EECS 150 - Components and Design Techniques for Digital Systems

Lec 9 – Putting it all together…

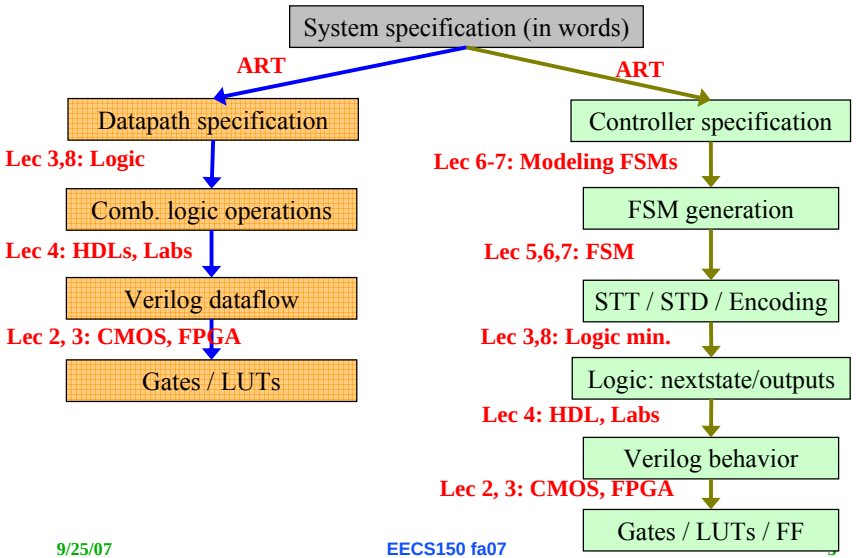
9-25-07

David Culler
Electrical Engineering and Computer Sciences
University of California, Berkeley
http://www.eecs.berkeley.edu/~culler
http://inst.eecs.berkeley.edu/~cs150

Outline

- Top-to-bottom
  - What have we covered so far?
- Combo Lock example
  - FSM to logic
  - Mapping to FPGAs
- Announcements
- Counters revisited
- Another example – Ant Brain

Digital design - as we've seen it



Where are we now?

- (Synchronous) Sequential systems
- Given datapath and control specifications
  - Generate comb. logic for datapath
    - » Minimize logic for efficient implementation
  - Generate FSM for controller
    - » Choose implementation, encoding
    - » Generate logic for nextstate and output
  - Describe datapath and controller in Verilog
    - » structure, dataflow and behavior
    - » Map onto gates or LUTs
- Seems like a good point to "test" your understanding!

Representation of digital designs

- Physical devices (transistors, relays)
- *Switches*
- *Truth tables*
- *Boolean algebra*
- *Gates*
- Waveforms
- Finite state behavior
- Register-transfer behavior
- Concurrent abstract specifications

scope of CS 150

more depth than 61C

focus on building systems

Logic Functions and Boolean Algebra

- **Any logic function that can be expressed as a truth table can be written as an expression in Boolean algebra using the operators: ', +, and •**

X	Y	X • Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X'	X' • Y
0	0	1	0
0	1	1	1
1	0	0	0
1	1	0	0

X	Y	X'	Y'	X • Y	X' • Y'	(X • Y) + (X' • Y')
0	0	1	1	0	1	1
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	1	0	0	1	0	1

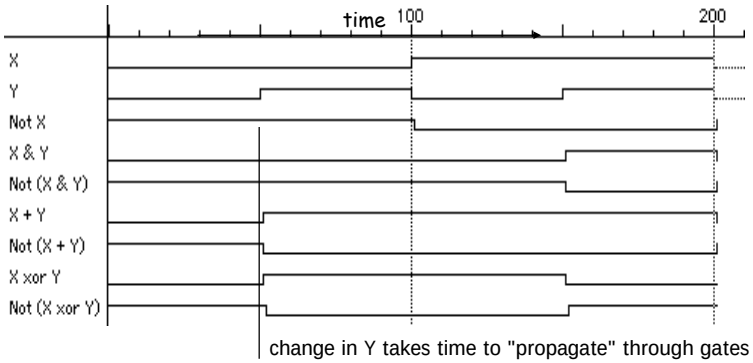
(X • Y) + (X' • Y') ≡ X = Y

Boolean expression that is true when the variables X and Y have the same value and false, otherwise

X, Y are Boolean algebra variables

Waveform View of Logic Functions

- **Just a sideways truth table**
  - **But note how edges don't line up exactly**
  - **It takes time for a gate to switch its output!**



An algebraic structure

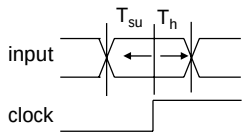
- **An algebraic structure consists of**
  - **a set of elements B**
  - **binary operations { + , • }**
  - **and a unary operation { ' }**
  - **such that the following axioms hold:**

1. set B contains at least two elements, a, b, such that a ≠ b

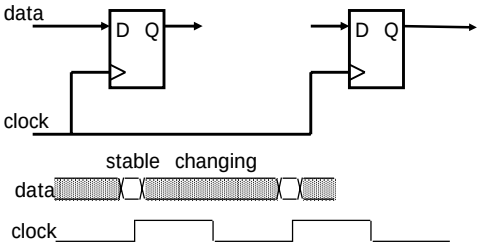
2. closure:	**a + b is in B**	**a • b is in B**
3. commutativity:	**a + b = b + a**	**a • b = b • a**
4. associativity:	**a + (b + c) = (a + b) + c**	**a • (b • c) = (a • b) • c**
5. identity:	**a + 0 = a**	**a • 1 = a**
6. distributivity:	**a + (b • c) = (a + b) • (a + c)**	**a • (b + c) = (a • b) + (a • c)**
7. complementarity:	**a + a' = 1**	**a • a' = 0**

Timing Methodologies (cont'd)

- **Definition of terms**
  - clock: periodic event, causes state of storage element to change; can be rising or falling edge, or high or low level
  - setup time: minimum time before the clocking event by which the input must be stable (Tsu)
  - hold time: minimum time after the clocking event until which the input must remain stable (Th)



there is a timing "window" around the clocking event during which the input must remain stable and unchanged in order to be recognized



Axioms & theorems of Boolean algebra

- **Identity**
  1. X + 0 = X 1D. X • 1 = X
- **Null**
  2. X + 1 = 1 2D. X • 0 = 0
- **Idempotency:**
  3. X + X = X 3D. X • X = X
- **Involution:**
  4. (X')' = X
- **Complementarity:**
  5. X + X' = 1 5D. X • X' = 0
- **Commutativity:**
  6. X + Y = Y + X 6D. X • Y = Y • X
- **Associativity:**
  7. (X + Y) + Z = X + (Y + Z) 7D. (X • Y) • Z = X • (Y • Z)

