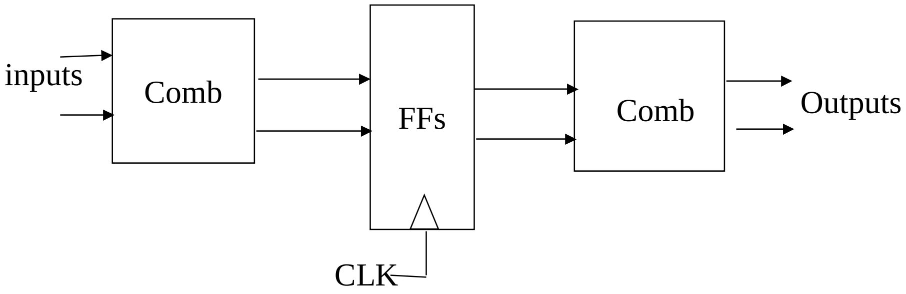
Sequential Circuits

- Another way to understand setup/hold/propagation time



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)

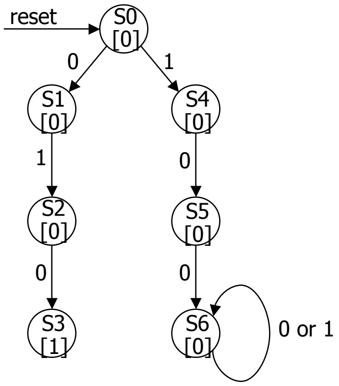
- Finite string pattern recognizer
  - one input (X) and one output (Z)
  - output is asserted whenever the input sequence …010… has been observed, as long as the sequence 100 has never been seen
- Step 1: understanding the problem statement
  - sample input/output behavior:

```
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 …
```

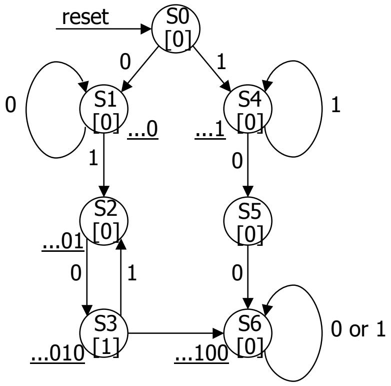
Finite string pattern recognizer (step 2)

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



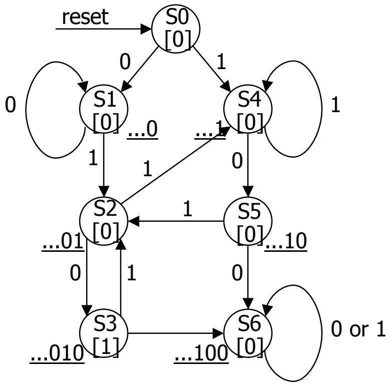
Finite string pattern recognizer (step 2, cont'd)

- 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

- 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

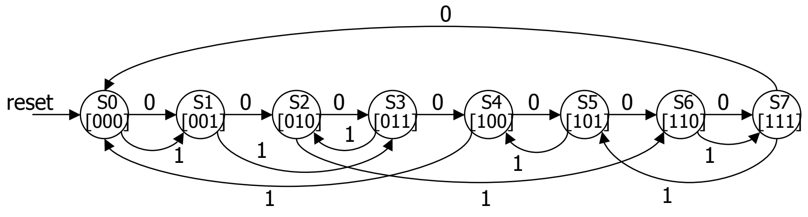
binary: 000, 001, 010, 011, 100, 101, 110, 111
Gray: 000, 001, 011, 010, 110, 111, 101, 100

