Digital design - as we’ve seen it

- System specification (in words)
  - Datapath specification
    - Lec 3.8: Logic
    - Lec 4: HDLs, Labs
  - Comb. logic operations
    - Lec 4, 7: HDLs, Labs
  - Verilog dataflow
    - Lec 2, 3: CMOS, FPGA
  - Gates / LUTs

- ART

- Controller specification
  - FSM generation
    - Lec 6-7: Modeling FSMs
    - Lec 5, 6, 7: FSM
    - STT / STD / Encoding
    - Logic: nextstate/outputs
      - Lec 3, 8: Logic min.
      - Lec 4: HDL, Labs
    - Verilog behavior
      - Lec 2, 3: CMOS, FPGA
      - Gates / LUTs / FF

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!
Scope of CS 150

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

Focus on building systems

More depth than 61C

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 •

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X \cdot Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X'</th>
<th>Y'</th>
<th>X \cdot Y'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[(X \cdot Y) + (X' \cdot Y') = X = Y\]

X, Y are Boolean algebra variables

An algebraic structure

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

1. set B contains at least two elements, a, b, such that \( a \neq b \)
2. closure: \( a + b \) is in B \( a \cdot b \) is in B
3. commutativity: \( a + b = b + a \) \( a \cdot b = b \cdot a \)
4. associativity: \( a + (b + c) = (a + b) + c \) \( a \cdot (b \cdot c) = (a \cdot b) \cdot c \)
5. identity: \( a + 0 = a \) \( a \cdot 1 = a \)
6. distributivity: \( a + (b \cdot c) = (a + b) \cdot (a + c) \) \( a \cdot (b + c) = (a \cdot b) + (a \cdot c) \)
7. complementarity: \( a + a' = 1 \) \( a \cdot a' = 0 \)
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)

  ![Timing Window Diagram]

  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. \( X + 0 = X \)
  2. \( X + 1 = 1 \)
  3. \( X + X = X \)
  4. \( (X')' = X \)
  5. \( X + X' = 1 \)
  6. \( X + Y = Y + X \)
  7. \( (X + Y) + Z = X + (Y + Z) \)

- **Null**
  1. \( X \cdot 1 = X \)
  2. \( X \cdot 0 = 0 \)
  3. \( X \cdot X = X \)
  4. \( X' \cdot X = 0 \)

- **Idempotency**
  1. \( X + X = X \)
  2. \( X \cdot X = X \)

- **Complementarity**
  1. \( X + X' = 1 \)
  2. \( X \cdot X' = 0 \)

- **Commutativity**
  3. \( X \cdot Y = Y \cdot X \)
  4. \( (X + Y) \cdot (X' + Z) = X \cdot Z + X' \cdot Y \)

- **Associativity**
  1. \( X + (Y + Z) = (X + Y) + Z \)
  2. \( (X \cdot Y) \cdot Z = X \cdot (Y \cdot Z) \)

- **Distributivity**
  3. \( (X + Y + Z) = (X \cdot Y) + (X + Z) \)
  4. \( (X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) = (X + Y) \cdot (Y + Z) \)

- **Axioms and theorems of Boolean algebra (cont’d)**

  - **Distribution:**
    1. \( X + (Y + Z) = (X + Y) + (X + Z) \)

  - **Uniting:**
    1. \( X \cdot (Y + Z) = X \cdot Y + X \cdot Z \)

  - **Absorption:**
    1. \( X \cdot X = X \)
    2. \( X \cdot X = X \)

  - **Factoring:**
    1. \( X \cdot X + Y = X \)
    2. \( X \cdot Y = X \)

  - **Concensus:**
    1. \( X \cdot X + X = X \)
    2. \( X \cdot Y + X' = X \cdot Z \)

Axioms and theorems of Boolean algebra (cont’)

- **de Morgan’s:**
  1. \( (X + Y + ...)' = X' \cdot Y' + ... \)
  2. \( (X \cdot Y \cdot ...)' = X' + Y' + ... \)

- **generalized de Morgan’s:**
  1. \( f(X_1, X_2, ..., X_n, 0, 1, +, *) = f(X_1', X_2', ..., X_n', 1, 0, +, *) \)

  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


table

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
<th>mux</th>
<th>open/closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>S2</td>
<td>C2</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S2</td>
<td>C2</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S2</td>
<td>C2</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S1</td>
<td>C1</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>S3</td>
<td>C3</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>S3</td>
<td>C3</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>S3</td>
<td>C3</td>
<td>closed</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>OPEN</td>
<td></td>
<td>open</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>ERR</td>
<td></td>
<td>closed</td>
</tr>
</tbody>
</table>

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, ERR = 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

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
<th>nstate</th>
<th>mux</th>
<th>open</th>
<th>Next state and output logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r)</td>
<td>(n)</td>
<td>(e)</td>
<td>(s2, s1, s0)</td>
<td>(n2, n1, n0)</td>
<td>(m1, m0)</td>
<td>(o)</td>
<td>(n2 n1 n0):</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>001</td>
<td>--</td>
<td>0</td>
<td>n2 = ¬r (n e s1 s0 + s2)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>001</td>
<td>01</td>
<td>n1 = ¬r (n e s0 + e s1 + ¬n s1 + s2)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>000</td>
<td>01</td>
<td>n0 = r + s2 + n e s1 + ¬n s0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>001</td>
<td>010</td>
<td>01</td>
<td>mux outputs (m1, m0):</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>010</td>
<td>010</td>
<td>10</td>
<td>m1 = s1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>000</td>
<td>10</td>
<td>m0 = s0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>011</td>
<td>011</td>
<td>11</td>
<td>open (o):</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>011</td>
<td>011</td>
<td>11</td>
<td>o = s2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>000</td>
<td>11</td>
<td>Take advantage of DCs!</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>011</td>
<td>111</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>111</td>
<td>111</td>
<td>--</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>000</td>
<td>000</td>
<td>--</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>--</td>
<td>--</td>
<td>How do we get these:</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>101</td>
<td>--</td>
<td>--</td>
<td>• K-maps?</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>110</td>
<td>--</td>
<td>--</td>
<td>• Tools</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>➢ Espresso</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>➢ Synplicity</td>
<td></td>
</tr>
</tbody>
</table>

