Single Cycle Control

- Very simple
  - Control signals are functions of opcode and possibly function fields
  - Combinational logic suffices

- Ex: RegWrite
  - Asserted on `R-type`, `lw`
  - Deasserted on `beq`, `sw`, `j`

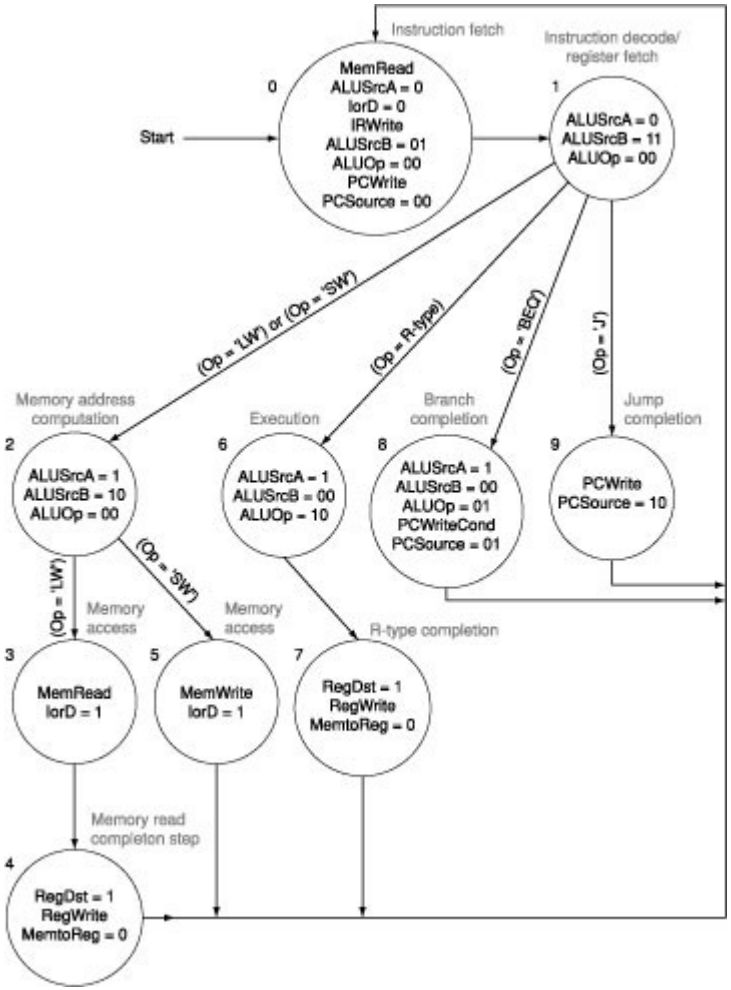
Multi-Cycle Control

- Much harder
  - Control signals depend on instruction and cycle
- Consider RegWrite

	Cycle				
Instruction	Fetch	Decode	Execute	Memory Access	Write Back
R-Type	0	0	0	1	
sw	0	0	0	0	
lw	0	0	0	0	1

- CPU must “remember” what cycle it is in
  - Control unit must maintain state
  - Several ways to do this

