CS 152
Computer Architecture and Engineering
Lecture 8

Single-Cycle (Con't)
Designing a Multicycle Processor

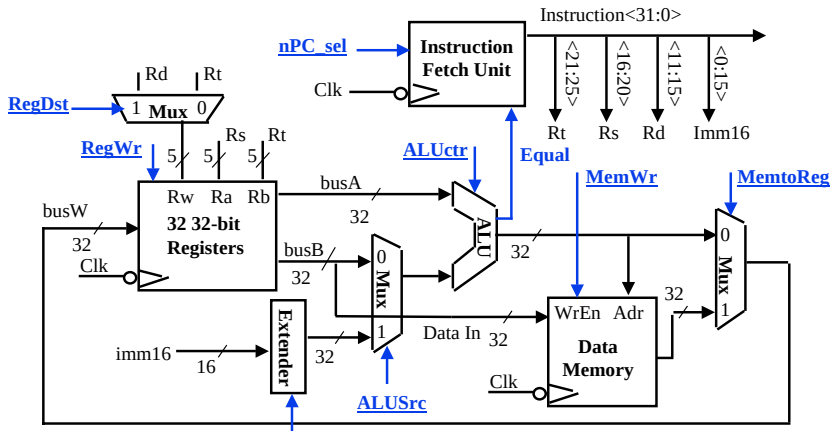
February 23, 2004

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

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

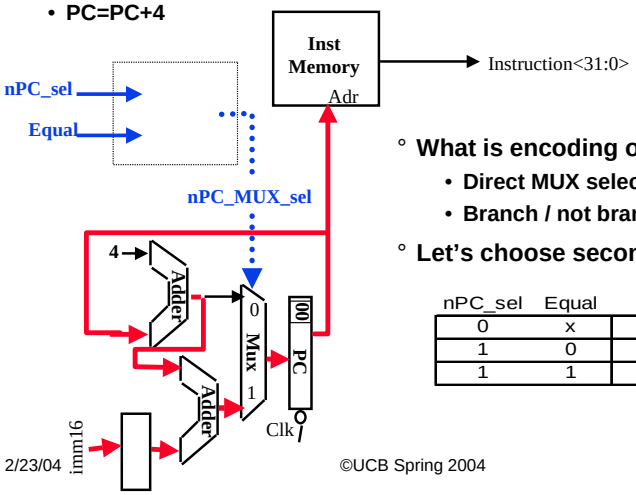
Recap: A Single Cycle Datapath

- Rs, Rt, Rd and Imed16 hardwired into datapath from Fetch Unit
- We have everything except control signals (underline)



Recap: Flexible Instruction Fetch

- Branch (nPC_sel = "Br"):
  - if (Equal == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4
- Other (nPC_sel = "+4"):
  - PC=PC+4



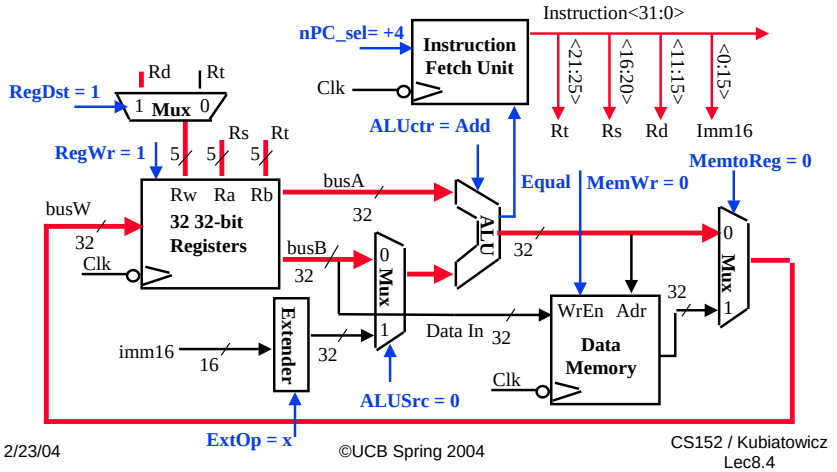
- What is encoding of nPC_sel?
  - Direct MUX select?
  - Branch / not branch
- Let's choose second option

nPC_sel	Equal	MUX
0	x	0
1	0	0
1	1	1

Recap: The Single Cycle Datapath during Add

31	26	21	16	11	6	0
op	rs	rt	rd	shamt	funct	

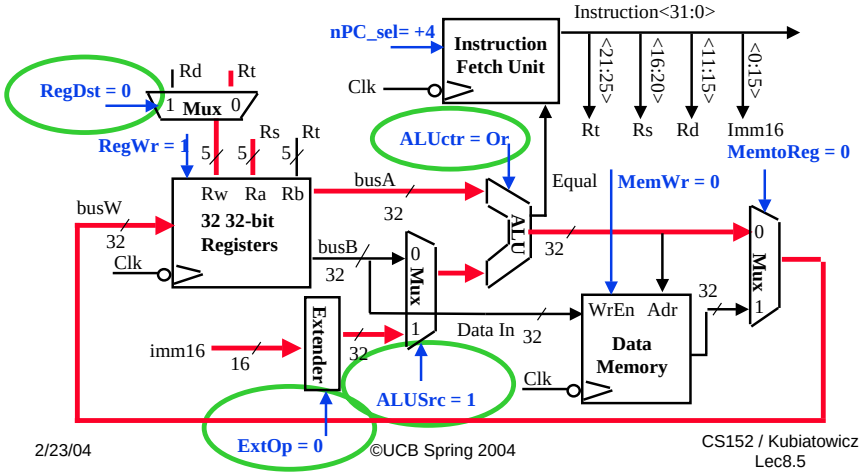
- R[rd] <- R[rs] + R[rt]



