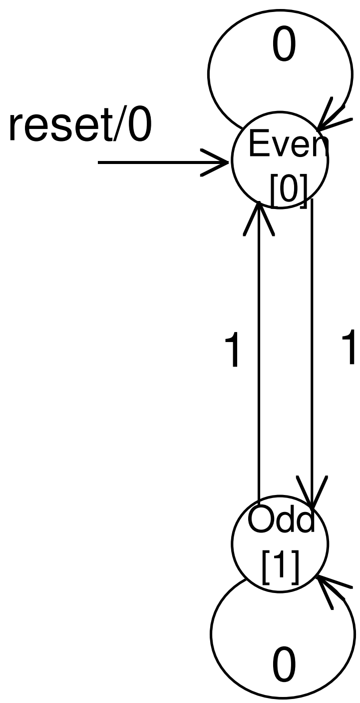
Sequential logic examples

- **Motivation**
  - counters are important sequential circuits where output = state
  - generalize to finite state machines where output = f(state, inputs) and the next state is also a function of state and inputs
  - FSMs are the decision making logic of digital designs
- **Finite state machine concept**
  - partitioning designs into datapath and control elements
  - when inputs are sampled and outputs asserted
- **Basic design approach: a 5-step design process**
- **State machine representations**
  - state diagram, hardware description language (ABEL-HDL)
- **Implementation examples and case studies**

Example: odd parity checker

- **Asserts output whenever its has seen an odd # of 1s**
  - input stream is sampled every clock cycle



state diagram

present state	input	next state	output
even	0	even	0
even	1	odd	0
odd	0	odd	1
odd	1	even	1

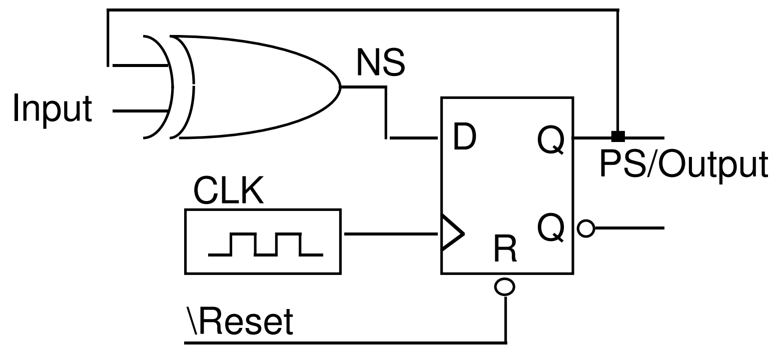
symbolic state transition table

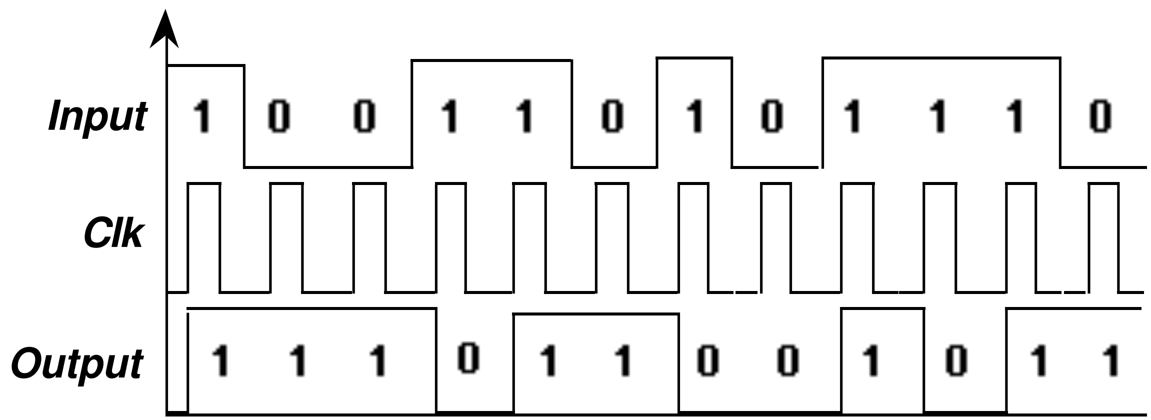
present state	input	next state	output
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

encoded state transition table

Example: odd parity checker

- **Next state and output functions**
  - NS = PS xor PI
  - OUT = PS



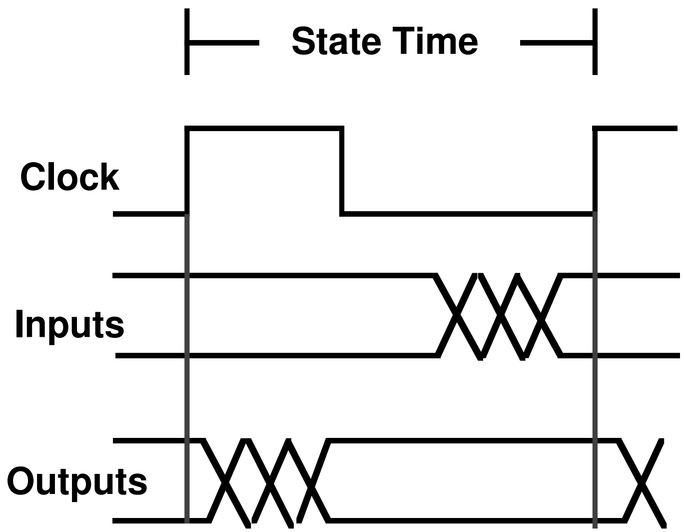


Timing in finite state machines

- **When are inputs sampled? next state computed? outputs asserted?**
- **State time**
  - time between clocking events
- **Clocking event causes state/outputs to transition, based on inputs**
  - in Moore machine
  - Mealy machine reacts (changes outputs) immediately to inputs
- **For set-up/hold time considerations**
  - inputs should be stable around clocking event
- **After propagation delay**
  - next state entered
  - outputs are stable

Timing in finite state machines (cont'd)

- **Positive edge-triggered synchronous system**



before rising edge:
outputs and next state being computed

on rising edge:
outputs and next state are latched into FFs

after propagation delay:
outputs and next state are stable
outputs may incur extra decoding delay

inputs can change after clock edge and hold time:
must settle into new values while allowing enough time for new outputs and state to be computed before next clock edge

FSM design in 5 easy steps

1. understand the statement of the specification

2. obtain an abstract specification of the FSM (e.g., state diagram)

3. perform a state minimization

4. perform a state assignment (or encoding)

5. implement the FSM

Finite state machine case studies

- **Mapping English language description to formal specifications**

- **Four case studies**
  - finite string pattern recognizer
  - complex counter with decision making
  - traffic light controller
  - digital combination lock
  - telephone keypad and display

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

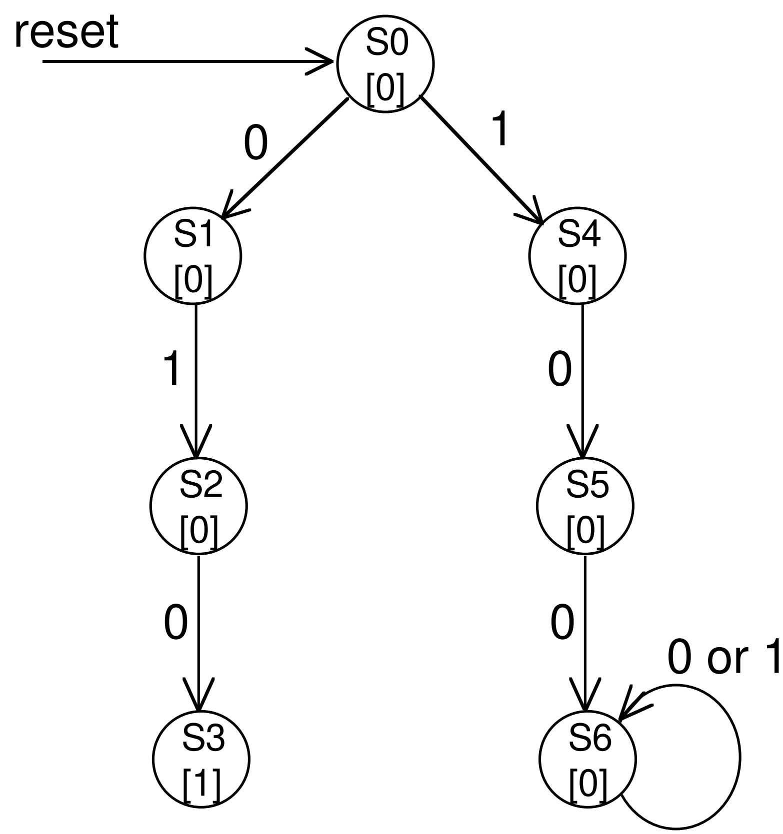
- **Step 1: understanding the problem statement**
  - sample input/output behavior:

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 …

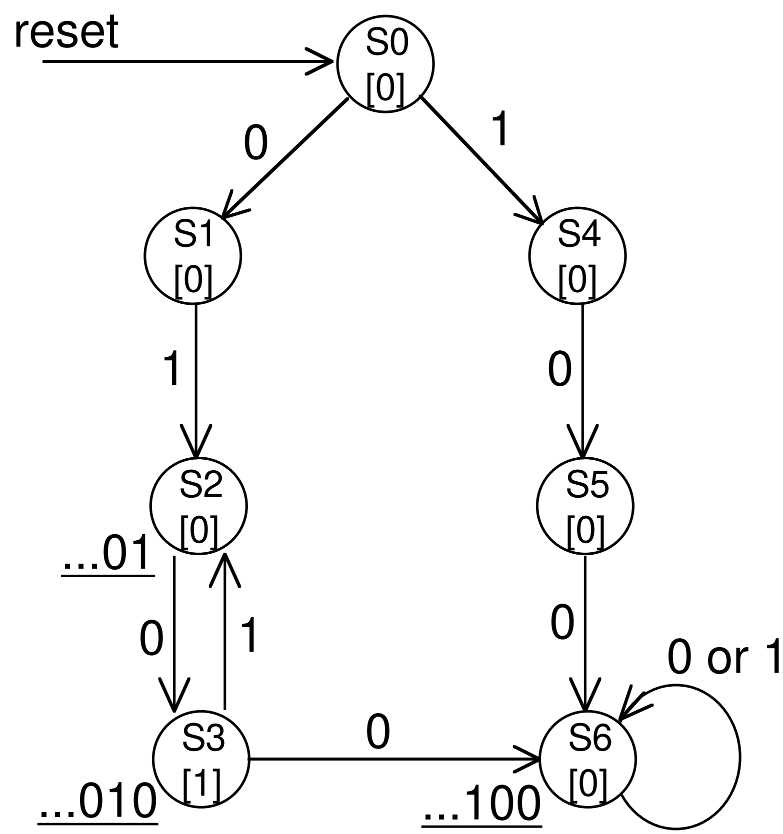
Finite string pattern recognizer (step 2)

