Chapter 5:
The Processor: Datapath & Control

• We're ready to look at an implementation of the MIPS
• Simplified to contain only:
  – memory-reference instructions: \texttt{lw, sw}
  – arithmetic-logical instructions: \texttt{add, sub, and, or, slt}
  – control flow instructions: \texttt{beq, j}
• Generic Implementation:
  – use the program counter (PC) to supply instruction address
  – get the instruction from memory
  – read registers
  – use the instruction to decide exactly what to do
• All instructions use the ALU after reading the registers
More Implementation Details

• Abstract / Simplified View:

Two types of functional units:
  – elements that operate on data values (combinational)
  – elements that contain state (sequential)
State Elements

• Unclocked vs. Clocked
• Clocks used in synchronous logic
  – when should an element that contains state be updated?

[Diagram showing clock signal with labeled edges: falling edge and rising edge, cycle time indicated]
An unclocked state element

- The set-reset latch
  - output depends on present inputs and also on past inputs
Latches and Flip-flops

- Output is equal to the stored value inside the element (don't need to ask for permission to look at the value)
- Change of state (value) is based on the clock
- Latches: whenever the inputs change, and the clock is asserted
- Flip-flop: state changes only on a clock edge (edge-triggered methodology)

"logically true", — could mean electrically low

A clocking methodology defines when signals can be read and written — wouldn't want to read a signal at the same time it was being written
D-latch

• Two inputs:
  – the data value to be stored (D)
  – the clock signal (C) indicating when to read & store D

• Two outputs:
  – the value of the internal state (Q) and its complement
D flip-flop

- Output changes only on the clock edge
Our Implementation

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements
Register File

- Built using D flip-flops
Register File

- Note: we still use the real clock to determine when to write
Simple Implementation

- Include the functional units we need for each instruction

Why do we need this stuff?
Building the Datapath

- Use multiplexors to stitch them together
Control

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
- Example:

  add $8, $17, $18  Instruction Format:

  \[
  \begin{array}{cccccc}
  & 000000 & 10001 & 10010 & 01000 & 00000 & 100000 \\
  \hline
  \text{op} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{func} \\
  \end{array}
  \]

- ALU's operation based on instruction type and function code
Control

- e.g., what should the ALU do with this instruction
- Example: `lw $1, 100($2)`

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>2</td>
<td>1</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit offset</th>
</tr>
</thead>
</table>

- ALU control input

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>subtract</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>set-on-less-than</td>
<td></td>
</tr>
</tbody>
</table>

- Why is the code for subtract 110 and not 011?
Control

• Must describe hardware to compute 3-bit ALU control input
  – given instruction type
    00 = lw, sw
    01 = beq, 11 = arithmetic
  – function code for arithmetic

• Describe it using a truth table (can turn into gates):

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>ALUOp0</th>
<th>F5</th>
<th>F4</th>
<th>F3</th>
<th>F2</th>
<th>F1</th>
<th>F0</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
</tbody>
</table>

ALUOp computed from instruction type
Control

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Control

- Simple combinational logic (truth tables)
• All of the logic is combinational
• We wait for everything to settle down, and the right thing to be done
  – ALU might not produce “right answer” right away
  – we use write signals along with clock to determine when to write
• Cycle time determined by length of the longest path

We are ignoring some details like setup and hold times
Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (2ns), ALU and adders (2ns), register file access (1ns)
Where we are headed

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
- One Solution:
  - use a “smaller” cycle time
  - have different instructions take different numbers of cycles
  - a “multicycle” datapath:
Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined solely by instruction
  - e.g., what should the ALU do for a “subtract” instruction?
- We’ll use a finite state machine for control
Review: finite state machines

- Finite state machines:
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)

- We’ll use a Moore machine (output based only on current state)
Review: finite state machines

• Example:

