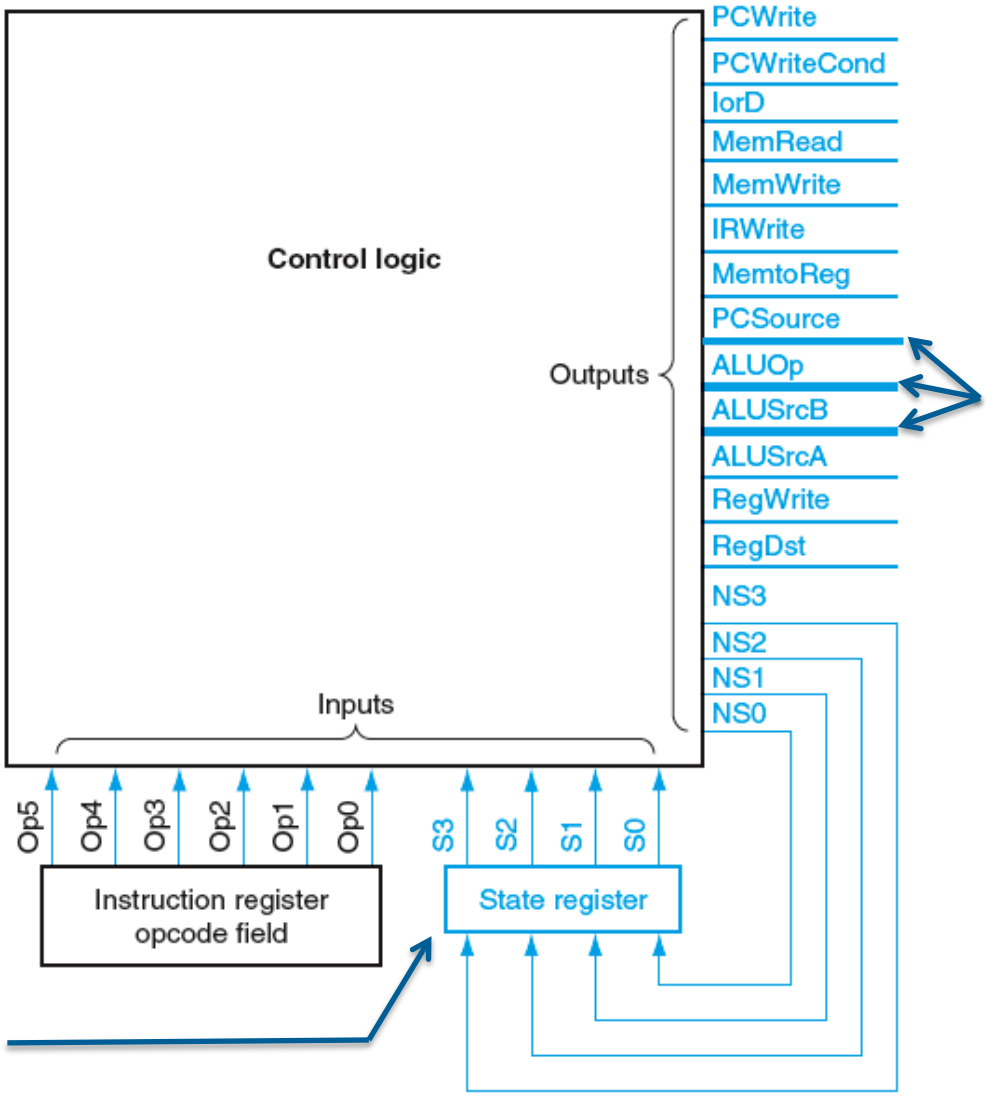
# MIPS Control Unit




2-bit signals
others are 1 bit each

State register holds state number

# MIPS Control Unit

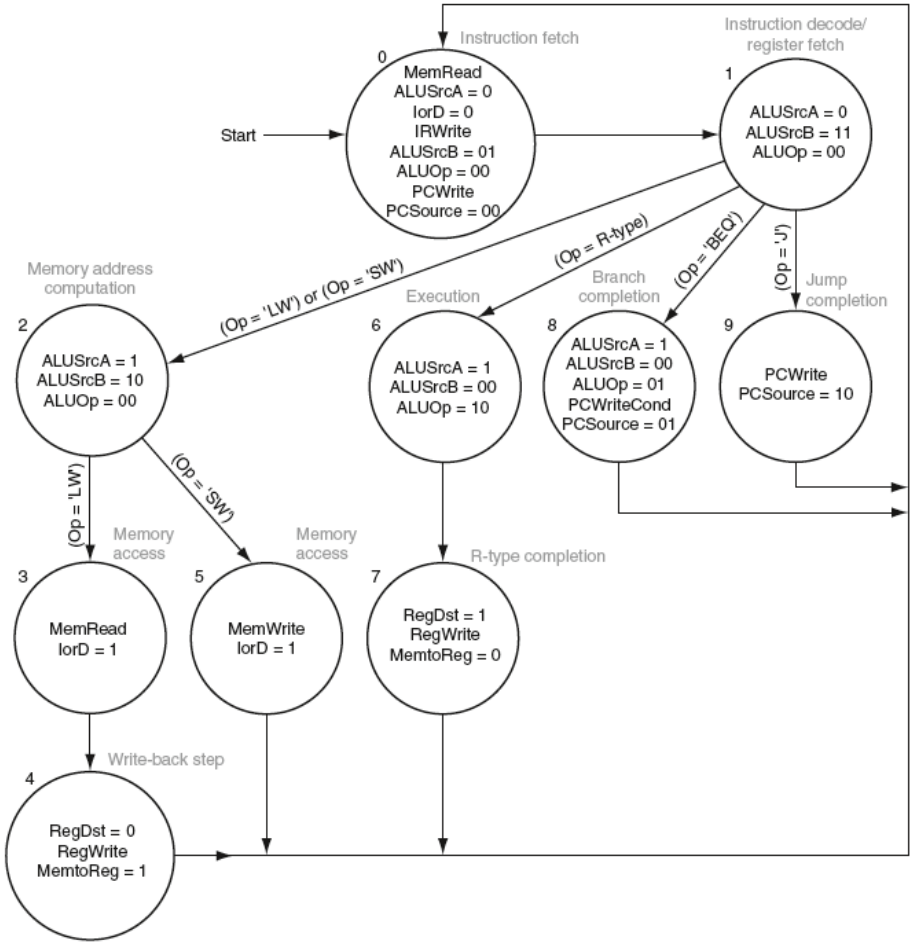
State alone is sufficient to determine output control signals

| Output | Current states | Op |
|---|---|---|
| PCWrite | state0 + state9 | |
| PCWriteCond | state8 | |
| IorD | state3 + state5 | |
| MemRead | state0 + state3 | |
| MemWrite | state5 | |
| IRWrite | state0 | |
| MemtoReg | state4 | |
| PCSource1 | state9 | |
| PCSource0 | state8 | |
| ALUOp1 | state6 | |
| ALUOp0 | state8 | |
| ALUSrcB1 | state1 +state2 | |
| ALUSrcB0 | state0 + state1 | |
| ALUSrcA | state2 + state6 + state8 | |
| RegWrite | state4 + state7 | |
| RegDst | state7 | |

# MIPS Control Unit

State alone determines output control signals

| Output | Current states |
|---|---|
| PCWrite | state0 + state9 |
| PCWriteCond | state8 |
| IorD | state3 + state5 |
| MemRead | state0 + state3 |
| MemWrite | state5 |
| IRWrite | state0 |
| MemtoReg | state4 |
| PCSource1 | state9 |
| PCSource0 | state8 |
| ALUOp1 | state6 |
| ALUOp0 | state8 |
| ALUSrcB1 | state1 +state2 |
| ALUSrcB0 | state0 + state1 |
| ALUSrcA | state2 + state6 + state8 |
| RegWrite | state4 + state7 |
| RegDst | state7 |



# MIPS Control Unit

Opcode as well as state are needed to determine next state

| Output | Current states | Op |
|---|---|---|
| NextState0 | state4 + state5 + state7 + state8 + state9 | |
| NextState1 | state0 | |
| NextState2 | state1 | (Op = 'lw') + (Op = 'sw') |
| NextState3 | state2 | (Op = 'lw') |
| NextState4 | state3 | |
| NextState5 | state2 | (Op = 'sw') |
| NextState6 | state1 | (Op = 'R-type') |
| NextState7 | state6 | |
| NextState8 | state1 | (Op = 'beq') |
| NextState9 | state1 | (Op = 'jmp') |

