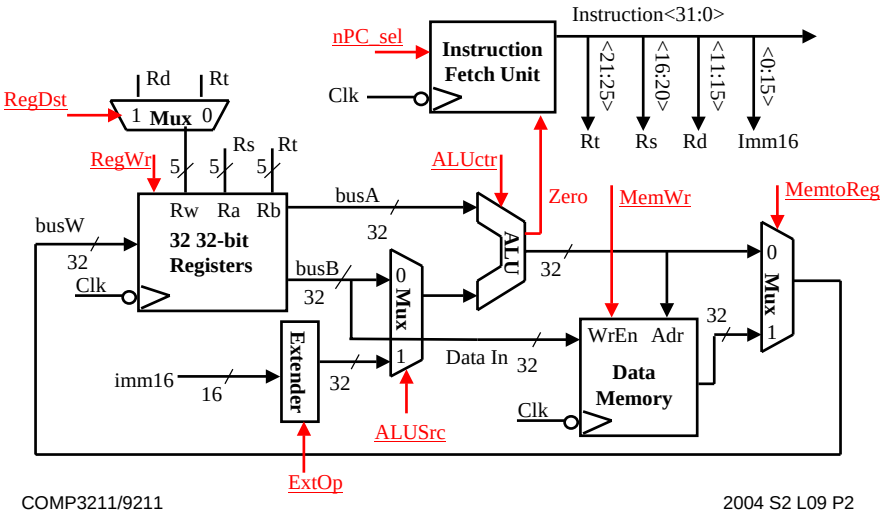
L09: Multicycle Processor Design

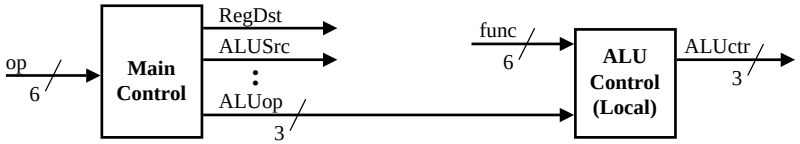
Adapted from:

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

Recap: A Single Cycle Datapath

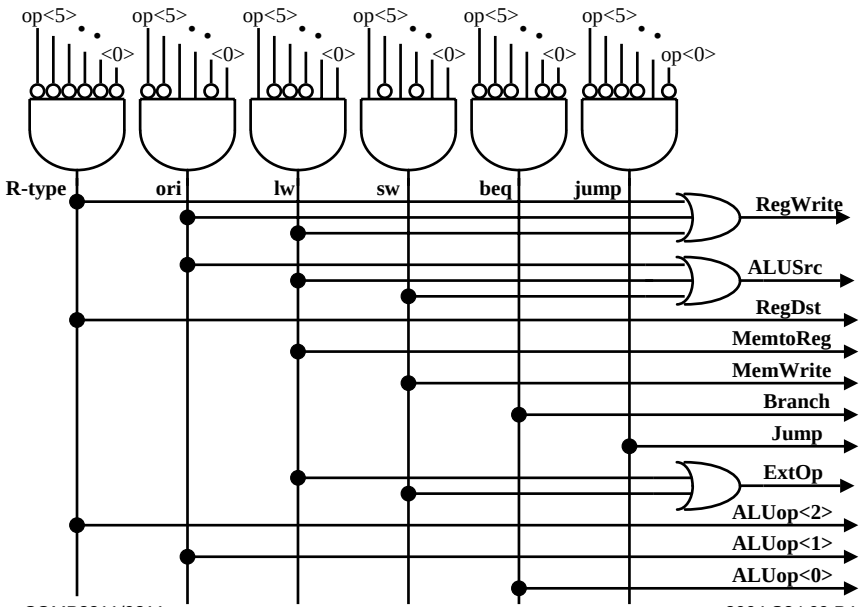


Recap: The "Truth Table" for the Main Control

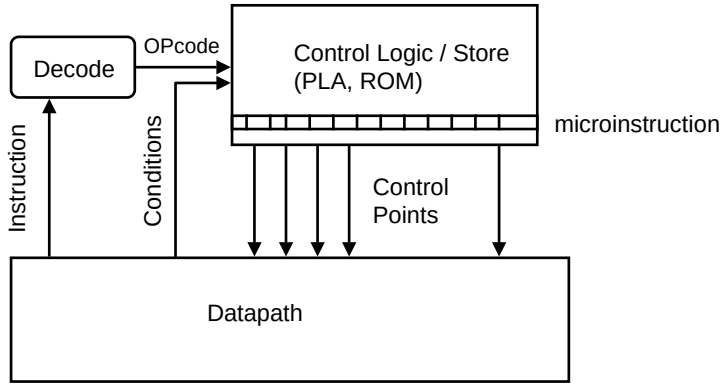


op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx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

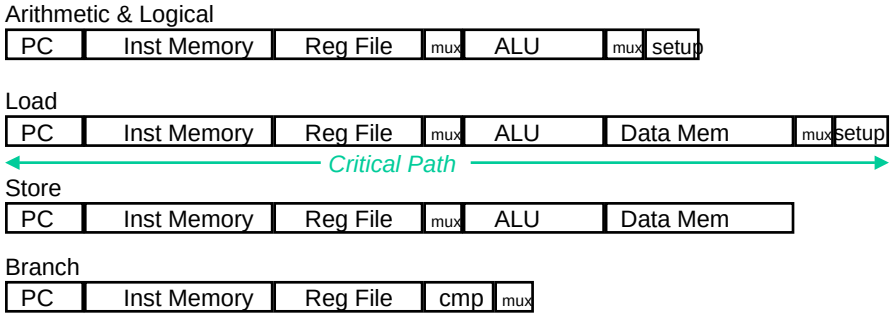


Recap: Systematic Generation of Control



- **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)**

What's wrong with our CPI=1 processor?

