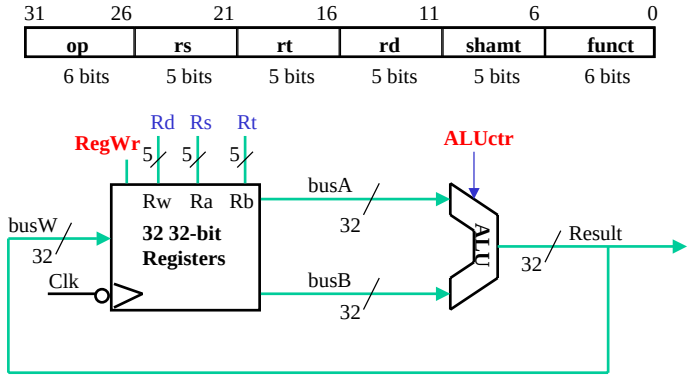
a) At start of cycle, fetch the instruction: mem[PC]

b) At end of cycle, update the program counter:
- **Sequential Code: PC ← PC + 4**
- **Branch and Jump: PC ← "something else"**



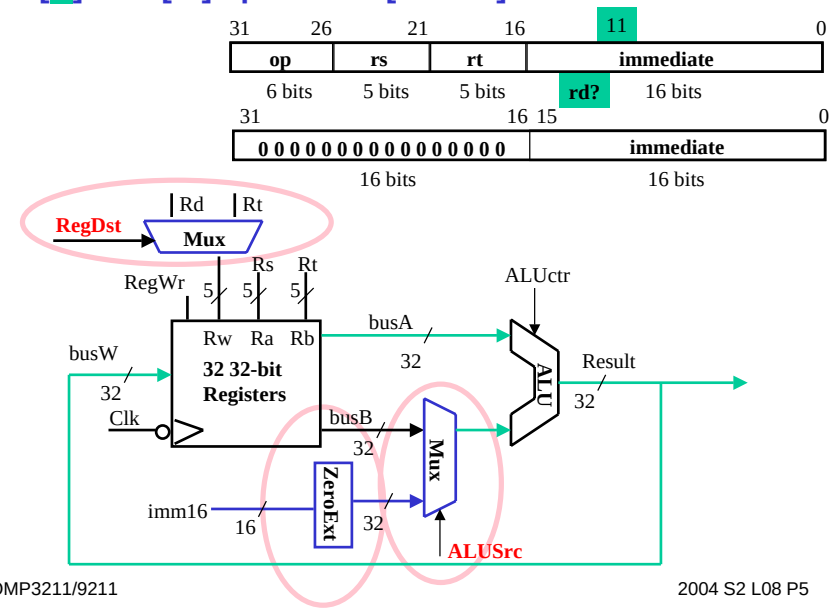
3b: Add & Subtract

R[rd] ← R[rs] op R[rt] **Example: addU rd, rs, rt**
- **Ra, Rb, and Rw come from instruction's rs, rt, and rd fields**
- **ALUctr and RegWr: control logic after decoding the instruction**



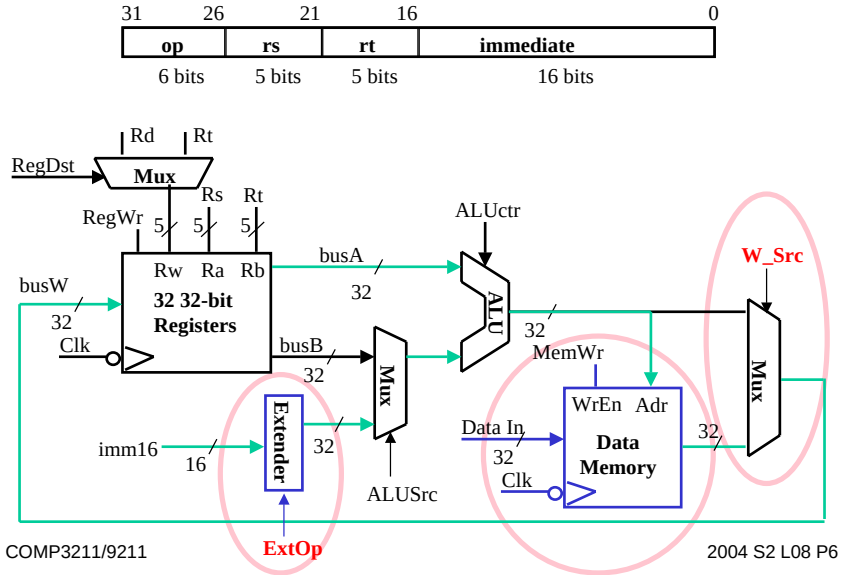
3c: Logical Operations with Immediate

R[rt] ← R[rs] op ZeroExt[imm16]



