Logic and Computer Design Fundamentals

# Chapter 4 – Arithmetic Functions

**Charles Kime & Thomas Kaminski**



(Hyperlinks are active in View Show mode)

## Overview

- Iterative combinational circuits
- Binary adders
  - Half and full adders
  - Ripple carry and carry lookahead adders
- Binary subtraction
- Binary adder-subtractors
  - Signed binary numbers
  - Signed binary addition and subtraction
  - Overflow
- Binary multiplication
- Other arithmetic functions
  - Design by contraction

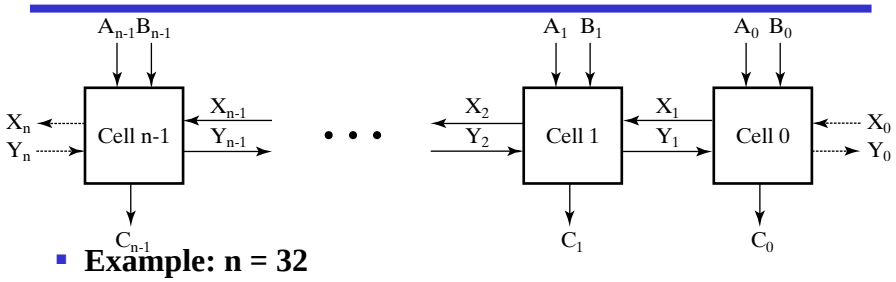
# Iterative Combinational Circuits

- **Arithmetic functions**
  - **Operate on binary vectors**
  - **Use the same subfunction in each bit position**
- **Can design functional block for subfunction and repeat to obtain functional block for overall function**
- ***Cell* - subfunction block**
- ***Iterative array* - a array of interconnected cells**
- **An iterative array can be in a <u>single</u> dimension (1D) or <u>multiple</u> dimensions**





# Block Diagram of a 1D Iterative Array




- **Example: n = 32**
  - **Number of inputs = ?**
  - **Truth table rows = ?**
  - **Equations with up to ? input variables**
  - **Equations with huge number of terms**
  - **Design impractical!**
- **Iterative array takes advantage of the regularity to make design feasible**

# Functional Blocks: Addition

- **Binary addition used frequently**
- **Addition Development:**
  - ***Half-Adder*** **(HA), a 2-input bit-wise addition functional block,**
  - ***Full-Adder*** **(FA), a 3-input bit-wise addition functional block,**
  - ***Ripple Carry Adder*****, an iterative array to perform binary addition, and**
  - ***Carry-Look-Ahead Adder*** **(CLA), a hierarchical structure to improve performance.**





# Functional Block: Half-Adder

- **A 2-input, 1-bit width binary adder that performs the following computations:**

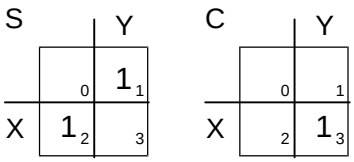
| | | | | |
|---|---|---|---|---|
| **X** | **0** | **0** | **1** | **1** |
| **+ Y** | **+ 0** | **+ 1** | **+ 0** | **+ 1** |
| **C S** | **0 0** | **0 1** | **0 1** | **1 0** |

- **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 truth table for S and C ⇒**

| X | Y | C | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

## Logic Simplification: Half-Adder

- **The K-Map for S, C is:**
- **This is a pretty trivial map! By inspection:**




$$S = X \cdot \overline{Y} + \overline{X} \cdot Y = X \oplus Y$$

$$S = (X + Y) \cdot \overline{(\overline{X} + \overline{Y})}$$

- **and**

$$C = X \cdot Y$$

$$C = \overline{(\overline{X \cdot Y})}$$

- **These equations lead to several implementations.**





## Five Implementations: Half-Adder

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

**(a)** $S = X \cdot \overline{Y} + \overline{X} \cdot Y$
$C = X \cdot Y$

**(b)** $S = (X + Y) \cdot (\overline{X} + \overline{Y})$
$C = \underline{X \cdot Y}$

**(c)** $S = \overline{(C + \overline{X} \cdot \overline{Y})}$
$C = X \cdot Y$

**(d)** $\underline{S} = (\underline{X} + \underline{Y}) \cdot \overline{C}$
$C = (X + Y)$

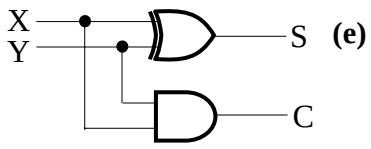
**(e)** $S = X \oplus Y$
$C = X \cdot Y$

- **(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 $\overline{C}$ function is used in a POS term for S.**