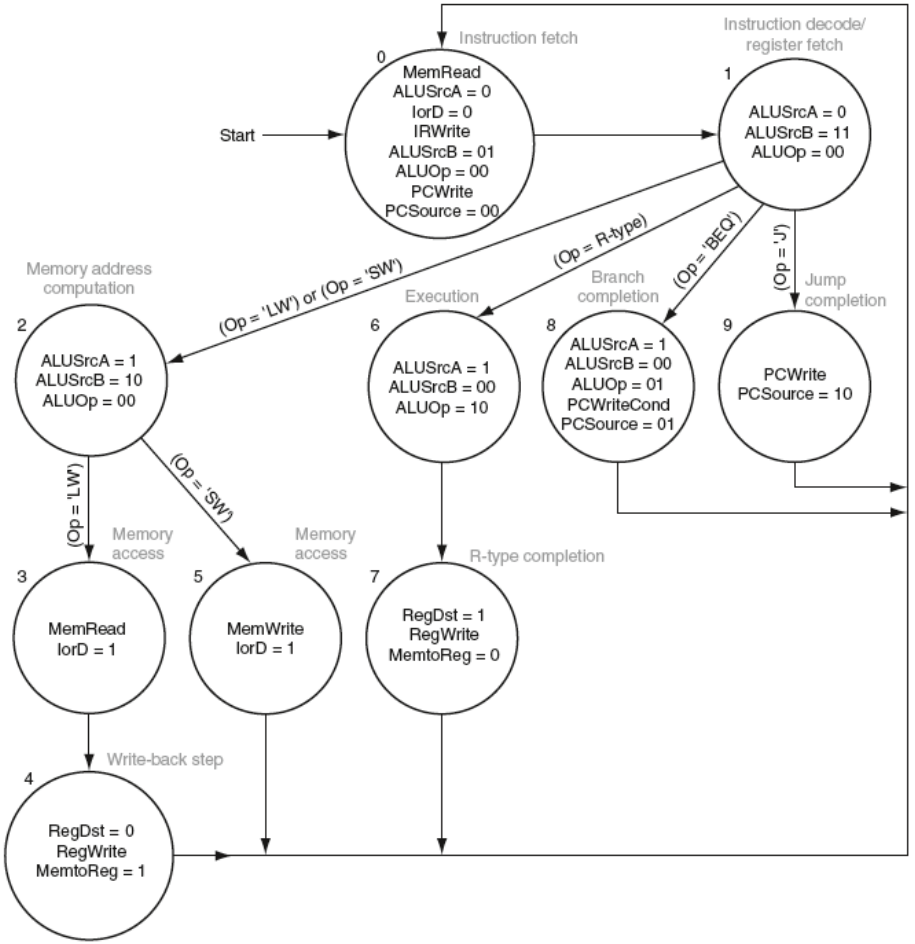
Four bits are needed for state number (since there are 10 states)

# MIPS Control Unit

| Output | Current states | Op |
|---|---|---|
| NextState0 | state4 + state5 + state7 + state8 + state9 | |
| NextState1 | state0 | |
| NextState2 | state1 | (Op = 'lw') + (Op = 'sw') |
| NextState3 | state2 | (Op = 'lw') |
| NextState4 | state3 | |
| NextState5 | state2 | (Op = 'sw') |
| NextState6 | state1 | (Op = 'R-type') |
| NextState7 | state6 | |
| NextState8 | state1 | (Op = 'beq') |
| NextState9 | state1 | (Op = 'jmp') |

Opcode as well as state are needed to determine next state

Four bits are needed for state number (since there are 10 states)



# Mapping Control to Gates

| Outputs | Input values (S[3–0]) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 |
| PCWrite | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| PCWriteCond | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| IorD | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| MemRead | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| MemWrite | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| IRWrite | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MemtoReg | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| PCSource1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| PCSource0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| ALUOp1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| ALUOp0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| ALUSrcB1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ALUSrcB0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ALUSrcA | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| RegWrite | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| RegDst | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Truth table gives the 16 control signals as function of 4-bit state

# MIPS Control Unit

The 16 control signals could be read from a lookup table in ROM

| Lower 4 bits of the address | Bits 19–4 of the word |
|---|---|
| 0000 | 1001010000001000 |
| 0001 | 0000000000011000 |
| 0010 | 0000000000010100 |
| 0011 | 0011000000000000 |
| 0100 | 0000001000000010 |
| 0101 | 0010100000000000 |
| 0110 | 0000000001000100 |
| 0111 | 0000000000000011 |
| 1000 | 0100000010100100 |
| 1001 | 1000000100000000 |

Use state number as index or address in ROM LUT (lookup table)

# MIPS Control Unit

Use state number (as row) and opcode (as column) to get next state number from second ROM LUT

| | Op [5–0] | | | | | |
|---|---|---|---|---|---|---|
| **Current state S[3–0]** | **000000 (R-format)** | **000010 (jmp)** | **000100 (beq)** | **100011 (lw)** | **101011 (sw)** | **Any other value** |
| 0000 | 0001 | 0001 | 0001 | 0001 | 0001 | 0001 |
| 0001 | 0110 | 1001 | 1000 | 0010 | 0010 | Illegal |
| 0010 | XXXX | XXXX | XXXX | 0011 | 0101 | Illegal |
| 0011 | 0100 | 0100 | 0100 | 0100 | 0100 | Illegal |
| 0100 | 0000 | 0000 | 0000 | 0000 | 0000 | Illegal |
| 0101 | 0000 | 0000 | 0000 | 0000 | 0000 | Illegal |
| 0110 | 0111 | 0111 | 0111 | 0111 | 0111 | Illegal |
| 0111 | 0000 | 0000 | 0000 | 0000 | 0000 | Illegal |
| 1000 | 0000 | 0000 | 0000 | 0000 | 0000 | Illegal |
| 1001 | 0000 | 0000 | 0000 | 0000 | 0000 | Illegal |

Undefined state/opcode combinations are “illegal”

# MIPS Control Unit

- State number alone is sufficient to determine the 16 control bits
- The 4-bit state number and 6-bit opcode form a 10-bit address
- 10-bit address determines control bits as well as next state
- Instruction execution corresponds to a sequence of states
- Control bits output in each state direct datapath actions
- Each state is one step in the execution and takes 1 clock cycle
- Other control unit implementations will be described next.

# Control Unit Implementation

Lookup Tables can consume large amounts of storage

Using a full 10-bit address (opcode + state) means 1024 entries

Each entry contains 16 control bits and 4 next state bits

Total size = 1024 * 20 bits (expensive in earlier days of computing)

One way to reduce the required storage is to use a PLA

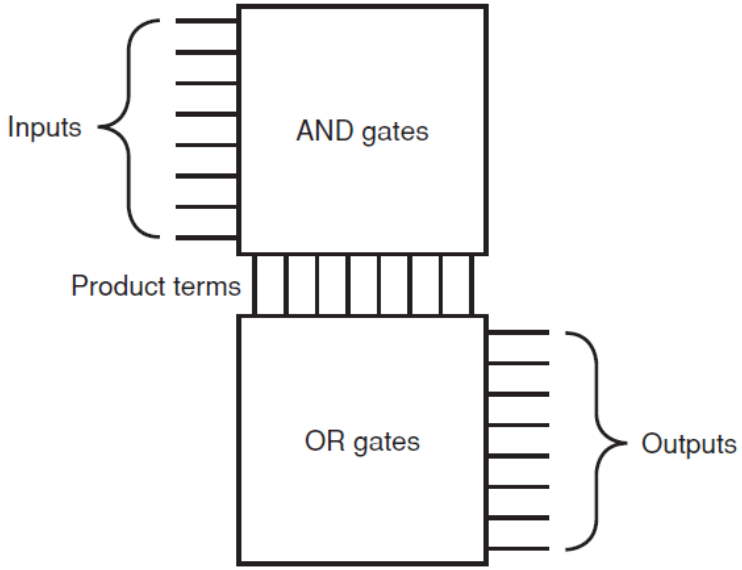
*"PLA"* means programmable logic array

# PLA Implementation

Each PLA output is a logical sum of one or more minterms

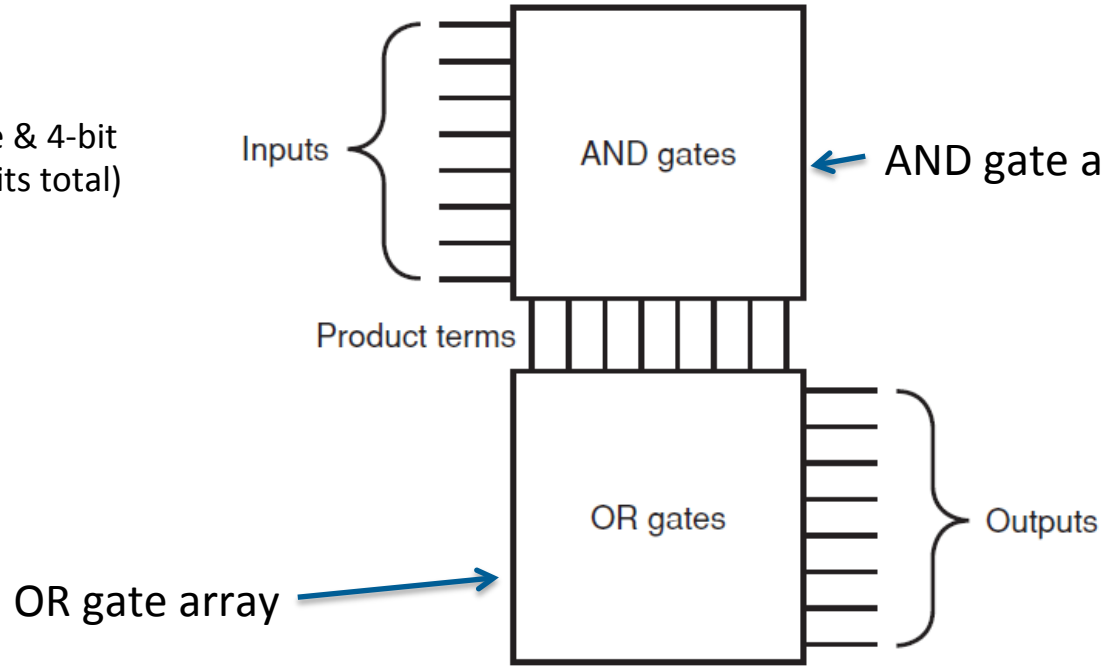
Minterm (or product term) is a logical AND of two or more inputs