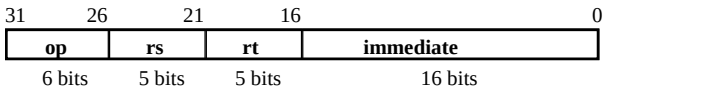
3d: Load Operations

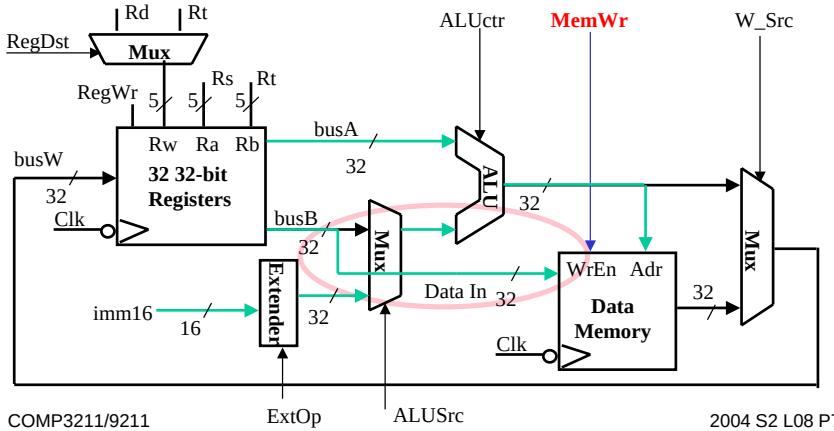
R[rt] ← Mem[R[rs] + SignExt[imm16]] Example: lw rt, rs, imm16



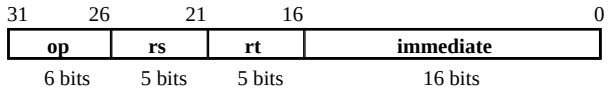
3e: Store Operations

Mem[R[rs] + SignExt[imm16]] ← R[rt] Example: sw rt, rs, imm16





3f: The Branch Instruction



beq rs, rt, imm16

- **mem[PC]** Fetch the instruction from memory
- **Equal ← R[rs] == R[rt]** Calculate the branch condition
- **if (COND eq 0)** Calculate the next instruction's address

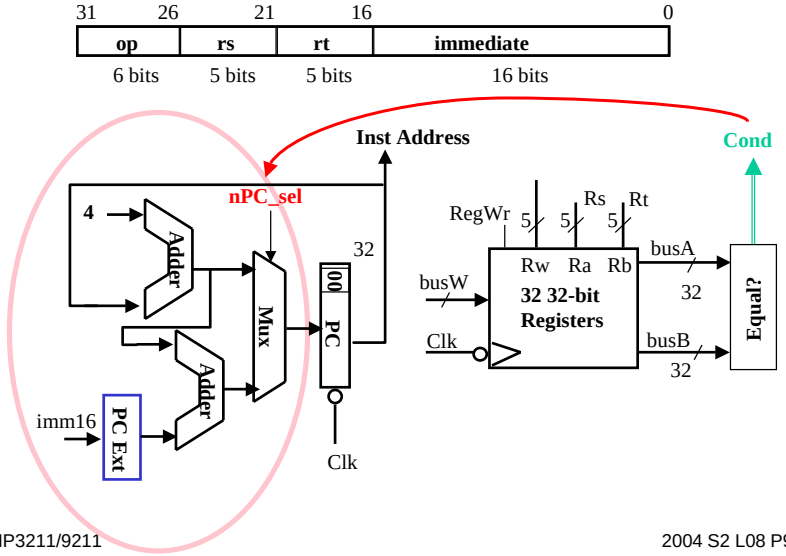
  PC ← PC + 4 + (SignExt(imm16) x 4)

