CS152 – Computer Architecture and Engineering

Lecture 9 – Multicycle Design
2003-09-22

Dave Patterson
(www.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs152/

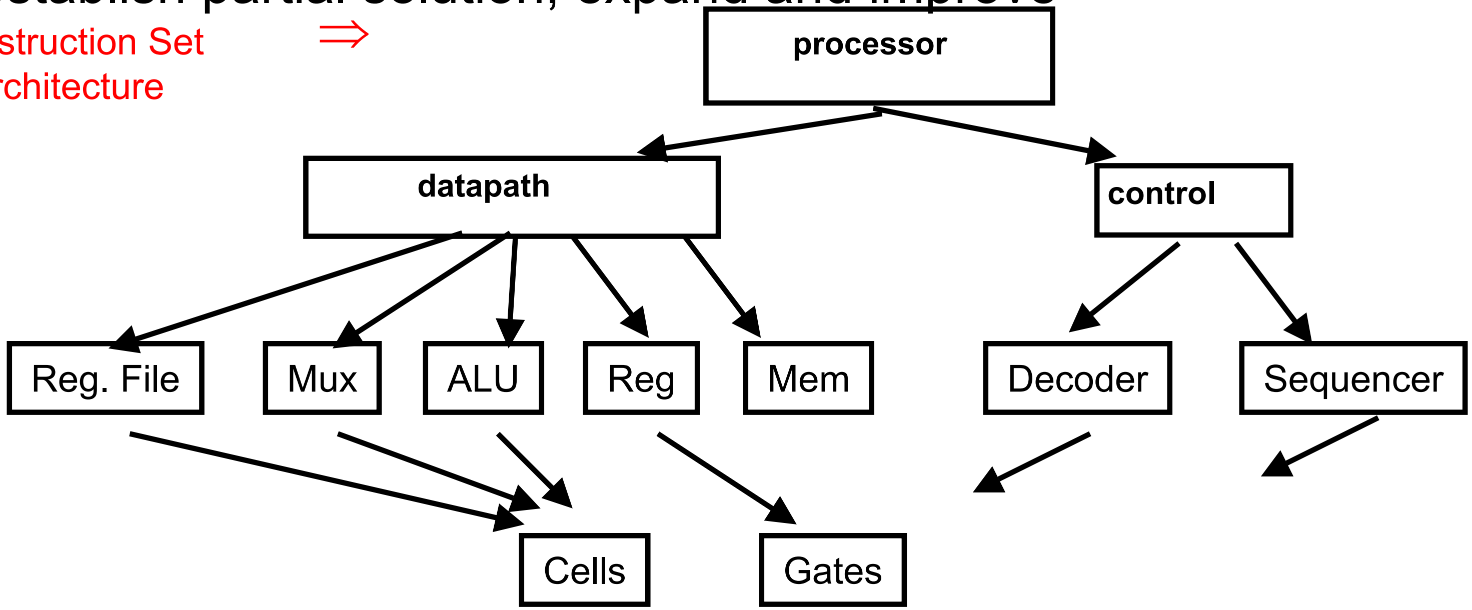
Review

- Synchronous circuit: from clock edge to clock edge, just define what happens in between; Flip flop defined to handle conditions
- Combinational logic has no clock
- Always statements create latches if you don't specify all output for all conditions
- Verilog does not turn hardware design into writing programs; describe your HW design
- Control implementation: turn truth tables into logic equations

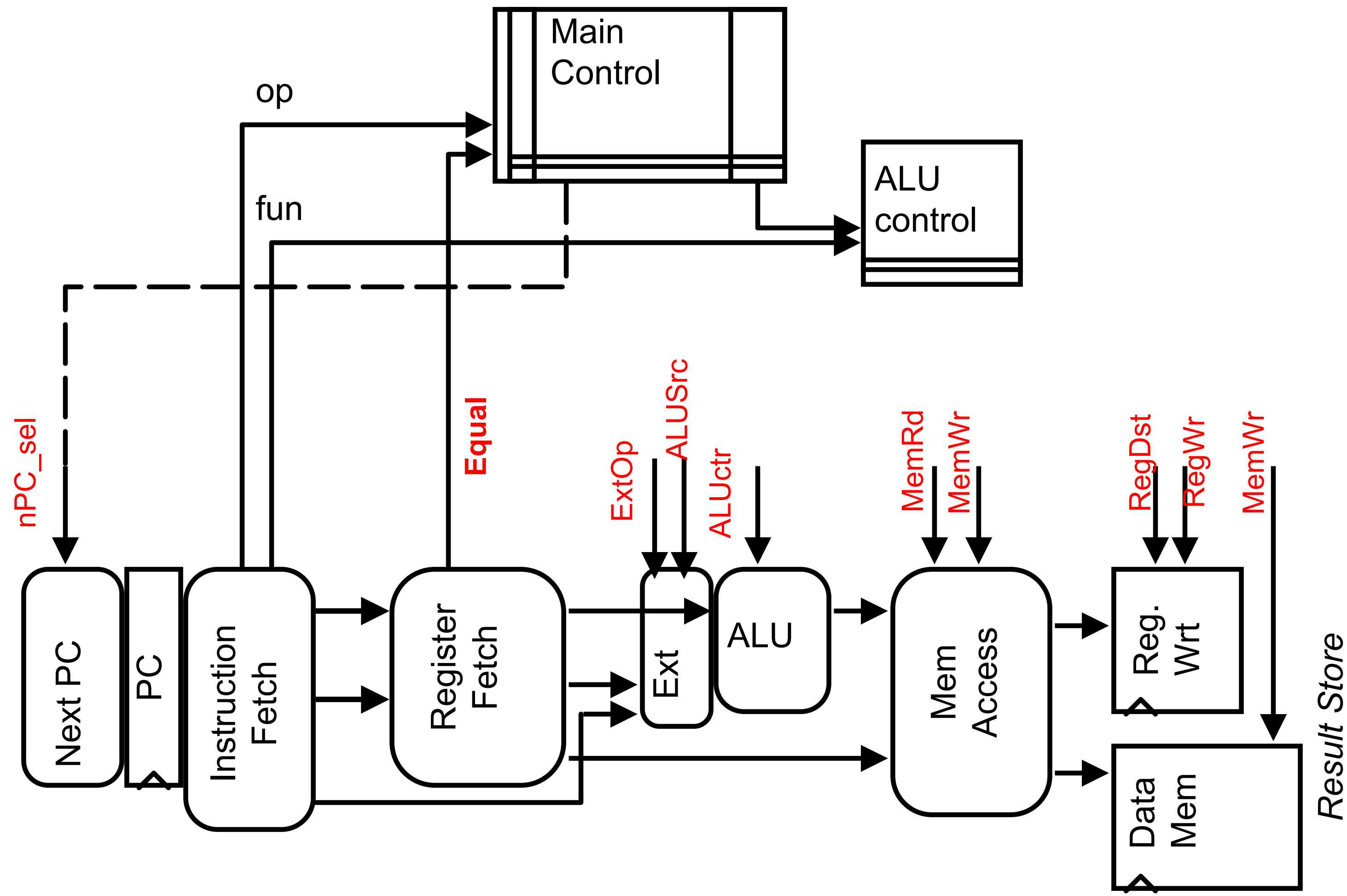
Recap: Processor Design is a Process

- Bottom-up
  - assemble components in target technology to establish critical timing
- Top-down
  - specify component behavior from high-level requirements
- Iterative refinement
  - establish partial solution, expand and improve

Instruction Set Architecture ⇒

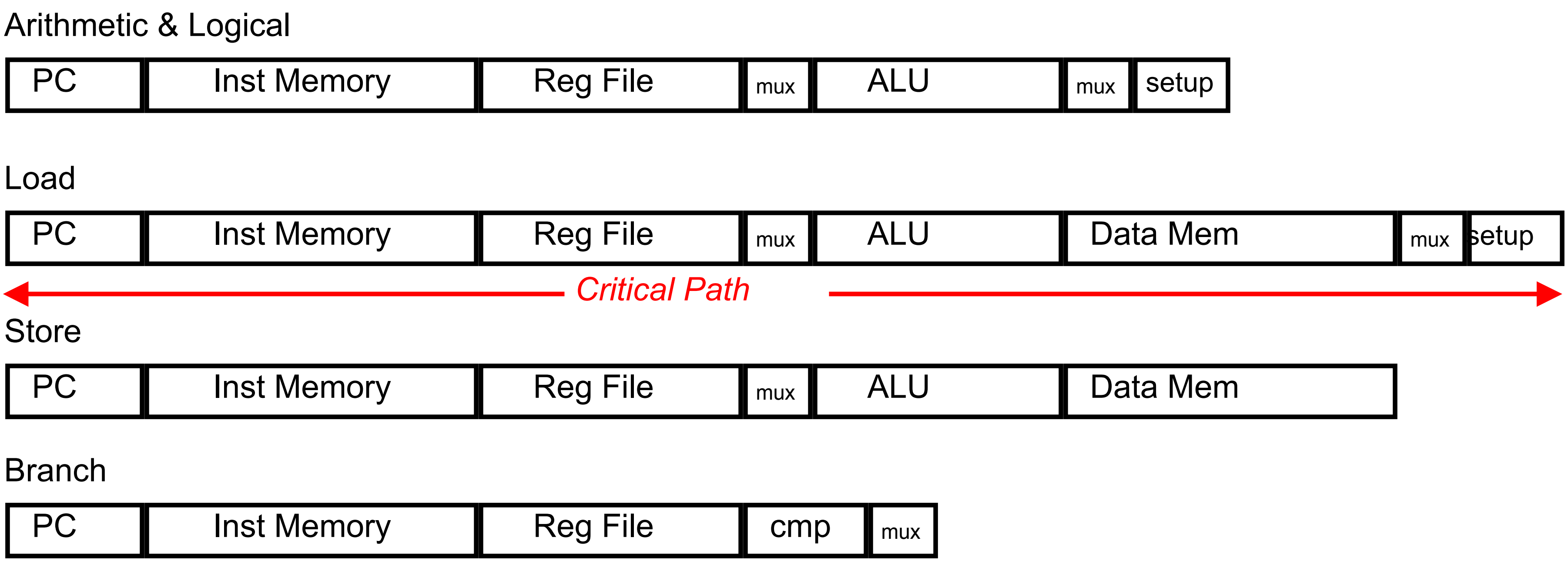


Abstract View of our single cycle processor



- looks like a FSM with PC as state

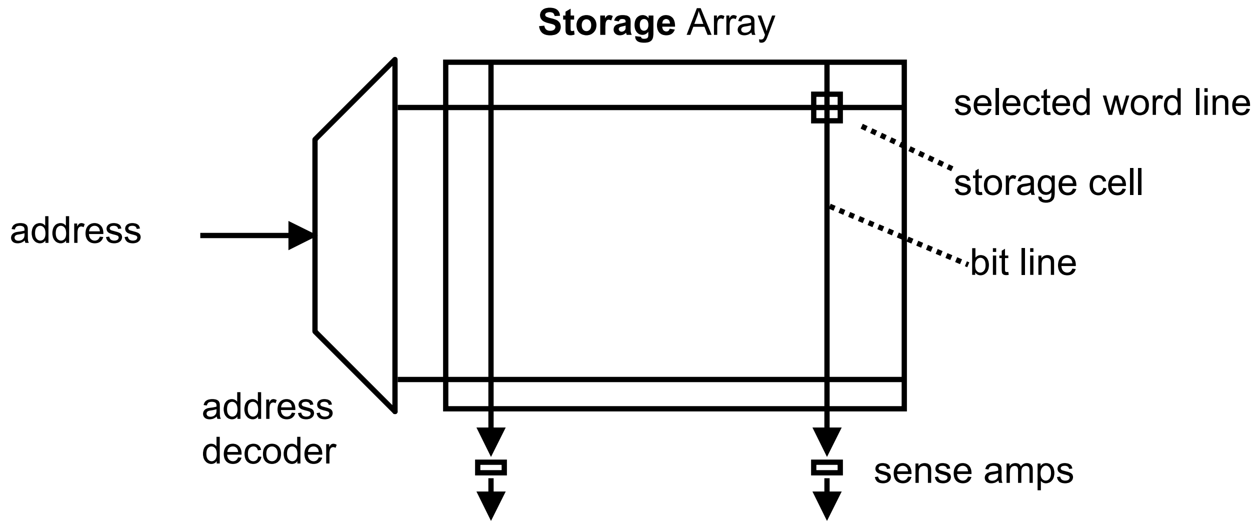
What's wrong with our CPI=1 processor?