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



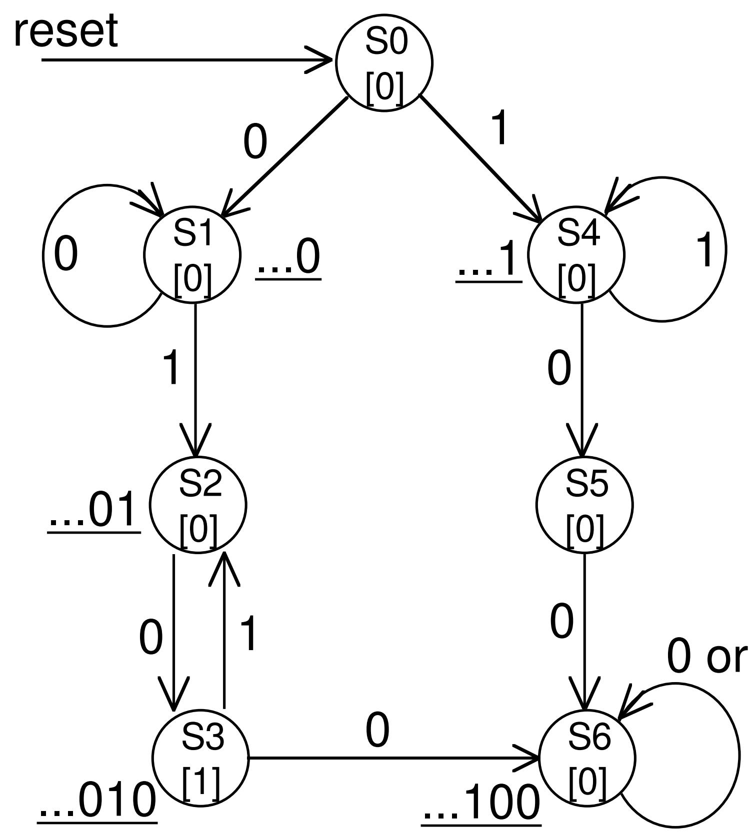
Finite string pattern recognizer (step 2, cont'd)

- **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)



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

- **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

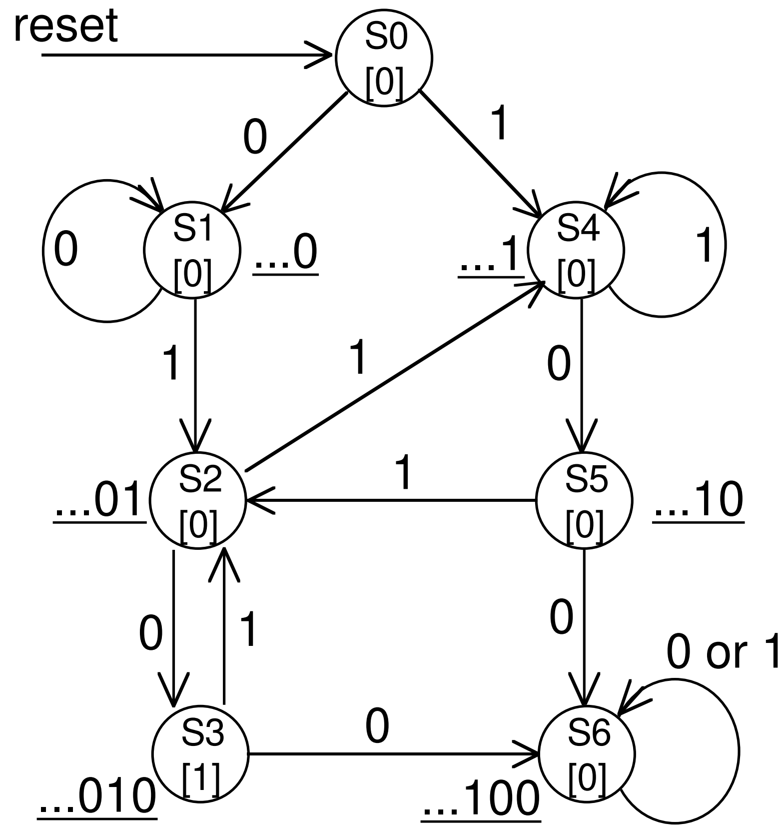


Finite string pattern recognizer (step 3)

- **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 4)

- **ABEL-HDL description including state assignment (or state encoding)**

```
MODULE string

 interface (clk, X, rst -> Q0..Q2, Z);

TITLE 'string recognizer for 010 and 100'

"Inputs
 clk, X, rst  pin;

"Outputs
 Q0..Q2        pin istype 'reg,buffer';
 Z             pin istype 'com';

"Aliases for state register and encoding
 SREG = [Q0..Q2];
 S0 = [0,0,0]; "reset state
 S1 = [0,0,1]; "strings ending in    ...0
 S2 = [0,1,0]; "strings ending in   ...01
 S3 = [0,1,1]; "strings ending in ...010
 S4 = [1,0,0]; "strings ending in    ...1
 S5 = [1,0,1]; "strings ending in   ...10
 S6 = [1,1,0]; "strings ending in ...100
```

```
EQUATIONS

 SREG.clk = clk;
 SREG.clr = rst;
 Z = (SREG.fb == S3);

STATE_DIAGRAM SREG

 state S0: if X then S4 else S1;
 state S1: if X then S2 else S1;
 state S2: if X then S4 else S3;
 state S3: if X then S2 else S6;
 state S4: if X then S4 else S5;
 state S5: if X then S2 else S6;
 state S6: if rst then S0 else S6;

END
```

Finite string pattern recognizer (step 5)

- **Derived equations**

```
Q0.D    = ( !rst & !X & !Q0.FB &  Q1.FB &  Q2.FB
          # !rst & !X &  Q0.FB & !Q1.FB
          # !rst &  X &                  !Q2.FB
          # !rst &       Q0.FB &         !Q2.FB ); " ISTYPE 'BUFFER'
Q0.C    = (  clk );

Q1.D    = ( !rst &  X & !Q0.FB &          Q2.FB
          # !rst & !X & !Q0.FB &  Q1.FB
          # !rst &       Q0.FB &  Q1.FB & !Q2.FB
          # !rst &       Q0.FB & !Q1.FB &  Q2.FB ); " ISTYPE 'BUFFER'
Q1.C    = (  clk );

Q2.D    = ( !rst & !X &         !Q1.FB & !Q2.FB
          # !rst & !X & !Q0.FB &         !Q2.FB
          # !rst & !X & !Q0.FB & !Q1.FB        ); " ISTYPE 'BUFFER'
Q2.C    = (  clk );

Z       = (             !Q0.FB &  Q1.FB &  Q2.FB );
```

Finite string pattern recognizer

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

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

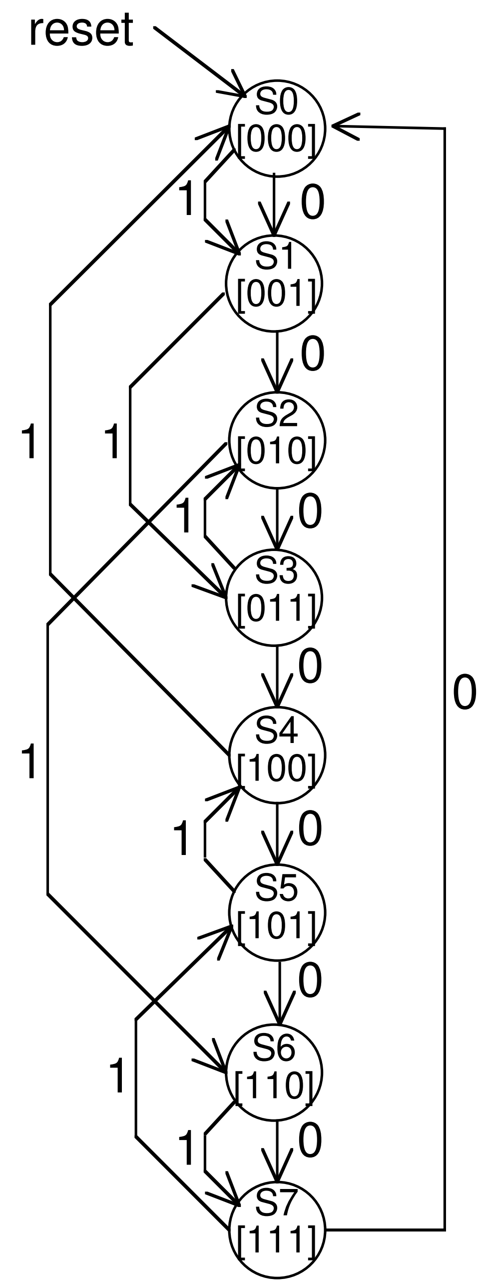
binary: 000, 001, 010, 011, 100, 101, 110, 111
Gray: 000, 001, 011, 010, 110, 111, 101, 100

- **Valid I/O behavior**

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 in HDL)