Recap: The Single Cycle Datapath during `Or` Immediate

31–26	21–16	16–0
op	rs	rt	immediate

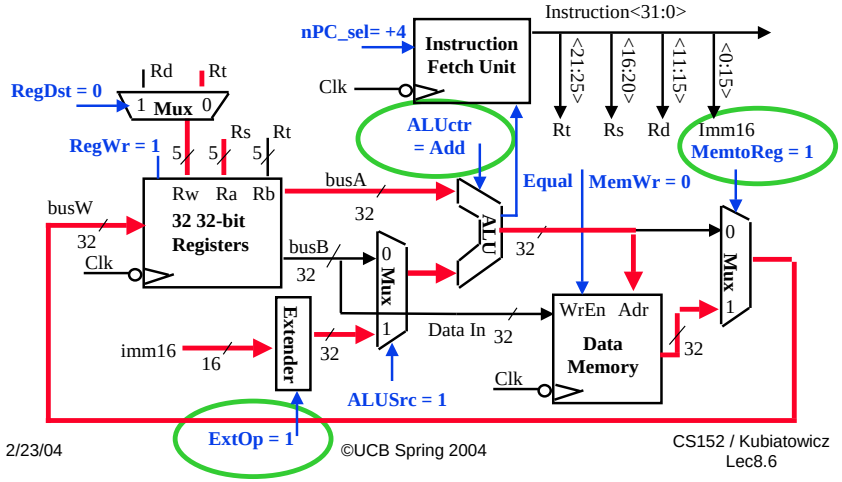
° R[rt] <- R[rs] or ZeroExt[Imm16]



Recap: The Single Cycle Datapath during Load

31–26	21–16	16–0
op	rs	rt	immediate

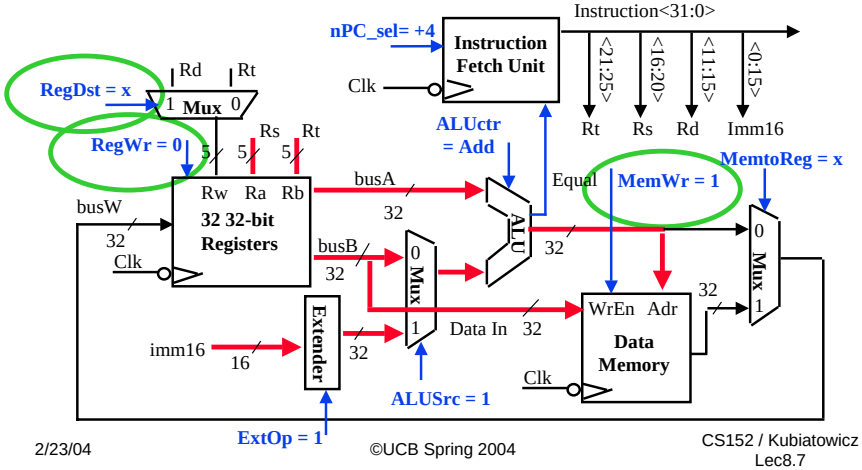
° R[rt] <- Data Memory {R[rs] + SignExt[imm16]}



Recap: The Single Cycle Datapath during `Store`

31–26	21–16	16–0
op	rs	rt	immediate

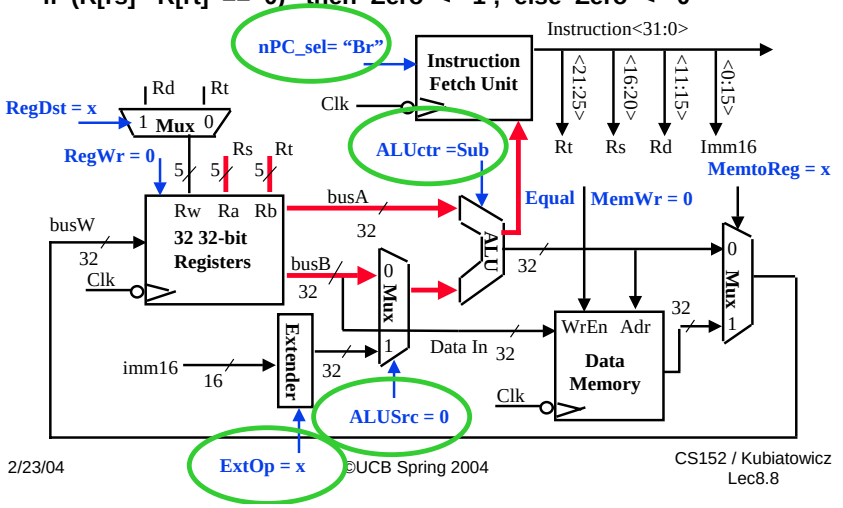
° Data Memory {R[rs] + SignExt[imm16]} <- R[rt]



Recap: The Single Cycle Datapath during Branch

31–26	21–16	16–0
op	rs	rt	immediate

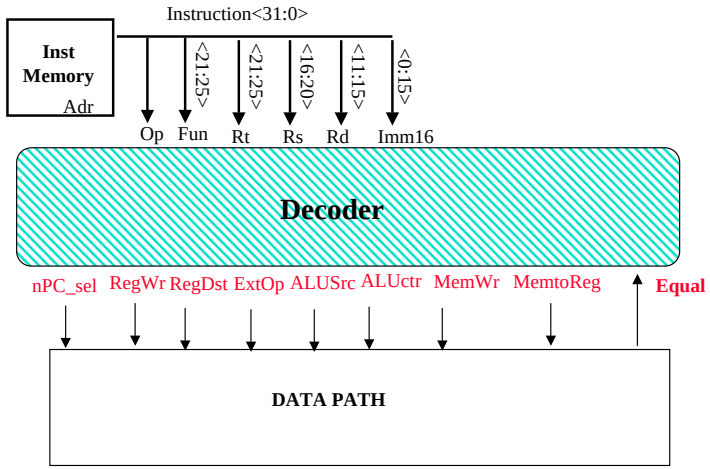
° if (R[rs] - R[rt] == 0) then Zero <- 1 ; else Zero <- 0



Recap: 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"	