  else

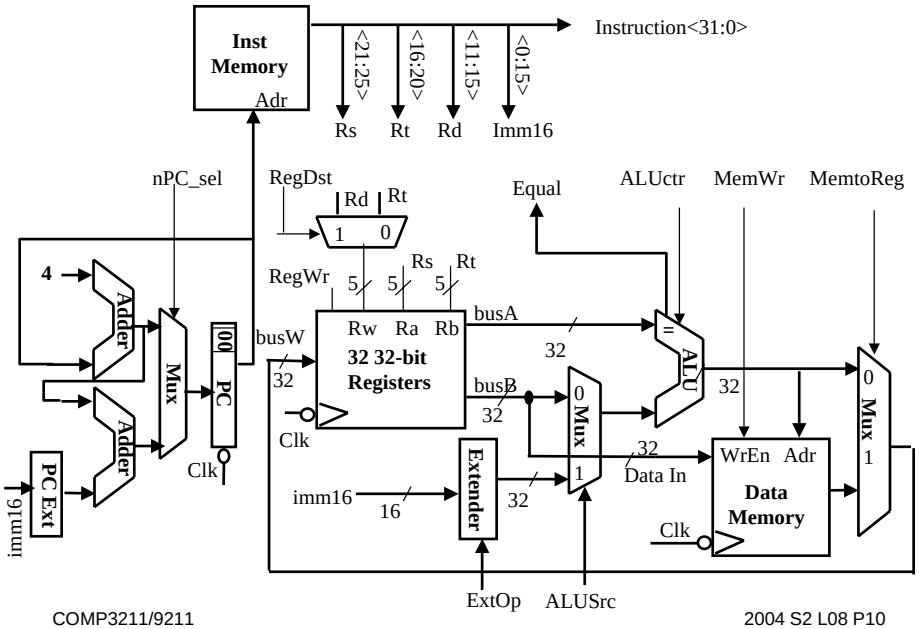
  PC ← PC + 4

Datapath for Branch Operations

beq rs, rt, imm16 **Datapath generates condition (equal)**



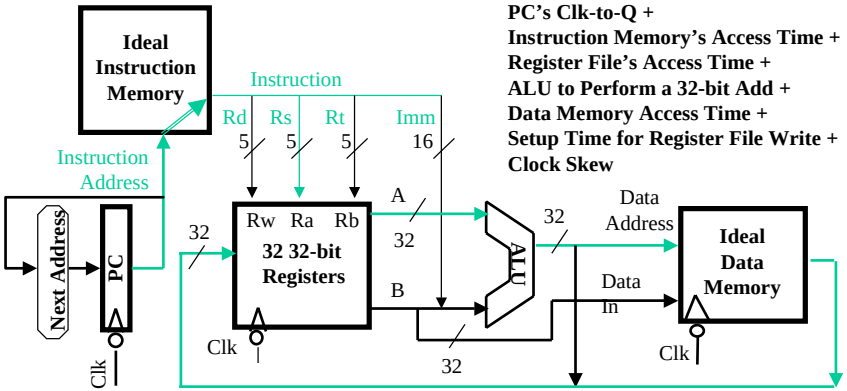
Putting it All Together: A Single Cycle Datapath



So what about the timing???

- **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 a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew



And what about the cycle time???

Given the following timings:

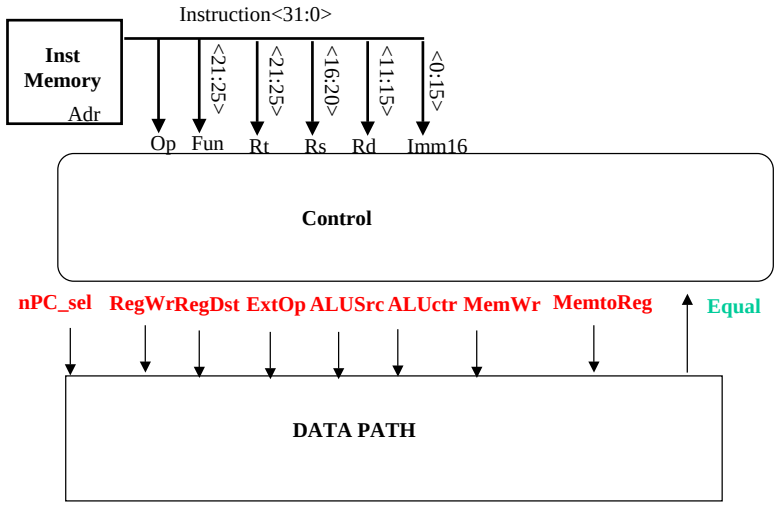
PC: t_{setup} = 0.5ns	Mem: t_{setup} = 1.0ns
PC: t_{pd} = 0.5ns	Mem: t_{access} = 2.0ns
RegFile: t_{setup} = 1.0ns	Add/Sub: t_{comb} = 4.0ns
RegFile: t_{access} = 2.0ns	OR: t_{comb} = 0.5ns

Assume all other datapath & control components have zero delay.

