Chapter 3 – Part 2
Combinational Logic Design

Functional Block: Half-Adder

- A 2-input, 1-bit width binary adder that produces the following values:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- A half adder adds two bits to produce a two-bit sum.
- The sum is expressed as a sum bit, S and a carry bit, C.
- The half adder can be specified as a combined truth table for S and C:

$X \oplus Y$ for S and $X \cdot Y$ for C.

Logic Simplification: Half-Adder

- The K-Map for S, C is:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- This is a pretty trivial map! By inspection:

$S = X \cdot Y + X \cdot Y = X \oplus Y$

$S = (X + Y) \cdot (X + Y)$

- And

$C = X \cdot Y$

$C = ((X \cdot Y))$

- These equations lead to several implementations.

Five Implementations: Half-Adder

- We can derive following sets of equations for a half adder:

(a) $S = X \cdot Y + X \cdot Y$  
(b) $S = (X + Y) \cdot (X + Y)$

(c) $S = (C + X \cdot Y)$

(d) $S = (X + Y) \cdot C$

(e) $S = (X + Y) \cdot C + X \cdot Y$

- (a), (b), and (e) are SOP, POS, and XOR implementations for S.
- In (c), the C function is used as a term in the AND-NOR implementation of S, and in (d), the C function is used in a POS term for S.

Implementations: Half-Adder

- The most common half adder implementation (e) is:

$S = X \oplus Y$

$C = (X \cdot Y)$

A NAND only implementation (equivalent to equation d) is:

$S = (X + Y) \cdot C$

$C = ((X \cdot Y))$
Functional Block: Full-Adder

- A full adder is similar to a half adder, but includes a carry-in bit from lower stages. Like the half-adder, it computes a sum bit, S, and a carry bit, C.
- For a carry-in (Z) of 0, it is the same as the half-adder:
  \[ S = X \oplus Y, \quad C = (X \cdot Y) + Z \]
- For a carry-in (Z) of 1:
  \[ S = X \oplus Y \oplus Z, \quad C = (X \cdot Y) + (X + Y) \cdot Z \]

Design: Full-Adder

- Full-Adder Function Table:
  \[
  \begin{array}{cccccc}
    X & Y & Z & C & S \\
    0 & 0 & 0 & 0 & 0 \\
    0 & 0 & 1 & 0 & 1 \\
    0 & 1 & 0 & 0 & 1 \\
    0 & 1 & 1 & 1 & 0 \\
    1 & 0 & 0 & 0 & 1 \\
    1 & 0 & 1 & 1 & 0 \\
    1 & 1 & 0 & 1 & 0 \\
    1 & 1 & 1 & 1 & 1 \\
  \end{array}
  \]

- Full-Adder K-Map:

  \[
  \begin{array}{ccc}
    X & Y & Z \\
    0 & 0 & 0 \\
    0 & 1 & 1 \\
    1 & 0 & 0 \\
    1 & 1 & 1 \\
  \end{array}
  \]

Implementation: Full Adder

- Full Adder Schematic:

Parallel Binary Adders

- To add more than one bit, we "bundle" sets of logical signals together and build devices that operate on the whole set in parallel.
- Example: 4-bit binary adder:
  - Adds an input vector "A(3,0)" to "B(3,0)" to get a sum S(3,0) thus:
    \[
    \begin{array}{cccccc}
      \text{Description} & \text{Subscript} & \text{Name} \\
      \text{Input Carry} & C_i & C_i \\
      \text{Augend} & A & A \\
      \text{Addend} & B & B \\
      \text{Sum} & S & S \\
      \text{Output Carry} & C_o & C_o \\
    \end{array}
    \]
  - Note: the carry out of Stage i becomes the carry in of Stage i+1.

