Sequential logic examples

- Finite state machine concept
  - FSMs are the decision making logic of digital designs
  - partitioning designs into datapath and control elements
  - when inputs are sampled and outputs asserted
- Basic design approach: a 4-step design process
- 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:
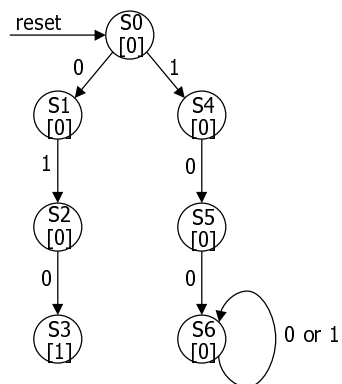
  X: 0 0 1 0 1 0 1 0 0 1 0 …
  Z: 0 0 0 1 0 1 0 1 0 0 0 …

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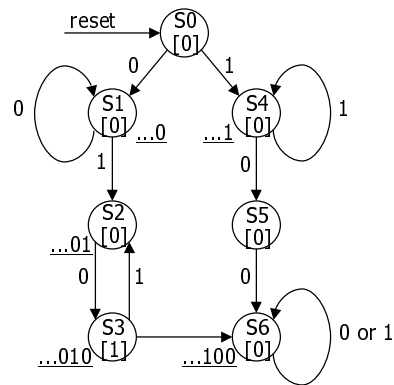
Finite string pattern recognizer (step 2, cont'd)

■ 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
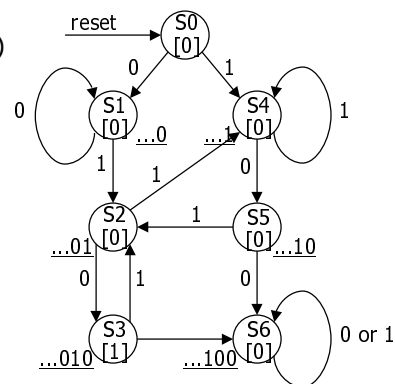
Finite string pattern recognizer (step 3)

❚ Verilog description including state assignment (or state encoding)

```
module string (clk, X, rst, Q0, Q1, Q2, Z);      always @(posedge clk) begin
input clk, X, rst;                                 if rst state = 'S0;
output Q0, Q1, Q2, Z;                              else
                                                     case (state)
reg state[0:2];                                        'S0: if (X) state = 'S4 else state = 'S1;
'define S0 = [0,0,0]; //reset state                    'S1: if (X) state = 'S2 else state = 'S1;
'define S1 = [0,0,1]; //strings ending in   ...0       'S2: if (X) state = 'S4 else state = 'S3;
'define S2 = [0,1,0]; //strings ending in  ...01       'S3: if (X) state = 'S2 else state = 'S6;
'define S3 = [0,1,1]; //strings ending in ...010       'S4: if (X) state = 'S4 else state = 'S5;
'define S4 = [1,0,0]; //strings ending in   ...1       'S5: if (X) state = 'S2 else state = 'S6;
'define S5 = [1,0,1]; //strings ending in  ...10       'S6: state = 'S6;
'define S6 = [1,1,0]; //strings ending in ...100       default: begin
                                                          $display ("invalid state reached");
assign Q0 = state[0];                                     state = 3'bxxxx;
assign Q1 = state[1];                                  endcase
assign Q2 = state[2];
assign Z = (state == 'S3);                         end

                                                 endmodule
```

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

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
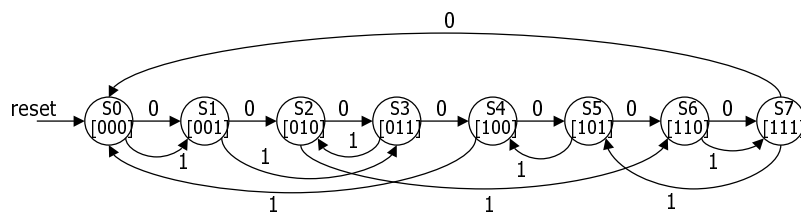
- Valid I/O behavior (partial)

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

Complex counter (state diagram)

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

Complex counter (state encoding)

