CS152
Computer Architecture and Engineering
Lecture 8

Single Cycle Datapath continued: Single Cycle Control

February 24, 2003

John Kubiatowicz (www.cs.berkeley.edu/~kubitron)

lecture slides: http://inst.eecs.berkeley.edu/~cs152/

Recap: Summary from last time

° **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**

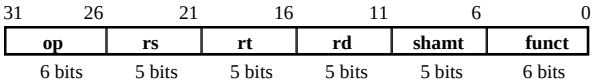
° **MIPS makes it easier**
- **Instructions same size**
- **Source registers always in same place**
- **Immediates same size, location**
- **Operations always on registers/immediates**

° **Single cycle datapath ⇒ CPI=1, Cycle Time = long!**

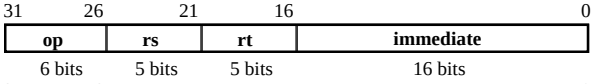
Recap: The MIPS Instruction Formats

° **All MIPS instructions are 32 bits long. The three instruction formats:**

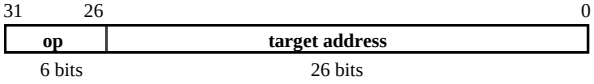
- **R-type**



- **I-type**



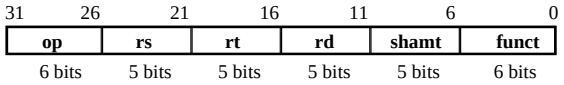
- **J-type**



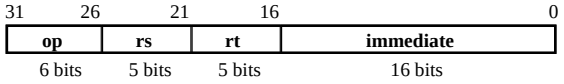
° **The different fields are:**
- **op: operation of the instruction**
- **rs, rt, rd: the source and destination registers specifier**
- **shamt: shift amount**
- **funct: selects the variant of the operation in the "op" field**
- **address / immediate: address offset or immediate value**
- **target address: target address of the jump instruction**

Recap: The MIPS-lite Subset (less filling?)

° **ADD and subtract**
- **add rd, rs, rt**
- **sub rd, rs, rt**



° **OR Imm:**
- **ori rt, rs, imm16**

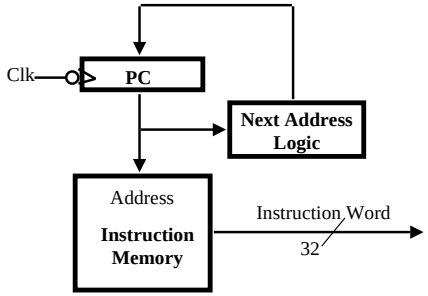


° **LOAD and STORE**
- **lw rt, rs, imm16**
- **sw rt, rs, imm16**

° **BRANCH:**
- **beq rs, rt, imm16**

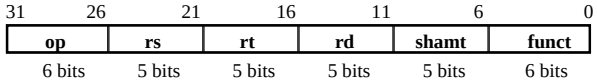
Recap: Overview of the Instruction Fetch Unit

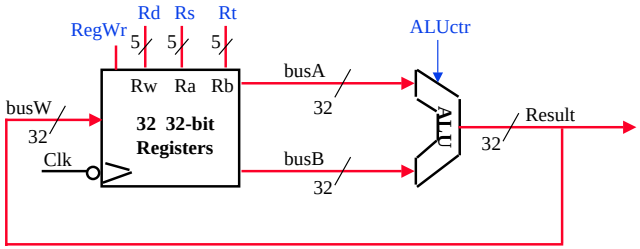
° The common RTL operations
  • Fetch the Instruction: mem[PC]
  • Update the program counter:
    - Sequential Code: PC <- PC + 4
    - Branch and Jump: PC <- "something else"



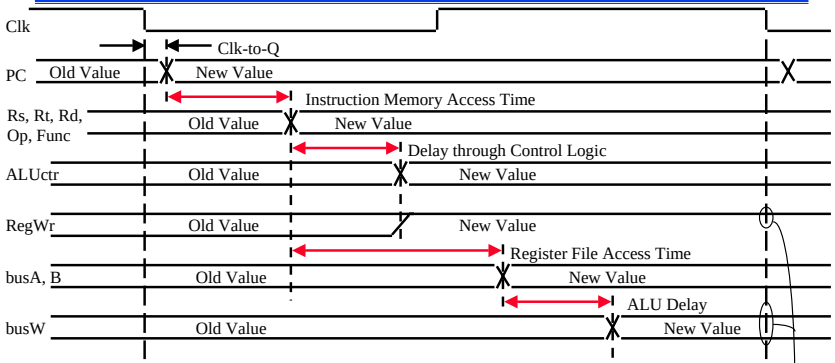
Recap: (3a) Add & Subtract

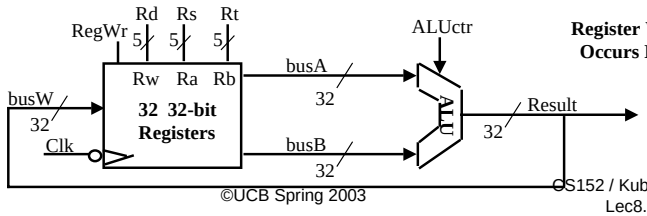
° R[rd] <- R[rs] op R[rt] Example: addU rd, rs, rt
  • Ra, Rb, and Rw come from instruction's rs, rt, and rd fields
  • ALUctr and RegWr: control logic after decoding the instruction





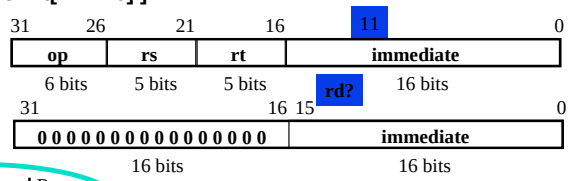
Recap: Register-Register Timing: One complete cycle