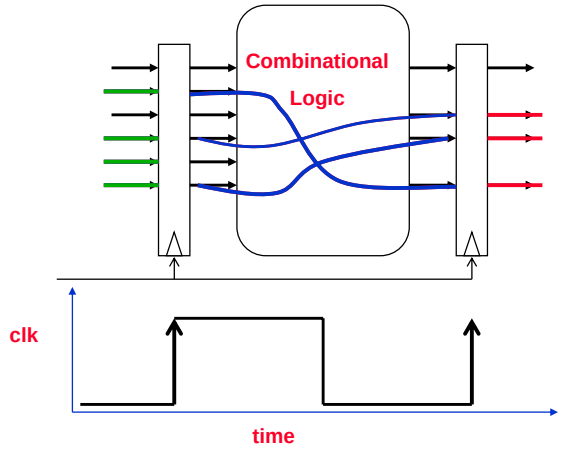
Axioms and theorems of Boolean algebra (cont'd)

- **Distributivity:**
  8. X • (Y + Z) = (X • Y) + (X • Z) 8D. X + (Y • Z) = (X + Y) • (X + Z)
- **Uniting:**
  9. X • Y + X • Y' = X 9D. (X + Y) • (X + Y') = X
- **Absorption:**
  10. X + X • Y = X 10D. X • (X + Y) = X
  11. (X + Y') • Y = X • Y 11D. (X • Y') + Y = X + Y
- **Factoring:**
  12. (X + Y) • (X' + Z) = X • Z + X' • Y 12D. X • Y + X' • Z = (X + Z) • (X' + Y)
- **Concensus:**
  13. (X • Y) + (Y • Z) + (X' • Z) = X • Y + X' • Z 13D. (X + Y) • (Y + Z) • (X' + Z) = (X + Y) • (X' + Z)

Axioms and theorems of Boolean algebra (cont')

- **de Morgan's:**
  14. (X + Y + ...)' = X' • Y' • ... 14D. (X • Y • ...)' = X' + Y' + ...
- **generalized de Morgan's:**
  15. f'(X1,X2,...,Xn,0,1,+,•) = f(X1',X2',...,Xn',1,0,•,+)
- **establishes relationship between • and +**

Recall: What makes Digital Systems tick?

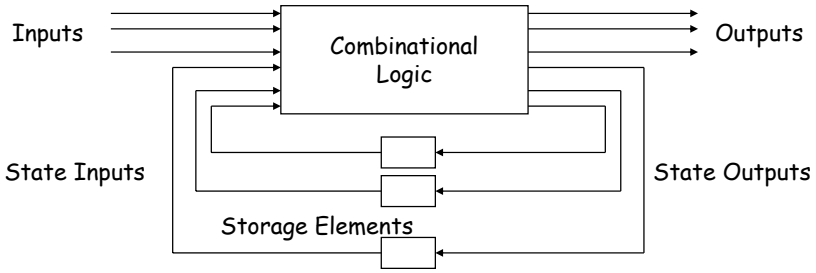


Sequential Logic Implementation

- Models for representing sequential circuits
  - Finite-state machines (Moore and Mealy)
  - Representation of memory (states)
  - Changes in state (transitions)
- Design procedure
  - State diagrams
  - Implementation choice: counters, shift registers, FSM
  - State transition table
  - State encoding
  - Combinational logic
    - » Next state functions
    - » Output functions

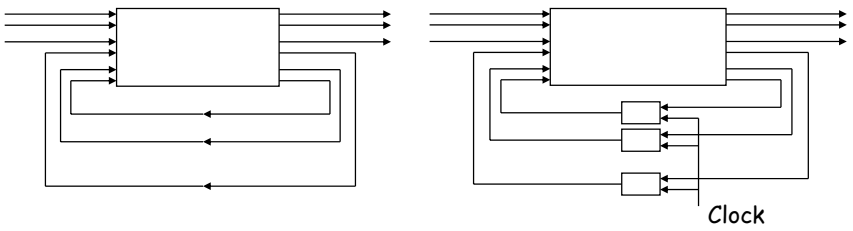
Abstraction of State Elements

- Divide circuit into combinational logic and state
- Localize feedback loops and make it easy to break cycles
- Implementation of storage elements leads to various forms of sequential logic



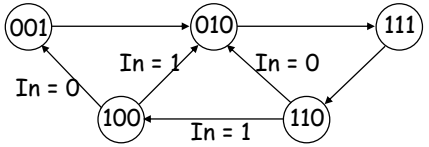
Forms of Sequential Logic

- Asynchronous sequential logic – state changes occur whenever state inputs change (elements may be simple wires or delay elements)
- Synchronous sequential logic – state changes occur in lock step across all storage elements (using a periodic waveform - the clock)



FSM Representations

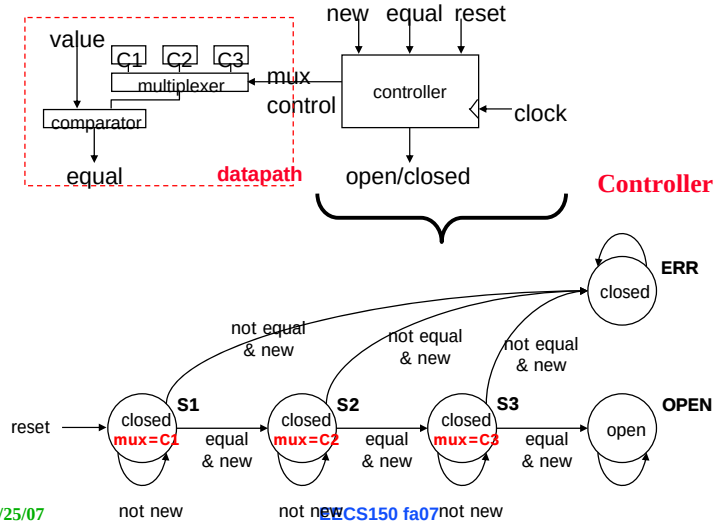
- States: determined by possible values in sequential storage elements
- Transitions: change of state
- Clock: controls when state can change by controlling storage elements



- Sequential Logic
  - Sequences through a series of states
  - Based on sequence of values on input signals
  - Clock period defines elements of sequence

