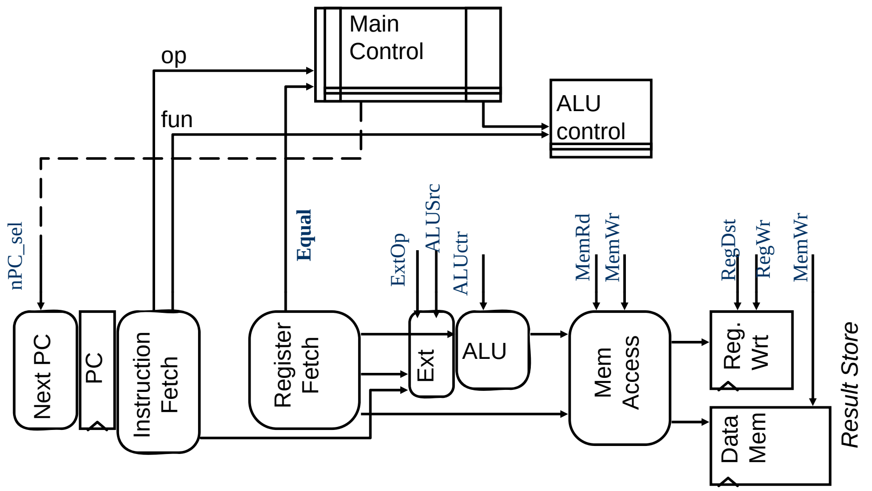
Computer System Architecture

Processor Part III

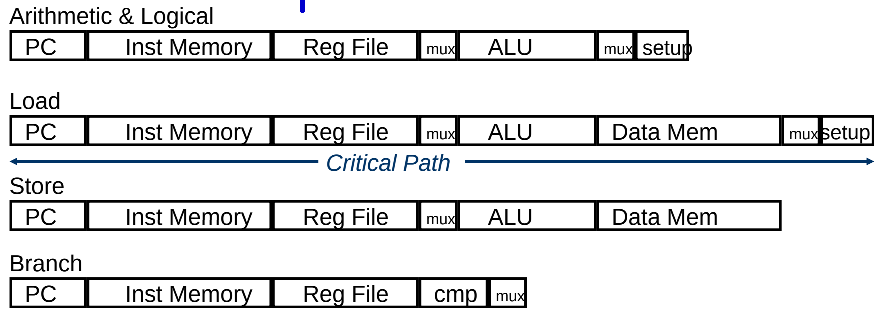
Chalermek Intanagonwiwat

Slides courtesy of John Hennessy and David Patterson

Abstract View of our single cycle processor

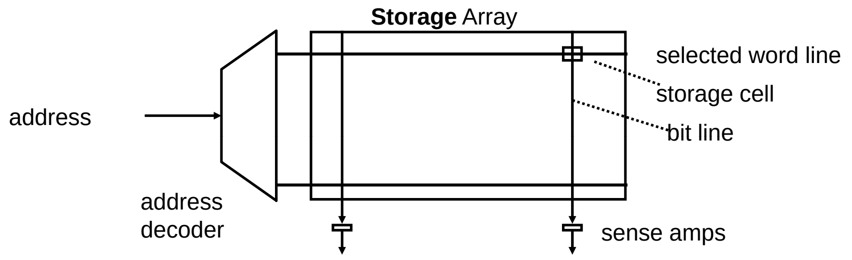


What's wrong with our CPI=1 processor?



- 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

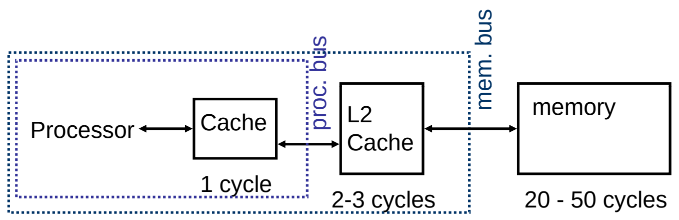
Memory Access Time



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

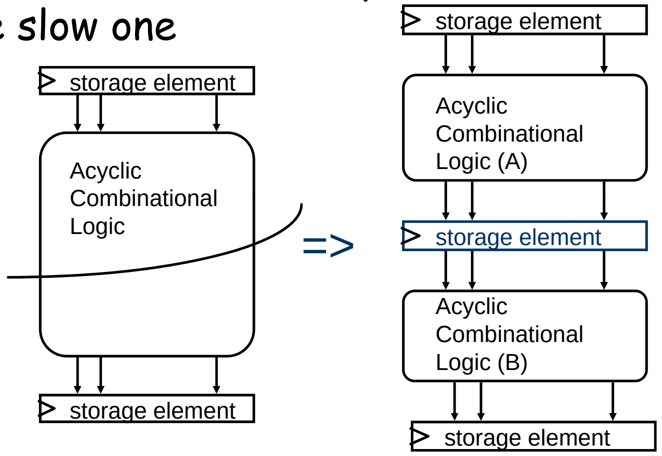
Memory Access Time (cont.)

