Digital Logic Design

4190.201

2014 Spring Semester

11.Sequential Logic Examples

Naehyuck Chang
Dept. of CSE
Seoul National University
naehyuck@snu.ac.kr

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

- Determine inputs and outputs
- Determine possible states of machine
  - State minimization
- Encode states and outputs into a binary code
  - State assignment or state encoding
  - Output encoding
  - Possibly input encoding (if under our control)
- Realize logic to implement functions for states and outputs
  - Combinational logic implementation and optimization
  - Choices in steps 2 and 3 can have large effect on resulting logic

Finite string pattern recognizer (step 1)

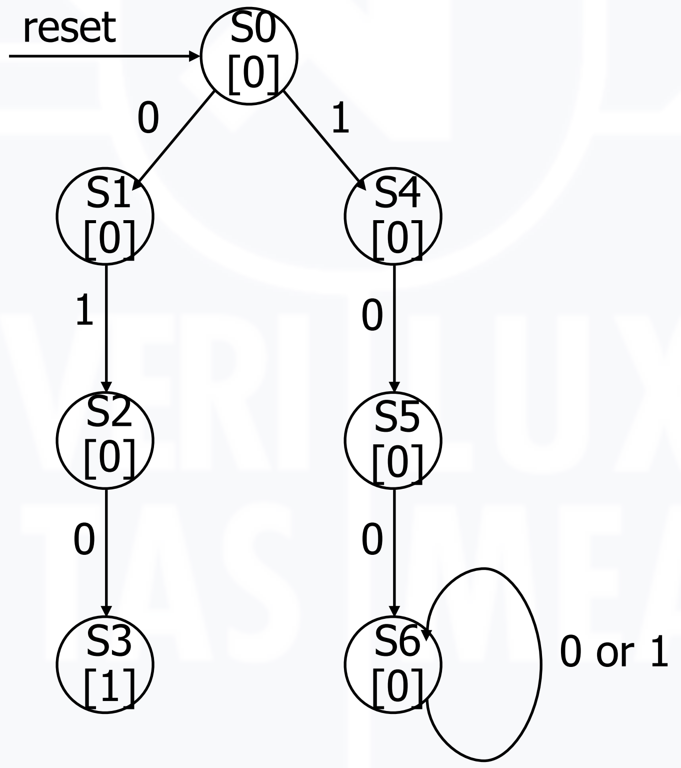
- Finite string pattern recognizer
  - One input (X) and one output (Z)
  - Output is asserted whenever the input sequence …010… has been observed, as long as the sequence …100… has never been seen

- 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 …
```

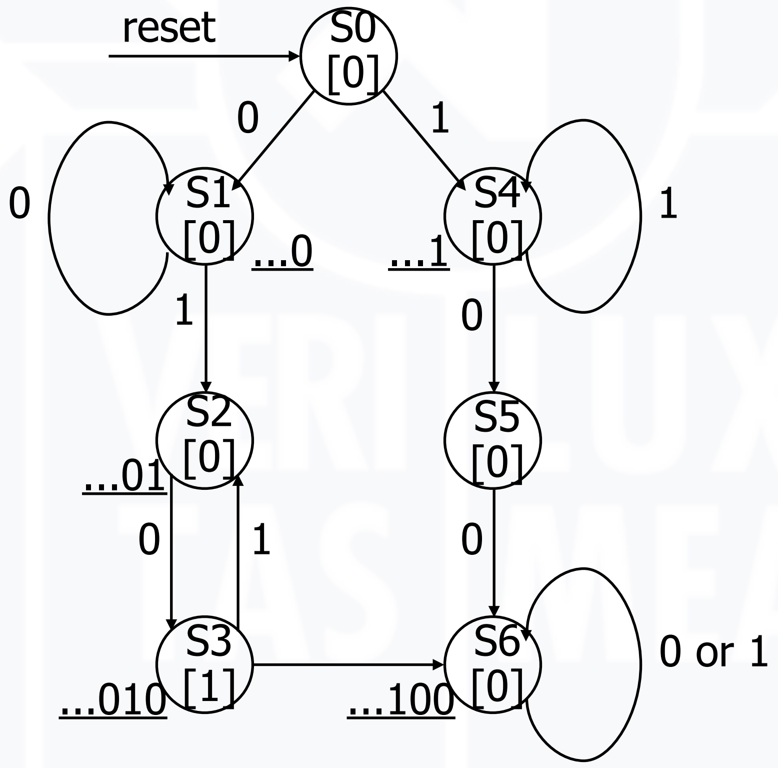
Finite string pattern recognizer (step 2)

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



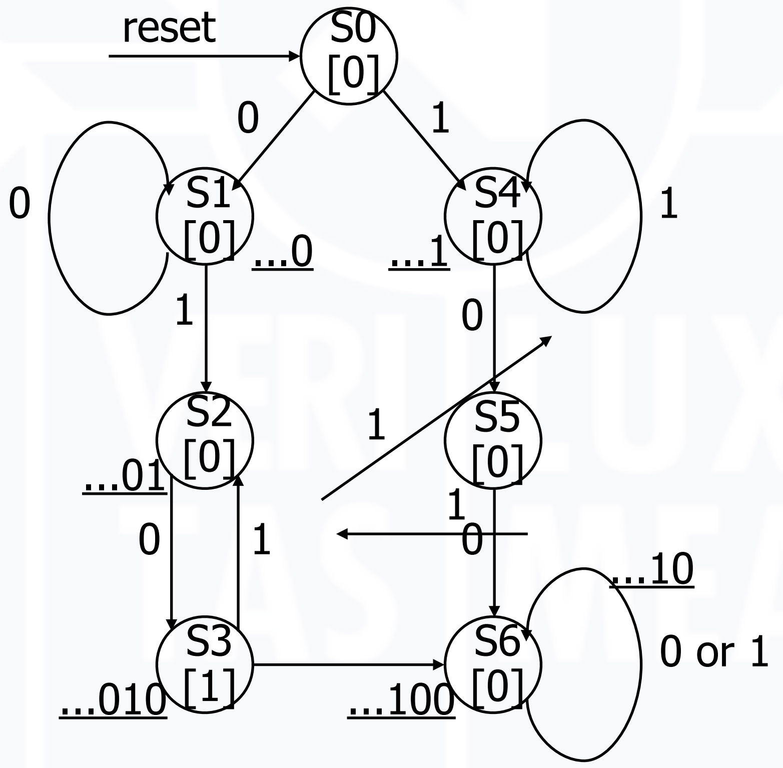
Finite string pattern recognizer (step 2, cont'd)

- 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);
input clk, X, rst;
output Q0, Q1, Q2, Z;

parameter S0 = [0,0,0]; //reset state
parameter S1 = [0,0,1]; //strings ending in    ...0
parameter S2 = [0,1,0]; //strings ending in   ...01
parameter S3 = [0,1,1]; //strings ending in ...010
parameter S4 = [1,0,0]; //strings ending in    ...1
parameter S5 = [1,0,1]; //strings ending in   ...10
parameter S6 = [1,1,0]; //strings ending in ...100

reg state[0:2];

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

