

## 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

## 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 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

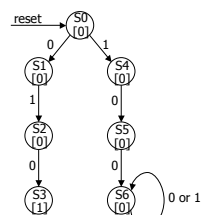
- ◆ Step 1: understanding the problem statement
  - sample input/output behavior:

X: 0 0 1 0 1 0 1 0 0 1 0 ...  
 Z: 0 0 0 1 0 1 0 1 0 0 0 ...

X: 1 1 0 1 1 0 1 0 0 1 0 ...  
 Z: 0 0 0 0 0 0 0 1 0 0 0 ...

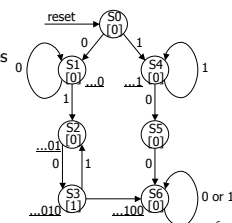
## Finite string pattern recognizer (step 2)

- ◆ Step 2: draw state diagram
  - for the strings that must be recognized, i.e., 010 and 100
  - a Moore implementation



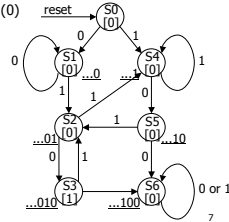
## Finite string pattern recognizer (step 2, cont'd)

- ◆ 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 = ...01 (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)1(00) S4 handles just this case
  - S5 = ...10; If next input is 1, then string could be prefix of (10)1(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



CSE370, Lecture 24

## Finite string pattern recognizer (step 3)

- ◆ Verilog description including state assignment (or state encoding)

```

module string (clk, X, rst, Q0, Q1, Q2, Z);
  input clk, X, rst;
  output Q0, Q1, Q2, Z;

  reg state[0:2];

  always @(posedge clk) begin
    if rst state = 'S0;
    else
      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 = 'S5;
        '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;
        endcase
      endcase
  end

  assign Q0 = state[0];
  assign Q1 = state[1];
  assign Q2 = state[2];
  assign Z = (state == 'S3);
endmodule
    
```

CSE370, Lecture 24

8

## Finite string pattern recognizer

- ◆ Review of process
  - understanding problem
    - ✎ write down sample inputs and outputs to understand specification
  - derive a state diagram
    - ✎ write down sequences of states and transitions for sequences to be recognized
  - minimize number of states
    - ✎ add missing transitions; reuse states as much as possible
  - state assignment or encoding
    - ✎ encode states with unique patterns
  - simulate realization
    - ✎ verify I/O behavior of your state diagram to ensure it matches specification

CSE370, Lecture 24

9

## 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)

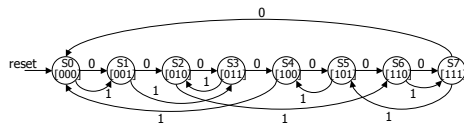
Mode Input M	Current State	Next State
0	000	001
0	001	010
1	010	110
1	110	111
1	111	101
0	101	110
0	110	111

CSE370, Lecture 24

10

## Complex counter (state diagram)

- ◆ Deriving state diagram
  - one state for each output combination
  - add appropriate arcs for the mode control



CSE370, Lecture 24

11

## Complex counter (state encoding)

- ◆ Verilog description including state encoding

```

module string (clk, M, rst, Z0, Z1, Z2);
  input clk, X, rst;
  output Z0, Z1, Z2;

  reg state[0:2];

  always @(posedge clk) begin
    if rst state = 'S0;
    else
      case (state)
        'S0: state = 'S1;
        'S1: if (M) state = 'S3 else state = 'S2;
        'S2: if (M) state = 'S6 else state = 'S3;
        'S3: if (M) state = 'S2 else state = 'S4;
        'S4: if (M) state = 'S0 else state = 'S5;
        'S5: if (M) state = 'S4 else state = 'S6;
        'S6: if (M) state = 'S7 else state = 'S7;
        'S7: if (M) state = 'S5 else state = 'S0;
      endcase
  end

  assign Z0 = state[0];
  assign Z1 = state[1];
  assign Z2 = state[2];
endmodule
    
```

