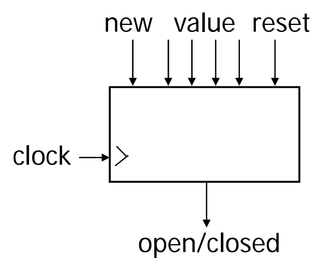
Lecture 27

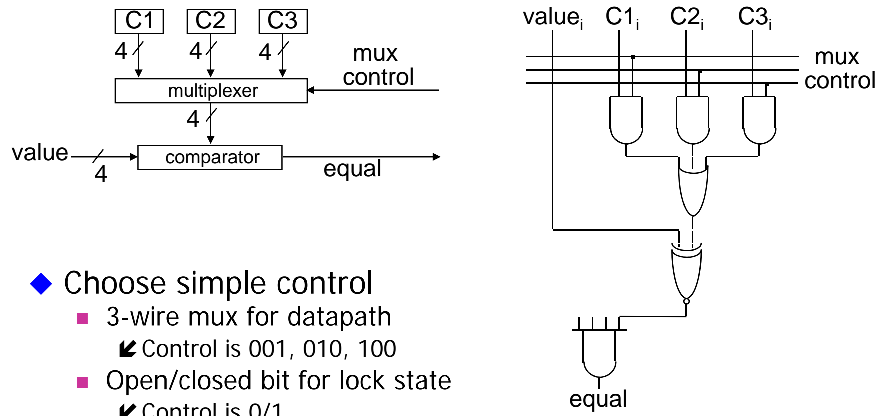
- Logistics
  - HW8 due Friday
  - Ants problem due Friday
  - Lab kit must be returned to Tony by Friday
  - Review Sunday 12/7 3pm, Location TBD
- Last lecture
  - State encoding
    - One-hot encoding
    - Output encoding
- Today:
  - Optimizing FSMs
    - Pipelining
    - Retiming
    - Partitioning

Example: Digital combination lock

- An output-encoded FSM
  - Punch in 3 values in sequence and the door opens
  - If there is an error the lock must be reset
  - After the door opens the lock must be reset
  - Inputs: sequence of number values, reset
  - Outputs: door open/close



Design the datapath



- Choose simple control
  - 3-wire mux for datapath
    - Control is 001, 010, 100
  - Open/closed bit for lock state
    - Control is 0/1

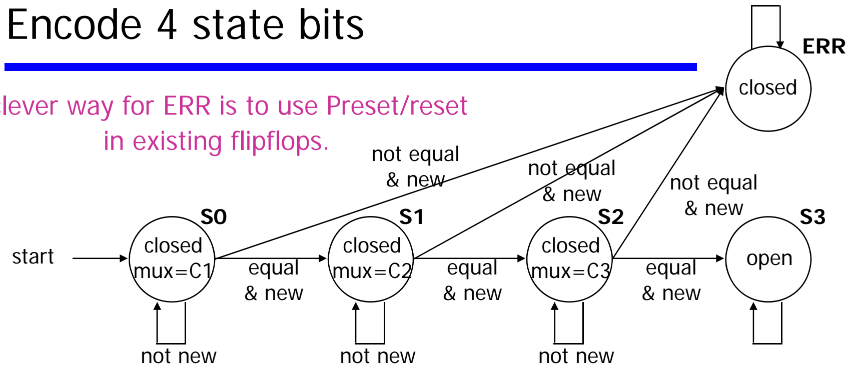
Output encode the FSM

- FSM outputs
  - Mux control is 100, 010, 001
  - Lock control is 0/1
- State are: S0, S1, S2, S3, or ERR
  - Can use 3, 4, or 5 bits to encode
  - Have 4 outputs, so choose 4 bits
    - Encode mux control and lock control in state bits
    - Lock control is first bit, mux control is last 3 bits

      S0 = 0001 (lock closed, mux first code)
      S1 = 0010 (lock closed, mux second code)
      S2 = 0100 (lock closed, mux third code)
      S3 = 1000 (lock open)
      ERR = 0000 (error, lock closed)

Encode 4 state bits

A clever way for ERR is to use Preset/reset in existing flipflops.



$S_0+ = S_0N'$

$S_1+ = S_0EN + S_1N'$

$S_2+ = S_1EN + S_2N'$

$S_3+ = S_2EN + S_3$

Not equal & new

Already in ERR

$Preset_0 = start$

$Preset_{1,2,3} = 0$

$Reset_0 = start'(E'N + (Q_0+Q_1+Q_2+Q_3)')$

$Reset_{1,2,3} = start + (E'N + (Q_0+Q_1+Q_2+Q_3)')$

$D_0 = Q_0N'$

$D_1 = Q_0EN + Q_1N'$

$D_2 = Q_1EN + Q_2N'$

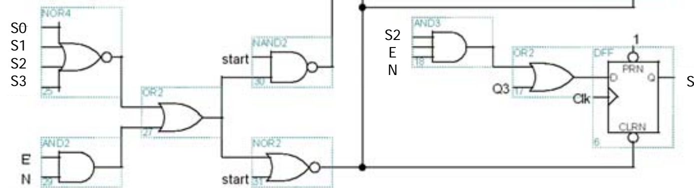
$D_3 = Q_2EN + Q_3$

$Preset_0 = start$

$Preset_{1,2,3} = 0$

$Reset_0 = start'(E'N + (Q_0+Q_1+Q_2+Q_3)')$

$Reset_{1,2,3} = start + (E'N + (Q_0+Q_1+Q_2+Q_3)')$



FSM design

- FSM-design procedure
  1. State diagram
  2. state-transition table
  3. State minimization
  4. State encoding
  5. Next-state logic minimization
  6. Implement the design