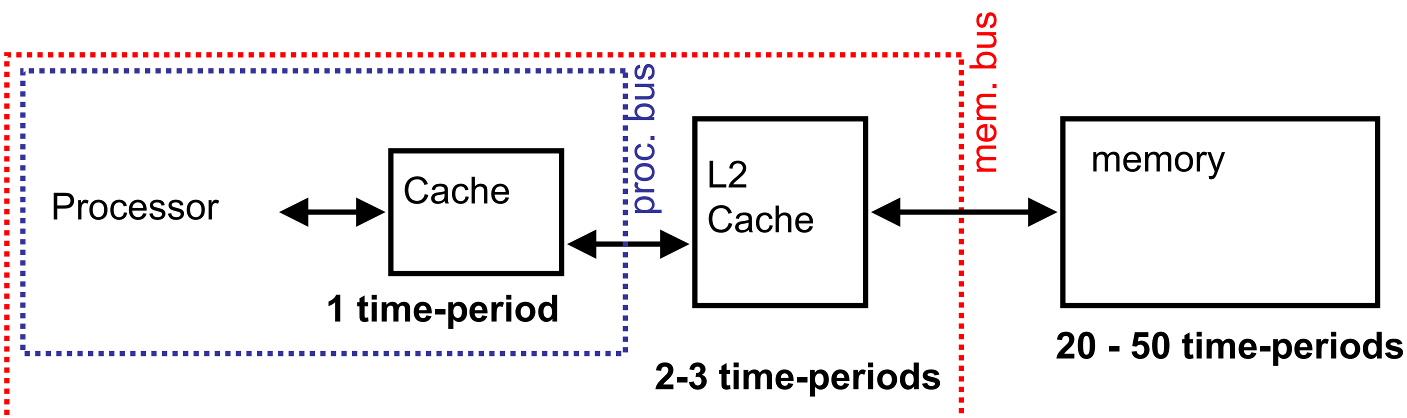
- Long Cycle Time
- All instructions take as much time as the slowest
- Real memory is not as nice as our idealized memory
  - cannot always get the job done in one (short) cycle

Memory Access Time

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

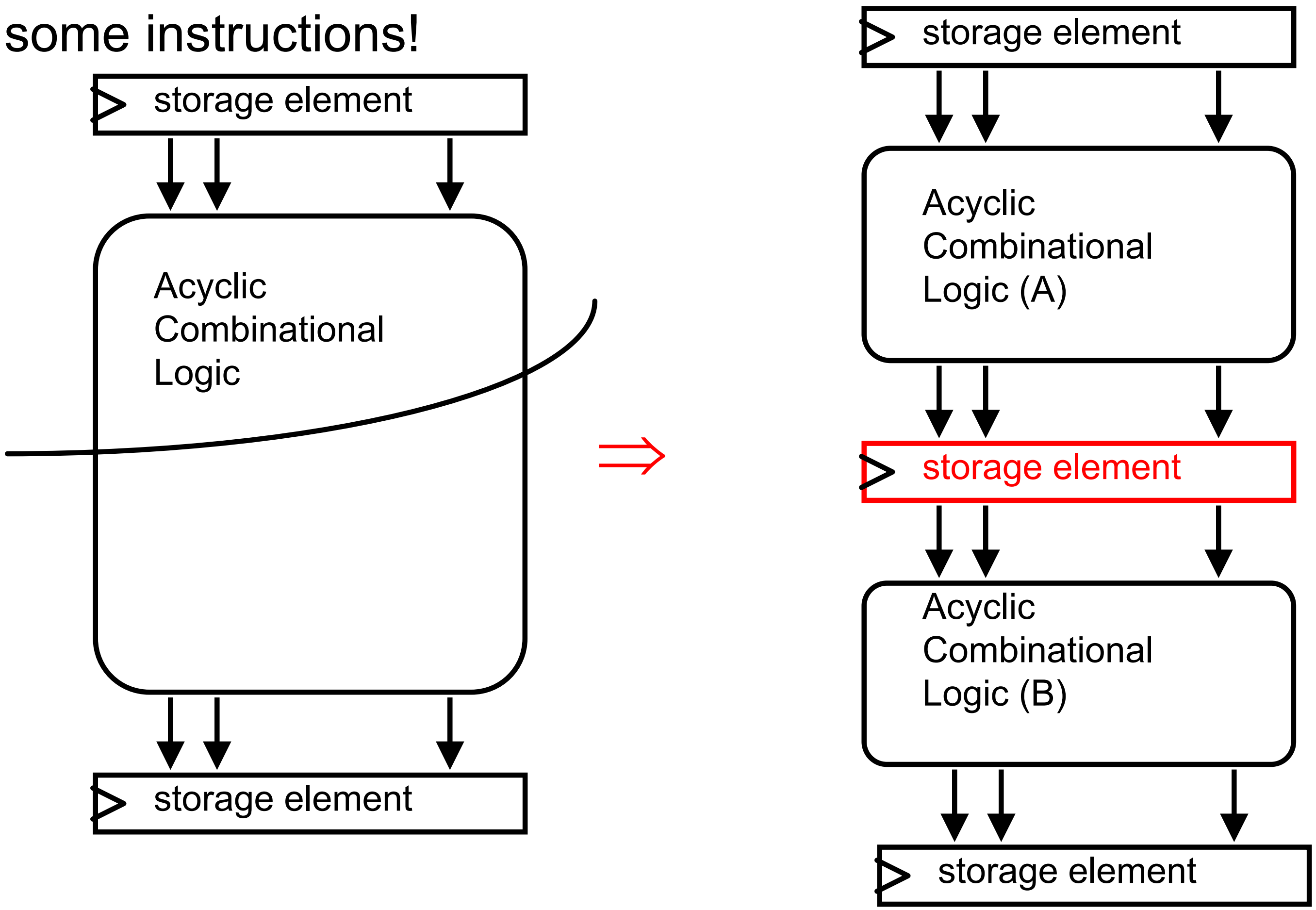


- => Use a hierarchy of memories

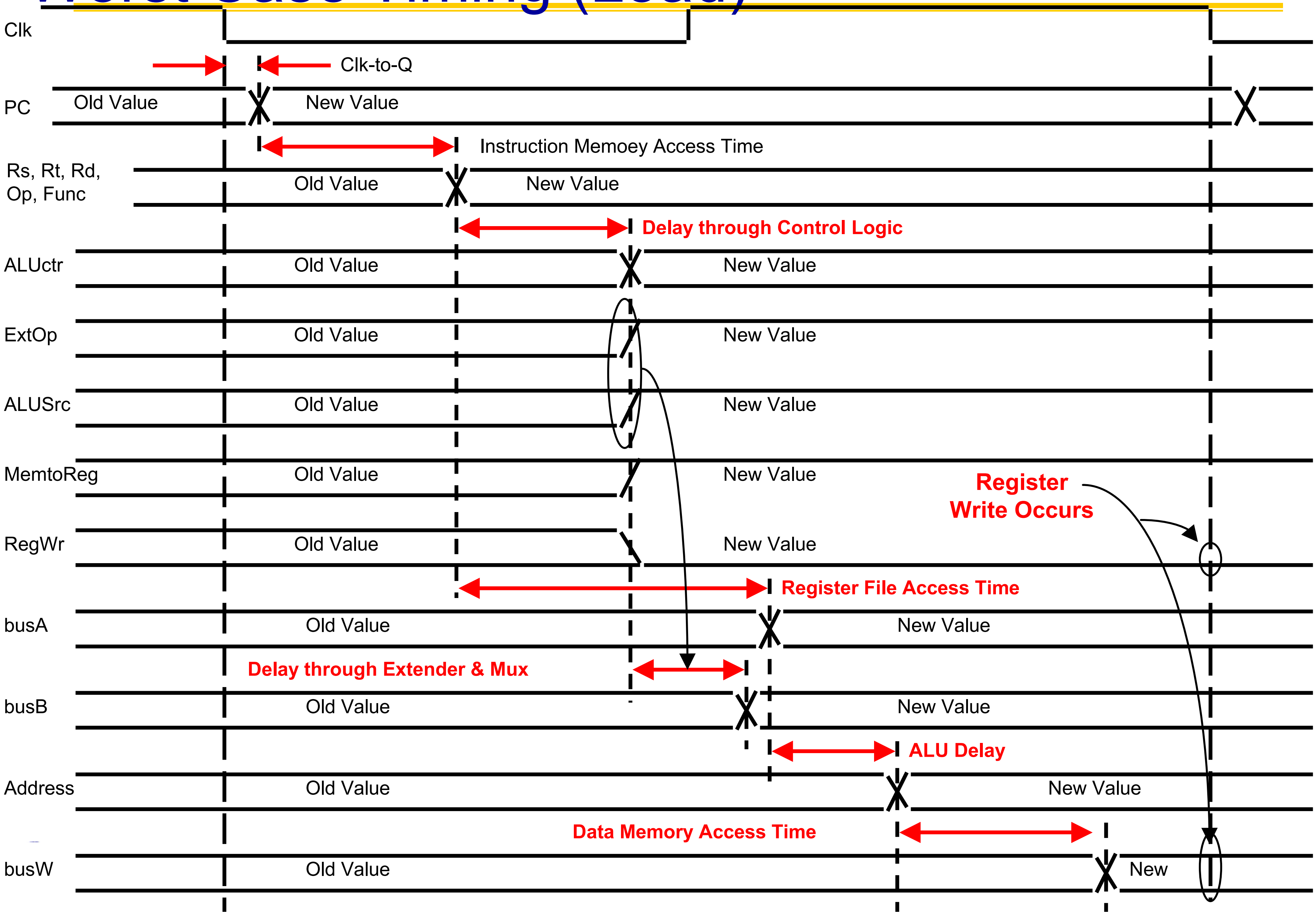


Reducing Cycle Time

- 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!

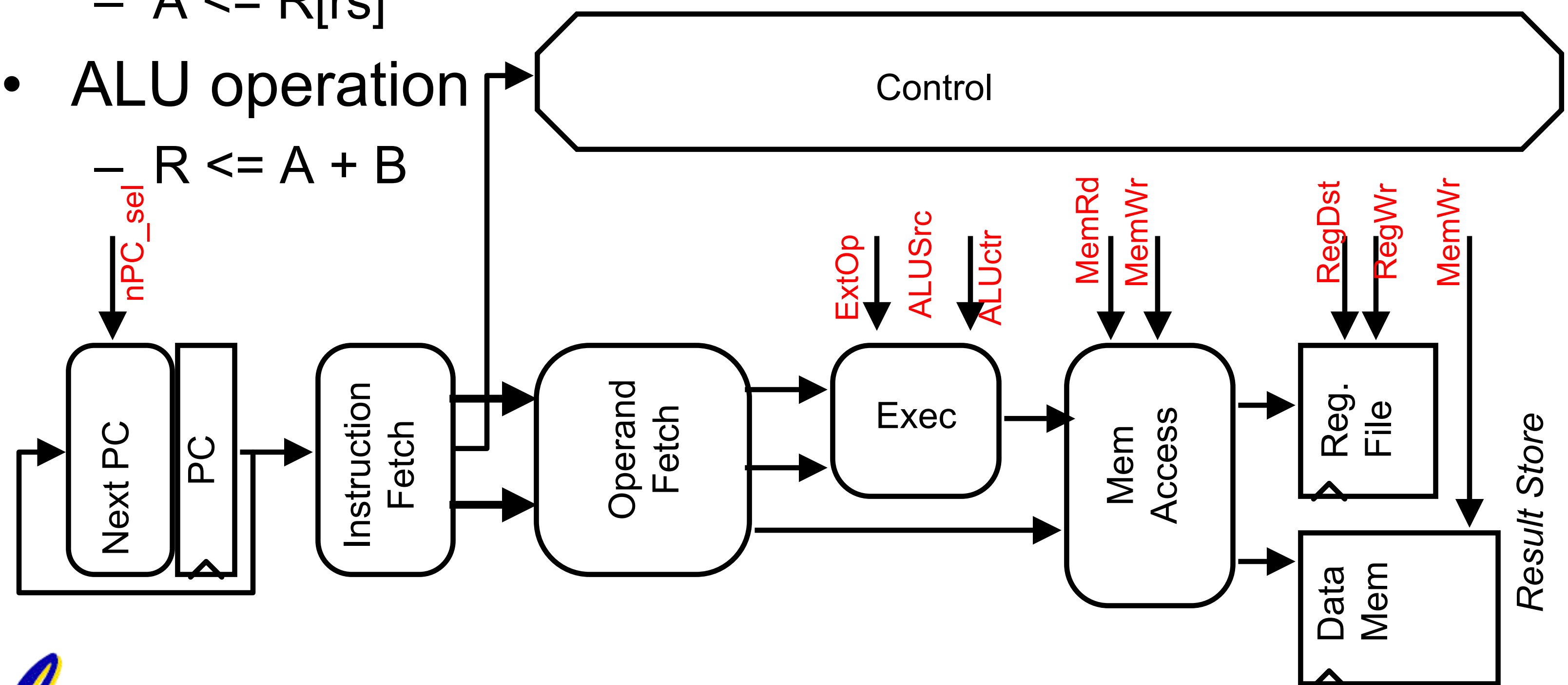


Worst Case Timing (Load)



