COMP3211 03S2 Lecture 21

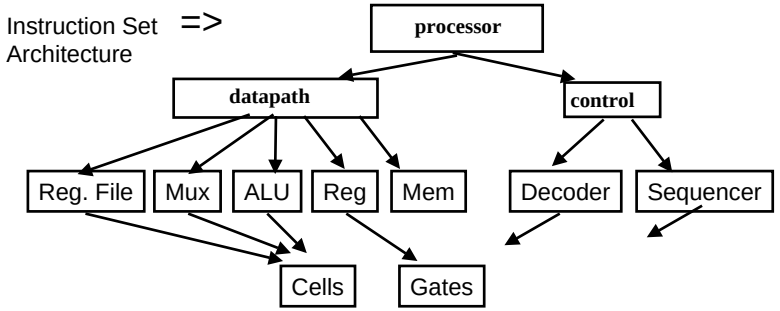
Multicycle Processor Design

Adapted from

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

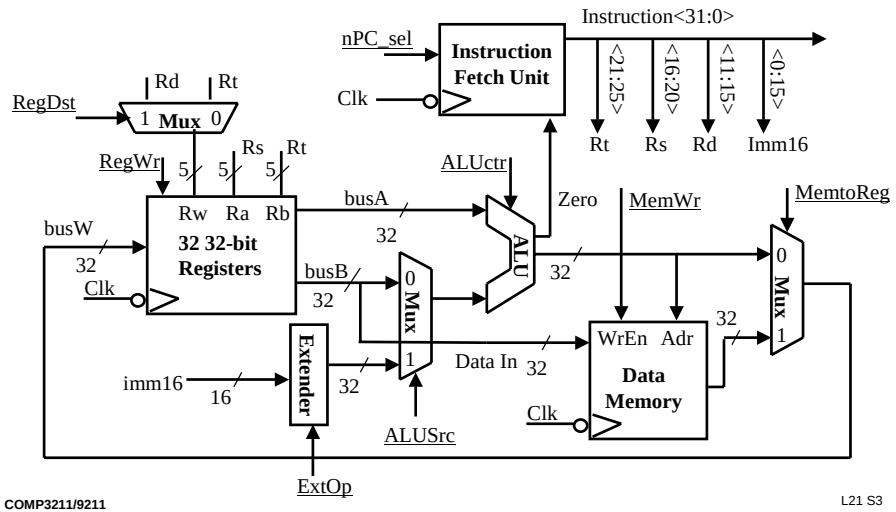
Recap: Processor Design is a Process

- **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**

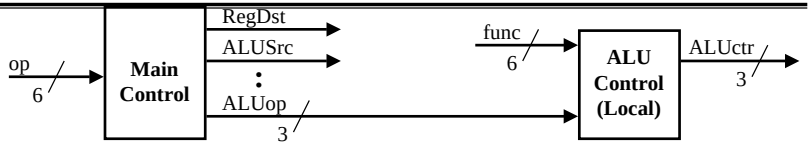


Recap: A Single Cycle Datapath

- **We have everything except control signals (underline)**
  - **Today's lecture will show you how to generate the control signals**

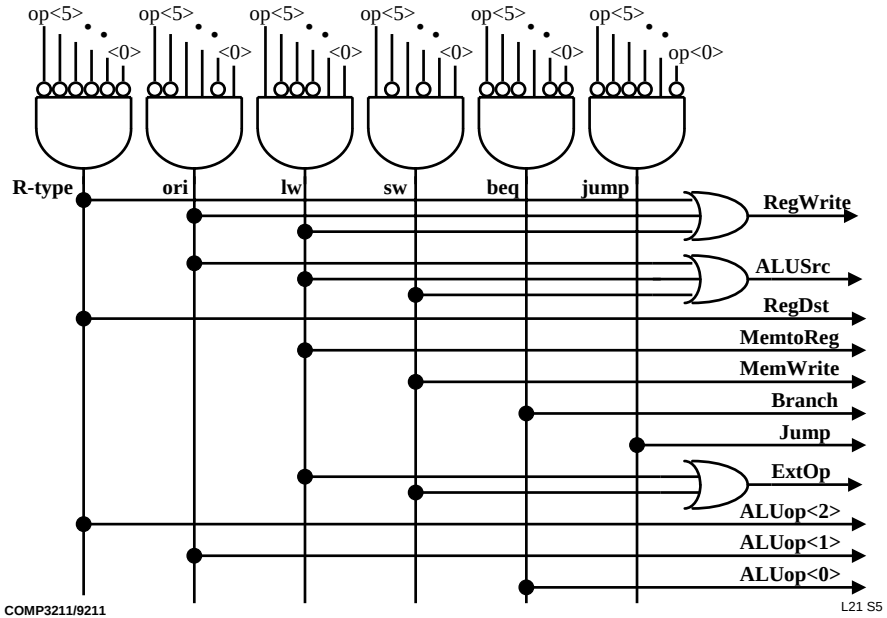


Recap: The "Truth Table" for the Main Control

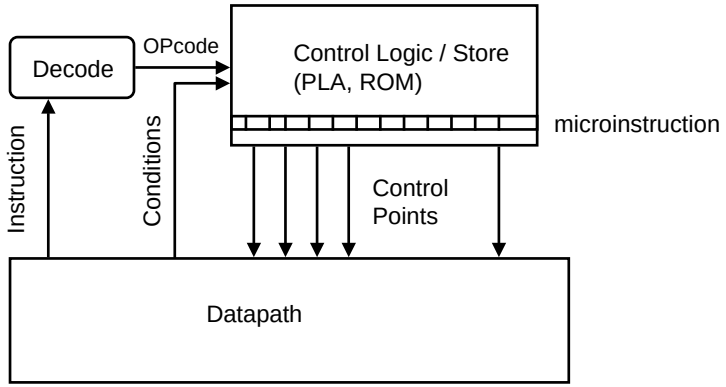


op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x

Recap: PLA Implementation of the Main Control



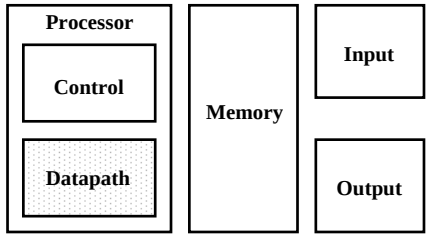
Recap: Systematic Generation of Control



- **In our single-cycle processor, each instruction is realized by exactly one control command or "*microinstruction*"**
  - **in general, the controller is a finite state machine**
  - **microinstruction can also control sequencing (see later)**

The Big Picture: Where are We Now?