A minterm corresponds to a single row in a truth table



# MIPS Core Subset PLA

6-bit Opcode & 4-bit state# (10 bits total)



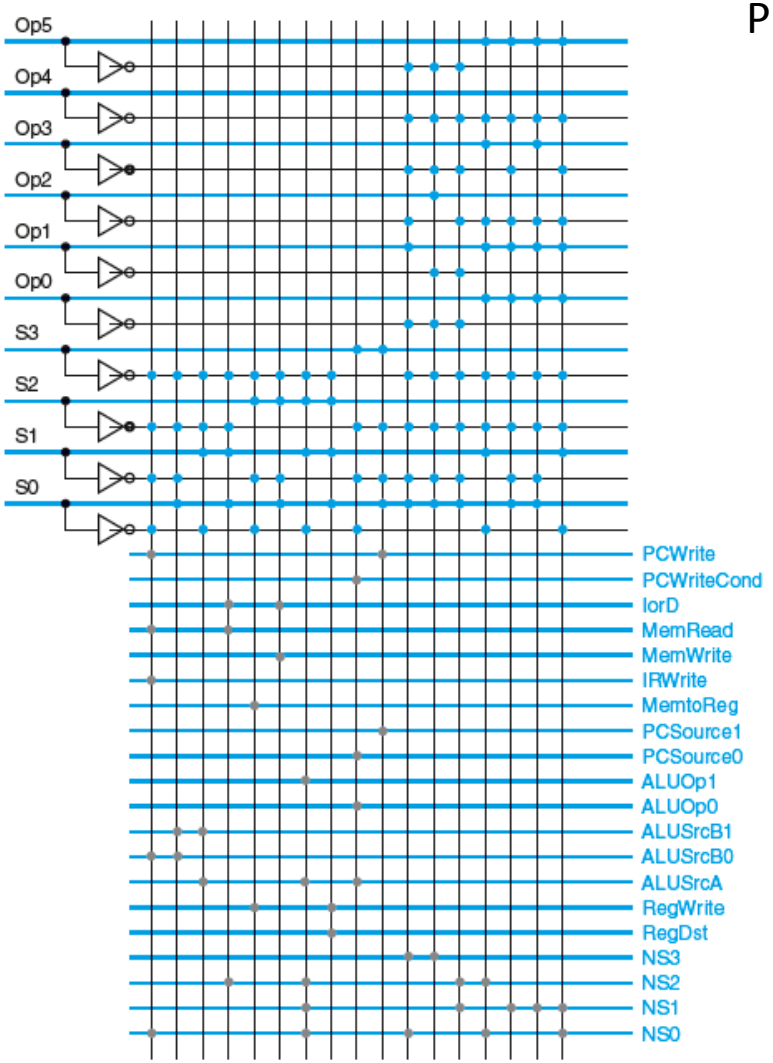

16 Control bits & 4-bit next state (20 bits)

- AND array generates products of inputs or inverted inputs
- OR array generates logical sums of product terms

# PLA Example

Programmable Logic Array

One vertical line for each of the 17 minterms



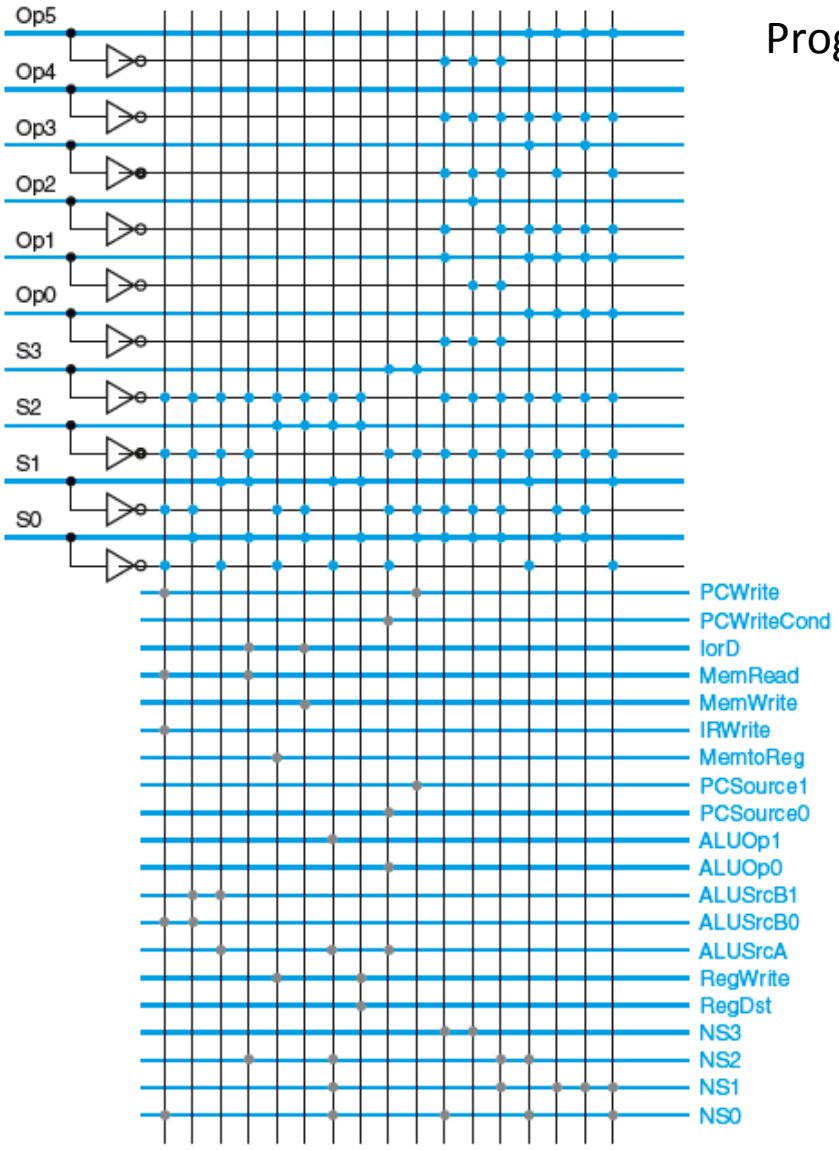

Size proportional to (#inputs + #outputs) * #product_terms

Horizontal output lines correspond to control lines and next state bits

# PLA Example

Programmable Logic Array

The 10 Inputs are opcode and current state




Size proportional to (10 + 20) * 17 = 510

The 20 outputs are control signals and next state

# PLA Implementation

Opcode bits and current state# and fed in for each cycle

The PLA outputs the resulting control signals and next state#

Control signals are output for each instruction step or subcycle

The sequence of state numbers identify the steps or transitions

This PLA supports our MIPS core instruction subset

# PLA Implementation

Each vertical line in the previous PLA is a minterm

The leftmost 10 depend only on the state

The remaining 7 depend on the state and opcode

The top half of the figure is the AND plane that computes minterms
Dots in top half indicate which inputs are fed into AND gate

The bottom half is the OR plane that sums the minterms
Dots in lower half indicate which minterms are fed into OR gate

