Overview
- Last lecture
  - Ant-brain FSM
- Today
  - Sequential Logic Examples

Sequential logic examples
- Basic design approach: a 4-step design process
- Hardware description languages and finite state machines
- Implementation examples and case studies
  - finite-string pattern recognizer
  - complex counter
  - traffic light controller
  - door combination lock

Finite string pattern recognizer (step 1)
- Finite string pattern recognizer
  - one input (X) and one output (Z)
  - output is asserted whenever the input sequence ...010... has been observed, as long as the sequence 100 has never been seen
- Step 1: understanding the problem statement
  - samples input/output behavior:
    - X: 00101010010...
    - Z: 00010101000...
    - X: 11011010010...
    - Z: 00000001000...

Finite string pattern recognizer (step 2)
- Step 2: draw state diagram
  - for the strings that must be recognized, i.e., 010 and 100
  - a Moore implementation

General FSM design procedure
- (1) Determine inputs and outputs
- (2) Determine possible states of machine
  - state minimization
- (3) Encode states and outputs into a binary code
  - state assignment or state encoding
  - output encoding
  - possibly input encoding (if under our control)
- (4) Realize logic to implement functions for states and outputs
  - combinational logic implementation and optimization
  - choices made in steps 2 and 3 can have large effect on resulting logic

Finite string pattern recognizer (step 2, cont’d)
- Exit conditions from state S3: have recognized ...010
  - if next input is 0 then have ...0100 = ...100 (state S6)
  - if next input is 1 then have ...0101 = ...101 (state S2)
- Exit conditions from S1: recognizes strings of form ...0 (no 1 seen)
  - loop back to S1 if input is 0
- Exit conditions from S4: recognizes strings of form ...1 (no 0 seen)
  - loop back to S4 if input is 1
Finite string pattern recognizer (step 2, cont'd)

- S2 and S5 still have incomplete transitions
  - S2 = . . . 01; If next input is 1, then string could be prefix of (01)(00)
  - S4 handles just this case
  - S5 = . . . 10; If next input is 1, then string could be prefix of (10)(0)
  - S2 handles just this case
- Reuse states as much as possible
  - look for same meaning
  - state minimization leads to smaller number of bits to represent states
- Once all states have a complete set of transitions we have a final state diagram

Finite string pattern recognizer (step 3)

- Verilog description including state assignment (or state encoding)

module string (clk, X, rst, Z0, Z1, Z2);
input clk, X, rst;
output Z0, Z1, Z2;
reg state[0:2];

// define states
assign S0 = [0,0,0];
assign S1 = [0,0,1];
assign S2 = [0,1,0];
assign S3 = [0,1,1];
assign S4 = [1,0,0];
assign S5 = [1,0,1];
assign S6 = [1,1,0];
assign Z0 = state[0];
assign Z1 = state[1];
assign Z2 = state[2];

always @(posedge clk) begin
  if (rst) state = S0;
  else begin
    case (state)
      S0: if (X) state = S4 else state = S1;
      S1: if (X) state = S2 else state = S1;
      S2: if (X) state = S4 else state = S3;
      S3: if (X) state = S2 else state = S6;
      S4: if (X) state = S4 else state = S5;
      S5: if (X) state = S2 else state = S6;
      S6: state = S6;
      default: begin
        $display("invalid state reached");
        state = 3'bxxx;
      end
    endcase
  end
end
endmodule

Complex counter

- A synchronous 3-bit counter has a mode control M
  - when M = 0, the counter counts up in the binary sequence
  - when M = 1, the counter advances through the Gray code sequence
  - binary: 000, 001, 010, 011, 100, 101, 110, 111
  - Gray: 000, 001, 011, 010, 110, 111, 101, 100

Valid I/O behavior (partial)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Input M</th>
<th>Current State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>011</td>
<td>100</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>110</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>
**Traffic light controller as two communicating FSMs**

- Without separate timer
  - S0 would require 7 states
  - S1 would require 3 states
  - S2 would require 7 states
  - S3 would require 3 states
  - S1 and S3 have simple transformation
  - S0 and S2 would require many more arcs
  - C could change in any of seven states

- By factoring out timer
  - greatly reduce number of states
  - 4 instead of 30
  - counter only requires seven or eight states
  - 12 total instead of 20

**Traffic light controller FSM**

- Specification of inputs, outputs, and state elements

**Complete traffic light controller**

- Tying it all together (FSM + timer)

**Traffic light controller FSM (cont’d)**

**Communicating finite state machines**

- One machine’s output is another machine’s input

**Timer for traffic light controller**

- Another FSM

```verilog
module FSM(HR, HY, HG, FR, FY, FG, ST, TS, TL, C, reset, Clk);
output HR;
output HY;
output HG;
output FR;
output FY;
output FG;
output ST;
input TS;
input TL;
input C;
input reset;
input Clk;
reg [6:1] state;
reg ST;
`define highwaygreen 6'b001100
`define highwayyellow 6'b010100
`define farmroadgreen 6'b100001
`define farmroadyellow 6'b100010
assign HR = state[6];
assign HY = state[5];
assign HG = state[4];
assign FR = state[3];
assign FY = state[2];
assign FG = state[1];
initial begin state = `highwaygreen; ST = 0; end
always @(posedge Clk)
begin
  if (reset)
  begin state = `highwaygreen; ST = 1; end
  else
  begin
    ST = 0;
    case (state)
      `highwaygreen:
        if (TL & C) begin state = `highwayyellow; ST = 1; end
      `highwayyellow:
        if (TS) begin state = `farmroadgreen; ST = 1; end
      `farmroadgreen:
        if (TL | !C) begin state = `farmroadyellow; ST = 1; end
      `farmroadyellow:
        if (TS) begin state = `highwaygreen; ST = 1; end
    endcase
  end
end
endmodule
```
**Data-path and control**