∎ Verilog description including state encoding

```
module string (clk, M, rst, Z0, Z1, Z2);      always @(posedge clk) begin
input clk, X, rst;                               if rst state = `S0;
output Z0, Z1, Z2;                               else
                                                   case (state)
reg state[0:2];                                      `S0: state = `S1;
`define S0 = [0,0,0];                                `S1: if (M) state = `S3 else state = `S2;
`define S1 = [0,0,1];                                `S2: if (M) state = `S6 else state = `S3;
`define S2 = [0,1,0];                                `S3: if (M) state = `S2 else state = `S4;
`define S3 = [0,1,1];                                `S4: if (M) state = `S0 else state = `S5;
`define S4 = [1,0,0];                                `S5: if (M) state = `S4 else state = `S6;
`define S5 = [1,0,1];                                `S5: if (M) state = `S7 else state = `S7;
`define S6 = [1,1,0];                                `S5: if (M) state = `S5 else state = `S0;
`define S7 = [1,1,1];                              endcase

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

---

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
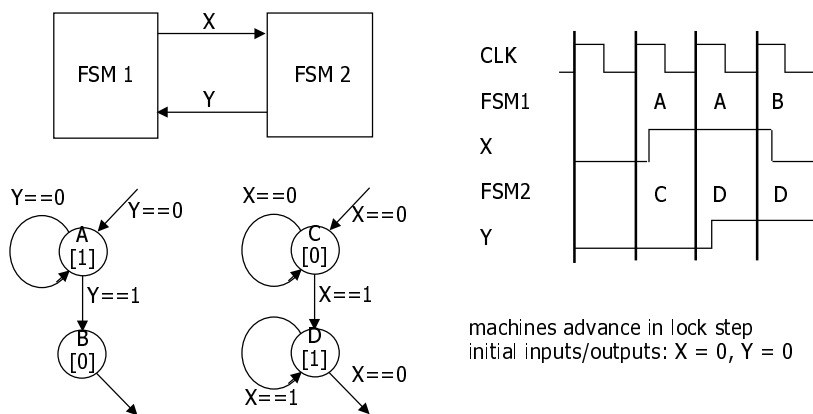   ∣ C could change in any of seven states
∎ By factoring out timer
  ∎ greatly reduce number of states
    ∣ 4 instead of 20
  ∎ counter only requires seven or eight states
    ∣ 12 total instead of 20
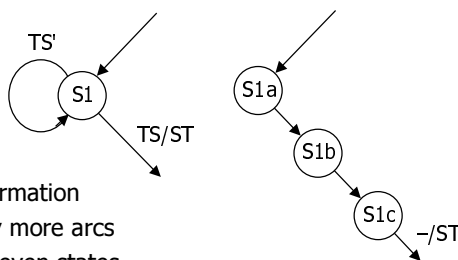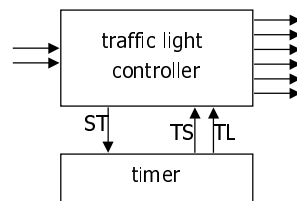
Communicating finite state machines

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

FSM 1 → X → FSM 2
FSM 1 ← Y ← FSM 2

CLK
FSM1   A   A   B
X
FSM2   C   D   D
Y

Y==0 → A [1] ← Y==0
Y==1
B [0]

X==0 → C [0] ← X==0
X==1
D [1] X==0
X==1

machines advance in lock step
initial inputs/outputs: X = 0, Y = 0

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 ← "puppeteer who pulls the strings"

status
info and   state
inputs

control
signal
outputs

data-path ← "puppet"
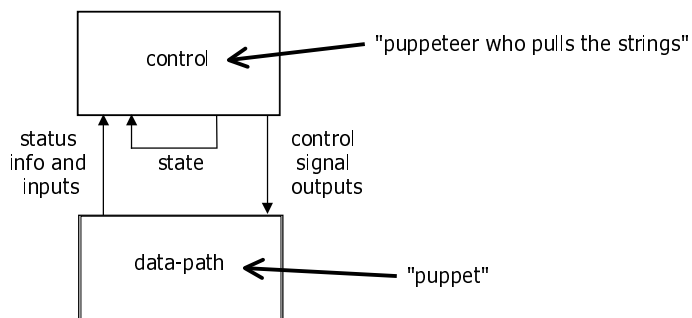
Digital combinational lock

∎ 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/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

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);
}
```
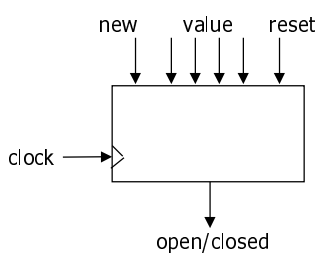
Determining details of the specification