# PLA Implementation

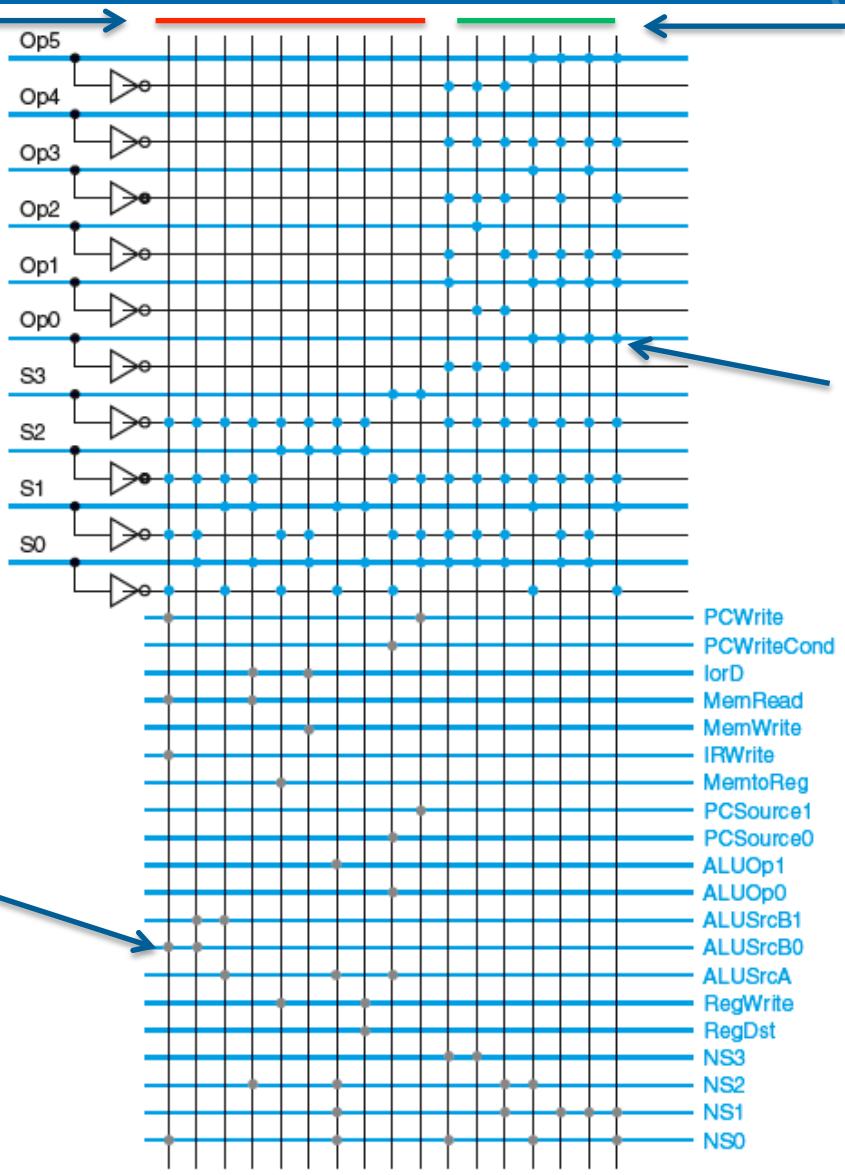
These 10 minterms depend only on state#

These 7 minterms depend on opcode as well as state#

Blue dots identify AND gate inputs

Black dots identify OR gate inputs

OR gates produce the sum of products



# PLA Implementation

Previous PLA can be replaced by two smaller PLAs (PLA1 & PLA2)

PLA1

- produces 10 minterms from 4 inputs (state)
- outputs the 16 control signal bits
- Size = (4 + 16) * 10 = 200

PLA2

- produces 7 minterms from 10 inputs (opcode & state)
- outputs 4-bit next state number
- Size = (10 + 4) * 7 = 98

Together the two consume less space than the single larger PLA

# Next State via Sequencer

- Most of the control logic is needed just for the next state number
- Our core MIPS system only has 10 states
- More realistic systems would have many more states
- Often the next state is just the previous state + 1
- The start state always comes after the final state in each instruction
- The start state begins the next instruction