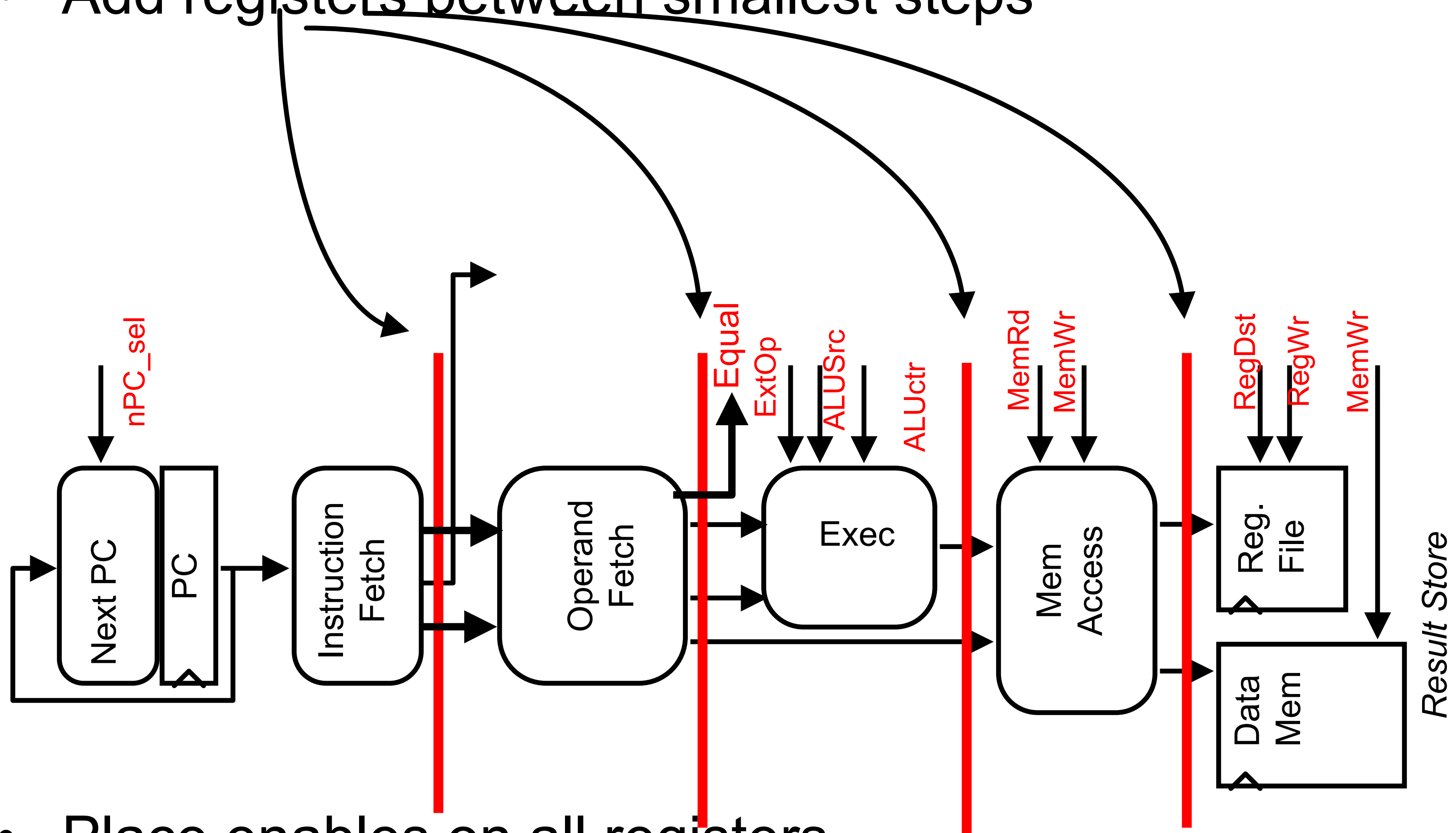
Basic Limits on Cycle Time

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



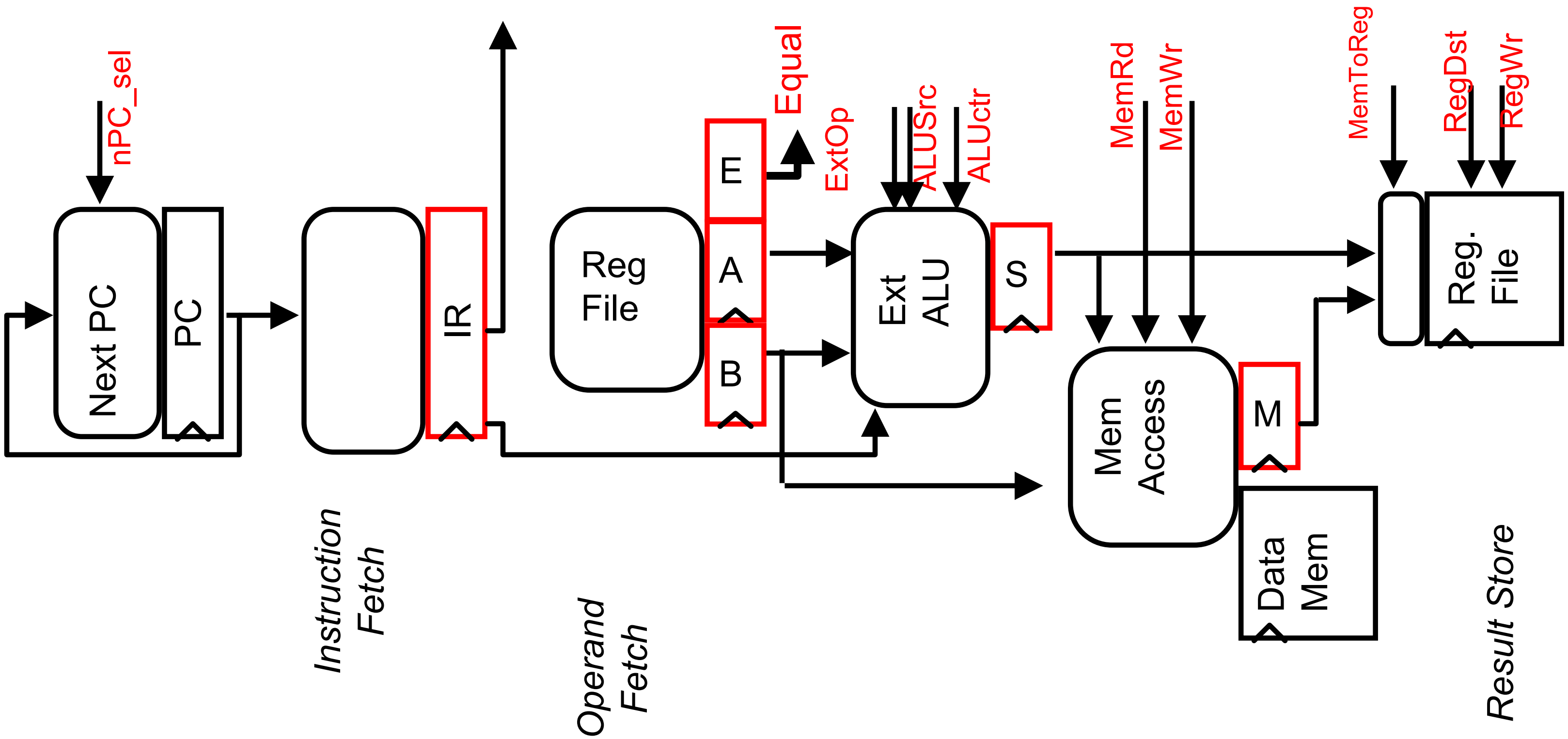
Partitioning the CPI=1 Datapath

- Add registers between smallest steps



- Place enables on all registers

Example Multicycle Datapath



- Critical Path ?

Administrivia

- Office hours in Lab
  - Mon 4 – 5:30 Jack, Tue 3:30-5 Kurt, Wed 3 – 4:30 John, Thu 3:30-5 Ben
- Dave's office hours Tue 3:30 – 5
- Lab 3 demo Friday, due Monday
- Midterm I Wednesday Oct 8 5:30 - 8:30pm

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, . . .)