4-bit Ripple-Carry Binary Adder

- A four-bit Full Adder made from four 1-bit Full Adders:
  \[
    \begin{array}{cccc}
      S(4) & S(3) & S(2) & S(1) \\
      C_{i+1} & C_i & C_{i-1} & C_{i-2} \\
    \end{array}
    \]

- Here FA is a Full-Adder from before:
Carry Propagation & Delay

- One problem with the addition of binary numbers is the length of time to propagate the ripple carry from the least significant bit to the most significant bit.
- The gate-level propagation path for a 4-bit ripple carry adder of the last example:
  
  ![Diagram of 4-bit ripple carry adder]

  Note: The "long path" is from A(0) or B(0) through the network to either C(4) or S(3).

Carry Look-Ahead

- Given Stage i from a Full Adder, we know that there will be a *carry generated* when \( A_i = B_i = 1 \), whether or not there is a carry-in.
- Alternately, there will be a *carry propagated* if the "HalfSum" is "1" and a carry-in, \( C_i \), occurs.
- These two signal conditions are called Generate denoted as \( G_i \) and Propagate denoted as \( P_i \) respectively and are shown here: \( \Rightarrow \)

Carry Look-Ahead (Continued)

- By defining the equations for the Full Adder in terms of the \( P_i \) and \( G_i \), we have:
  
  \[
  P_i = A_i \oplus B_i \quad G_i = A_i B_i
  \]
  
  And the output sum \( S(i) \) and carry \( C(i+1) \) is defined as:
  
  \[
  S_i = P_i \oplus C_i \quad C_{i+1} = G_i + P_i C_i
  \]
  
  Starting the stage numbering at zero, we have:
  
  \[
  C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0
  \]
  
  where \( C_4 \) is a carry in to the least significant bit.

Group Carry Look-Ahead Logic

- Figure 3-28 in the text shows how to implement a carry look-ahead circuit for four bits. This could be extended to more than four bits. In practice, though, it becomes more difficult to implement this over more than a few bits. The concept can be extended another level by considering a Group Generate \( (G_{0-3}) \) and Group Propagate \( (P_{0-3}) \) logic condition:
  
  \[
  G_{0-3} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 P_0 G_0
  \]

  Using these two equations:
  
  \[
  C_4 = G_{0-3} + P_{0-3} C_0
  \]

  Thus, it is possible to have four 4-bit adders use one of the same carry look-ahead circuits to add 16 bits!

Complements

- Subtraction of numbers requires a different algorithm from that for addition
- Adding the complement of a number is equivalent to subtraction
- We will discuss two complements:
  - Diminished Radix Complement
  - Radix Complement
- Subtraction will be done by adding the complement of the subtrahend
**Diminished Radix Complement**

- Given a number $N$ in Base $r$ having $n$ digits, the $(r - 1)'s$-complement (called the Diminished Radix Complement) is defined as: $(r^n - 1) - N$
- Example:
  - For $r = 10$, $N = 1234_{10}$, $n = 4$ (4 digits), we have:
    
    $r^n - 1 = 10,000 - 1 = 9999_{10}$
  - The 9's complement of 1234 is then:
    
    $9999_{10} - 1234_{10} = 8765_{10}$

**Binary 1's Complement**

- For $r = 2$, $N = 01110011_2$, $n = 8$ (8 digits):
  
  $(r^n - 1) = 256 - 1 = 255_{10}$ or $11111111_2$
  - The 1's complement of 01110011, is then:
    
    $11111111_{10}$
    
    $01100110_2$
  - Since the $2^n - 1$ factor consists of all 1's and since $1 - 0 = 1$ and $1 - 1 = 0$, the one's complement is obtained by complementing each individual bit (bitwise NOT).

**Radix Complement**

- Given a number $N$ in Base $r$ having $n$ digits, the $r$'s complement (called the radix complement) is defined as:
  
  - $r^n - N$ for $N \neq 0$
  - $0$ for $N = 0$
- Example:
  - For $r = 10$, $N = 1234_{10}$, $n = 4$ (4 digits), we have:
    
    $r^n = 10,000_{10}$
  - The 10's complement of 1234 is then
    
    $10,000_{10} - 1234_{10} = 8766_{10}$
  - or $8765 + 1$ (9's complement plus 1)

**Binary 2's Complement**

- For $r = 2$, $N = 01110011_2$, $n = 8$ (8 digits), we have:
  