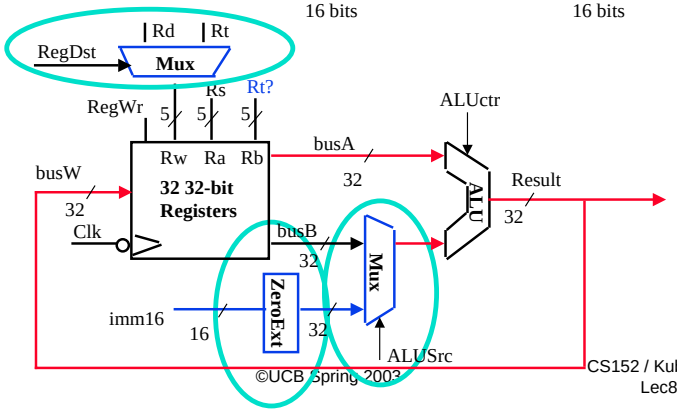




Recap: (3b) Logical Operations with Immediate

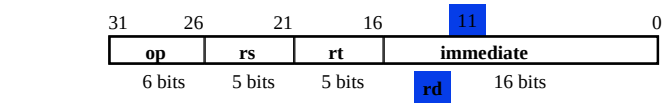
° R[rt] <- R[rs] op ZeroExt[imm16]]

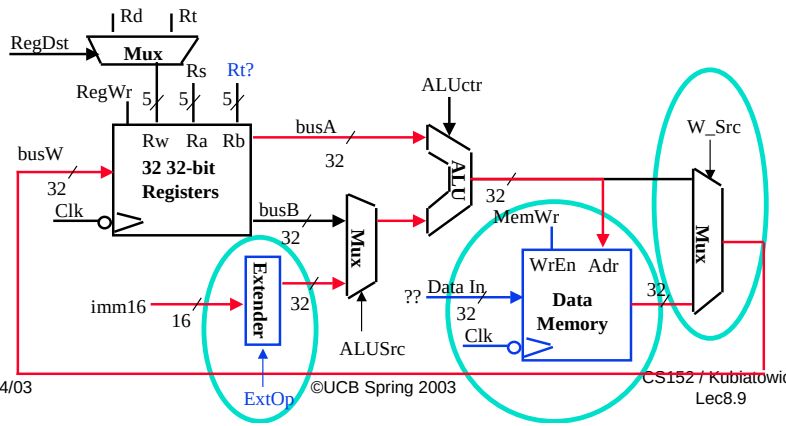




Recap: (3c) Load Operations

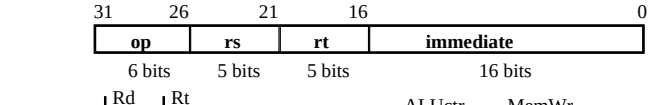
° R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: lw rt, rs, imm16

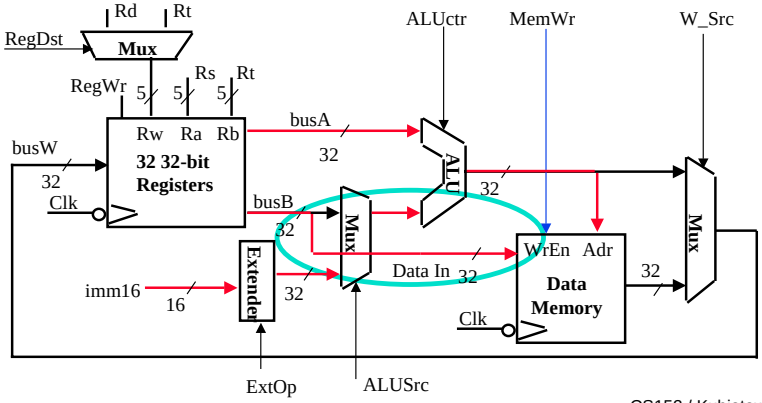




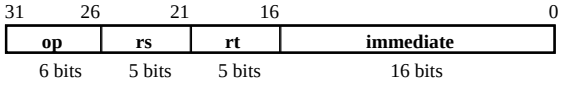
(3d) Recap: Store Operations

° Mem[R[rs] + SignExt[imm16] <- R[rt]] Example: sw rt, rs, imm16





(3e) Recap: The Branch Instruction

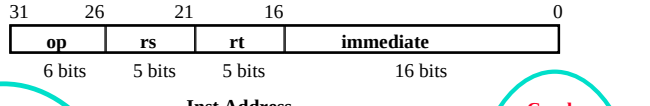


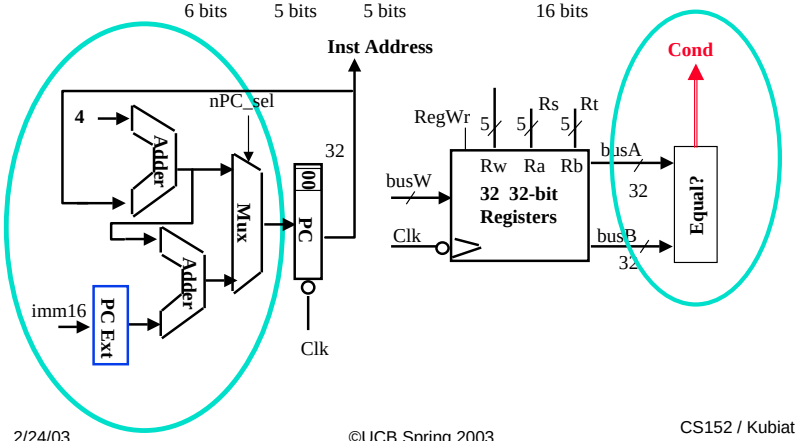
° beq rs, rt, imm16

- mem[PC] Fetch the instruction from memory
- Equal <- R[rs] == R[rt] Calculate the branch condition
- if (Equal) Calculate the next instruction's address
  - PC <- PC + 4 + (SignExt(imm16) x 4)
- else
  - PC <- PC + 4

