Goals for Lecture

- XOR/XNOR
- Combinational Logic Blocks

Exclusive OR/ Exclusive NOR

- The eXclusive OR (XOR) function is an important Boolean function used extensively in logic circuits. Uses for the XOR gate include:
  - Adders/subtractors
  - Parity generators/checkers
  - Signature analyzers
  - Pseudo-random sequence generators

- Definitions
  - The XOR function is: \( X \oplus Y = \overline{XY} + \overline{X}\overline{Y} \)
  - The eXclusive NOR (XNOR) function, otherwise known as Equivalence is: \( \overline{X \oplus Y} = XY + \overline{X}\overline{Y} \)

Tables for XOR/XNOR

- Operator Rules: XOR XNOR

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X ( \oplus ) Y</th>
<th>X ( \equiv ) Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- The XOR function means: X OR Y, but NOT BOTH
- The XNOR function, denoted by the operator \( \equiv \), is also known as the Equivalence function.

XOR/XNOR Extension

- The XOR function can be extended to 3 or more variables. For more than 2 variables, it is called a modulo 2 sum (Mod 2 sum), instead of XOR:
  \[
  X \oplus Y \oplus Z = \overline{XYZ} + \overline{XY}Z + XYZ + \overline{X}YZ
  \]

- The XOR definition, Boolean identities and Boolean theorems give:
  \[
  X \oplus 0 = X \quad X \oplus 1 = \overline{X}
  X \oplus X = 0 \quad X \oplus \overline{X} = 1
  X \oplus Y = \overline{X} \oplus Y \quad X \oplus Y = X \oplus Y
  X \oplus Y = Y \oplus X
  (X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z
  \]

XOR Implementations

- The simple SOP implementation uses the following structure:

- A NAND only implementation is:
XOR Implementations (Cont.)
- The AND-OR implementation is the SOP form for the XOR function:
  \[ X \oplus Y = \overline{X}Y + XY \]
- The multiple-level NAND implementation can be derived by combining inversions as follows:
  \[ X \oplus Y = \overline{X}Y + XY \]
  \[ = \overline{X}Y + XY = \overline{X}Y + XY \]
  \[ = \overline{X}Y \cdot (X + Y) \]
  \[ = \overline{X}Y \cdot X + \overline{X}Y \cdot Y \]

Odd Function
- The modulo 2 sum function for n variables
  - Contains an odd number of 1's, and
  - Is therefore called the odd function.
- The inverse of the modulo 2 sum function for n variables
  - Contains an even number of 1's, and
  - Is therefore called the even function.
- Implementation of even and odd functions for greater than 4 variables as a single gate is difficult, so “trees” of 2 to 4 input XOR or XNORs are used.

Example: Odd Function Implementation
- Three-Input Odd Function:
- Four Input Odd Function:

Product terms of Odd and Even
- For an n-bit even or odd function, there will be \((2^n)/2\) or \(2^n - 1\) product terms of n variables (minterms).

Parity Generators/Checkers
- A parity tree for n data bits generates a parity bit that is appended to the data bits to form an n + 1-bit codeword
- Example: 3-bit even parity generator
- A parity tree for n + 1 bits checks the codeword for correct parity:
  - C=0 if the codeword parity is correct
  - C=1 if the codeword parity is incorrect
- Example: 4-bit even parity checker

Positive and Negative Logic
- The same physical gate has different logical meanings depending on interpretation of the signal levels.
- Positive Logic
  - Logic 1 is set to HIGH (more positive) signal levels
  - Logic 0 is set to LOW (less positive) signal levels
- Negative Logic
  - Logic 1 is set to LOW (more negative) signal levels
  - Logic 0 is set to HIGH (less negative) signal levels
- A gate which implements a Positive Logic AND function will implement a Negative Logic OR function.
Design Hierarchy (MK 3.1)

- Combinational Circuits
- A combinational logic circuit has:
  - A set of \( m \) Boolean inputs,
  - A set of \( n \) Boolean outputs, and
  - \( n \) switching functions mapping the \( 2^m \) input combinations to a output such that the current output depends only on the current inputs.
- A block diagram:

Hierarchical Design

- The function mapping inputs to outputs may be very complex
  - To control complexity, we decompose the function into smaller pieces called blocks
  - The blocks are subdivided into finer blocks
  - The "leaves" in the hierarchy are called primitive blocks
- Example: 16 input parity tree
  - Top Level: 16 inputs, one output
  - 2nd Level: Five 4-bit parity trees in two levels
  - 3rd Level: Three 2-bit exclusive-OR functions
  - Primitive level: Four 2-input NANDs
  - The design requires \( 5 \times 3 \times 4 = 60 \) two-input NAND gates

Reusable Functions and Design

- Whenever possible, we try to decompose a complex design into common, reusable function blocks
- These blocks are tested and well documented
- Computer-aided design (CAD) tools might include them in libraries
- Computer-aided manufacturing (CAM) tools might know how to manufacture and test them
- Other tools:
  - Schematic Capture
  - Logic Simulators
  - Timing Verifiers
  - Hardware Description Languages (HDL)

What is "Design (Technology)"?

- Design = specification, synthesis, analysis
  - What do I want?
  - Create
  - Check
- Design Technology = Technology that enables design to occur in a timely, cost-effective way
  - (What happens if we can't design new products quickly?)
- Libraries
- Tools
- Methods

Top-Down versus Bottom-Up

- A Top-Down design proceeds from an abstract, high level specification to a more and more detailed design by decomposition and successive refinement
- A Bottom-Up design starts with detailed primitive elements and combines them into larger and larger and more complex functions
- Designs usually proceed from both directions simultaneously
  - Top-Down design answers: What are we building?
  - Bottom-Up design answers: How do we build it?
- Top-Down controls complexity while Bottom-Up controls the details

Analysis Procedure

- Switching Functions from Logic Diagrams
  - Given a logic diagram, the analysis process provides a set of Boolean equations, a truth table, or a verbal explanation of circuit behavior.
  - Procedure:
    1. Determine that the circuit is combinational (no feedback loops), then:
    2. Identify and label all gate outputs that are a function of the input variables. Obtain the Boolean functions for these labeled gate outputs.
    3. Identify and label all gate outputs that are a function of inputs or previously labeled gates. Obtain Boolean functions for them.
    4. Repeat Step 2 until all outputs are completed.
    5. Back substitute until all functions are specified in terms of inputs only.
Analysis Example

- Step 2: Label all outputs of gates near inputs.

- Write Boolean equations for them:
  \[ T_1 = \overline{B} + C \]
  \[ T_2 = B \cdot \overline{E} \]

Analysis (Continued)

- Step 3: Identify and label all gate outputs that are a function of inputs or previously labeled gates. Obtain Boolean functions for them.
  \[ T_3 = \overline{D} + T_2 \]

- Step 4: Repeat Step 3 until all done
  \[ T_4 = T_1 T_3 \]
  \[ F = A + T_4 \]

Analysis (Continued)

- Step 4: Back substitute until all functions are specified in terms of inputs only
  \[ F = A + T_4 \]
  \[ T_4 = T_1 T_3 \]
  \[ T_3 = \overline{D} + T_2 \]
  \[ T_2 = B \cdot \overline{E} \]
  \[ T_1 = \overline{B} + C \]
  - Substituting:
    \[ T_3 = \overline{D} + (B \cdot E) \]
    \[ T_4 = (B + C) \cdot (D + (B \cdot E)) \]
    \[ F = A + (B + C) \cdot (D + (B \cdot E)) \]

Analysis Example: Code Converter

Truth Tables from Logic Diagrams

1. Determine the number of input variables, \( n \). There will be \( 2^n \) input vectors from zero to \( 2^n - 1 \). Enter them in the table.
2. Label the outputs of selected gates with symbols and enter a column for each one in the table.
3. Obtain the truth table for the outputs of those gates that are a function of only input variables.
4. Proceed to fill in the outputs of all gates that are derived from inputs and previously calculated terms.