- **The Five Classic Components of a Computer**

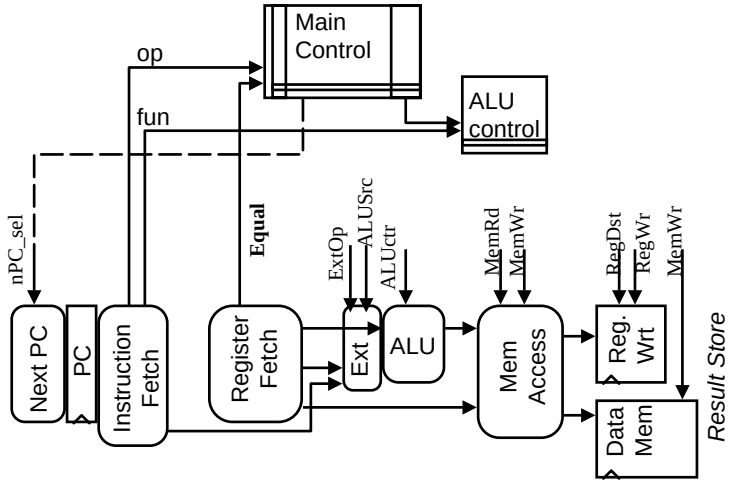


- **Today's Topic: Designing the Datapath for the Multiple Clock Cycle Datapath**

Outline of Today's Lecture

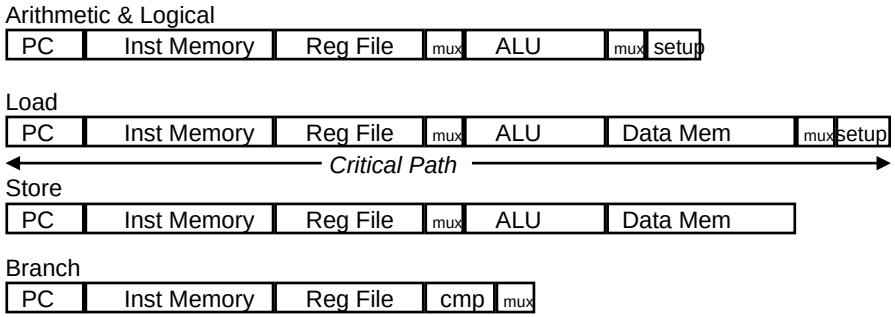
- **Recap: single cycle processor**
- **Faster designs**
- **Multicycle Datapath**
- **Performance Analysis**
- **Multicycle Control**

Abstract View of our single cycle processor



- **looks like a FSM with PC as state**

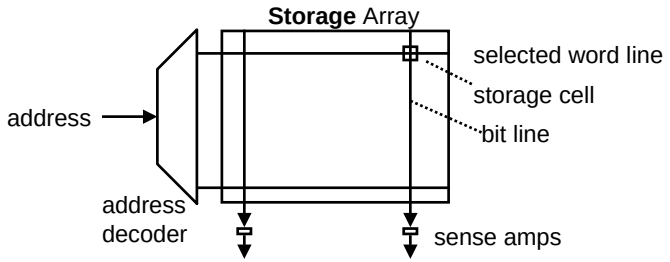
What's wrong with our CPI=1 processor?



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

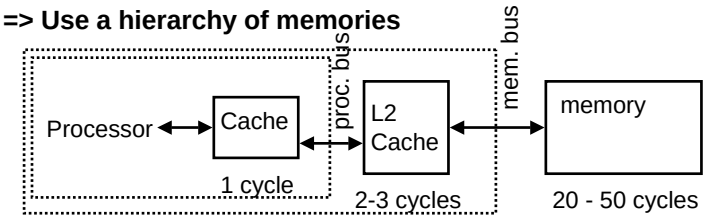
Memory Access Time

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



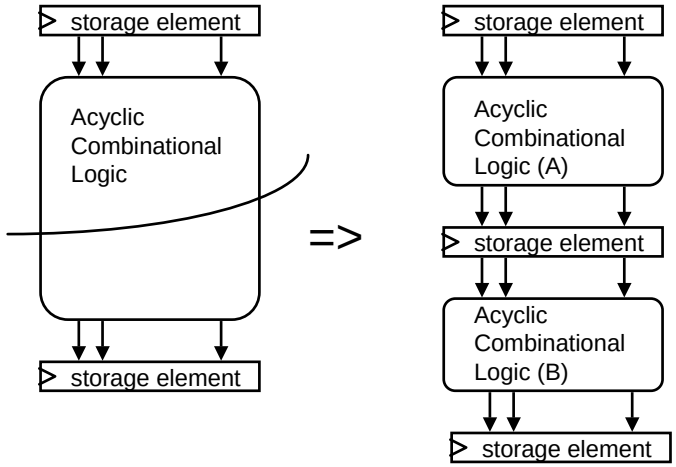
  - **question: register file vs. memory**

- **=> Use a hierarchy of memories**



Reducing Cycle Time

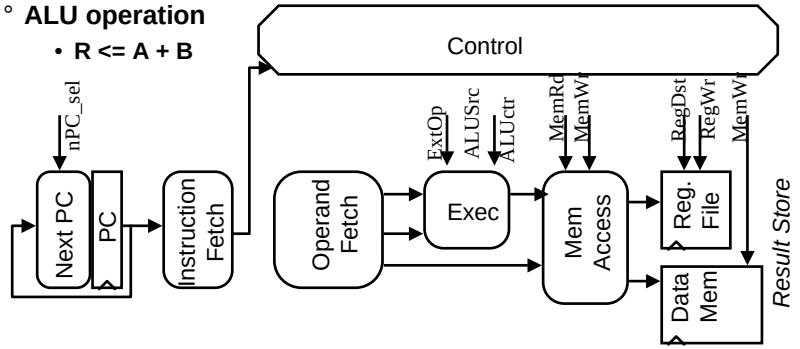
- **Cut combinational dependency graph and insert register / latch**
- **Do same work in two fast cycles, rather than one slow one**