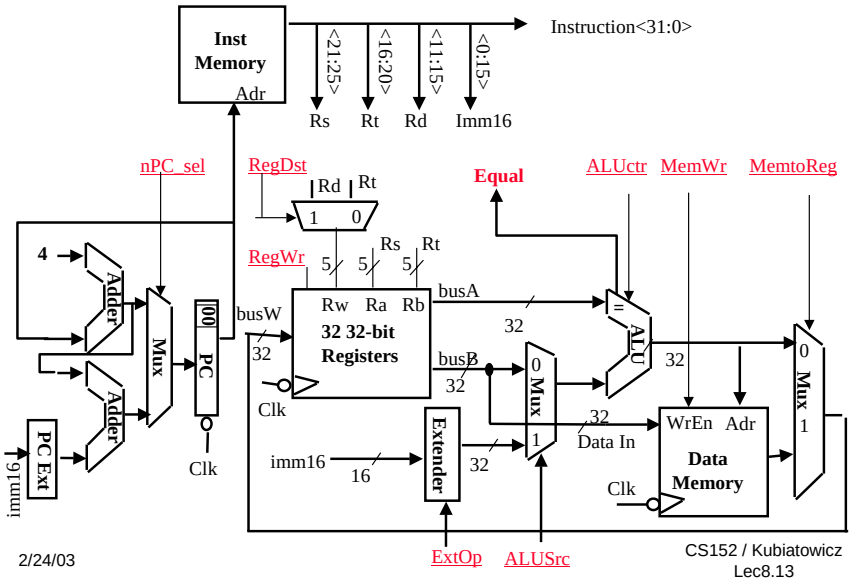
Recap: Datapath for Branch Operations

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



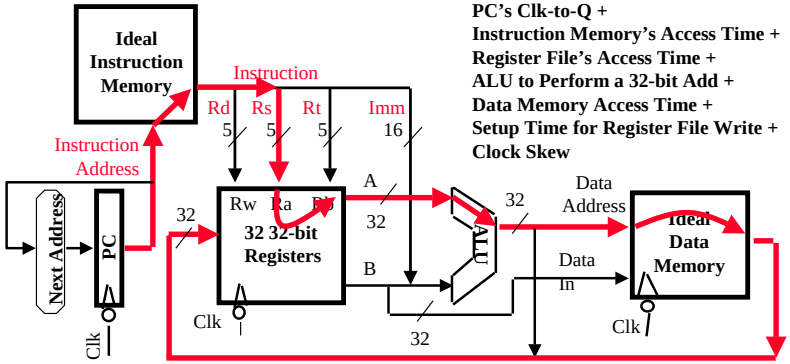


Putting it All Together: A Single Cycle Datapath



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."

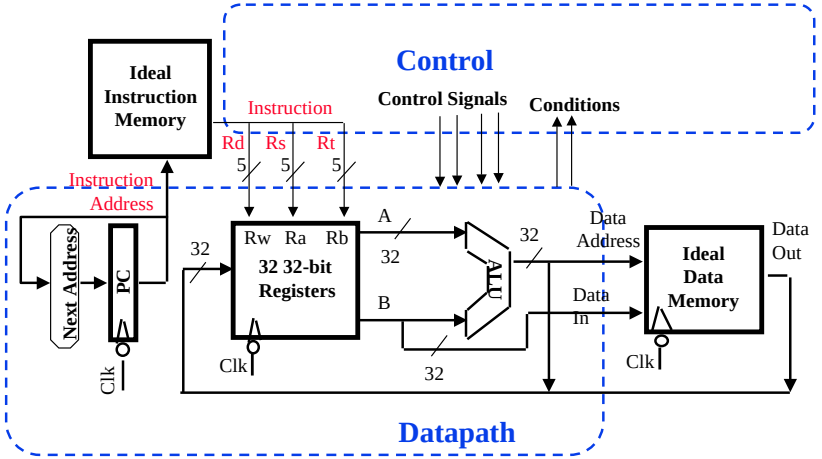


Critical Path (Load Operation) =
PC's Clk-to-Q +
Instruction Memory's Access Time +
Register File's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

Questions and Administrative Matters

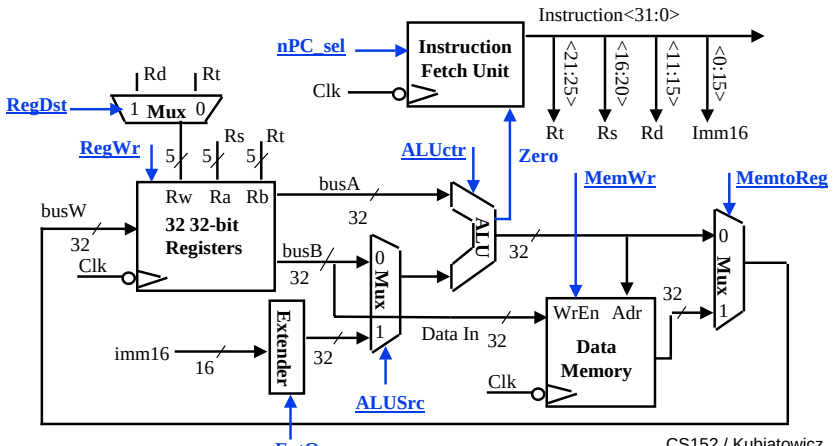
- Homework 3/Lab 3:
  - Get working on it!
  - Probably need to do post place and route simulation
    - Post-map simulation doesn't seem to have timing…?
- Midterm two weeks from Wednesday (3/12):
  - 5:30pm to 8:30pm, TBA (Probably 277 Cory)
  - Make-up quiz?
  - No class on that day (Office hours in my office)
  - Review section previous Sunday?
- Midterm reminders:
  - Pencil, calculator, one double-sided 8.5" x 11" page of *handwritten* notes
  - Sit in every other chair, every other row (odd row & odd seat)
- Meet at LaVal's pizza after the midterm
  - I'll Buy!

An Abstract View of the Implementation



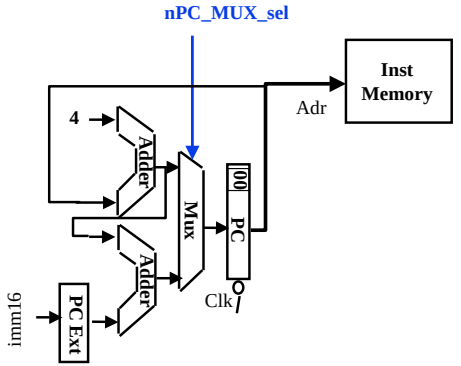
Recap: A Single Cycle Datapath

- Rs, Rt, Rd and Imed16 hardwired into datapath from Fetch Unit
- We have everything except control signals (underline)
  - Today's lecture will show you how to generate the control signals