- Digital hardware systems = data-path + control
  - datapath: registers, counters, combinational functional units (e.g., ALU), communication (e.g., busses)
  - control: FSM generating sequences of control signals that instructs datapath what to do next

```
control

status info and inputs

state

control signal outputs

data-path

"puppeteer who pulls the strings"
```

**Digital combinational lock**

- Door combination lock:
  - punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
  - inputs: sequence of input values, reset
  - outputs: door open/closed
  - memory: must remember combination or always have it available
  - open questions: how do you set the internal combination?
    - stored in registers (how loaded?)
    - hardwired via switches set by user

**Implementation in software**

```
integer combination_lock ( ) {
    integer v1, v2, v3;
    integer error = 0;
    static integer c[3] = 3, 4, 2;

    while (!new_value( ));
    v1 = read_value( );
    if (v1 != c[1]) then error = 1;

    while (!new_value( ));
    v2 = read_value( );
    if (v2 != c[2]) then error = 1;

    while (!new_value( ));
    v3 = read_value( );
    if (v3 != c[3]) then error = 1;

    if (error == 1) then return(0); else return (1);
}```

**Determining details of the specification**

- How many bits per input value?
- How many values in sequence?
- How do we know a new input value is entered?
- What are the states and state transitions of the system?

**Digital combination lock state diagram**

- States: 5 states
  - represent point in execution of machine
  - each state has outputs
  - Transitions: 6 from state to state, 5 self transitions, 1 global
  - changes of state occur when clock says its ok
  - based on value of inputs
  - Inputs: reset, new, results of comparisons
  - Output: open/closed

**Data-path and control structure**

- Data-path
  - storage registers for combination values
  - multiplier
  - comparator
- Control
  - finite-state machine controller
  - control for data-path (which value to compare)
State table for combination lock

- Finite-state machine
  - refine state diagram to take internal structure into account
  - state table ready for encoding

<table>
<thead>
<tr>
<th>next state</th>
<th>new state</th>
<th>equal state</th>
<th>new state</th>
<th>equal state</th>
<th>max open/closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>S4</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S2</td>
<td>C2</td>
<td>closed</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Data-path implementation (cont’d)

- Tri-state logic
  - utilize a third output state: “no connection” or “float”
  - connect outputs together as long as only one is “enabled”
  - open-collector gates can only output 0, not 1
  - can be used to implement logical AND with only wires

<table>
<thead>
<tr>
<th>value</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>max control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Encodings for combination lock

- Encode state table
  - state can be: S1, S2, S3, OPEN, or ERR
    - needs at least 3 bits to encode: 000, 001, 010, 011, 100
    - and as many as 5: 00000, 00001, 00010, 00011, 00100, 00101, 01000, 01001, 01010, 01011, 01100, 01101, 01110, 01111, 10000, 10001, 10010, 10011, 10100, 10101, 10110, 10111, 11000, 11001, 11010, 11011, 11100, 11101, 11110, 11111

Data-path implementation for combination lock

- Multiplexer
  - easy to implement as combinational logic when few inputs
  - logic can easily get too big for most PLDs

<table>
<thead>
<tr>
<th>value</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>max control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Tri-state gates

- The third value
  - logic values: “0”, “1”
  - don’t care: “X” (must be 0 or 1 in real circuit)
  - third value or state: “2” — high impedance, infinite R, no connection

<table>
<thead>
<tr>
<th>value</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>max control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Tri-state and multiplexing

- When using tri-state logic:
  - (1) make sure never more than one “driver” for a wire at any one time
    - pull high and low at the same time can severely damage circuits
  - (2) make sure to only use value on wire when it’s being driven (using a floating value may cause failures)

- Using tri-state gate to implement an economical multiplexer

<table>
<thead>
<tr>
<th>Input</th>
<th>OE</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

CSE370, Lecture 23 25

CSE370, Lecture 23 26

CSE370, Lecture 23 27

CSE370, Lecture 23 28
Open-collector gates and wired-AND

- Open collector: another way to connect gate outputs to the same wire
  - gate only has the ability to pull its output low
  - it cannot actively drive the wire high (default — pulled high through resistor)
- Wired-AND can be implemented with open collector logic
  - if A and B are "1", output is actively pulled low
  - if C and D are "1", output is actively pulled low
  - if one gate output is low and the other high, then low wins
  - if both gate outputs are "1", the wire value "flaunts", pulled high by resistor
  - time to high transition usually slower than it would have been with a gate pulling high
  - hence, the two NAND functions are ANDed together

Digital combination lock (new data-path)

- Decrease number of inputs
- Remove 3 code digits as inputs
  - use code registers
  - make them loadable from value
  - need 3 load signal inputs (net gain in input (4+3)=3=9)
  - could be done with 2 signals and decoder
    (00, 01, 10, 11, load none)

CSE370, Lecture 23