- **ABEL-HDL description including state encoding**

```
MODULE counter

 interface (clk, M, rst -> Z0..Z2);

TITLE 'binary and Gray code counter'

"Inputs
 clk, M, rst  pin;
"Outputs
 [Z0..Z2]     pin istype 'reg,buffer';

"Aliases for state register and encoding
 SREG = [Z0..Z2];
 S0 = [0,0,0];
 S1 = [0,0,1];
 S2 = [0,1,0];
 S3 = [0,1,1];
 S4 = [1,0,0];
 S5 = [1,0,1];
 S6 = [1,1,0];
 S7 = [1,1,1];
```

```
EQUATIONS

 SREG.clk = clk;
 SREG.clr = rst;

STATE_DIAGRAM SREG

 state S0: goto S1;
 state S1: if M then S3 else S2;
 state S2: if M then S6 else S3;
 state S3: if M then S2 else S4;
 state S4: if M then S0 else S5;
 state S5: if M then S4 else S6;
 state S6: goto S7;
 state S7: if M then S5 else S0;

END
```

Complex counter (realization)

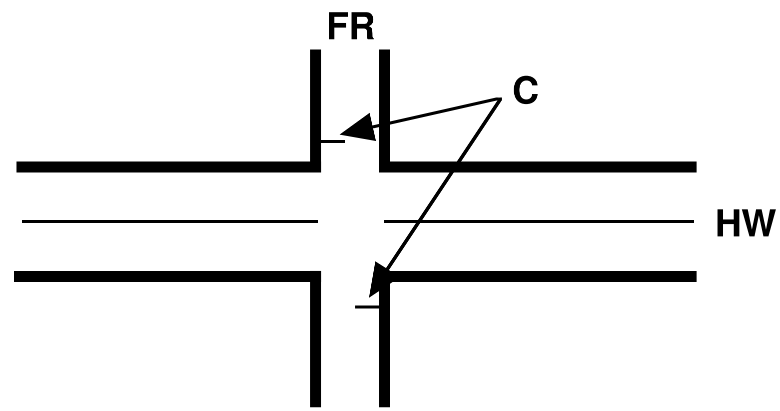
- **Derived equations**

```
Z0.D   = ( !rst &  M &  Z0.FB &          Z2.FB
         # !rst & !M &  Z0.FB &         !Z2.FB
         # !rst &       Z0.FB & !Z1.FB &  Z2.FB
         # !rst & !M & !Z0.FB &  Z1.FB &  Z2.FB
         # !rst &  M &           Z1.FB & !Z2.FB );
Z0.C   = (  clk );

Z1.D   = ( !rst &  M & !Z0.FB &          Z2.FB
         # !rst & !M &          !Z1.FB &  Z2.FB
         # !rst &                Z1.FB & !Z2.FB );
Z1.C   = (  clk );

Z2.D   = ( !rst &  M &  Z0.FB &  Z1.FB
         # !rst &  M & !Z0.FB & !Z1.FB
         # !rst & !M &                  !Z2.FB );
Z2.C   = (  clk );
```

Traffic light controller

- **Highway/farmroad intersection with car sensors for farmroad**
  - busy highway, little used farmroad
  - detectors C sense the presence of cars waiting on the farmroad
  - if no car on farmroad (FR), light remains green for highway (HW)
  - If car on FR, HW lights go from G to Y to R, then FR lights can go to green
  - FR lights stay green only as long as there are cars on FR but no longer than a pre-defined interval
  - when no more cars on FR, FR lights go from G to Y to R, then HW lights can go to green.
  - even if FR vehicles are waiting, HW lights stay green for a pre-defined minimum time

assume you have an interval timer that generates:
a short time pulse (TS) and
a long time pulse (TL),
in response to a set (ST) signal.
TS is to be used for timing yellow lights and
TL for green lights



Traffic light controller (inputs/outputs/states)

- **Tabulation of inputs and outputs**

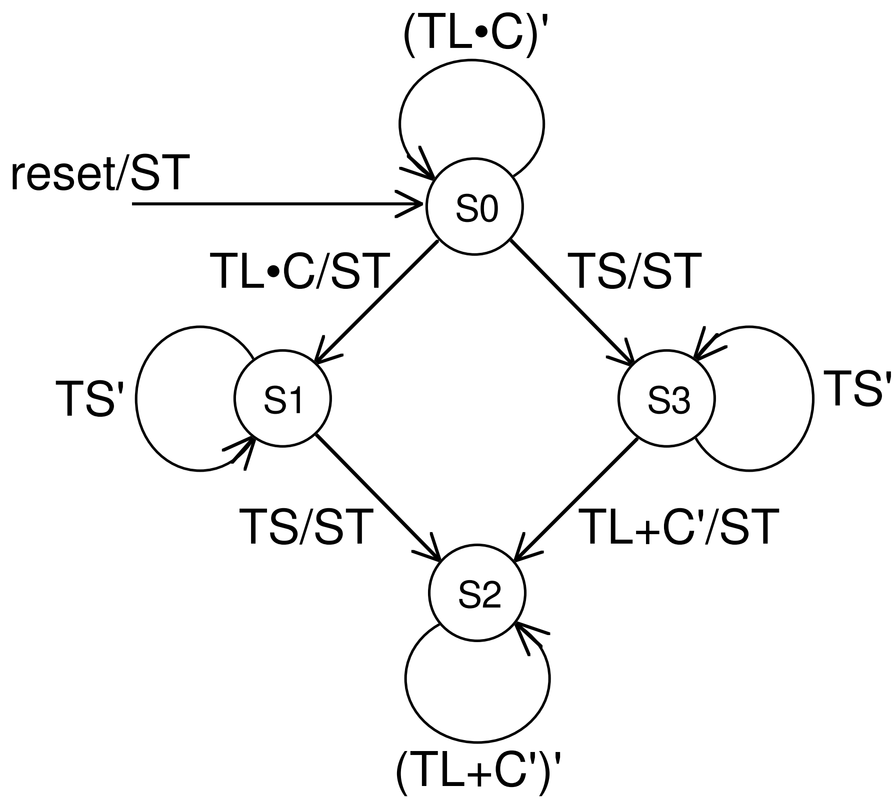
inputs	description	outputs	description
reset	place FSM in initial state	HG, HY, HR	assert green/yellow/red highway ligths
C	detect vehicle on the farm road	FG, FY, FR	assert green/yellow/red highway ligths
TS	short time interval expired	ST	start timing a short or long interval
TL	long time interval expired		

TS asserted three cycles after ST and until next ST
TL asserted seven cycles after ST and until next ST

- **Tabulation of unique states**
  - some light configurations imply others

state	description
S0	highway green (farm road red)
S1	highway yellow (farm road red)
S2	farm road green (highway red)
S3	farm road yellow (highway red)

Traffic light controller (state diagram)



S0: HG

S1: HY

S2: FG

S3: FY

State assignment or state encoding

- **Process of choosing a code for each state of the FSM**
  - a state code is a bit vector with anywhere from log n to n bits, where n is the number of states
- **Judicious state assignment can improve the speed and size of the combinational logic needed for the FSM**
  - tight encoding vs. loose encoding
  - min-bit-width (log n) vs. one-hot codes (n)
  - less FFs vs. simpler logic
- **Using output signals as part of the state vector can help minimize the number of state FFs needed**
- **Reset state – how is it reached?**
  - reset input
  - self-starting
  - set/reset FFs available

Traffic light controller (state table/state encodings)

- **Generate state table with symbolic states**
- **Consider state assignments**

output encoding – similar problem to state assignment
(Green = 00, Yellow = 01, Red = 10)

Inputs C	TL	TS	Present State	Next State	Outputs ST	H	F
0	–	–	HG	HG	0	Green	Red
–	0	–	HG	HG	0	Green	Red
1	1	–	HG	HY	1	Green	Red
–	–	0	HY	HY	0	Yellow	Red
–	–	1	HY	FG	1	Yellow	Red
1	0	–	FG	FG	0	Red	Green
0	–	–	FG	FY	1	Red	Green
–	1	–	FG	FY	1	Red	Green
–	–	0	FY	FY	0	Red	Yellow
–	–	1	FY	HG	1	Red	Yellow

SA1:	HG = 00	HY = 01	FG = 11	FY = 10	
SA2:	HG = 00	HY = 10	FG = 01	FY = 11	
SA3:	HG = 0001	HY = 0010	FG = 0100	FY = 1000	(one-hot)

Traffic light controller (realizations of different state assignments)

State assignment 1:

NS1 = C•TL'•PS1•PS0 + TS•PS1'•PS0 + TS•PS1•PS0' + C'•PS1•PS0 + TL•PS1•PS0
NS0 = C•TL•PS1'•PS0' + C•TL'•PS1•PS0 + PS1'•PS0
ST = C•TL•PS1'•PS0' + TS•PS1'•PS0 + TS•PS1•PS0' + C'•PS1•PS0 + TL•PS1•PS0
H1 = PS1 H0 = PS1'•PS0
F1 = PS1' F0 = PS1•PS0'

State assignment 2:

NS1 = C•TL•PS1' + TS'•PS1 + C'•PS1'•PS0
NS0 = TS•PS1•PS0' + PS1'•PS0 + TS'•PS1•PS0
ST = C•TL•PS1' + C'•PS1'•PS0 + TS•PS1
H1 = PS0 H0 = PS1•PS0'
F1 = PS0' F0 = PS1•PS0

State assignment 3:

NS3 = C'•PS2 + TL•PS2 + TS'•PS3
NS2 = TS•PS1 + C•TL'•PS2
NS1 = C•TL•PS0 + TS'•PS1
NS0 = C'•PS0 + TL'•PS0 + TS•PS3
ST = C•TL•PS0 + TS•PS1 + C'•PS2 + TL•PS2 + TS•PS3
H1 = PS3 + PS2 H0 = PS1
F1 = PS1 + PS0 F0 = PS3