Truth Tables from Logic Diagrams

- Procedure:
  - Determine the number of input variables, \( n \). There will be \( 2^n \) input vectors from zero to \( 2^n - 1 \).
    Enter them in the table.
  - Label the outputs of selected gates with symbols and enter a column for each one in the table.
  - Obtain the truth table for the outputs of those gates that are a function of only input variables.
  - Proceed to fill in the outputs of all gates that are derived from inputs and previously calculated terms.
- Example: Find the function table for the code converter.
Code Converter Truth Table

- Four inputs give 16 input vectors.
- Start with F0, F1 and z.

<table>
<thead>
<tr>
<th>ABCD</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Truth Table Fill-in

- Now we can calculate x, y, and F2.

<table>
<thead>
<tr>
<th>ABCD</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Complete Entries

- Finally we can fill in w to complete the table:

<table>
<thead>
<tr>
<th>ABCD</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What Does the Circuit Do?

- By inspection, the output variable vector (w,x,y,z) is just the input variable vector (A,B,C,D) plus three.
- The function(s) F(A,B,C,D) = (w,x,y,z) are:
  - “ADD THREE TO THE INPUT VECTOR”
  - Function F1 has the meaning:
    - “ADD ONE TO THE UPPER TWO BITS”
  - Similarly, function F2 has the meaning:
    - “ADD ONE TO THE UPPER BIT”
- Generally, it is not this obvious to figure out what the functions mean.

Final Note (and warning)

- The use of “Don’t Cares” in the original specification can cloud the analysis.
  - Note that the functions for the “w” bit differ from the implementation in Ex. 3-2 of the book.
  - The book used “Don’t Cares” to simplify the logic. The example here did not.
  - This can be seen by inspecting the two K-maps for the function w:

Logic Design: Functional Blocks

- Analysis: From a design to a specification of the behavior
  - Logic diagram to equations
  - Logic diagram to function table
  - “Word description” of circuit operation
- Synthesis: From a specification to design implementation
  - Define the problem
  - Generate function table or equations
  - Minimize the Boolean function
  - Implement the circuit
Combinational Logic Implementation

- A combinational logic circuit has:
  - A set of \( m \) Boolean inputs,
  - A set of \( n \) Boolean outputs, and
  - A function mapping inputs to outputs.
- We think of the function as \( n \) separate Boolean functions of \( m \) inputs
- Procedure:
  - Treat each output as a separate function
  - Minimize the equations for each function
  - Implement each function independently
  - Sometimes an implementation can share product or sum logic terms to arrive at a lower literal cost solution.

Design Procedure (MK 3.4)

- First, start with the specification of the circuit to be designed.
  - Note: this can sometimes require a lot of work to complete the specification process, especially if it is poorly specified initially
- Second, follow these steps: We will study the design of a code converter to see these steps.
  - Identify the inputs and outputs
  - Derive truth table
  - Obtain simplified Boolean equations
  - Draw the logic diagram
  - Check your work to verify correctness

Code Converter Design Example

- A code converter transforms one internal representation of data to another
- We will start with a table of the desired conversion and minimize the resulting multiple output Boolean function
- Sometimes terms can be shared to minimize the implementation cost
- The Problem:
  - Design a BCD to Excess-3 code converter
  - Specification:
    - BCD code -- 4-bit patterns "0000" to "1001" for digits 0 to 9 base 10
    - Excess-3 -- BCD code plus binary "0011" for digits 0 to 9 base 10

Example: BCD to Excess 3

Function table:

<table>
<thead>
<tr>
<th>Input BCD</th>
<th>Output Excess-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 1 0 1</td>
</tr>
</tbody>
</table>

Note:
All BCD codes greater than "9" can be assigned "Don't Cares" in the K-Map. Such BCD codes are never possible.