- **We would like to balance delays of circuits A & B**

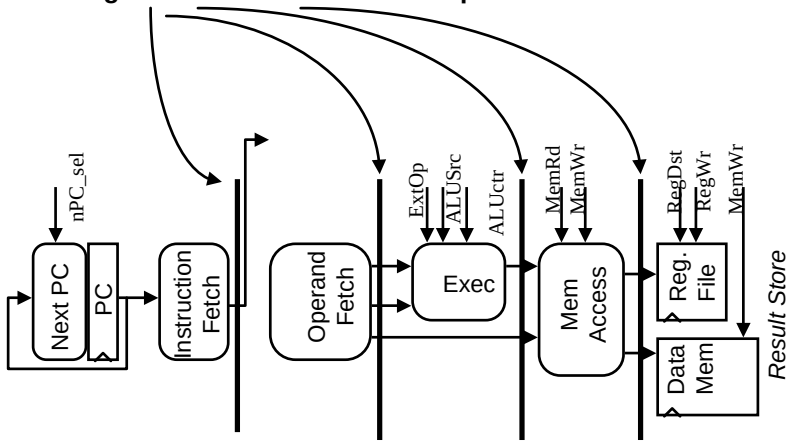
Basic Limits on Cycle Time

- 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

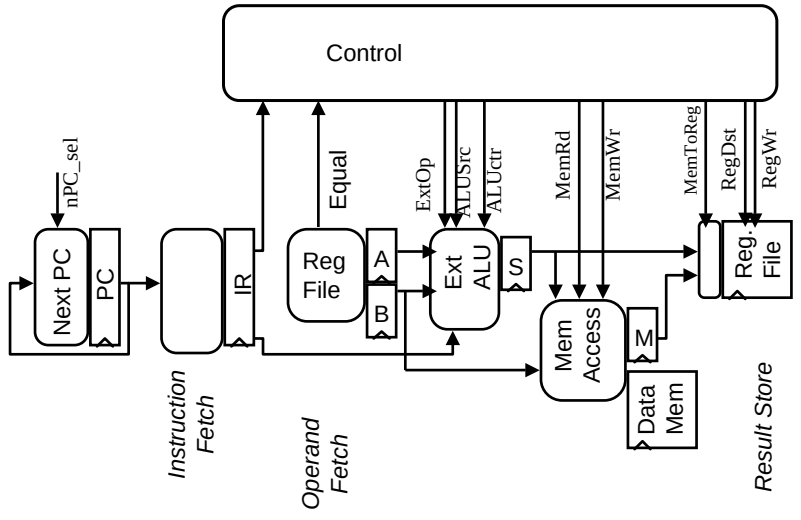


- Want to balance time spent in each stage
  - Remember each cycle takes the same amount of time
  - The longest stage delay defines the cycle time

Multicycle datapath control

- For single-cycle processor, each instruction is realized by exactly one set of control signals or control points
- For multicycle processor, each instruction is completed in several cycles with each cycle having different control signals. I.e., An instructions is realized by a sequence of control signal sets

Example Multicycle Datapath



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

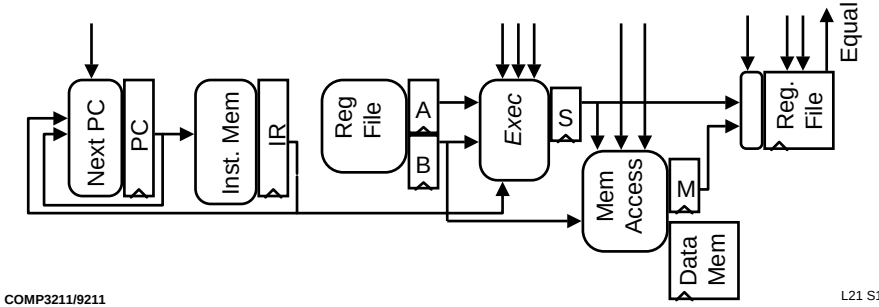
Step 4: R-rtype (add, sub, . . .)