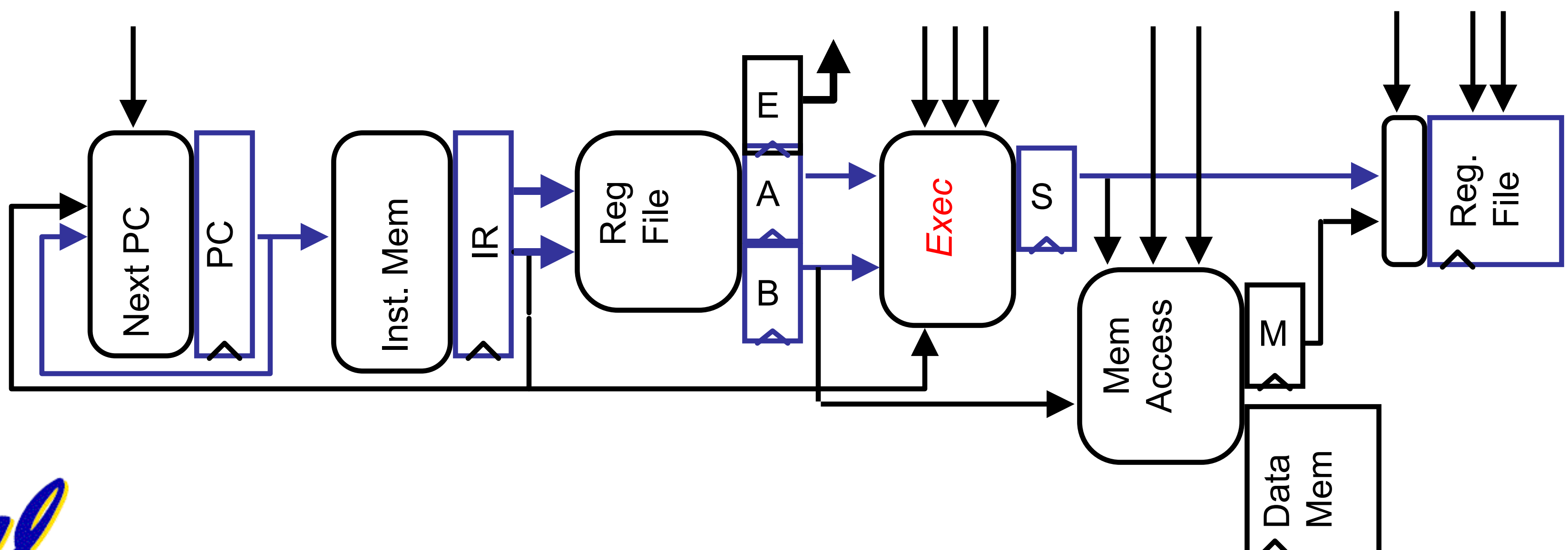
- Logical Register Transfer

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

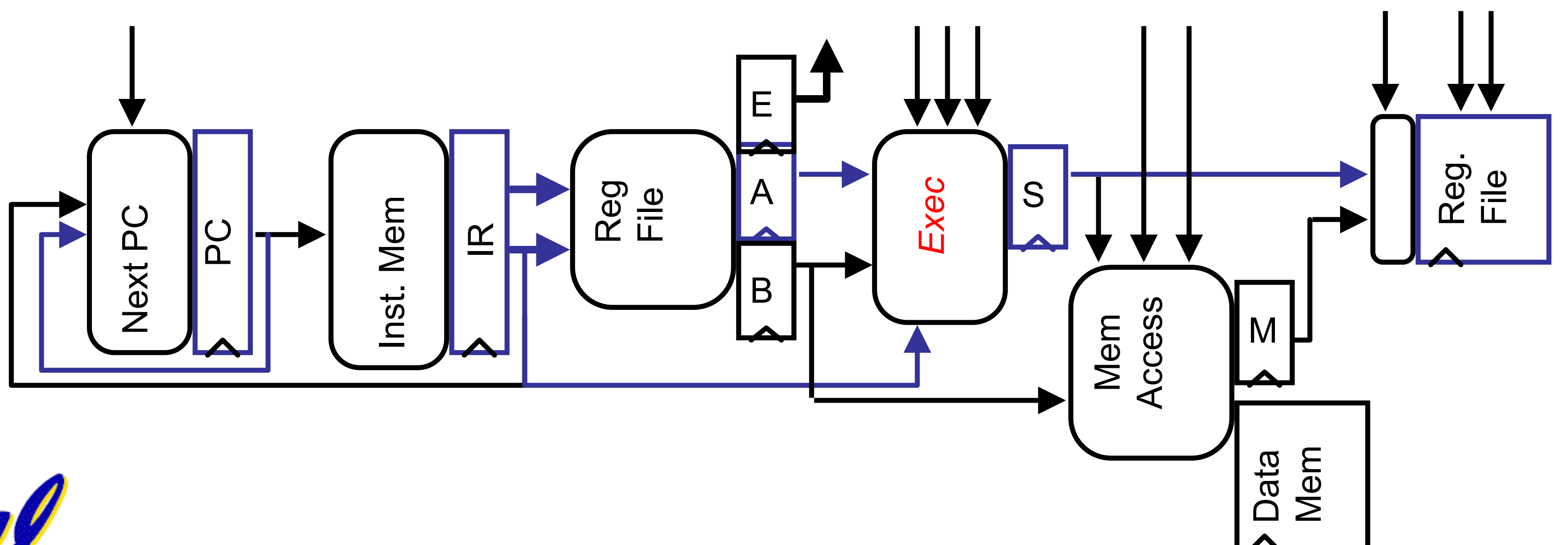
- Logical Register Transfer

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

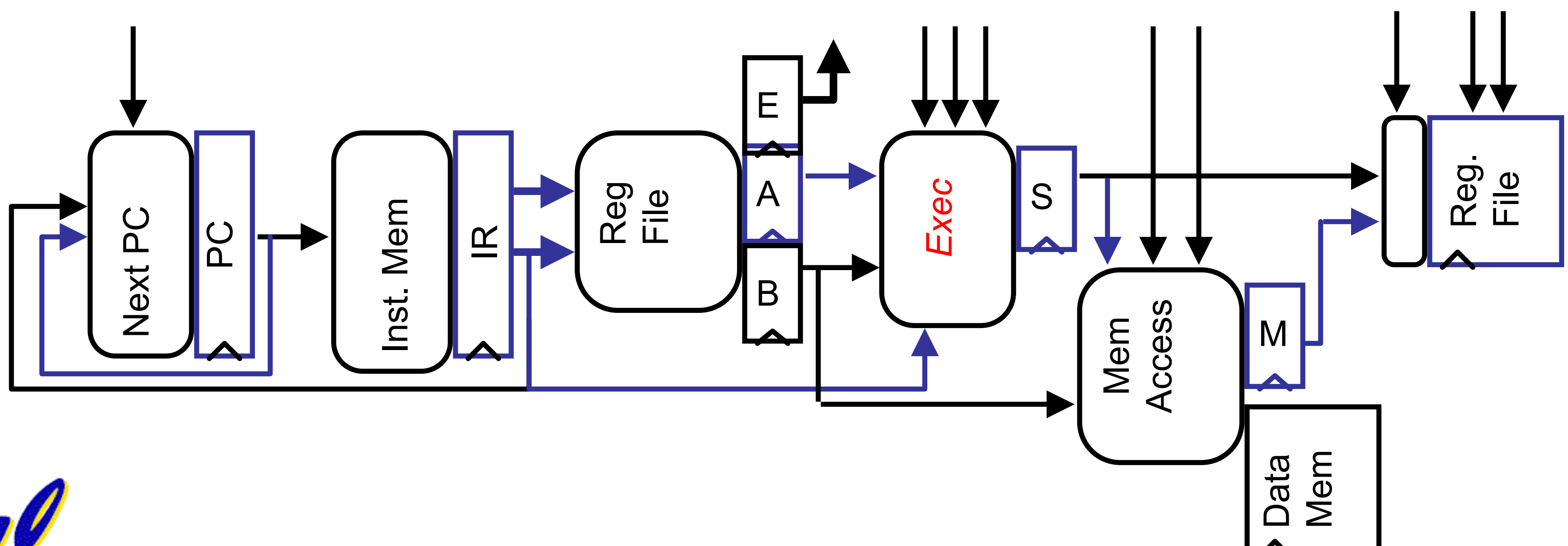
- Logical Register Transfer

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

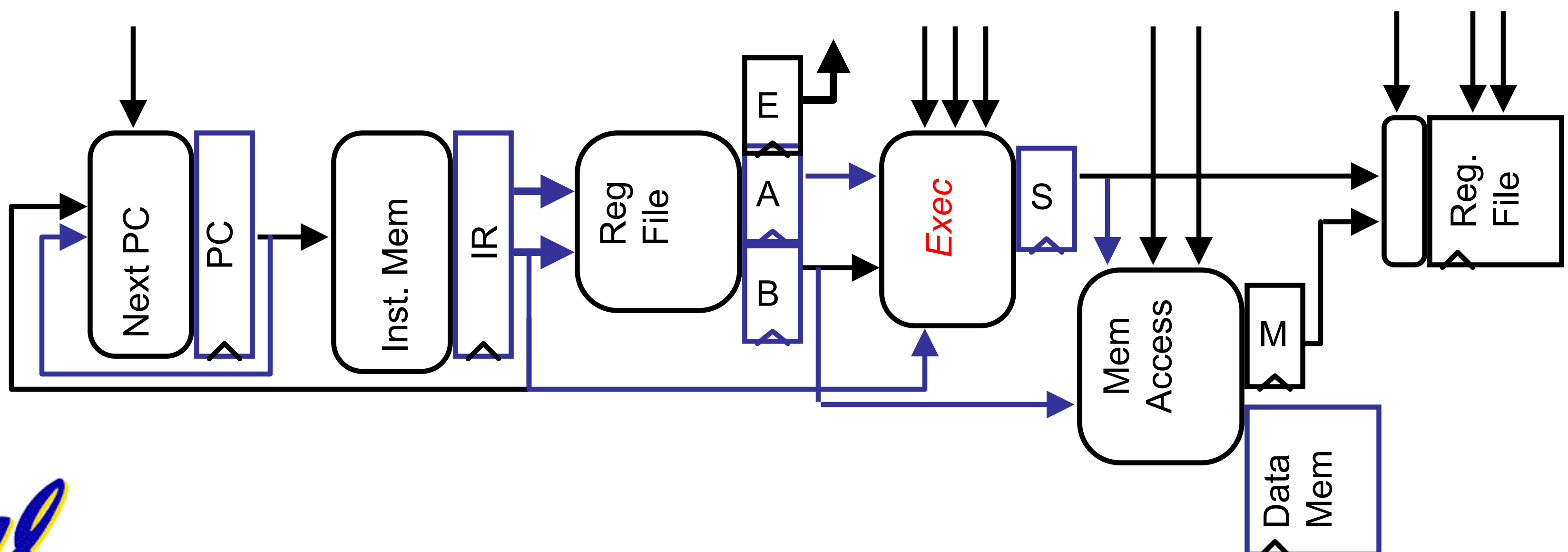
- Logical Register Transfer

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

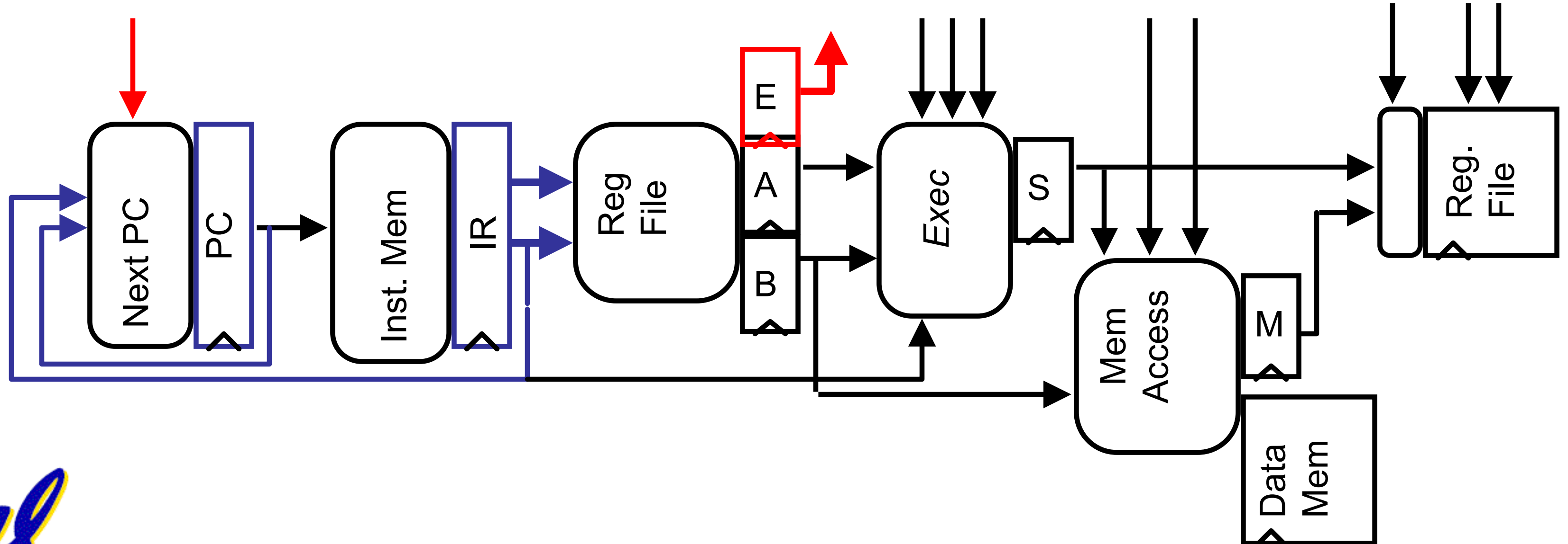
- Logical Register Transfer

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

- Physical Register Transfers

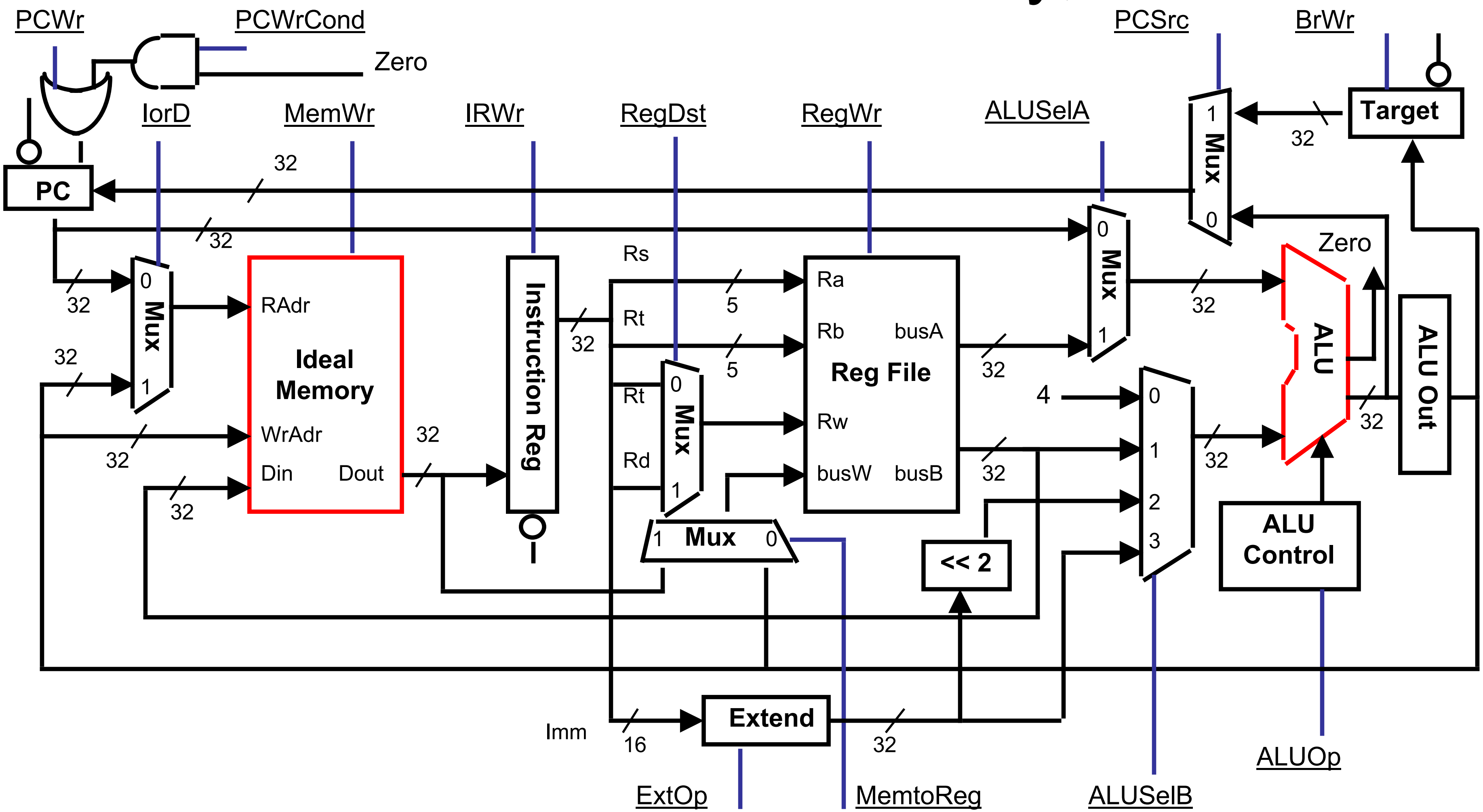
inst	Physical Register Transfers
	IR <= MEM[pc]
BEQ	E<= (R[rs] = R[rt])
	if (!E) PC <= PC + 4; else PC <=PC+4+{SExt(Im16),2b0}

Time



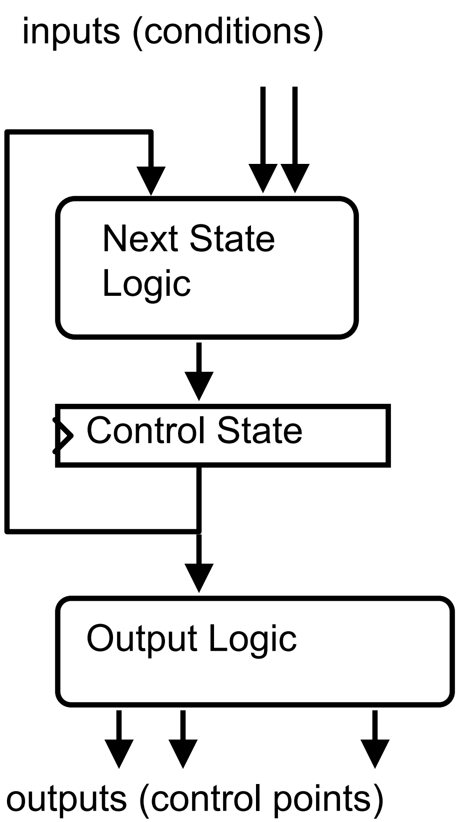
Alternative datapath (book): Multiple Cycle Datapath

