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 $t3$ take 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 micro-programming
- Implementation can be derived from specification

Deciding the Control

- In each clock cycle, decide all the action that needs to be taken
- The control signal can be 0 and 1 or x (don’t care)
- Make a signal an x if you can to reduce control
- An action that may destroy any useful value be not allowed
- Control signal required
  - ALU: SRC1 (1 bit), SRC2(2 bits),
    - operation (Add, Sub, or from FC)
  - Memory: address (I or D), read, write, data in IR or MDR
  - Register File: address r or d, data (MDR/ALUOUT), read, write
  - PC: PCwrite, PCwrite-conditional, Data (PC+4, branch, jump)
- Control signal can be implied (register file read are values in A and B registers actually A and B need not be registers at all)
- Explicit control vs indirect control (derived based on input like instruction being executed, or function code field) bits

Graphical Specification of FSM

- How many state bits will we need?
- 4 bits.
- Why?

Finite State Machine: Control Implementation

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.

$$\begin{array}{c}
000000 \ 000001 \ 000010 \ 000011 \\
000100 \ 000101 \ 000110 \ 000111 \\
001000 \ 001001 \ 001010 \ 001011 \\
001100 \ 001101 \ 001110 \ 001111 \\
010000 \ 010001 \ 010010 \ 010011 \\
010100 \ 010101 \ 010110 \ 010111 \\
011000 \ 011001 \ 011010 \ 011011 \\
011100 \ 011101 \ 011110 \ 011111 \\
100000 \ 100001 \ 100010 \ 100011 \\
100100 \ 100101 \ 100110 \ 100111 \\
101000 \ 101001 \ 101010 \ 101011 \\
101100 \ 101101 \ 101110 \ 101111 \\
110000 \ 110001 \ 110010 \ 110011 \\
110100 \ 110101 \ 110110 \ 110111 \\
111000 \ 111001 \ 111010 \ 111011 \\
111100 \ 111101 \ 111110 \ 111111
\end{array}$$

$m$ is the “height”, and $n$ is the “width”

ROM Implementation

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

ROM Implementation

- 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)

ROM vs PLA

Another Implementation Style

- Complex instruction: the “next state” is often current state + 1

Details-1

<table>
<thead>
<tr>
<th>Dispatch/ROM1</th>
<th>Dispatch/ROM2</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>000010</td>
<td>0110</td>
</tr>
<tr>
<td>000010</td>
<td>000011</td>
<td>0110</td>
</tr>
<tr>
<td>000100</td>
<td>000101</td>
<td>0110</td>
</tr>
<tr>
<td>000110</td>
<td>000111</td>
<td>0110</td>
</tr>
<tr>
<td>001000</td>
<td>001001</td>
<td>0000</td>
</tr>
<tr>
<td>001010</td>
<td>001011</td>
<td>0000</td>
</tr>
<tr>
<td>001100</td>
<td>001101</td>
<td>0000</td>
</tr>
<tr>
<td>001110</td>
<td>001111</td>
<td>0000</td>
</tr>
<tr>
<td>010000</td>
<td>010001</td>
<td>0000</td>
</tr>
<tr>
<td>010010</td>
<td>010011</td>
<td>0000</td>
</tr>
<tr>
<td>010100</td>
<td>010101</td>
<td>0000</td>
</tr>
<tr>
<td>010110</td>
<td>010111</td>
<td>0000</td>
</tr>
<tr>
<td>011000</td>
<td>011001</td>
<td>0000</td>
</tr>
<tr>
<td>011010</td>
<td>011011</td>
<td>0000</td>
</tr>
<tr>
<td>011100</td>
<td>011101</td>
<td>0000</td>
</tr>
<tr>
<td>011110</td>
<td>011111</td>
<td>0000</td>
</tr>
<tr>
<td>100000</td>
<td>100001</td>
<td>0000</td>
</tr>
<tr>
<td>100010</td>
<td>100011</td>
<td>0000</td>
</tr>
<tr>
<td>100100</td>
<td>100101</td>
<td>0000</td>
</tr>
<tr>
<td>100110</td>
<td>100111</td>
<td>0000</td>
</tr>
</tbody>
</table>

Details-2

<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 ROM1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Use dispatch ROM2</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 is a “microinstruction”?

- **Microcode memory**
- **Datapath**
- **Microinstruction format**
- **Maximally vs. Minimally Encoded**
- **Microcode: Trade-offs**
- **The Big Picture**

