High level representation of the control needed for the multicycle datapath. Each of the top 2 boxes is a state; the lower boxes are small FSMs.
Finite State Machines

**Mealy machine**

next state function: \( F_{\text{next state}}(I, S(i)) \to S(i+1) \)

output function: \( F_{\text{output}}(I, S(i)) \to O(i) \)

current state

next state function

output function

Inputs

Outputs

Current state

S(i) - Current state

Copyright 1995 by Coherence LTD., all rights reserved (Revised: Oct 97 by Rafi Lohev, Oct 99 by Yair Wiseman)
Example -- Traffic Lights Controller

- The controller operates the traffic lights at one intersection.
- Detectors in both directions provide input signals to controllers: *car approaching NS / car approaching EW*.
- Outputs control the traffic light: *set to green / set to red*.
- When cars approach in both directions, *toggle* the light.
- The controller **needs two states, green/red**.

A state/output table is used to construct the controller:

<table>
<thead>
<tr>
<th>state(i)</th>
<th>carNS</th>
<th>carEW</th>
<th>state(i+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS go</td>
<td>no</td>
<td>no</td>
<td>NS go</td>
</tr>
<tr>
<td>NS go</td>
<td>no</td>
<td>yes</td>
<td>EW go</td>
</tr>
<tr>
<td>NS go</td>
<td>yes</td>
<td>no</td>
<td>NS go</td>
</tr>
<tr>
<td>NS go</td>
<td>yes</td>
<td>yes</td>
<td>EW go</td>
</tr>
<tr>
<td>EW go</td>
<td>no</td>
<td>no</td>
<td>EW go</td>
</tr>
<tr>
<td>EW go</td>
<td>no</td>
<td>yes</td>
<td>EW go</td>
</tr>
<tr>
<td>EW go</td>
<td>yes</td>
<td>no</td>
<td>NS go</td>
</tr>
<tr>
<td>EW go</td>
<td>yes</td>
<td>yes</td>
<td>NS go</td>
</tr>
</tbody>
</table>
State Diagram

NSgo

EWgo

carEW or both directions

carNS or both directions

carNS or nothing

carEW or nothing

Calculate next state transitions using the table and Boolean identities

This example is a Moore machine → the outputs are a function only of the state. In this case the output function is trivial: In state NSgo we want a signal for a NS green light. In state EWgo we want a signal for an EW green light.
Implementation

Finite state machine with 4 state variables (up to 16 states)
General Digital System

- Control
  - Combinational Logic
  - State Register
- Architecture
- CLOCK
- Opcode
Control for the Multicycle Datapath

- Control for the multicycle datapath requires a finite state machine or FSM.
  Each state will generate the correct control signals for one clock cycle.
- Instructions can extend for at most 5 clock cycles: Fetch, Decode, Exec, Mem, WriteBack.
  The first two steps are the same for all instructions.
  R-type requires 2 additional steps.
  Load requires 3 additional steps.
  Store requires the same first step as Load plus 1 additional step to write memory.
  Branch and Jump each require 1 additional step.
  total different states = 10.

- We use a simple form of FSM, called a Moore machine.
  the outputs of this type of machine are a function of current state only, inputs only affect the choice of next state.
  other type of machine -- Mealy machine -- has outputs which are a function of both current state and inputs.
  both machine types are equivalent (one can be converted to another by adding states).
First 2 FSM States

Instruction Fetch

Start

0

ALUSelA=0
IorD=0
IRWrite
ALUSelB=01
ALUOp=00
PCWrite
PCSource=00

Decode/Register Fetch

1

ALUSelA=0
ALUSelB=11
ALUOp=00
TargetWrite

Ld / Str FSM

Opcode= (Ld or Str)

R-type FSM

Opcode = R-type

Branch FSM

Opcode = BEQ

Jump FSM

Opcode = JMP
Memory Reference FSM

From state 1

Opcode = (Ld or Str)