- 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

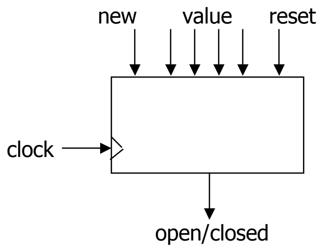


Digital combinational lock

- 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

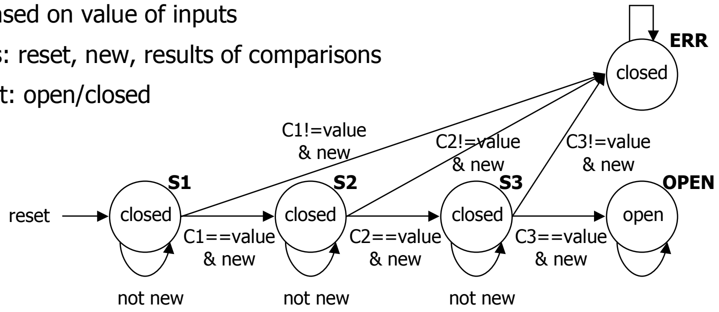
Determining details of the specification

- 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?



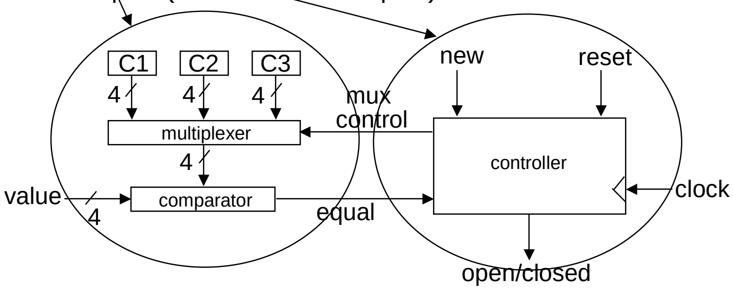
Digital combination lock state diagram

- 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)



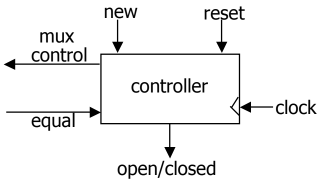
State table for combination lock

- 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
    - 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

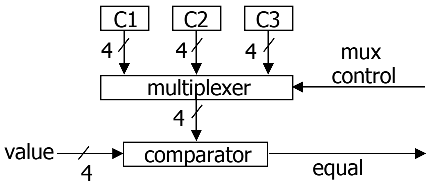


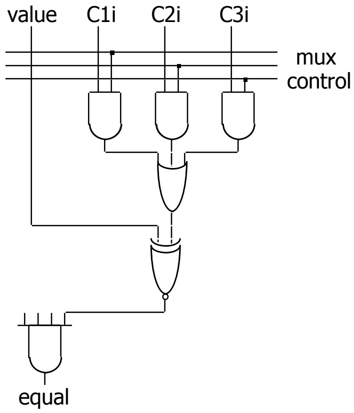
reset	new	equal	state	next state	mux	open/closed
1	–	–	–	0001	001	0
0	0	–	0001	0001	001	0
0	1	0	0001	0000	–	0
0	1	1	0001	0010	010	0
...						
0	1	1	0100	1000	–	1
...						

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

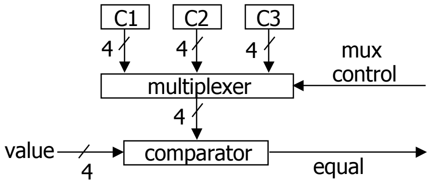
- Multiplexer
  - easy to implement as combinational logic when few inputs
  - logic can easily get too big for most PLDs

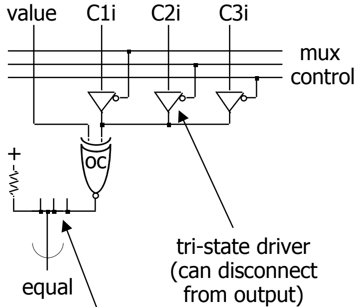




Data-path implementation (cont'd)

- 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 driver (can disconnect from output)

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

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
- Three case studies
  - understand I/O behavior
  - draw diagrams
  - enumerate states for the "goal"
  - expand with error conditions
  - reuse states whenever possible