- => Use a hierarchy of memories



Reducing Cycle Time

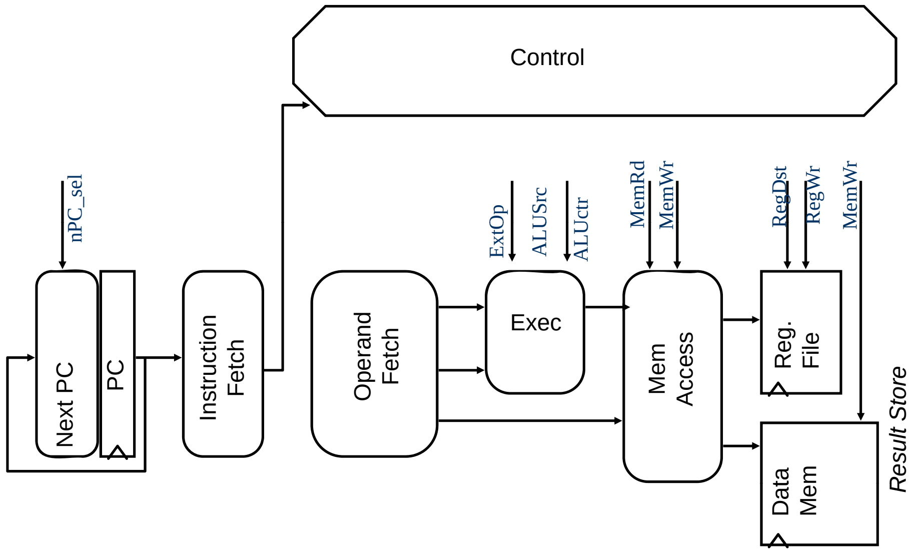
- Cut combinational dependency graph and insert register / latch
- Do same work in two fast cycles, rather than one slow one


storage element
Acyclic Combinational Logic
storage element
=>
storage element
Acyclic Combinational Logic (A)
storage element
Acyclic Combinational Logic (B)
storage element

Basic Limits on Cycle Time

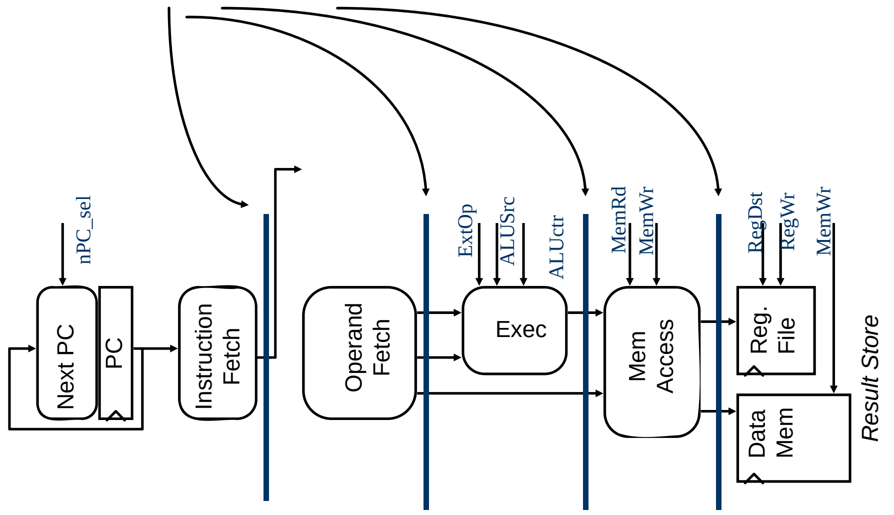
- Next address logic
  - PC <= branch ? PC + offset : PC + 4
- Instruction Fetch
  - InstructionReg <= Mem[PC]
- Register Access
  - A <= R[rs]
- ALU operation
  - R <= A + B

Basic Limits on Cycle Time (cont.)