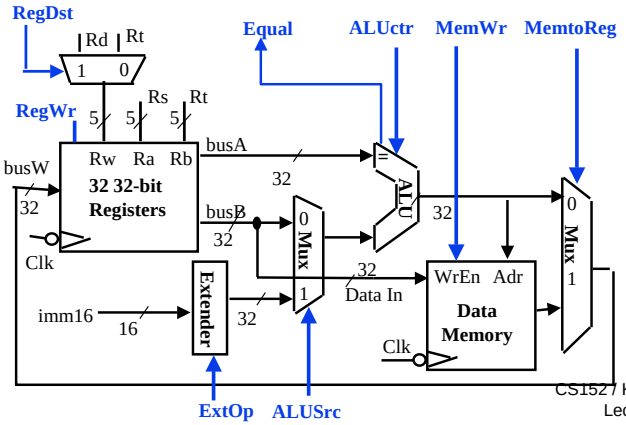
Recap: Meaning of the Control Signals

- nPC_MUX_sel: 0 ⇒ PC <– PC + 4
  1 ⇒ PC <– PC + 4 + SignExt(Im16) || 00
- Later in lecture: higher-level connection between mux and branch cond

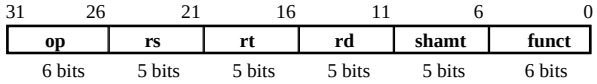


Recap: Meaning of the Control Signals

- 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



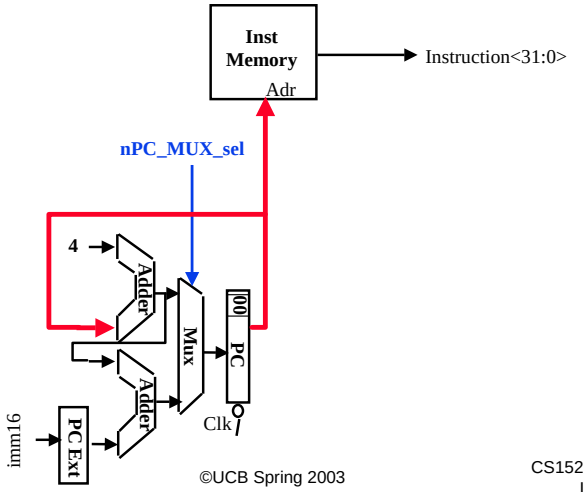
RTL: The Add Instruction



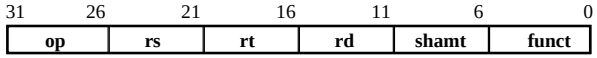
- add rd, rs, rt
  - mem[PC] — Fetch the instruction from memory
  - R[rd] <- R[rs] + R[rt] — The actual operation
  - PC <- PC + 4 — Calculate the next instruction's address

Instruction Fetch Unit at the Beginning of Add

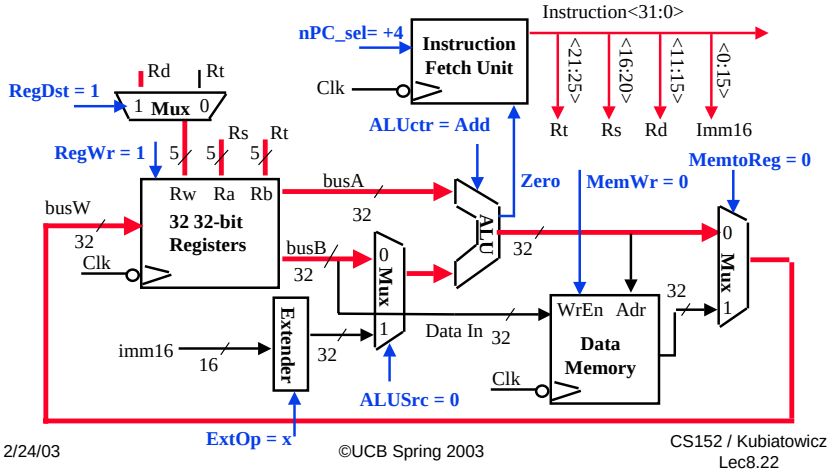
° Fetch the instruction from Instruction memory: Instruction <- mem[PC]
  • This is the same for all instructions



The Single Cycle Datapath during Add

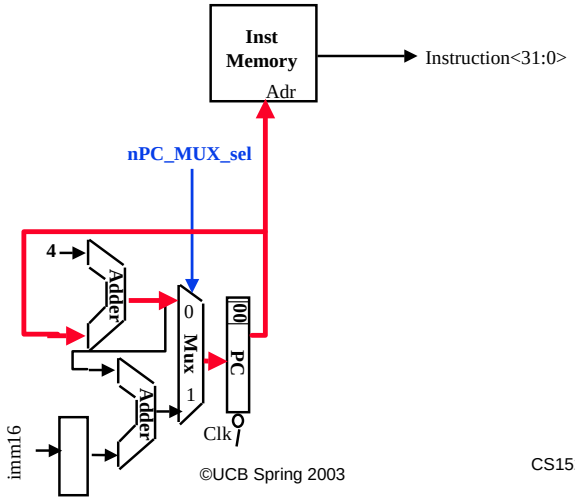


° R[rd] <- R[rs] + R[rt]

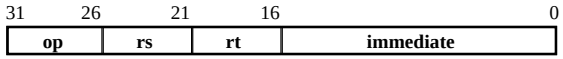


Instruction Fetch Unit at the End of Add

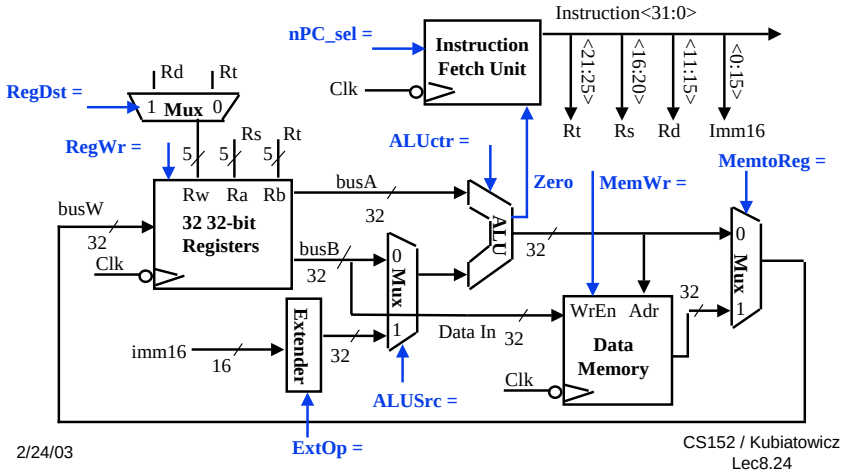
° PC <- PC + 4
  • This is the same for all instructions except: Branch and Jump