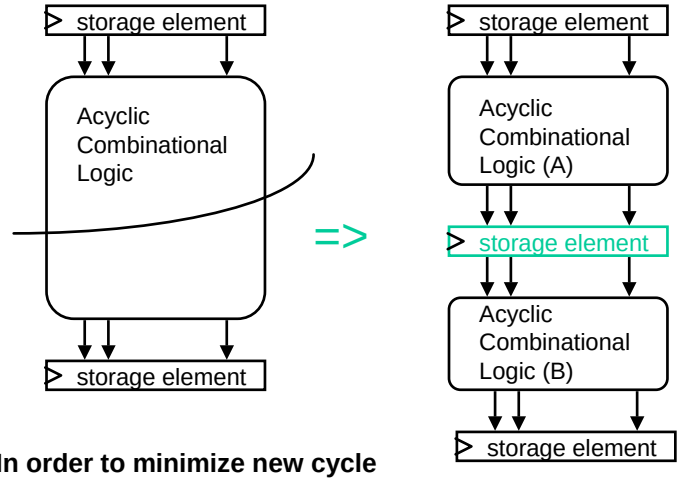


Problems:

1. **Long cycle times due to long combinational delays**
2. **All instructions take as much time as the slowest**
3. **Real memory is not so nice as our idealized memory**
   - **cannot always get the job done in one (short) cycle**

Approach to Reducing Cycle Times (Solving 1)

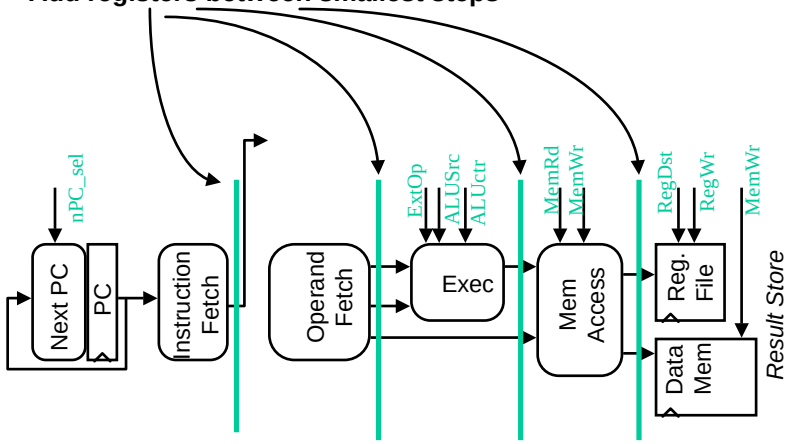
- **Cut combinational dependency graph and insert register / latch**
- **Do same work in two fast cycles, rather than one slow one**



- **In order to minimize new cycle time we have to balance delays of circuits A & B**

Partitioning the CPI=1 Datapath

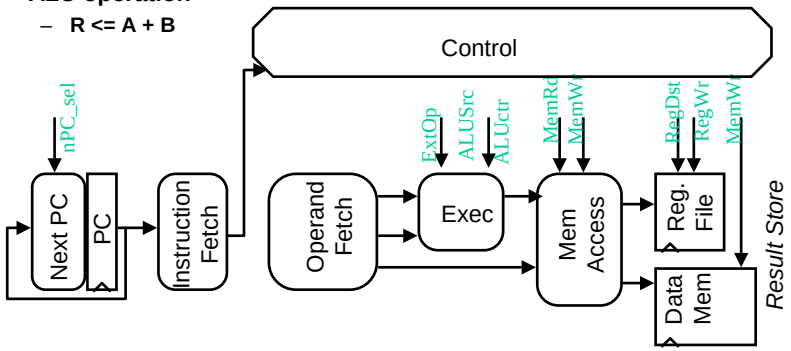
- **Add registers between smallest steps**



- **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**

Basic Limits on Cycle Time

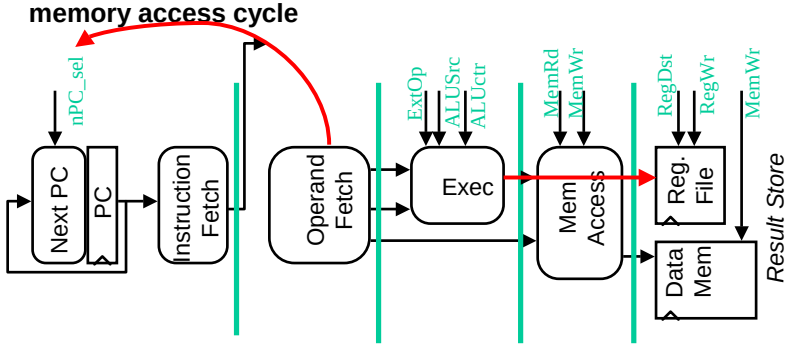
- Next address logic
  - PC <= branch ? PC + offset : PC + 4
- Instruction Fetch
  - InstructionReg <= Mem[PC]
- Register Access
  - A <= R[rs]
- ALU operation
  - R <= A + B
- Memory access
  - R <= Mem[EA]
- Result store
  - R[rd|rt] <= R
  - Mem[EA] <= B



