The Processor: Datapath & Control

- We're ready to look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: lw, sw
  - arithmetic-logical instructions: add, sub, and, or, slt
  - control flow instructions: 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

Control

- e.g., what should the ALU do with this instruction
  Example: lw $1, 100($2)
  
<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

- ALU control input
  
  | 000 | AND | 001 | OR |
  | 010 | add | 110 | subtract |
  | 111 | set-on-less-than |

- Why is the code for subtract 110 and not 011?
• 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>Funct Sel</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp1</td>
<td>ALUOp0</td>
<td>F5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

• 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? 10 states, \(< 2^{4} \) 4 bits

Graphical Specification of FSM

Finite State Machine for Control

Implementation:

PLA Implementation

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

- Red color shows lines that are "high" or "1"
- Orange data are AND gates
- Grey data are OR gates, 1 or hi-Z

State 1 (0001) is followed by state 2 (0010) if Op = 100010, with ALUsrce0 and ALUsrce1 set to "1" (true).

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.

$$m \times n$$

m is the "height", 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., $2^6 = 1024$ different addresses)
- How many outputs are there?
  - 16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is $2^6 \times 20 = 20K$ 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 ($\text{inputs} \times \text{product-terms}) + (\text{outputs} \times \text{product-terms})$
  - 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**

<table>
<thead>
<tr>
<th>Instruction Register Opcode Field</th>
<th>Instruction Register Opcode Field</th>
<th>Instruction Register Opcode Field</th>
<th>Instruction Register Opcode Field</th>
<th>Instruction Register Opcode Field</th>
<th>Instruction Register Opcode Field</th>
<th>Instruction Register Opcode Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
</tr>
<tr>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
</tr>
<tr>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
</tr>
<tr>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
</tr>
<tr>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
</tr>
<tr>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
</tr>
<tr>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</td>
<td>Add 1</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>Addr Control</th>
<th>Write</th>
<th>Memory</th>
<th>PCWrite</th>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>PC</td>
<td>Load</td>
</tr>
<tr>
<td>Sequence</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>IR</td>
<td>Jump</td>
</tr>
<tr>
<td>Dispatch</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>RT</td>
<td>Add</td>
</tr>
<tr>
<td>ALU</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>Shift</td>
</tr>
<tr>
<td>Instruction</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>IR</td>
<td>Logic</td>
</tr>
</tbody>
</table>

- Will two implementations of the same architecture have the same microcode?
- What would a microassembler do?
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>Sequencing Control</td>
<td>Explicit Next-State Function</td>
<td>Microprogram Counter + Dispatch ROMS</td>
</tr>
<tr>
<td>Logic Representation</td>
<td>Logic Equation</td>
<td>Truth Tables</td>
</tr>
<tr>
<td>Implementation Technique</td>
<td>Programmable Logic Array</td>
<td>Read Only Memory</td>
</tr>
</tbody>
</table>