Traffic light controller (HDL description)

```
module traffic

 interface (clk, C, reset, TS, TL -> Q0, Q1, HG, HY, HR, FG, FY, FR, ST);

title 'traffic light controller'

"Inputs
 clk, C, rst, TS, TL       pin;
"Outputs (registered)
 Q0, Q1                     pin istype 'reg,buffer';
"Outputs (combinational)
 HG, HY, HR, FG, FY, FR, ST   pin istype 'com';

"Aliases for state register
 SREG = [Q0,Q1];
 S0 = [0,0];
 S1 = [0,1];
 S2 = [1,0];
 S3 = [1,1];

EQUATIONS

 SREG.clk = clk;
 HG = (!Q0 & !Q1);
 HY = (!Q0 &  Q1);
 HR = ( Q0 & !Q1) # ( Q0 &  Q1);
 FG = ( Q0 & !Q1);
 FY = ( Q0 &  Q1);
 FR = (!Q0 & !Q1) # (!Q0 &  Q1);
```

combined Moore and Mealy behavior in ABEL

```
STATE_DIAGRAM SREG

 state S0: if rst then S0 with ST = 1
                 else if (TL & C)  then S1 with ST = 1
                                   else S0 with ST = 0;
 state S1: if rst then S0 with ST = 0
                 else if TS        then S2 with ST = 1
                                   else S1 with ST = 0;
 state S2: if rst then S0 with ST = 0
                 else if (TL # !C) then S3 with ST = 1
                                   else S2 with ST = 0;
 state S3: if rst then S0 with ST = 0
                 else if TS        then S0 with ST = 1
                                   else S3 with ST = 0;

END
```

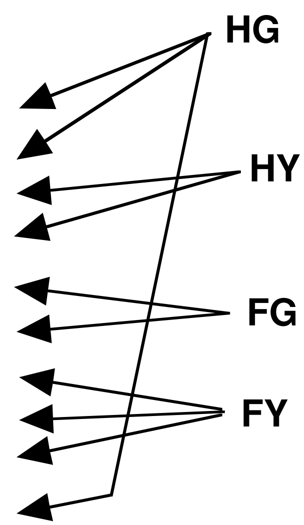
Choices for state encoding

- **With n state bits for m states there are $2^n! / (2^n - m)!$ [$\log n \leq m < n$]**
  - 2^n codes possible for the 1st state
  - 2^n-1 for the 2nd
  - 2^n-2 for the 3rd, ...
  - a huge number of combinations even for small values of n and m
    - 4 states in 2 state bits yields 24 possible state assignments
    - 4 states in 3 state bits yields 1,680 possible state assignments
    - 4 states in 4 state bits yields 43,680 possible state assignments
- **Possible strategies**
  - sequential – just number states in the order they appear in state table
  - gray-code – try to give adjacent states (with an arc between them) codes that differ in only one bit position
  - random – pick random codes
  - one-hot – use as many state bits as there are states (bit=1 implies state)
  - output – use outputs to help encode states
  - no guarantee of optimality – another intractable problem
    - no known optimal algorithms, just heuristics

Outputs as part of state vector

- **Reuse outputs as state bits**
  - use outputs to help distinguish states
  - why create new functions for state bits when output can serve as well in a synch-Mealy realization common in PLDs?

Inputs			Present State	Next State	Outputs		
C	TL	TS			ST	H	F
0	–	–	HG	HG	0	00	10
–	0	–	HG	HG	0	00	10
1	1	–	HG	HY	1	00	10
–	–	0	HY	HY	0	01	10
–	–	1	HY	FG	1	01	10
1	0	–	FG	FG	0	10	00
0	–	–	FG	FY	1	10	00
–	1	–	FG	FY	1	10	00
–	–	0	FY	FY	0	10	01
–	–	1	FY	HG	1	10	01



HG = ST'•H1'•H0'•F1•F0' + ST•H1•H0'•F1'•F0
HY = ST•H1'•H0'•F1•F0' + ST'•H1'•H0•F1•F0'
FG = ST•H1'•H0•F1•F0' + ST'•H1•H0'•F1'•F0'
HY = ST•H1•H0'•F1'•F0' + ST'•H1•H0'•F1'•F0

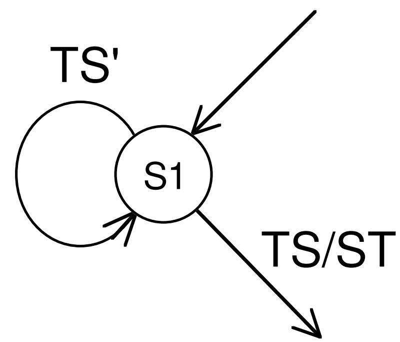
Output patterns are unique to states, we do not need ANY state bits – implement 5 functions (one for each output) instead of 7 (outputs plus 2 state bits)

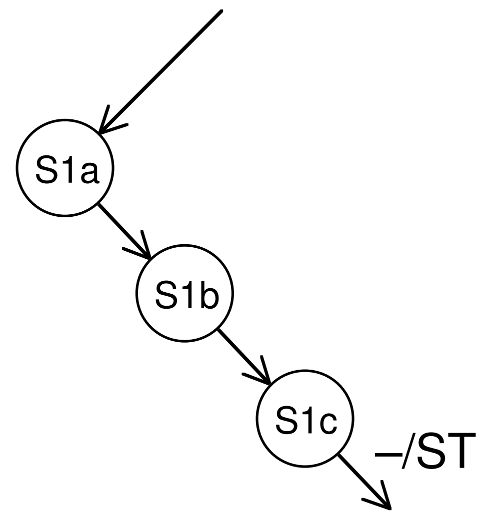
One-hot state assignment

- **n states encoded using n flipflops**
  - easiest to design/debug/extend (bit per state)
  - HG = 0001 HY = 0010 FG = 0100 FY = 1000
- **Small logic functions**
  - each state function only requires predecessor states' bits as input
  - complexity of functions directly related to complexity of state diagram
  - one product term per incoming arc
  - very common in PLDs
- **Impractical for large machines**
  - too many states require too many flip-flops
  - decompose state machines into smaller pieces that can be one-hot encoded
    - n+m states for two machines (n and m states, respectively) versus n*m states for a single one

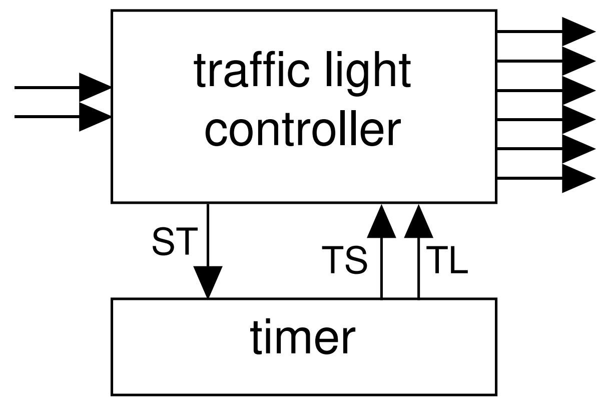
Traffic light controller as two communicating FSMs

- **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

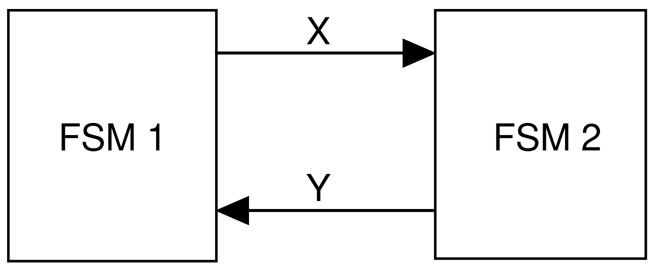


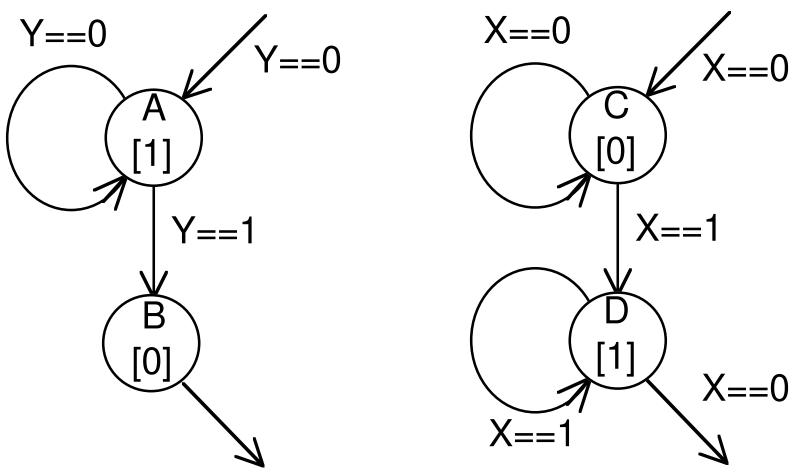


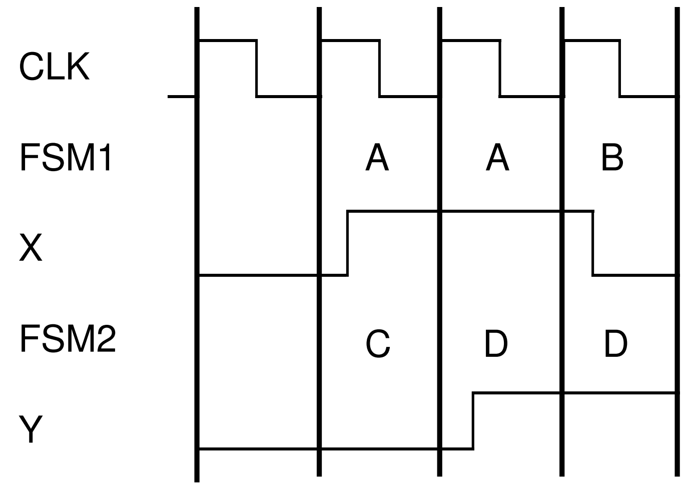
- **By factoring out timer**
  - greatly reduce number of states
    - 4 instead of 20
  - counter only requires seven or eight states
    - 12 total instead of 20