  $(r^n) = 256_{10}$ or $100000000_2$
  - The 2's complement of 01110011 is then:
    
    $100000000_{10}$
    
    $01100110_2$
  - Note the result is the 1's complement plus 1

**Alternate 2's Complement**

- Given: an $n$-bit binary number, beginning at the right and proceeding left:
  
  - Copy all least significant 0's
  - Copy the first 1
  - Complement all bits thereafter.
  - 2's Complement Example:
    
    $10010 100$
    
    - Copy underlined bits:
      
      $100$
    - and complement bits to the left:
      
      $0110100$

**Subtraction with Radix Complements**

- For $n$-digit, unsigned numbers $M$ and $N$, find $M - N$ in base $r$:
  
  - Add the $r$'s complement of the subtrahend $N$ to the minuend $M$:
    
    $M + (r^n - N) = M + r^n$
  - If $M \geq N$, the sum produces end carry $r^n$ which is discarded; from above, $M - N$ remains.
  - If $M < N$, the sum does not produce an end carry and, from above, is equal to $r^n - (N - M)$, the $r$'s complement of $(N - M)$.
  - To obtain the result $-(N - M)$, take the $r$'s complement of the sum and place a – in front.
### Unsigned 10’s Complement Subtraction Example 1

Find $543_{10} - 123_{10}$

$543_{10}$

$10^{'s\, comp}$

$877_{10}$

$420_{10}$

The carry of 1 indicates that no correction of the result is required.

### Unsigned 10’s Complement Subtraction Example 2

Find $123_{10} - 543_{10}$

$123_{10}$

$10^{'s\, comp}$

$457_{10}$

$580_{10}$

The carry of 0 indicates that a correction of the result is required.

Result = $-(520)$

### Unsigned 2’s Complement Subtraction Example 1

Find $01010100_2 - 01000011_2$

$01010100_2$

$10101000_2$

$10111100_2$

$00010000_2$

The end-around carry occurs.

### Unsigned 2’s Complement Subtraction Example 2

Find $01000011_2 - 01010100_2$

$01000011_2$

$10100001_2$

$10110111_2$

$00010001_2$

The carry of 0 indicates that a correction of the result is required.

Result = $-(00010001)$

### Subtraction with Diminished Radix Complement

For n-digit, unsigned numbers $M$ and $N$, find $M - N$ in base $r$:

1. Add the $(r-1)$’s complement of the subtrahend $N$ to the minuend $M$:
   
   $M + (r^n - 1 - N) = M - N + r^n - 1$

2. If $M \geq N$, the result is excess by $r^n - 1$. The end carry $r^n$ when discarded removes $r^n$, leaving a result short by 1. To fix this shortage, whenever and end carry occurs we add 1 in the LSB position. This is called end-around carry.

3. If $M < N$, the sum does not produce an end carry and, from above, is equal to $r^n - 1 - (N - M)$, the $r^n$’s complement of $(N - M)$.

4. To obtain the result $(N-M)$, take the $r^n$’s complement of the sum and place a minus in front.

### Unsigned 1’s Complement Subtraction Example 1

Find $01010100_2 - 01000011_2$

$01010100_2$

$10101000_2$

$10111100_2$

$00010000_2$

$1\_1's\, comp$

The end-around carry occurs.
Unsigned 1’s Complement Subtraction Example 2

- Find \( \overline{01000011}_2 - \overline{01010100}_2 \)
  
  \[
  \begin{array}{c}
  01000011 \\
  \hline
  - 01010100 \\
  \hline
  11011110 \\
  \hline
  00010001
  \end{array}
  \]
  
  - The carry of 0 indicates that a correction of the result is required.
  - Result = \( -00010001 \)

Signed Integers

- Positive numbers and zero can be represented by unsigned \( n \)-digit, radix \( r \) numbers. We need a representation for negative numbers.
- To represent a sign (+ or –) we need exactly one more bit of information (1 binary digit gives \( 2^1 = 2 \) elements which is exactly what is needed).
- Since computers use binary numbers, by convention, (and, for convenience), the most significant bit is interpreted as a sign bit:
  
  \[
  s = \begin{cases} 0 & \text{for Positive numbers} \\ 1 & \text{for Negative numbers} \end{cases}
  \]
  
  and \( a_i = 0 \) or 1 represent in some form the magnitude.