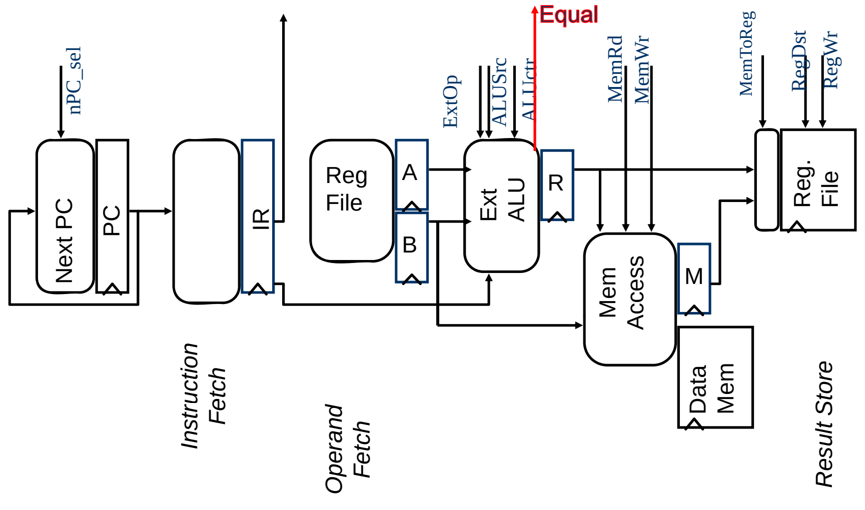


Partitioning the CPI=1 Datapath

- Add registers between smallest steps



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

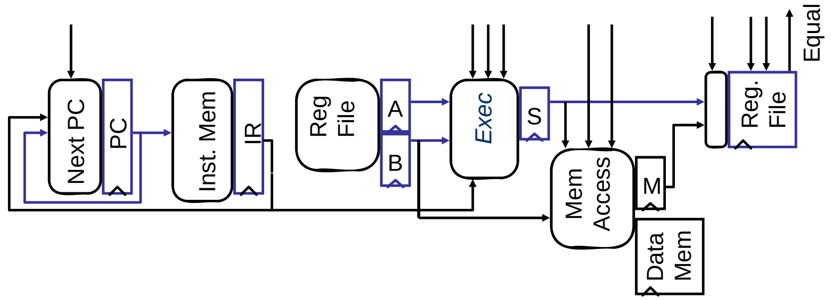
Step 5: Physical RTs => Control

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

- Logical Register Transfer
- Physical Register Transfers

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

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

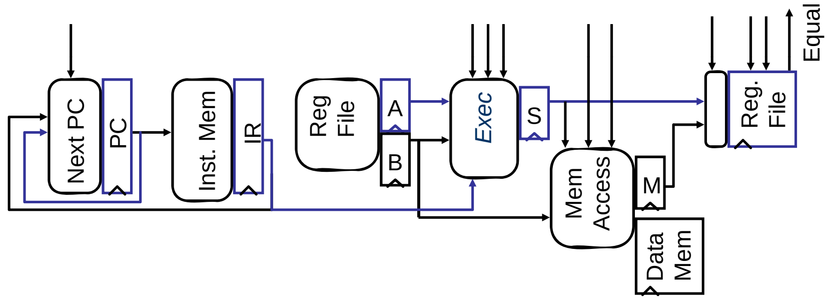


Step 4: Logical immed

- Logical Register Transfer
- Physical Register Transfers

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

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

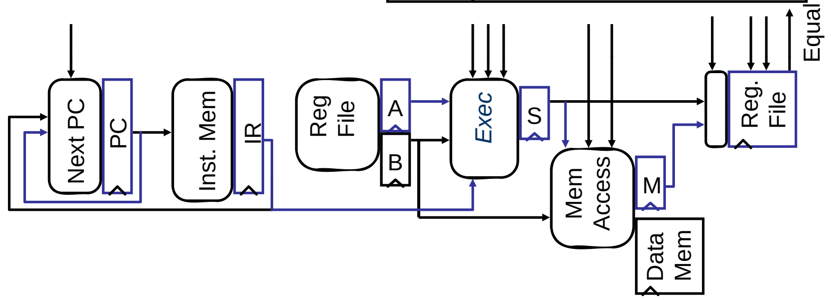


Step 4: Load

- Logical Register Transfer
- Physical Register Transfers

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

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

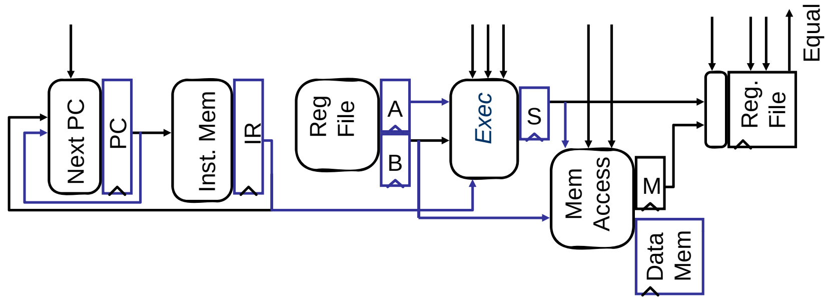


Step 4: Store