° **Logical Register Transfer**

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

° **Physical Register Transfers**

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



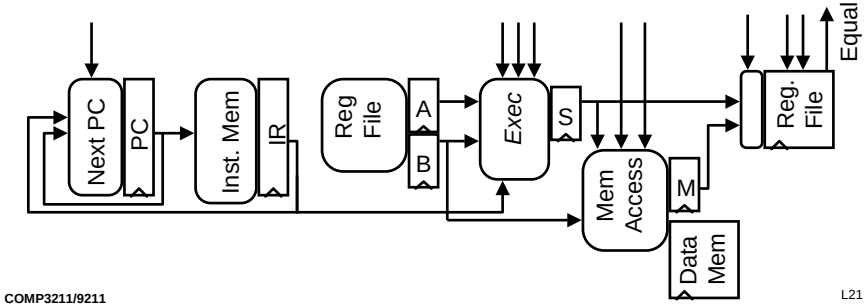
Step 4:Logical immed

° **Logical Register Transfer**

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

° **Physical Register Transfers**

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



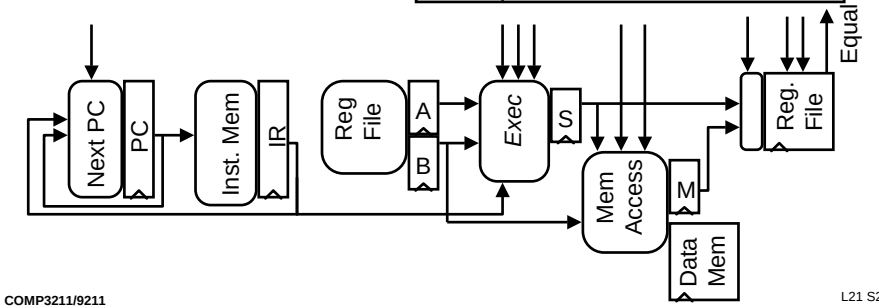
Step 4 : Load

° **Logical Register Transfer**

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

° **Physical Register Transfers**

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



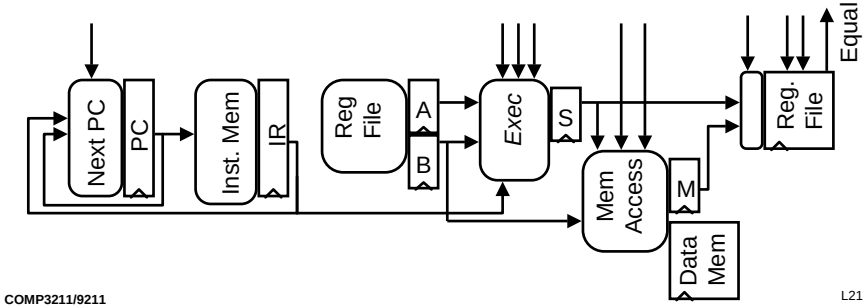
Step 4 : Store

° **Logical Register Transfer**

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

° **Physical Register Transfers**

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



Step 4 : Branch

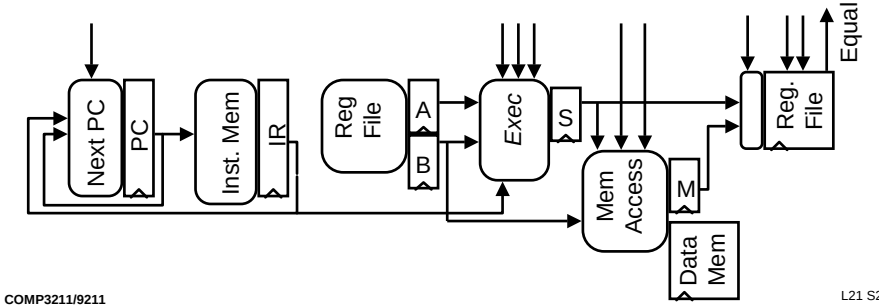
° **Logical Register Transfer**

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

° **Physical Register Transfers**

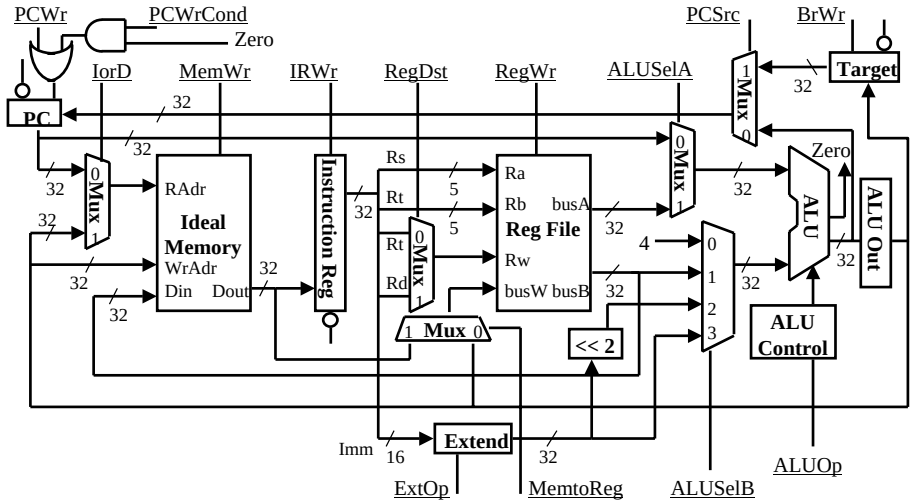
inst	Physical Register Transfers
	IR <– MEM[pc]
BEQ|$\bar{E}$q	PC <– PC + 4

inst	Physical Register Transfers
	IR <– MEM[pc]
BEQ|Eq	PC <– PC + sx(Im16) || 00



Alternative datapath (book): Multiple Cycle Datapath

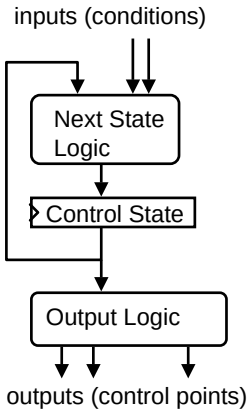
° **Miminizes Hardware: 1 memory, 1 adder**

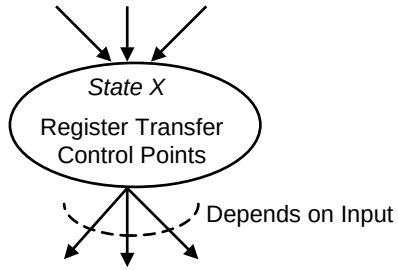


Our Control Model

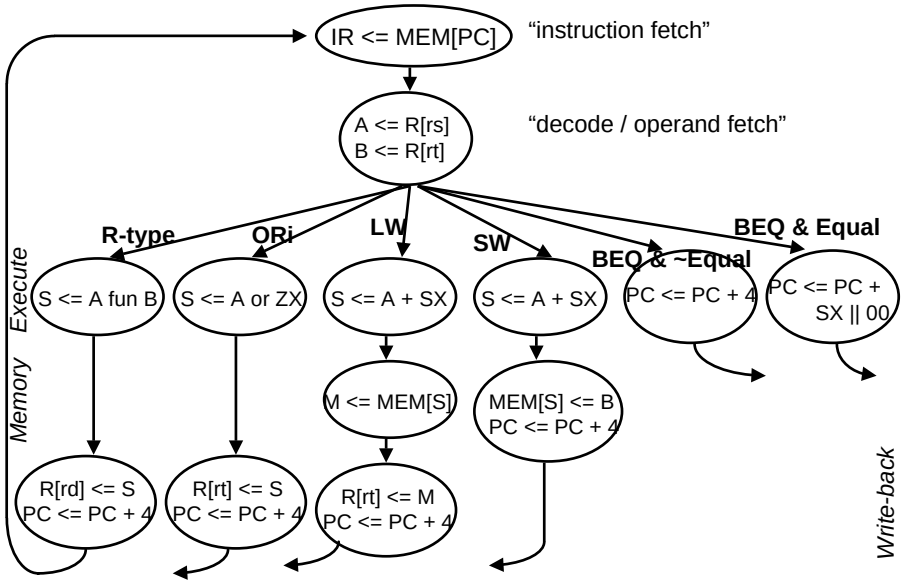
° **State specifies control points for Register Transfer**