Example (Cont.): BCD to Excess 3

- Map functions and find minimum cost SOP equations for each

Example (Cont.): BCD to Excess 3

- Next, we will manipulate the equations to expose some shared terms:
  - The term (C + D) can be used more than once to simplify the implementation
  - See Fig. 3-10 in Mano and Kime for the implementation
Functional Block: Decoders (MK 3.5)

- A Decoder converts n binary bits to a maximum of $2^n$ unique output lines.
- An m-to-n line decoder, where $m < 2n$, can be used to:
  - Generate $2n$ (or fewer) minterms
  - Select one of $2n$ items
- Decoders are sometimes known as demultiplexers when enabled with a separate data-in line.

2-to-4 Line Decoder

This device takes:
- n=2 input lines
- m=2$^2$=4 output lines

Implementing Logic with Decoders

- Decoders provide minterms directly. Simply "OR" the appropriate minterm outputs to make any logic function desired.
- Active low decoders behave as the first NAND gate in a NAND-NAND, Sum of Products implementation.
- Active high decoders behave as first stage AND gates in a AND-OR Sum of Products implementation.
- Two or more active high decoders driven from different bits of a binary code can be used to form minterms by "ANDING" their outputs. Similarly, active low decoders can be used to form minterms by "ORING" their outputs.

Example: $F(A,B) = \Sigma m(0,3)$

For this we use a 2-to-4 line decoder and sum minterms 0 and 3 with an OR gate:

Example: $F(X,Y,Z) = \Sigma m(0,3,5,6)$

For this we use a 2-to-4 line decoder and sum minterms 0, 3, 5, and 6 with an OR gate:
Implementing Larger Minterms

- Minterm m15 is formed by "ANDING" the D3 outputs of each decoder.
- Similarly m0 is formed by "ANDING" the D0 outputs of each decoder.
- What minterm is formed by "ANDING" D1 (upper) and D2 (lower) outputs?

This works best with widely scattered, sparse minterms.

Functional Block: Encoders (MK 3.6)

- Encoders perform the "inverse" operation of decoders, taking a code in one format and encoding it into another format.
- Many encoders consist of just OR gates. For example an 8-to-3 binary encoder consists of three 4-input OR gates, OR2, OR1 and OR0. Input i, i = 0,...,7 is connected to an input on OR j if the binary representation of i has a 1 in position j.
- A priority encoder is used to generate a code for the "most significant" bit set in a string of bits. This can be used to find the first one in a word, or to select external events in priority order. An example of a MSI priority encoder is the 74F148, 8 line to 3 line priority encoder. It can be cascaded to encode higher numbers of bits.

Encoder Example

- Encode 4 lines 0, 1, 2, 3 into the corresponding binary codes.

Review: Decoders and Encoders

- A Decoder converts n binary bits to a maximum of $2^n$ unique output lines.
- Decoders are sometimes know as demultiplexers when enabled with a separate data-in line.
- Decoders implement minterms directly.
- Use a decoder and an OR gate to form Sum-of-Minterms directly.
- Encoders perform the "inverse" operation of decoders, taking a code in one format and encoding it into another format.

Multiplexers (MK 3.7)

- A Multiplexer (MUX) is another common functional block.
- A Multiplexer uses n binary select bits to choose from a maximum of $2^n$ unique input lines.
- Like a decoder, it decodes minterms internally.
- Unlike a decoder, it has only one output line.
- The decoded minterms are used to select data from one of up to $2^n$ unique data input lines.
- The output of the multiplexer is the data input whose index is specified by the n bit code.

Example: A 4-to-1 multiplexer

- The 4-to-1 line multiplexer uses the same minterm decoder core.
- It is like a demultiplexer with individual data input lines (instead of just one) and an output OR gate.
### Functions with Multiplexers

- It is possible to implement any Boolean function of $n$ variables with a $2^n$ input multiplexer.
- Simply tie each input to the "1" or "0" line as desired.
- It is also possible to implement any $n+1$ variable function with a $2^n$ multiplexer. Simply use the $(n+1)^{st}$ variable in true or complement form depending upon what the truth table requires.
- A Boolean function of more than $n$ variables can be partitioned into several easily implemented sub-functions defined on a subset of the variables.
- The multiplexer will then select among these sub-functions.

### Example: Gray to Binary Code

- The Gray code has adjacent elements separated by only one bit change.
- We wish to convert a 3-bit Gray code to a binary code.
- The function table on the right documents the required conversion.

