Control & Execution

Finite State Machines for Control

MIPS Execution
On the leading edge of the clock, the input of a latch is transferred to the output and held.

We must be sure the output of the combinational logic has **settled** before the next leading clock edge.
Asynchronous Systems

No clock!
The data carries a “valid” signal along with it
System goes at greatest possible speed.
Only “computes” when necessary.

Everything we look at in this class will be synchronous
Fetching Sequential Instructions

How about branch?
Datapath for R-type Instructions

- **Inst Bits 25-21**: Read Reg. 1 (rs)
- **Inst Bits 20-16**: Read Reg. 2 (rt)
- **Inst Bits 15-11**: Write Reg. (rd)
- **Write Data**: ALU Operation
- **RegWrite**: ALU Operation

Data Inputs:
- Data 1: 32 bits
- Data 2: 32 bits
The select signal determines which of the inputs is connected to the output.
Inside there is a 32 way MUX per bit

For EACH bit in the 32 bit register

And this is just one port! Remember, we have data1 and data2 coming out of the register file!

LOT’S OF CONNECTIONS!
Our Register File has 3 ports

This is one reason we have only a small number of registers

What's another reason?

REALLY LOTS OF CONNECTIONS!
Implementing Logical Functions

Suppose we want to map M input bits to N output bits.

For example, we need to take the OPCODE field from the instruction and determine what OPERATION to send to the ALU.
Remember our ALU?

Flags
V, C

N Flag

R

Z Flag

Sub

Add/Sub

Bidirectional Shifter

Boolean

That's a lot of stuff

Math

Bool

Shft
Implementing Logical Functions

Suppose we want to map M input bits to N output bits

For example, we need to take the OPCODE field from the instruction and determine what OPERATION to send to the ALU.
We can get 1 bit out with a MUX

Wire these to HIGH or LOW depending on the value you want OUT for that INPUT

For example, 3 input AND has INPUT 7 wired HIGH and all the others wired LOW.
Or use a ROM

M-bit Address → Read-Only Memory → N-bit Result
Or use sum-of-products

Think of the SUM of PRODUCTS form.

The AND layer generates the products of various input bits

The OR layer combines the products into various outputs

You could also use two NAND layers instead

Could be implemented using Boolean gates, or also using a “programmable logic array” (PLA) [similar to a PROM, but both the AND and the OR parts are programmable].
Finite State Machines

• A set of STATES
• A set of INPUTS
• A set of OUTPUTS
• A function to map the STATE and the INPUT into the next STATE and an OUTPUT

Remember automata?
Traffic Light Controller
## Controller State Table

<table>
<thead>
<tr>
<th>State</th>
<th>NScar</th>
<th>Ewcar</th>
<th>Next</th>
<th>NSlight</th>
<th>EWlight</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSgreen</td>
<td>0</td>
<td>0</td>
<td>NSgreen</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NSgreen</td>
<td>0</td>
<td>1</td>
<td>EWgreen</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>NSgreen</td>
<td>1</td>
<td>0</td>
<td>NSgreen</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NSgreen</td>
<td>1</td>
<td>1</td>
<td>EWgreen</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>EWgreen</td>
<td>0</td>
<td>0</td>
<td>EWgreen</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>EWgreen</td>
<td>0</td>
<td>1</td>
<td>EWgreen</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>EWgreen</td>
<td>1</td>
<td>0</td>
<td>NSgreen</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>EWgreen</td>
<td>1</td>
<td>1</td>
<td>NSgreen</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Implementing an FSM
FSM Example: Recognizing Numbers

Recognize the regular expression for floating point numbers

\[ \texttt{[\ \textbackslash t]* [-+]?[0-9]*(. [0-9]*)? (e[-+]?[0-9]+)?} \]

Examples:

- \texttt{+123.456e23}  \quad \text{“a” matches itself}
- \texttt{0.456}  \quad \text{“[abc]” matches one of a, b, or c}
- \texttt{1.5e-10}  \quad \text{“[a-z]” matches one of a, b, c, d, …, x, y, or z}
- \texttt{-123}  \quad \text{“0*” matches zero or more 0’s (””, “0”, “00”, “0000”)}
- \texttt{“Z?”” matches zero or 1 Z’s}
### FSM Table

**IN : STATE → NEW STATE**

<table>
<thead>
<tr>
<th>Input</th>
<th>New State</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>start → start</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>.</td>
<td>start → frac</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>.</td>
<td>sign → frac</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>.</td>
<td>whole → frac</td>
</tr>
<tr>
<td>.</td>
<td>whole → done</td>
</tr>
<tr>
<td>e</td>
<td>whole → exp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>New State</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>frac → done</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>frac → exp</td>
</tr>
<tr>
<td>.</td>
<td>exp → exp</td>
</tr>
</tbody>
</table>

**STATE ASSIGNMENTS**

<table>
<thead>
<tr>
<th>State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0 000</td>
</tr>
<tr>
<td>sign</td>
<td>1 001</td>
</tr>
<tr>
<td>whole</td>
<td>2 010</td>
</tr>
<tr>
<td>frac</td>
<td>3 011</td>
</tr>
<tr>
<td>exp</td>
<td>4 100</td>
</tr>
<tr>
<td>done</td>
<td>5 101</td>
</tr>
<tr>
<td>error</td>
<td>6 110</td>
</tr>
</tbody>
</table>
Our ROM has:

• 10 inputs
• 5 outputs
FSM Summary

With *just* a register and some logic, we can implement complicated sequential functions like recognizing a FP number.

This is useful in its own right for compilers, input routines, etc.

The reason we’re looking at it here is to see how designers implement the complicated sequences of events required to implement instructions.

Think of the OP-CODE as playing the role of the input character in the recognizer. The character AND the state determine the next state (and action).
Programming also

Finite State Machines are great for encoding actions in games and graphical user interfaces.


http://gameprogrammingpatterns.com/state.html
Five Execution Steps

1. Instruction Fetch

2. Instruction Decode and Register Fetch

3. Execution, Memory Address Computation, or Branch Completion

4. Memory Access or R-type instruction completion

5. Memory Read Completion

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

An FSM looks at the op-code to determine how many...
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"

\[
\begin{align*}
IR &= \text{Memory}[PC] ; \quad \text{IR is “Instruction Register”} \\
PC &= PC + 4 ;
\end{align*}
\]
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:

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

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

MDR = Memory[ALUOut];  \textit{MDR is Memory Data Register}

or

\text{Memory}[\text{ALUOut}] = B;

R-type instructions finish

\text{Reg}[\text{IR}[15-11]] = \text{ALUOut};
Step 5 Memory Read Completion

Reg[IR[20-16]] = MDR;
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>IR = Memory[PC]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC = PC + 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>A = Reg [IR[25-21]]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>decode/register fetch</td>
<td>B = Reg [IR[20-16]]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALUOut = PC + (sign-extend (IR[15-0]) &lt;&lt; 2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address</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></td>
</tr>
<tr>
<td>computation, branch/</td>
<td></td>
<td></td>
<td></td>
<td>PC = PC[31-28] ll</td>
</tr>
<tr>
<td>jump completion</td>
<td></td>
<td></td>
<td></td>
<td>(IR[25-0]&lt;&lt;2)</td>
</tr>
<tr>
<td>Memory access or R-type</td>
<td>Reg [IR[15-11]] = ALUOut</td>
<td>Load: MDR = Memory[ALUOut] or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td></td>
<td></td>
<td></td>
<td>Store: Memory [ALUOut] = B</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>