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



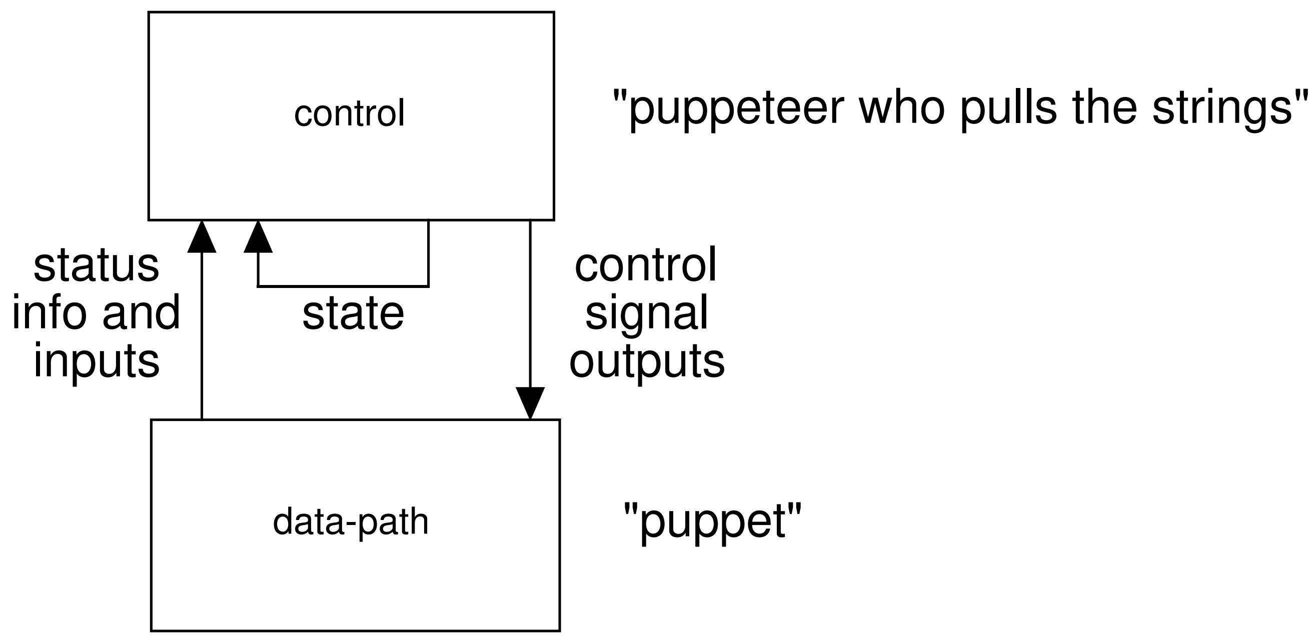




machines advance in lock step
initial inputs/outputs: X = 0, Y = 0

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



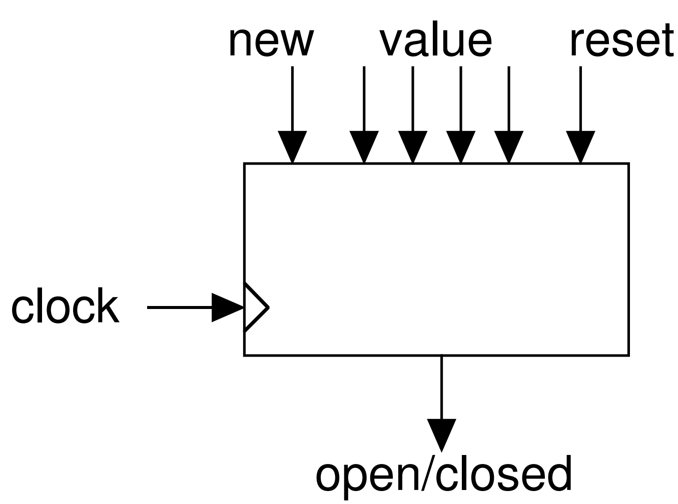
- **Door combination lock:**
  - punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
  - inputs: sequence of input values, reset
  - outputs: door open/close
  - memory: must remember combination or always have it available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