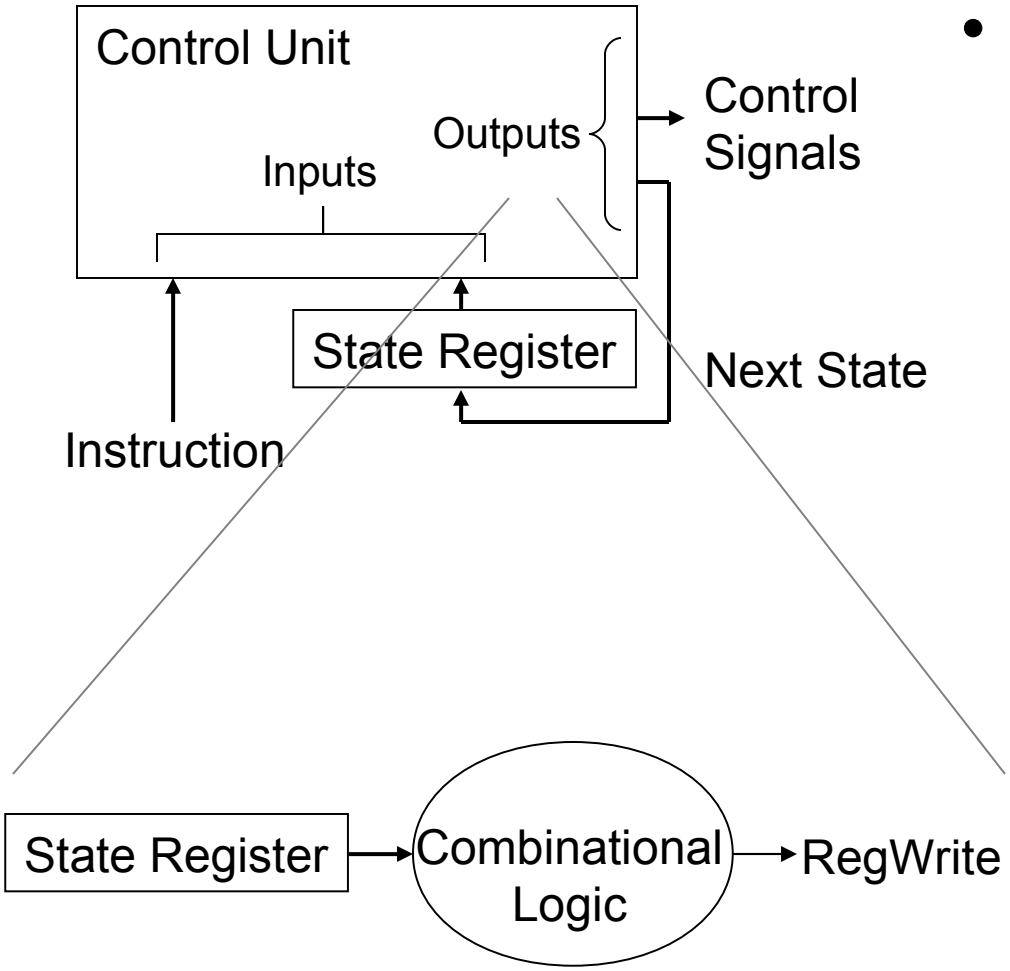
Review


Instruction fetch
Instruction decode/
register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
Execution
Branch
completion
Jump
completion
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Memory read
completon step
4
RegDst = 1
RegWrite
MemtoReg = 0