Last topic: more FSM optimization techniques

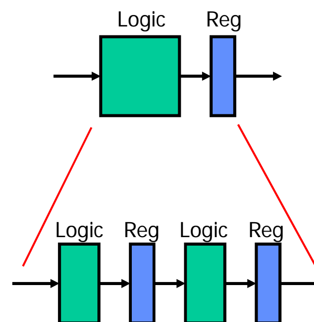
- Want to optimize FSM for many reasons beyond state minimization and efficient encoding
- Additional techniques
  - Pipelining --- allows faster clock speed
  - Retiming --- can reduce registers or change delays
  - Partitioning --- can divide to multiple devices, simpler logic

Pipelining related definitions

- Latency: Time to perform a computation
  - Data input to data output
- Throughput: Input or output data rate
  - Typically the clock rate
- Combinational delays drive performance
  - Define $d \equiv$ delay through slowest combinational stage
    $n \equiv$ number of stages from input to output
  - Latency $\propto n * d$ (in sec)
  - Throughput $\propto 1/d$ (in Hz)

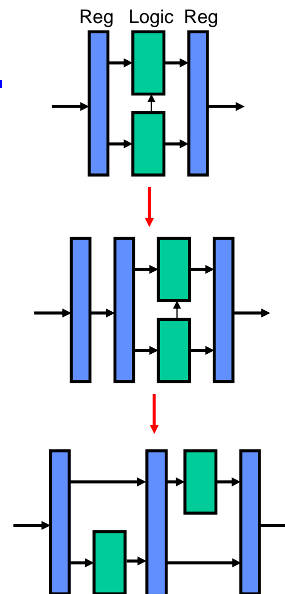
Pipelining

- What?
  - Subdivide combinational logic
  - Add registers between logic
- Why?
  - Trade latency for throughput
  - Increased throughput
    - Reduce logic delays
    - Increase clock speed
  - May increased latency
  - Increase circuit utilization
    - Simultaneous computations



Pipelining

- When?
  - Need throughput more than latency
    - Signal processing
  - Logic delays > setup/hold times
  - Acyclic logic
- Where?
  - At natural breaks in the combinational logic
  - Adding registers makes sense

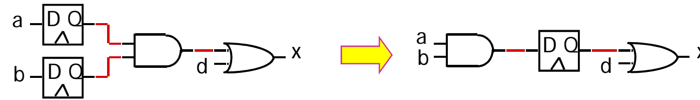


Retiming

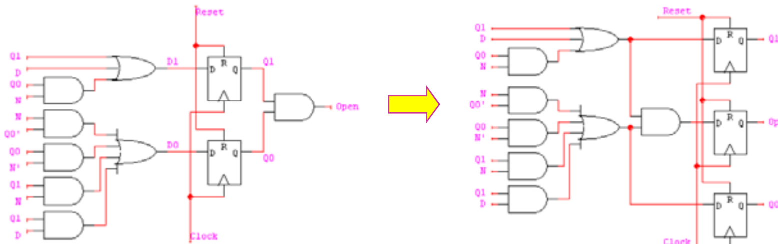
- Pipelining adds registers
  - To increase the clock speed
- Retiming moves registers around
  - Reschedules computations to optimize performance
    - Change delay patterns
    - Reduce register count
  - Without altering functionality

Retiming examples

- Reduce register count



- Change output delays

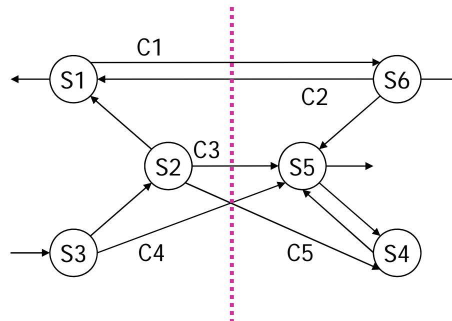


FSM partitioning

- Break a large FSM into two or more smaller FSMs
- Rationale
  - Less states in each partition
    - Simpler minimization and state assignment
    - Smaller combinational logic
    - Shorter critical path
  - But more logic overall
- Partitions are synchronous
  - Same clock!!!