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)

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

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

<table>
<thead>
<tr>
<th>reset</th>
<th>new</th>
<th>equal</th>
<th>state</th>
<th>rstate</th>
<th>mux</th>
<th>open</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1000</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0010</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0010</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0010</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0010</td>
<td>0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0010</td>
<td>0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0010</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0001</td>
<td>1</td>
<td>--</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0000</td>
<td>0</td>
<td>--</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0000</td>
<td>0</td>
<td>--</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>11--</td>
<td>-----</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>1-1</td>
<td>-----</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>11--</td>
<td>-----</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>1-1</td>
<td>-----</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-11</td>
<td>-----</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-11</td>
<td>-----</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-11</td>
<td>-----</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-11</td>
<td>-----</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

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

Mux control:
- CI = 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

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

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
**LUT as general logic gate**

**Example: 4-lut**

- 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**

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>0000</th>
<th>F(0,0,0,0)</th>
<th>store in 1st latch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0001</td>
<td>F(0,0,0,1)</td>
<td>store in 2nd latch</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>F(0,0,1,0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0011</td>
<td>F(0,0,1,1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0101</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0110</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0111</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1001</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1010</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1011</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1101</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1110</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Implements any function of 2 inputs.

**Inputs**

- How many of these are there?
- How many functions of n inputs?

**User Programmability**

- Latch-based (Xilinx, Altera, …)

**Example:**

- 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 $2^n \times 1$ 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

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

<table>
<thead>
<tr>
<th>In</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Counter Example (cont’d)

- **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**

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**

More Complex Counter Example (cont’d)

- **Step 3: K-maps for Next State Functions**
**Self-Starting Counters (cont’d)**

- Re-deriving state transition table from don’t care assignment

```
<table>
<thead>
<tr>
<th>Present State</th>
<th>C+</th>
<th>B+</th>
<th>A+</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>11</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Next State</th>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**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

```
A: Following wall, touching Go forward, turning left slightly
B: Following wall, not touching Go forward, turning right slightly
C: Break in wall Go forward, turning right slightly
D: Hit wall again Back to state A
E: Wall in front Turn left until...
F: ...we are here, same as state B
G: Turn left until...
LOST: Forward until we touch something
```
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

<table>
<thead>
<tr>
<th>state</th>
<th>L</th>
<th>R</th>
<th>next state</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOST</td>
<td>0</td>
<td>0</td>
<td>LOST</td>
<td>F</td>
</tr>
<tr>
<td>LOST</td>
<td>0</td>
<td>1</td>
<td>E/G</td>
<td>F</td>
</tr>
<tr>
<td>LOST</td>
<td>1</td>
<td>0</td>
<td>E/G</td>
<td>F</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>B</td>
<td>TR, F</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>TL, F</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>E/G</td>
<td>TL, F</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>C</td>
<td>TR, F</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>A</td>
<td>TR, F</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Synthesis

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

<table>
<thead>
<tr>
<th>state</th>
<th>L</th>
<th>R</th>
<th>next state</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>1 0 0</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>1 0 1</td>
</tr>
<tr>
<td>01 0 1</td>
<td>0</td>
<td>0</td>
<td>01 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>01 0 1</td>
<td>1</td>
<td>0</td>
<td>00 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>01 1 0</td>
<td>0</td>
<td>0</td>
<td>10 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>01 1 0</td>
<td>1</td>
<td>0</td>
<td>01 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Synthesis of Next State and Output Functions

<table>
<thead>
<tr>
<th>State Inputs</th>
<th>Next State Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0 0</td>
<td>000 1 0 0</td>
</tr>
<tr>
<td>000 0 1</td>
<td>001 1 0 0</td>
</tr>
<tr>
<td>000 1 0</td>
<td>001 1 0 0</td>
</tr>
<tr>
<td>001 0 0</td>
<td>011 0 0 1</td>
</tr>
<tr>
<td>001 0 1</td>
<td>010 0 0 1</td>
</tr>
<tr>
<td>001 1 0</td>
<td>010 0 0 1</td>
</tr>
<tr>
<td>010 0 0</td>
<td>011 1 0 1</td>
</tr>
<tr>
<td>010 0 1</td>
<td>010 1 0 1</td>
</tr>
<tr>
<td>010 1 0</td>
<td>001 1 0 1</td>
</tr>
<tr>
<td>011 0 0</td>
<td>100 1 1 0</td>
</tr>
<tr>
<td>011 0 1</td>
<td>101 1 1 0</td>
</tr>
<tr>
<td>011 1 0</td>
<td>100 1 1 0</td>
</tr>
<tr>
<td>100 0 0</td>
<td>100 1 1 0</td>
</tr>
<tr>
<td>100 1 0</td>
<td>101 1 1 0</td>
</tr>
</tbody>
</table>

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

Circuit Implementation

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

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 states 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

<table>
<thead>
<tr>
<th>state inputs</th>
<th>next state outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y</td>
<td>L, R</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>-1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>-1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>-1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ X \quad Y \quad X' \quad Y' \quad X' \quad X' \quad X' \quad X' \]
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

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?

Good luck on the Midterm...
A Simple Shift Register

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

`Probably not what you want!`

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

`What order does this run?`

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

`This works`

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

`This works too…`

The Circuit

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