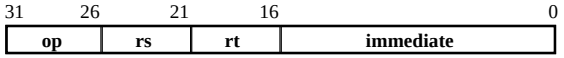
The Single Cycle Datapath during Or Immediate



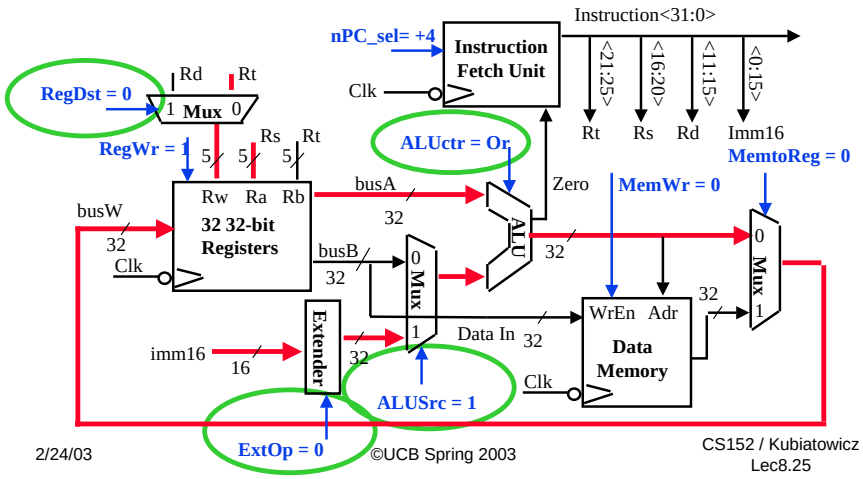
° R[rt] <- R[rs] or ZeroExt[Imm16]



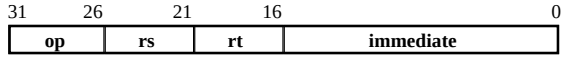
The Single Cycle Datapath during Or Immediate



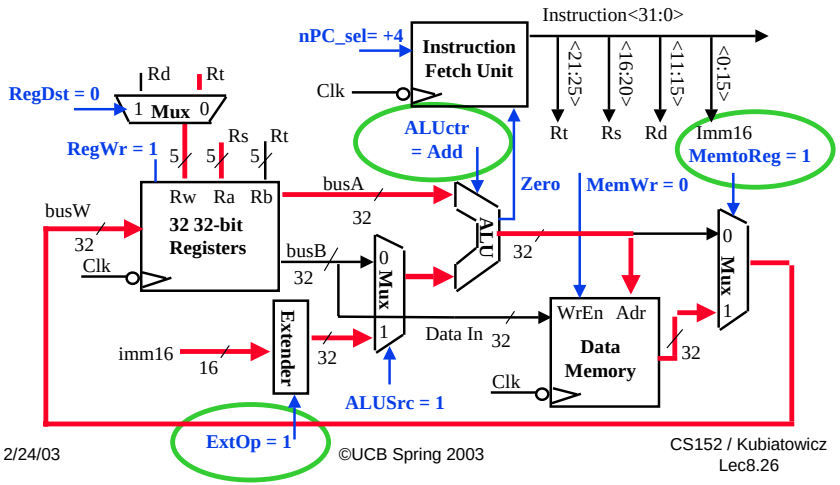
° R[rt] <- R[rs] or ZeroExt[Imm16]



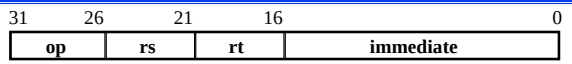
The Single Cycle Datapath during Load



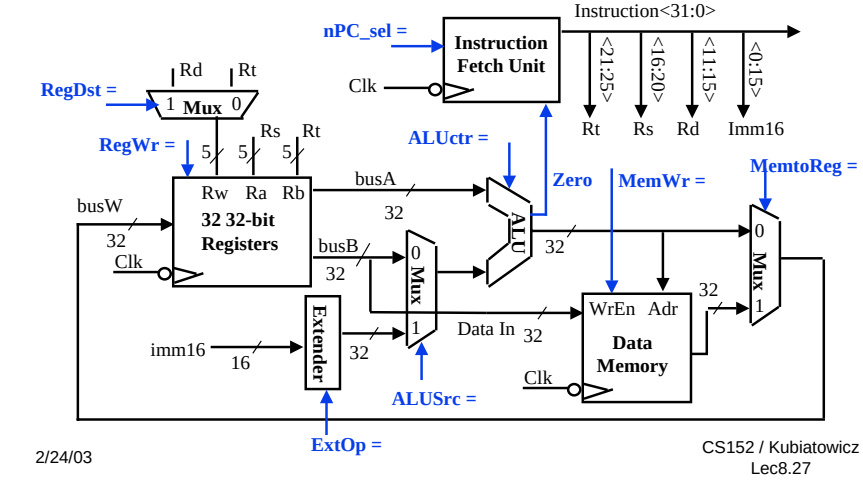
° R[rt] <- Data Memory {R[rs] + SignExt[imm16]}