Tailor ex. steps to instruction needs (Solving 2)

Examples:

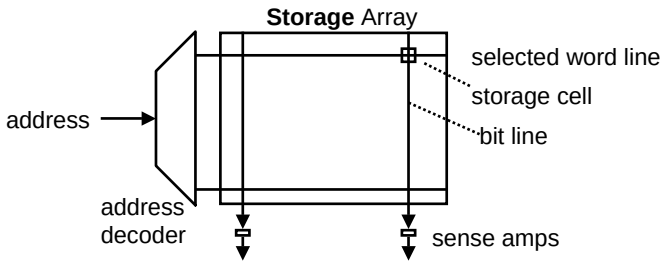
- R-type instructions don't access memory, so no need to have memory access cycle



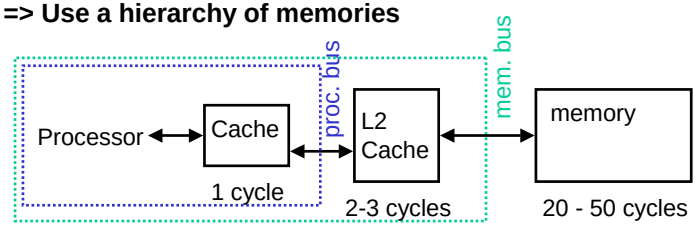
- If the register file can do quick comparisons on register contents, branches don't need to perform ALU operations

Coping with long memory access times (Solving 3)

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

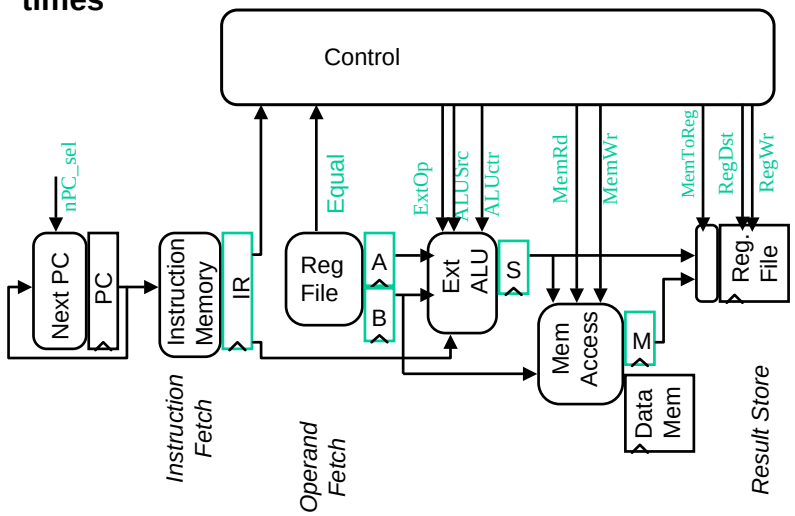


- => Use a hierarchy of memories



Multicycle Datapath Design

- Note insertion of registers to allow for shorter cycle times



Multicycle datapath control

- In a single cycle processor, each instruction is realized by exactly one set of control signals or control points
- In a multicycle processor, each instruction is completed in several cycles
  - each cycle has different control signals
  - instructions are realized by a sequence of control signal sets
- Follow the usual processor design steps to derive control logic

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

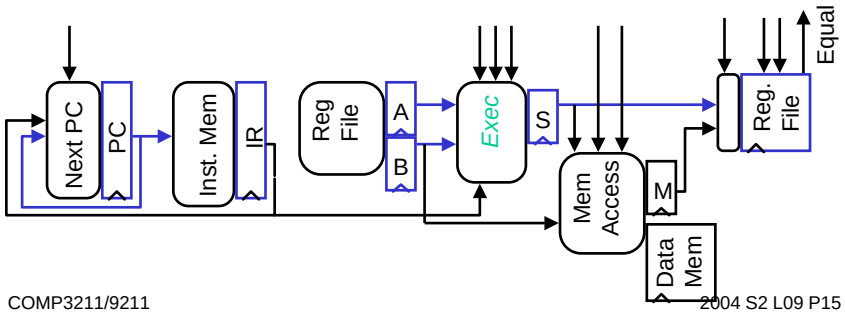
Step 4: R-type (add, sub, . . .)

