Lecture on Datapath
Outline

- Will explain the concepts of datapath and controller by means of an example
Fibonacci Sequence

- **Fibonacci sequence**
  - \( F(1) = 1 \)
  - \( F(2) = 1 \)
  - \( F(N) = F(N-1) + F(N-2) \)
  - e.g. 1, 1, 2, 3, 5, 8, 13 ...
  - With a 4-bit datapath, can only count up to 15, or assume \( N \leq 7 \), \( F(7) = 13 \)

- **Pseudo code**

  ```c
  int fibonacci (int N)
  {
      int N1 = 1, N2 = 1;
      int F, temp, c;
      for (c = N-2; c > 0; c--)
      {
          temp = N1;
          N1 = N1 + N2;
          N2 = temp;
      }
      F = N1;
      return F;
  }
  ```
Design Choices

• Can design as a single finite state machine, but very complicated - need different states to represent different combinations of $F(N)$, $F(N-1)$, and $F(N-2)$ ...

• Alternatively, can use adders/subtractors and registers to perform additions, loop count, comparisons, etc. This is referred to as the *datapath*.

• Then, build a finite state machine to control the *datapath*. 
Datapath

Decoder

R0
R1
R2
R3

MUX

ALU

Mem in
Mem out

Load

MUX

Zero flag

ALU operation
How to Implement a Register?

- Just flip-flops with a MUX in front of it to select between keeping old value or enabling the writing of new value.
How to Implement a Decoder?

- Example: 2-to-4 decoder

<table>
<thead>
<tr>
<th>addr</th>
<th>0001</th>
<th>0010</th>
<th>0100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram of 2-to-4 decoder with inputs and outputs.
• Instruction format: <opcode, operand1, operand2>
• Instructions:
  000 -- --  noop
  001 aa --  set
  010 aa --  increment
  011 aa --  decrement
  100 aa --  load
  101 aa --  store
  110 aa bb  add
  111 aa bb  copy

FSM & Nano-decoder

<table>
<thead>
<tr>
<th>FSM</th>
<th>alu_op</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>r_addr1</td>
<td></td>
</tr>
<tr>
<td>r_addr2</td>
<td></td>
</tr>
<tr>
<td>w_addr</td>
<td></td>
</tr>
<tr>
<td>we</td>
<td></td>
</tr>
<tr>
<td>load</td>
<td></td>
</tr>
</tbody>
</table>

Mem & Nano-

decoder

<table>
<thead>
<tr>
<th>start</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero_flag</td>
</tr>
<tr>
<td>done</td>
</tr>
</tbody>
</table>
## Nano-decoder

<table>
<thead>
<tr>
<th>opcode</th>
<th>opcode</th>
<th>opr1</th>
<th>opr2</th>
<th>alu_op</th>
<th>raddr1</th>
<th>raddr2</th>
<th>waddr</th>
<th>we</th>
<th>load</th>
</tr>
</thead>
<tbody>
<tr>
<td>noop</td>
<td>000</td>
<td>--</td>
<td>--</td>
<td>000</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>set</td>
<td>001</td>
<td>aa</td>
<td>--</td>
<td>001</td>
<td>--</td>
<td>--</td>
<td>aa</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>incr</td>
<td>010</td>
<td>aa</td>
<td>--</td>
<td>010</td>
<td>aa</td>
<td>--</td>
<td>aa</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>decr</td>
<td>011</td>
<td>aa</td>
<td>--</td>
<td>011</td>
<td>aa</td>
<td>--</td>
<td>aa</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>load</td>
<td>100</td>
<td>aa</td>
<td>--</td>
<td>100</td>
<td>--</td>
<td>--</td>
<td>aa</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>store</td>
<td>101</td>
<td>aa</td>
<td>--</td>
<td>101</td>
<td>aa</td>
<td>--</td>
<td>--</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>add</td>
<td>110</td>
<td>aa</td>
<td>bb</td>
<td>110</td>
<td>aa</td>
<td>bb</td>
<td>aa</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>copy</td>
<td>111</td>
<td>aa</td>
<td>bb</td>
<td>111</td>
<td>bb</td>
<td>--</td>
<td>aa</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Hints: can assign don’t cares so that alu_op = opcode, and raddr2 = opr2
- waddr (write address), we (write enable), load, and raddr1 logic more complicated
More on Datapath

• Assume 4-bits represent only positive numbers $0 \ldots 15$

• Increment can be implemented as
  - $A + 0$, carry_in = 1

• Decrement can be implemented as
  - $A + "1111"$, carry_in = 0

• zero_flag = 1 if the output of the ALU is “0000”
Fibonacci Sequence

- **Fibonacci sequence**
  - $F(1) = 1$
  - $F(2) = 1$
  - $F(N) = F(N-1) + F(N-2)$
  - e.g. 1, 1, 2,3, 5, 8, 13 ...
  - With a 4-bit datapath, can only count up to 15, or assume $N \leq 7$, $F(7) = 13$

- **Pseudo code**

```cpp
int fibonacci (int N)
{
  int N1 = 1, N2 = 1;
  int F, temp, c;

  for (c = N-2; c > 0; c--)
  {
    temp = N1;
    N1 = N1 + N2;
    N2 = temp;
  }

  F = N1;

  return F;
}
```
Pseudo Machine Code

• Specification
  - Wait until start = 1
  - Assume N-2 is provided at mem_in for computing F(N)
  - Use store instruction to output the final answer F(N) to mem_out and set done = 1 (done = 0 during other cycles)

• Register allocation
  - R0: count
  - R1: N1
  - R2: N2
  - R3: temp

• Pseudo machine code

```plaintext
while (not start) { 
    noop 
}
load R0  // set c = N-2
set R1   // set N1 = 1
set R2   // set N2 = 1
store R0 // test zero
while (not zero_flag) { 
    copy R3 R1 
    add R1 R2 
    copy R2 R3 
    decre R0 
}
store R1, done = 1
```

go back to initial state
FSM (Moore Machine)

<table>
<thead>
<tr>
<th>state</th>
<th>start/zero_flag</th>
<th>opcode</th>
<th>opr1</th>
<th>opr2</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>000</td>
<td>--</td>
<td>--</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>100</td>
<td>R0</td>
<td>--</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>001</td>
<td>R1</td>
<td>--</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>001</td>
<td>R2</td>
<td>--</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>101</td>
<td>R0</td>
<td>--</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>111</td>
<td>R3</td>
<td>R1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td>110</td>
<td>R1</td>
<td>R2</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td>111</td>
<td>R2</td>
<td>R3</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>I</td>
<td>011</td>
<td>R0</td>
<td>--</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>J</td>
<td>101</td>
<td>R1</td>
<td>--</td>
<td>1</td>
</tr>
</tbody>
</table>

* Note: when start = 1, just start the calculation over
Additional Comments

• In the Fibonacci example, we’ve “hard-coded” the algorithm into a finite state machine

• In ECE 30 (or another computer architecture class), you will learn about microprocessors

• Rather than hard-coding an algorithm into a finite state machine, a microprocessor supports an instruction-set, where instructions control a corresponding datapath

• Instructions are fetched from external memory and sequenced via a program counter