Multi-Cycle Control Implementation

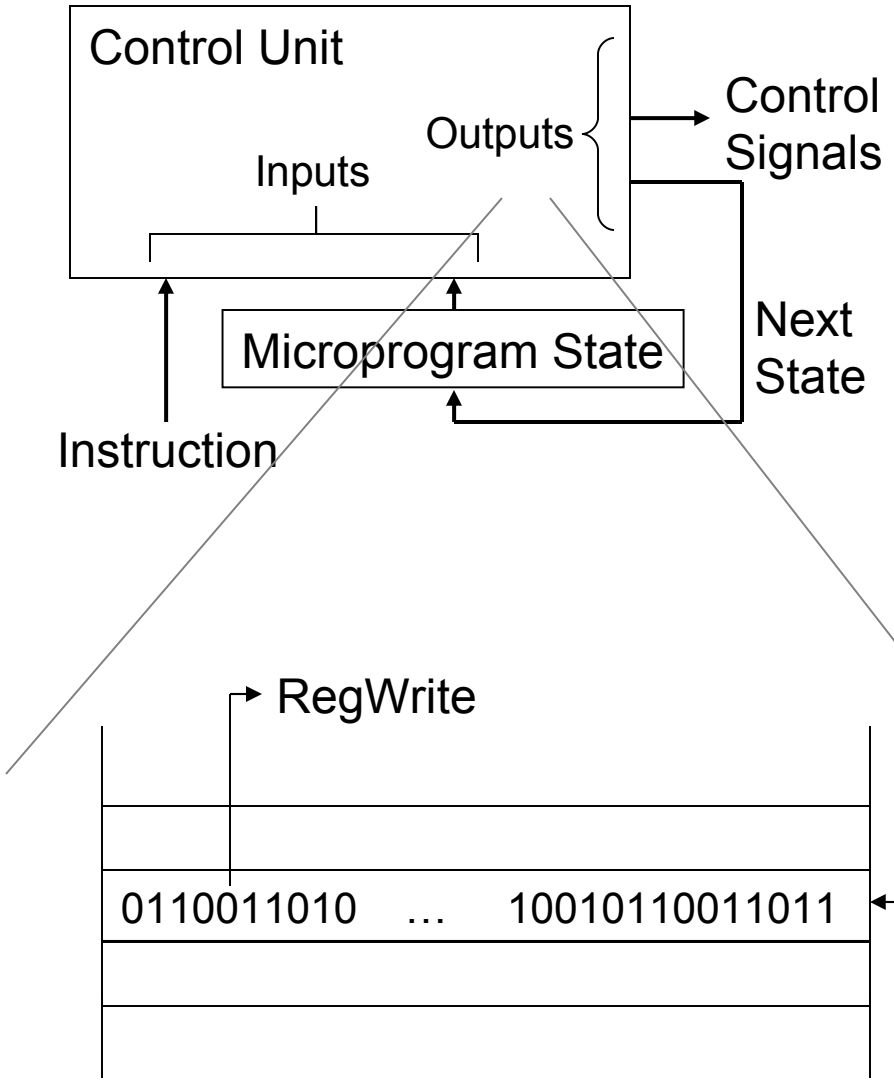
- Two main control implementations
  - State machine
    - Translate finite state machine diagrams to hardware
    - Control signals function of current state
  - Microprogram
    - A small control program runs in parallel to CPU datapath
    - Program outputs are control signals
- Logically similar in many respects
  - Control “remembers” state and changes signals
  - Implementation very different
  - Combinations also possible

State Machine Control



- From digital design:
  - Create state machine
  - Assign state values
  - Derive control signal functions
  - Derive next state functions

Microprogram Control



- Control Unit is now an indexed ROM
  - Memory bits set control signals and next state
  - Microprogram state and instruction select the memory value