```
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 = 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

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

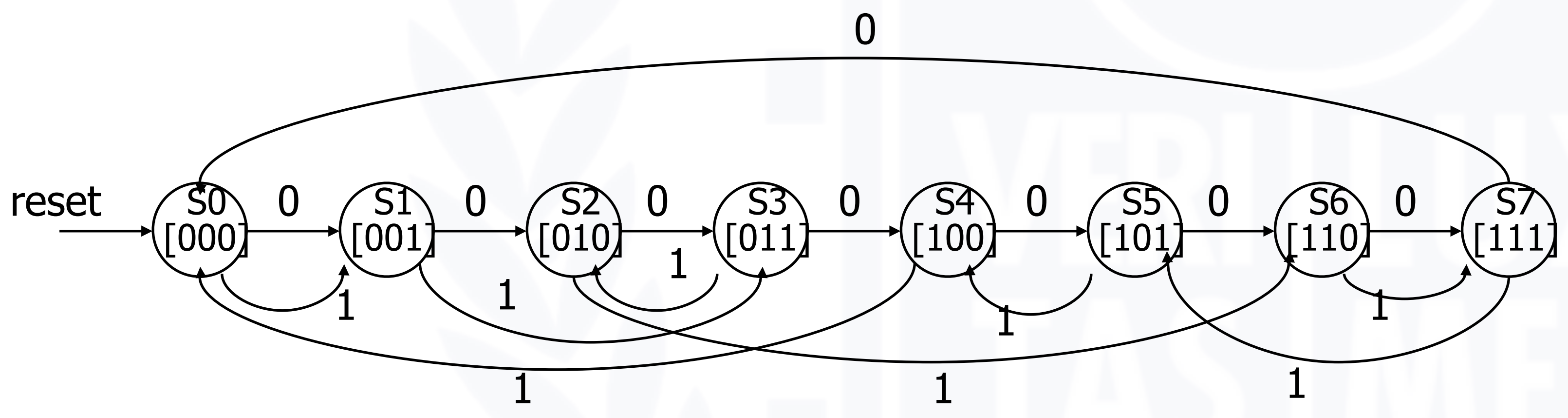
binary:	000, 001, 010, 011, 100, 101, 110, 111
Gray:	000, 001, 011, 010, 110, 111, 101, 100

- 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)

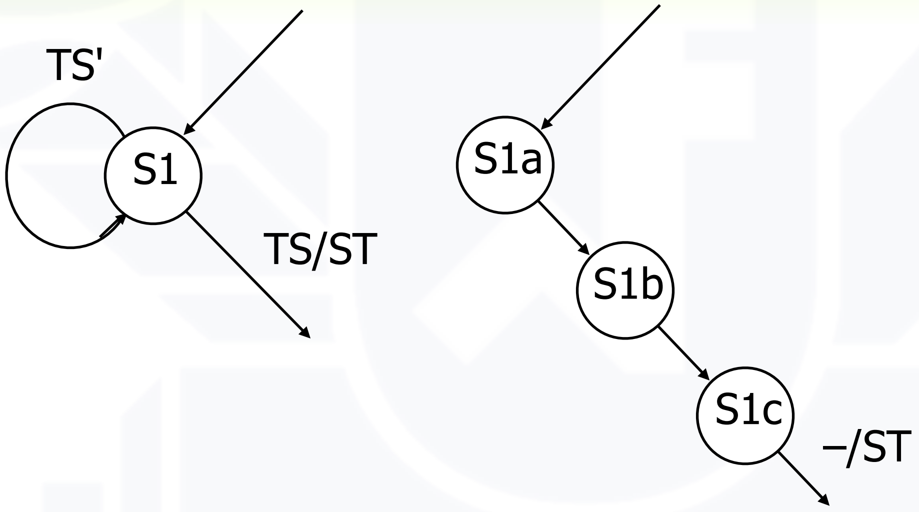
- Verilog description including state encoding