- Logical Register Transfer
- Physical Register Transfers

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

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



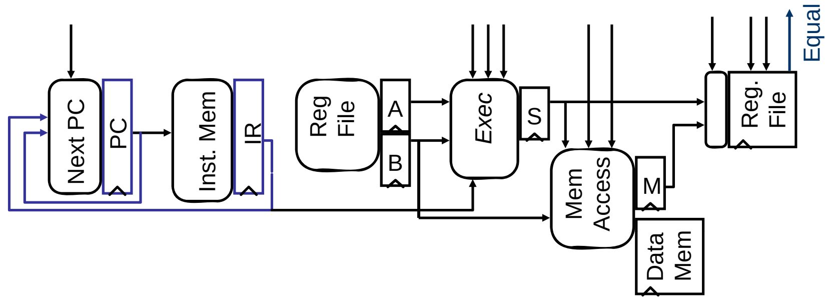
Step 4: Branch

- Logical Register Transfer
- Physical Register Transfers

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

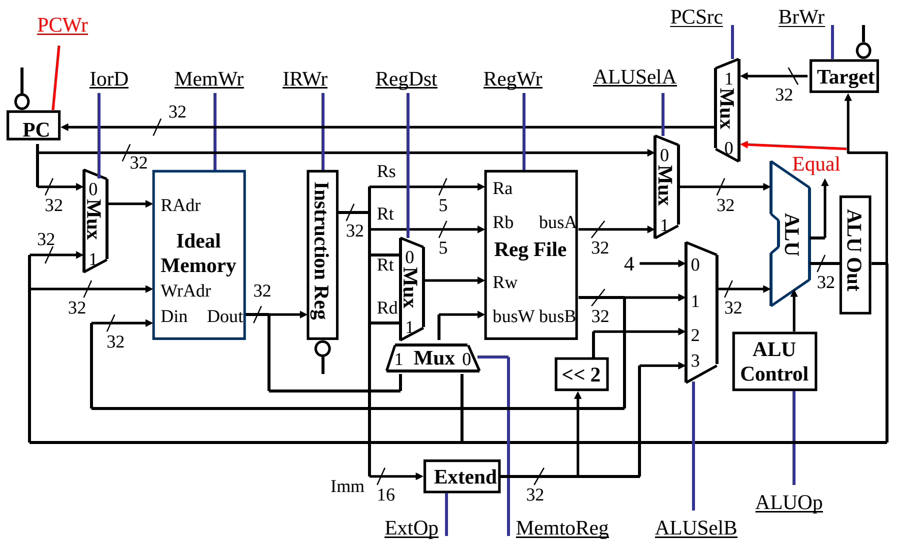
inst	Physical Register Transfers
	IR <– MEM[pc]
BEQ|$\overline{Eq}$	PC <– PC + 4

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



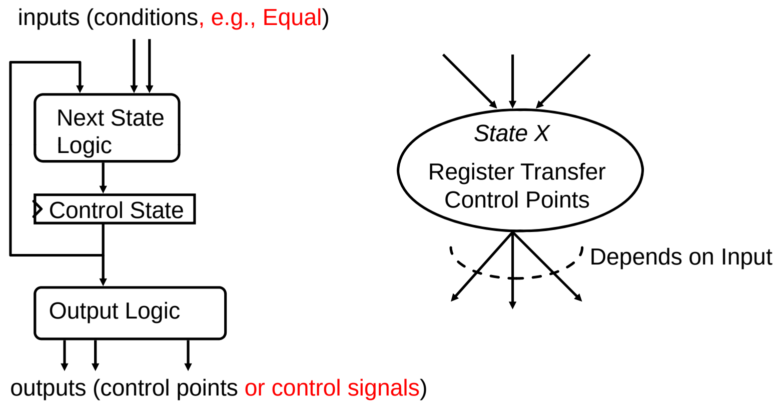
Multiple Cycle Datapath

- Miminizes Hardware: 1 memory, 1 adder



Our Control Model

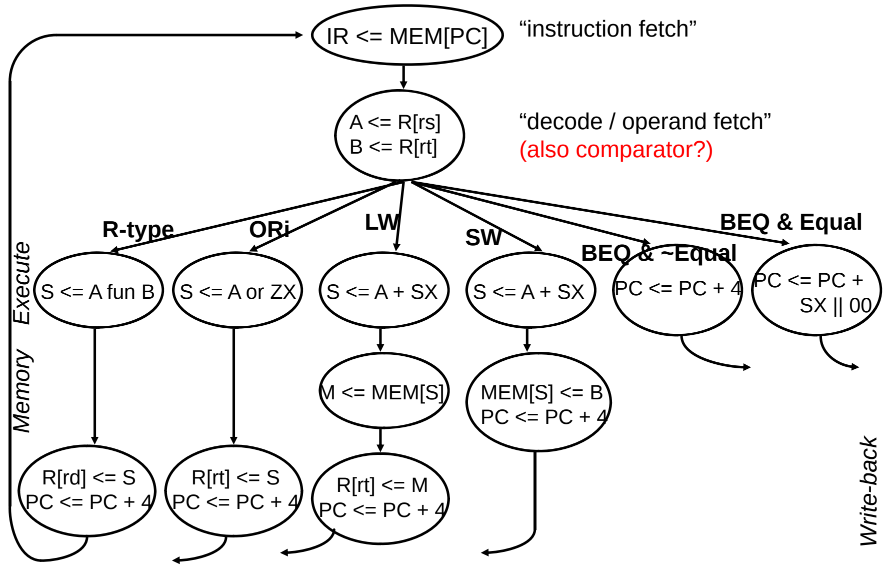
- State specifies control points for Register Transfer
- Transfer occurs upon exiting state (same falling edge)


inputs (conditions, e.g., Equal)
Next State Logic
Control State
Output Logic
outputs (control points or control signals)
State X
Register Transfer Control Points
Depends on Input