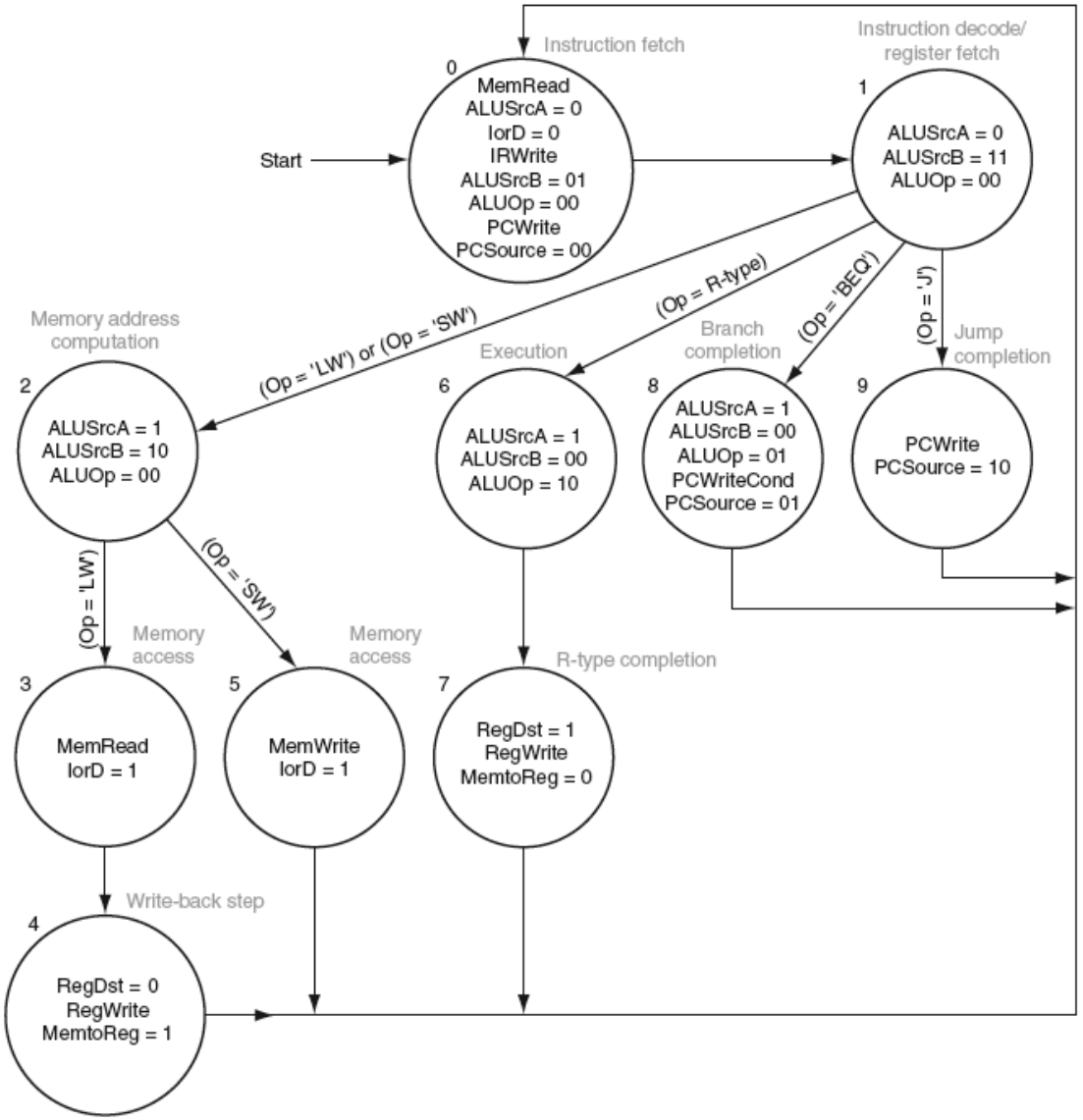
# State Diagram

State1 always comes after state0

State4 always comes after state3

State7 always comes after state 6

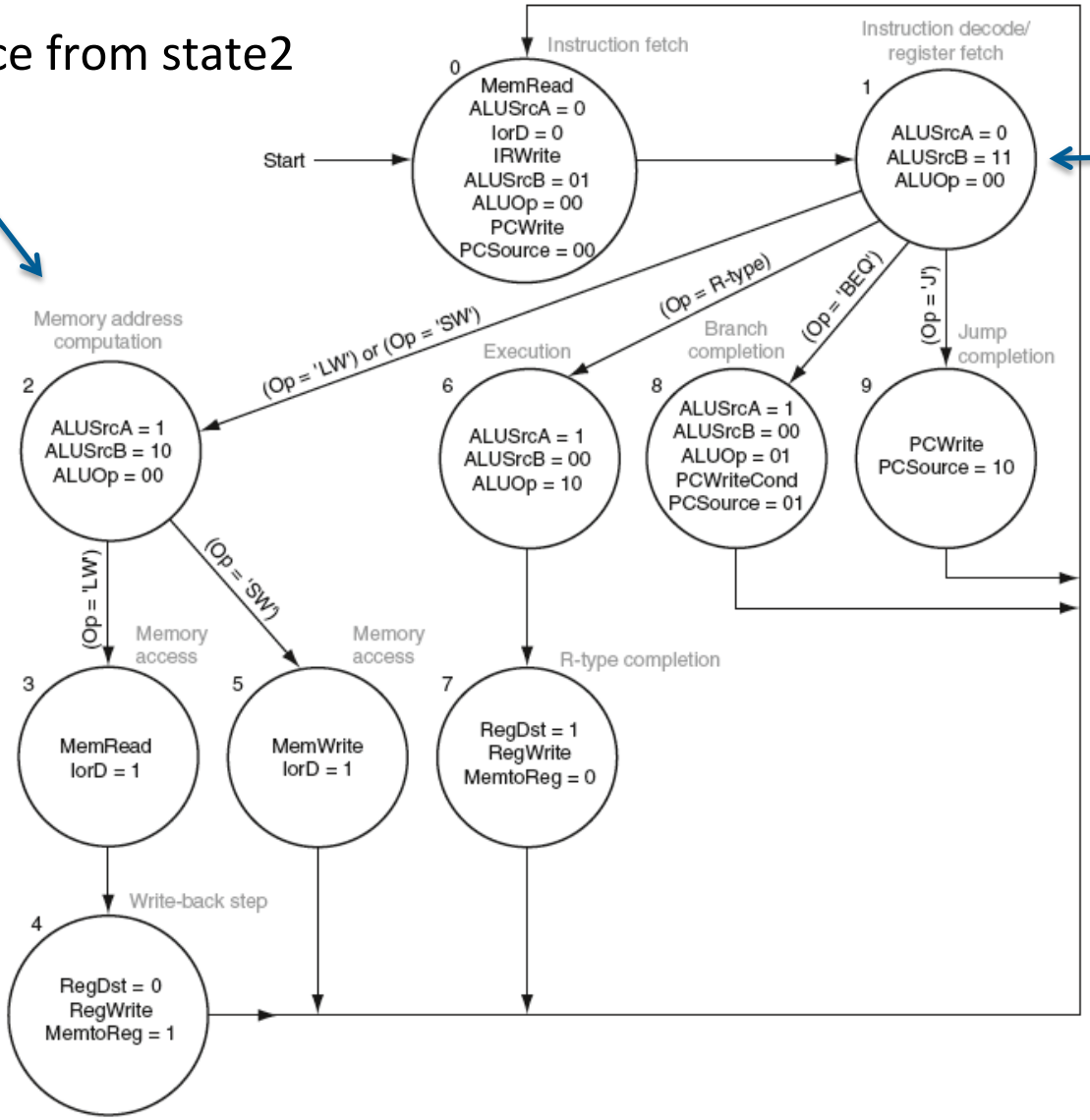
State0 always comes after states 4, 5, 7, 8 and 9



# State Diagram

2-way choice from state2

4-way choice from state1



# State Transitions

To go to the next sequential state, just increment the current state

Set the state to 0 to go back to the initial state

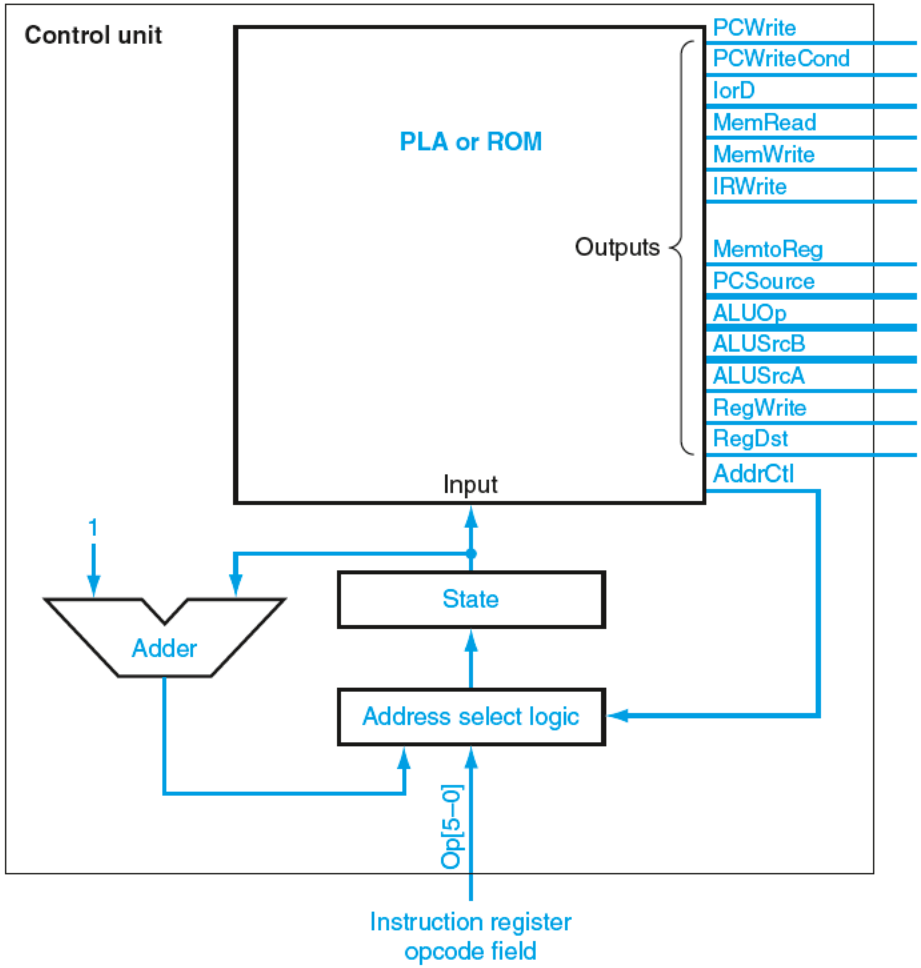
Small lookup tables can be used to choose a non-sequential state

The choice from state 1 and from state2 is based on the opcode

Two ROMs (addressed by opcode) can be used

A 2-bit control signal can be used with a MUX for 4 possible choices

# State Register Control




2-bit AddrCtl causes Address select logic to output the next state number

# State Register Control

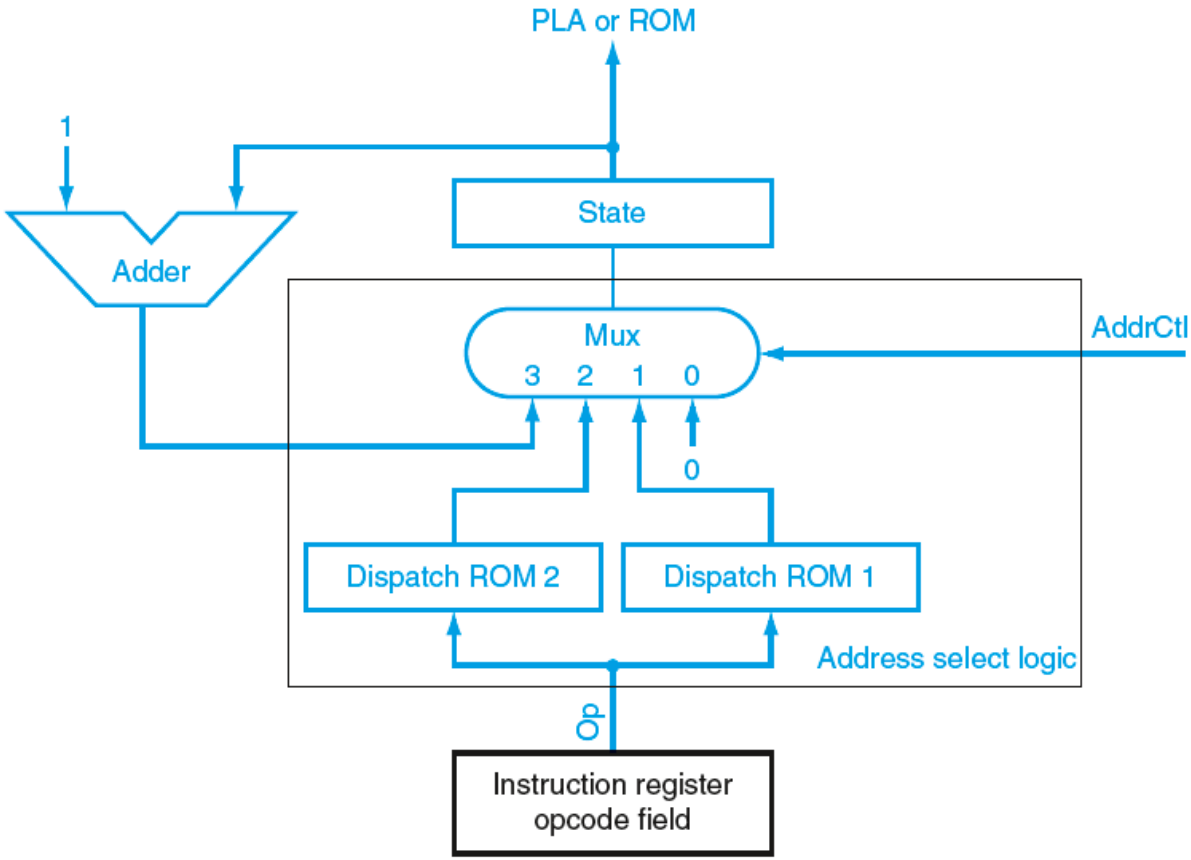
AddrCtl is defined as follows:

| AddrCtl value | Action |
|---|---|
| 0 | Set state to 0 |
| 1 | Dispatch with ROM 1 |
| 2 | Dispatch with ROM 2 |
| 3 | Use the incremented state |