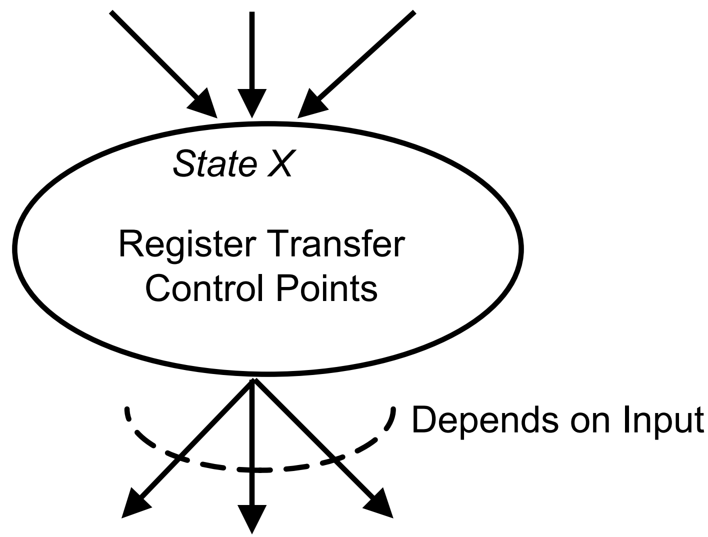
- Minimizes Hardware: 1 memory, 1 adder



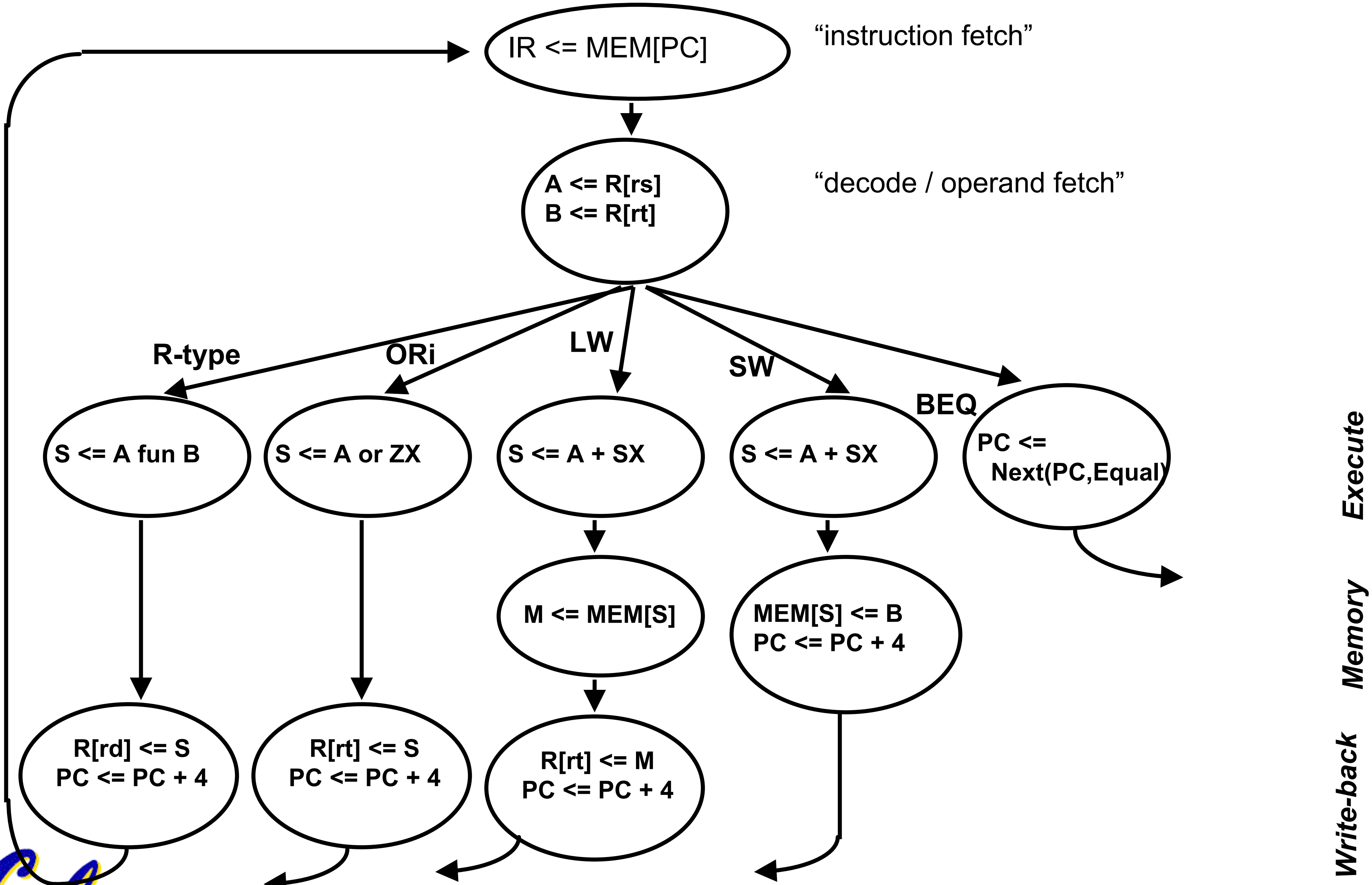
Our Control Model

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





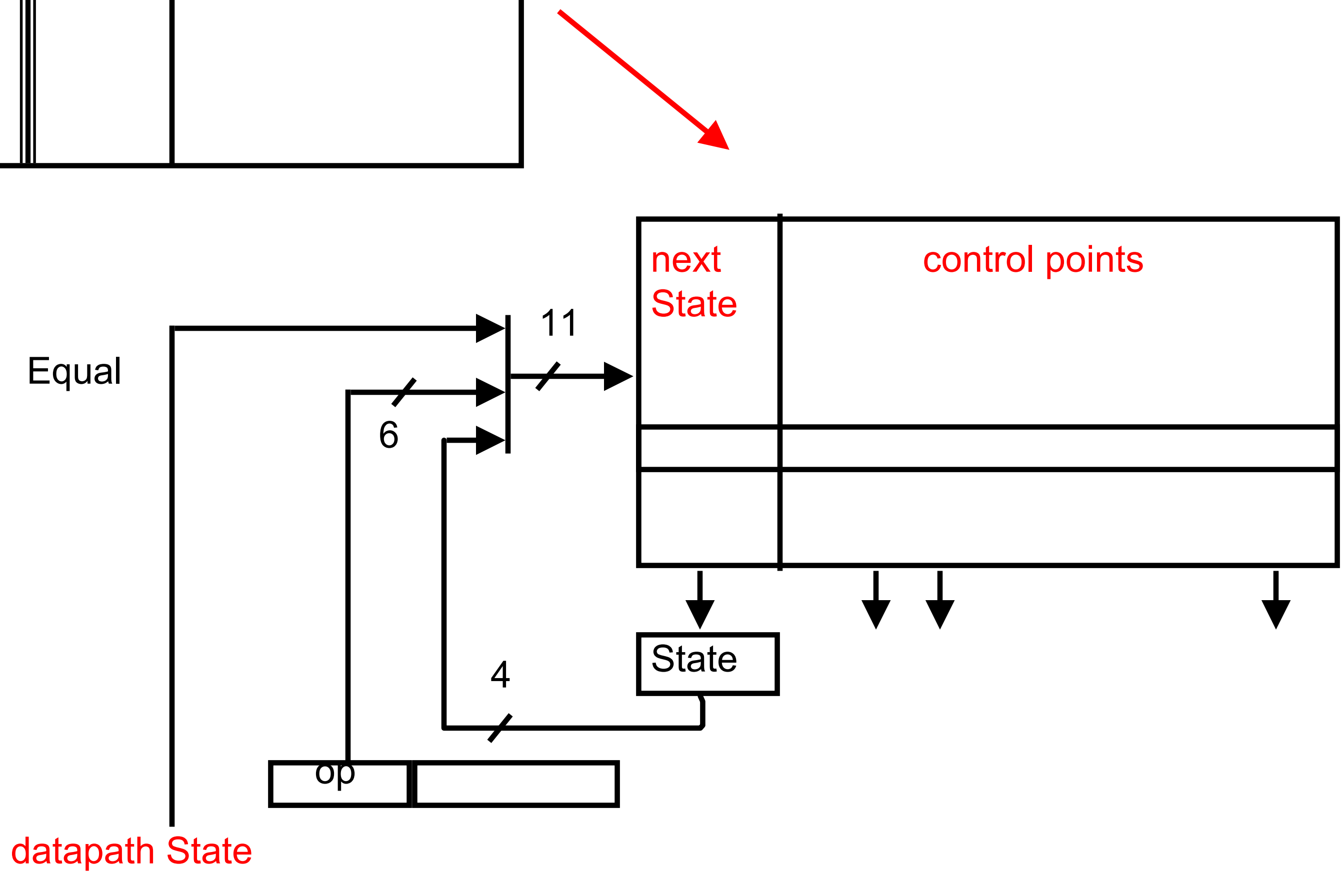
Step 4 ⇒ Control Spec for multicycle proc



Traditional FSM Controller

state	op	cond	next state	control points

Truth Table

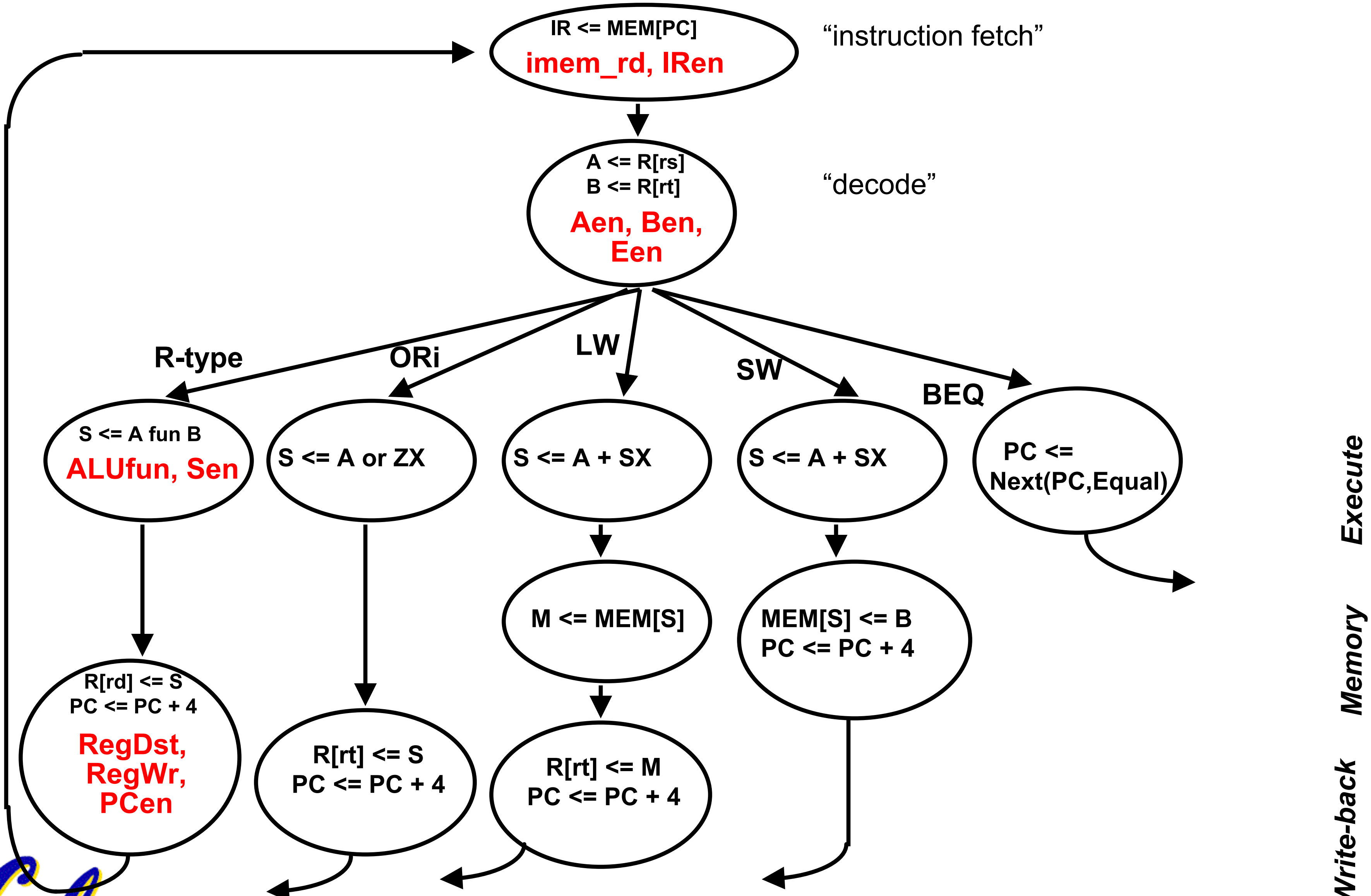


Step 5 ⇒ (datapath + state diagram ⇒ control)

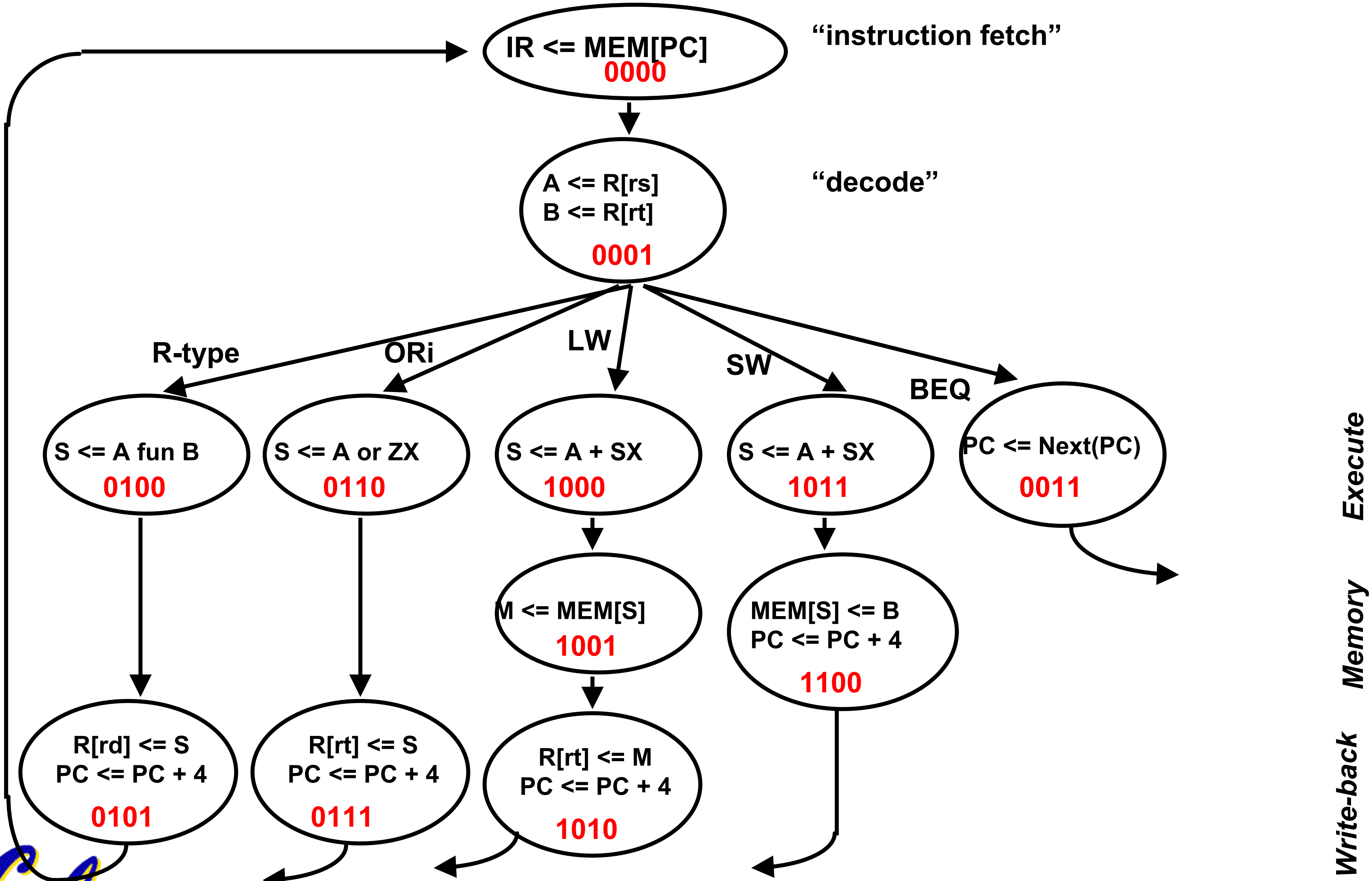
- Translate RTs into control points
- Assign states

- Then go build the controller

Mapping Register Transfers to Control Points



Assigning States



(Mostly) Detailed Control Specs (missing⇒0)

	State	Op field	Eq	Next	IR	PC en sel	Ops A B E	Exec Ex Sr ALU S	Mem R W M	Write-Back M-R Wr 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

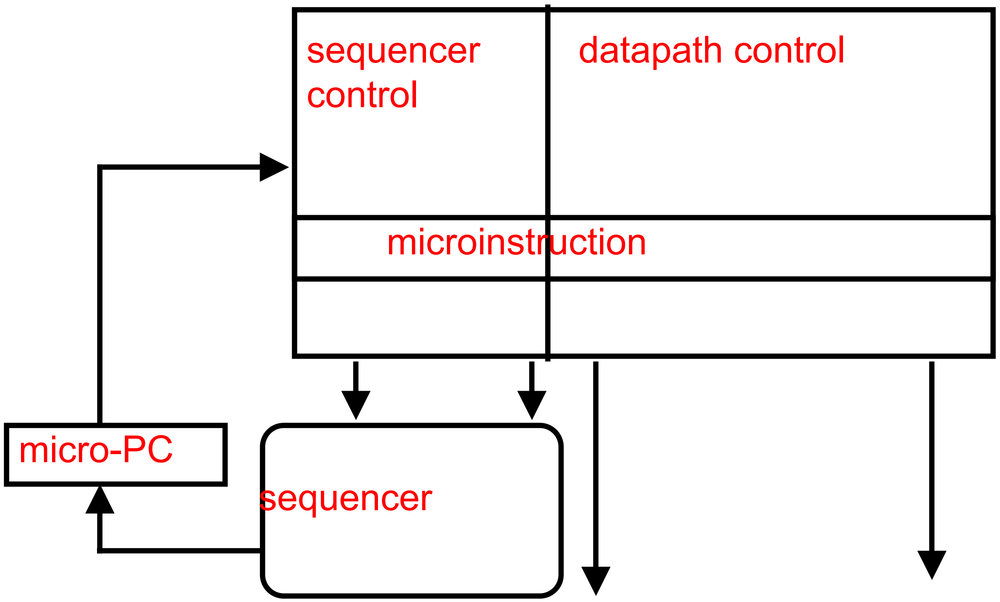
- What is the average CPI?
  - state diagram gives CPI for each instruction type
  - workload gives frequency of each type

