First Steps

- Simple binary calculator
- Basic circuit elements
- Specifying circuits with hardware description languages
- Using simulation to verify circuit operation
- Representing numeric values in circuits
- Logic equations and combinational circuits

Jon Turner
A Simple Binary Calculator

- Three operations
  - clear stored value
  - load new value
  - add to stored value

- Components
  - flip flops store information
  - multiplexor selects 1-of-2 inputs
  - adder computes sum
  - wires that connect components
Basic Circuit Elements

**AND gate**
\[ \begin{array}{c}
A \\
B \\
\end{array} \quad \begin{array}{c}
\rightarrow \\
A \cdot B \\
\end{array} \]

**OR gate**
\[ \begin{array}{c}
A \\
B \\
\end{array} \quad \begin{array}{c}
\rightarrow \\
A + B \\
\end{array} \]

**Inverter**
\[ \begin{array}{c}
A \\
\end{array} \quad \begin{array}{c}
\rightarrow \\
A' \\
\end{array} \]

**Truth table for logic gates**

<table>
<thead>
<tr>
<th>AB</th>
<th>A·B</th>
<th>A+B</th>
<th>A'</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**2:1 multiplexor**
\[ \begin{array}{c}
A \\
B \\
\end{array} \quad \begin{array}{c}
0 \quad 1 \\
\hline
1 \quad 0 \\
\end{array} \quad \begin{array}{c}
C \\
\end{array} \quad \begin{array}{c}
X = C' \cdot A + C \cdot B \\
\end{array} \]

**D flip-flop**
\[ \begin{array}{c}
D \\
\rightarrow \end{array} \quad \begin{array}{c}
Q \\
C \quad \text{store value of } D \text{ on rising clock edge} \\
\end{array} \]

is shorthand for

is shorthand for

for gates and mux, outputs change as inputs change for flip flop, output changes only at clock edge
VHDL Specification of Calculator

entity calculator is port (  
  clk: in std_logic;  
clear, load, add: in std_logic;  -- operation signals  
dIn: in std_logic_vector(15 downto 0);  -- input data  
result: out std_logic_vector(15 downto 0));  -- output result  
end calculator;  
architecture al of calculator is  
signal dReg: std_logic_vector(15 downto 0);  
begin  
  process (clk) begin  
    if rising_edge(clk) then  
      if clear = '1' then  
        dReg <= x"0000";  
      elsif load = '1' then  
        dReg <= dIn;  
      elsif add = '1' then  
        dReg <= dReg + dIn;  
      end if;  
    end if;  
  end process;  
  result <= dReg;  
end al;
Simulation of Binary Calculator
Exercises

1. Re-draw this circuit showing all individual gates. Note that the $C$ input has no numeric label, meaning it is a single bit signal.

2. Draw a diagram of the calculator circuit, extended to include a flip operation that complements all bits in $dReg$ (that is, it converts 0s to 1s and 1s to 0s).
Binary and Hexadecimal

- Computers represent data in binary form because circuits for binary data are simple and reliable
- Binary number representation
  \[110.01 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} (= 6.25)\]
- Hexadecimal (base 16) provide more convenient way for people to write binary numbers
  \[\text{hexadecimal conversion}\]
  \[
  \begin{array}{ccc}
  0000 = 0 & 0101 = 5 & 1010 = 10 = a \\
  0001 = 1 & 0110 = 6 & 1011 = 12 = b \\
  0010 = 2 & 0111 = 7 & 1100 = 12 = c \\
  0011 = 3 & 1000 = 8 & 1101 = 13 = d \\
  0100 = 4 & 1001 = 9 & 1110 = 14 = e \\
  & 1111 = 15 = f \\
  \end{array}
  \]
Modular and Signed Arithmetic

- Computers use modular arithmetic in which values wrap around circularly
  - to add \( A + B \), start at position for \( A \) and then count clockwise \( B \) positions
  - think of modular arithmetic like “clock arithmetic”

- Finite representation can cause “overflow” yielding incorrect results
- Associating certain bit patterns with negative values yields signed arithmetic (called 2s complement)
- To negate, subtract value from 0
  - or, find rightmost 1 & flip all bits to its left
- Different error conditions for signed vs. unsigned values
Exercises

1. What is the decimal equivalent of 1011.011?
2. Rewrite 10110110101101 in hexadecimal.
3. What is the decimal equivalent of 2a5? Don’t use a calculator.
4. Suppose a5 represents a number in an 8 bit 2s-complement number system. Is this value positive or negative? What is the corresponding value with the opposite sign (write it in hex)? Check your answer by adding the two hex values together and confirming that the eight bit result is zero.
Elementary Binary Logic Functions