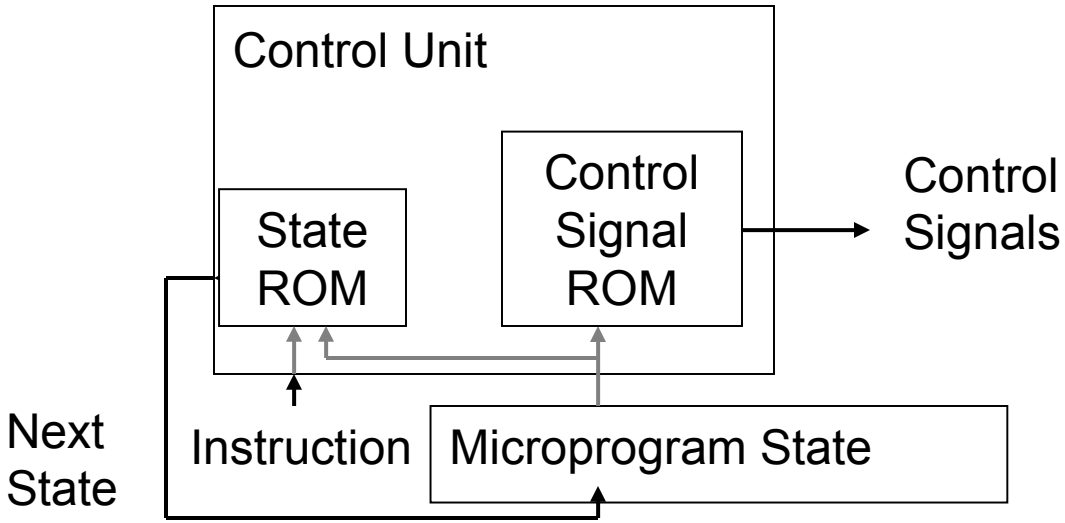
Single Microprogram ROM

- Inputs
  - 6 bits from instruction opcode
  - 4 bits from current state
- Outputs
  - 16 bits for control signals
  - 4 bits for next state
- ROM Size
  - 2^{10} 20-bit words
  - Total size: 20 kbits

Single Microprogram ROM

- ROM requires 20 kbits
- Many entries are redundant
  - Decode cycle always follows fetch cycle
    - Addresses XXXXXX 0000 always contain same value
      - 2^6-1 = 63 wasted entries
  - Some inputs invalid
    - Addresses 111111 XXXX indicate invalid instruction
      - 2^4-1 = 15 wasted entries per bad opcode
- Save space by dividing control into two ROMs
  - One ROM generates the next state
  - One ROM generates the control signals

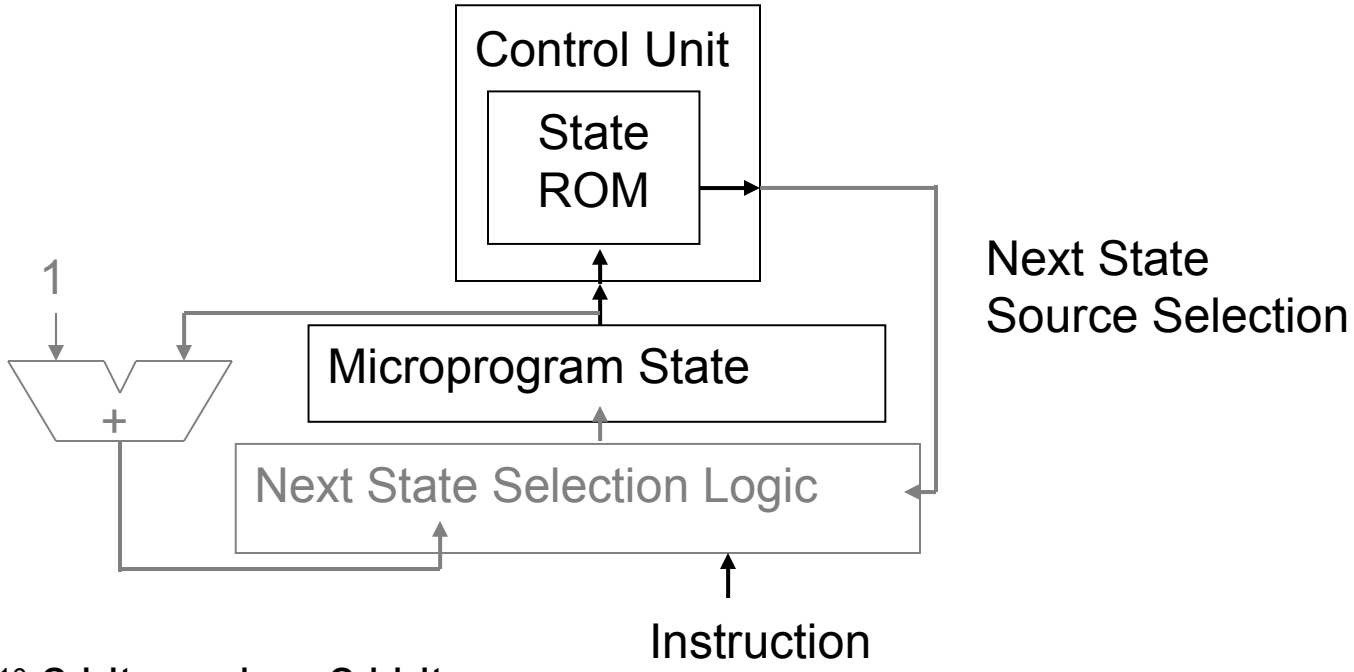
Two Microprogram ROMs



- Total ROM Size
  - 2^{10} 4 bit words for State ROM
  - 2^{4} 16 bit words for Control ROM
  - 4096 + 256 = 4352 ≈ 4.3 kbits