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

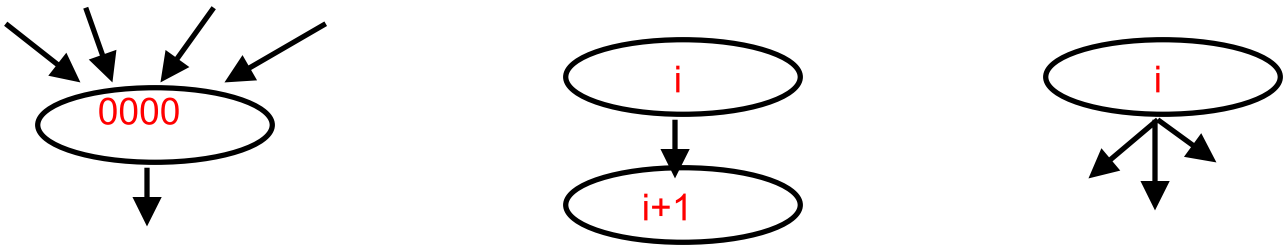
Controller Design

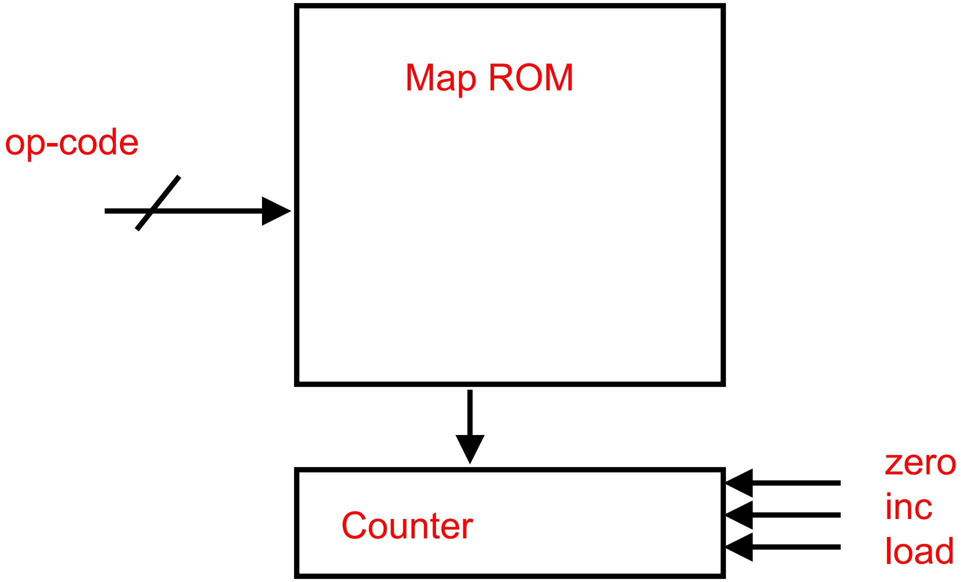
- The state diagrams that arise define the controller for an instruction set processor are highly structured
- Use this structure to construct a simple “microsequencer”
- Control reduces to programming this very simple device

  ⇒ microprogramming



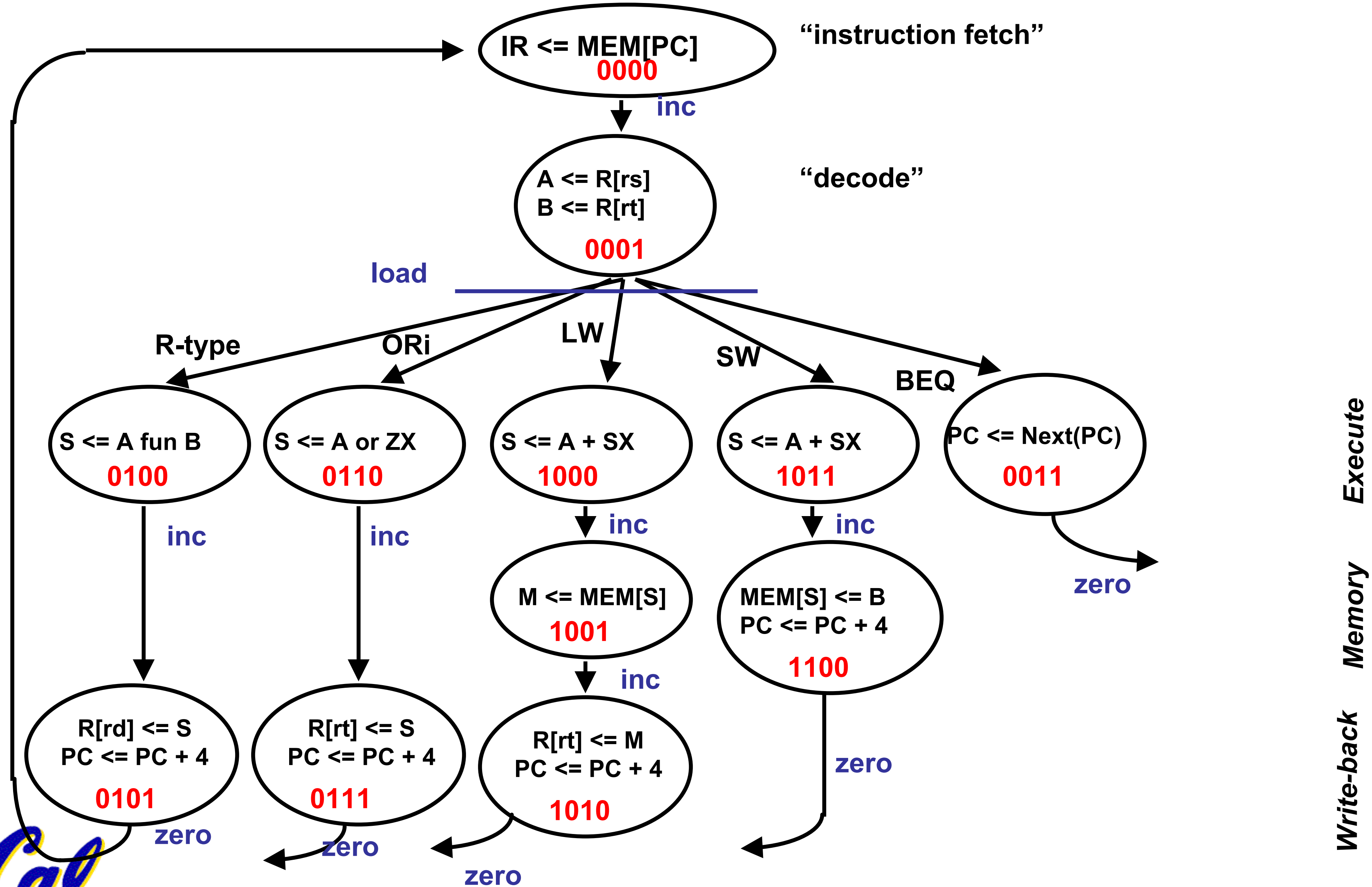
Example: Jump-Counter



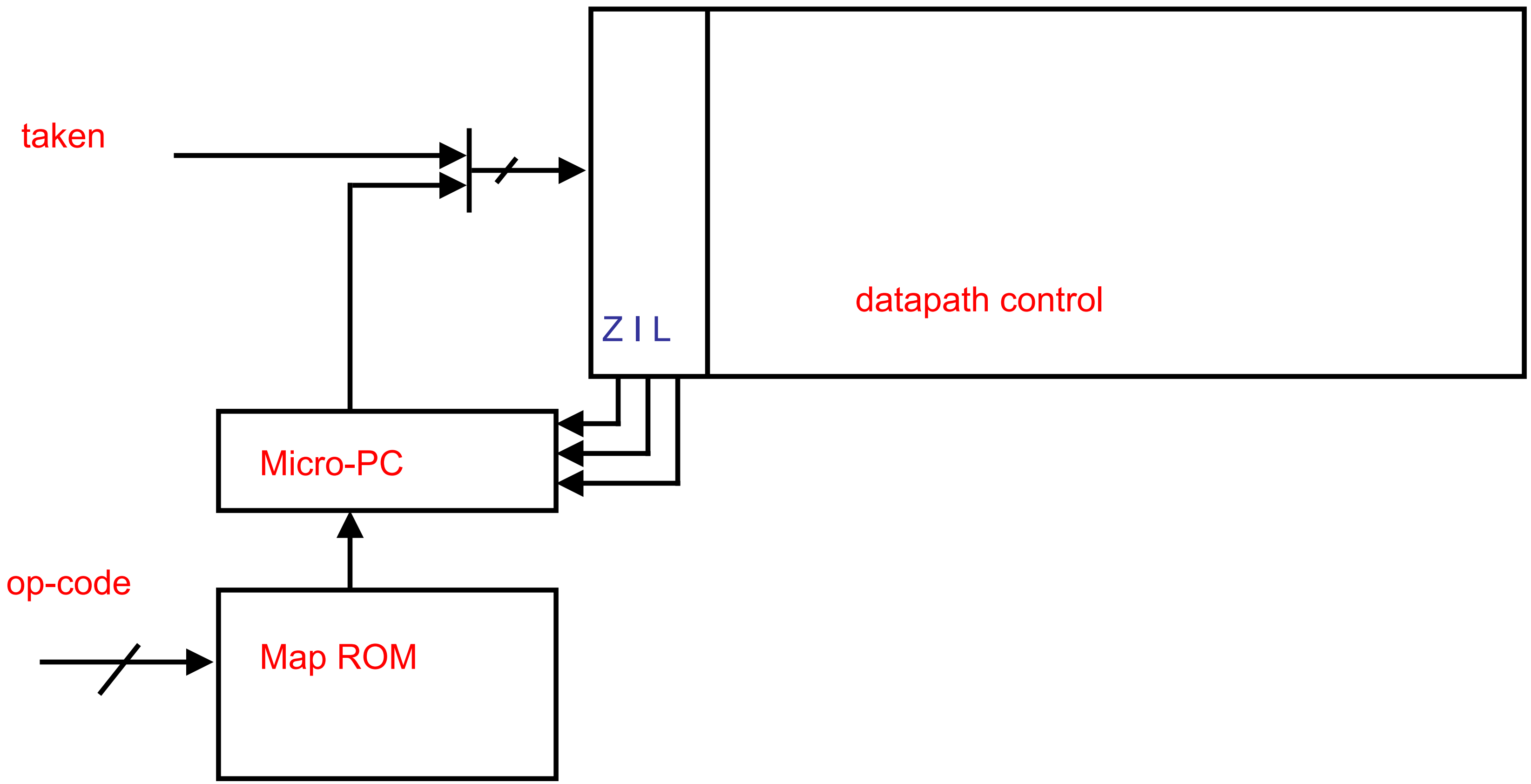


None of above: Do nothing (for wait states)

Using a Jump Counter



Our Microsequencer



Microprogram Control Specification

	uPC	Taken	Next	IR	PC	Ops en sel	Exec A B	Mem Ex Sr ALU S	Write-Back R W M	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	

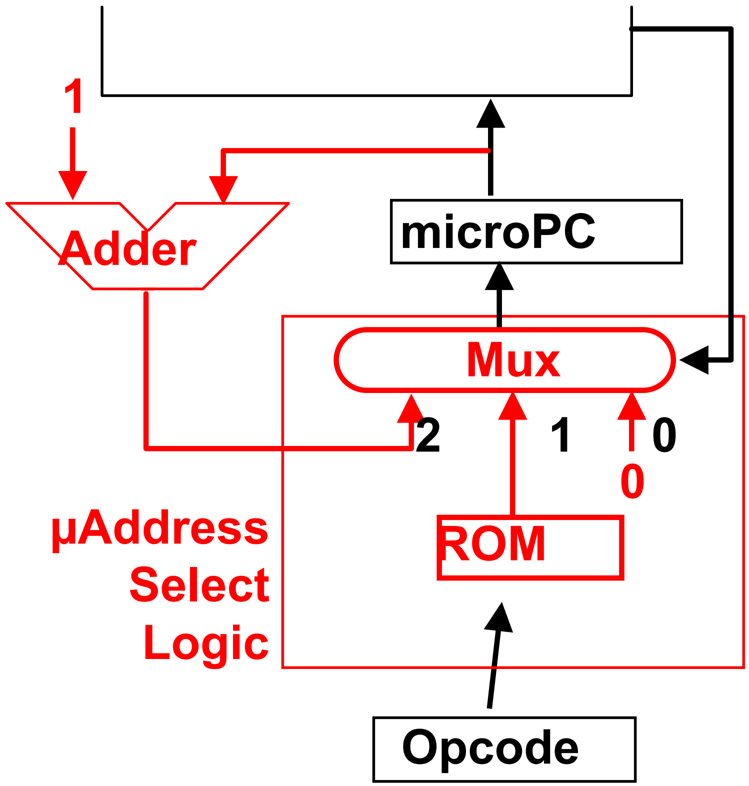
Adding the Dispatch ROM

- Sequencer-based control
  - Called "microPC" or "µPC" vs. state register

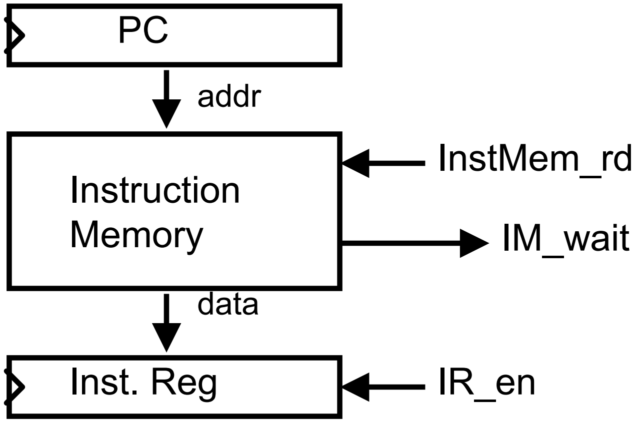
Control Value	*Effect*
00	Next µaddress = 0
01	Next µaddress = dispatch ROM
10	Next µaddress = µaddress + 1

ROM:

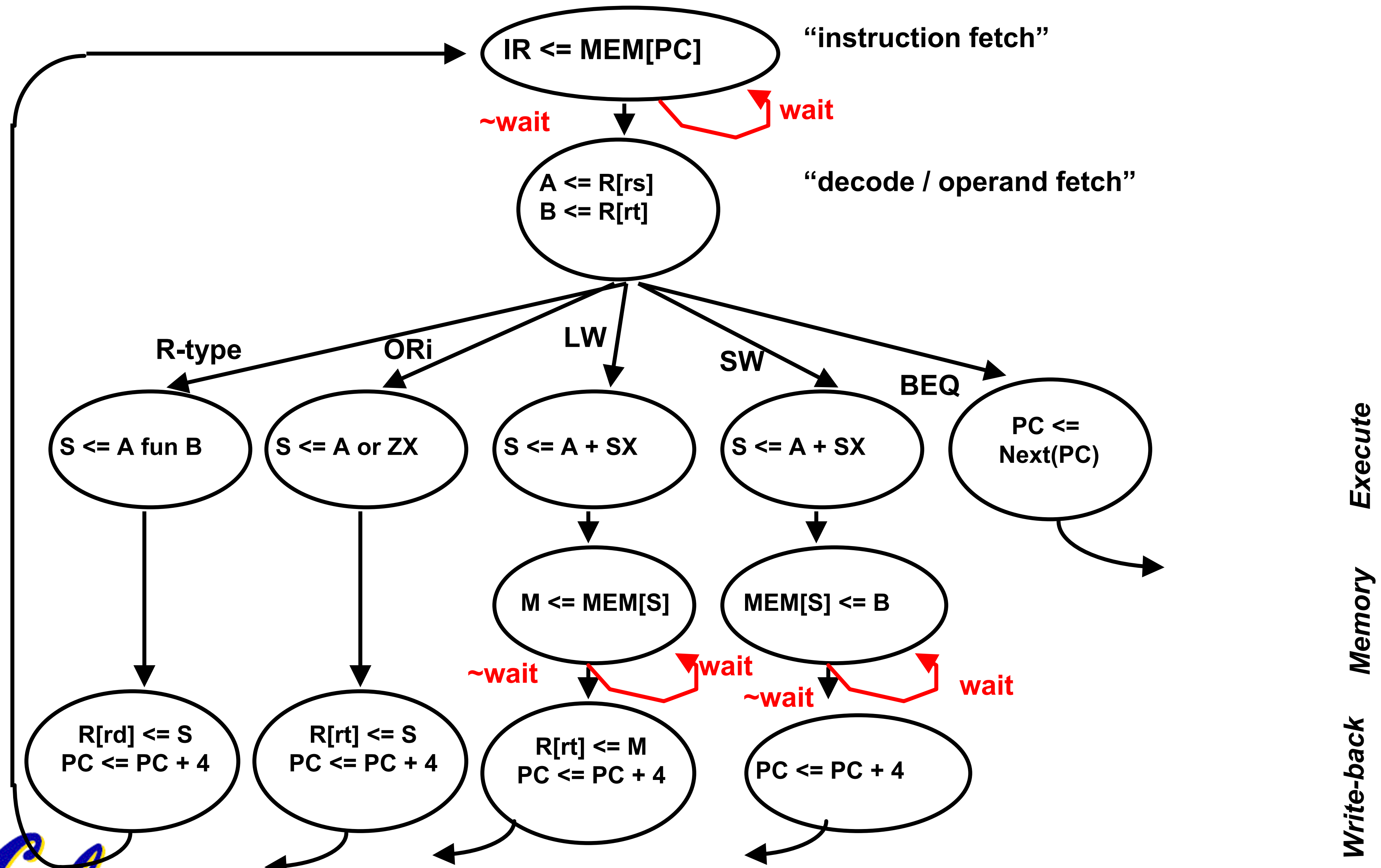
R-type	000000	0100
BEQ	000100	0011
ori	001101	0110
LW	100011	1000
SW	101011	1011



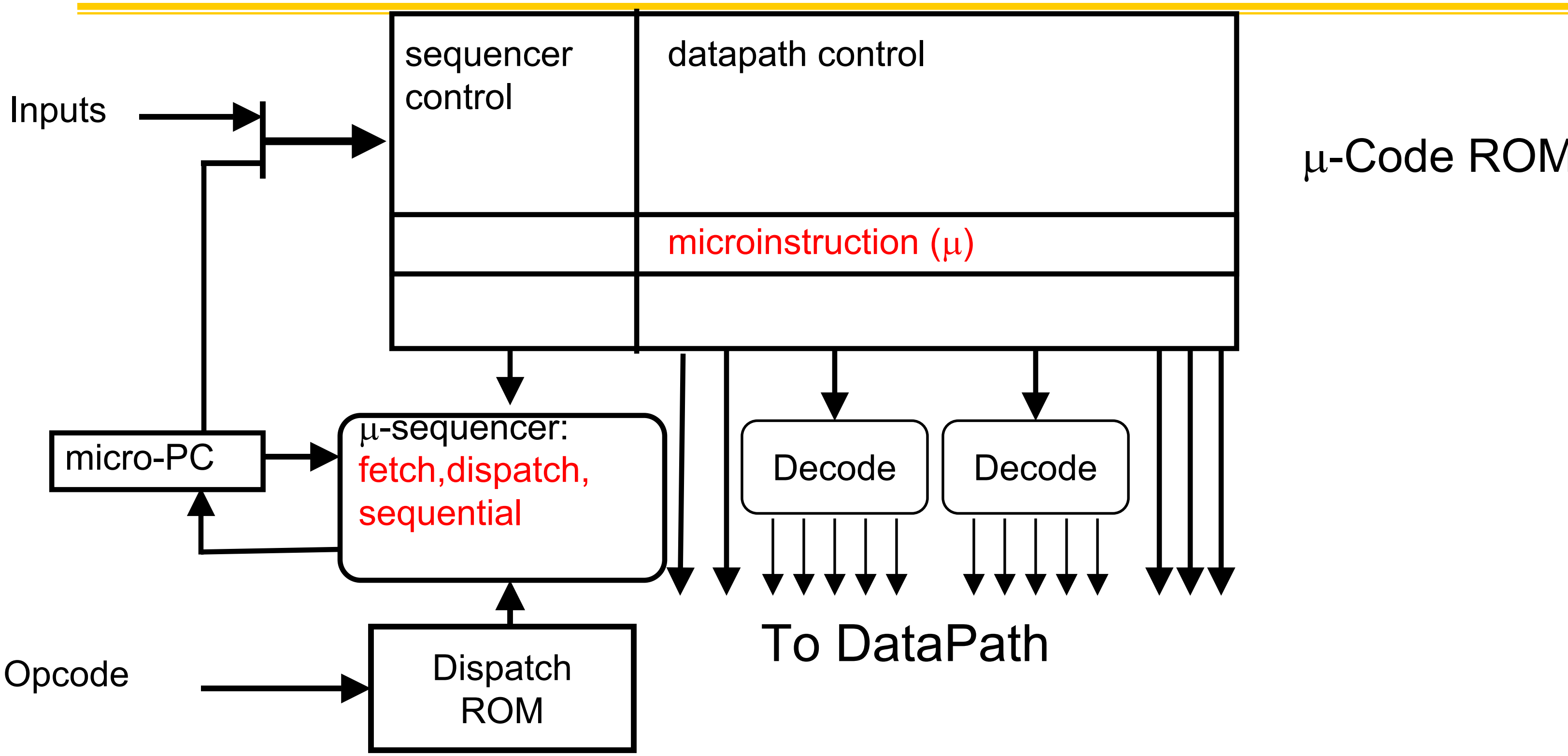
Example: Controlling Memory



Controller handles non-ideal memory



Microprogramming

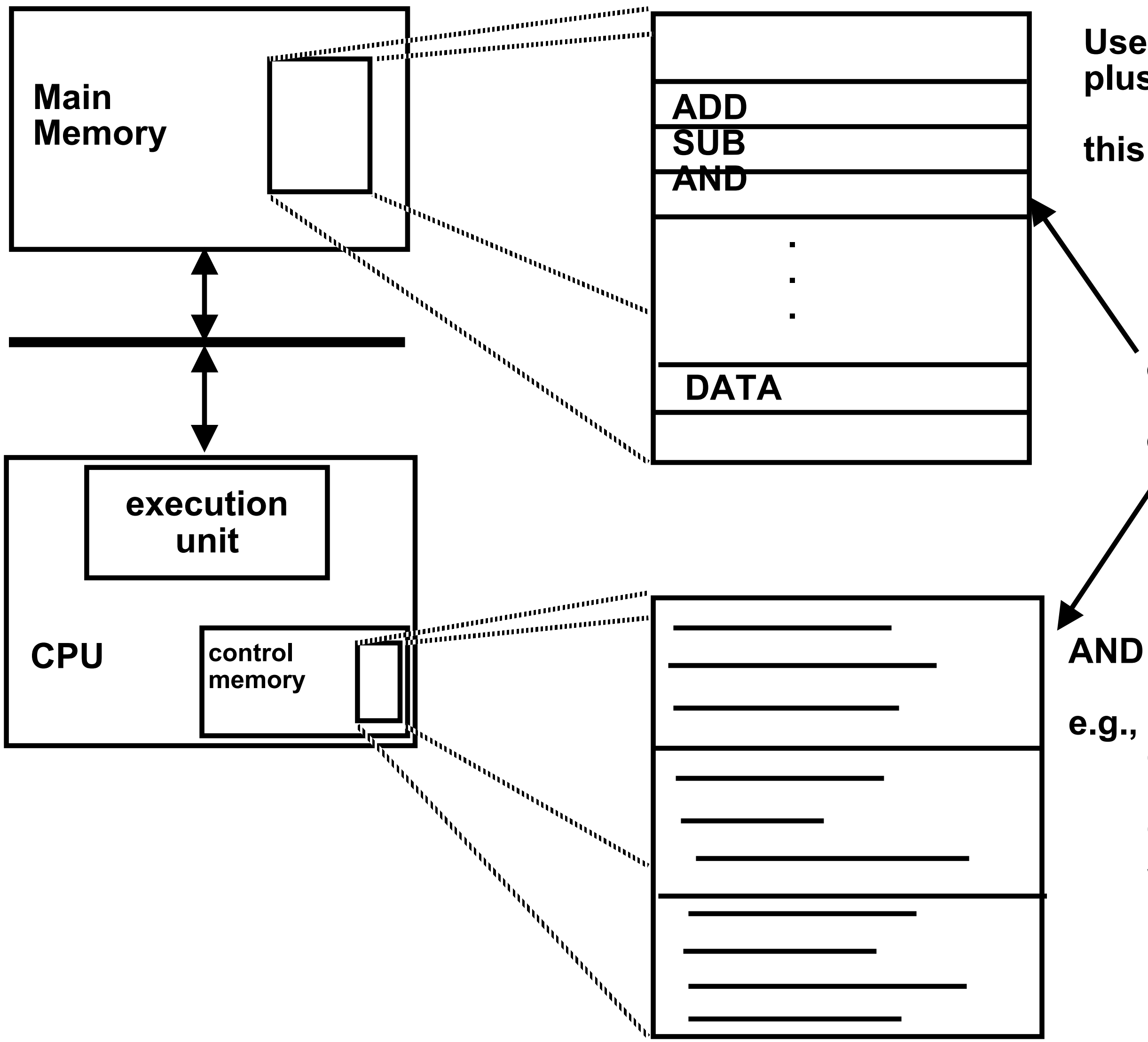


- Microprogramming is a fundamental concept
  - implement an instruction set by building a very simple processor and interpreting the instructions
  - essential for very complex instructions and when few register transfers are possible
  - overkill when ISA matches datapath 1-1

Microprogramming

- Microprogramming is a convenient method for implementing *structured* control state diagrams:
  - Random logic replaced by microPC sequencer and ROM
  - Each line of ROM called a μinstruction:
    contains sequencer control + values for control points
  - limited state transitions:
    branch to zero, next sequential,
    branch to μinstruction address from displatch ROM
- Horizontal μCode: one control bit in μInstruction for every control line in datapath
- Vertical μCode: groups of control-lines coded together in μInstruction (e.g. possible ALU dest)
- Control design reduces to Microprogramming
  - Part of the design process is to develop a “language” that describes control and is easy for humans to understand

“Macroinstruction” Interpretation



User program
plus Data

this can change!

one of these is
mapped into one
of these

AND microsequence

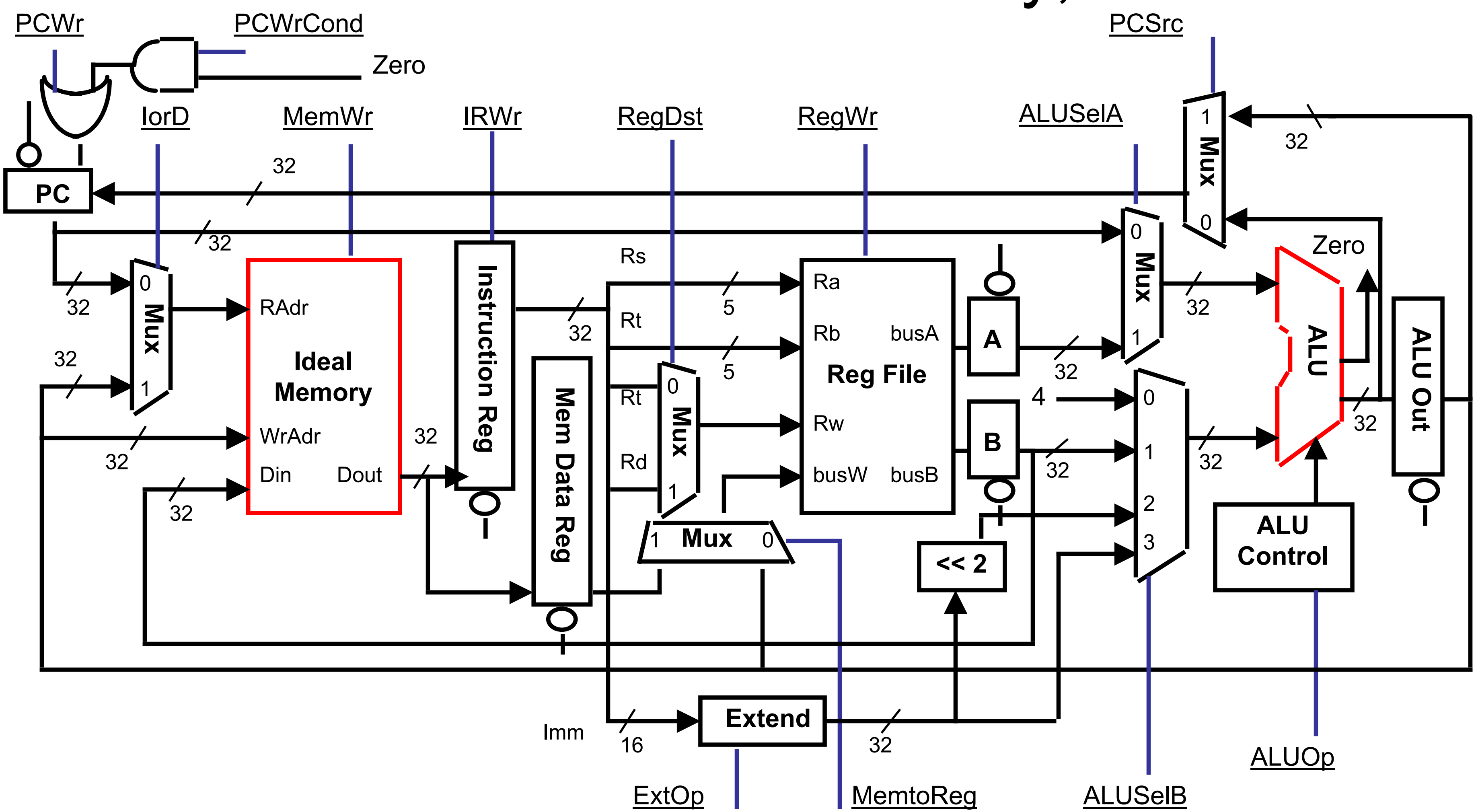
e.g., Fetch
Calc Operand Addr
Fetch Operand(s)
Calculate
Save Answer(s)