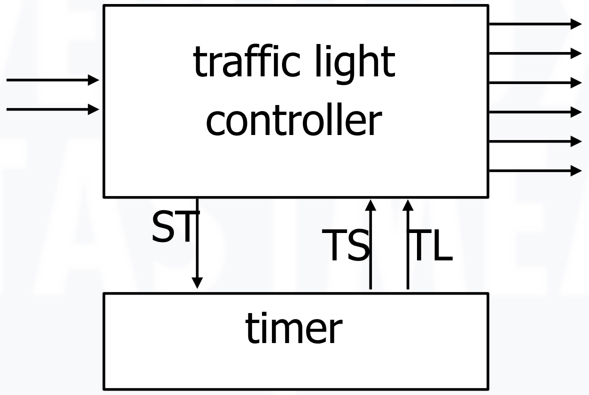
```
module string (clk, M, rst, Z0, Z1, Z2);
input clk, X, rst;
output Z0, Z1, Z2;

parameter S0 = [0,0,0];
parameter S1 = [0,0,1];
parameter S2 = [0,1,0];
parameter S3 = [0,1,1];
parameter S4 = [1,0,0];
parameter S5 = [1,0,1];
parameter S6 = [1,1,0];
parameter S7 = [1,1,1];

reg state[0:2];

assign Z0 = state[0];
assign Z1 = state[1];
assign Z2 = state[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

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





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;
```

```
parameter highwaygreen    = 6'b001100;
parameter highwayyellow   = 6'b010100;
parameter farmroadgreen   = 6'b100001;
parameter 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];
```

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

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 triggerred by clock edge

Timer for traffic light controller

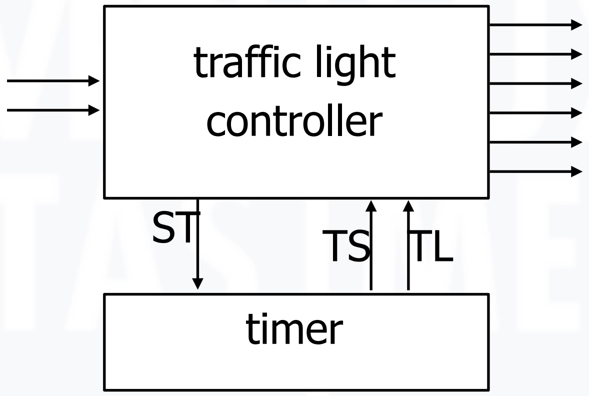
- Another FSM

```
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
```

Complete traffic light controller

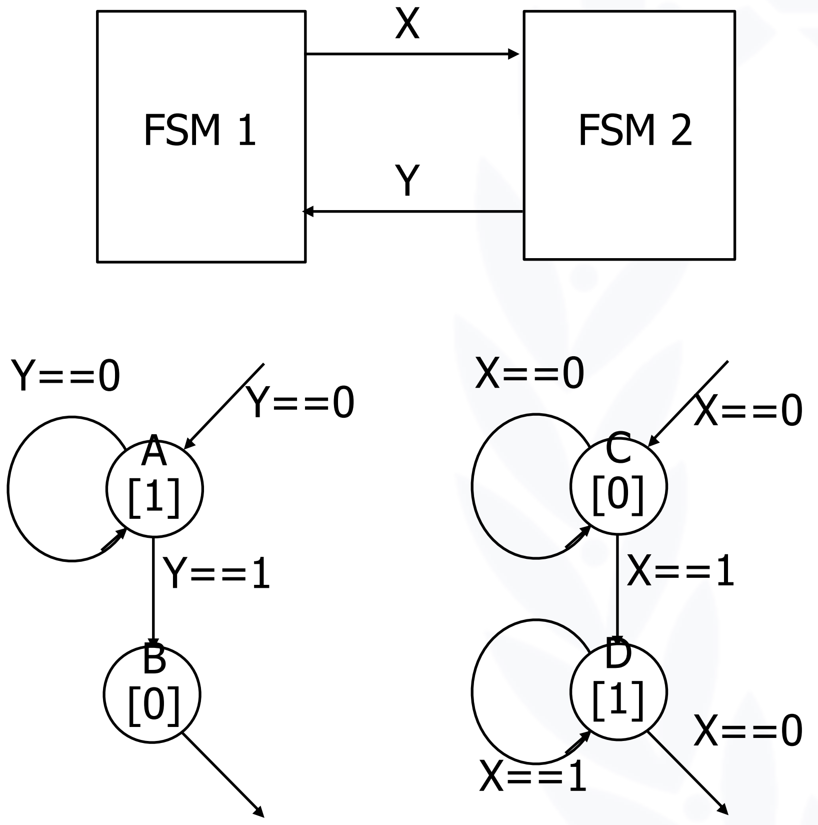
- Tying it all together (FSM + timer)
  - Structural Verilog (same as a schematic drawing)