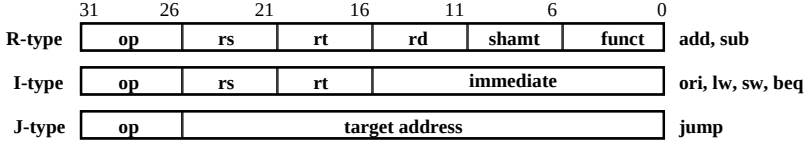
Step 5: Assemble Control logic



A Summary of the Control Signals

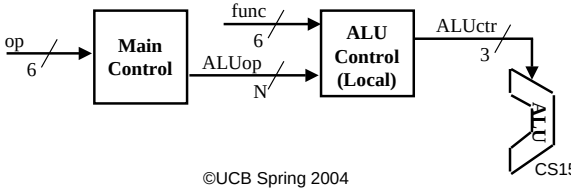
See Appendix A

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

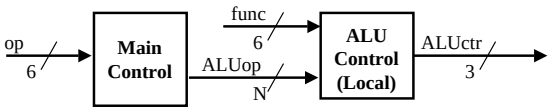


The Concept of Local Decoding

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	Subtract



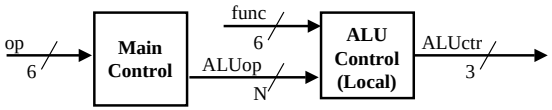
The Encoding of ALUop



- 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

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

The Decoding of the "func" Field



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

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

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
	"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