Implementation in software

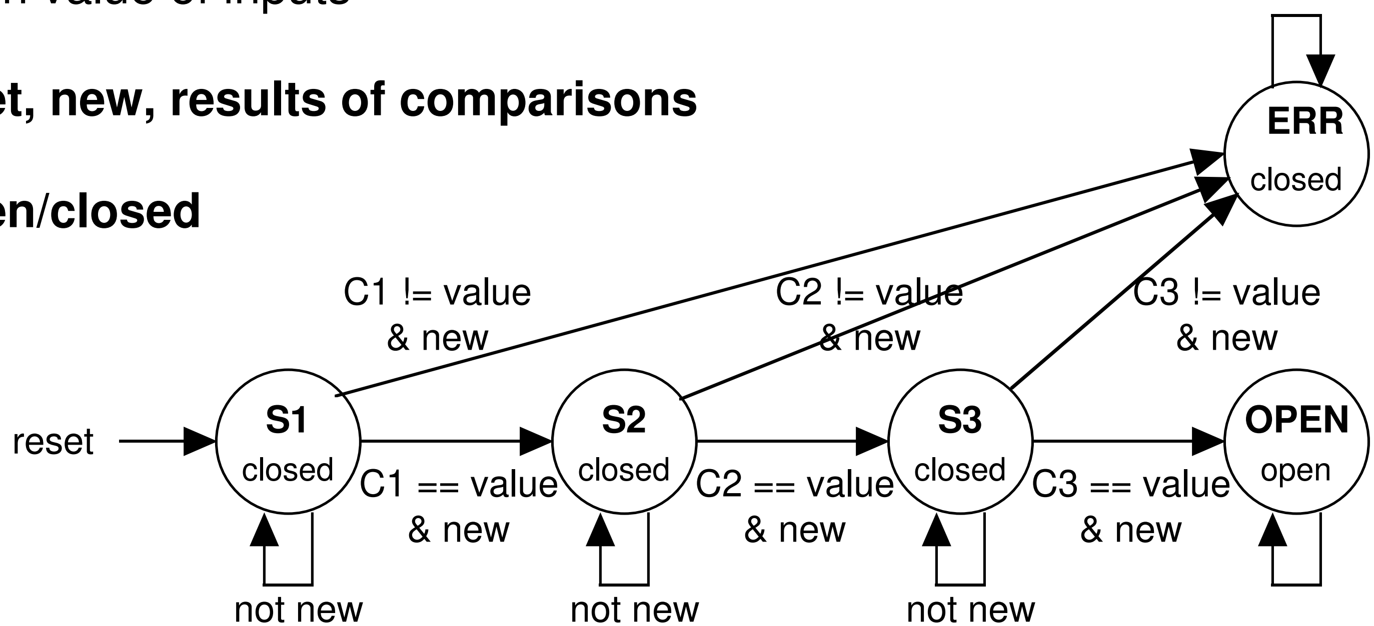
```
integer combination_lock ( ) {
	integer v1, v2, v3;
	integer error = 0;
	static c[3] = 3, 4, 2;

	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?

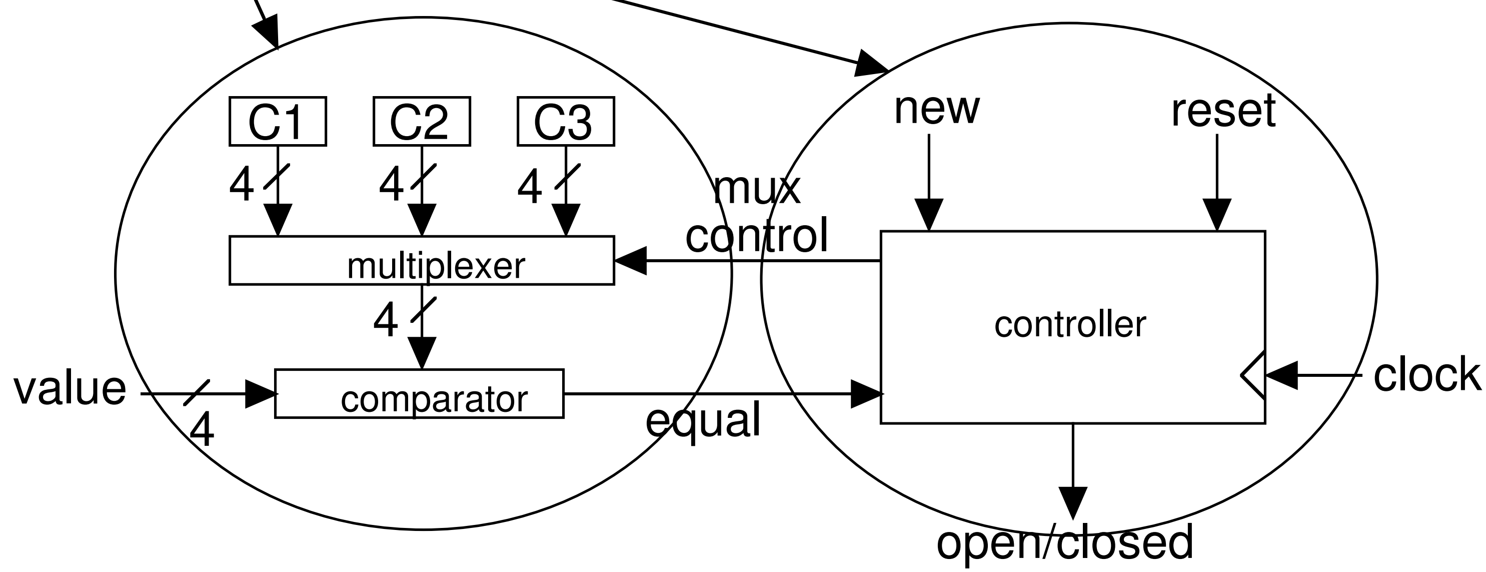


- **States: 5 states**
  - represent point in execution of machine
  - each state has outputs
- **Transitions: 6 from state to state, 5 self transitions, 1 global**
  - changes of state occur when clock says its ok
  - based on value of inputs
- **Inputs: reset, new, results of comparisons**
- **Output: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



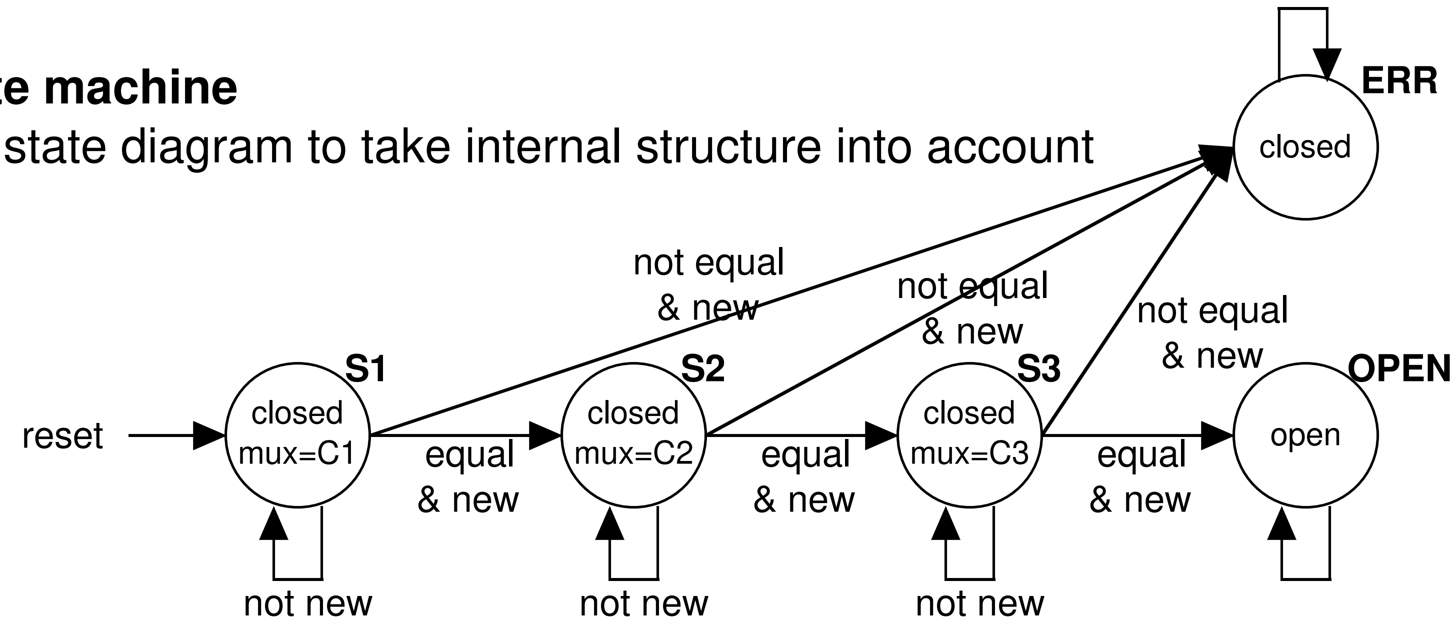
Data-path and control structure

- **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

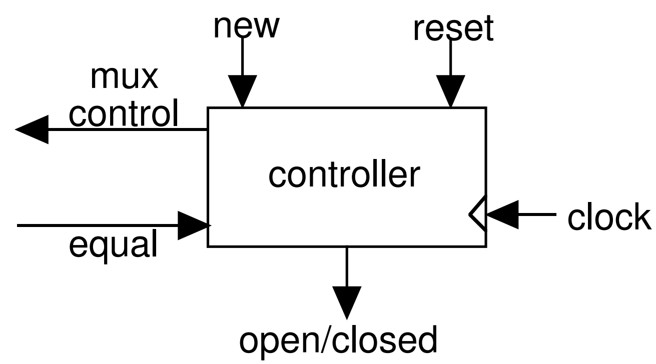


  - state table ready for encoding

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
    - 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

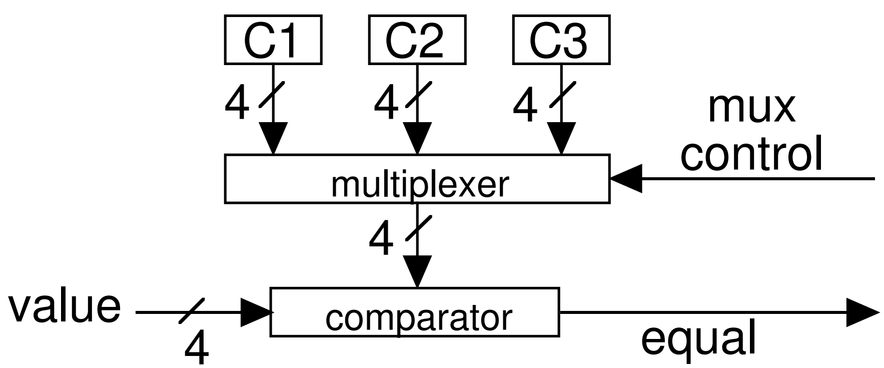


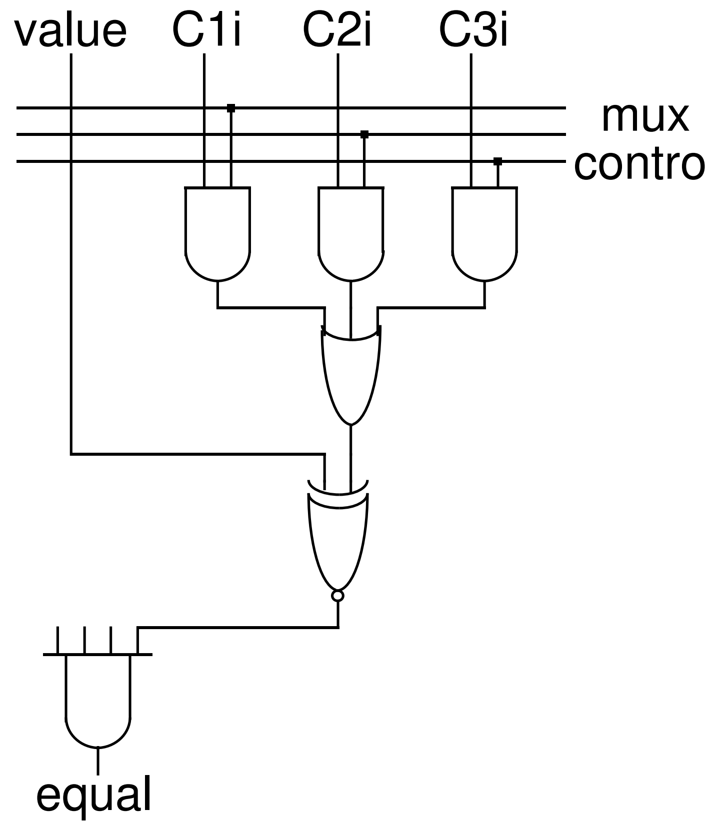
reset	new	equal	state	next state	mux	open/closed
1	–	–	–	0001	001	0
0	0	–	0001	0001	001	0
0	1	0	0001	0000	–	0
0	1	1	0001	0010	010	0
...						
0	1	1	0100	1000	–	1
...						

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

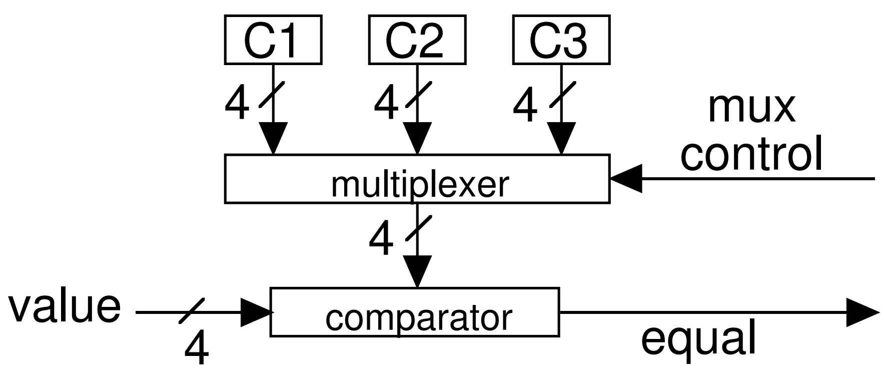
- **Multiplexer**
  - easy to implement in one PLD block per bit
  - one more block needed for equal AND gate

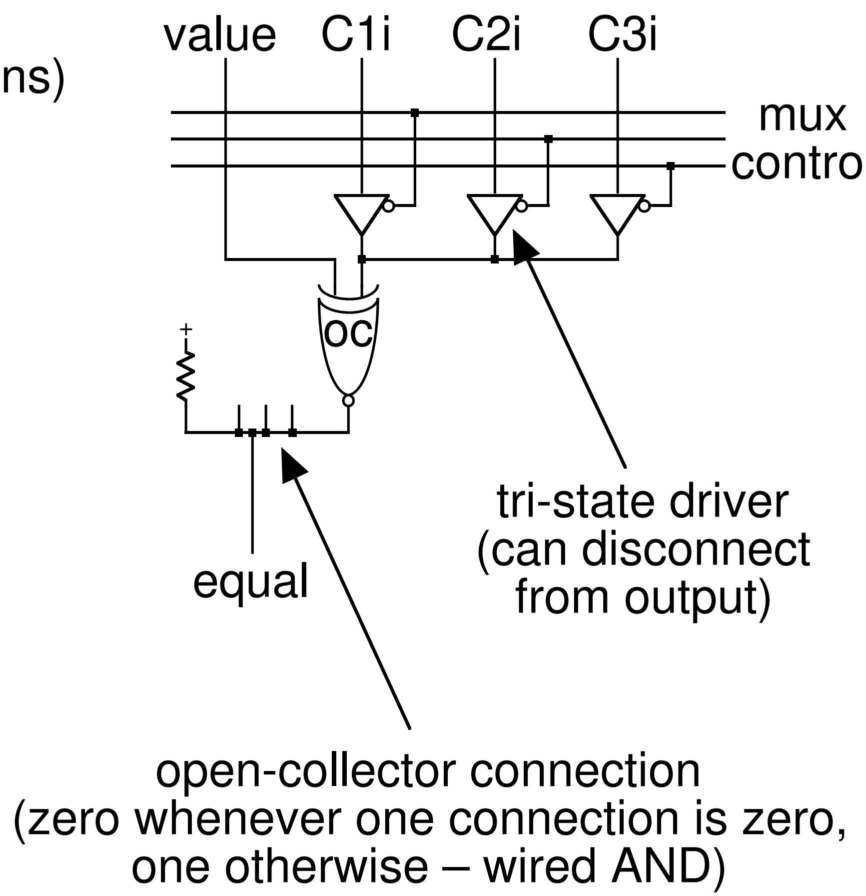




Data-path implementation for combination lock (cont'd)

- **Tri-state logic**
  - one tri-state driver per PLD block (need 3 per bit)
  - better to use between PLDs
  - no OC drivers inside PLDs (only at pins)





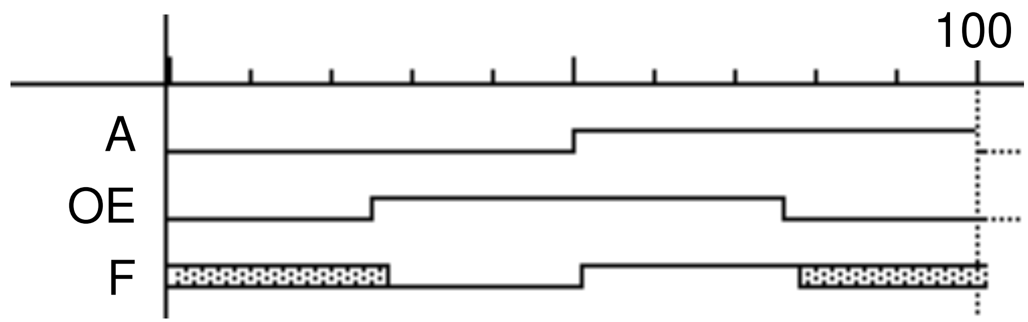
Tri-state gates

- **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 <u>disconnected</u> 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)

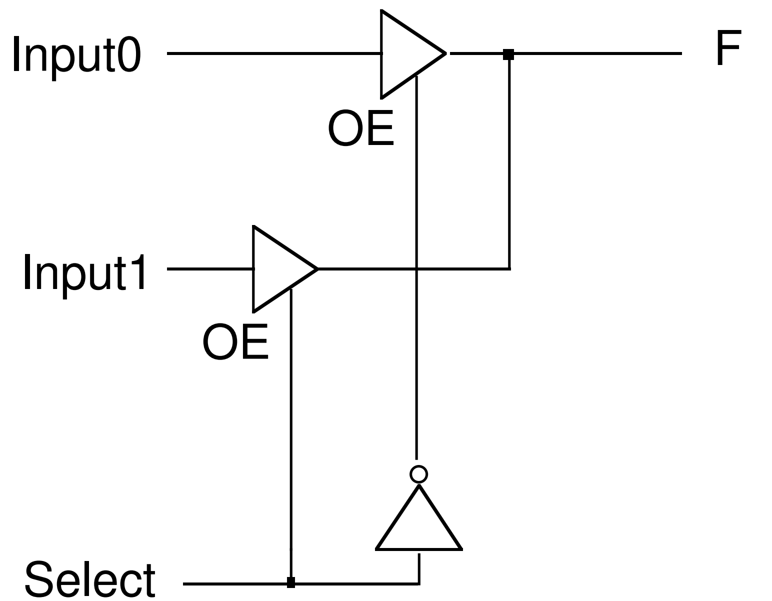
non-inverting tri-state buffer

A	OE	F
X	0	Z
0	1	0
1	1	1



- **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

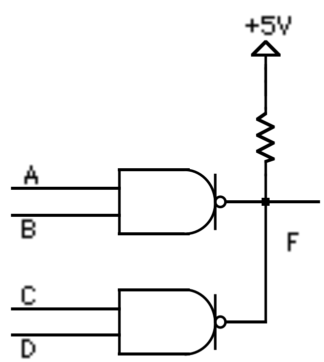
when Select is low
Input0 is connected to F

this is essentially a 2:1 mux

- **Open collector: 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 (HDL description)