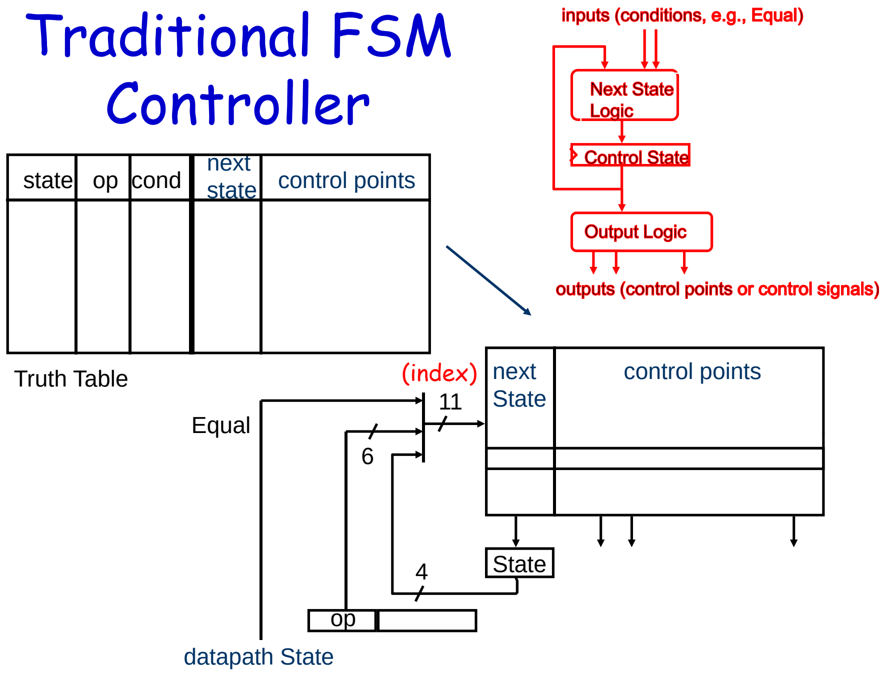
Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

Step 4 => Control Specification



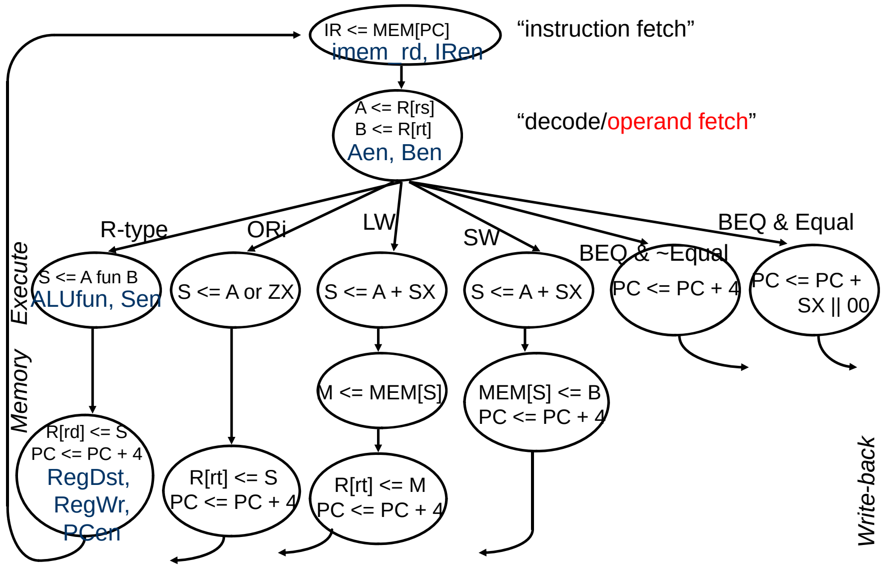
Traditional FSM Controller



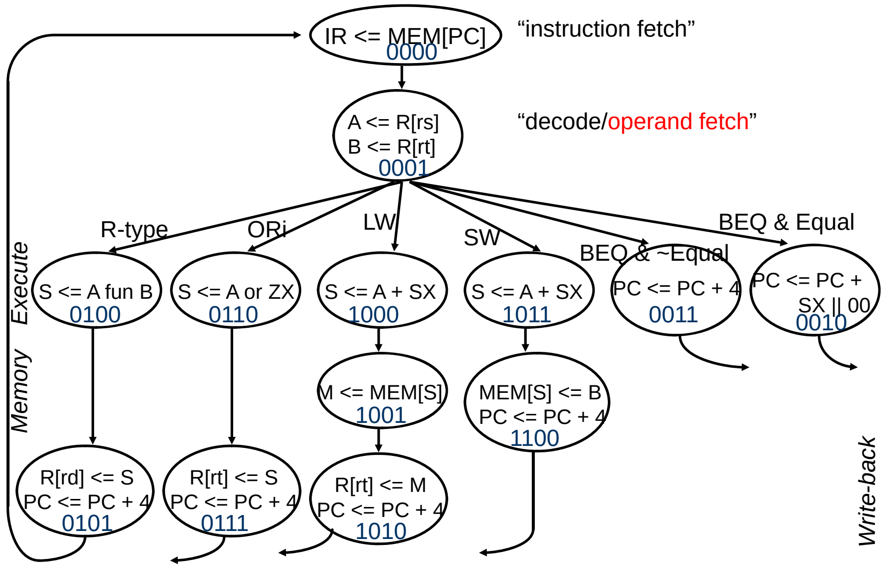
Step 5: datapath + state diagram => control