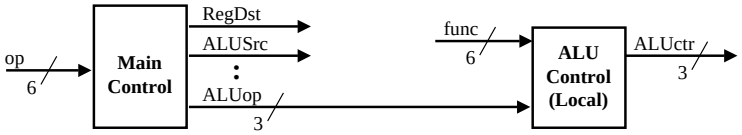
Step 5: Logic for each control signal

```
`define   Rtype   6`b000000;
`define   BEQ     6`b000100;
`define   Ori     6`b001101;
`define   Load    6`b100011;
`define   Store   6`b101011;

          … etc …

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
	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
Jump	0	0	0	0	0
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

The "Truth Table" for RegWrite

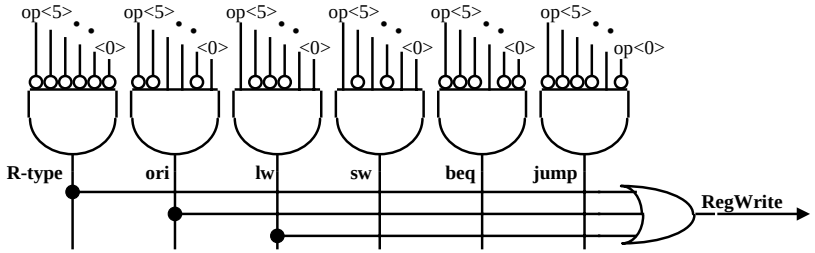
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	0	0	0

° RegWrite = R-type + ori + lw

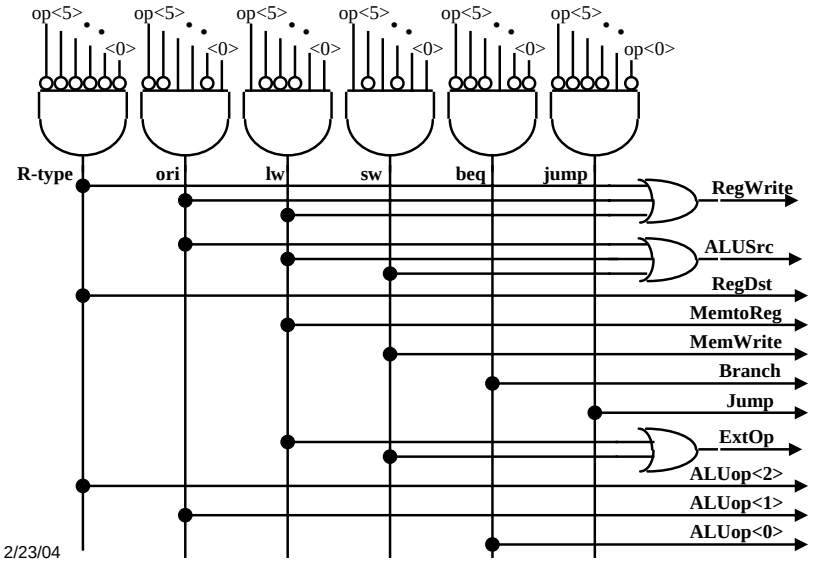
= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)

+ !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



Administrative Issues

° Read Chapter 5

° This lecture and next one slightly different from the book

° Design Document for lab 3 due in section this Thursday!

- Describe your division of labor
- Your testing methodology (how will you test each step of the way?)
- Top-level block diagrams

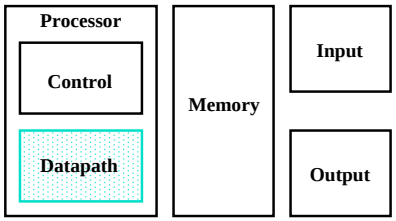
° Midterm on Wednesday 3/10

- 5:30pm to 8:30pm, location TBA
- No class on that day:
- Pencil, calculator, one 8.5" x 11" (both sides) of handwritten notes

° Meet at LaVal's pizza after the midterm

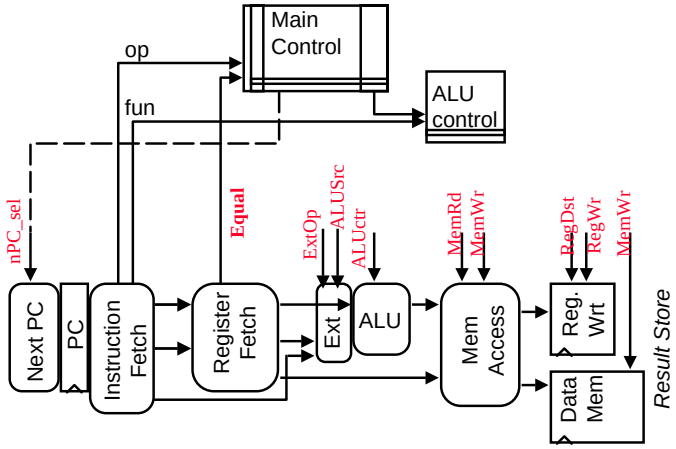
The Big Picture: Where are We Now?

° The Five Classic Components of a Computer



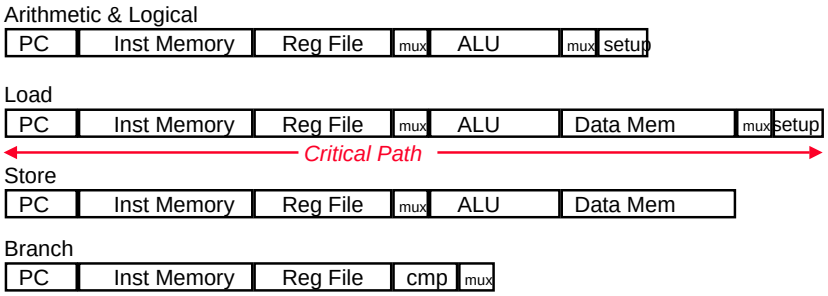
° Today's Topic: Designing the Datapath for the Multiple Clock Cycle Datapath

Abstract View of our single cycle processor



° looks like a FSM with PC as state

What's wrong with our CPI=1 processor?



° Long Cycle Time

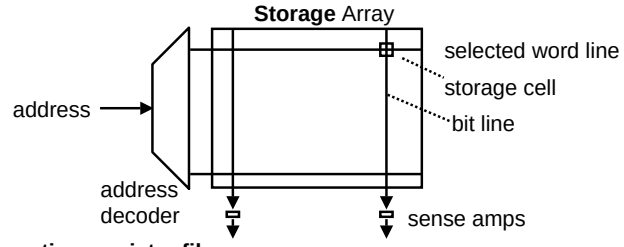
° All instructions take as much time as the slowest

° Real memory is not as nice as our idealized memory