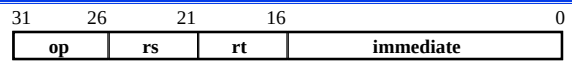
The Single Cycle Datapath during Store



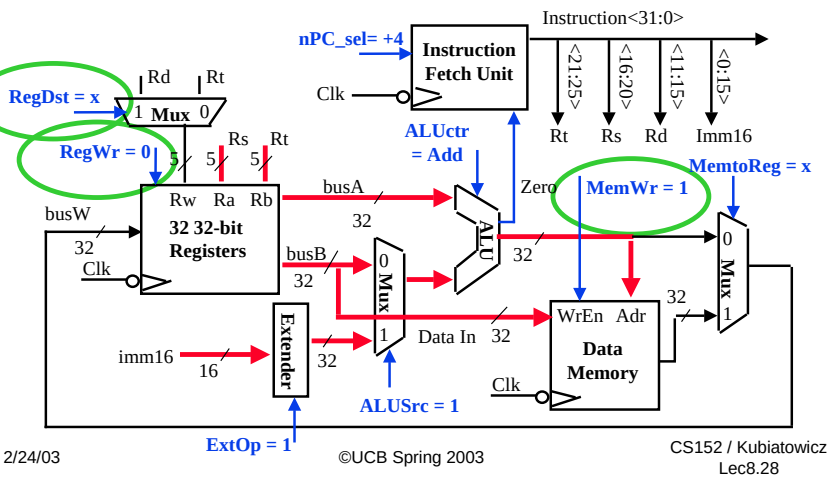
° Data Memory {R[rs] + SignExt[imm16]} <- R[rt]



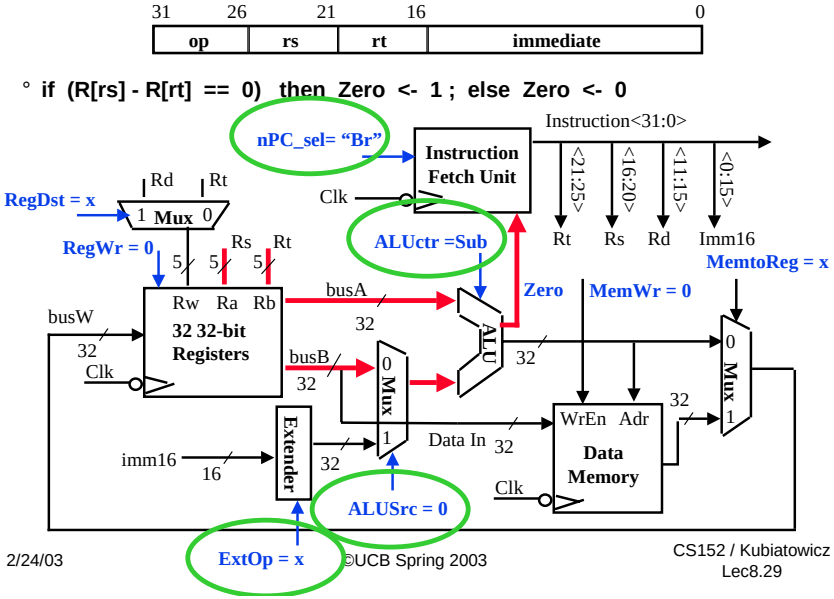
The Single Cycle Datapath during Store



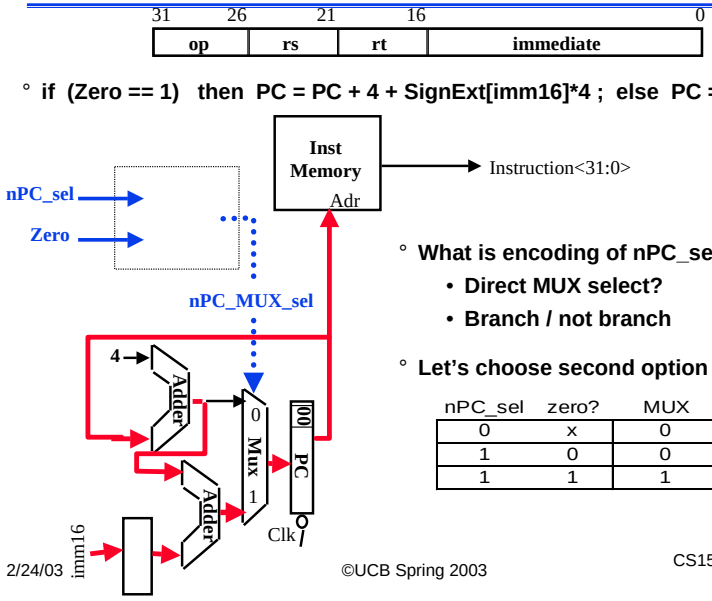
° Data Memory {R[rs] + SignExt[imm16]} <- R[rt]



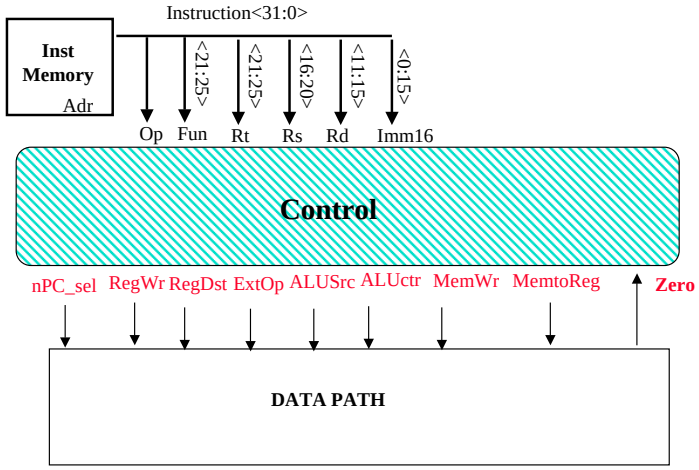
The Single Cycle Datapath during Branch



Instruction Fetch Unit at the End of Branch



Step 4: Given Datapath: RTL -> Control



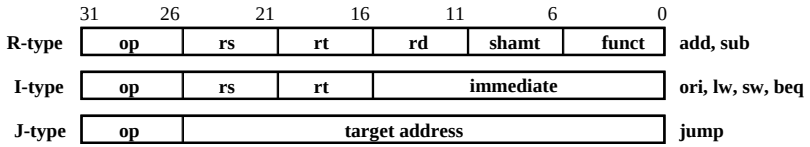
A Summary of Control Signals

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"	
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

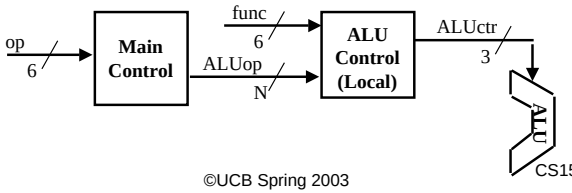
See Appendix A → func / op

	add	sub	ori	lw	sw	beq	jump
	func: 10 0000 op: 00 0000	func: 10 0010 op: 00 0000	We Don't Care :-) op: 00 1101	op: 10 0011	op: 10 1011	op: 00 0100	op: 00 0010
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