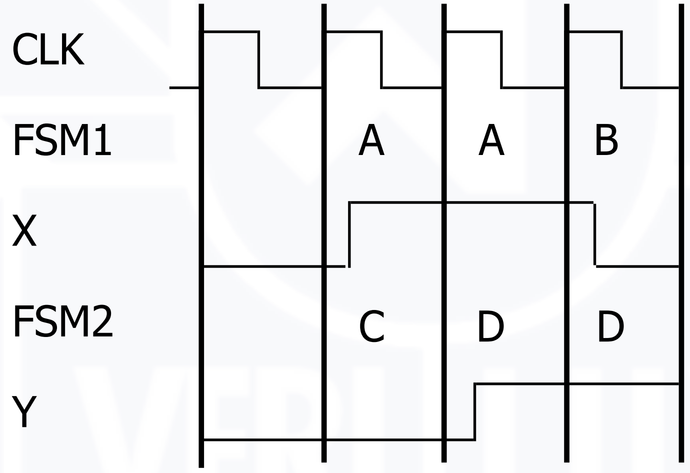
```
module main(HR, HY, HG, FR, FY, FG, reset, C, Clk);
 output HR, HY, HG, FR, FY, FG;
 input  reset, C, Clk;

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



Communicating finite state machines

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

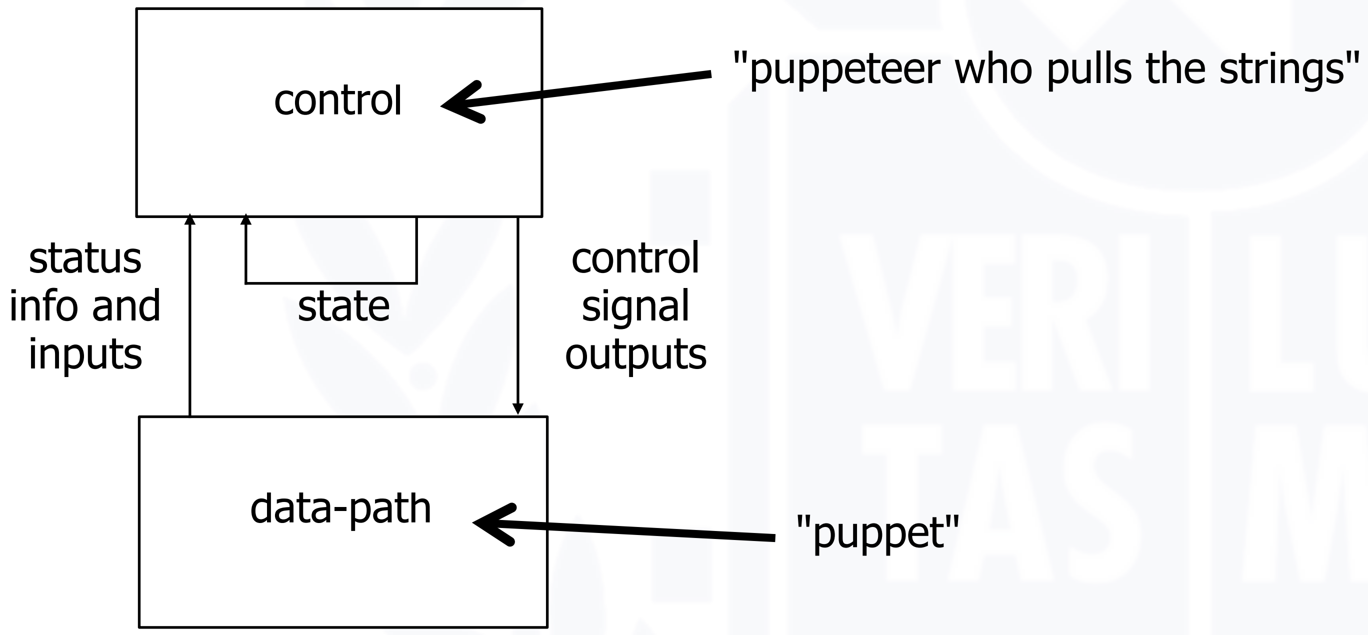




Machines advance in lock step
initial inputs/outputs: X = 0, Y = 0

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



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

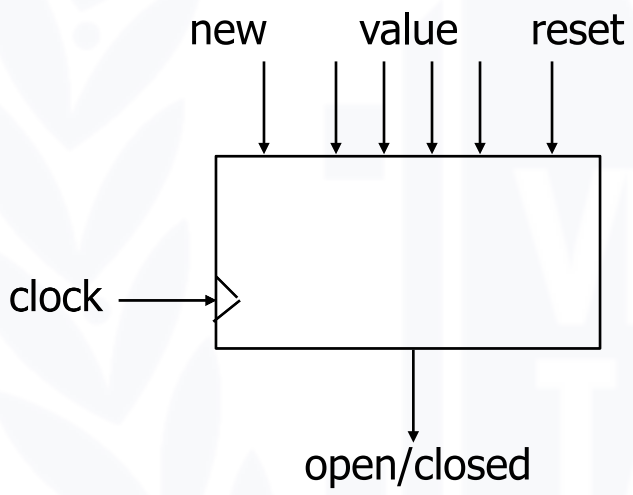
  - Open questions: how do you set the internal combination?
    - Stored in registers (how loaded?)
    - Hardwired via switches set by user

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);
}
```