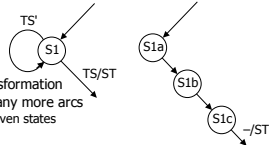
CSE370, Lecture 24

12

## Traffic light controller as two communicating FSMs

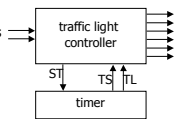
### Without separate timer

- 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 20
- counter only requires seven or eight states
  - 12 total instead of 20



CSE370, Lecture 24

13

## Traffic light controller FSM

### Specification of inputs, outputs, and state elements

```
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];
endmodule
```

specify state bits and codes for each state as well as connections to outputs

CSE370, Lecture 24

14

## Traffic light controller FSM (cont'd)

```
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
```

case statement triggered by clock edge

CSE370, Lecture 24

15

## Timer for traffic light controller

### Another FSM

```
module Timer(TS, TL, ST, Clk);
    output TS;
    output TL;
    input ST;
    input Clk;
    integer value;
    assign TS = (value >= 4); // 5 cycles after reset
    assign TL = (value >= 14); // 15 cycles after reset
    always @(posedge ST) value = 0; // async reset
    always @(posedge Clk) value = value + 1;
endmodule
```

CSE370, Lecture 24

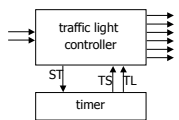
16

## Complete traffic light controller

### Tying it all together (FSM + timer)

- structural Verilog not supported by DesignWorks (which uses schematic)

```
module main(HR, HY, HG, FR, FY, FG, reset, C, Clk);
    output HR, HY, HG, FR, FY, FG;
    input reset, C, Clk;
    Timer part1(TS, TL, ST, Clk);
    FSM part2(HR, HY, HG, FR, FY, FG, ST, TS, TL, C, reset, Clk);
endmodule
```

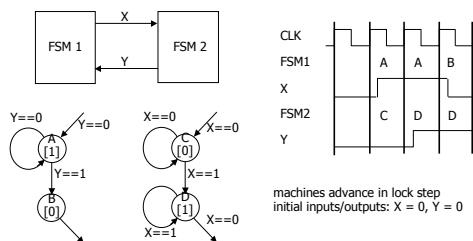


CSE370, Lecture 24

17

## Communicating finite state machines

### One machine's output is another machine's input

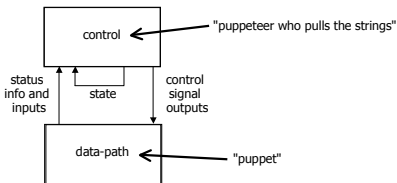


CSE370, Lecture 24

18

## 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



CSE370, Lecture 24

19

## Digital combination 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/close
  - 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