- cannot always get the job done in one (short) cycle

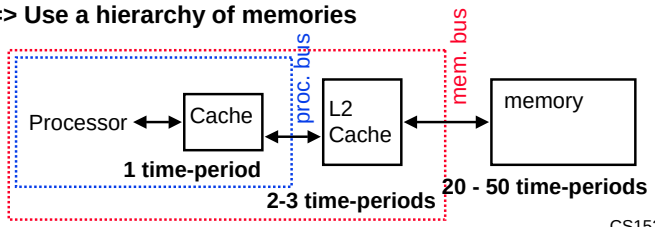
Memory Access Time

° Physics => fast memories are small (large memories are slow)



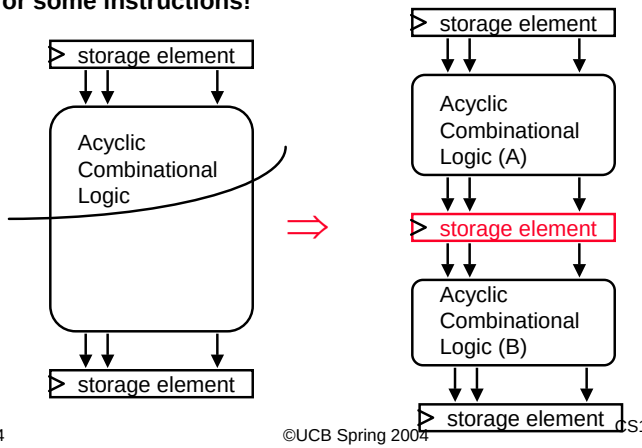
- question: register file vs. memory

° => Use a hierarchy of memories

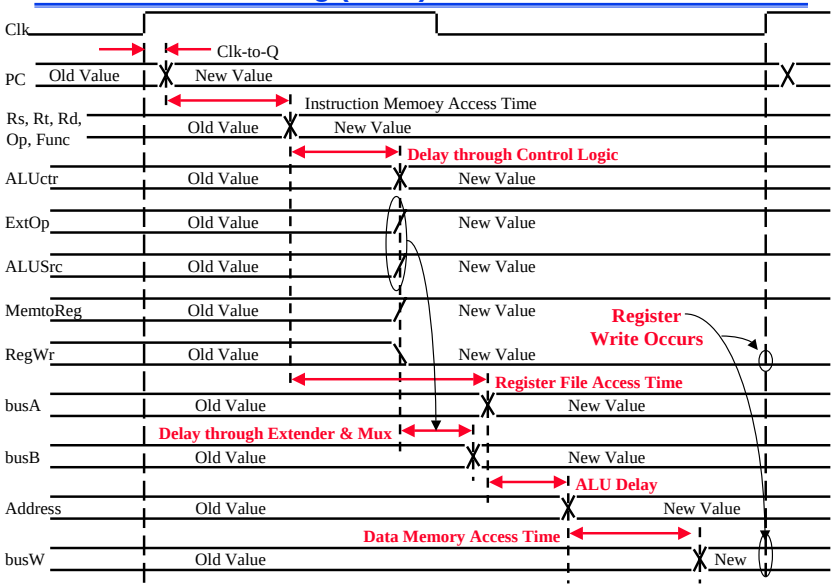


Reducing Cycle Time

- Cut combinational dependency graph and insert register / latch
- Do same work in two fast cycles, rather than one slow one
- May be able to short-circuit path and remove some components for some instructions!

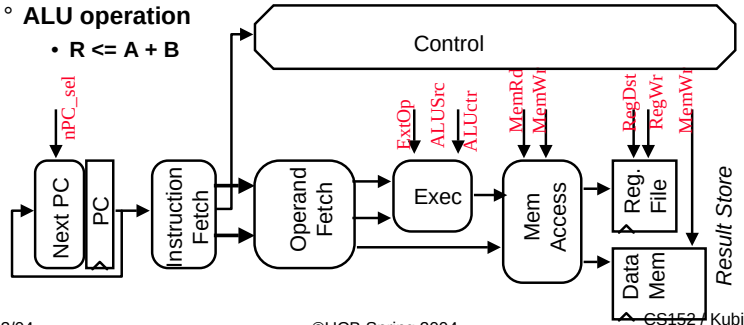


Worst Case Timing (Load)



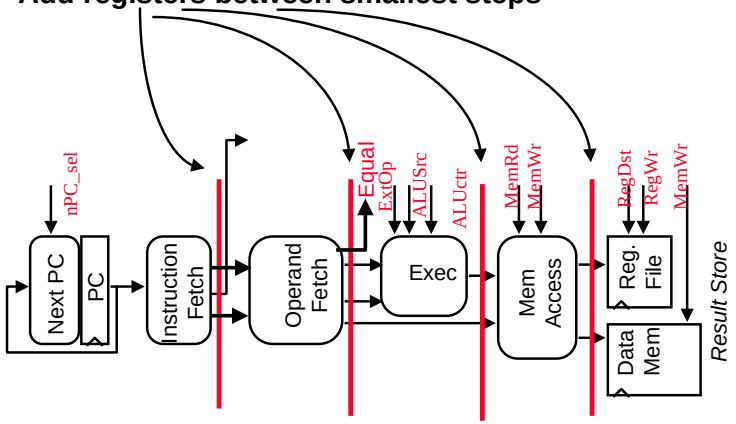
Basic Limits on Cycle Time

- Next address logic
  - PC <= branch ? PC + offset : PC + 4
- Instruction Fetch
  - InstructionReg <= Mem[PC]
- Register Access
  - A <= R[rs]
- ALU operation
  - R <= A + B



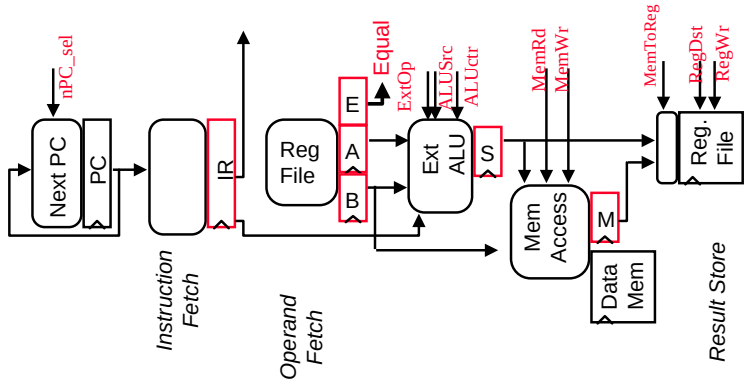
Partitioning the CPI=1 Datapath

- Add registers between smallest steps



- Place enables on all registers

Example Multicycle Datapath



° **Critical Path ?**

Recall: Step-by-step Processor Design

Step 1: ISA => Logical Register Transfers

Step 2: Components of the Datapath

Step 3: RTL + Components => Datapath

Step 4: Datapath + Logical RTs => Physical RTs

Step 5: Physical RTs => Control

Step 4: R-rtype (add, sub, . . .)

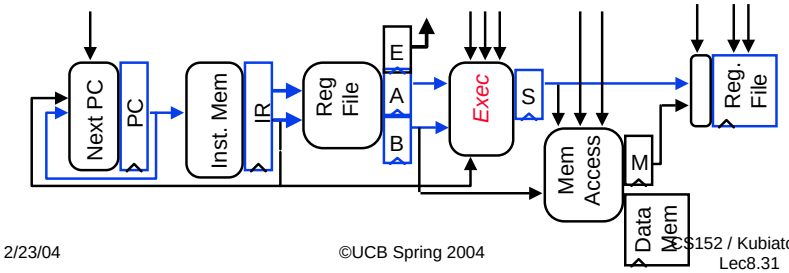
° **Logical Register Transfer**

inst	Logical Register Transfers
ADDU	R[rd] <– R[rs] + R[rt]; PC <– PC + 4

° **Physical Register Transfers**

Time

inst	Physical Register Transfers
	IR <– MEM[pc]
ADDU	A<– R[rs]; B <– R[rt]
	S <– A + B
	R[rd] <– S; PC <– PC + 4



Step 4: Logical immed

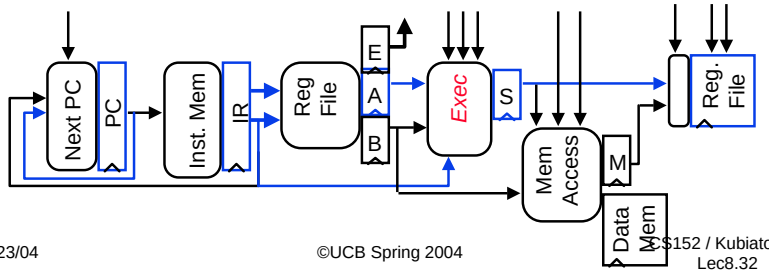
° **Logical Register Transfer**