° **Transfer occurs upon exiting state (same falling edge)**

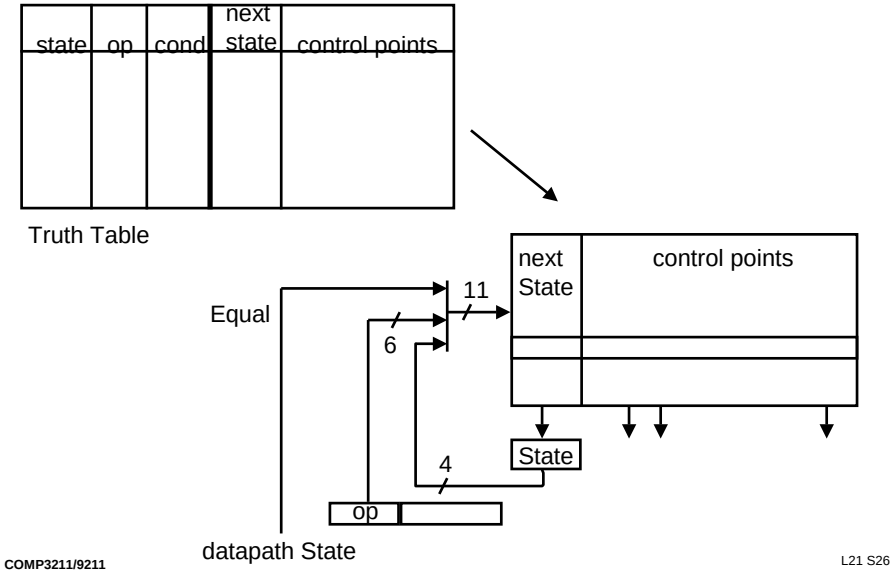




Step 4 => Control Specification for multicycle proc



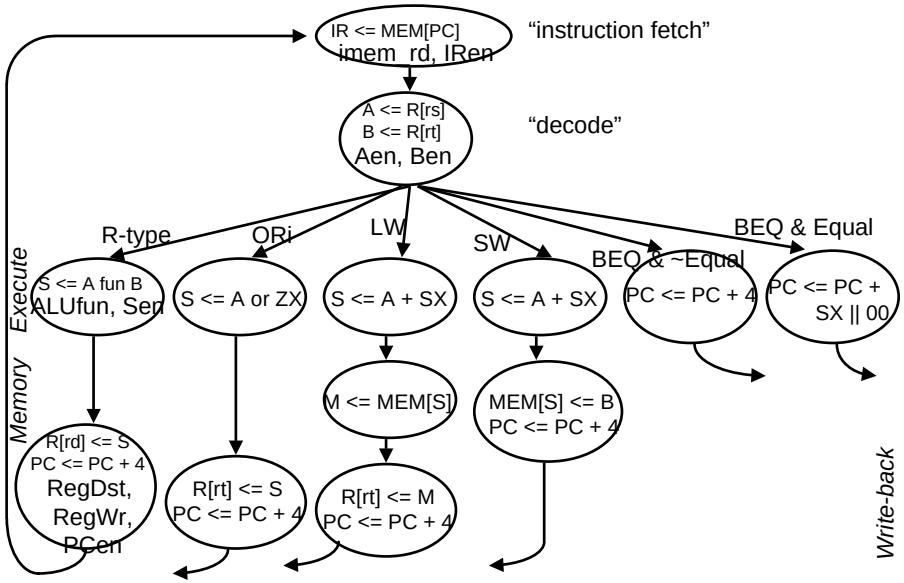
Traditional FSM Controller



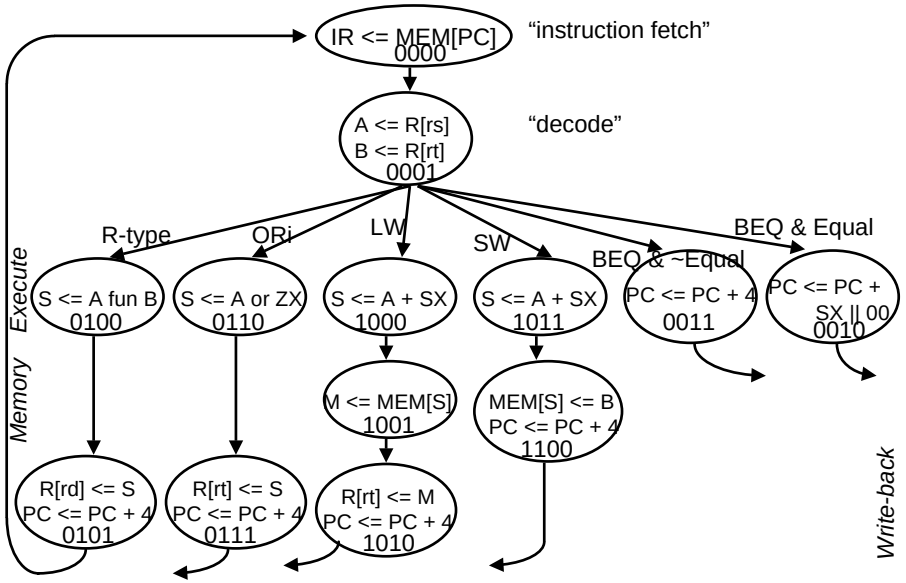
Step 5: datapath + state diagram => control

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

Mapping RTs to Control Points



Assigning States



Detailed Control Specification

	State	Op field	Eq	Next	IR	PC en sel	Ops A B	Exec Ex Sr ALU S	Mem R W M	Write-Back M-R Wr Dst
	0000	??????	?	0001	1					
	0001	BEQ	0	0011			1 1	*-all same in Moore machine*		
	0001	BEQ	1	0010			1 1			
	0001	R-type	x	0100			1 1			
	0001	orI	x	0110			1 1			
	0001	LW	x	1000			1 1			
	0001	SW	x	1011			1 1			
	0010	xxxxxx	x	0000		1 1				
	0011	xxxxxx	x	0000		1 0				
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 0	
	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	

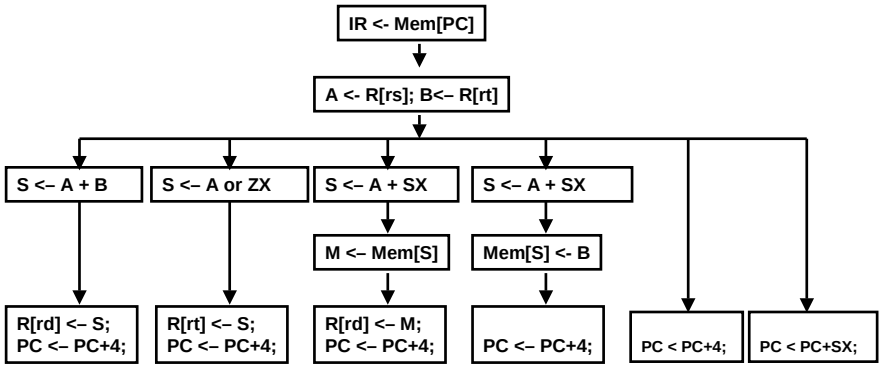
Performance Evaluation

° **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 $freq_i$
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

° **Improvement achieved if 4.1 * CC(multicycle) < 1 * CC(single cycle) → ref. Slide 14**

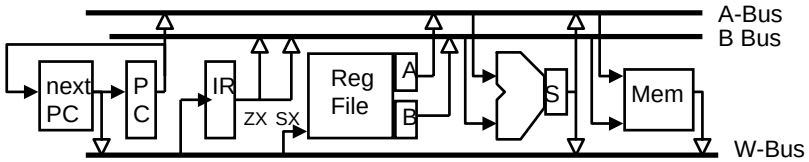
How Effectively are we utilizing our hardware?