Signed Integer Representations

- Signed-Magnitude – here the \( n – 1 \) digits are interpreted as a positive magnitude.
- Signed-Complement – here the digits are interpreted as the rest of the complement of the number. There are two possibilities here:
  - Signed One’s Complement –
    * Uses 1’s Complement Arithmetic
  - Signed Two’s Complement –
    * Use 2’s Complement Arithmetic

Signed Integer Representation Example

- \( r = 2, n = 3 \)

<table>
<thead>
<tr>
<th>Number</th>
<th>Sign-Mag</th>
<th>1’s Comp</th>
<th>2’s Comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td>011</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>+2</td>
<td>010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>+1</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>+0</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>-0</td>
<td>_111</td>
<td>_111</td>
<td>_111</td>
</tr>
<tr>
<td>-1</td>
<td>010</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>-2</td>
<td>111</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>-3</td>
<td>111</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>-4</td>
<td>111</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

Signed-Magnitude Arithmetic

- Addition:
  * If signs are the same:
    1. Add the magnitudes.
    2. Check for overflow (a carry into the sign bit).
    3. The sign of the result is the same.
  * If the signs differ:
    1. Subtract the subtrahend from the minuend
    2. If a borrow occurs, take the two’s complement of result
    3. Overflow will never occur.
- Subtraction:
  * Complement the sign bit of the number you are subtracting and follow the rules for addition.

Signed-Complement Arithmetic

- Addition:
  1. Add the numbers including the sign bits, discarding a carry out of the sign bits (2’s Complement), or using an end-around carry (1’s Complement).
  2. If the sign bits were the same for both numbers and the sign of the result is different, an overflow has occurred.
  3. The sign of the result is computed in step 1.
- Subtraction:
  Form the complement of the number you are subtracting and follow the rules for addition.
Examples

- Signed-magnitude arithmetic
- Signed 2’s complement arithmetic
- Signed 1’s complement arithmetic

2’s Complement Adder/Subtractor

- Subtraction can be accomplished by addition of the Two’s Complement.
  1. Complement each bit (One’s Comp.)
  2. Add one to the result.
- The following circuit computes A - B:
  - When the Carry-In is 1, the 2’s comp of B is formed using XORs to form the 1’s comp and adding the 1 on C(0).

Overflow Detection and comments

- Overflow detection example
- Comments on hardware for
  - Sign-Magnitude
  - Complement-Based
  - Overflow detection hardware

Binary Multiplication

- The binary digit multiplication table is trivial:
  \[
  \begin{array}{c|c|c|c}
    a & b & a \times b \\
    \hline
    0 & 0 & 0 \\
    0 & 1 & 0 \\
    1 & 0 & 0 \\
    1 & 1 & 1 \\
  \end{array}
  \]
- This is simply the Boolean AND function.
- Form larger products the same way we form larger products in base 10.

Example: (237 x 149) Base 10

- Partial products are: 237 x 9, 237 x 4, and 237 x 1
- Note that the partial product summation for n digit, base 10 numbers requires adding up to n digits (with carries).
- Note also n x m digit multiply generates up to an m+n digit result.

Review of Decimal Multiplication

- Perform base 10 multiplication by:
  - Computing partial products, and
  - Justifying and summing the partial products.
- To compute partial products:
  - Multiply the row of multiplicand digits by each multiplier digit, one at a time.
- Partial product formation here require carries to be added – more complex than binary

Example: (237 x 149) Base 10

- Partial products are: 237 x 9, 237 x 4, and 237 x 1
- Note that the partial product summation for n digit, base 10 numbers requires adding up to n digits (with carries).
- Note also n x m digit multiply generates up to an m+n digit result.
### Binary Multiplication Algorithm

- We compute base 2 multiplication by:
  - Computing partial products, and
  - Justifying and summing the partial products. (same as decimal!)
- To compute partial products:
  - Multiply the row of multiplicand digits by each multiplier digit, one at a time.
  - With binary numbers, partial products are very simple! They are either:
    - all zero (if the multiplier digit is zero), or
    - the same as the multiplicand (if the multiplier digit is one).
- Note: No carries are added in partial product formation!