B.21 A friend would like you to build an “electronic eye” for use as a fake security device. The device consists of three lights lined up in a row, controlled by the outputs Left, Middle, and Right, which if asserted, indicate that a light should be on. Only one light is on at a time, and the light “moves” from left to right and then from right to left, thus scaring away thieves who believe that the device is monitoring their activity. Draw the graphical representation for the finite state machine used to specify the electronic eye. Note that the rate of the eye’s movement will be controlled by the clock speed (which should not be too great) and that there are essentially no inputs.
Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional “internal” registers
Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!
Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

\[
\begin{align*}
IR &= \text{Memory}[PC]; \\
PC &= PC + 4;
\end{align*}
\]

*Can we figure out the values of the control signals?*

*What is the advantage of updating the PC now?*
Step 2: Instruction Decode and Register Fetch

• Read registers rs and rt in case we need them
• Compute the branch address in case the instruction is a branch
• RTL:

\[
A = \text{Reg}[\text{IR}[25-21]]; \\
B = \text{Reg}[\text{IR}[20-16]]; \\
\text{ALUOut} = \text{PC} + (\text{sign-extend}(\text{IR}[15-0]) \ll 2);
\]

• We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)
Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
  
  - Memory Reference:
    \[ \text{ALUOut} = A + \text{sign-extend}(\text{IR}[15-0]) \; \]
  
  - R-type:
    \[ \text{ALUOut} = A \text{ op B} \; \]
  
  - Branch:
    \[ \text{if } (A==B) \; \text{PC} = \text{ALUOut}; \]
Step 4 (R-type or memory-access)

- Loads and stores access memory
  
  \[
  \text{MDR} = \text{Memory}[\text{ALUOut}]; \\
  \text{or} \\
  \text{Memory}[\text{ALUOut}] = B;
  \]

- R-type instructions finish
  
  \[
  \text{Reg}[\text{IR}[15-11]] = \text{ALUOut};
  \]

*The write actually takes place at the end of the cycle on the edge*
Write-back step

- \( \text{Reg[IR[20-16]] = MDR}; \)

What about all the other instructions?
### Summary:

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td></td>
<td>IR = Memory[PC]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC = PC + 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td></td>
<td>A = Reg [IR[25-21]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B = Reg [IR[20-16]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUOut = PC + (sign-extend (IR[15-0]) &lt;&lt; 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/jump completion</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend (IR[15-0])</td>
<td>if (A ==B) then PC = ALUOut</td>
<td>PC = PC [31-28] II (IR[25-0]&lt;&lt;2)</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg [IR[15-11]] = ALUOut</td>
<td>Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td></td>
<td>Load: Reg[IR[20-16]] = MDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simple Questions