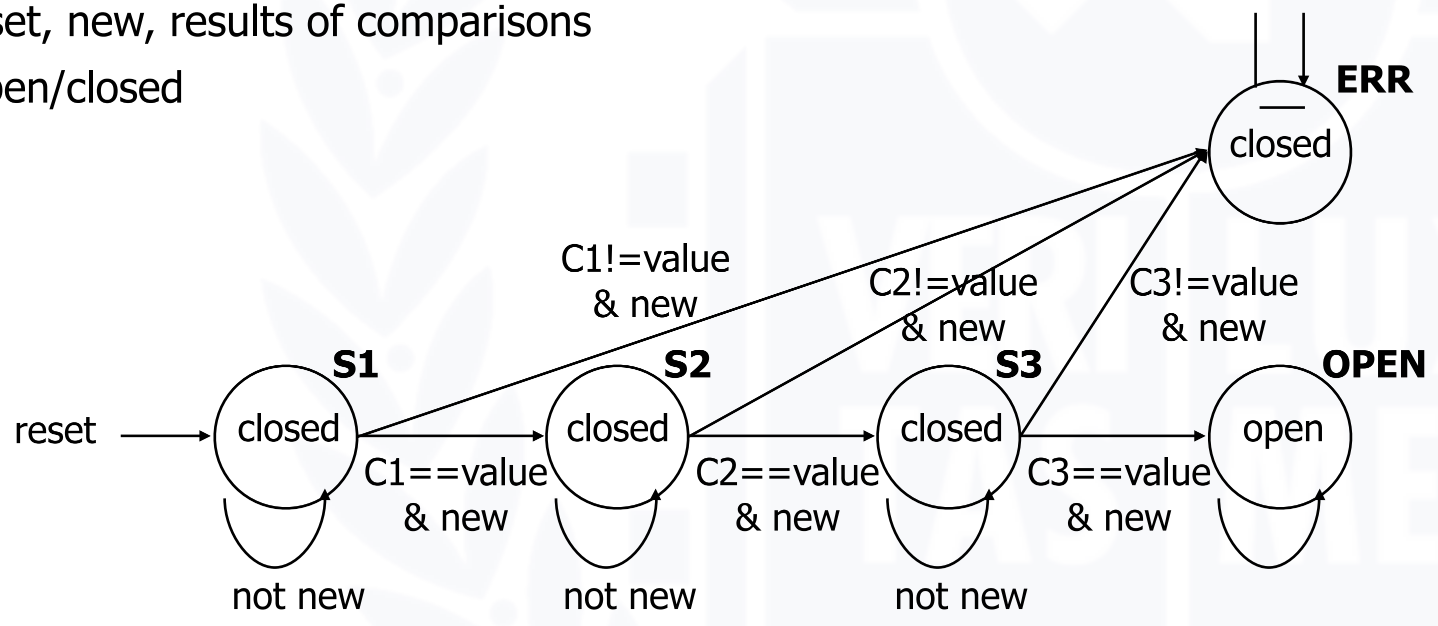
Determining details of the specification