<table>
<thead>
<tr>
<th>Gray</th>
<th>Binary</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>x y z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Gray to Binary Code Converter requires us to implement three separate, three-input Boolean functions.

### Gray to Binary (Continued)

The K-Maps

- $x(A,B,C) = C$, is an easy function to implement. (No logic gates needed!)
- Function $y(A,B,C) = B'C + B'C$ is a bit harder to implement.
- Function $z(A,B,C)$ looks familiar.

In this case, the MUX elements are acting like a "Read Only Memory" (ROM).
Other MUX Implementations

We can also use two 4-to-1 MUX blocks and implement y and z.
Suppose we factor out A and use B and C as the select inputs to the multiplexers.

\[ \begin{array}{c|c|c|c|c|c|c|c|c} 
Y & 00 & 01 & 11 & B & 10 \\
\hline
A & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\hline
B & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
\hline
C & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array} \]

MUX: (Cont.) Factoring Out C

We could have factored out other variables. As in the book, we will factor out C and apply AB to the select inputs:

<table>
<thead>
<tr>
<th>Gray</th>
<th>Binary</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0 0 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This is slightly larger than selecting A to factor out.

MUX: (Cont.) Factoring out B

<table>
<thead>
<tr>
<th>Gray</th>
<th>Binary</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 1 0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0 0 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: We re-arranged the table (fixing A and C and varying B from 0 to 1 in each cell) to simplify this procedure. It still looks like factoring A was better.

MUX: (Cont.) Factoring out A

<table>
<thead>
<tr>
<th>Gray</th>
<th>Binary</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 0 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 1 0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1 0 0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0 1 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>0 1 0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 1 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: We re-arranged the table (fixing B and C and varying A from 0 to 1 in each cell) to simplify this procedure. Factoring A is best! Note also that x = C holds.

Summary

- Know the functions performed by the following functional blocks:
  - Decoders
  - Demultiplexers
  - Encoders
  - Multiplexers
- Know how to implement Boolean functions using:
  - Multiplexers
  - Decoders
Functional Blocks: Addition

- Binary addition occurs frequently in digital and computer systems.
- In this section, we:
  - Develop a Half-Adder (HA), a 2-input bitwise addition functional block,
  - Develop a Full-Adder (FA), a 3-input bitwise addition functional block,
  - Iterate full-adders using a ripple-carry to perform parallel binary addition, and
  - Develop a Carry-Look-Ahead Adder (CLA) to improve performance.

Functional Block: Half-Adder

- A 2-input, 1-bit width binary adder that produces the following values:
  \[
  \begin{array}{c|c|c|c|c}
  X & 0 & 0 & 1 & 1 \\
  + Y & 0 & 0 & 1 & 1 \\
  C & 0 & 1 & 0 & 1 \\
  S & 0 & 1 & 1 & 0 \\
  \end{array}
  \]

Logic Simplification: Half-Adder

- The K-Map for S, C is:
- This is a pretty trivial map! By inspection:
  \[
  S = X \cdot Y + X \odot Y \\
  C = X \cdot Y \\
  \]
- These equations lead to several implementations.

Five Implementations: Half-Adder

- We can derive following sets of equations for a half-adder:
  (a) \( S = X \cdot Y + X \odot Y \) 
  (b) \( S = (X \cdot Y) \odot (X + Y) \) 
  (c) \( S = (C \cdot X \cdot Y) \) 
  (d) \( S = (X + Y) \odot C \) 
  (e) \( S = X \odot Y \) 

- 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.
- (a), (b), and (e) are SOP, POS, and XOR implementations for S.