Logical Register Transfer

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

Physical Register Transfers

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



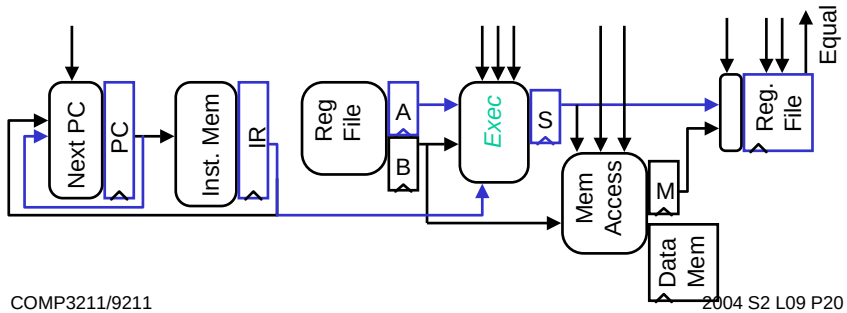
Step 4:Logical immed

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



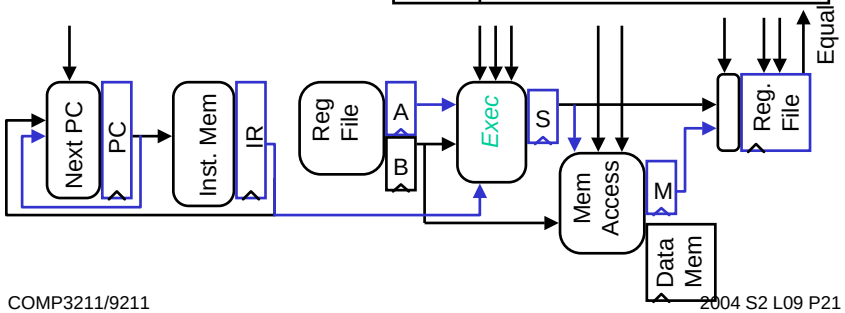
Step 4 : Load

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



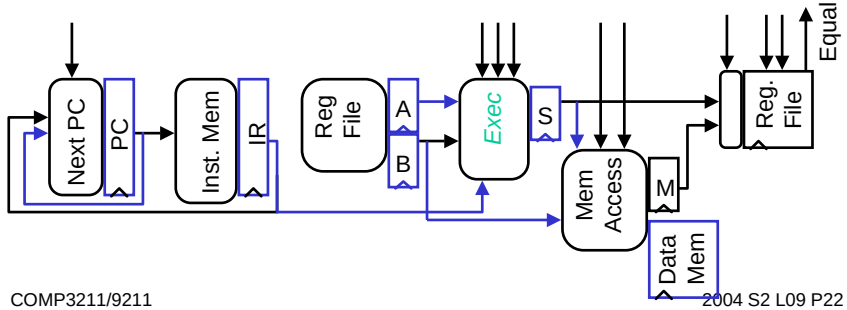
Step 4 : Store

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



Step 4 : Branch

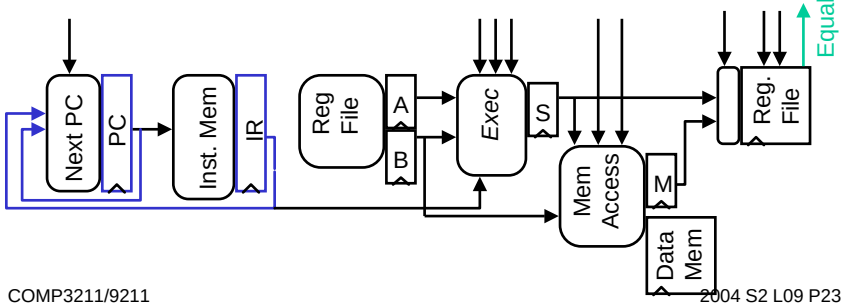
Logical Register Transfer

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

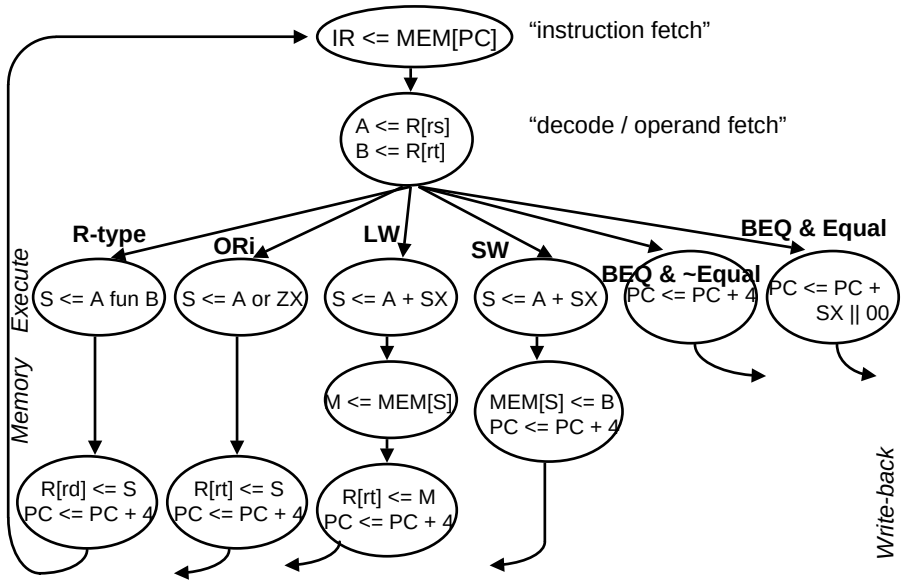
Physical Register Transfers

inst	Physical Register Transfers
	IR <– MEM[pc]
	A <– R[rs]; B <– R[rt]
BEQ|$\overline{\text{Eq}}$	PC <– PC + 4

inst	Physical Register Transfers
	IR <– MEM[pc]
	A <– R[rs]; B <– R[rt]
BEQ|Eq	PC <– PC + sx(Im16) || 00