- 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

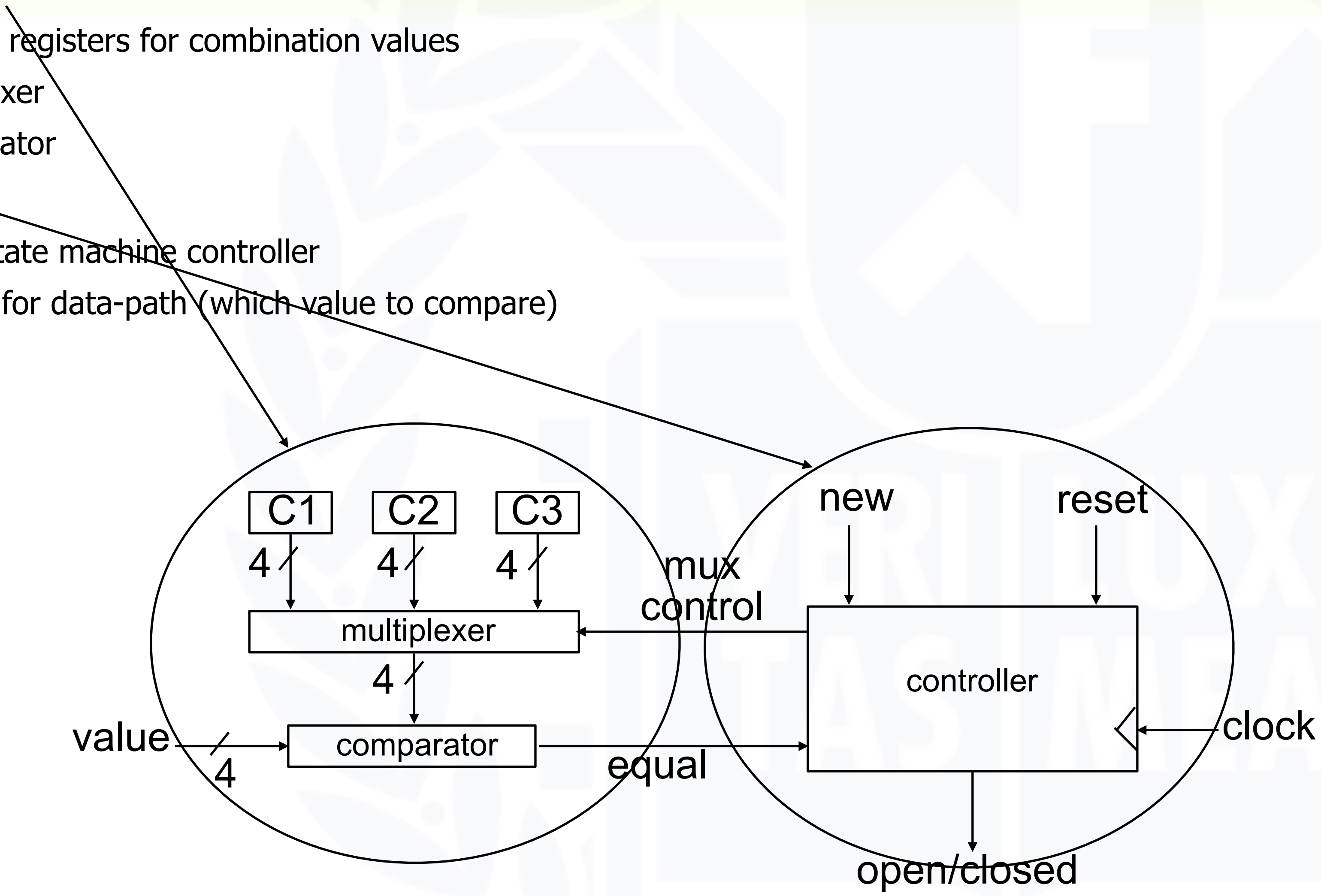
- States: 5 states
  - Represent point in execution of machine
  - Each state has outputs
- Transitions: 6 from state to state, 5 self transitions, 1 global
  - Changes of state occur when clock says its ok
  - Based on value of inputs
- Inputs: reset, new, results of comparisons
- Output: open/closed



Embedded Low-Power Laboratory

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)



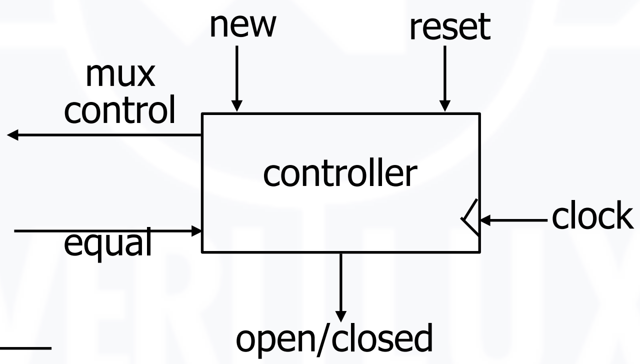
State table for combination lock

- 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
    - 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



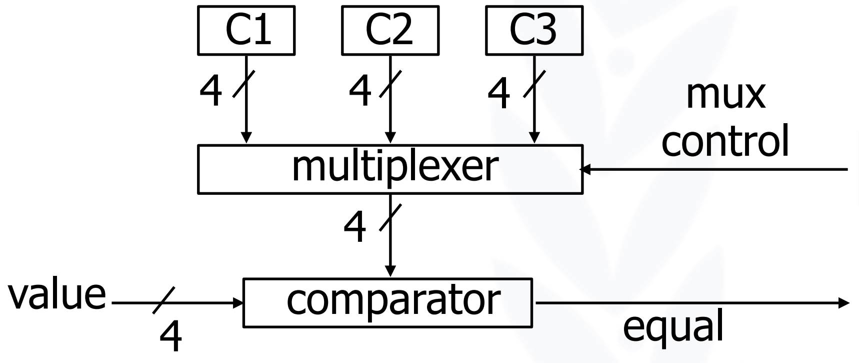
reset	new	equal	state	next state	mux	open/closed
1	–	–	–	0001	001	0
0	0	–	0001	0001	001	0
0	1	0	0001	0000	–	0
0	1	1	0001	0010	010	0
...						
0	1	1	0100	1000	–	1
...						