In state1, next state is looked up in dispatch ROM1

In state2, next state is looked up in dispatch ROM2

# State Register Control




AddrCtl selects from: 0, ROM1, ROM2 or Output of Adder as next state

# Dispatch ROMs

| Dispatch ROM 1 | | |
|---|---|---|
| **Op** | **Opcode name** | **Value** |
| 000000 | R-format | 0110 |
| 000010 | jmp | 1001 |
| 000100 | beq | 1000 |
| 100011 | lw | 0010 |
| 101011 | sw | 0010 |

| Dispatch ROM 2 | | |
|---|---|---|
| **Op** | **Opcode name** | **Value** |
| 100011 | lw | 0011 |
| 101011 | sw | 0101 |

6-bit opcode as index implies up to $2^6 = 64$ entries

Only 5 entires are used from ROM1

Only 2 entries are used from ROM2

# AddrCtl

AddrCtl is determined just by the state number

| State number | Address-control action | Value of AddrCtl |
|---|---|---|
| 0 | Use incremented state | 3 |
| 1 | Use dispatch ROM 1 | 1 |
| 2 | Use dispatch ROM 2 | 2 |
| 3 | Use incremented state | 3 |
| 4 | Replace state number by 0 | 0 |
| 5 | Replace state number by 0 | 0 |
| 6 | Use incremented state | 3 |
| 7 | Replace state number by 0 | 0 |
| 8 | Replace state number by 0 | 0 |
| 9 | Replace state number by 0 | 0 |

State number determines AddrCtl as well as datapath control bits

# Control Word Contents

AddrCtl as well as datapath control bits can be stored in a control memory

| State number | Control word bits 17–2 | Control word bits 1–0 |
|---|---|---|
| 0 | 1001010000001000 | 11 |
| 1 | 0000000000011000 | 01 |
| 2 | 0000000000010100 | 10 |
| 3 | 0011000000000000 | 11 |
| 4 | 0000001000000010 | 00 |
| 5 | 0010100000000000 | 00 |
| 6 | 0000000001000100 | 11 |
| 7 | 0000000000000011 | 00 |
| 8 | 0100000010100100 | 00 |
| 9 | 1000000100000000 | 00 |

State number serves as address of all 18 bits (16 control + 2 AddrCtl)

Each 18-bit pattern can be viewed as a single “micro-instruction”

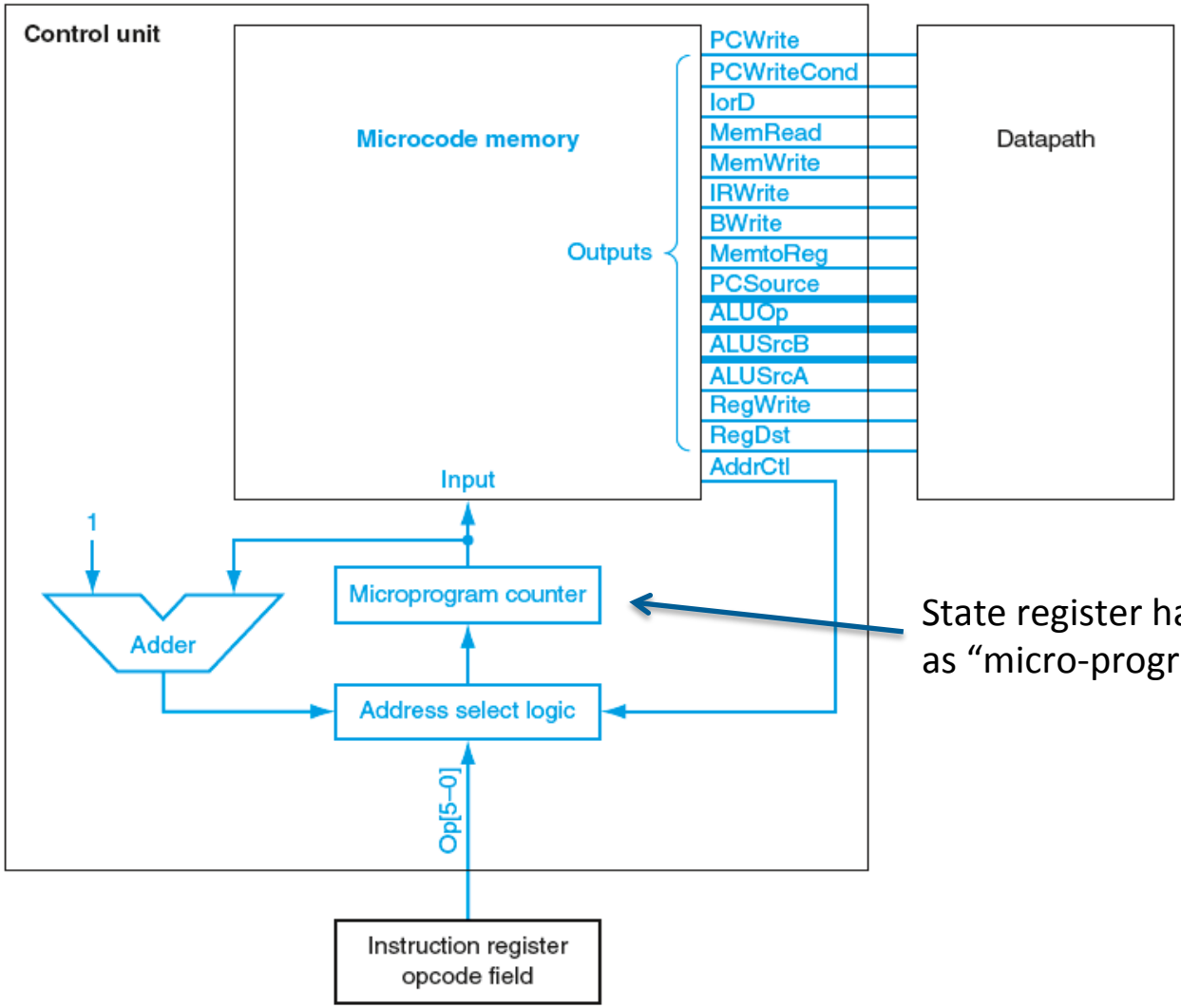
# Control ROM Contents

Micro-instructions are contained in the control store (ROM)

| State number | Control word bits 17–2 | Control word bits 1–0 |
|---|---|---|
| 0 | 1001010000001000 | 11 |
| 1 | 0000000000011000 | 01 |
| 2 | 0000000000010100 | 10 |
| 3 | 0011000000000000 | 11 |
| 4 | 0000001000000010 | 00 |
| 5 | 0010100000000000 | 00 |
| 6 | 0000000001000100 | 11 |
| 7 | 0000000000000011 | 00 |
| 8 | 0100000010100100 | 00 |
| 9 | 1000000100000000 | 00 |

Each micro-instruction takes one clock cycle

# Micro-programming




State register has been renamed as “micro-program counter”

# Micro-Instruction Fields

| Field name | Value | Signals active | Comment |
|---|---|---|---|
| ALU control | Add | ALUOp = 00 | Cause the ALU to add. |
| | Subt | ALUOp = 01 | Cause the ALU to subtract; this implements the compare for branches. |
| | Func code | ALUOp = 10 | Use the instruction's function code to determine ALU control. |
| SRC1 | PC | ALUSrcA = 0 | Use the PC as the first ALU input. |
| | A | ALUSrcA = 1 | Register A is the first ALU input. |
| SRC2 | B | ALUSrcB = 00 | Register B is the second ALU input. |
| | 4 | ALUSrcB = 01 | Use 4 as the second ALU input. |
| | Extend | ALUSrcB = 10 | Use output of the sign extension unit as the second ALU input. |
| | Extshft | ALUSrcB = 11 | Use the output of the shift-by-two unit as the second ALU input. |
| Register control | Read | | Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B. |
| | Write ALU | RegWrite, RegDst = 1, MemtoReg = 0 | Write a register using the rd field of the IR as the register number and the contents of ALUOut as the data. |
| | Write MDR | RegWrite, RegDst = 0, MemtoReg = 1 | Write a register using the rt field of the IR as the register number and the contents of the MDR as the data. |