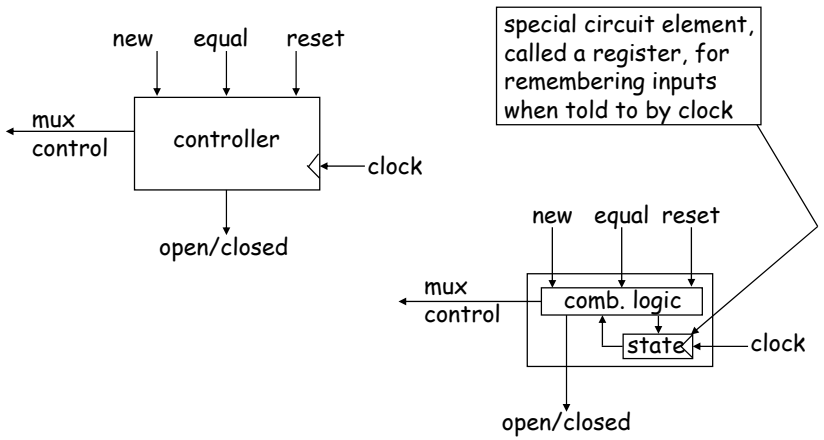
Example: FSM Design – Combo lock

- Combination lock



Combo lock - controller implementation

- Implementation of the controller



Combo Lock - State Encoding

reset	new	equal	state	nstate	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	0	-	S2	S2	C2	closed
0	1	0	S2	ERR	-	closed
0	1	1	S2	S3	C3	closed
0	0	-	S3	S3	C3	closed
0	1	0	S3	ERR	-	closed
0	1	1	S3	OPEN	-	closed
0	-	-	OPEN	OPEN	-	open
0	-	-	ERR	ERR	-	closed

Symbolic states and outputs

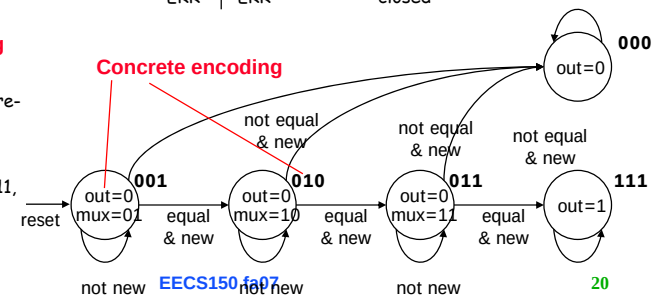
One possible encoding

Mux control:
C1 = 01, C2 = 10, C3 = 11 (pre-established)

State encoding:
S1 = 001, S2 = 010, S3 = 011, OPEN = 111, Error = 000

Output encoding:
Closed = 0, Open = 1

Concrete encoding



FSM implementation

- Steps for the hardware designer:
  - Word specification
  - FSM design
  - Encoding
  - Verification!
- At this point, hand over to synthesis tools:
  - Describe FSM behavior in Verilog
  - Synthesize controller
- Good encoding
  - Better performance
  - Fewer state bits
  - Possibility of state minimization
  - Tools also try to figure this out

For this example, go through the logic synthesis steps (ideally, tools take care of all this).

Example: Combo Lock

reset (r)	new (n)	equal (e)	state (s_2 s_1 s_0)	nstate (n_2 n_1 n_0)	mux (m_1 m_0)	open (o)
1	-	-	---	001	--	0
0	0	-	001	001	01	0
0	1	0	001	000	01	0
0	1	1	001	010	01	0
0	0	-	010	010	10	0
0	1	0	010	000	10	0
0	1	1	010	011	10	0
0	0	-	011	011	11	0
0	1	0	011	000	11	0
0	1	1	011	111	11	0
0	-	-	111	111	--	1
0	-	-	000	000	--	0
0	-	-	100	---	--	-
0	-	-	101	---	--	-
0	-	-	110	---	--	-

Next state and output logic

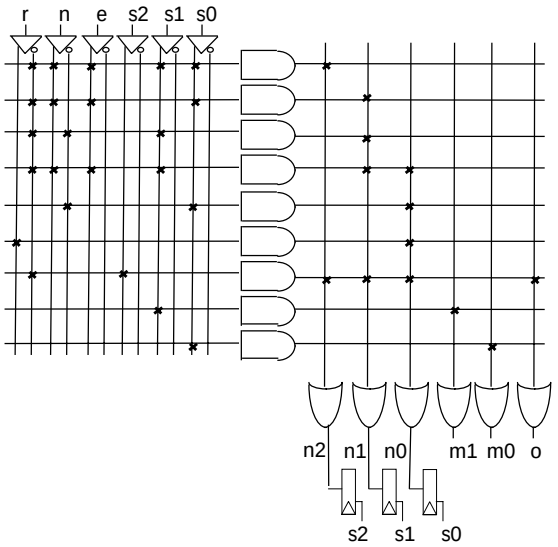
nextstate (n2 n1 n0):
n2 = ~r (n e s1 s0 + s2)
n1 = ~r (n e s0 + e s1 + ~n s1 + s2)
n0 = r + s2 + n e s1 + ~n s0

mux outputs (m1, m0):
m1 = s1
m0 = s0
open (o):
o = s2

Take advantage of DCs!

How do we get these:
- K-maps?
- Tools
  - Espresso
  - Synplicity

Logic Implementation (on PLA)



Next state and output logic

nextstate (n2 n1 n0):
n2 = ~r (n e s1 s0 + s2)
n1 = ~r (n e s0 + e s1 + ~n s1 + s2)
n0 = r + s2 + n e s1 + ~n s0

mux outputs (m1, m0):
m1 = s1
m0 = s0

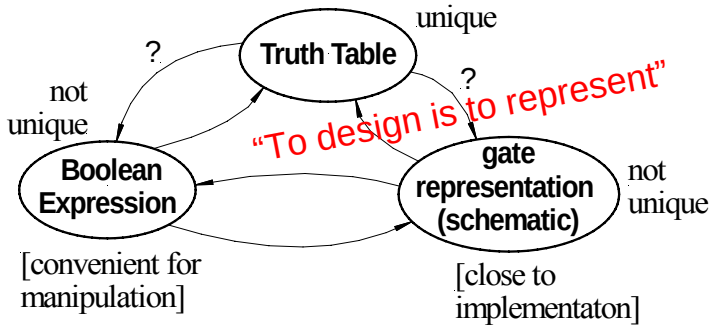
open (o):
o = s2

Alternate logic implementations

- PALs
- Multi-level circuits
  - Library of gates for implementation technology
- LUTs on FPGA
- …

Alternate Logic Representations

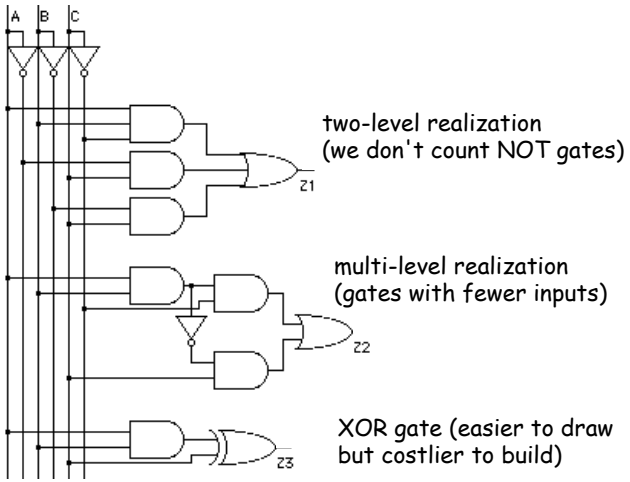
* **Theorem: Any Boolean function that can be expressed as a truth table can be written as an expression in Boolean Algebra using AND, OR, NOT.**



How do we convert from one to the other?