Implementations: Half-Adder

- The most common half adder implementation (e) is:
  \( S = X \odot Y \) 
  \( C = X \cdot Y \)
- A NAND only implementation (equivalent to equation d) is:
  \( S = (X + Y) \cdot C \) 
  \( C = (X \odot Y) \)
Design: Full-Adder

- Full-Adder Function Table:

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

- Full-Adder K-Map:

\[ S = X \oplus Y \oplus Z \]
\[ C = X \cdot Y + Z \]

From the K-Map, we get:

- The S function is the three-bit XOR function (Odd Function):
  \[ S = X \oplus Y \oplus Z \]
- The Carry bit C is 1 if both X and Y are 1 (the sum is 2), or if the sum is 1 and a carry-in (Z) occurs. Thus C can be re-written as:
  \[ C = X \cdot Y + (X \oplus Y) \cdot Z \]
- The term X\(Y\) is carry generate.
- The term X\(Y\) is carry propagate.

Implementation: Full Adder

- Full Adder Schematic

Here X, Y, and Z, and C (from the previous pages) are A, B, C, and Ci respectively. Also,

- G = Generate and
- P = Propagate.
- Note: This really is a combination of a 3-bit odd function (for S = sum) and Carry logic:
  \[ G = \text{Generate} \text{ OR} (P = \text{Propagate} \text{ AND} C_i = \text{Carry In}) \]

\[ C_o = G + P \cdot C_i \]

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.

  \[ \text{Example: 4-bit binary adder:} \]

  Adds an input vector "A(3..0)" to "B(3..0)" to get a sum S(3..0) thus:

  \[ \begin{align*}
  \text{Input Carry} & \quad 0110 \\
  \text{Augend} & \quad 1011 \\
  \text{Addend} & \quad 0111 \\
  \text{Sum} & \quad 1111 \\
  \text{Output} & \quad 0011 \\
  \text{Carry} & \quad C_{i+1}
  \end{align*} \]

  \[ \text{Note: the carry out of Stage} \]

  \[ \text{becomes the carry in of Stage} \]

  \[ \text{Stage} i+1. \]

4-bit Ripple-Carry Binary Adder

- A four-bit Full Adder made from four 1-bit Full Adders:

\[ A(3) \quad B(3) \quad A(2) \quad B(2) \quad A(1) \quad B(1) \quad A(0) \quad B(0) \]

\[ S(3) \quad S(2) \quad S(1) \quad S(0) \]

- 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:

\[ \text{Note: The "long path" is from} \]

\[ \text{A(0)} \text{or B(0) through the network to either} \]

\[ \text{C(4)} \text{or S(3).} \]
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 “Half-Sum” 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: $\Rightarrow$

$$
\begin{array}{c|c|c|c}
&A_i & B_i & C_i+1 \\
\hline
& Pi & Si & C_i \\
\end{array}
$$

Carry Look-Ahead (Continued)

- Look at the following addition examples, all of which generate a carry of 1 out of the third stage:

<table>
<thead>
<tr>
<th>Example</th>
<th>Carry Propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1111 + 0000$</td>
<td>$10000$</td>
</tr>
<tr>
<td>$1110 + 0010$</td>
<td>$11010$</td>
</tr>
<tr>
<td>$1111 + 1011$</td>
<td>$11000$</td>
</tr>
<tr>
<td>$1111 + 1000$</td>
<td>$11000$</td>
</tr>
</tbody>
</table>

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:

$$
C_4 = G_0 + P_0 C_0
$$

- 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 $r = 10$, $N = 123410$, $n = 4$ (4 digits), we have: $(10^n - 1) = 10,000 - 1 = 999910$
  - The 9's complement of 123410 is then:
    - 999910 - 123410 = 876510
Binary 1's Complement

- For \( r = 2 \), \( N = 01110011_2 \), \( n = 8 \) (8 digits):
  - \( r^n = 256 \cdot 1 = 255_{10} \) or \( 11111111_2 \)
  - The 1's complement of \( 01110011_2 \) is then:
    \[
    \begin{array}{c}
    \text{11111111} \\
    \text{− 01110011} \\
    \text{10001100}
    \end{array}
    \]
- 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 \( N \neq 0 \) and
  - 0 for \( N = 0 \)
- The radix complement is obtained by adding 1 to the diminished radix complement
- 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
      \[
      10,000 - 1234 = 8766_{10} \text{ or } 8765 + 1 \quad \text{(9's complement plus 1)}
      \]