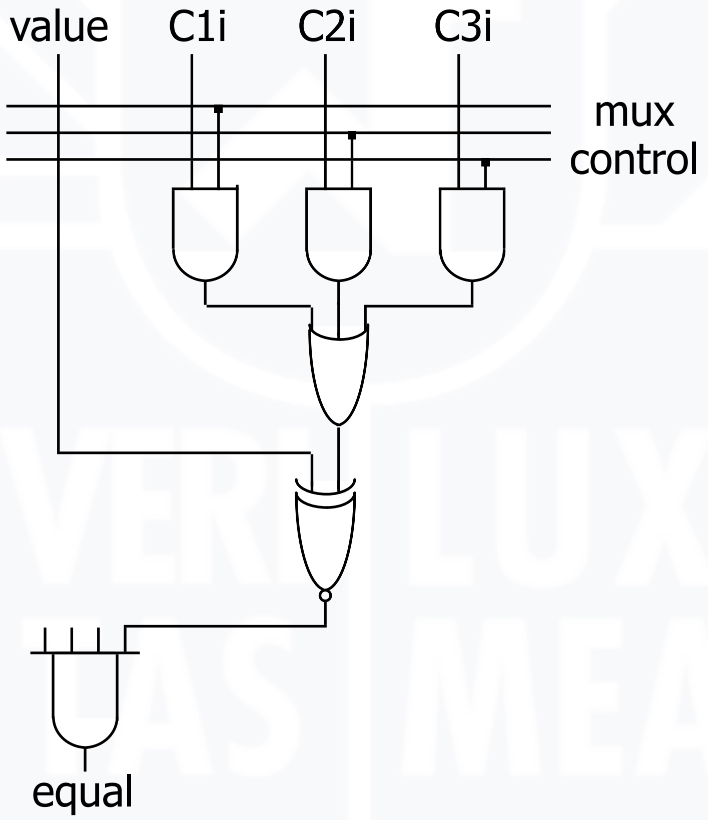
Mux is identical to last 3 bits of state
open/closed is identical to first bit of state
therefore, we do not even need to implement
FFs to hold state, just use outputs

Embedded Low-Power Laboratory

Data-path implementation for combination lock

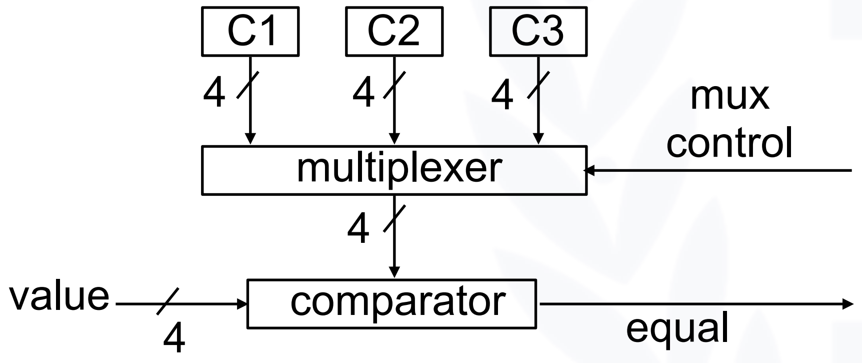
- Multiplexer
  - Easy to implement as combinational logic when few inputs
  - Logic can easily get too big for most PLDs

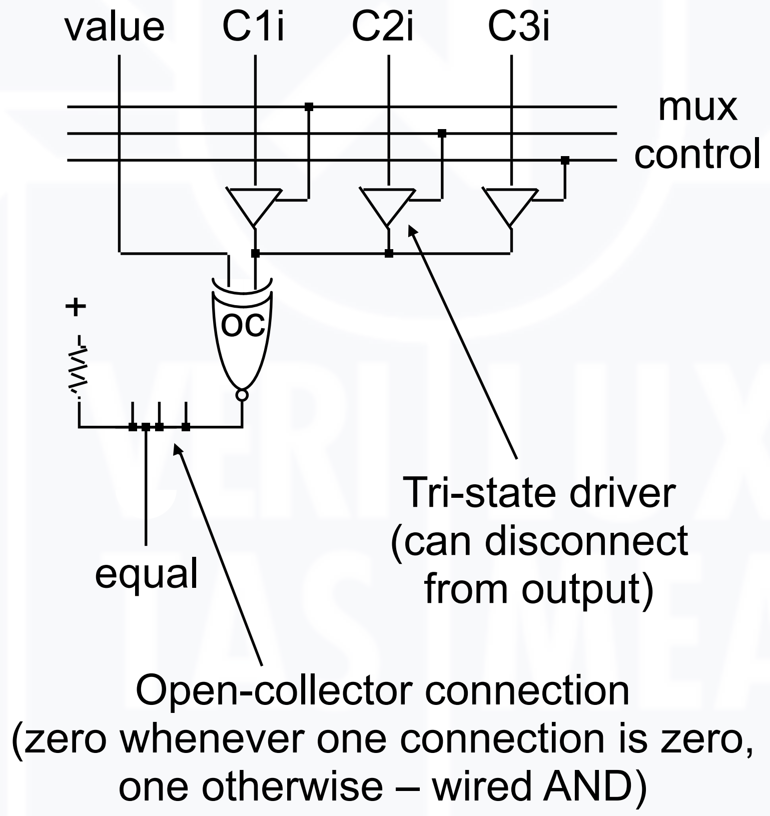




Data-path implementation (cont'd)

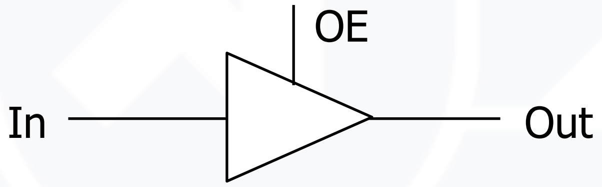
- Tri-state logic
  - Utilize a third output state: "no connection" or "float"
  - Connect outputs together as long as only one is "enabled"
  - Open-collector gates can only output 0, not 1
    - Can be used to implement logical AND with only wires




value
C1i
C2i
C3i
mux
control
+
OC
equal
Tri-state driver
(can disconnect
from output)
Open-collector connection
(zero whenever one connection is zero,
one otherwise – wired AND)