Choosing different realizations of a function

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

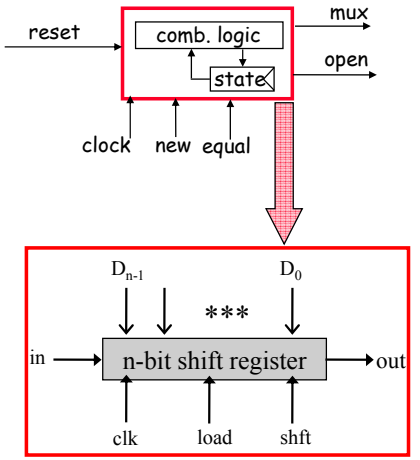


Which realization is best?

- **Reduce number of inputs**
  - **literal: input variable (complemented or not)**
    - » **approximate cost of logic gate is 2 transistors per literal**
  - **Fewer literals means less transistors - smaller circuits**
  - **Fewer inputs implies faster gates**
  - **Fan-ins (# of gate inputs) are limited in some technologies**
- **Reduce number of gates**
  - **Fewer gates (and the packages they come in) means smaller circuits**
- **Reduce number of levels of gates**
  - **Fewer level of gates implies reduced signal propagation delays**
- **How do we explore tradeoffs between increased circuit delay and size?**
  - **Automated tools to generate different solutions**
  - **Logic minimization: reduce number of gates and complexity**
  - **Logic optimization: reduction while trading off against delay**

Alternate Implementation: Controller based on Shift Register

- **Previous implementation**
  - **Comb. logic as gates (PLA)**
  - **State bits in latches**
- **Alternative**
  - **Shift reg to manipulate state**
  - **Simplify comb. logic**



Controller using Shift Register

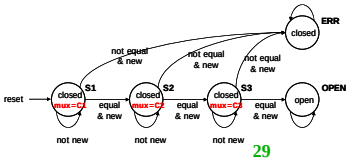
reset	new	equal	state	nstate	mux	open
(r)	(n)	(e)	($s_3s_2s_1s_0$)	($n_3n_2n_1n_0$)	(m_1 m_0)	(o)
1	-	-	----	1000	--	0
0	0	-	1000	1000	01	0
0	1	0	1000	0000	01	0
0	1	1	1000	0100	01	0
0	0	-	0100	0100	10	0
0	1	0	0100	0000	10	0
0	1	1	0100	0010	10	0
0	0	-	0010	0010	11	0
0	1	0	0010	0000	11	0
0	1	1	0010	0001	11	0
0	-	-	0001	0001	--	1
0	-	-	0000	0000	--	0
0	-	-	11--	----	--	-
0	-	-	1-1-	----	--	-
0	-	-	1--1	----	--	-
0	-	-	-11-	----	--	-
0	-	-	-1-1	----	--	-
0	-	-	--11	----	--	-

One-hot encoding scheme: state transition is a shift right

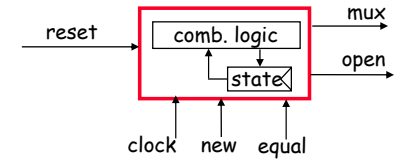
Mux control:
C1 = 01, C2 = 10, C3 = 11 (pre-established)

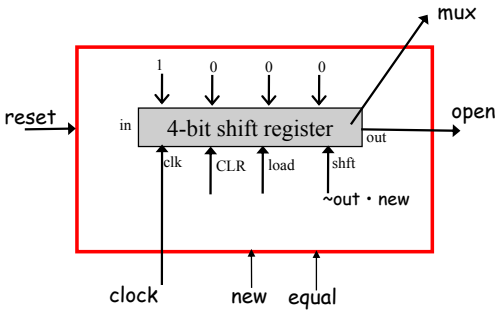
State encoding:
S1 = 1000, S2 = 0100,
S3 = 0010, OPEN = 0001, Error = 0000

Output encoding:
Closed = 0, Open = 1



Combo lock controller on shift reg

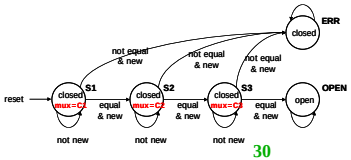




4-bit shift register:
[D3, D2, D1, D0] ← [0, 0, 0, 0]

Shift Reg		Controller
clk	←	clock
shft	←	(~out • new)
CLR	←	(~equal • new • ~out)
load	←	reset
in	←	0
out	→	open

Mux control (read register contents):
m1 = ~s3
m0 = ~s2

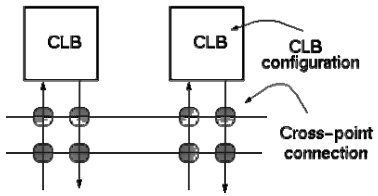


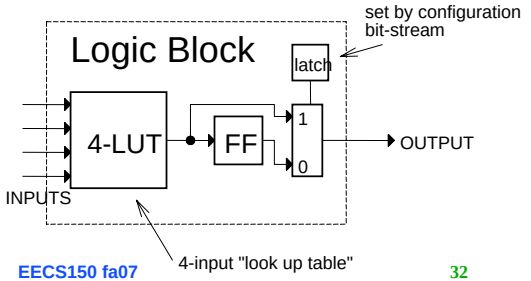
How does the combo lock look on an FPGA?

- **Latches**
  - **implement shift register (chain of 4 latches)**
- **LUTs**
  - **Combinational logic for out and mux control**
- **Routing fabric**
  - **Connect logical nets between CLBs**

Inside the FPGA

- **Network of Combinational logic blocks, memory and I/O**
  - **rich interconnect network**
  - **special units – multipliers, carry-logic**
- **CLBs**
  - **3 or 4-input *look up table (LUT)***
  - **implements combinational logic functions**
  - **Register optionally stores output of LUT**
- **Logic on FPGA**
  - **Configure LUTs (table of entries)**
  - **Configure latches in CLB**
  - **Program interconnect**





LUT as general logic gate

- **An n-lut as a direct implementation of a function** truth-table**.**
- **Each latch location holds the value of the function corresponding to one input combination.**

Example: 2-lut

INPUTS	AND	OR
00	0	0
01	0	1
10	0	1
11	1	1

Implements *any* function of 2 inputs.

How many of these are there?