° **Example: memory is used twice, at different times**
- **Ave mem access per inst = 1 + Freq(lw) + Freq(sw) ~ 1.4**
- **if CPI is 4.1, imem utilization = 1/4.1, dmem = 0.4/4.1**

° **We could reduce HW without hurting performance**
- **extra control**

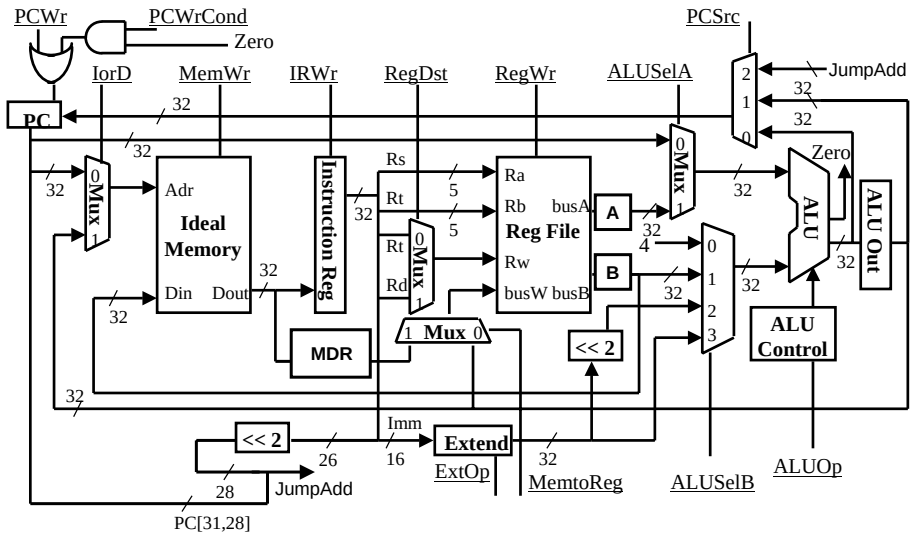
“Princeton” Organization



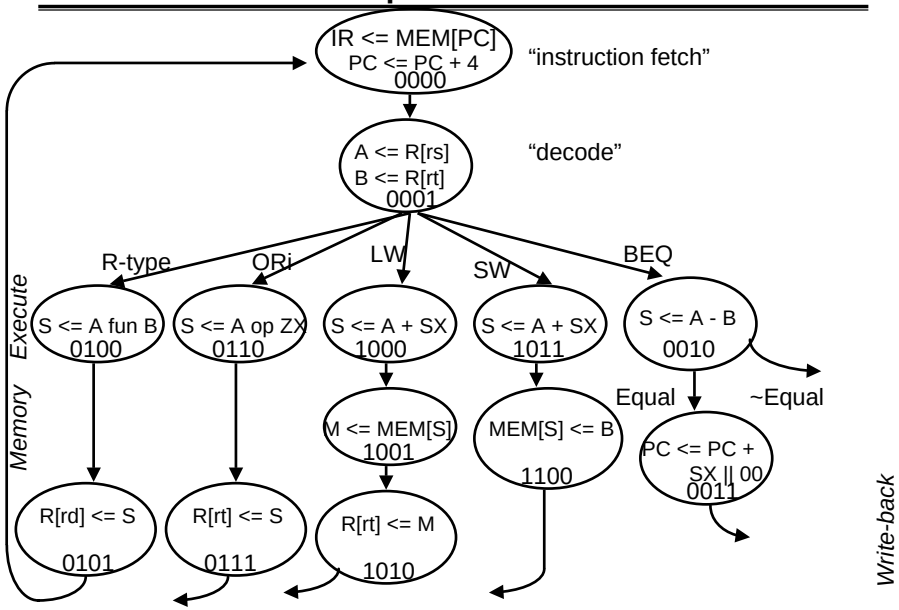
- Single memory for instruction and data access
  - memory utilization → 1.4/4.1
- In this case our state diagram does not change
  - several additional control signals
  - must ensure each bus is only driven by one source on each cycle

Alternative datapath (book): Multiple Cycle Datapath

- Miminizes Hardware: 1 memory, 1 adder



Our Controller FSM Spec



Summary of actions in each step (RTL)

Step name	Action for R–type instructions	Action for memory–reference instrucitons	Actions for branches	Action for jumps
IF	IR = Memory[PC] PC = PC+4			
RF/ID	A = Reg[IR[25–21] B = Reg[IR[20–16] ALUOut = PC + (signExt(IR[15–0])<<2)			
EXEC	ALUOUt = A op B	ALUOut = A +signExt SignExt(IR[15–0])	if(A==B) then PC = ALUOut	PC = PC[31–28]|| (IR[25–0]<<2)
MEM		Load: M=Memory[ALUOut] or Store: Memory[ALUOut] = B		
WB	Reg[IR[15–11]]= ALUOut	Load:Reg[IR[20–16]] = M		

Microprogramming

° Control is the hard part of processor design
  ° Datapath is fairly regular and well-organized
  ° Memory is highly regular
  ° Control is irregular and global

Microprogramming:

-- A Particular Strategy for Implementing the Control Unit of a processor by "programming" at the level of register transfer operations

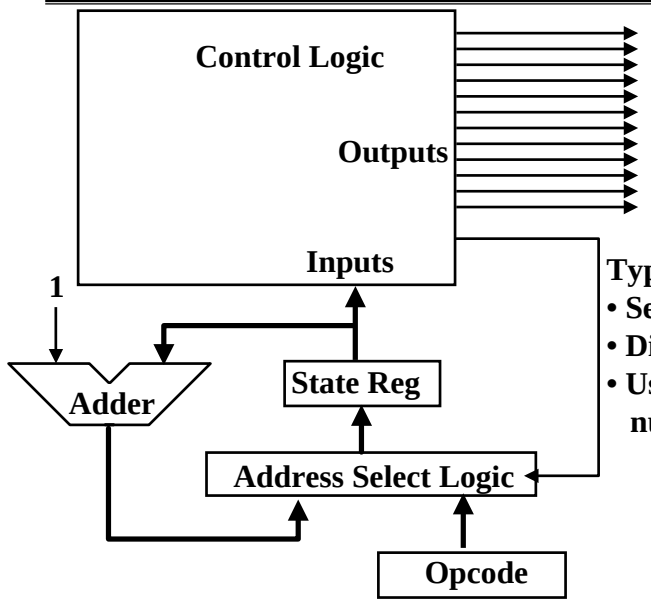
Microarchitecture:

-- Logical structure and functional capabilities of the hardware as seen by the microprogrammer