**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>BUS</th>
<th>SR2</th>
<th>Register control</th>
<th>Memory</th>
<th>PCWrite control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>You can add two values together.</td>
<td>You can add two values together.</td>
<td>You can add two values together.</td>
<td>You can add two values together.</td>
<td>You can add two values together.</td>
<td>You can add two values together.</td>
<td>You can add two values together.</td>
</tr>
<tr>
<td>ALU</td>
<td>Subtraction</td>
<td>Subtraction</td>
<td>Subtraction</td>
<td>Subtraction</td>
<td>Subtraction</td>
<td>Subtraction</td>
<td>Subtraction</td>
</tr>
<tr>
<td>Func code</td>
<td>You can use the function code to determine ALU control.</td>
<td>You can use the function code to determine ALU control.</td>
<td>You can use the function code to determine ALU control.</td>
<td>You can use the function code to determine ALU control.</td>
<td>You can use the function code to determine ALU control.</td>
<td>You can use the function code to determine ALU control.</td>
<td>You can use the function code to determine ALU control.</td>
</tr>
<tr>
<td>SRC1</td>
<td>You can use the PC as the first ALU input.</td>
<td>You can use the PC as the first ALU input.</td>
<td>You can use the PC as the first ALU input.</td>
<td>You can use the PC as the first ALU input.</td>
<td>You can use the PC as the first ALU input.</td>
<td>You can use the PC as the first ALU input.</td>
<td>You can use the PC as the first ALU input.</td>
</tr>
<tr>
<td>SRC2</td>
<td>You can use 4 as the second ALU input.</td>
<td>You can use 4 as the second ALU input.</td>
<td>You can use 4 as the second ALU input.</td>
<td>You can use 4 as the second ALU input.</td>
<td>You can use 4 as the second ALU input.</td>
<td>You can use 4 as the second ALU input.</td>
<td>You can use 4 as the second ALU input.</td>
</tr>
<tr>
<td>Extend</td>
<td>You can use the output of the sign extension unit as the second ALU input.</td>
<td>You can use the output of the sign extension unit as the second ALU input.</td>
<td>You can use the output of the sign extension unit as the second ALU input.</td>
<td>You can use the output of the sign extension unit as the second ALU input.</td>
<td>You can use the output of the sign extension unit as the second ALU input.</td>
<td>You can use the output of the sign extension unit as the second ALU input.</td>
<td>You can use the output of the sign extension unit as the second ALU input.</td>
</tr>
<tr>
<td>Extshft</td>
<td>You can use the output of the shift-by-two unit as the second ALU input.</td>
<td>You can use the output of the shift-by-two unit as the second ALU input.</td>
<td>You can use the output of the shift-by-two unit as the second ALU input.</td>
<td>You can use the output of the shift-by-two unit as the second ALU input.</td>
<td>You can use the output of the shift-by-two unit as the second ALU input.</td>
<td>You can use the output of the shift-by-two unit as the second ALU input.</td>
<td>You can use the output of the shift-by-two unit as the second ALU input.</td>
</tr>
<tr>
<td>Read</td>
<td>You can 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.</td>
<td>You can 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.</td>
<td>You can 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.</td>
<td>You can 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.</td>
<td>You can 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.</td>
<td>You can 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.</td>
<td>You can 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.</td>
</tr>
<tr>
<td>Write</td>
<td>You can write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.</td>
<td>You can write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.</td>
<td>You can write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.</td>
<td>You can write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.</td>
<td>You can write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.</td>
<td>You can write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.</td>
<td>You can 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>ALU PCSource</td>
<td>You can write the output of the ALU into the PC.</td>
<td>You can write the output of the ALU into the PC.</td>
<td>You can write the output of the ALU into the PC.</td>
<td>You can write the output of the ALU into the PC.</td>
<td>You can write the output of the ALU into the PC.</td>
<td>You can write the output of the ALU into the PC.</td>
<td>You can write the output of the ALU into the PC.</td>
</tr>
<tr>
<td>PCWrite</td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</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>PCWriteCond</td>
<td>Write the PC with the jump address from the instruction.</td>
<td>Write the PC with the jump address from the instruction.</td>
<td>Write the PC with the jump address from the instruction.</td>
<td>Write the PC with the jump address from the instruction.</td>
<td>Write the PC with the jump address from the instruction.</td>
<td>Write the PC with the jump address from the instruction.</td>
<td>Write the PC with the jump address from the instruction.</td>
</tr>
<tr>
<td>Fetch AddrCtl</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
</tr>
<tr>
<td>Dispatch 1 AddrCtl</td>
<td>Dispatch using the ROM 1.</td>
<td>Dispatch using the ROM 1.</td>
<td>Dispatch using the ROM 1.</td>
<td>Dispatch using the ROM 1.</td>
<td>Dispatch using the ROM 1.</td>
<td>Dispatch using the ROM 1.</td>
<td>Dispatch using the ROM 1.</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 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**
Exceptions

- What should the machine do if there is a problem
- Exceptions are just that
  - Changes in the normal execution of a program
- Two types of exceptions
  - External Condition: I/O interrupt, power failure, user termination signal (Ctrl-C)
  - Internal Condition: Bad memory read address (not a multiple of 4), illegal instructions, overflow/underflow.
- Interrupts – external
- Exceptions – internal
- Usually we refer to both by the general term “Exception”
- In either case, we need some mechanism by which we can handle the exception generated.
- Control is transferred to an exception handling mechanism, stored at a pre-specified location
- Address of instruction is saved in a register called EPC

Exceptions

- We need two special registers
  - EPC: 32 bit register to hold address of current instruction
  - Cause: 32 bit register to hold information about the type of exception that has occurred.
- Simple Exception Types
  - Undefined Instruction
  - Arithmetic Overflow
- Another type is Vectored Interrupts
  - Do not need cause register
  - Appropriate exception handler jumped to from a vector table

How Exceptions are Handled

- Usually we refer to both by the general term “Exception”
- In either case, we need some mechanism by which we can handle the exception generated.
- Address of instruction is saved in a register called EPC

Two new states for the Multi-cycle CPU

- Address of exception handler depends on the problem
  - Undefined Instruction C0 00 00 00
  - Arithmetic Overflow C0 00 00 20
  - Addresses are separated by a fixed amount, 32 bytes in MIPS
- PC is transferred to a register called EPC
- If interrupts are not vectored, then we need another register to store the cause of problem
- In what state what exception can occur?

Final Words on Single and Multi-Cycle Systems

- Single cycle implementation
  - Simpler but slowest
  - Require more hardware
- Multi-cycle
  - Faster clock
  - Amount of time it takes depends on instruction mix
  - Control more complicated
- Exceptions and Other conditions add a lot of complexity
- Other techniques to make it faster

Conclusions on Chapter 5

- Control is the most complex part
- Can be hard-wired, ROM-based, or micro-programmed
- Simpler instructions also lead to simple control
- Just because machine is micro-programmed, we should not add complicated instructions
- Sometimes simple instructions are more effective than a single complex instruction
- More complex instructions may have to be maintained for compatibility reasons