### Example: (101 x 011) Base 2

Partial products are:

- $101 \times 0 = 000$
- $101 \times 1 = 101$
- $101 \times 1 = 101$

Note that the partial product summation for $n$ digit, base 2 numbers requires adding up to $n$ digits (with carries) in a column.

Note also $n \times m$ digit multiply generates up to an $m + n$ digit result (same as decimal).

### Multiplier Boolean Equations

- We can also make an $n \times m$ "block" multiplier and use that to form partial products.
- Example: $2 \times 2$ - The logic equations for each partial-product binary digit are shown below:
- We need to "add" the columns $a_1 \cdot b_1$, $(a_1 \cdot b_0) + (a_0 \cdot b_1)$, and $(a_0 \cdot b_0)$ to get the product bits $P_0$, $P_1$, $P_2$, and $P_3$.
- Note that some columns may generate carries.

### Multiplier Arrays Using Adders

- An implementation of the $2 \times 2$ multiplier array is shown (Figure 3-33):

### Multiplier Using Wide Adders

- A more "structured" way to develop an $n \times m$ multiplier is to sum partial products using adder trees.
- The partial products are formed using an $n \times m$ array of AND gates.
- Partial products are summed using $m-1$ adders of width $n$ bits.
- Example: 4-bit by 3-bit adder
- Text Figure 3-34 shows a $4 \times 3 = 12$ element array of AND gates and 2 4-bit adders.

### Cellular Multiplier Arrays

- Another way to implement multipliers is to use an $n \times m$ cellular array structure of uniform elements as shown:
- Each element computes a single bit product equal to $a_i \cdot b_j$, and implements a single bit full adder.
BCD Addition

- A four-bit binary adder, with a correction circuit, can be used for BCD digit addition.
- For the correction circuit:
  - The BCD code is correct if the sum is 9 or less
  - The BCD code is incorrect:
    - if the adder output is 10, 11, 12, 13, 14, 15, or
    - if the 4-bit adder produces a carry out
  - We correct the sum by subtracting 10 (usually by the equivalent addition of 6 modulo 16)
- The Boolean equation with value 1 for doing a correction:
  - If \( C = 1 \), 0110 is added; else 0000 is added. This can be done by adding \( 0 \ C C 0 \). Also \( C \) is the Output carry.

BCD Addition (Continued)

- The resulting circuit (Figure 3.35):

Overview of Verilog – Part 1

- Objectives
  - To become familiar with the hardware description language (HDL) approach to specifying designs
  - Be able to read a simple Verilog HDL description
  - Be able to write a simple Verilog HDL description using a limited set of syntax and semantics
  - Understanding the need for a “hardware view” when reading and writing an HDL

Verilog Notation - 1

- Verilog is:
  - Case sensitive
  - Based on the programming language C
- Comments
  - Single Line
    - \( // \) [end of line]
  - Multiple Line
    - \( /* */ \)
- List element separator: ,
- Statement terminator: ;
Verilog Keywords & Constructs - 1

- Keywords are lower case
- module – fundamental building block for Verilog designs
  - Used to construct design hierarchy
  - Cannot be nested
- endmodule – ends a module – not a statement
  => no ";
- Module Declaration
  - module module_name (module_port, module_port, …);
  - Example: module full_adder (A, B, c_in, c_out, S);

Verilog Keywords & Constructs - 2

- Input Declaration
  - Scalar
    - input list of input identifiers;
    - Example: input A, B, c_in;
  - Vector
    - input[range] list of input identifiers;
    - Example: input[15:0] A, B, data;
- Output Declaration
  - Scalar Example: output c_out, OV, MINUS;
  - Vector Example: output[7:0] ACC, REG_IN, data_out;