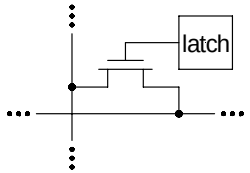
How many functions of n inputs?

Example: 4-lut

INPUTS	
0000	F(0,0,0,0) ← store in 1st latch
0001	F(0,0,0,1) ← store in 2nd latch
0010	F(0,0,1,0) ←
0011	F(0,0,1,1) ←
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

User Programmability

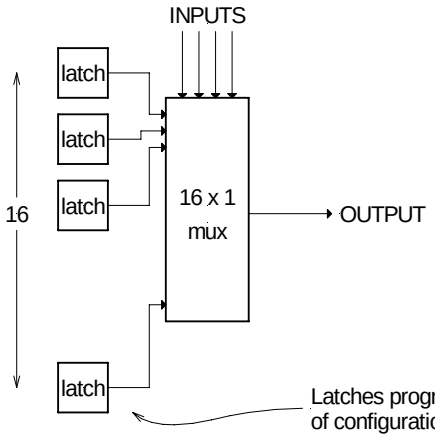
- **Latch-based (Xilinx, Altera, …)**



+ **reconfigurable**
– **volatile**
– **relatively large.**

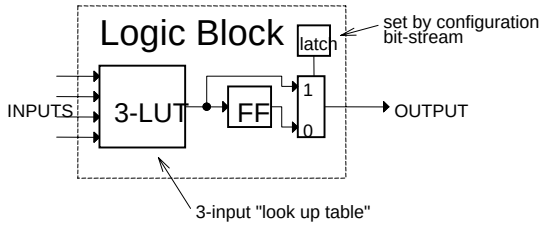
- **Latches are used to:**
  1. **make or break cross-point connections in the interconnect**
  2. **define the function of the logic blocks**
  3. **set user options:**
     - » **within the logic blocks**
     - » **in the input/output blocks**
     - » **global reset/clock**
- **"Configuration bit stream" can be loaded under user control:**
  - **All latches are strung together in a shift chain:**

4-LUT Implementation

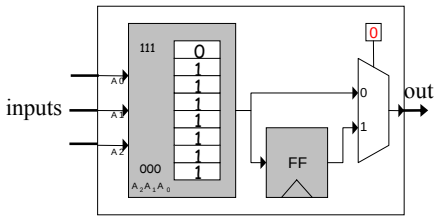


- **n-bit LUT is implemented as a 2^n x 1 memory:**
  - **inputs choose one of 2^n memory locations.**
  - **memory locations (latches) are normally loaded with values from user's configuration bit stream.**
  - **Inputs to mux control are the CLB inputs.**
- **Result is a general purpose "logic gate".**
  - **n-LUT can implement *any* function of n inputs!**

Configuring CLBs

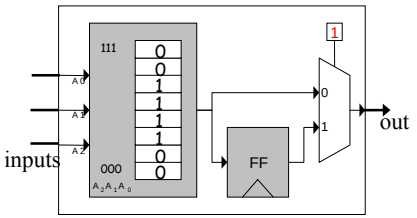


NAND gate in FPGA CLB



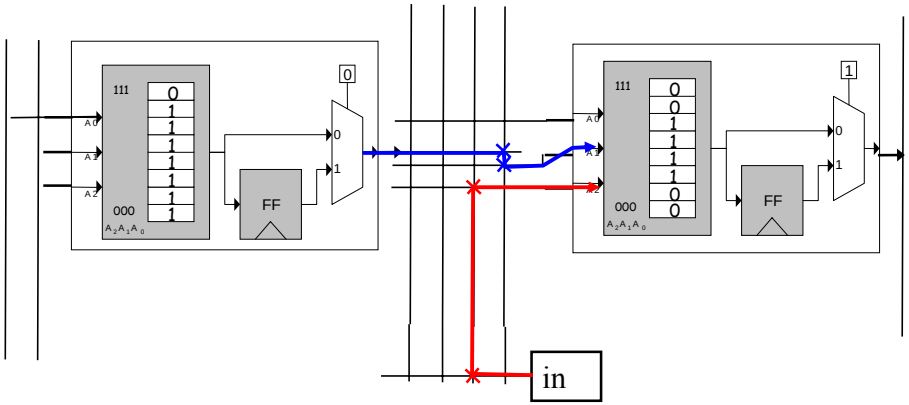
out = ~(A1 A2 A3)

Nextstate bit in FPGA CLB



nextstate = A2 xor A1

Configuring Routes



out = ~(A1 A2 A3)

nextstate = A2 xor A1

Sequential Systems – more examples

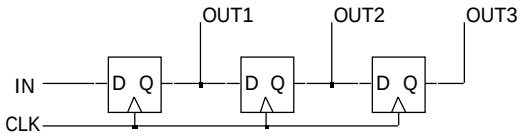
- **Beat the combo lock example to death**
  - **Direct FSM implementation**
  - **Shift register**
    - » **Multiple logic representations**
    - » **gates to LUTs**
- **Up next**
  - **A few quick counter examples**
  - **Another design problem – Ant Brain**

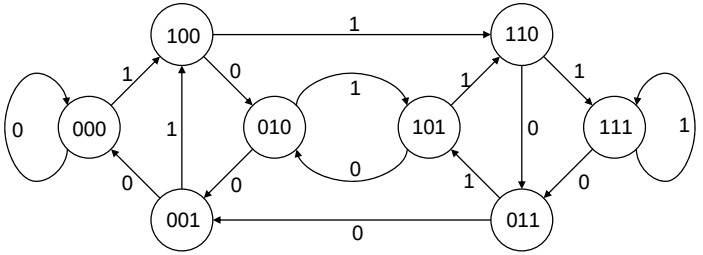
Announcements/Reminders

- **First mid term – Thursday 9/27**
  - **No notes (… to discuss)**
  - **Review materials are in the HW4**
  - **Review session tonight 8-10 642-WALK (9255)**
  - **Trying to make the exams routine**
- **Feel free to approach us with questions…**
- **No discussion Thurs, yes friday**
- **Lab 5 – Where's the music?**
  - **Normal lab lecture on Friday**

Can Any Sequential System be Represented with a State Diagram?