CSE370, Lecture 24

20

## 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 (v2 != c[3]) then error = 1;

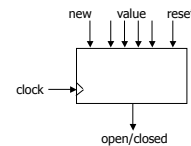
    if (error == 1) then return(0); else return (1);
}
```

CSE370, Lecture 24

21

## 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?

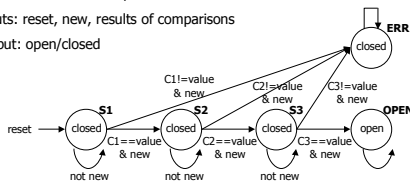


CSE370, Lecture 24

22

## 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

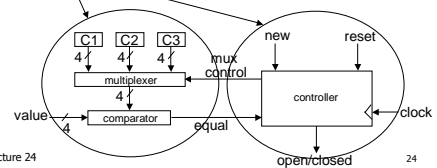


CSE370, Lecture 24

23

## Data-path and control structure

- ◆ Data-path
  - storage registers for combination values
  - multiplexer
  - comparator
- ◆ Control
  - finite-state machine controller
  - control for data-path (which value to compare)



CSE370, Lecture 24

24

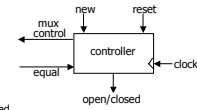
## State table for combination lock

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

reset	new	equal	state	next state	mux	open/closed
1	-	-	-	S1	C1	closed
0	0	-	S1	S1	C1	closed
0	1	0	S1	ERR	-	closed
0	1	1	S1	S2	C2	closed
...	...	...	...	...	...	...
0	1	1	S3	OPEN	-	open
...	...	...	...	...	...	...

## 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: 00001, 00010, 00100, 01000, 10000
    - choose 4 bits: 0001, 0010, 0100, 1000, 0000
  - output mux can be: C1, C2, or C3
    - needs 2 to 3 bits to encode
    - choose 3 bits: 001, 010, 100
  - output open/closed can be: open or closed
    - needs 1 or 2 bits to encode
    - choose 1 bit: 1, 0

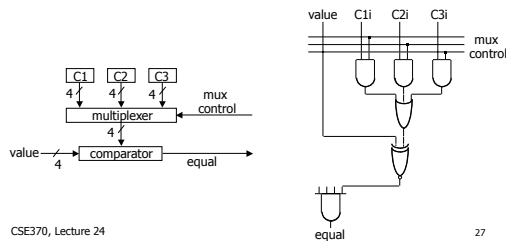


reset	new	equal	state	next state	mux	open/closed
1	-	-	-	0001	001	0
0	0	-	0001	0001	001	0
0	1	0	0001	0000	-	0
0	1	1	0001	0010	010	0
...	...	...	...	...	...	...
0	1	1	0100	1000	-	1
...	...	...	...	...	...	...

mux is identical to last 3 bits of state  
open/closed is identical to first bit of state  
therefore, we do not even need to implement  
FFs to hold state, just use outputs

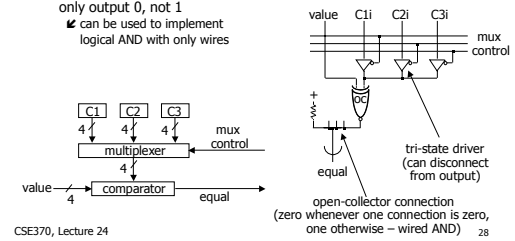
## Data-path implementation for combination lock

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



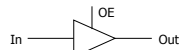
## Data-path implementation (cont'd)

- 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

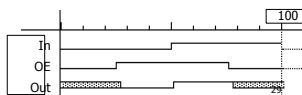


## Tri-state gates

- The third value
  - logic values: "0", "1"
  - don't care: "X" (must be 0 or 1 in real circuit!)
  - third value or state: "Z" — high impedance, infinite R, no connection
- Tri-state gates
  - additional input — output enable (OE)
  - output values are 0, 1, and Z
  - when OE is high, the gate functions normally
  - when OE is low, the gate is disconnected from wire at output
  - allows more than one gate to be connected to the same output wire
    - as long as only one has its output enabled at any one time (otherwise, sparks could fly)

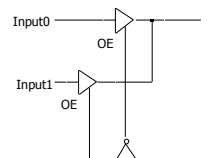


	In	OE	Out
non-inverting	X	0	Z
tri-state	0	1	0
buffer	1	1	1



## Tri-state and multiplexing

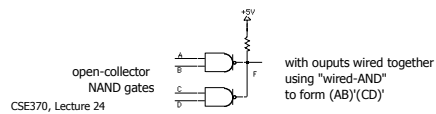
- When using tri-state logic
  - (1) make sure never more than one "driver" for a wire at any one time (pulling high and low at the same time can severely damage circuits)
  - (2) make sure to only use value on wire when its being driven (using a floating value may cause failures)
- Using tri-state gates to implement an economical multiplexer



when Select is high  
Input1 is connected to F  
when Select is low  
Input0 is connected to F  
this is essentially a 2:1 mux

## 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 "floats", pulled high by resistor
    - ⚡ low to high transition usually slower than it would have been with a gate pulling high
  - hence, the two NAND functions are ANDed together

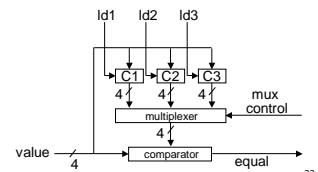


CSE370, Lecture 24

31

## Digital combination lock (new data-path)

- ◆ Decrease number of 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 (ld1, ld2, ld3, load none)



CSE370, Lecture 24

32