# Micro-Instruction Fields

| Field name | Value | Signals active | Comment |
|---|---|---|---|
| Memory | Read PC | MemRead, IorD = 0, IRWrite | Read memory using the PC as address; write result into IR (and the MDR). |
| | Read ALU | MemRead, IorD = 1 | Read memory using ALUOut as address; write result into MDR. |
| | Write ALU | MemWrite, IorD = 1 | Write memory using the ALUOut as address, contents of B as the data. |
| PC write control | ALU | PCSource = 00, PCWrite | Write the output of the ALU into the PC. |
| | ALUOut-cond | PCSource = 01, PCWriteCond | If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut. |
| | Jump address | PCSource = 10, PCWrite | Write the PC with the jump address from the instruction. |
| Sequencing | Seq | AddrCtl = 11 | Choose the next microinstruction sequentially. |
| | Fetch | AddrCtl = 00 | Go to the first microinstruction to begin a new instruction. |
| | Dispatch 1 | AddrCtl = 01 | Dispatch using the ROM 1. |
| | Dispatch 2 | AddrCtl = 10 | Dispatch using the ROM 2. |

# Micro-programming

- Each micro-instruction performs one step in a machine instruction
- Each micro-instruction takes one clock cycle
- A machine instruction corresponds to a “micro-program”
- A micro-program is a sequence of micro-instructions
- Control memory containing micro-programs is on CPU chip
- Extra time is needed to retrieve and execute micro-programs

# Micro-programming

- Time required to retrieve micro-progams slows the system
- Micro-programming provides flexibility
- Complex instructions are easier to implement as micro-code
- CISC systems employ micro-programming
- Changing micro-code changes behavior of the system
- RISC systems use faster hardwired logic instead

# Micro-code Encoding

- Each micro-instruction in our conrol ROM is 18 bits wide
- None of the fields within the micro-instructions are encoded
- This is fast since no decoding is required
- Systems of this type are said to be “minimally encoded”
- This is also called “horizonal micro-code”
- Micro-code is sometimes called firmware
- Firmware is harder to change than software
- Hardware is more difficult to change that firmware

# Micro-code Encoding

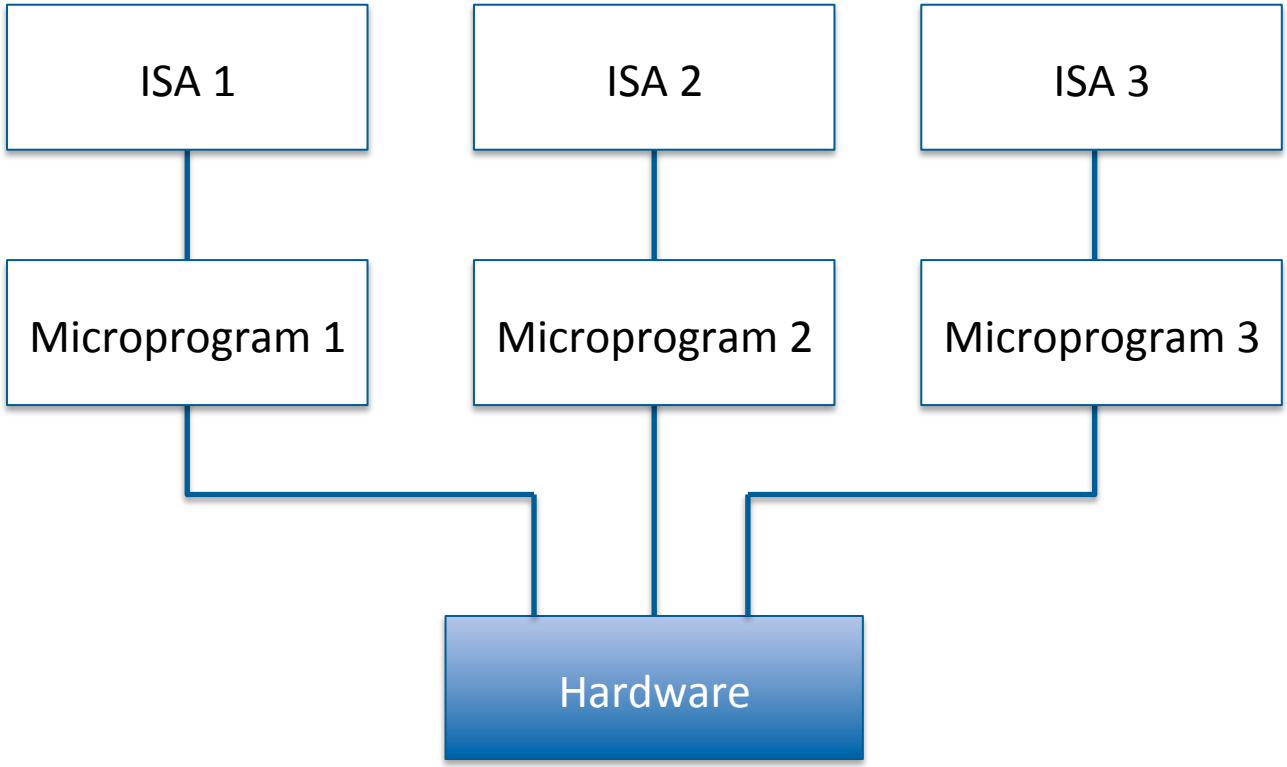
- A writeable control store employs RAM for micro-code
- More realistic systems are more complex than our core MIPS
- Such systems could require very wide micro-instructions
- Encoding fields within these micro-instructions would save bits
- However these take longer to process due to need for decoding
- These systems employ maximally encoding (“vertical micro-code)

# Emulation

- Micro-programming enables one system to behave like another
- This is called “emulation”
- It allows the same hardware to be used with different ISAs
- ISA is the instruction set architecture
- Different machine code can execute on the same hardware
- The micro-code is said to interpret the machine instructions

# Emulation

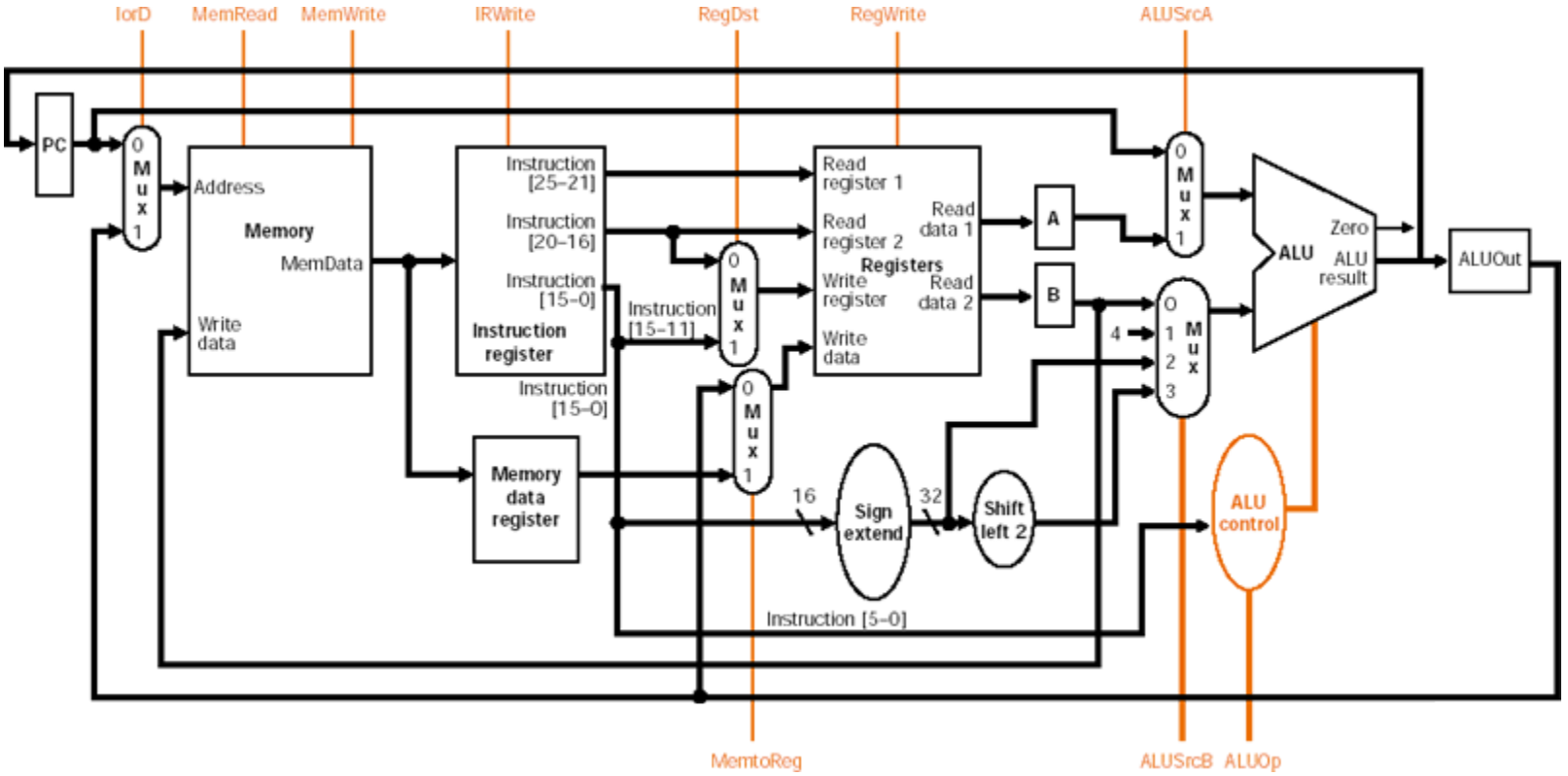
Emulation allows the hardware to run different machine programs