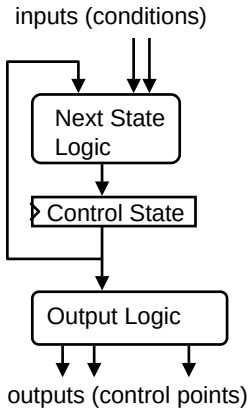


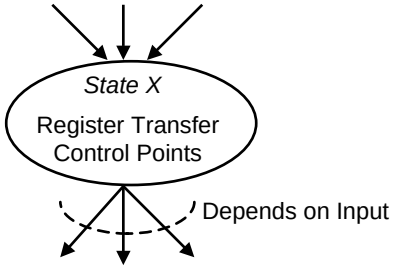
Step 4 => Control Specification for multicycle proc



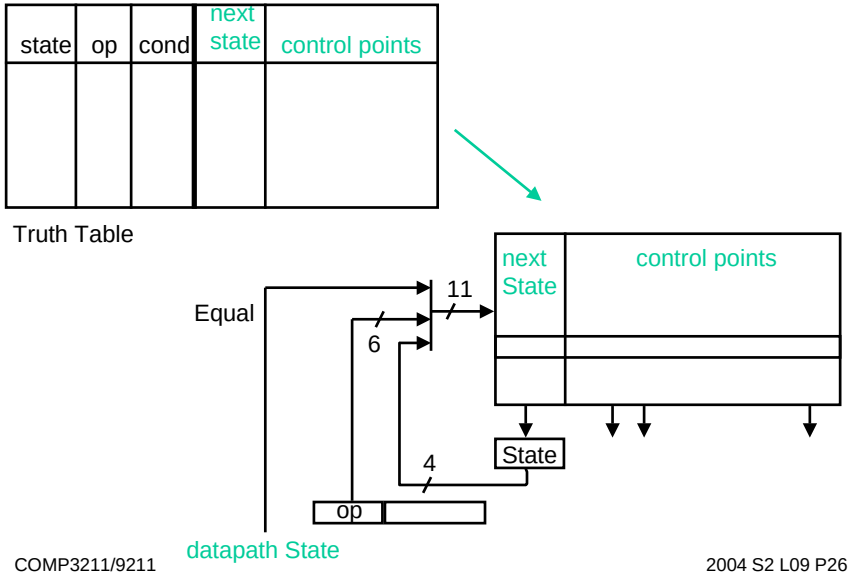
Our Control Model

- **State specifies control points and sets up Register Transfer**
- **Transfer occurs upon exiting state (same falling edge)**





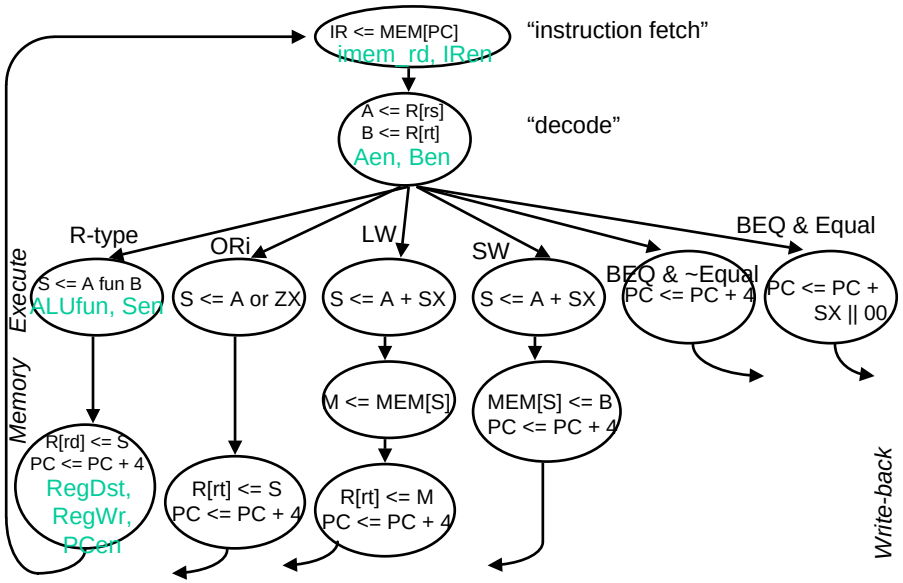
Traditional FSM Controller



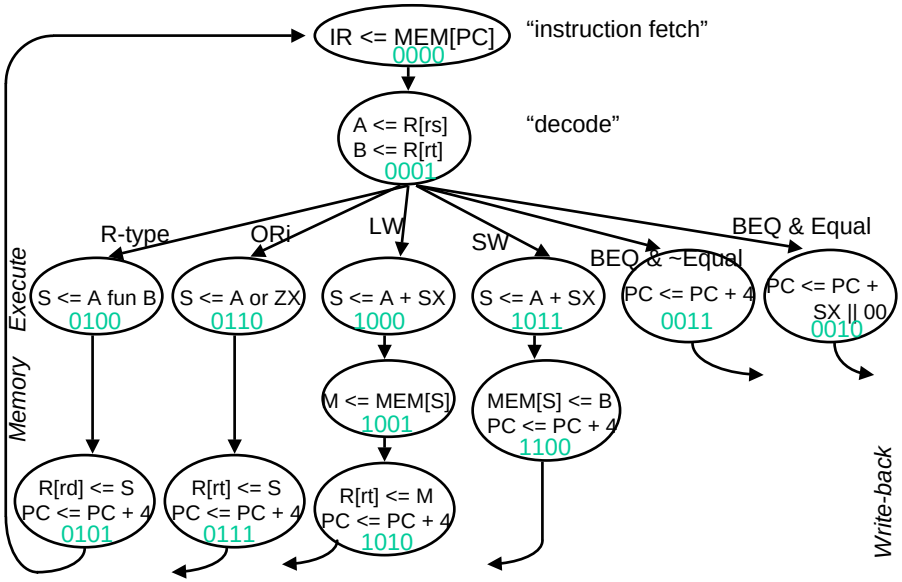
Step 5: datapath + state diagram => control