inst	Logical Register Transfers
ORI	R[rt] <– R[rs] OR ZExt(Im16); PC <– PC + 4

° **Physical Register Transfers**

Time

inst	Physical Register Transfers
	IR <– MEM[pc]
ORI	A<– R[rs]; B <– R[rt]
	S <– A *or* ZExt(Im16)
	R[rt] <– S; PC <– PC + 4



Step 4 : Load

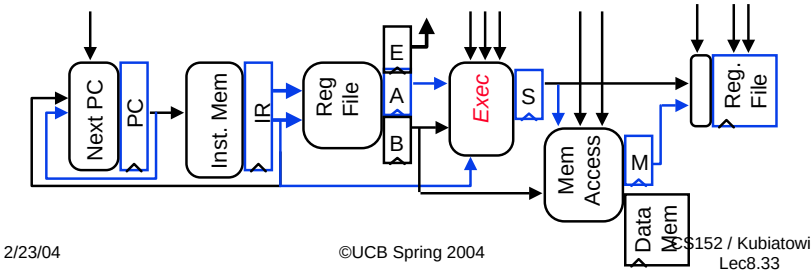
° Logical Register Transfer

inst	Logical Register Transfers
LW	R[rt] <– MEM[R[rs] + SExt(Im16)];
	PC <– PC + 4

° Physical Register Transfers

Time

inst	Physical Register Transfers
	IR <– MEM[pc]
LW	A<– R[rs]; B <– R[rt]
	S <– A + SExt(Im16)
	M <– MEM[S]
	R[rd] <– M; PC <– PC + 4



Step 4 : Store

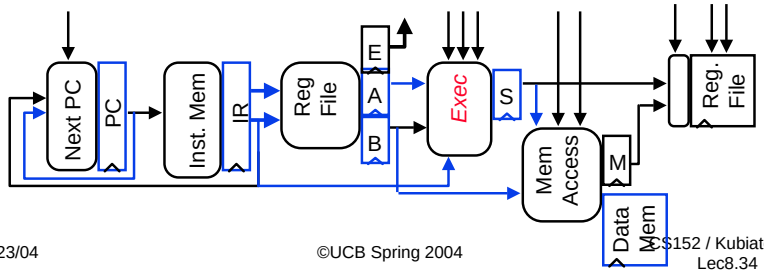
° Logical Register Transfer

inst	Logical Register Transfers
SW	MEM[R[rs] + SExt(Im16)] <– R[rt];
	PC <– PC + 4

° Physical Register Transfers

Time

inst	Physical Register Transfers
	IR <– MEM[pc]
SW	A<– R[rs]; B <– R[rt]
	S <– A + SExt(Im16);
	MEM[S] <– B PC <– PC + 4



Step 4 : Branch

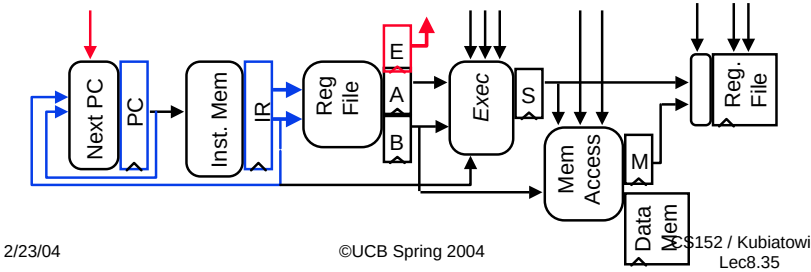
° Logical Register Transfer

inst	Logical Register Transfers
BEQ	if R[rs] == R[rt]
	then PC <= PC + 4+SExt(Im16) || 00
	else PC <= PC + 4

° Physical Register Transfers

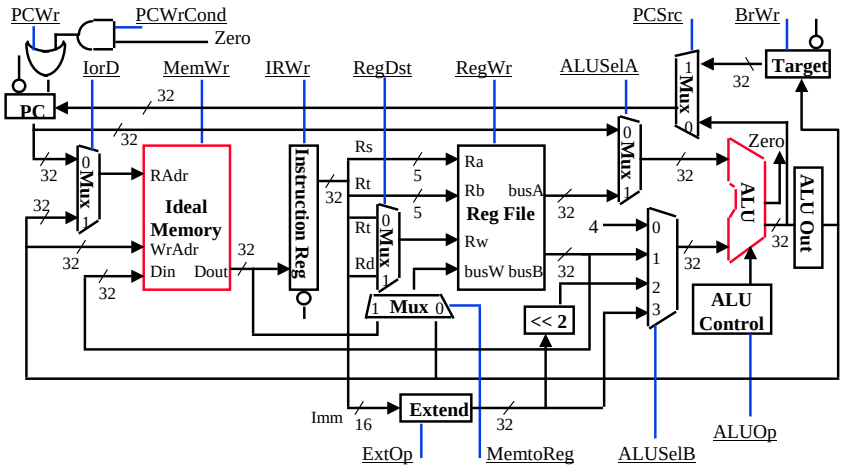
Time

inst	Physical Register Transfers
	IR <– MEM[pc]
BEQ	E<– (R[rs] = R[rt])
	if !E then PC <– PC + 4 else PC <–PC+4+SExt(Im16)||00



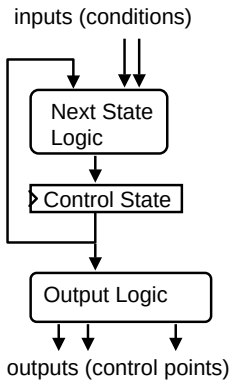
Alternative datapath (book): Multiple Cycle Datapath

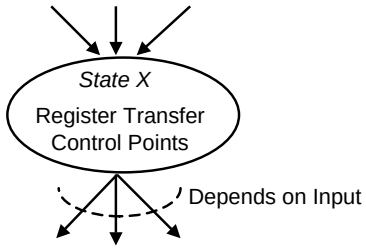
° Miminizes Hardware: 1 memory, 1 adder



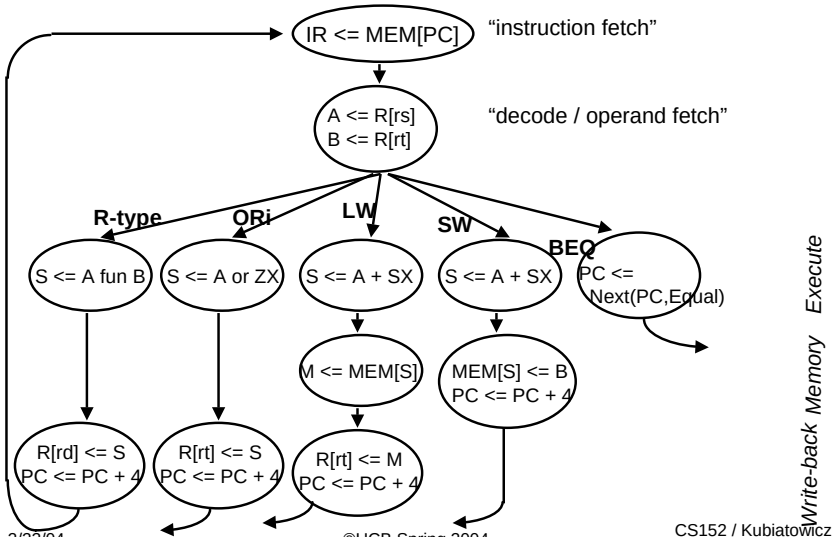
Our Control Model

- State specifies control points for Register Transfer
- Transfer occurs upon exiting state (same falling edge)

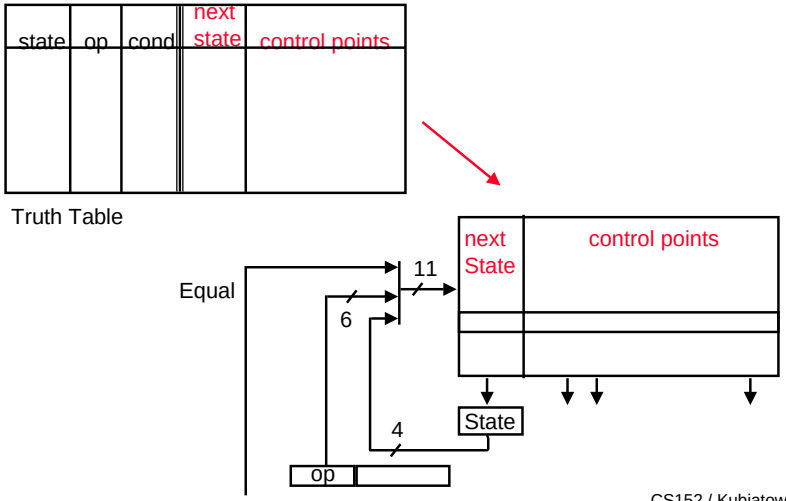




Step 4 ⇒ Control Specification for multicycle proc



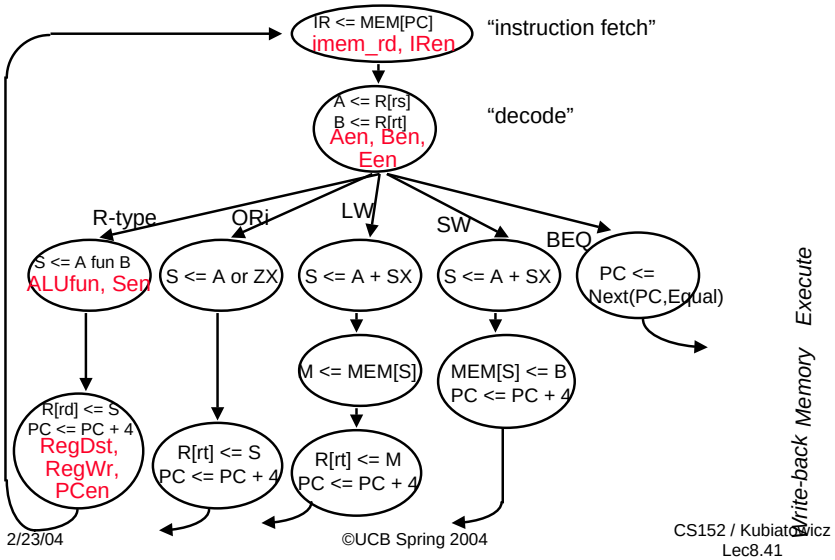