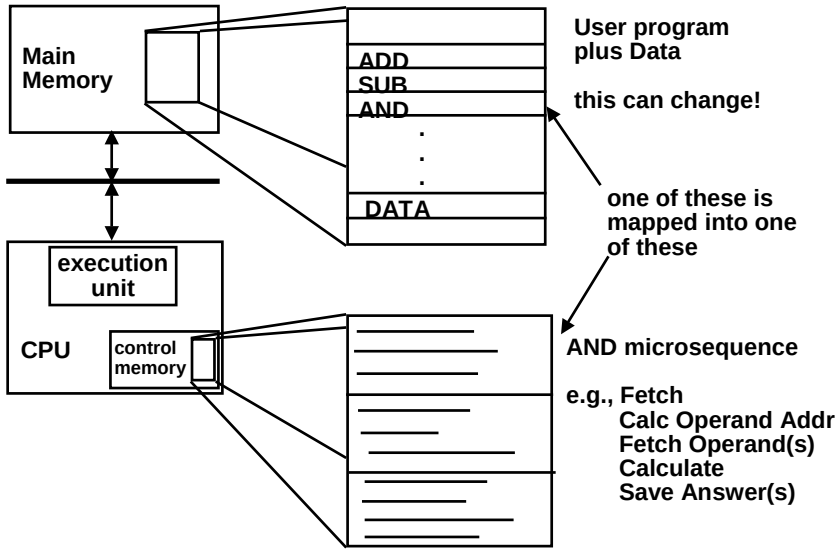
Historical Note:

IBM 360 Series first to distinguish between architecture & organization
Same instruction set across wide range of implementations, each with different cost/performance

Sequencer-based control unit



"Macroinstruction" Interpretation



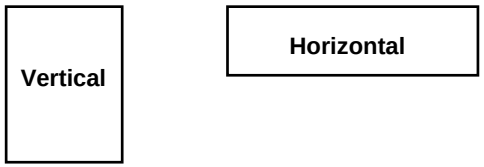
Variations on Microprogramming

° "Horizontal" Microcode
  – control field for each control point in the machine

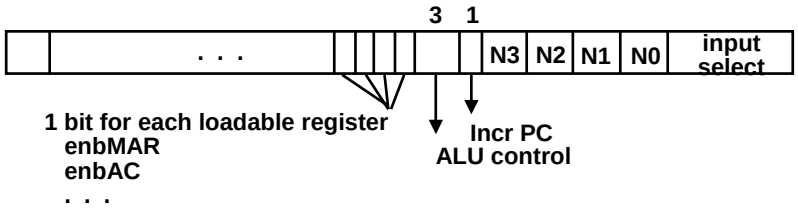
μseq	μaddr	A-mux	B-mux	bus enables	register enables	

° "Vertical" Microcode
  – compact microinstruction format for each class of microoperation
  – local decode to generate all control points

branch: μseq-op μadd
execute: ALU-op A,B,R
memory: mem-op S, D



Extreme Horizontal



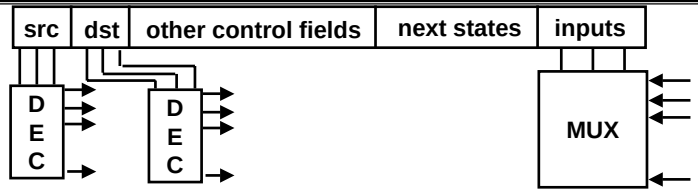
Depending on bus organization, many potential control combinations simply wrong, i.e., implies transfers that can never happen at the same time.

Makes sense to encode fields to save ROM space

Example: mem_to_reg and ALU_to_reg should never happen simultenously; => encode in single bit which is decoded rather than two separate bits

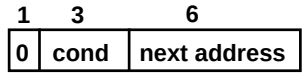
NOTE: encoding should be just sufficient that parallel actions that the datapath supports should still be specifiable in a single microinstruction

More Vertical Format

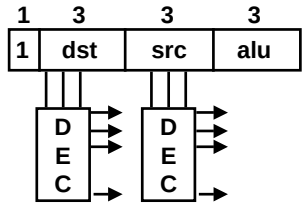


Some of these may have nothing to do with registers!

Multiformat Microcode:



Branch Jump



Register Xfer Operation

Horizontal vs. Vertical Microprogramming

Most microprogramming-based controllers vary between:

horizontal organization (1 control bit per control point)

vertical organization (fields encoded in the control memory and must be decoded to control something)

Horizontal

+ more control over the potential parallelism of operations in the datapath
- uses up lots of control store

Vertical

+ easier to program, not very different from programming a RISC machine in assembly language
- extra level of decoding may slow the machine down

Let's design a microprogram for our MIPS subset

° Defining a microinstruction format
- Start with a list of control signals
- Grouping 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
- Determine symbolic value for each field
  - each of this value has corresponding control signals

° Creating the microprogram
- For a specified instruction set

° Implementing the microprogram

Define a microinstruction format

- Two design goals:
  - Readability
    - E.g., one field for one functional block
  - Compatibility
    - Each field responsible for specifying a nonoverlapping set of control signals

Our microinstruction format

Field name	Values
Label	Any string
ALU control	Add, Sub, Func code
SRC1	PC, A
SRC2	4, B, Extshft, Extend
Register control	Read, Write ALU, Write M
Memory	Read PC, Read ALU, Write ALU
PCWrite control	ALUOut–cond, Jump address
sequencing	Seq, Fetch, Dispatch i

More on sequencing

- Determining what instruction should be executed next
- Unlike in normal program, the next instruction should be handled explicitly
- Three values in sequencing
  - Seq: choose the next microinstruction sequentially
    - The one physically next to the current microinstruction in the control memory
  - Fetch: go to the microinstruction for a new instruction
    - The microinstruction is labeled with “Fetch”
  - Dispatch: jump to some microinstruction, according to the Op field of the instruction

Creating the microprogram

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write M			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Sub	A	B			ALUOut–cond	Fetch
JUMP1						Jump address	Fetch

More on Dispatch

- Implemented by dispatch table
- We have two dispatch tables in our design

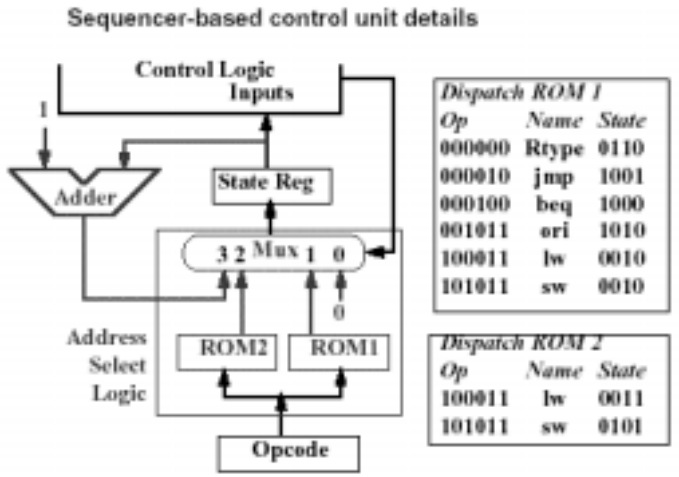
Microcode dispatch table 1

Opcode field	Opcode name	Value
000000	R–format	Rformat1
000010	jmp	JUMP1
000100	beq	BEQ1
100011	lw	Mem1
101011	sw	Mem1