- **Shift Register**
  - **Input value shown on transition arcs**
  - **Output values shown within state node**

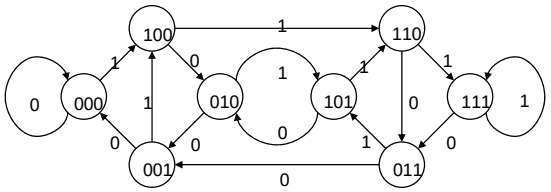




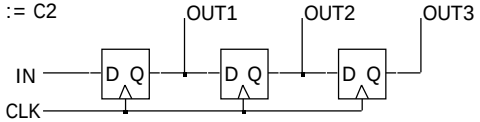
Counter Example

- **Shift Register**
  - **Input determines next state**

In	C1	C2	C3	N1	N2	N3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	0	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	1	0	1
1	0	1	1	1	0	1
1	1	0	0	1	1	0
1	1	0	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	1	1	1

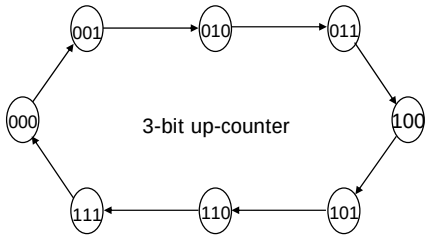


N1 := In
N2 := C1
N3 := C2



Counters are Simple Finite State Machines

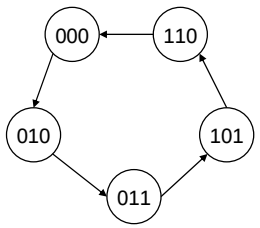
- **Counters**
  - **Proceed thru well-defined state sequence in response to enable**
- **Many types of counters: binary, BCD, Gray-code**
  - **3-bit up-counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, ...**
  - **3-bit down-counter: 111, 110, 101, 100, 011, 010, 001, 000, 111, ...**




```
module binary_upcntr (q, clk)
  inputs     clk;
  outputs    [2:0] q;
  reg        [2:0] q, p;

  always @(q) // Next state
    case (q)
      3'b000: p = 3'b001;
      3'b001: p = 3'b010;
      …
      3'b111: p = 3'b000;
    endcase

  always @(posedge clk) // Update state
    q <= p;

endmodule
```


More Complex Counter Example

- **Complex Counter**
  - **Repeats five states in sequence**
  - **Not a binary number representation**
- **Step 1: Derive the state transition diagram**
  - **Count sequence: 000, 010, 011, 101, 110**
- **Step 2: Derive the state transition table from the state transition diagram**

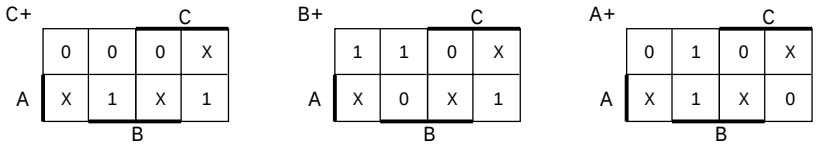


Present State			Next State		
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	–	–	–
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	–	–	–
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	–	–	–

note the don't care conditions that arise from the unused state codes

More Complex Counter Example (cont'd)

- **Step 3: K-maps for Next State Functions**



C+ := A

B+ := B' + A'C'

A+ := BC'

Self-Starting Counters (cont'd)

- **Re-deriving state transition table from don't care assignment**

C+ Karnaugh map:

A \ B	00	01	11	10
0	0	0	0	0
1	1	1	1	1

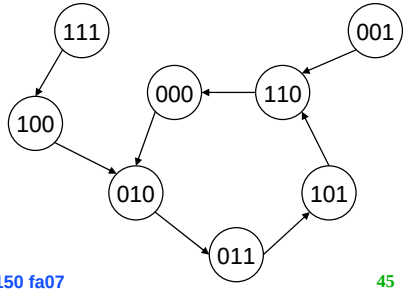
B+ Karnaugh map:

A \ B	00	01	11	10
0	1	1	0	1
1	1	0	0	1

A+ Karnaugh map:

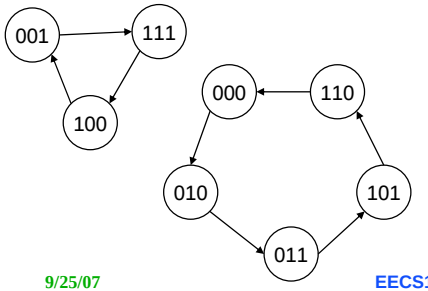
A \ B	00	01	11	10
0	0	1	0	0
1	0	1	0	0

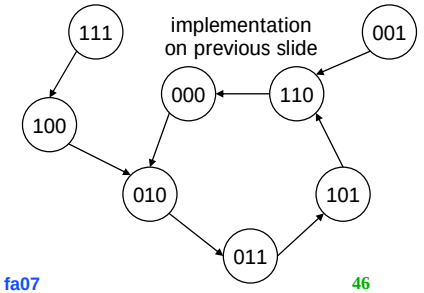
Present State C	B	A	Next State C+	B+	A+
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	1	0	0



Self-Starting Counters

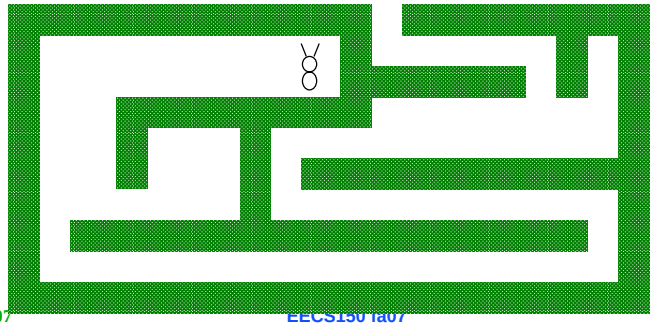
- **Start-up States**
  - **At power-up, counter may be in an unused or invalid state**
  - **Designer must guarantee it (eventually) enters a valid state**
- **Self-starting Solution**
  - **Design counter so that invalid states eventually transition to a valid state**
  - **May limit exploitation of don't cares**



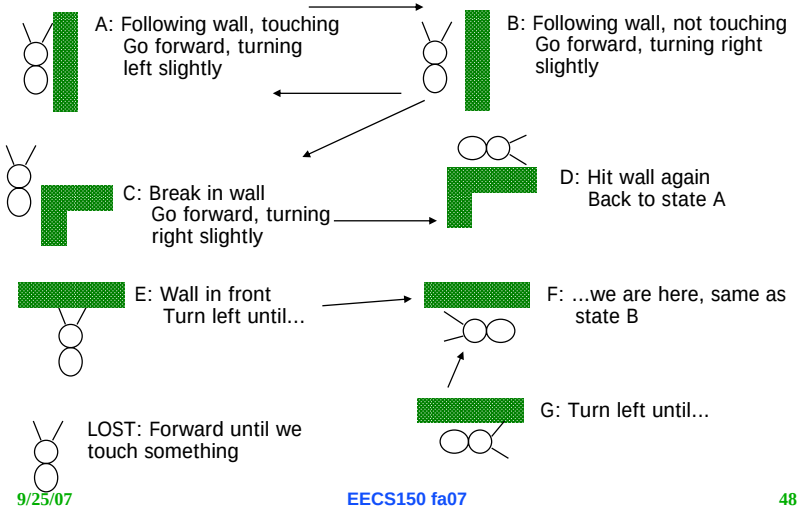


Final Example: Ant Brain (Ward, MIT)