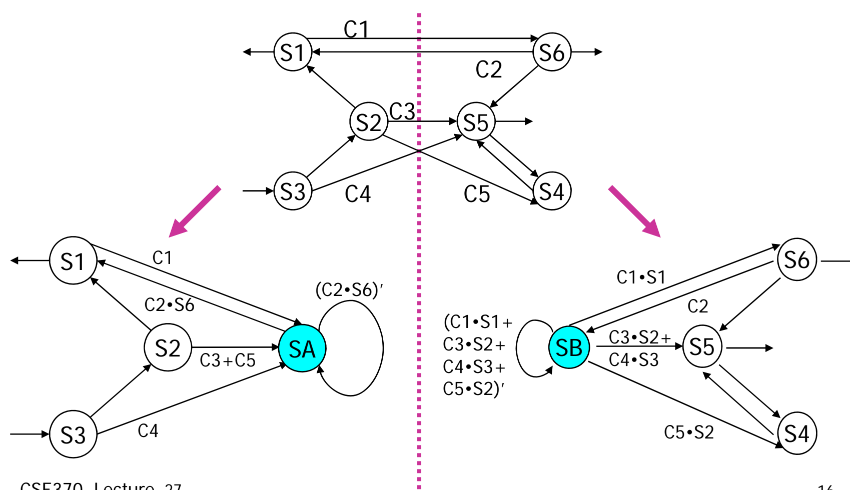
Example: Partition the machine

- Partition into two halves



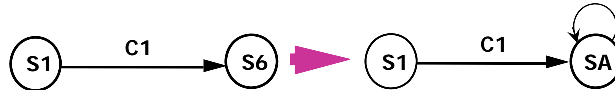
Introduce idle states

- SA and SB handoff control between machines

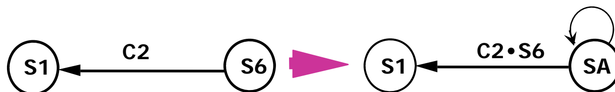


Partitioning rules

Rule #1: Source-state transformation
Replace by transition to idle state (SA)

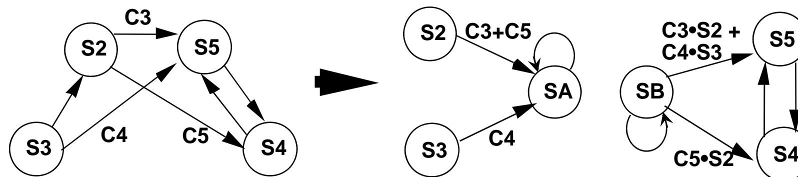


Rule #2: Destination state transformation
Replace with exit transition from idle state

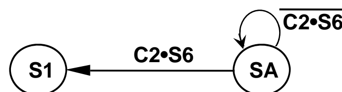


Partitioning rules (con't)

Rule #3: Multiple transitions with same source or destination
Source ⇒ Replace by transitions to idle state (SA)
Destination ⇒ Replace with exit transitions from idle state



Rule #4: Hold condition for idle state
OR exit conditions and invert



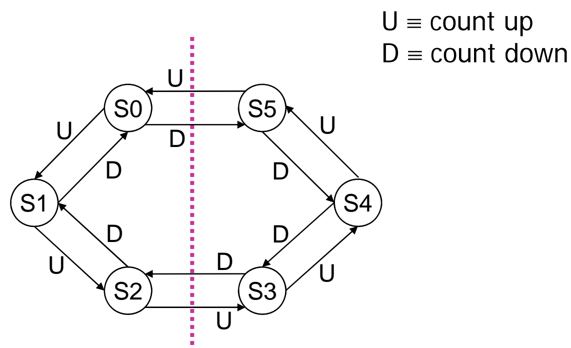
Mealy versus Moore partitions

- Mealy machines undesirable
  - Inputs can affect outputs immediately
    - "output" can be a handoff to another machine!!!
- Moore machines desirable
  - Input-to-output path always broken by a flip-flop
  - But...may take several clocks for input to propagate to output

Example: Six-state up/down counter

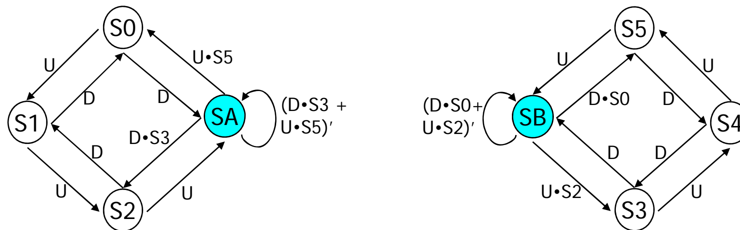
- Break into 2 parts

U ≡ count up
D ≡ count down



Example: 6 state up/down counter (con't)

- Count sequence S_0, S_1, S_2, S_3, S_4, S_5
  - S_2 goes to S_A and holds, leaves after S_5
  - S_5 goes to S_B and holds, leaves after S_2
  - Down sequence is similar



Minimize communication between partitions

- Ideal world: Two machines handoff control
  - Separate I/O, states, etc.
- Real world: Minimize handoffs and common I/O
  - Minimize number of state bits that cross boundary
  - Merge common outputs

Done!