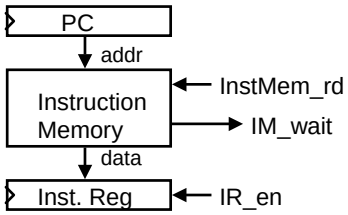
Microcode dispatch table 2

Opcode field	Opcode name	Value
100011	lw	LW2
101011	sw	SW2

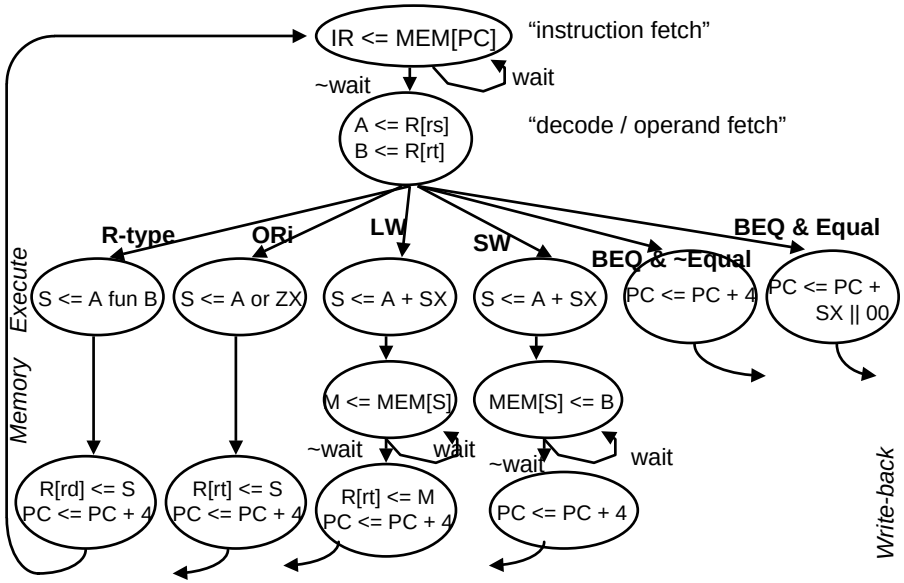
Controller arrangement



Controlling Memory

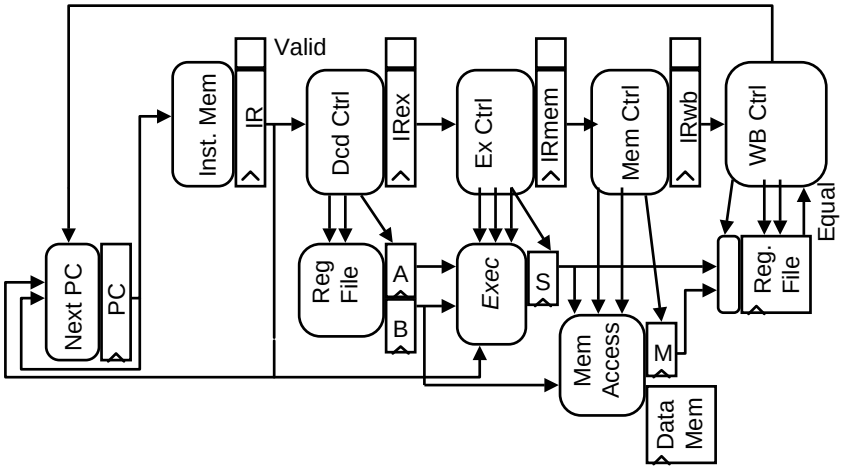


Controller handles non-ideal memory



Time-state Control Path

° Local decode and control at each stage



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

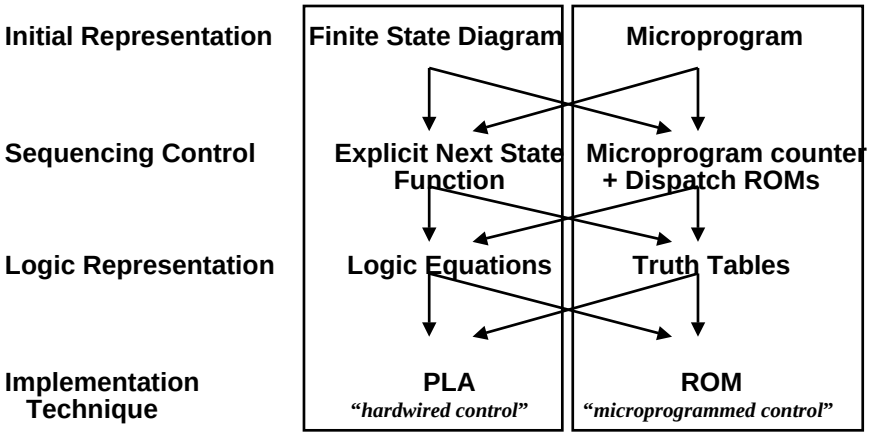
° Partition datapath into equal size chunks to minimize cycle time

° Control is specified by finite state diagram

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

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.



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

° Slow

° Costly to implement

Summary: 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)

Summary (cont'd)

- 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
- Control is more complicated with:
  - complex instruction sets
  - restricted datapaths (e.g. single memory)
- Simple Instruction set and powerful datapath => simple control
  - could try to reduce hardware (see the book)
  - rather go for speed => many instructions at once!

Where to get more information?

- D. Patterson, "Microprograming," Scientific America, March 1983.
- D. Patterson and D. Ditzel, "The Case for the Reduced Instruction Set Computer," Computer Architecture News 8, 6 (October 15, 1980)

Exercise

- For the microcontroller given on the overhead, which assigns sequentially increasing addresses to each microinstruction of the microprogram studied in Slide 48, list the addresses of the microinstructions that are read in order to execute the following program segment:

```
     SW    $0, 10($3)
     LW    $1, 10($3)
     BEQ   $0, $1, L1
     SUB   $5, $6, $7
L1:  SW    $5, 20($3)
```