Traditional FSM Controller



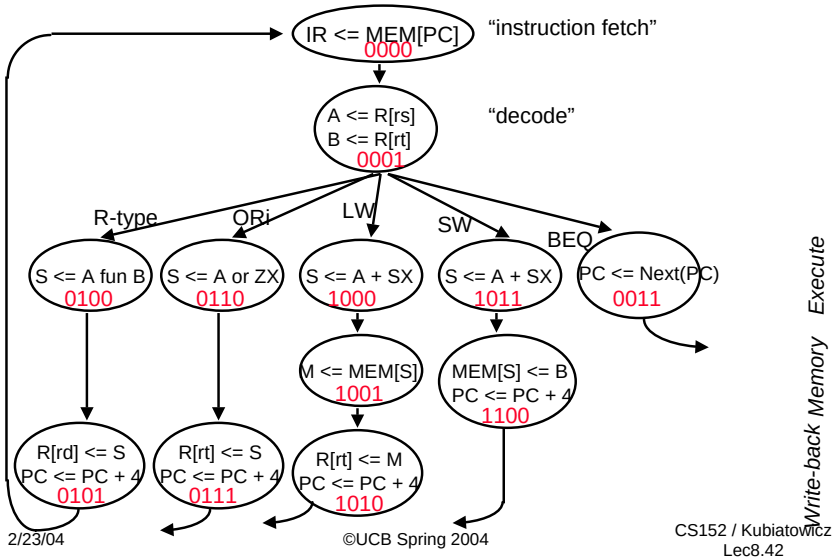
Step 5 ⇒ (datapath + state diagram ⇒ control)

- Translate RTs into control points
- Assign states
- Then go build the controller

Mapping RTs to Control Points



Assigning States



(Mostly) Detailed Control Specification (missing⇒0)

	State	Op field	Eq	Next	IR	PC en	PC sel	Ops A	Ops B	Ops E	Exec Ex	Exec Sr	Exec ALU	Exec S	Mem R	Mem W	Mem M	Write-Back M-R	Write-Back Wr	Write-Back Dst
	0000	??????	?	0001	1															
	0001	BEQ	x	0011				1	1	1										
	0001	R-type	x	0100				1	1	1										
	0001	ORI	x	0110				1	1	1										
	0001	LW	x	1000				1	1	1										
	0001	SW	x	1011				1	1	1										
BEQ:	0011	xxxxxx	0	0000		1	0											x	0	x
	0011	xxxxxx	1	0000		1	1											x	0	x
R:	0100	xxxxxx	x	0101							0	1	fun	1						
	0101	xxxxxx	x	0000		1	0											0	1	1
ORi:	0110	xxxxxx	x	0111							0	0	or	1						
	0111	xxxxxx	x	0000		1	0											0	1	0
LW:	1000	xxxxxx	x	1001							1	0	add	1						
	1001	xxxxxx	x	1010											1	0	1			
	1010	xxxxxx	x	0000		1	0											1	1	0
SW:	1011	xxxxxx	x	1100							1	0	add	1						
	1100	xxxxxx	x	0000		1	0								0	1	0			

-all same in Moore machine

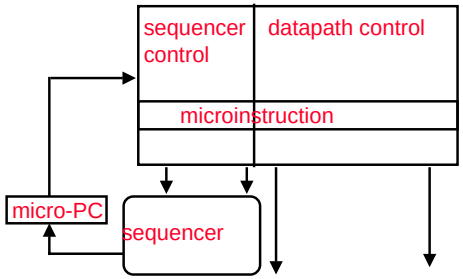
Performance Evaluation

- **What is the average CPI?**
  - **state diagram gives CPI for each instruction type**
  - **workload gives frequency of each type**

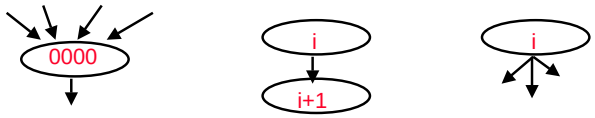
Type	CPI_i for type	Frequency	CPI_i x $freqI_i$
Arith/Logic	4	40%	1.6
Load	5	30%	1.5
Store	4	10%	0.4
branch	3	20%	0.6
		Average CPI:	4.1

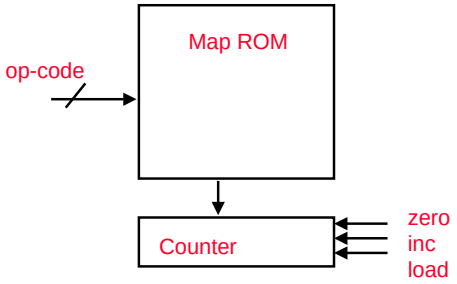
Controller Design

- The state digrams that arise define the controller for an instruction set processor are highly structured
- Use this structure to construct a simple "microsequencer"