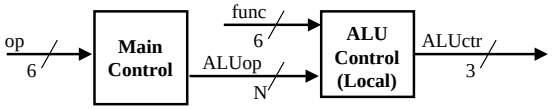


The Concept of Local Decoding

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



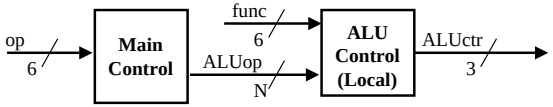
The Encoding of ALUop



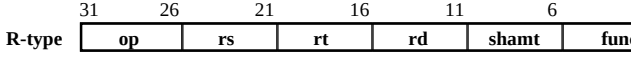
- In this exercise, ALUop has to be 2 bits wide to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract
- To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) "R-type" instructions
  - "I-type"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



P. 286 text:

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

The Truth Table for ALUctr

ALUop (Symbolic)	R-type	ori	lw	sw	beq
ALUop<2:0>	“R-type”	Or	Add	Add	Subtract
	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			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

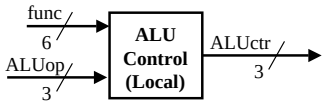
° **ALUctr<1> = !ALUop<2> & !ALUop<1> +**
ALUop<2> & !func<2> & !func<0>

The Logic Equation for ALUctr<0>

ALUop			func				ALUctr<0>
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<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 <= (OP == 'BEQ) ? 'Br : 'plus4;

° ALUsrc <= (OP == 'Rtype) ? 'regB : 'immed;

° ALUctr <= (OP == 'Rtype') ? funct :
(OP == 'ORi) ? 'ORfunction :
(OP == 'BEQ) ? 'SUBfunction : 'ADDfunction;

° ExtOp <= _____

° MemWr <= _____

° MemtoReg <= _____

° RegWr: <= _____

° RegDst: <= _____

Step 5: Logic for each control signal

° nPC_sel <= (OP == 'BEQ) ? 'Br : 'plus4;

° ALUsrc <= (OP == 'Rtype) ? 'regB : 'immed;

° ALUctr <= (OP == 'Rtype') ? funct :
(OP == 'ORi) ? 'ORfunction :
(OP == 'BEQ) ? 'SUBfunction : 'ADDfunction;

° ExtOp <= (OP == 'ORi) : 'ZEROextend : 'SIGNextend;

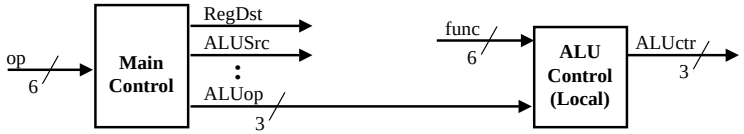
° MemWr <= (OP == 'Store) ? 1 : 0;

° MemtoReg <= (OP == 'Load) ? 1 : 0;

° RegWr: <= ((OP == 'Store) || (OP == 'BEQ)) ? 0 : 1;

° RegDst: <= ((OP == 'Load) || (OP == 'ORi)) ? 0 : 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
nPC_sel	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

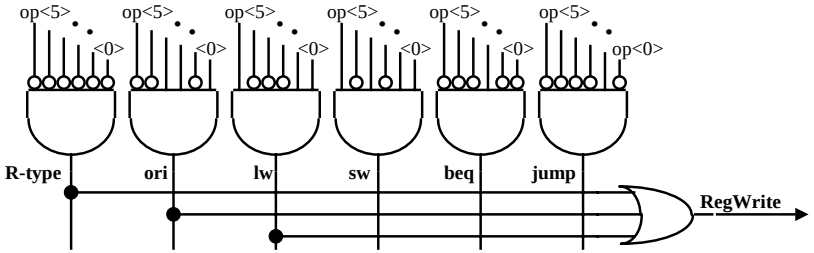
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	0	0	0

° RegWrite = R-type + ori + lw

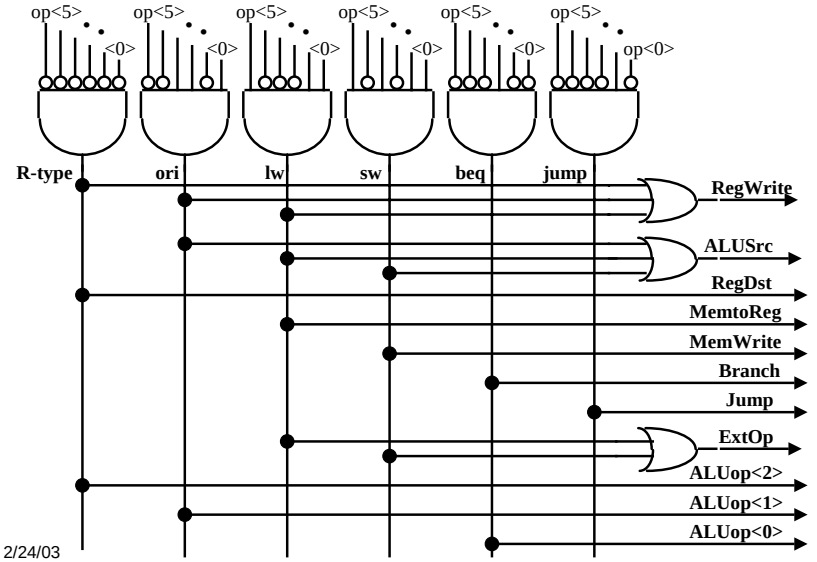
= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)