- 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			
	0001	BEQ	1	0010			1 1			
	0001	R-type	x	0100			1 1	-all same in Moore machine		
	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

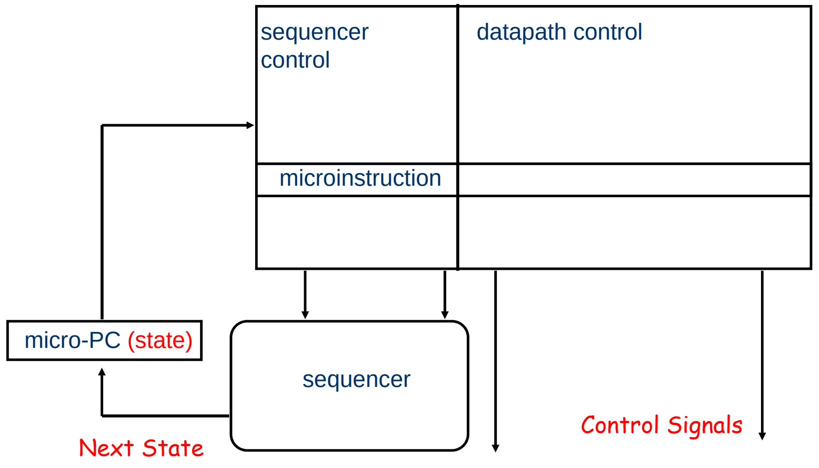
- What is the average CPI?
  - state diagram gives CPI for each instruction type
  - workload gives frequency of each type

Type	CPI_i for type	Frequency	CPI_i x $freqI_i$
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

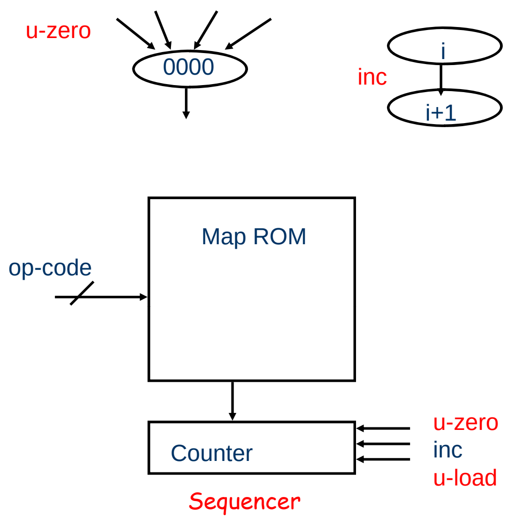
Controller Design

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

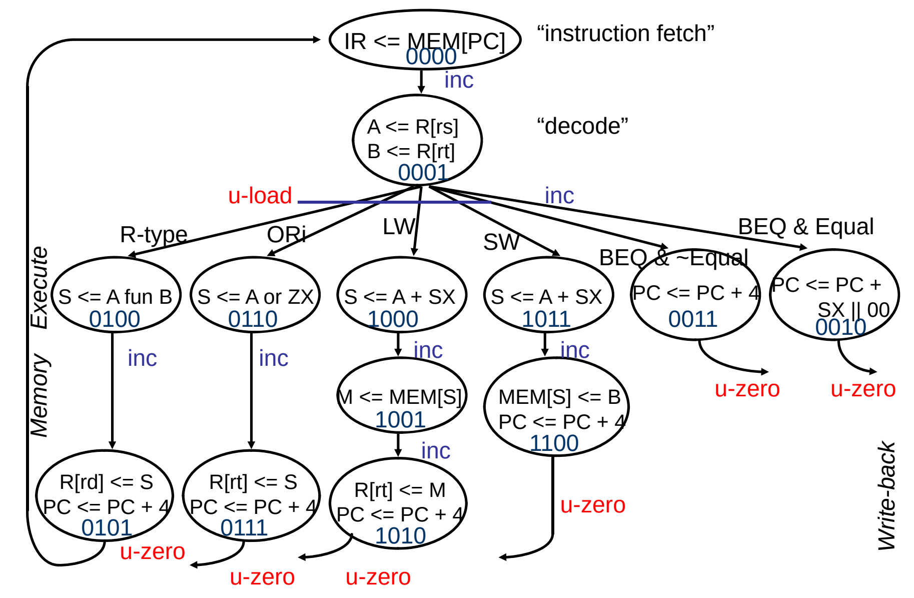
Controller Design (cont.)