2. Compute address
   ALUSelA = 1
   ALUSelB = 10
   ALUOp = 00

3. Memory read
   Opcode = Ld
   ALUSelA = 1
   lorD = 1
   ALUSelB = 10
   ALUOp = 00

4. Ld Write back
   ALUSelA = 1
   lorD = 1
   RegWrite
   MemtoReg = 1
   RegDest = 0
   ALUSelB = 10
   ALUOp = 00

5. Memory write
   Opcode = Str
   MemWrite
   ALUSelA = 1
   lorD = 1
   ALUSelB = 10
   ALUOp = 00

To state 0
From state 1

 Opcode = R-type

 Execute

 ALUSelA = 1
 ALUSelB = 00
 ALUOp = 10

 R-type write back

 ALUSelA = 1
 RegDst = 1
 RegWrite
 MemtoReg = 0
 ALUSelB = 00
 ALUOp = 10

 To state 0
**FSMs for Branch and Jump**

From state 1:

- Opcode = BEQ

Branch completion:
- ALUSelA = 1
- ALUSelB = 00
- ALUOp = 01
- PCWriteCond
- PCSource = 01

To state 0

Jump completion:
- Opcode = JMP
  - PCWrite
  - PCSource = 10

To state 0
Start

0

ALUSelA=0
IorD=0
IRWrite
ALUSelB=01
ALUOp=00
PCWrite
PCSource=00

Instruction Fetch

Branch completion

ALUSelA=1
ALUSelB=00
ALUOp=01
PCWriteCond
PCSource=01

Decode/ Register Fetch

1

ALUSelA=0
ALUSelB=11
ALUOp=00
TargetWrite

ALUSelA=1
ALUSelB=10
ALUOp=00

2

Compute address

Opcode = (Ld or Str)

Opcode = Ld

ALUSelA=1
IorD=1
ALUSelB=10
ALUOp=00

3

Memory read

Opcode = Str

Memory write

ALUSelA=1
IorD=1
RegWrite
MemtoReg=1
RegDst=0
ALUSelB=10
ALUOp=00

4

Ld Write back

ALUSelA=1
IorD=1
RegWrite
MemtoReg=1
RegDst=0
ALUSelB=10
ALUOp=00

5

MemWrite

ALUSelA=1
IorD=1
ALUSelB=10
ALUOp=00

6

Execute

ALUSelA=1
ALUSelB=10
ALUOp=00

R-type write back

ALUSelA=1
RegDst=1
RegWrite
MemtoReg=0
ALUSelB=00
ALUOp=10

7

ALUSelA=1
ALUSelB=00
ALUOp=10

8

Branch completion

Jump completion

PCWrite
PCSource=01

ALUSelA=1
ALUSelB=00
ALUOp=01
PCWriteCond
PCSource=01

ALUSelA=1
ALUSelB=00
ALUOp=01
PCWriteCond
PCSource=01

ALUSelA=1
ALUSelB=00
ALUOp=01
PCWriteCond
PCSource=01

ALUSelA=1
ALUSelB=00
ALUOp=01
PCWriteCond
PCSource=01

ALUSelA=1
ALUSelB=00
ALUOp=01
PCWriteCond
PCSource=01

ALUSelA=1
ALUSelB=00
ALUOp=01
PCWriteCond
PCSource=01

ALUSelA=1
ALUSelB=00
ALUOp=01
PCWriteCond
PCSource=01
FSM implementation for multicycle controller

Combinational Logic

Inputs:
- Op-code (6 bits)

State Register:
- 4 bits: 10 states

Outputs:
- Next state
- 4 next state outputs

Logic Gates:
- PCWrite
- PCWriteCond
- IorD
- MemWrite
- IRWrite
- MemtoReg
- PCSource
- TargetWrite
- ALUop
- ALUseIA
- ALUseIB
- RegWrite
- RegDst
**ROM implementation of control**

ROM = Read Only Memory

Each state + input combination "reads" the correct command line values from a word in ROM memory.