Microprogram State ROM

- The State ROM still wastes space
  - Most states increment to next value
  - Select source of next state in microinstruction

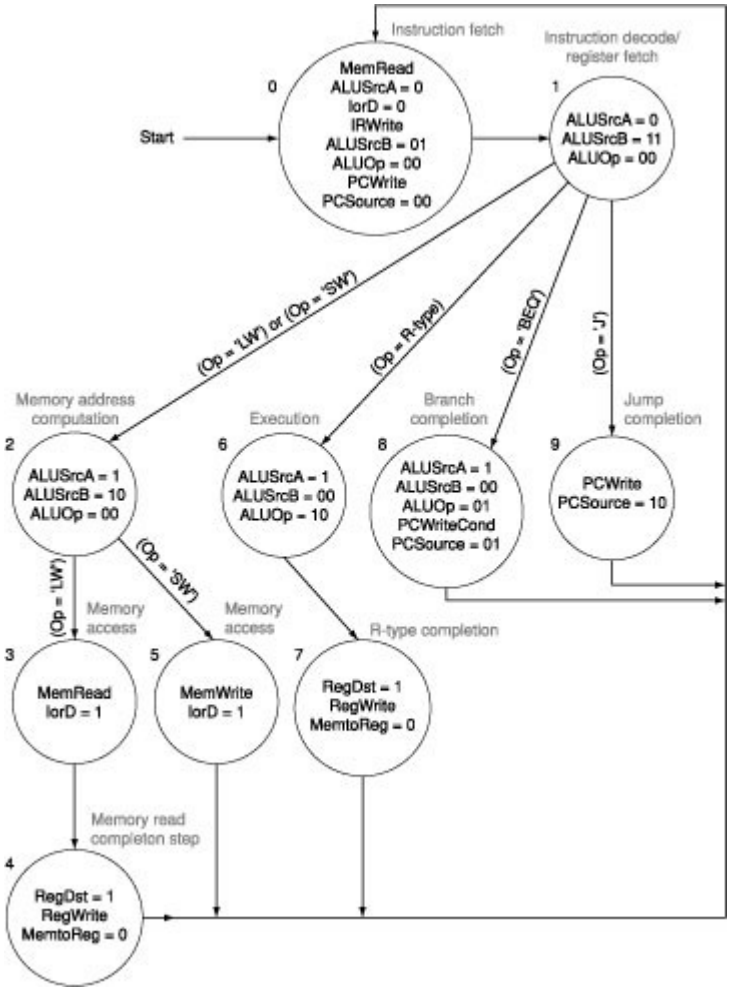


-
-
- ROM Size 2^{10} 2 bit words = 2 kbits

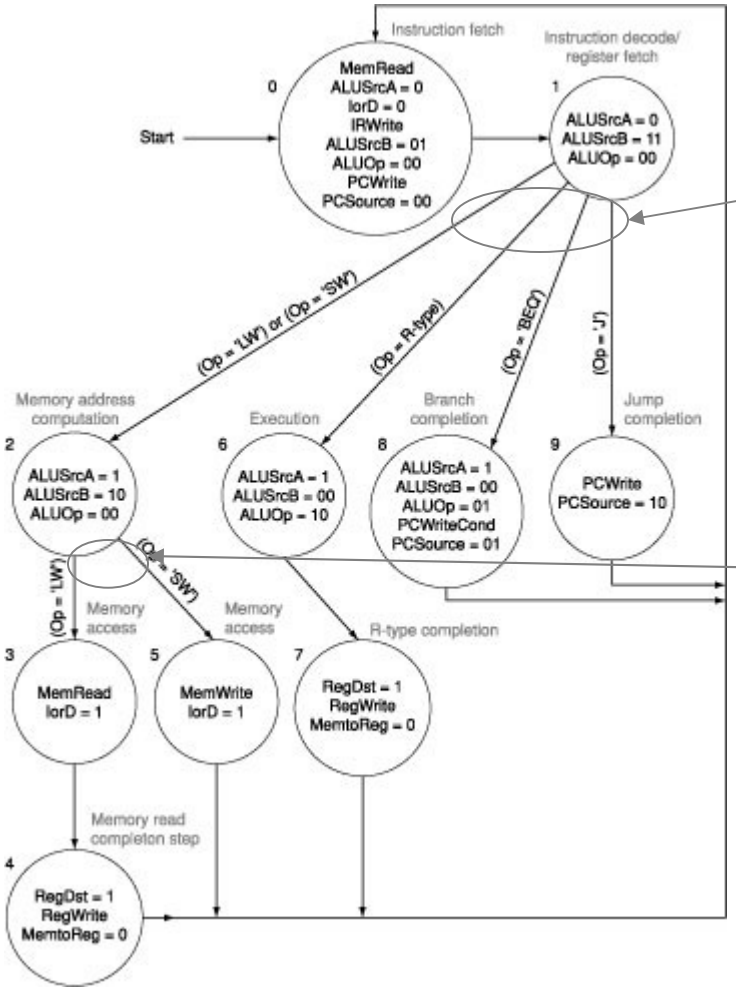
Next State Selection Logic

- How to select next non-sequential state?
  - Combinational logic from opcode and current state
  - Use more ROMs!
- Use dispatch ROM to “jump” in microprogram
  - opcode used as input to dispatch table
  - Next state produced as output
  - Use one dispatch table per decision point