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

Mapping RTs to Control Points



Assigning States



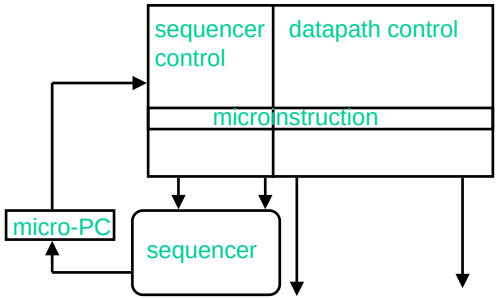
Detailed Control Specification

	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			
	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 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	

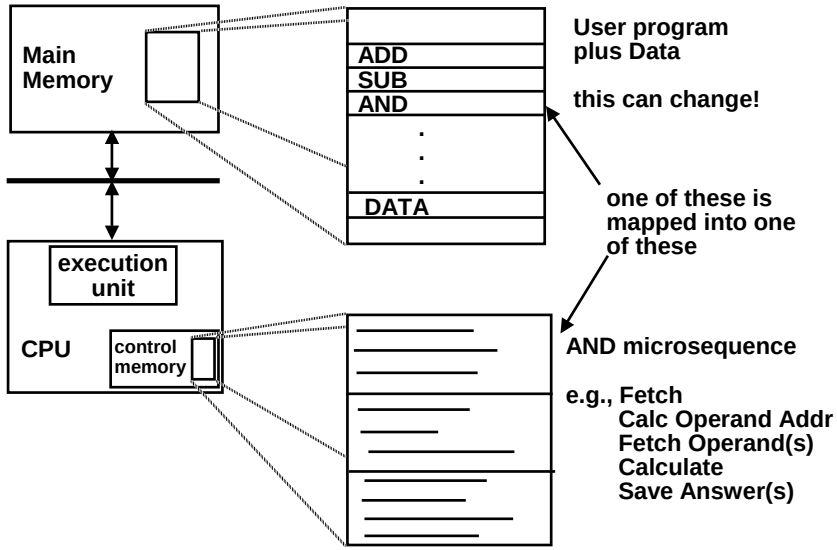
Recall that for a given state, all outputs are the same in Moore machine

Controller Design

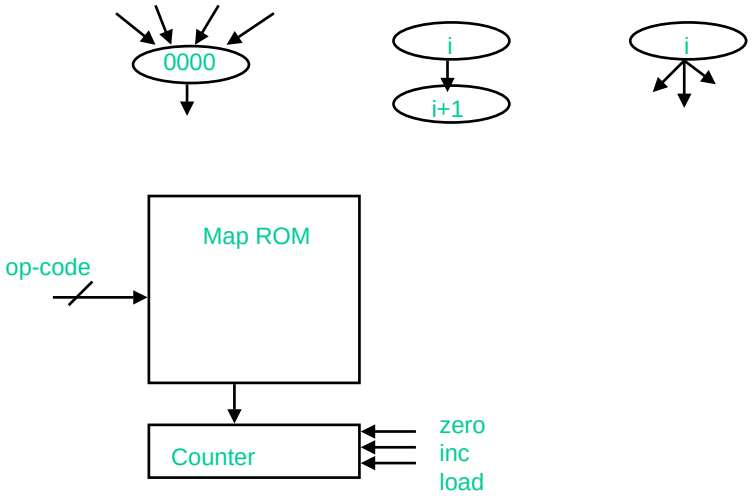
- **The state diagrams that define the controller for an instruction set processor are highly structured**
- **Use this structure to construct a simple "microsequencer"**
- **Control reduces to programming this very simple device**
  - **Microprogramming**