---

**Diagram Description:**

- **State Register:** 4 bits, 10 states
  - s0, s1, s2, s3

- **Opcode:** 6 bits
  - Enables direct access to a word in ROM

- **ROM:** 20 bits word size
  - 16 command lines
  - 10 address lines
  - 4 next state outputs

- **Outputs:**
  - PCWrite
  - PCWriteCond
  - IorD
  - MemWrite
  - IRWrite
  - MemtoReg
  - PCSource
  - TargetWrite
  - ALUop
  - ALUseIA
  - ALUseIB
  - RegWrite
  - RegDst
  - ns3, ns2, ns1, ns0
What is inside ROM?

- 000000 is the opcode of R-type.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000 0000</td>
<td>10001X00X000010X 0001</td>
</tr>
<tr>
<td>000000 0001</td>
<td>00X00XXX1000110X 0110</td>
</tr>
<tr>
<td>000000 0010</td>
<td>XXXXXXXXXXXXXXXXXX XXXX</td>
</tr>
<tr>
<td>000000 0011</td>
<td>XXXXXXXXXXXXXXXXXX XXXX</td>
</tr>
<tr>
<td>000000 0100</td>
<td>XXXXXXXXXXXXXXXXXX XXXX</td>
</tr>
<tr>
<td>000000 0101</td>
<td>XXXXXXXXXXXXXXXXXX XXXX</td>
</tr>
<tr>
<td>000000 0110</td>
<td>00X00XXX101000X 0111</td>
</tr>
<tr>
<td>000000 0111</td>
<td>00X000XXX1010011 0000</td>
</tr>
<tr>
<td>000000 1000</td>
<td>XXXXXXXXXXXXXXXXXX XXXX</td>
</tr>
<tr>
<td>000000 1001</td>
<td>XXXXXXXXXXXXXXXXXX XXXX</td>
</tr>
</tbody>
</table>

- Wasteful because many entries of table are not used.
- In Real MIPS more codes are used but still this method is wasteful.
To save ROM, words of commands in the Microstore are arranged in sequence, so that a simple counter can move from state to state, like a program counter which sequences through instructions. The contents of the microstore can be programmed similar to ordinary program code -- also called microinstructions. The sequencer control may decide if the next state should be determined by the op-code (for example after our "decode" state) or reset to "0" which starts fetch of the next instruction. Otherwise, the adder automatically increments the microinstruction state to the next location.
## Dispatch ROM tables

### Dispatch ROM1

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0110</td>
</tr>
<tr>
<td>000010</td>
<td>1001</td>
</tr>
<tr>
<td>000100</td>
<td>1000</td>
</tr>
<tr>
<td>100011</td>
<td>0010</td>
</tr>
<tr>
<td>101011</td>
<td>0010</td>
</tr>
</tbody>
</table>

### Dispatch ROM2

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td>0011</td>
</tr>
<tr>
<td>101011</td>
<td>0101</td>
</tr>
</tbody>
</table>

000000  - R-type  
000010  - JMP  
000100  - BEQ  
100011  - LW  
101011  - SW
### What is inside MicroStore?

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1X001X00X000010X 11</td>
</tr>
<tr>
<td>0001</td>
<td>00X00XXX1000110X 01</td>
</tr>
<tr>
<td>0010</td>
<td>00X00XXX001100X 10</td>
</tr>
<tr>
<td>0011</td>
<td>00100XXX001100X 11</td>
</tr>
<tr>
<td>0100</td>
<td>001001XXX0011010 00</td>
</tr>
<tr>
<td>0101</td>
<td>00X10XXX001100X 00</td>
</tr>
<tr>
<td>0110</td>
<td>00X00XXX101000X 11</td>
</tr>
<tr>
<td>0111</td>
<td>00X00XXX1010011 00</td>
</tr>
<tr>
<td>1000</td>
<td>01X00X01X011000X 00</td>
</tr>
<tr>
<td>1001</td>
<td>1XX00X10XXXXXXX0X 00</td>
</tr>
</tbody>
</table>

- We Can use hybrid approach - Combinational Logic for Dispatch tables and ROM for MicroStore.