What is the minimum time needed to fetch and execute an ADD instruction? What about ORi, LOAD, STORE, and BEQ?

What is the cycle time of our design?

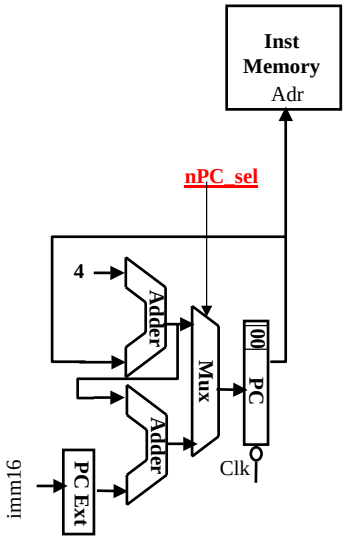
Step 4: Given Datapath: RTL → Control



Meaning of the Control Signals (see 3 slides back)

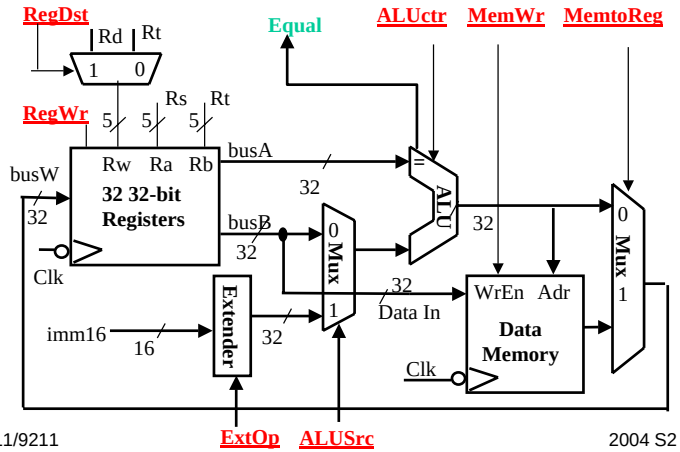
Rs, Rt, Rd and Imed16 hardwired into datapath

nPC_sel: 0 ⇒ PC ← PC + 4; 1 ⇒ PC ← PC + 4 + SignExt(Im16) || 00



Meaning of the Control Signals (see 4 slides back)

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



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[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 = EQUAL, ALUctr = “sub”	

Step 5: Logic for each control signal

nPC_sel ⇐ if (OP == BEQ) then EQUAL else 0

ALUsrc ⇐ if (OP == "000000") then "regB"
else "immed"

ALUctr ⇐ if (OP == "000000") 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)

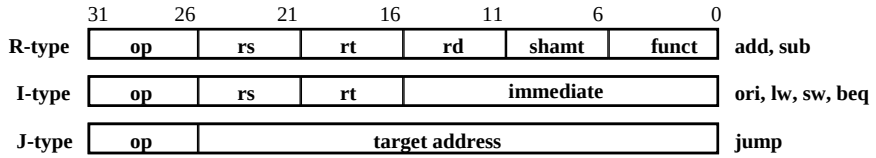
RegWr ⇐ if ((OP == Store) || (OP == BEQ)) then 0
else 1

RegDst ⇐ if ((OP == Load) || (OP == ORi)) then 0
else 1

A Summary of the Control Signals

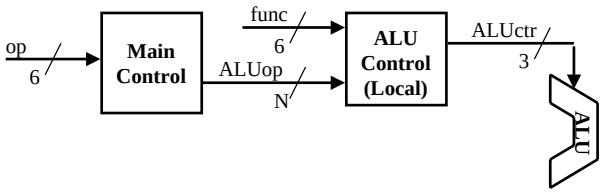
Predefined Codes (see Appendix A)