- Digital circuits represent information using two voltage levels
  - *binary variables* are used to denote these values
  - by convention, the values are called “1” and “0” and we usually think of them as meaning “True” and “False”

- Functions of binary variables called *logic functions*
  - $\text{AND}(A,B) = 1$ if $A=1$ and $B=1$, else it is zero
    - AND is generally written in shorthand $A \cdot B$ (or $A&B$ or $A\wedge B$)
  - $\text{OR}(A,B) = 1$ if $A=1$ or $B=1$, else it is zero
    - OR is generally written in shorthand form $A+B$ (or $A|B$ or $A\vee B$)
  - $\text{NOT}(A) = 1$ if $A=0$ else it is zero
    - NOT is generally written in shorthand form $\bar{A}$ (or $\neg A$ or $A'$)

- AND, OR and NOT can be used to express all other logic functions
Logic Equations and Circuits

- Boolean algebra defines rules for manipulating symbolic binary logic expressions
  - A symbolic binary logic expression consists of binary variables and operators AND, OR, and NOT (e.g. \( A + B \cdot C' \))

- The possible values for any Boolean expression can be tabulated in a *truth table*

- Can define circuit for expression by combining gates

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>B( \cdot C' )</th>
<th>A + B( \cdot C' )</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</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>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
From Logic Equations to Circuits

- Any Boolean expression can be converted to a logic circuit made up of AND, OR and NOT gates.
  step 1: add parentheses to expression to fully define order of operations - \( A + (B \cdot (C')) \)
  step 2: create gate for “last” operation in expression;
    gate’s output is value of expression; gate’s inputs are sub-expressions combined by operation
    \[ A \quad \rightarrow \quad \begin{array}{c} (B \cdot (C')) \end{array} \quad \rightarrow \quad A + B \cdot C' \]
  step 3: repeat for sub-expressions, continue until done
- Number of simple gates to implement expression equals number of operations in expression
  » simpler expression yields less expensive circuit
  » Boolean algebra provides rules for simplifying logic
Exercises

1. Write logic equations for each of the outputs of the circuit at right.
2. Write a logic equation for the function defined by the truth table shown below.

<table>
<thead>
<tr>
<th>ABC</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
</tr>
</tbody>
</table>

3. Draw logic diagrams for the following three expressions.

\[ AB + C(B' + D) \quad ABC(C' + D)(B' + D') \quad (A + B')(A' + C + D)'(B + D) \]
Simplification of Logic Equations

\[ F = X'YZ + X'YZ' + XZ \]

\[ F = X'Y(Z + Z') + XZ \]

\[ F = X'Y \cdot 1 + XZ = X'Y + XZ \]
Identities for Simplifying Circuits

- Elementary identities
  - \( X + X' = 1 \)
  - \( X \cdot X' = 0 \)

- Distributive laws
  - \( X \cdot (Y + Z) = X \cdot Y + X \cdot Z \)
  - \( X + (Y \cdot Z) = (X + Y) \cdot (X + Z) \)

- DeMorgans laws
  - \((X + Y)' = X' \cdot Y'\)
  - \((X \cdot Y)' = X' + Y'\)

- Other handy identities
  - \( A + A \cdot B = A \)
  - \( A + A' \cdot B = A + B \)

<table>
<thead>
<tr>
<th>XY</th>
<th>((X + Y))'</th>
<th>((X + Y))'</th>
<th>(X' \cdot Y')</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
What Makes a Circuit Combinational?

- In combinational circuits, output values depend only on current input values (not previous inputs)
  - non-combinational circuits use feedback cycles to implement storage
  - circuits with no cycles are guaranteed to be combinational
- Since outputs depend only on current inputs, they can be specified by truth table or logic equations
Exercises

1. Verify the equation

\[ AB' + A'CD' + A'B'D + A'B'CD' = B' + A'CD' \]

using a truth table

2. Simplify the expression \( AB' + A'CD' + A'B'D + A'B'CD' \) using Boolean algebra.

3. Simplify the expression \( A(B' + A'CD') + A'B'(D + CD') \) using Boolean algebra.

4. Write two logic equations that define the same circuit as the VHDL code fragment shown below. Assume \( A, B, C \) and \( D \) are single bit signals

```vhdl
if A=1 then
    X <= B or not C; Y <= not B;
elif D=0 then
    X <= B and C; Y <= not C;
else
    X <= 1; Y <= B;
end if
```