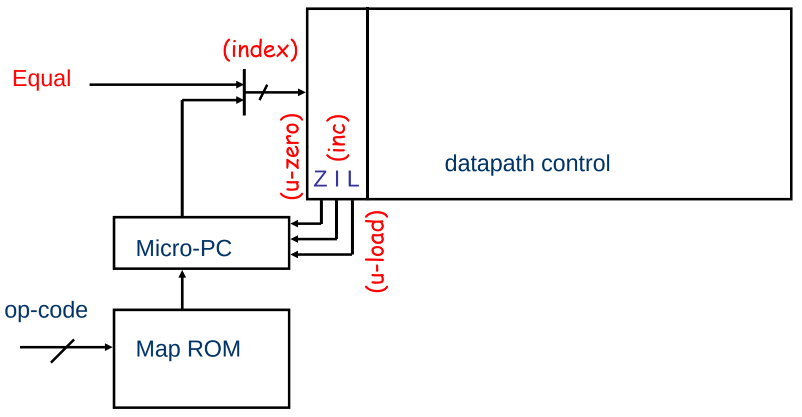
Example: Jump-Counter



Using a Jump Counter



Our Microsequencer

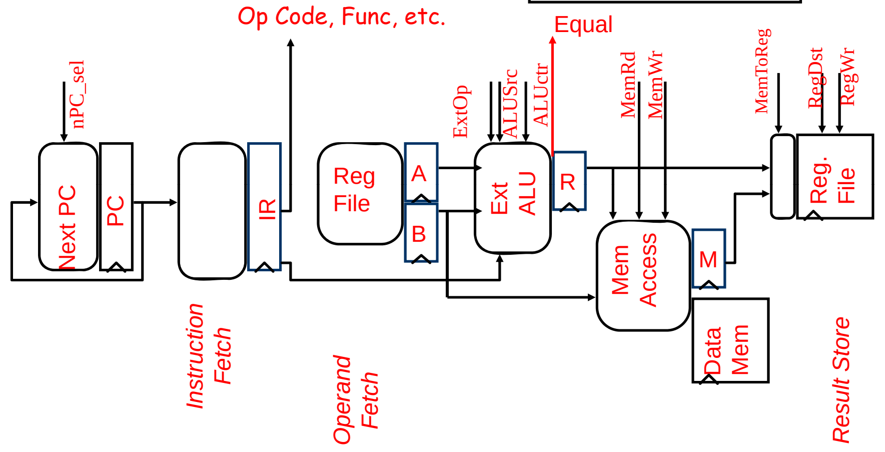


Microprogram Control Specification

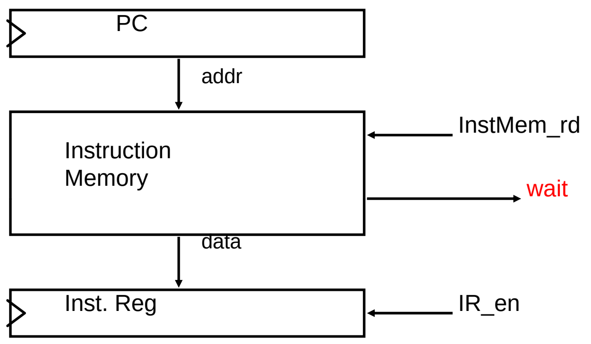
	µPC	Equal	Next	IR	PC en sel	Ops A B	Exec Ex Sr ALU S	Mem R W M	Write-Back M-R Wr Dst
	0000	?	inc	1					
	0001	0	u-load						
	0001	1	inc						
	0010	x	u-zero		1 1				
BEQ	0011	x	u-zero		1 0				
R:	0100	x	inc				0 1 fun 1		
	0101	x	u-zero		1 0				0 1 1
ORi:	0110	x	inc				0 0 or 1		
	0111	x	u-zero		1 0				0 1 0
LW:	1000	x	inc				1 0 add 1		
	1001	x	inc					1 0 0	
	1010	x	u-zero		1 0				1 1 0
SW:	1011	x	inc				1 0 add 1		
	1100	x	u-zero		1 0			0 1	

Mapping ROM

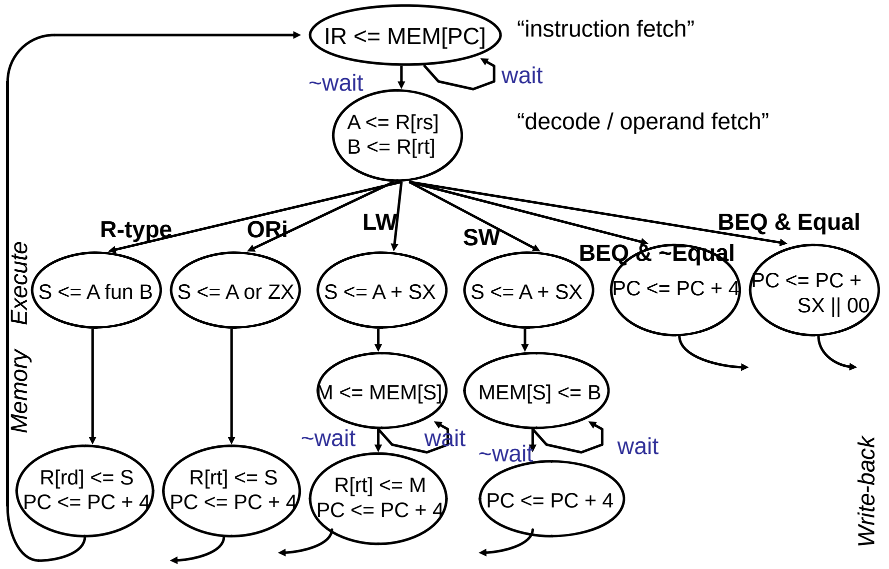
	Op Code	Micro-PC
R-type	000000	0100
BEQ	000100	0011
ori	001101	0110
LW	100011	1000
SW	101011	1011