∎ 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?
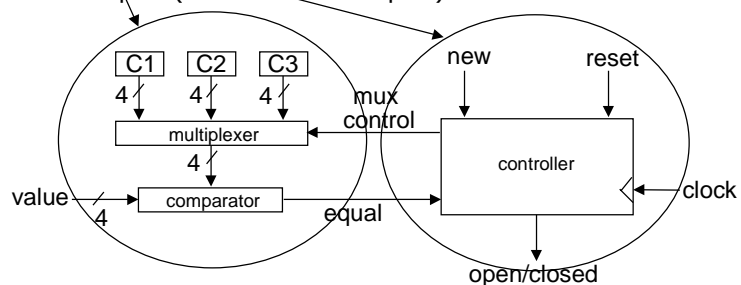

Digital combination lock state diagram

∎ 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
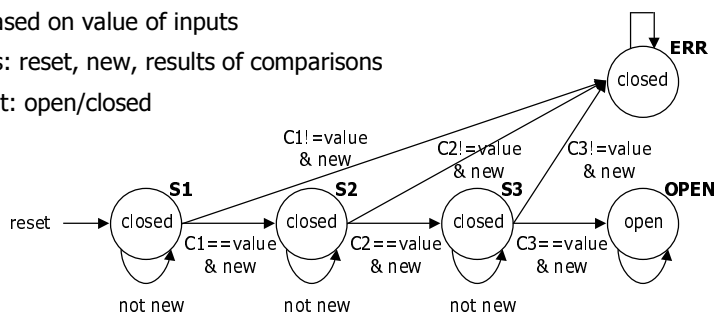
Data-path and control structure

∎ Data-path
  ∎ storage registers for combination values
  ∎ multiplexer
  ∎ 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

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
  - I needs at least 3 bits to encode: 000, 001, 010, 011, 100
  - I and as many as 5: 00001, 00010, 00100, 01000, 10000
  - I choose 4 bits: 0001, 0010, 0100, 1000, 0000
- ∎ output mux can be: C1, C2, or C3
  - I needs 2 to 3 bits to encode
  - I choose 3 bits: 001, 010, 100
- ∎ output open/closed can be: open or closed
  - I needs 1 or 2 bits to encode
 - I choose 1 bit: 1, 0
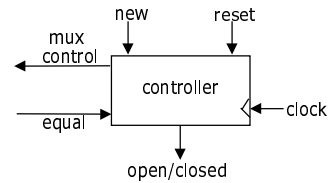
reset	new	equal	state	next state	mux	open/closed
1	–	–	–	0001	001	0
0	0	–	0001	0001	001	0
0	1	0	0001	0000	–	0
0	1	1	0001	0010	010	0
...						
0	1	1	0100	1000	–	1
...						