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 \( 01110011_2 \) is then:
    \[
    \begin{array}{c}
    \text{100000000} \\
    \text{− 01110011} \\
    \text{10001101}
    \end{array}
    \]
- 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:
  - \( 10010100 \)
  - Copy underlined bits:
    \[
    \text{01101}
    \]
  - and complement bits to the left:
    \[
    \text{01101100}
    \]

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 \( M \):
    - \( M + (r^n − N) = M − N + r^n \)
  - 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 \( 543_{10} − 123_{10} \)
  \[
  \begin{array}{cccccc}
  543 & & & & & 543 \\
  \text{− 123} & \text{10's comp} & & \text{+ 877} & & \text{420}
  \end{array}
  \]
- The carry of 1 indicates that no correction of the result is required.
### Unsigned 10's Complement Subtraction

**Example 2**

Find \(123_{10} - 543_{10}\)

<table>
<thead>
<tr>
<th>123</th>
<th>0 123</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-543)</td>
<td>10's comp</td>
</tr>
</tbody>
</table>

- The carry of 0 indicates that a correction of the result is required.
- Result = \(-520\)

### Unsigned 2's Complement Subtraction

**Example 1**

Find \(01010100_2 - 01000011_2\)

<table>
<thead>
<tr>
<th>01010100</th>
<th>01010100</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-01000011)</td>
<td>2's comp</td>
</tr>
</tbody>
</table>

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

**Example 2**

Find \(01000011_2 - 01010100_2\)

<table>
<thead>
<tr>
<th>01000011</th>
<th>01000011</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-01010100)</td>
<td>2's comp</td>
</tr>
</tbody>
</table>

- 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 \(r\):

- Add the \((r-1)\)'s complement of the subtrahend \(N\) to the minuend \(M\):
  \[ M + (r^n - 1 - N) = M - N + r^n - 1 \]
- 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.
- If \(M < N\), the sum does not produce an end carry and, from above, is equal to \(r^n - 1 - (N - M)\), the \(r-1\)'s complement of \((N - M)\).
- To obtain the result \((N - M)\), take the \(r-1\)'s complement of the sum and place an \(-1\) in front.

### Unsigned 1's Complement Subtraction

**Example 1**

Find \(01010100_2 - 01000011_2\)

<table>
<thead>
<tr>
<th>01010100</th>
<th>01010100</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-01000011)</td>
<td>1's comp</td>
</tr>
</tbody>
</table>

- The end-around carry occurs.

**Example 2**

Find \(01000011_2 - 01010100_2\)

<table>
<thead>
<tr>
<th>01000011</th>
<th>01000011</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-01010100)</td>
<td>1's comp</td>
</tr>
</tbody>
</table>

- 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:

\[ s \cdot a_{n-1} \ldots a_2a_1a_0 \]

where:

- \( s = 0 \) for Positive numbers
- \( s = 1 \) for Negative numbers
- \( a_i = 0 \) 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>100</td>
<td>111</td>
<td>—</td>
</tr>
<tr>
<td>−1</td>
<td>011</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>−2</td>
<td>110</td>
<td>101</td>
<td>110</td>
</tr>
<tr>
<td>−3</td>
<td>111</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>−4</td>
<td>—</td>
<td>—</td>
<td>100</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.

Sign-Magnitude Examples

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.
Signed 2’s Complement Examples

Signed 1’s Complement Examples

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

Binary Multiplication

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

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

<table>
<thead>
<tr>
<th>$237 \times 9$</th>
<th>$237 \times 4$</th>
<th>$237 \times 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2133</td>
<td>948</td>
<td>237</td>
</tr>
</tbody>
</table>

$\text{Total} = 35313$

### Binary Multiplication Algorithm

- We compute base 2 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.
  - 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: $101 \times 0$, $101 \times 1$, and $101 \times 1$
- 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).

<table>
<thead>
<tr>
<th>$101 \times 0$</th>
<th>$101 \times 1$</th>
<th>$101 \times 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
</tbody>
</table>

$\text{Total} = 100111$