• How many cycles will it take to execute this code?

```
lw $t2, 0($t3)
lw $t3, 4($t3)
beq $t2, $t3, Label #assume not
add $t5, $t2, $t3
sw $t5, 8($t3)
Label: ...```

• What is going on during the 8th cycle of execution?
• In what cycle does the actual addition of $t2 and $t3 takes place?
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
How many state bits will we need?
Finite State Machine for Control

- Implementation:

![Finite State Machine Diagram]
PLA Implementation

• If I picked a horizontal or vertical line could you explain it?
ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address $2^m$ entries in the ROM.
  - our outputs are the bits of data that the address points to.

$m$ is the "heigth", and $n$ is the "width"
ROM Implementation

- How many inputs are there?
  6 bits for opcode, 4 bits for state = 10 address lines
  (i.e., $2^{10} = 1024$ different addresses)
- How many outputs are there?
  16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is $2^{10} \times 20 = 20K$ bits (and a rather unusual size)
- Rather wasteful, since for lots of the entries, the outputs are the same
  — i.e., opcode is often ignored
ROM vs PLA

- Break up the table into two parts
  - 4 state bits tell you the 16 outputs, \(2^4 \times 16\) bits of ROM
  - 10 bits tell you the 4 next state bits, \(2^{10} \times 4\) bits of ROM
  - Total: 4.3K bits of ROM

- PLA is much smaller
  - can share product terms
  - only need entries that produce an active output
  - can take into account don't cares

- Size is (\#inputs \(\times\) \#product-terms) + (\#outputs \(\times\) \#product-terms)
  - For this example = \((10 \times 17) + (20 \times 17) = 460\) PLA cells

- PLA cells usually about the size of a ROM cell (slightly bigger)
Another Implementation Style

- Complex instructions: the "next state" is often current state + 1
Details

### Dispatch ROM 1

<table>
<thead>
<tr>
<th>Op</th>
<th>Opcode name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>R-format</td>
<td>0110</td>
</tr>
<tr>
<td>000010</td>
<td>jmp</td>
<td>1001</td>
</tr>
<tr>
<td>000100</td>
<td>beq</td>
<td>1000</td>
</tr>
<tr>
<td>100011</td>
<td>lw</td>
<td>0010</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
<td>0010</td>
</tr>
</tbody>
</table>

### Dispatch ROM 2

<table>
<thead>
<tr>
<th>Op</th>
<th>Opcode name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td>lw</td>
<td>0011</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
<td>0101</td>
</tr>
</tbody>
</table>

#### State number

<table>
<thead>
<tr>
<th>State number</th>
<th>Address-control action</th>
<th>Value of AddrCtl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Use dispatch ROM 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Use dispatch ROM 2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
</tbody>
</table>
Microprogramming

- What are the “microinstructions”?
Microprogramming

- A specification methodology
  - appropriate if hundreds of opcodes, modes, cycles, etc.
  - signals specified symbolically using microinstructions

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU control</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Register control</th>
<th>Memory</th>
<th>PCWrite control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>ALU</td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td>Read</td>
<td></td>
<td>Dispatch 1</td>
<td></td>
</tr>
<tr>
<td>Mem1</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td></td>
<td></td>
<td>Dispatch 2</td>
<td></td>
</tr>
<tr>
<td>LW2</td>
<td></td>
<td></td>
<td></td>
<td>Read ALU</td>
<td></td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write MDR</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>SW2</td>
<td></td>
<td></td>
<td></td>
<td>Write ALU</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>Rformat1</td>
<td>Func code</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write ALU</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>BEQ1</td>
<td>Subt</td>
<td>A</td>
<td>B</td>
<td>ALUOut-cond</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUMP1</td>
<td></td>
<td></td>
<td></td>
<td>Jump address</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Will two implementations of the same architecture have the same microcode?
- What would a microassembler do?
## Microinstruction format

<table>
<thead>
<tr>
<th>Field name</th>
<th>Value</th>
<th>Signals active</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALU control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>ALUOp = 00</td>
<td></td>