	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

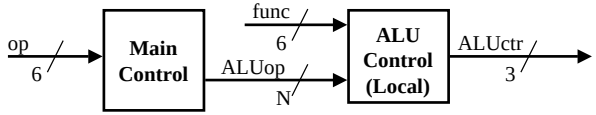


Local Decoding of ALU control signal

	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



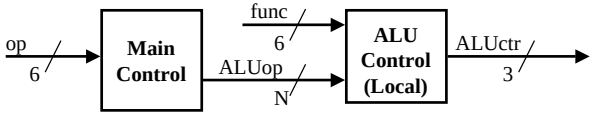
The Encoding of ALUop



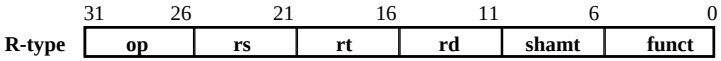
- For the MIPS subset we consider here, ALUop has to be 2 bits wide to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract
- To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) "R-type" instructions
  - "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 "func" Field

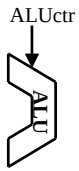


	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx



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

ALU control as defined in Ch 4



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
	"R-type"	Or	Add	Add	Subtract
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-than

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

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

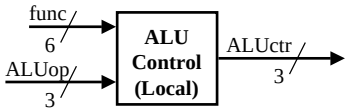
ALUctr<1> = !ALUop<2> & !ALUop<1> +
ALUop<2> & !func<2> & !func<0>

The 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<1> +
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 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"

ExtOp <= if (OP == ORi) then "zero" else "sign"

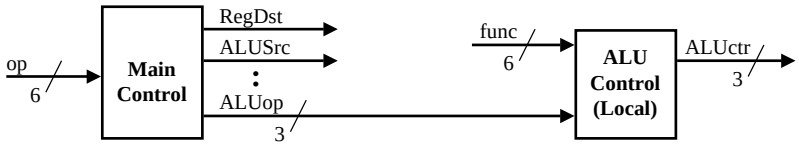
MemWr <= (OP == Store)

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

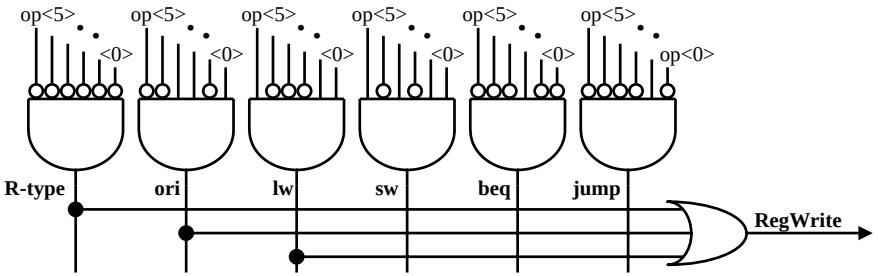
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	0	0	0

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

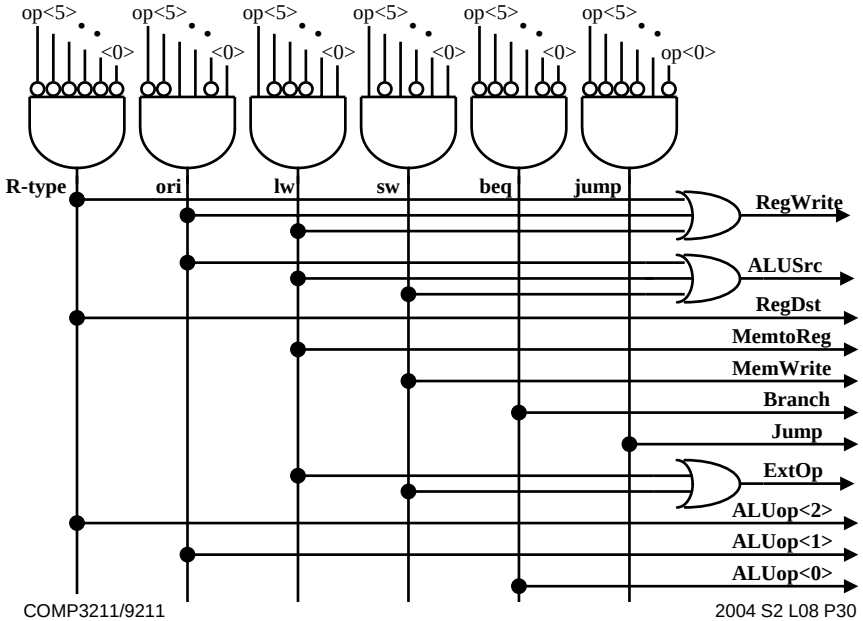
  = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0>(R-type)