Verilog Keywords & Constructs - 3

- Primitive Gates
  - buf, not, and, or, nand, nor, xor, xnor
  - Syntax: gate_operator instance_identifier (output, input_1, input_2, …)
  - Examples:
    and A1 (F, A, B); //F = A B
    or O1 (w, a, b, c)
    O2 (x, h, c, d, e); //w=a+b+c=x+h+c+d+e

Verilog Operators - 1

- Bitwise Operators
  - ~ NOT
  - & AND
  - | OR
  - ^ XOR
  - ^~ or ~^ XNOR
  - Example: input[3:0] A, B;
  - output[3:0] Z ;
  - assign Z = A | -B;

Verilog Operators - 2

- Arithmetic Operators
  +, -, (plus others)
- Logical & Relational Operators
  !, &&, ||, =, !=, <=, < (plus others)
- Concatenation & Replication Operators
  [identifier_1, identifier_2, …]
  (n[identifier])
  * Examples: [REG_IN[6:0],Serial_in], [S[1'b0]]

Structural Verilog

- Circuits can be described by a netlist as a text alternative to a diagram - Example (See Figure 3-59 in text):
  module fig359s (A0, B0, C0, C1, S0);
  input A0, B0, C0;
  output C1, S0;
  //Seven internal wires needed
  wire[7:0] N;
  //Ports on primitive gates listed output port first
  not G1 (N[3],C0), G2 (N[5],N[2]), G3 (N[6],N[3]);
  nand G4 (N[1],A0,B0), G5 (N[4],N[1],B0), G6 (N[6],N[2],N[4]);
  nor G7 (N[4],N[1],N[3]);
  xor G8 (N[6],N[7],N[5]);
  endmodule
Dataflow Verilog - 1

- Circuit function can be described by **assign** statements using Boolean equations (See Figure 3-59 in text):

```verilog
module fig359d (A0, B0, C0, C1, S0);
input A0, B0, C0;
output C1, S0;
wire[1:2] N;
assign N[1] = ~(A0 & B0); /* Note: Cannot write ~& for NAND */
assign N[2] = ~(A0 | B0);
assign C1 = ~((N[1] & ~C0) | N[2]);
assign S0 = (~N[2] & N[1])^(~(~C0));
endmodule
```

Dataflow Verilog - 2

- Circuit function can be described by **assign** statements using the conditional operator with binary combinations as in a truth table (See Figure 3-14 in text):

```verilog
module fig314dm (A, E_n, D_n);
input[1:0] A;
input E_n;
output[3:0] D_n;
// Conditional: (X) ? Y: Z - if X is true, then Y, else Z
assign D_n = {4{E_n}} & (A == 2'b00) ? 4'b1110:
            (A == 2'b01) ? 4'b1101:
            (A == 2'b10) ? 4'b1011:
            (A == 2'b11) ? 4'b0111:
            4'bxxxx;
endmodule
```

Behavioral & Hierarchical Verilog

- Circuit function can be described by **assign** statements at higher than the logic level (See Figure 3-31 in text):

```verilog
module addsub (A, B, R, sub) ;
input [3:0] A, B ;
output [3:0] R ;// See Fig. 3-51 for carry out
input sub ;
wire [3:0] data_out;
add A1 (A, data_out, sub, R);
M1comp C1 (B, data_out, sub);
endmodule
```

Behavioral & Hierarchical Verilog

```verilog
module add (X, Y, C_in, S);
input [3:0] X, Y;
input C_in;
output [3:0] S;
assign S = X + Y + {3'b0, C_in};
endmodule
module M1comp (data_in, data_out, comp);
input[3:0] data_in;
input comp;
output [3:0] data_out;
assign data_out = {4{comp}} ^ data_in;
endmodule
```

Summary

- Analysis
  - Forward - backward trace through the circuit to obtain output equations or truth table
  - Vice versa will also find the equations and truth table
- Know the functions performed by the following functional blocks:
  - Decoders, Demultiplexers, Encoders, Multiplexers
- Know how to implement Boolean functions using:
  - Multiplexers
  - Decoders