<td>Cause the ALU to add.</td>
</tr>
<tr>
<td>Subt</td>
<td>ALUOp = 01</td>
<td></td>
<td>Cause the ALU to subtract; this implements the compare for branches.</td>
</tr>
<tr>
<td>Func code</td>
<td>ALUOp = 10</td>
<td></td>
<td>Use the instruction's function code to determine ALU control.</td>
</tr>
<tr>
<td><strong>SRC1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>ALUSrcA = 0</td>
<td></td>
<td>Use the PC as the first ALU input.</td>
</tr>
<tr>
<td>A</td>
<td>ALUSrcA = 1</td>
<td></td>
<td>Register A is the first ALU input.</td>
</tr>
<tr>
<td><strong>SRC2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>ALUSrcB = 00</td>
<td></td>
<td>Register B is the second ALU input.</td>
</tr>
<tr>
<td>4</td>
<td>ALUSrcB = 01</td>
<td></td>
<td>Use 4 as the second ALU input.</td>
</tr>
<tr>
<td>Extend</td>
<td>ALUSrcB = 10</td>
<td></td>
<td>Use output of the sign extension unit as the second ALU input.</td>
</tr>
<tr>
<td>Extshift</td>
<td>ALUSrcB = 11</td>
<td></td>
<td>Use the output of the shift-by-two unit as the second ALU input.</td>
</tr>
<tr>
<td><strong>Register control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td></td>
<td>RegWrite, RegDst = 1, MemtoReg = 0</td>
<td>Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.</td>
</tr>
<tr>
<td>Write ALU</td>
<td></td>
<td>RegWrite, RegDst = 1, MemtoReg = 0</td>
<td>Write a register using the rd field of the IR as the register number and the contents of the MDR as the data.</td>
</tr>
<tr>
<td>Write MDR</td>
<td></td>
<td>RegWrite, RegDst = 1, MemtoReg = 1</td>
<td>Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read PC</td>
<td>MemRead, lorD = 0</td>
<td></td>
<td>Read memory using the PC as address; write result into IR (and the MDR).</td>
</tr>
<tr>
<td>Read ALU</td>
<td>MemRead, lorD = 1</td>
<td></td>
<td>Read memory using the ALUOut as address; write result into MDR.</td>
</tr>
<tr>
<td>Write ALU</td>
<td>MemWrite, lorD = 1</td>
<td></td>
<td>Write memory using the ALUOut as address, contents of B as the data.</td>
</tr>
<tr>
<td><strong>PC write control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>PCSource = 00</td>
<td></td>
<td>Write the output of the ALU into the PC.</td>
</tr>
<tr>
<td>ALUOut-cond</td>
<td>PCSource = 01, PCWriteCond</td>
<td></td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</td>
</tr>
<tr>
<td>jump address</td>
<td>PCSource = 10, PCWrite</td>
<td></td>
<td>Write the PC with the jump address from the instruction.</td>
</tr>
<tr>
<td><strong>Sequencing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq</td>
<td>AddrCtl = 11</td>
<td></td>
<td>Choose the next microinstruction sequentially.</td>
</tr>
<tr>
<td>Fetch</td>
<td>AddrCtl = 00</td>
<td></td>
<td>Go to the first microinstruction to begin a new instruction.</td>
</tr>
<tr>
<td>Dispatch 1</td>
<td>AddrCtl = 01</td>
<td></td>
<td>Dispatch using the ROM 1.</td>
</tr>
<tr>
<td>Dispatch 2</td>
<td>AddrCtl = 10</td>
<td></td>
<td>Dispatch using the ROM 2.</td>
</tr>
</tbody>
</table>
Maximally vs. Minimally Encoded

• No encoding:
  – 1 bit for each datapath operation
  – faster, requires more memory (logic)
  – used for Vax 780 — an astonishing 400K of memory!

• Lots of encoding:
  – send the microinstructions through logic to get control signals
  – uses less memory, slower

• Historical context of CISC:
  – Too much logic to put on a single chip with everything else
  – Use a ROM (or even RAM) to hold the microcode
  – It’s easy to add new instructions
Microcode: Trade-offs

- Distinction between specification and implementation is sometimes blurred

- Specification Advantages:
  - Easy to design and write
  - Design architecture and microcode in parallel

- Implementation (off-chip ROM) Advantages
  - Easy to change since values are in memory
  - Can emulate other architectures
  - Can make use of internal registers

- Implementation Disadvantages, SLOWER now that:
  - Control is implemented on same chip as processor
  - ROM is no longer faster than RAM
  - No need to go back and make changes
## The Big Picture

<table>
<thead>
<tr>
<th>Initial representation</th>
<th>Finite state diagram</th>
<th>Microprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequencing control</strong></td>
<td>Explicit next state function</td>
<td>Microprogram counter + dispatch ROMS</td>
</tr>
<tr>
<td><strong>Logic representation</strong></td>
<td>Logic equations</td>
<td>Truth tables</td>
</tr>
<tr>
<td><strong>Implementation technique</strong></td>
<td>Programmable logic array</td>
<td>Read only memory</td>
</tr>
</tbody>
</table>