# Other Control Implementations

- Instructions can be executed in multiple steps
- Longest step determines clock period
  - Critical path: memory access time
  - Each step takes one clock cycle
- Different control signals are generated in different clock cycles
- Change in state occurs with each cycle
- Sequential logic contains state
- Finite-state machine can model behavior

# Multi-cycle Datapath



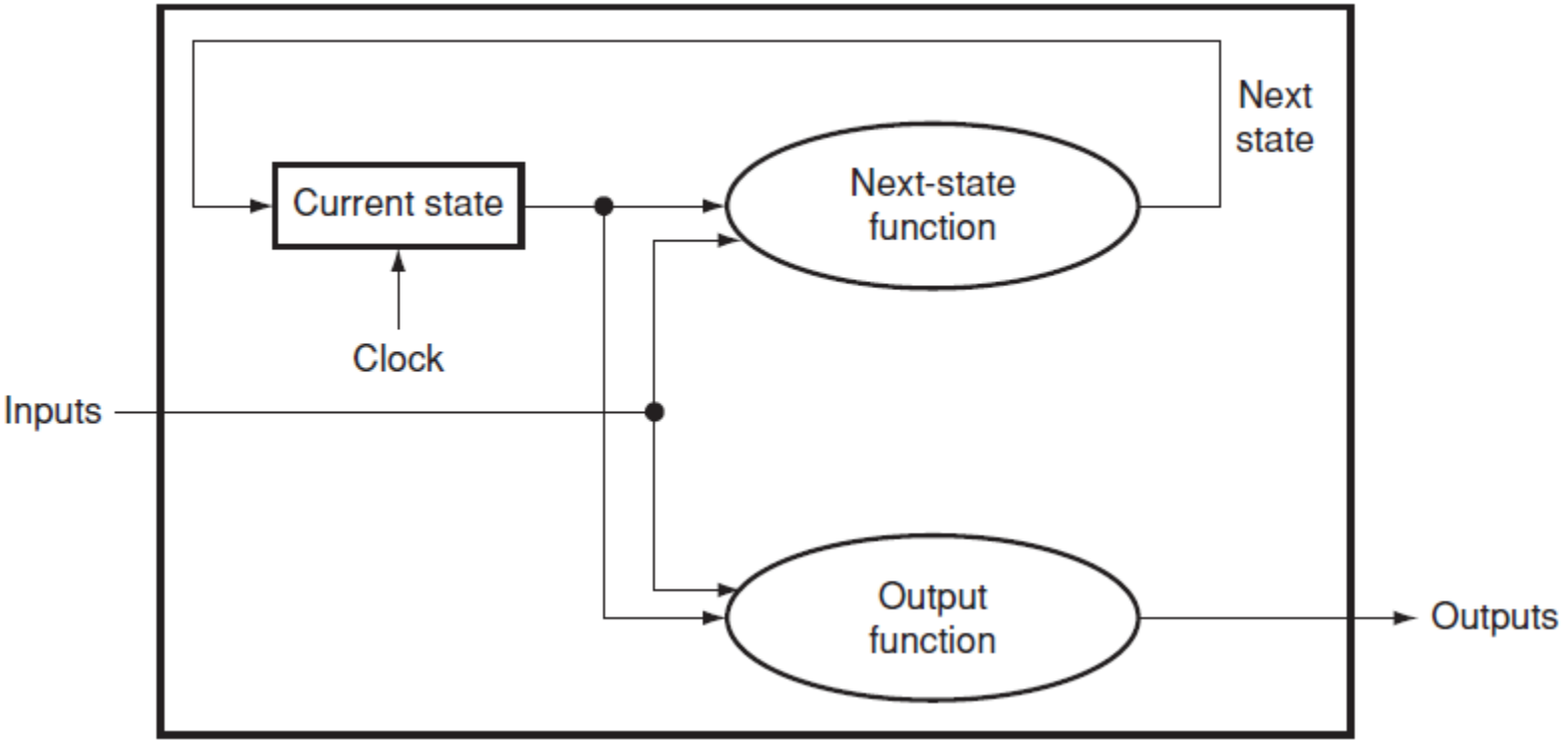

- Component re-use eliminates duplication and reduces cost

# Finite State Machine Control

Also called “state machine” or FSM and contains:

- Set of states (one of which is the initial start state)
- Set of inputs
- Set of outputs
- Next-state function maps current state and inputs to a new state (state transition)
- Output function maps current state to set of outputs
  - Outputs may also depend on inputs
  - “Moore machine” outputs depend only on state
  - “Mealy machine” outputs depend on state and inputs

# Finite State Machine Control

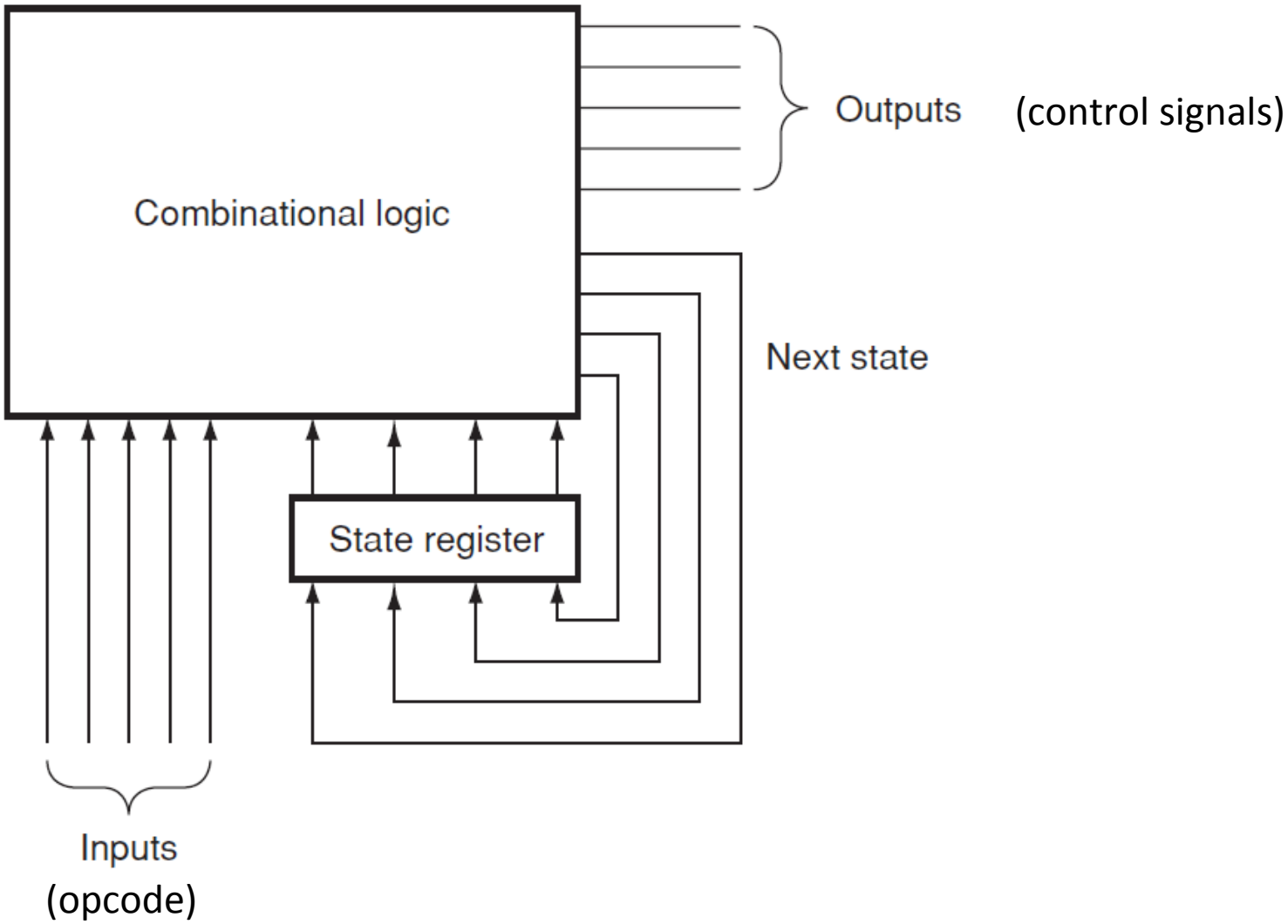



Sequential logic, internal storage contains the state information

# FSM for MIPS Control Unit

- Set of inputs are the opcode bits
- Set of outputs are the control signals
- Next-state function is implemented with combinational logic
- New state is computed synchronously with clock cycle

# FSM for MIPS Control Unit



# FSM for Multicycle Control

MIPS core instruction subset requires 10 states