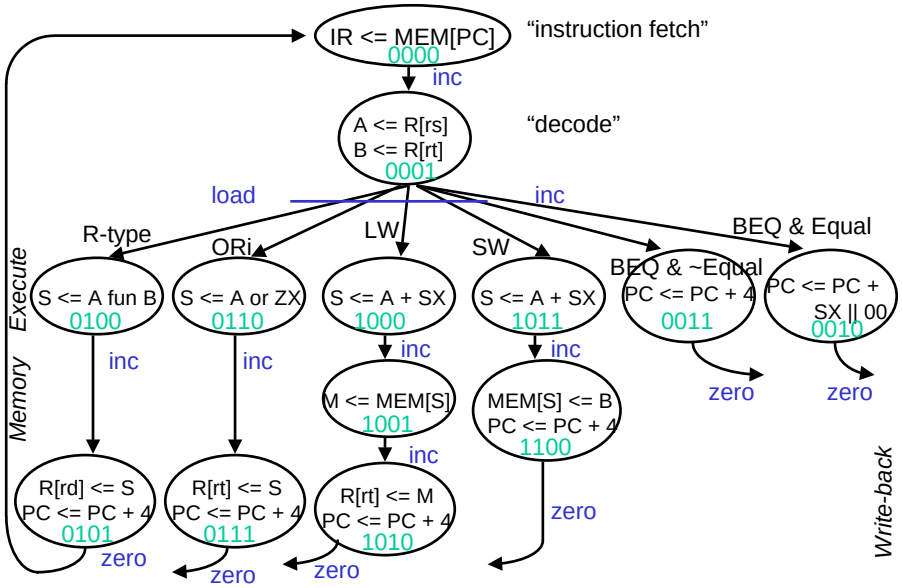
"Macroinstruction" Interpretation



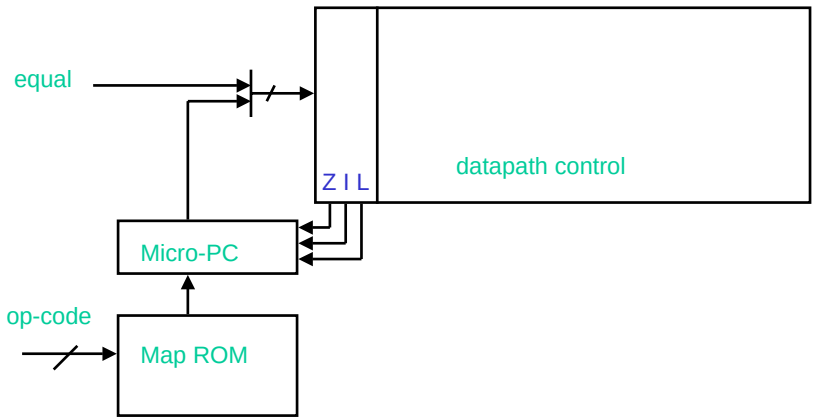
Example: Jump Counter



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			
	0001	1	inc			1 1			
	0010	x	zero		1 1				
BEQ:	0011	x	zero		1 0				
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	