```
module lock

 interface (clk, rst, new, C1_1..C1_4, C2_1..C2_4, C3_1..C3_4, value1..value4 -> out);

title 'digital combination lock'

"Inputs
 clk, rst, new  pin;
 C1_1..C1_4     pin;
 C2_1..C2_4     pin;
 C3_1..C3_4     pin;
 value1..value4 pin;

"Outputs
 out            pin  istype 'reg,buffer';

"Internal signals
 mux1..mux3     node istype 'reg,buffer';
 equal          node;

"Alias for state register and encoding
 SREG   = [mux1..mux3,out];
 S1     = [1,0,0,0];
 S2     = [0,1,0,0];
 S3     = [0,0,1,0];
 ERR    = [0,0,0,0];
 OPEN   = [0,0,0,1];
 cntrl1 = [mux1, mux1, mux1, mux1];
 cntrl2 = [mux2, mux2, mux2, mux2];
 cntrl3 = [mux3, mux3, mux3, mux3];
```

```
EQUATIONS

 SREG.clk = clk;
 mux1.set = rst;
 mux2.clr = rst;
 mux3.clr = rst;
 out.clr  = rst;

 equal = ([value1..value4] ==
                     ( (cntrl1 & [C1_1..C1_4])
                     # (cntrl2 & [C2_1..C2_4])
                     # (cntrl3 & [C3_1..C3_4]) ) );

STATE_DIAGRAM SREG

 state S1:   if !new then S1
               else if equal then S2   else ERR;
 state S2:   if !new then S2
               else if equal then S3   else ERR;
 state S3:   if !new then S3
               else if equal then OPEN else ERR;
 state ERR:  goto ERR;
 state OPEN: goto OPEN;

END
```

Digital combination lock (PLD implementation)

- **Inputs**
  - clk, rst, new
  - C1_1..C1_4, C2_1..C2_4, C3_1..C3_4
  - value1..value4
- **Combinational outputs**
  - equal
- **Registered outputs**
  - mux1..mux3
  - out
- **Fitting**
  - at least one block for each output – 5
  - equal will probably require 3 blocks (one per bit) plus 1 (for AND gate) – 4
  - needs 19 inputs and 1 output

Digital combination lock (logic)

- **ABEL output**

P-Terms	Fan-in	Fan-out	Type	Name (attributes)
2/6	6	1	Pin	out.REG
1/1	1	1	Pin	out.C
1/1	1	1	Pin	out.SR
1/1	1	1	Node	mux1.SP
1/5	5	1	Node	mux1.REG
1/1	1	1	Node	mux1.C
2/6	6	1	Node	mux2.REG
1/1	1	1	Node	mux2.C
1/1	1	1	Node	mux2.SR
2/6	6	1	Node	mux3.REG
1/1	1	1	Node	mux3.C
1/1	1	1	Node	mux3.SR
1/4	4	1	Node	equal
497/44	19	1	Node	code1
497/44	19	1	Node	code2
497/44	19	1	Node	code3
497/44	19	1	Node	code4
=========				
2004/211				

```
        Best P-Term Total: 192
              Total Pins: 20
             Total Nodes: 8
  Average P-Term/Output: 20
```

- **Design won't fit in any standard PLD**
  - too many input signals
  - needs blocks with 44 product terms per output (inverted) for 4 signals
  - must try to break large functions into smaller ones (multi-level logic)

Digital combination lock (introduce intermediate signals)

```
module lock2

 interface (clk, rst, new, C1_1..C1_4, C2_1..C2_4, C3_1..C3_4, value1..value4 -> out);

title 'digital combination lock'

"Inputs
 clk, rst, new  pin;
 C1_1..C1_4     pin;
 C2_1..C2_4     pin;
 C3_1..C3_4     pin;
 value1..value4 pin;

"Outputs
 out            pin  istype 'reg,buffer';

"Internal signals
 mux1..mux3     node istype 'reg,buffer';
 equal          node;
 code1..code4   node istype 'keep';
 equal1,equal2  node istype 'keep';

"Alias for state register and encoding
 SREG   = [mux1..mux3,out];
 S1     = [1,0,0,0];
 S2     = [0,1,0,0];
 S3     = [0,0,1,0];
 ERR    = [0,0,0,0];
 OPEN   = [0,0,0,1];
 cntrl1 = [mux1, mux1, mux1, mux1];
 cntrl2 = [mux2, mux2, mux2, mux2];
 cntrl3 = [mux3, mux3, mux3, mux3];
```

```
EQUATIONS

 SREG.clk = clk;
 mux1.set = rst;
 mux2.clr = rst;
 mux3.clr = rst;
 out.clr  = rst;

 [code1..code4] = ( (cntrl1 & [C1_1..C1_4])
                  # (cntrl2 & [C2_1..C2_4])
                  # (cntrl3 & [C3_1..C3_4]) );

 equal1 = (code1 == value1) & (code2 == value2);
 equal2 = (code3 == value3) & (code4 == value4);
 equal  = equal1 & equal2;

STATE_DIAGRAM SREG

 state S1:   if !new then S1
               else if equal then S2   else ERR;
 state S2:   if !new then S2
               else if equal then S3   else ERR;
 state S3:   if !new then S3
               else if equal then OPEN else ERR;
 state ERR:  goto ERR;
 state OPEN: goto OPEN;

END
```

Digital combination lock (equations for intermediate signals)

```
equal = (equal1 & equal2);

code1 = (mux2 & C2_1
     # mux1 & C1_1
     # mux3 & C3_1);

code2 = (mux2 & C2_2
     # mux1 & C1_2
     # mux3 & C3_2);

code3 = (mux2 & C2_3
     # mux1 & C1_3
     # mux3 & C3_3);

code4 = (mux2 & C2_4
     # mux1 & C1_4
     # mux3 & C3_4);

equal1 = (!code1 & !value1 & !code2 & !value2
     # code1 & value1 & !code2 & !value2
     # !code1 & !value1 & code2 & value2
     # code1 & value1 & code2 & value2);

equal2 = (!code3 & !value3 & !code4 & !value4
     # code3 & value3 & !code4 & !value4
     # !code3 & !value3 & code4 & value4
     # code3 & value3 & code4 & value4);
```

Digital combination lock (new logic)

- **ABEL output**