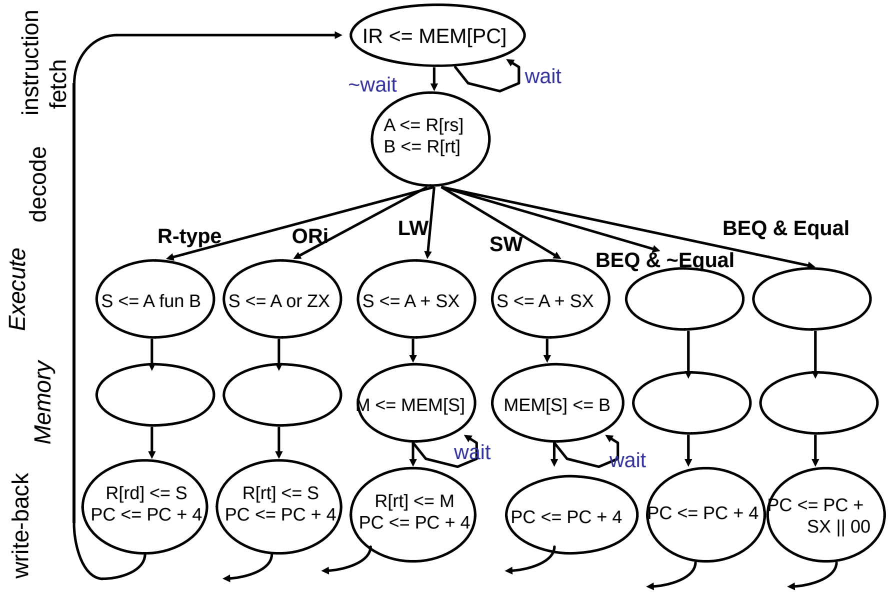
Example: Controlling Memory



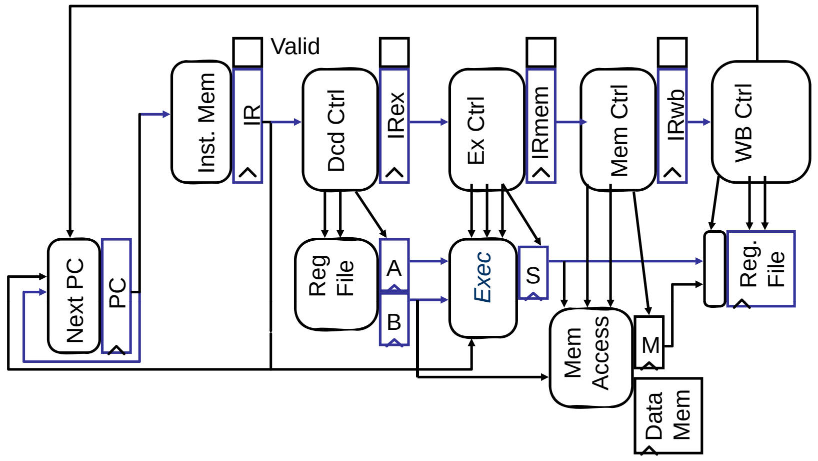
Handle non-ideal memory



Really Simple Time-State Control

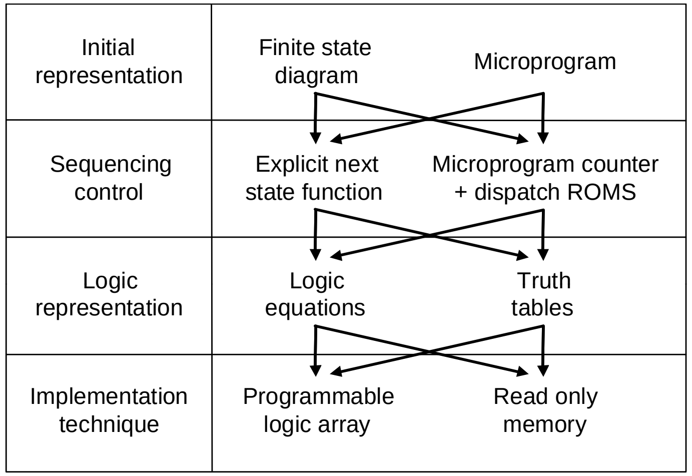


Time-state Control Path



- Local decode and control at each stage

The Big Picture



Initial representation	Finite state diagram	Microprogram
Sequencing control	Explicit next state function	Microprogram counter + dispatch ROMS
Logic representation	Logic equations	Truth tables
Implementation technique	Programmable logic array	Read only memory

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

Summary (cont.)

- 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.)

- 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

Summary (cont.)

- 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!