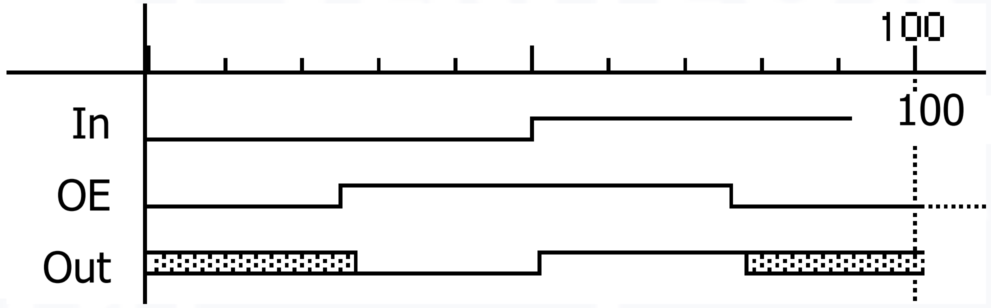
Tri-state gates

- 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)



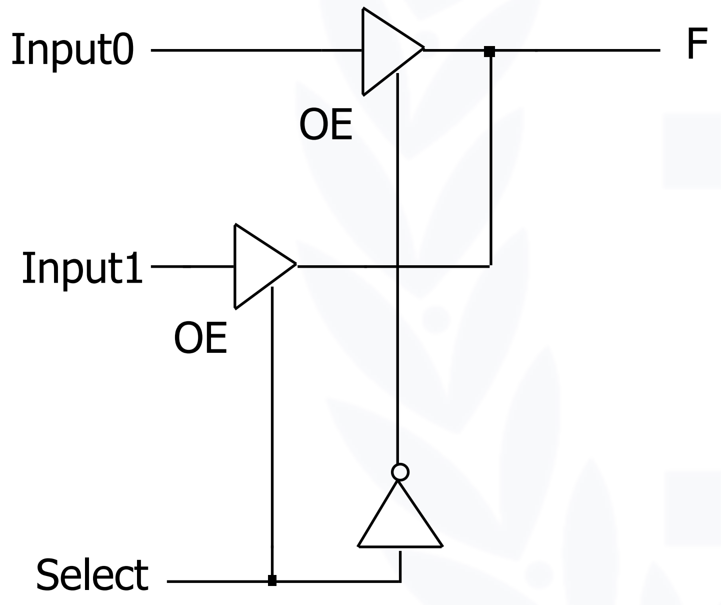
Non-inverting tri-state buffer

In	OE	Out
X	0	Z
0	1	0
1	1	1



Tri-state and multiplexing

- When using tri-state logic
  - 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)
  - 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

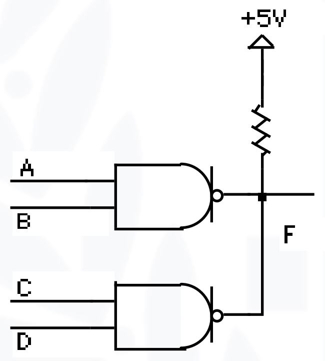
When Select is low
Input0 is connected to F

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

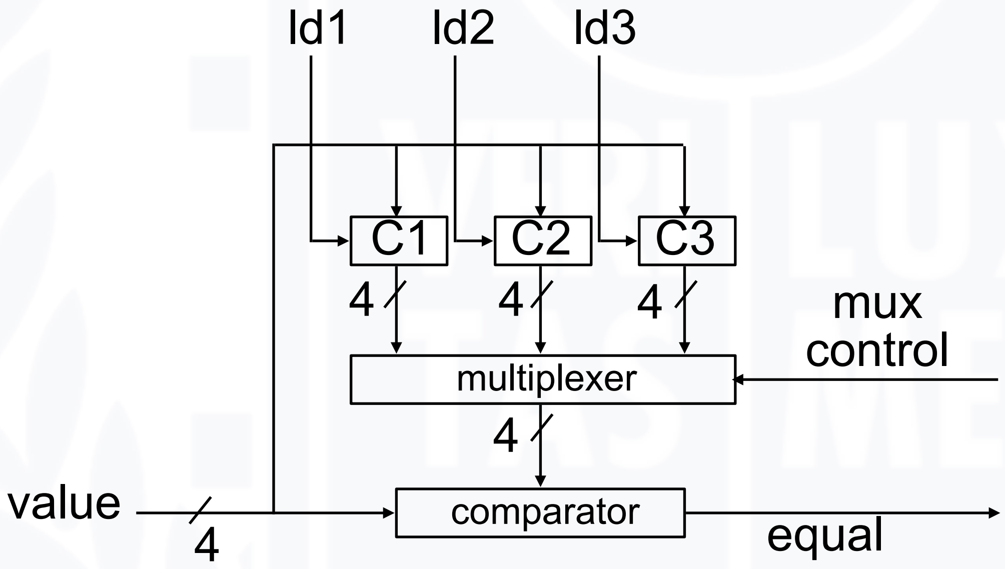
Open-collector
NAND gates



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
  - 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