Mapping ROM

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

Exercise

For the given micro-sequencer, 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)
```

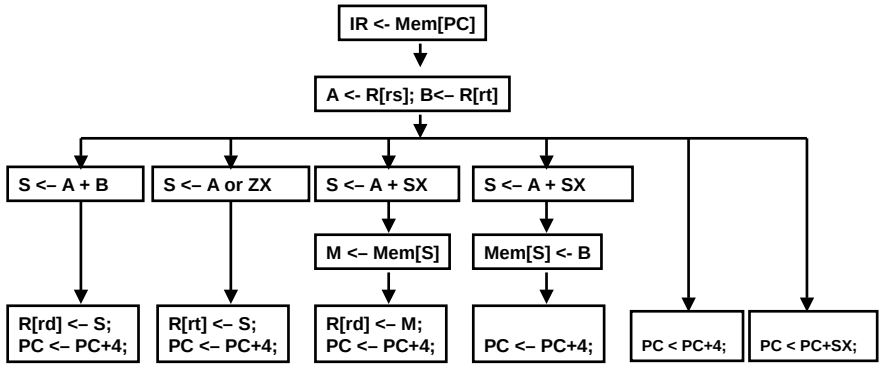

Performance Evaluation

- **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

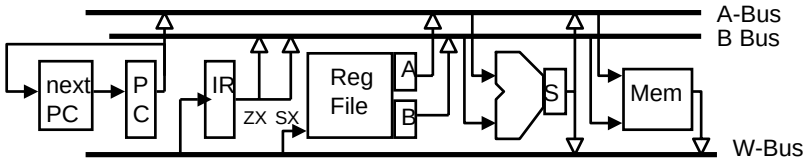
- **Improvement achieved if**
  4.1 * CC(multicycle) < 1 * CC(single cycle)

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**

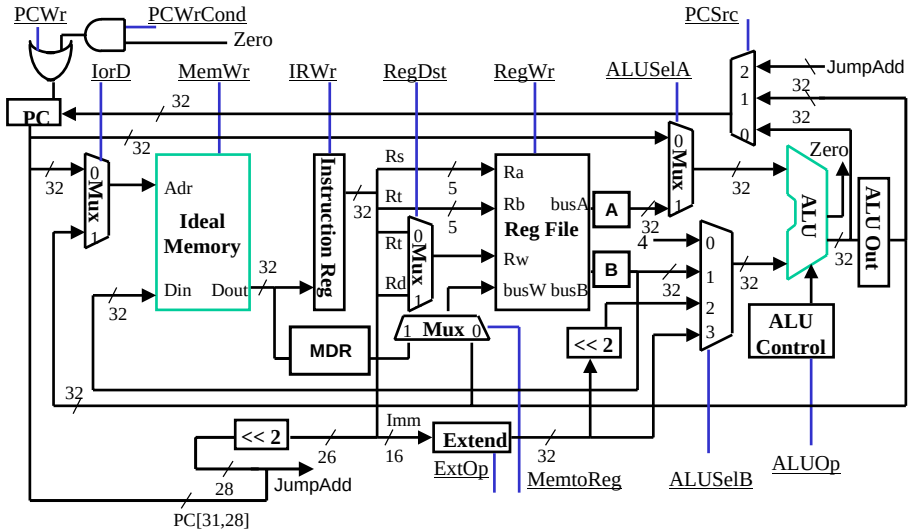
"Princeton" Organization: Making better use of memory



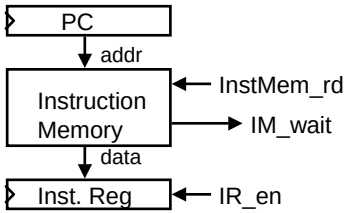
- **Single memory for instruction and data access**
  - **memory utilization → 1.4/4.1**
- **In this case our state diagram does not change**
  - **several additional control signals**
  - **must ensure each bus is only driven by one source on each cycle**

Alternative representation of multicycle datapath

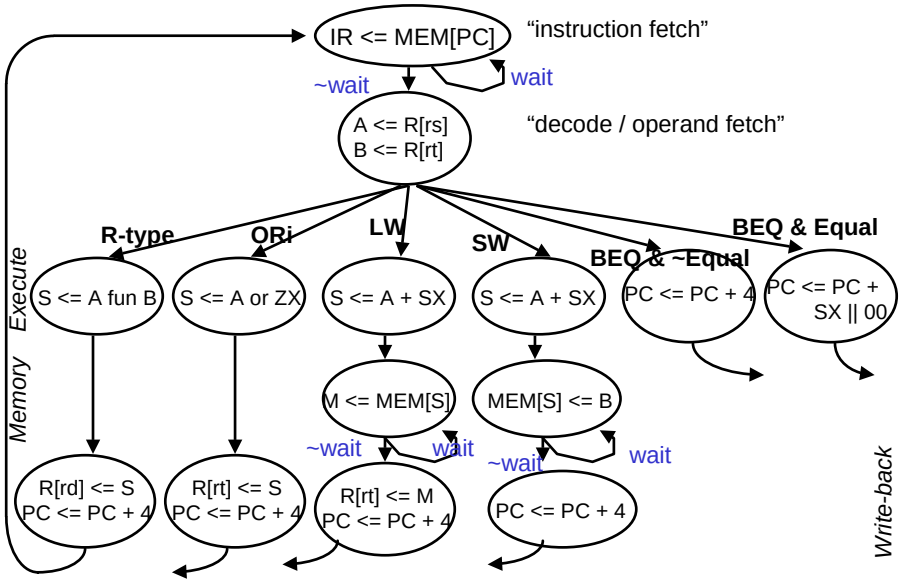
- **Miminizes Hardware: 1 memory, 1 adder**



Controlling Memory

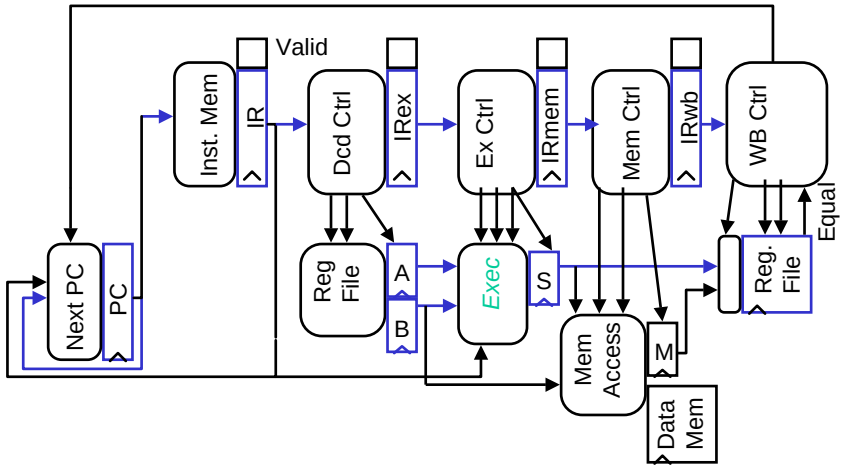


Controller handles non-ideal memory



Time-state Control Path

- Another approach to control implementation
- Local decode and control at each stage

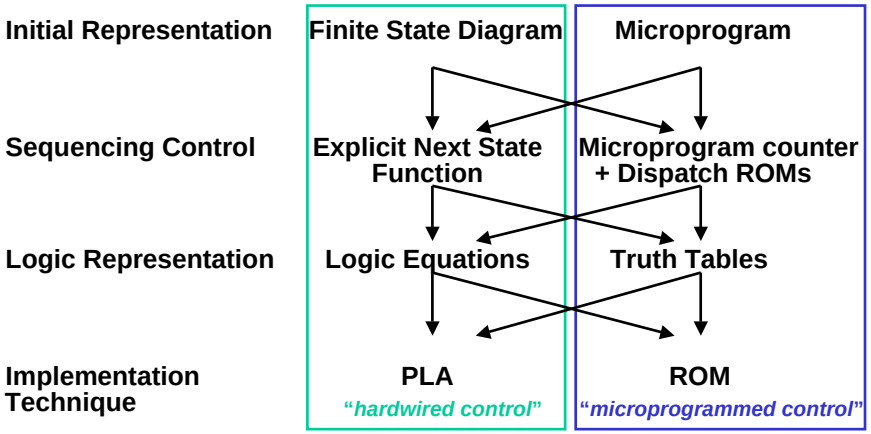


Summary

- 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)