```
 P-Terms   Fan-in  Fan-out  Type  Name (attributes)
---------  ------  -------  ----  -----------------
   2/6        6       1     Pin   out.REG
   1/1        1       1     Pin   out.C
   1/1        1       1     Pin   out.SR
   1/1        1       1     Node  mux1.SP
   1/5        5       1     Node  mux1.REG
   1/1        1       1     Node  mux1.C
   2/6        6       1     Node  mux2.REG
   1/1        1       1     Node  mux2.C
   1/1        1       1     Node  mux2.SR
   2/6        6       1     Node  mux3.REG
   1/1        1       1     Node  mux3.C
   1/1        1       1     Node  mux3.SR
   1/2        2       1     Node  equal
   3/8        6       1     Node  code1
   3/8        6       1     Node  code2
   3/8        6       1     Node  code3
   3/8        6       1     Node  code4
   4/4        4       1     Node  equal1
   4/4        4       1     Node  equal2
=========
  36/73         Best P-Term Total: 36
                       Total Pins: 20
                      Total Nodes: 10
            Average P-Term/Output: 2
```

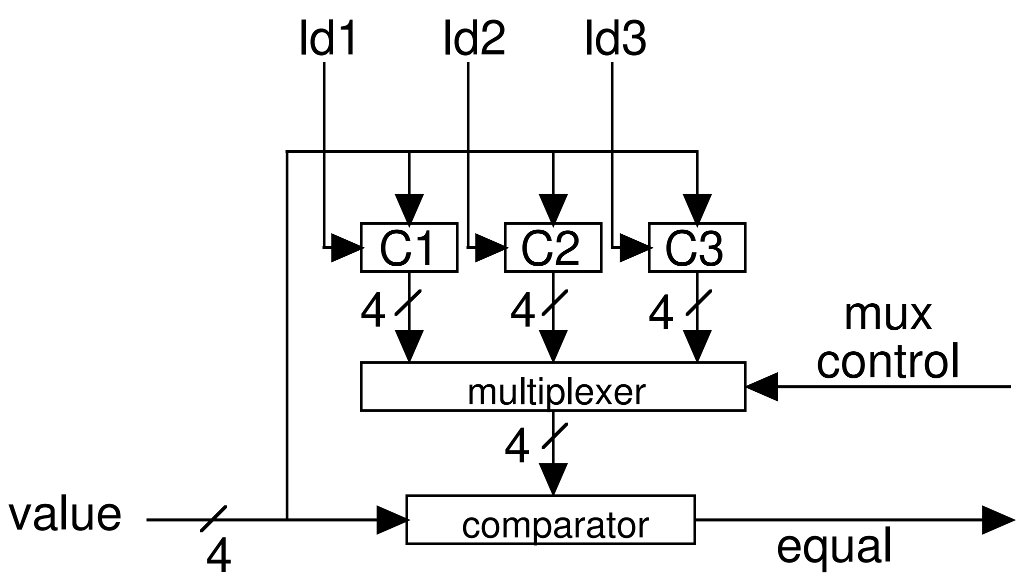
Digital combination lock (PLD mapping)

- **40 pin chip**
- **Enough registers and input pins**
- **Does not include code registers**
  - requires separate chip with 12 register bits

```
                        E0900

              +----------\        /----------+
              |           \      /           |
              |            -----             |
        clk   |  1                       40  |  Vcc
        rst   |  2                       39  |  C2_4
       C1_1   |  3                       38  |  C1_4
       C2_1   |  4                       37  |  C3_3
      code1   |  5                       36  |  C3_4
      code2   |  6                       35  |  value1
      code3   |  7                       34  |  value2
      code4   |  8                       33  |  value3
     equal1   |  9                       32  |  value4
     equal2   | 10                       31  |  new
       mux1   | 11                       30  |
       mux2   | 12                       29  |
       mux3   | 13                       28  |
        out   | 14                       27  |
              | 15                       26  |
              | 16                       25  |
       C3_1   | 17                       24  |  C2_3
       C1_2   | 18                       23  |  C1_3
       C2_2   | 19                       22  |  C3_2
        GND   | 20                       21  |
              |                              |
              `------------------------------'
```

Digital combination lock (new data-path to 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)



Digital combination lock (HDL description of new data-path)

```
module lock3

 interface (clk, rst, new, ld1..ld3, value1..value4 -> out);

title 'digital combination lock'

"Inputs
 clk, rst, new  pin;
 ld1..ld3       pin;
 value1..value4 pin;

"Outputs
 out            pin  istype 'reg,buffer';

"Internal signals
 C1_1..C1_4     node istype 'reg';
 C2_1..C2_4     node istype 'reg';
 C3_1..C3_4     node istype 'reg';
 mux1..mux3     node istype 'reg';
 equal          node;
 code1..code4   node istype 'keep';
 equal1,equal2  node istype 'keep';

"Alias for state register and encoding
 value  = [value1..value4];
 C1     = [C1_1..C1_4];
 C2     = [C2_1..C2_4];
 C3     = [C3_1..C3_4];
 SREG   = [mux1..mux3,out];
 S1     = [1,0,0,0];
 S2     = [0,1,0,0];
 S3     = [0,0,1,0];
 ERR    = [0,0,0,0];
 OPEN   = [0,0,0,1];
 cntrl1 = [mux1, mux1, mux1, mux1];
 cntrl2 = [mux2, mux2, mux2, mux2];
 cntrl3 = [mux3, mux3, mux3, mux3];
```

```
EQUATIONS

 C1.clk   = clk;
 C2.clk   = clk;
 C3.clk   = clk;
 SREG.clk = clk;
 mux1.set = rst;
 mux2.clr = rst;
 mux3.clr = rst;
 out.clr  = rst;

 WHEN ld1 THEN C1 := value ELSE C1 := C1.fb;
 WHEN ld2 THEN C2 := value ELSE C2 := C2.fb;
 WHEN ld1 THEN C3 := value ELSE C3 := C3.fb;

 [code1..code4] = ( (cntrl1 & C1.fb)
                  # (cntrl2 & C2.fb)
                  # (cntrl3 & C3.fb) );

 equal1 = (code1 == value1) & (code2 == value2);
 equal2 = (code3 == value3) & (code4 == value4);
 equal  = equal1 & equal2;

STATE_DIAGRAM SREG

 state S1:   if !new then S1
               else if equal then S2   else ERR;
 state S2:   if !new then S2
               else if equal then S3   else ERR;
 state S3:   if !new then S3
               else if equal then OPEN else ERR;
 state ERR:  goto ERR;
 state OPEN: goto OPEN;

END
```

Digital combination lock (logic for new data-path)

P-Terms	Fan-in	Fan-out	Type	Name (attributes)
2/6	6	1	Pin	out.REG
1/1	1	1	Pin	out.C
1/1	1	1	Pin	out.SR
2/2	3	1	Node	C1_1.REG
1/1	1	1	Node	C1_1.C
2/2	3	1	Node	C1_2.REG
1/1	1	1	Node	C1_2.C
2/2	3	1	Node	C1_3.REG
1/1	1	1	Node	C1_3.C
2/2	3	1	Node	C1_4.REG
1/1	1	1	Node	C1_4.C
2/2	3	1	Node	C2_1.REG
1/1	1	1	Node	C2_1.C
2/2	3	1	Node	C2_2.REG
1/1	1	1	Node	C2_2.C
2/2	3	1	Node	C2_3.REG
1/1	1	1	Node	C2_3.C
2/2	3	1	Node	C2_4.REG
1/1	1	1	Node	C2_4.C
2/2	3	1	Node	C3_1.REG
1/1	1	1	Node	C3_1.C
2/2	3	1	Node	C3_2.REG
1/1	1	1	Node	C3_2.C
2/2	3	1	Node	C3_3.REG
1/1	1	1	Node	C3_3.C
2/2	3	1	Node	C3_4.REG
1/1	1	1	Node	C3_4.C
1/5	5	1	Node	mux1.REG
1/1	1	1	Node	mux1.SET
1/1	1	1	Node	mux1.C
2/6	6	1	Node	mux2.REG
1/1	1	1	Node	mux2.CLR
1/1	1	1	Node	mux2.C
2/6	6	1	Node	mux3.REG
1/1	1	1	Node	mux3.CLR
1/1	1	1	Node	mux3.C
1/2	2	1	Node	equal
3/8	6	1	Node	code1
3/8	6	1	Node	code2
3/8	6	1	Node	code3
3/8	6	1	Node	code4
4/4	4	1	Node	equal1
4/4	4	1	Node	equal2
=========				
72/109				

Best P-Term Total: 72
Total Pins: 11
Total Nodes: 22
Average P-Term/Output: 2

Digital combination lock (PLD mapping)

- **Entire data-path in one chip**

- **Including code registers**
  - loaded from value inputs
  - keep ld1..ld3 secure

```
                     E0900

           +----------\       /----------+
           |           \     /           |
           |            -----            |
    clk    |  1                      40  |  Vcc
    rst    |  2                      39  |
 value1    |  3                      38  |
    ld1    |  4                      37  |
  code1    |  5                      36  |
  code2    |  6                      35  |
  code3    |  7                      34  |  out
  code4    |  8                      33  |  mux3
 equal1    |  9                      32  |  mux2
 equal2    | 10                      31  |  mux1
   C1_1    | 11                      30  |  C3_4
   C1_2    | 12                      29  |  C3_3
   C1_3    | 13                      28  |  C3_2
   C1_4    | 14                      27  |  C3_1
   C2_1    | 15                      26  |  C2_4
   C2_2    | 16                      25  |  C2_3
 value2    | 17                      24  |
 value3    | 18                      23  |  new
 value4    | 19                      22  |  ld2
    GND    | 20                      21  |
           |                             |
           `-----------------------------'
```

Section review

- **Basic timing behavior of FSMs**
  - when are inputs sampled, next state/outputs transition and stabilize
  - Moore and Mealy (sync and async) machine organizations
  - outputs = F(state) vs. outputs = F(state, inputs)

- **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