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

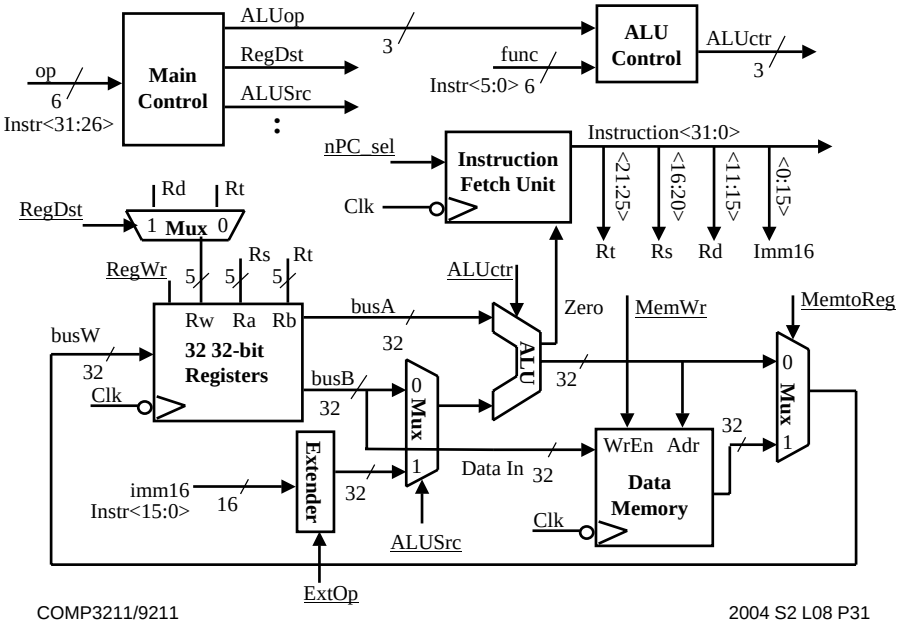
  + op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0>(lw)



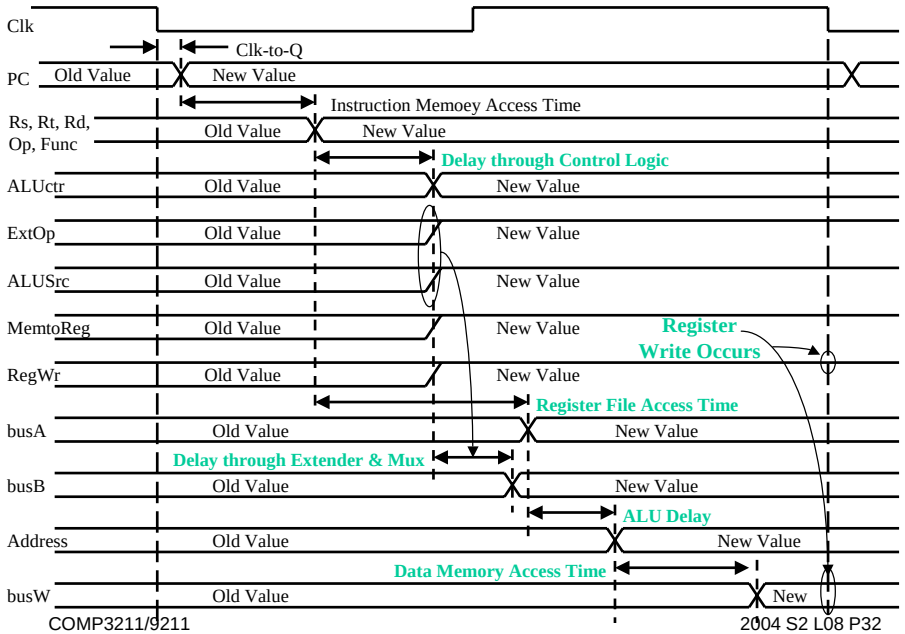
PLA Implementation of the Main Control



Putting it All Together: A Single Cycle Processor



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**