Dispatch ROM Points



Dispatch ROM Points



Dispatch ROM 1

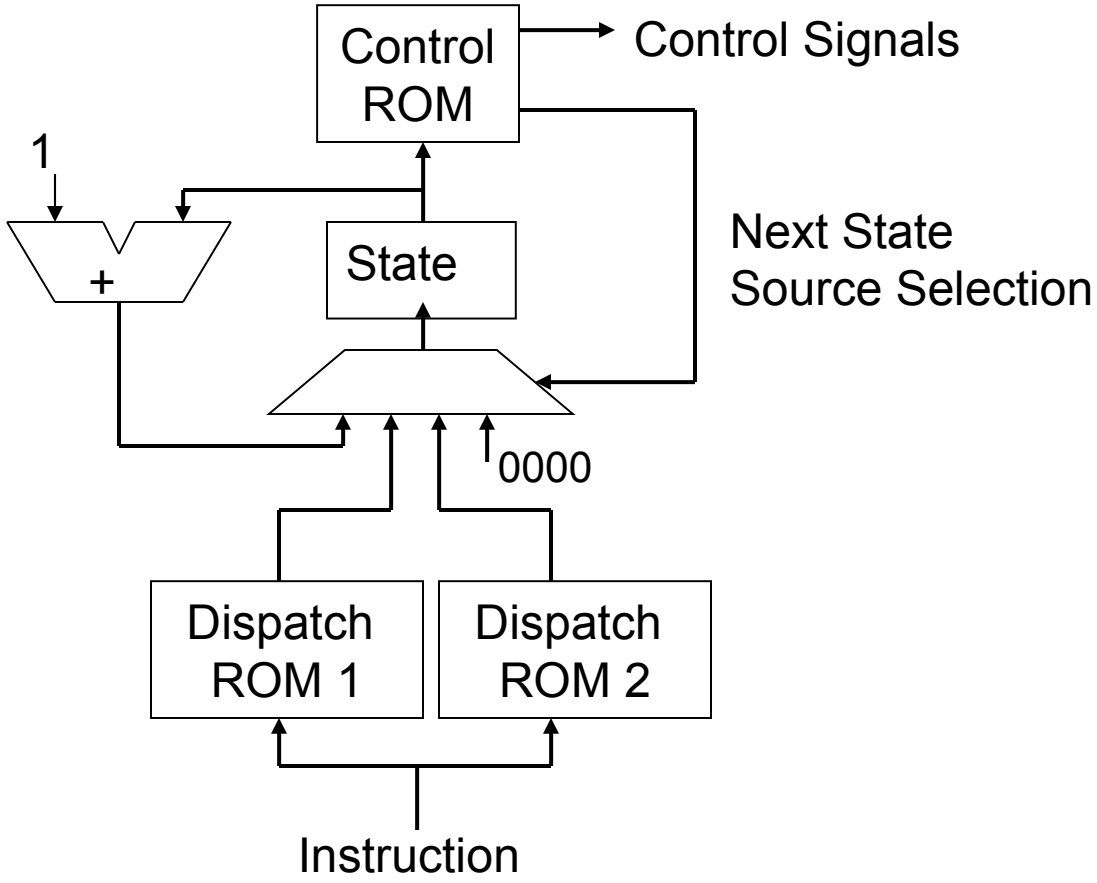
Dispatch ROM 2

Microprogram ROM Size

- Control signal ROM
  - 2^4 16 bit words
  - 256 bits
- Next state ROM
  - 2^4 2 bit words
  - 32 bits
- Dispatch ROMs
  - 2 ROMs at 2^6 4 bit words
  - 512 bits
- Total ROM size = 800 bits
- Still many other ways to reduce size

Same inputs, so combine into a single ROM.

Final Microprogram Control Design



Control ROM Format

- Each entry 18 bits wide
  - 16 leftmost bits drive control signals
    - Order signals according to Figure C.3.6
      - PCWrite = M[17]

        …
      - RegDst = M[2]
  - 2 rightmost bits select next state source (M[1:0])
    - 00: Next state = 0000
    - 01: Use dispatch ROM 1
    - 10: Use dispatch ROM 2
    - 11: Increment state

Microcode Generation

- Creating the memory entries manually is difficult and error prone for real designs
  - Analogous to programming in machine code
- Microassemblers can perform that function
- What “language” to use?
  - Microinstructions
    - Similar to assembly language
    - See section 5.7
  - Register Transfer Language
    - Describe operations at the hardware level