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

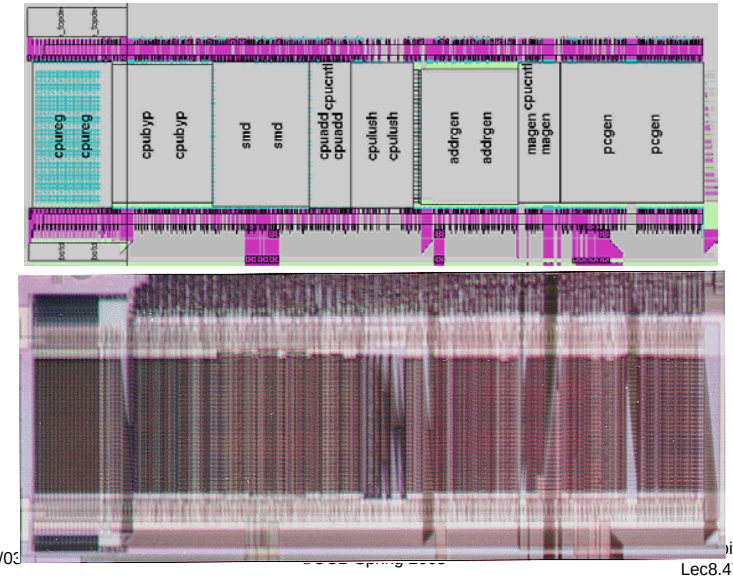
+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



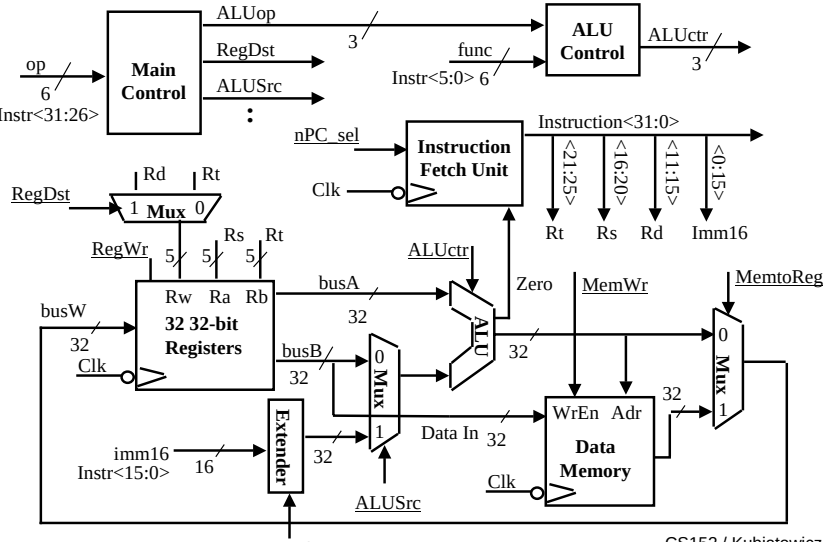
PLA Implementation of the Main Control



A Real MIPS Datapath (CNS T0)

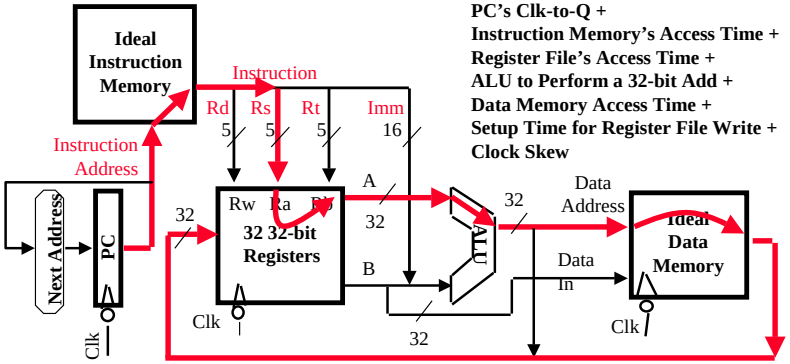


Putting it All Together: A Single Cycle Processor



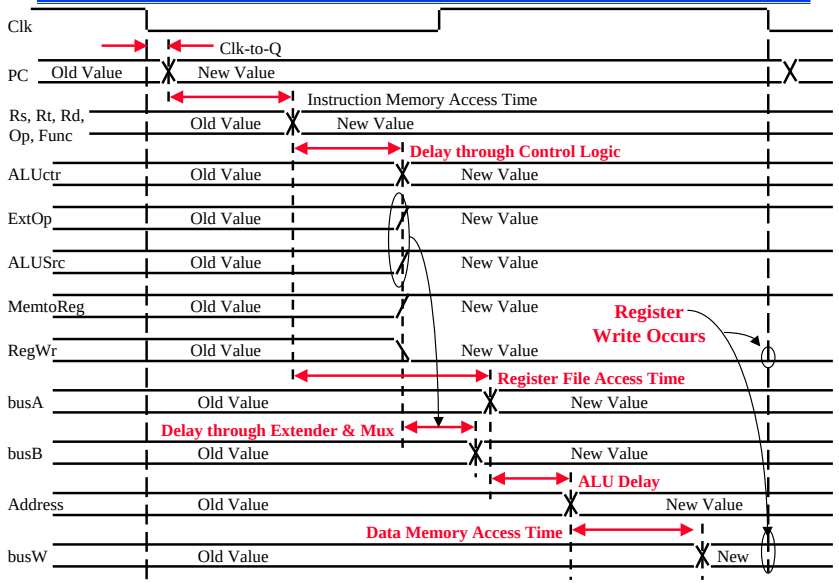
Recap: An Abstract View of the Critical Path (Load)

° 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

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

Preview

Next Time:

° How to design with High-Level Design Languages (HDL)
  • Multi-level design: successive refinement
  • Datapath in Schematics
  • Control in HDL with eventual synthesis

° MultiCycle Data Path
  • CPI ≥ 1, CycleTime much shorter (~1/5 of time)

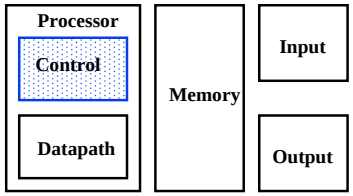
Summary

° 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



Where to get more information?

° Chapter 5.1 to 5.3 of your text book:
- David Patterson and John Hennessy, "Computer Organization & Design: The Hardware / Software Interface," Second Edition, Morgan Kaufman Publishers, San Mateo, California, 1998.

° One of the best PhD thesis on processor design:
- Manolis Katevenis, "Reduced Instruction Set Computer Architecture for VLSI," PhD Dissertation, EECS, U C Berkeley, 1982.

° For a reference on the MIPS architecture:
- Gerry Kane, "MIPS RISC Architecture," Prentice Hall.