- **Sensors:** **L and R antennae, 1 if in touching wall**
- **Actuators:** **F - forward step, TL/TR - turn left/right slightly**
- **Goal:** **find way out of maze**
- **Strategy:** **keep the wall on the right**

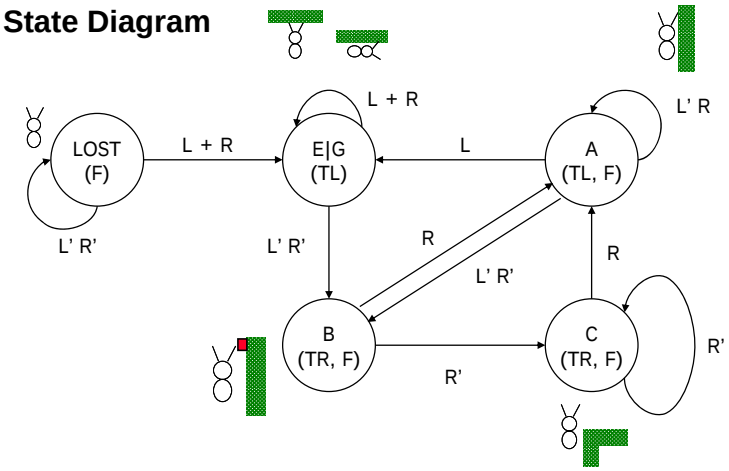


Ant Behavior



Designing an Ant Brain

- **State Diagram**

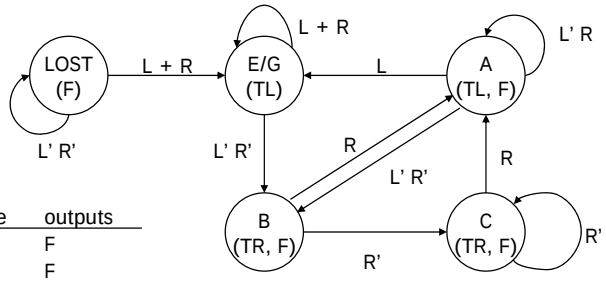


Synthesizing the Ant Brain Circuit

- **Encode States Using a Set of State Variables**
  - **Arbitrary choice - may affect cost, speed**
- **Use Transition Truth Table**
  - **Define next state function for each state variable**
  - **Define output function for each output**
- **Implement next state and output functions using combinational logic**
  - **2-level logic (ROM/PLA/PAL)**
  - **Multi-level logic**
  - **Next state and output functions can be optimized together**

Transition Truth Table

- **Using symbolic states and outputs**



state	L	R	next state	outputs
LOST	0	0	LOST	F
LOST	–	1	E/G	F
LOST	1	–	E/G	F
A	0	0	B	TL, F
A	0	1	A	TL, F
A	1	–	E/G	TL, F
B	–	0	C	TR, F
B	–	1	A	TR, F
…	…	…	…	…

Synthesis

- **5 states : at least 3 state variables required (X, Y, Z)**
  - **State assignment (in this case, arbitrarily chosen)**

LOST - 000
E/G - 001
A - 010
B - 011
C - 100

state X,Y,Z	L	R	next state X', Y', Z'	outputs F	TR	TL
0 0 0	0	0	0 0 0	1	0	0
0 0 0	0	1	0 0 1	1	0	0
…	…	…	…	…		
0 1 0	0	0	0 1 1	1	0	1
0 1 0	0	1	0 1 0	1	0	1
0 1 0	1	0	0 0 1	1	0	1
0 1 0	1	1	0 0 1	1	0	1
0 1 1	0	0	1 0 0	1	1	0
0 1 1	0	1	0 1 0	1	1	0
…	…	…	…	…		

it now remains to synthesize these 6 functions

Synthesis of Next State and Output Functions

state X,Y,Z	inputs L R	next state X^+,Y^+,Z^+	outputs F TR TL
0 0 0	0 0	0 0 0	1 0 0
0 0 0	- 1	0 0 1	1 0 0
0 0 0	1 -	0 0 1	1 0 0
0 0 1	0 0	0 1 1	0 0 1
0 0 1	- 1	0 1 0	0 0 1
0 0 1	1 -	0 1 0	0 0 1
0 1 0	0 0	0 1 1	1 0 1
0 1 0	0 1	0 1 0	1 0 1
0 1 0	1 -	0 0 1	1 0 1
0 1 1	- 0	1 0 0	1 1 0
0 1 1	- 1	0 1 0	1 1 0
1 0 0	- 0	1 0 0	1 1 0
1 0 0	- 1	0 1 0	1 1 0

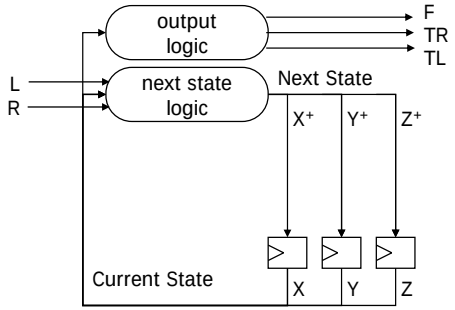
e.g.

TR = X + Y Z

X^+ = X R' + Y Z R' = R' TR

Circuit Implementation

- **Outputs are a function of the current state only - Moore machine**

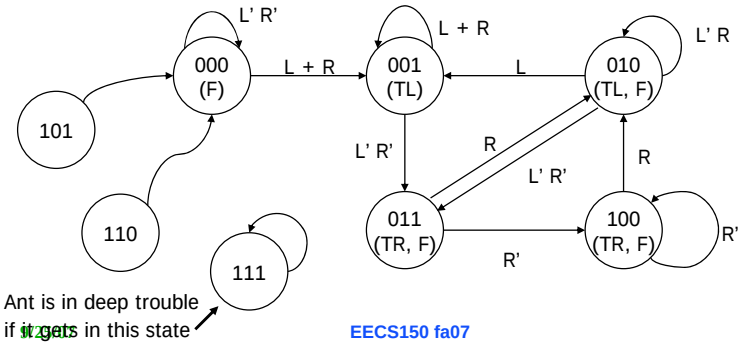


Verilog Sketch

```
module ant_brain (F, TR, TL, L, R)
  inputs     L, R;
  outputs    F, TR, TL;
  reg        X, Y, Z;

  assign F  = function(X, Y, Z, L, R);
  assign TR = function(X, Y, Z, L, R);
  assign TL = function(X, Y, Z, L, R);

  always @(posedge clk)
    begin
      X <= function (X, Y, Z, L, R);
      Y <= function (X, Y, Z, L, R);
      Z <= function (X, Y, Z, L, R);
    end
  endmodule
```