Designing a Microinstruction Set

1) Start with list of control signals
2) Group signals together that make sense (vs. random): called "fields"
3) Place fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)
4) To minimize the width, encode operations that will never be used at the same time
5) Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals
    – Use computers to design computers

Again: Alternative multicycle datapath (book)

- Miminizes Hardware: 1 memory, 1 adder



1&2) Start with list of control signals, grouped into fields

Single Bit Control

Signal name	*Effect when deasserted*	*Effect when asserted*
ALUSelA	1st ALU operand = PC	1st ALU operand = Reg[rs]
RegWrite	None	Reg. is written
MemtoReg	Reg. write data input = ALU	Reg. write data input = memory
RegDst	Reg. dest. no. = rt	Reg. dest. no. = rd
MemRead	None	Memory at address is read, MDR <= Mem[addr]
MemWrite	None	Memory at address is written
IorD	Memory address = PC	Memory address = S
IRWrite	None	IR <= Memory
PCWrite	None	PC <= PCSource
PCWriteCond	None	IF ALUzero then PC <= PCSource
PCSource	PCSource = ALU	PCSource = ALUout
ExtOp	Zero Extended	Sign Extended

Multiple Bit Control

Signal name	***Value***	***Effect***
ALUOp	**00**	**ALU adds**
	01	**ALU subtracts**
	10	**ALU does function code**
	11	**ALU does logical OR**
ALUSelB	**00**	**2nd ALU input = 4**
	01	**2nd ALU input = Reg[rt]**
	10	**2nd ALU input = extended,shift left 2**
	11	**2nd ALU input = extended**

3&4) Microinstruction Format: unencoded vs. encoded fields

Field Name	*Width* wide	narrow	*Control Signals Set*
ALU Control	4	2	ALUOp
SRC1	2	1	ALUSelA
SRC2	5	3	ALUSelB, ExtOp
ALU Destination	3	2	RegWrite, MemtoReg, RegDst
Memory	3	2	MemRead, MemWrite, IorD
Memory Register	1	1	IRWrite
PCWrite Control	3	2	PCWrite, PCWriteCond, PCSource
Sequencing	3	2	AddrCtl
Total width	24	15	bits

5) Legend of Fields and Symbolic Names

Field Name	*Values for Field*	*Function of Field with Specific Value*
ALU	Add	ALU adds
	Subt.	ALU subtracts
	Func code	ALU does function code
	Or	ALU does logical OR
SRC1	PC	1st ALU input = PC
	rs	1st ALU input = Reg[rs]
SRC2	4	2nd ALU input = 4
	Extend	2nd ALU input = sign ext. IR[15-0]
	Extend0	2nd ALU input = zero ext. IR[15-0]
	Extshft	2nd ALU input = sign ex., sl IR[15-0]
	rt	2nd ALU input = Reg[rt]
destination	rd ALU	Reg[rd] = ALUout
	rt ALU	Reg[rt] = ALUout
	rt Mem	Reg[rt] = Mem
Memory	Read PC	Read memory using PC
	Read ALU	Read memory using ALUout for addr
	Write ALU	Write memory using ALUout for addr
Memory register	IR	IR = Mem
PC write	ALU	PC = ALU
	ALUoutCond	IF ALU Zero then PC = ALUout
Sequencing	Seq	Go to sequential µinstruction
	Fetch	Go to the first microinstruction
	Dispatch	Dispatch using ROM.

Quick check: what do these fieldnames mean?

Destination:

Code	Name	RegWrite	MemToReg	RegDest
00	---	0	X	X
01	rd ALU	1	0	1
10	rt ALU	1	0	0
11	rt MEM	1	1	0

SRC2:

Code	Name	ALUSelB	ExtOp
000	---	X	X
001	4	00	X
010	rt	01	X
011	ExtShft	10	1
100	Extend	11	1
111	Extend0	11	0

Specific Sequencer from before

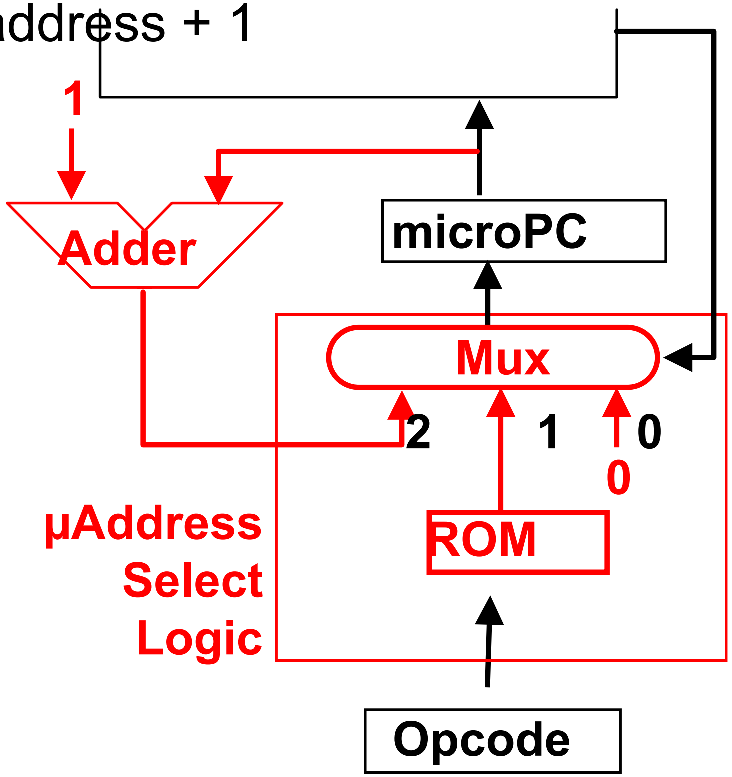
Sequencer-based control unit from last lecture

– Called "microPC" or "µPC" vs. state register

Code	*Name*	*Effect*
00	fetch	Next µaddress = 0
01	dispatch	Next µaddress = dispatch ROM
10	seq	Next µaddress = µaddress + 1

ROM:

R-type	000000	0100
BEQ	000100	0011
ori	001101	0110
LW	100011	1000
SW	101011	1011



Legacy Software and Microprogramming

- IBM bet company on 360 Instruction Set Architecture (ISA): single instruction set for many classes of machines
  - (8-bit to 64-bit)
- Stewart Tucker stuck with job of what to do about software compatibility
  - If microprogramming could easily do same instruction set on many different microarchitectures, then why couldn't multiple microprograms do multiple instruction sets on the same microarchitecture?
  - Coined term "emulation": instruction set interpreter in microcode for non-native instruction set
  - Very successful: in early years of IBM 360 it was hard to know whether old instruction set or new instruction set was more frequently used

Microprogramming Pros and Cons

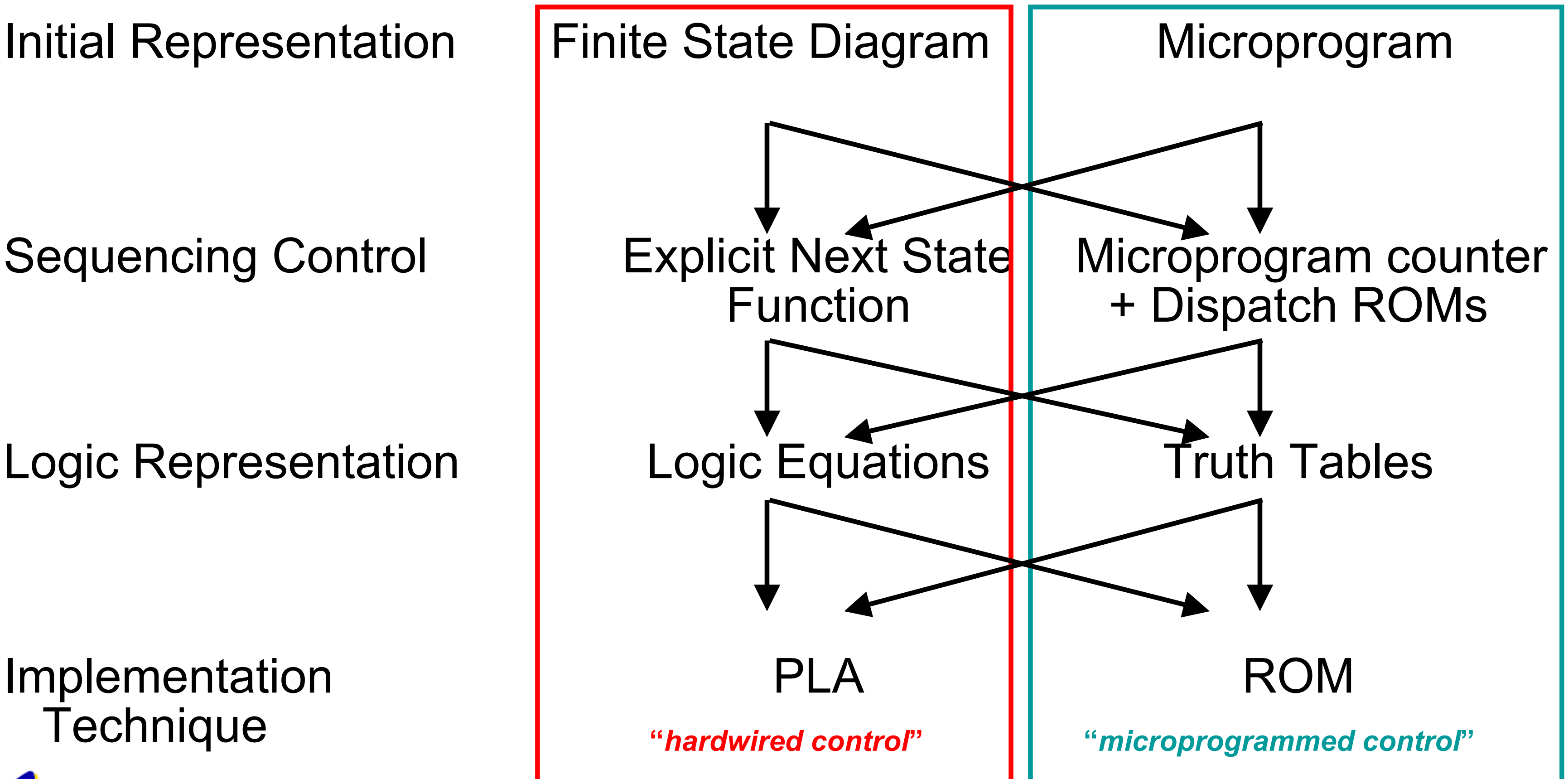
- Ease of design
- Flexibility
  - Easy to adapt to changes in organization, timing, technology
  - Can make changes late in design cycle, or even in the field
- Can implement very powerful instruction sets (just more control memory)
- Generality
  - Can implement multiple instruction sets on same machine.
  - Can tailor instruction set to application.
- Compatibility
  - Many organizations, same instruction set
- Costly to implement
- Slow

Thought: Microprogramming one inspiration for RISC

- If simple instruction could execute at very high clock rate…
- If you could even write compilers to produce microinstructions…
- If most programs use simple instructions and addressing modes…
- If microcode is kept in RAM instead of ROM so as to fix bugs …
- If same memory used for control memory could be used instead as cache for "macroinstructions"…
- Then why not skip instruction interpretation by a microprogram and simply compile directly into lowest language of machine? (microprogramming is overkill when ISA matches datapath 1-1)

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 (1 of 3)

- 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) (2 of 3)

- Control is specified by finite state diagram
- 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 $\Rightarrow$ simple control
  – could try to reduce hardware (see the book)
  – rather go for speed => many instructions at once!

Summary (3 of 3)

- Microprogramming is a fundamental concept
  - implement an instruction set by building a very simple processor and <u>interpreting</u> the instructions
  - essential for very complex instructions and when few register transfers are possible
  - *Control design reduces to Microprogramming*
- Design of a Microprogramming language
  - Start with list of control signals
  - Group signals together that make sense (vs. random): called “fields”
  - Place fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)
  - To minimize the width, encode operations that will never be used at the same time
  - Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals

Where to get more information?

- Multiple Cycle Controller: Appendix C of your text book.
- Microprogramming: Section 5.7 of your text book.
- D. Patterson, “Microprograming,” Scientific American, March 1983.
- D. Patterson and D. Ditzel, “The Case for the Reduced Instruction Set Computer,” Computer Architecture News 8, 6 (October 15, 1980)

Microprogram it yourself!

Label	*ALU*	*SRC1*	*SRC2*	*Dest.*	*Memory*	*Mem. Reg.*	*PC Write*	*Sequencing*
Fetch:	Add	PC	4		Read PC	IR	ALU	Seq