- Control reduces to programming this very simple device
  - ⇒ microprogramming



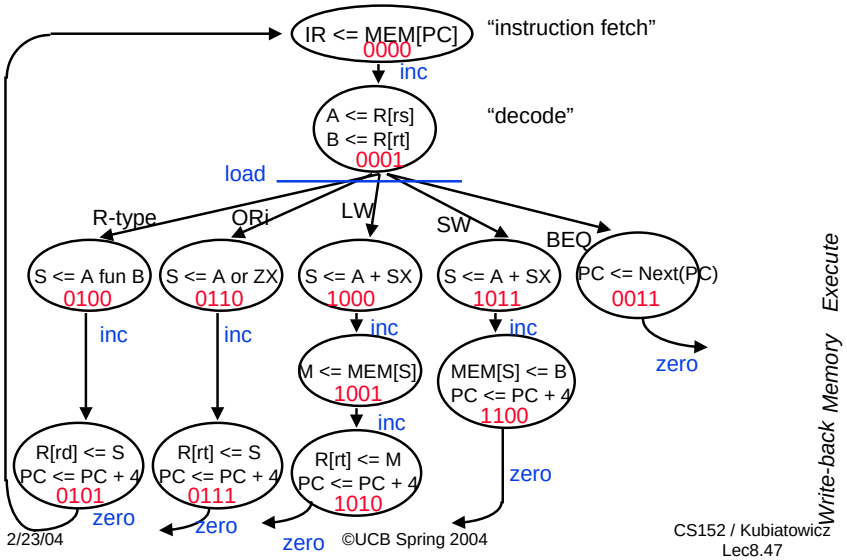
Example: Jump-Counter



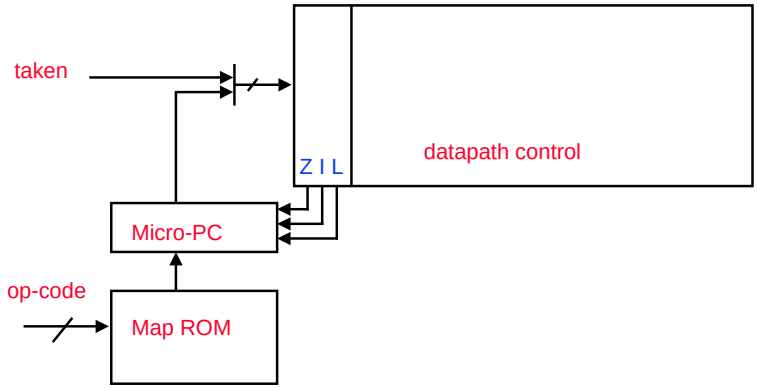


None of above: Do nothing (for wait states)

Using a Jump Counter



Our Microsequencer

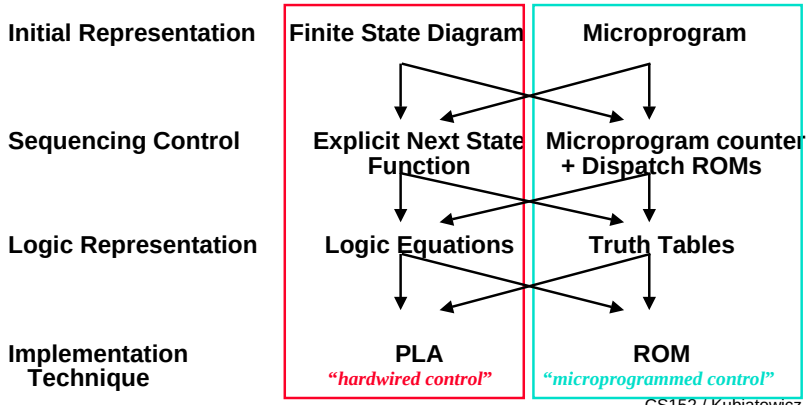


Microprogram Control Specification

	µPC	Taken	Next	IR	PC en sel	Ops A B	Exec Ex Sr ALU S	Mem R W M	Write-Back M-R Wr Dst
	0000	?	inc	1					
	0001	0	load			1 1			
BEQ	0011	0	zero		1 0				
	0011	1	zero		1 1				
R:	0100	x	inc				0 1 fun 1		
	0101	x	zero		1 0				0 1 1
ORi:	0110	x	inc				0 0 or 1		
	0111	x	zero		1 0				0 1 0
LW:	1000	x	inc				1 0 add 1		
	1001	x	inc					1 0 1	
	1010	x	zero		1 0				1 1 0
SW:	1011	x	inc				1 0 add 1		
	1100	x	zero		1 0			0 1 0	

Overview of Control

° Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.



Summary

° Disadvantages of the Single Cycle Processor
- Long cycle time
- Cycle time is too long for all instructions except the Load

° Multiple Cycle Processor:
- Divide the instructions into smaller steps
- Execute each step (instead of the entire instruction) in one cycle

° Partition datapath into equal size chunks to minimize cycle time
- ~10 levels of logic between latches

° Follow same 5-step method for designing "real" processor

Summary (cont'd)

° Control is specified by finite state digram

° Specialize state-diagrams easily captured by microsequencer
- simple increment & "branch" fields
- datapath control fields

° Control design reduces to Microprogramming

° Control is more complicated with:
- complex instruction sets
- restricted datapaths (see the book)

° Simple Instruction set and powerful datapath ⇒ simple control
- could try to reduce hardware (see the book)
- rather go for speed => many instructions at once!