    - For example, A <= Reg[IR[25:21]]

Fetch: Control Signals in Microcode

- All the signals
- Some are don't care
  - MemtoReg
  - RegDst
- Some, not listed in FSM diagram, need a value
  - MemWrite
  - RegWrite

Signal	Value
PCWrite	1
PCWriteCond	0 or X?
IorD	0
MemRead	1
MemWrite	0
IRWrite	1
MemtoReg	X
PCSource1	0
PCSource0	0
ALUOp1	0
ALUOp0	0
ALUSrcB1	0
ALUSrcB0	1
ALUSrcA	0
RegWrite	0
RegDst	X

Fetch: Next State Selection

- Decode always comes after Fetch
  - Next state control = 11 (increment)
  - Assumes Decode State = Fetch State + 1

Fetch Cycle Microcode Entry

- Assume Fetch cycle assigned 0000
- Now, combine control signal portion with next state selection
- Control ROM entry for address 0000

- MIF Entry

1001010000001000 11

0x0 : 0x25023;

Encoding Control Signals

- Consider the IRWrite signal

	0000	0001	0010	0011	0100	0101	0110	0111	1000
IRWrite	1	0	0	0	0	0	0	0	0

- Only asserted during Fetch cycle
- Can we save space by combining signals to generate IRWrite or remove IRWrite?
  - No signal duplicates IRWrite, so no replacement
  - Consider the PCWrite and MemRead signals

	0000	0001	0010	0011	0100	0101	0110	0111	1000
PCWrite	1	0	0	0	0	0	0	0	1
MemRead	1	0	0	1	0	0	0	0	0

  - See any pattern?