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
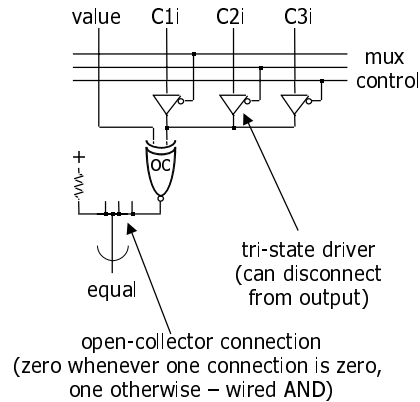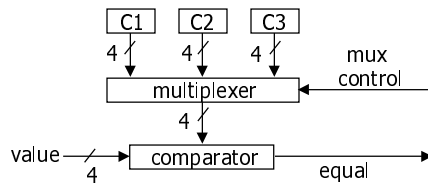
∎ Multiplexer
- ∎ easy to implement as combinational logic when few inputs
- ∎ logic can easily get too big for most PLDs
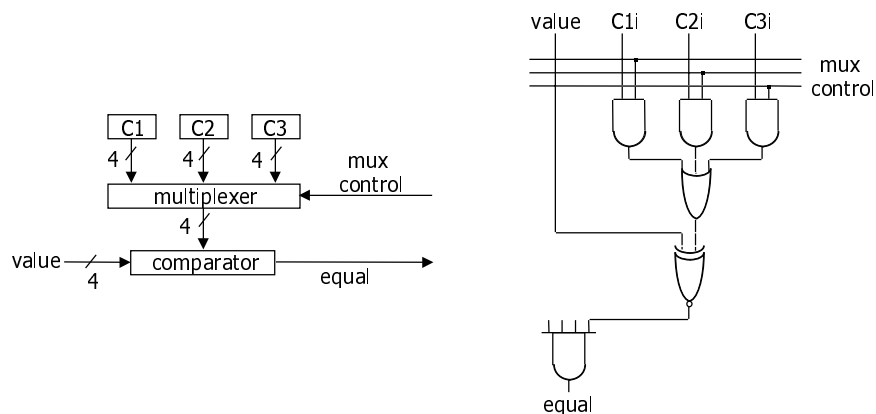
## Data-path implementation (cont'd)

∎ 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 driver
(can disconnect from output)

equal

open-collector connection
(zero whenever one connection is zero,
one otherwise – wired AND)

value ——/—— $\rightarrow$ comparator ——— equal $\rightarrow$
   4

mux control

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
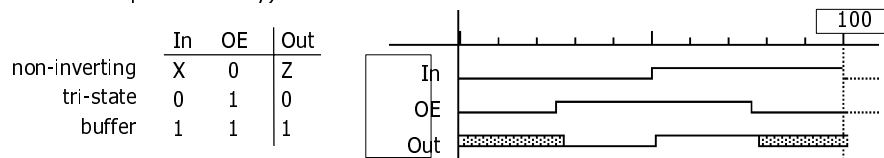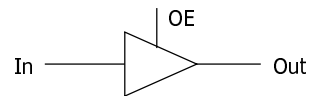
Tri-state and multiplexing

- 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)
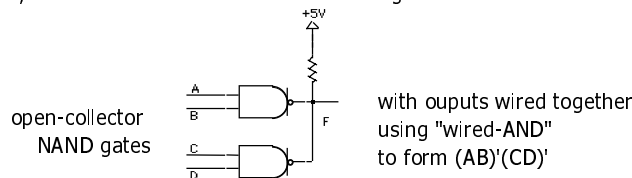- Using tri-state gates to implement an economical multiplexer

Input0 ——————▷ ———————— F
           OE
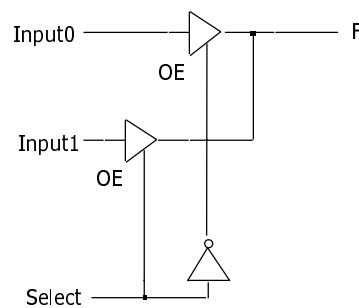
Input1 ———▷
        OE

Select ——————————△

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

open-collector
NAND gates

+5V

A
B ———▷
       F
C
D ———▷

with ouputs wired together
using "wired-AND"
to form (AB)'(CD)'

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
      (ld1, ld2, ld3, load none)
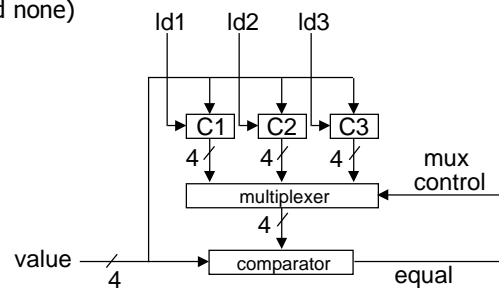
Section summary

∎ FSM design
  ∎ understanding the problem
  ∎ generating state diagram
  ∎ implementation using synthesis tools
  ∎ iteration on design/specification to improve qualities of mapping
  ∎ communicating state machines
∎ Four case studies
  ∎ understand I/O behavior
  ∎ draw diagrams
  ∎ enumerate states for the "goal"
  ∎ expand with error conditions
  ∎ reuse states whenever possible