Don't Cares in FSM Synthesis

- **What happens to the "unused" states (101, 110, 111)?**
- **Exploited as don't cares to minimize the logic**
  - **If states can't happen, then don't care what the functions do**
  - **if states do happen, we may be in trouble**

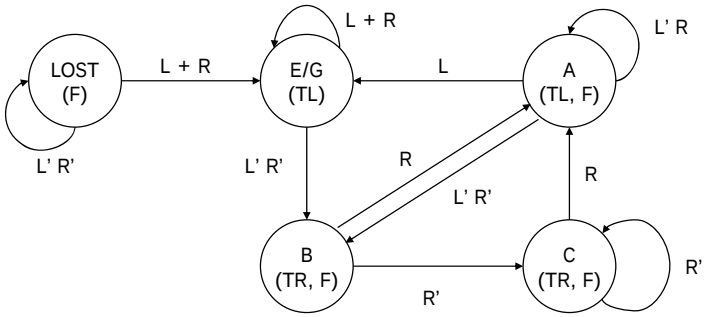


State Minimization

- Fewer states may mean fewer state variables
- High-level synthesis may generate many redundant states
- Two state are equivalent if they are impossible to distinguish from the outputs of the FSM, i. e., for any input sequence the outputs are the same
- Two conditions for two states to be equivalent:
  - 1) Output must be the same in both states
  - 2) Must transition to equivalent states for all input combinations

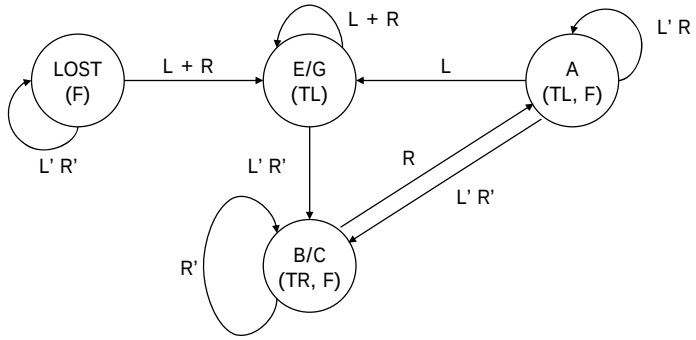
Ant Brain Revisited

- Any equivalent states?



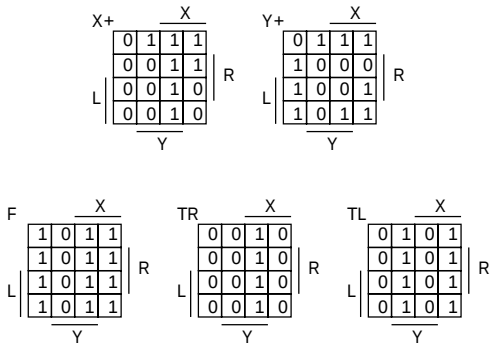
New Improved Brain

- Merge equivalent B and C states
- Behavior is exactly the same as the 5-state brain
- We now need only 2 state variables rather than 3



New Brain Implementation

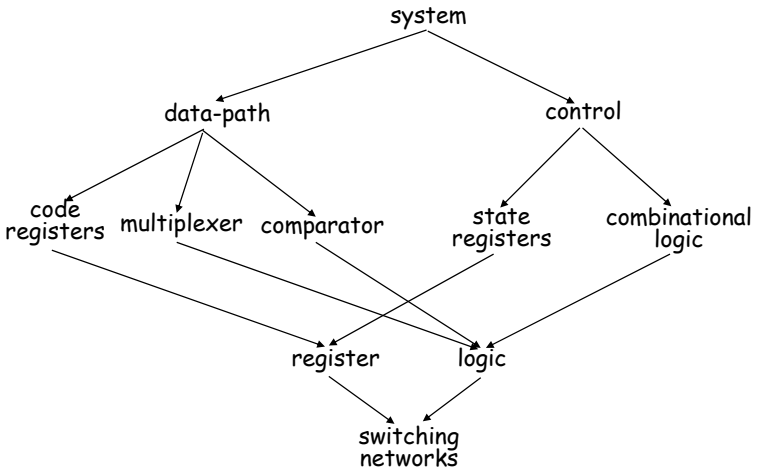
state X,Y	inputs L R	next state X',Y'	outputs F TR TL
0 0	0 0	0 0	1 0 0
0 0	- 1	0 1	1 0 0
0 0	1 -	0 1	1 0 0
0 1	0 0	1 1	0 0 1
0 1	- 1	0 1	0 0 1
0 1	1 -	0 1	0 0 1
1 0	0 0	1 1	1 0 1
1 0	0 1	1 0	1 0 1
1 0	1 -	0 1	1 0 1
1 1	- 0	1 1	1 1 0
1 1	- 1	1 0	1 1 0



Sequential Logic Implementation Summary

- **Models for representing sequential circuits**
  - Abstraction of sequential elements
  - Finite state machines and their state diagrams
  - Inputs/outputs
  - Mealy, Moore, and synchronous Mealy machines
- **Finite state machine design procedure**
  - Deriving state diagram
  - Deriving state transition table
  - Determining next state and output functions
  - Implementing combinational logic

Design hierarchy



Good luck on the Midterm…

Final Word: Blocking Vs Non-Blocking

- **Two types of procedural assignments**
  - Blocking
  - Non-Blocking
- **Why do we need them**
  - Express parallelism (not straight line C)
- **Synchronous system**
  - All flip-flops clock data simultaneously
  - How do we express parallelism in this operation?

A Simple Shift Register

```
reg a, b, c;
always @(posedge clock)
begin
       a = 1;
       b = a;
       c = b;
end
```

Probably not what you want!

```
reg a, b, c;
always @(posedge clock)
       a = 1;
always @(posedge clock)
       b = a;
always @(posedge clock)
       c = b;
```

What order does this run?

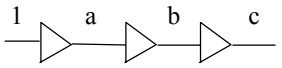
```
reg a, b, c;
always @(posedge clock)
begin
       a <= 1;
       b <= a;
       c <= b;
end
```

This works

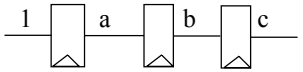
```
reg a, b, c;
always @(posedge clock)
       a <= 1;
always @(posedge clock)
       b <= a;
always @(posedge clock)
       c <= b;
```

This works too…

The Circuit




```
reg a, b, c;
always @(posedge clock)
begin
       a = 1;
       b = a;
       c = b;
end
```




```
reg a, b, c;
always @(posedge clock)
begin
       a <= 1;
       b <= a;
       c <= b;
end
```

Non-Blocking: RHS computed at beginning of execution instance. LHS updated after all events in current instance computed.