# Implementations: Half-Adder

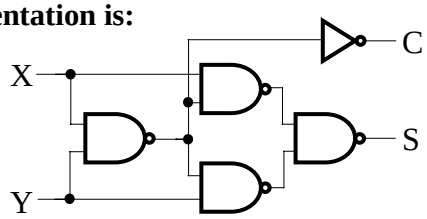
- **The most common half adder implementation is:**

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




- **A NAND only implementation is:**

$$S = (\overline{X + Y}) \cdot C$$
$$C = (\overline{(X \cdot Y)})$$






# 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:**
  - **For a carry- in (Z) of 1:**

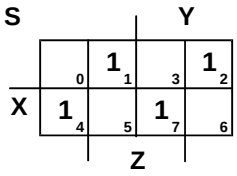
| | | | | |
|---|---|---|---|---|
| **Z** | **0** | **0** | **0** | **0** |
| **X** | **0** | **0** | **1** | **1** |
| **+ Y** | **+ 0** | **+ 1** | **+ 0** | **+ 1** |
| **C S** | **0 0** | **0 1** | **0 1** | **1 0** |
| **Z** | **1** | **1** | **1** | **1** |
| **X** | **0** | **0** | **1** | **1** |
| **+ Y** | **+ 0** | **+ 1** | **+ 0** | **+ 1** |
| **C S** | **0 1** | **1 0** | **1 0** | **1 1** |

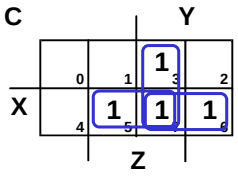
# Logic Optimization: Full-Adder

- **Full-Adder Truth Table:**

| 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 |

- **Full-Adder K-Map:**








# Equations: Full-Adder

- **From the K-Map, we get:**

$$S = X\,\overline{Y}\,\overline{Z} + \overline{X}\,Y\,\overline{Z} + \overline{X}\,\overline{Y}\,Z + X\,Y\,Z$$

$$C = X\,Y + X\,Z + Y\,Z$$

- **The S function is the three-bit XOR function (Odd Function):**

$$S = X \oplus Y \oplus Z$$

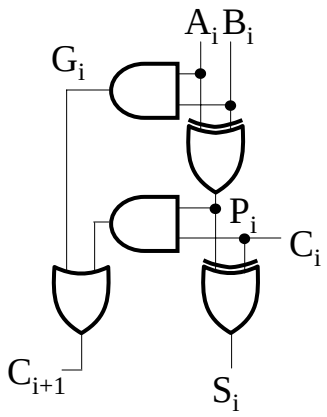
- **The Carry bit C is 1 if both X and Y are 1 (the sum is 2), or if the sum is 1 and a carry-in (Z) occurs. Thus C can be re-written as:**

$$C = X\,Y + (X \oplus Y)\,Z$$

- **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, $C_i$ and $C_o$, respectively. Also,**
  **G = generate and**
  **P = propagate.**
- **Note: This is really a combination of a 3-bit odd function (for S)) and Carry logic (for $C_o$):**




**(G = Generate) OR (P =Propagate AND $C_i$ = Carry In)**

**$C_o = G + P \cdot C_i$**





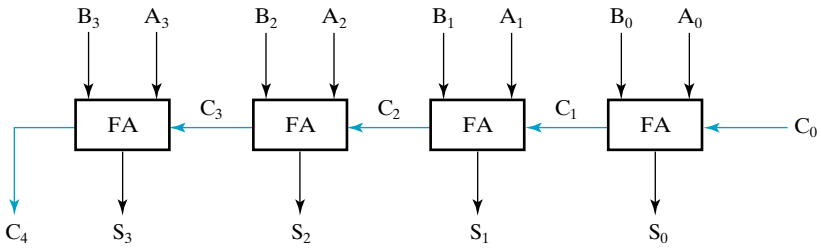
# Binary Adders

- **To add multiple operands, we "bundle" logical signals together into vectors and use functional blocks that operate on the vectors**
- **Example: 4-bit ripple carry adder: Adds input vectors A(3:0) and B(3:0) to get a sum vector S(3:0)**
- **Note: carry out of cell i becomes carry in of cell i + 1**

| Description | Subscript 3 2 1 0 | Name |
|---|---|---|
| Carry In | 0 1 1 0 | $C_i$ |
| Augend | 1 0 1 1 | $A_i$ |
| Addend | 0 0 1 1 | $B_i$ |
| Sum | 1 1 1 0 | $S_i$ |
| Carry out | 0 0 1 1 | $C_{i+1}$ |

# 4-bit Ripple-Carry Binary Adder

- **A four-bit Ripple Carry Adder made from four 1-bit Full Adders:**





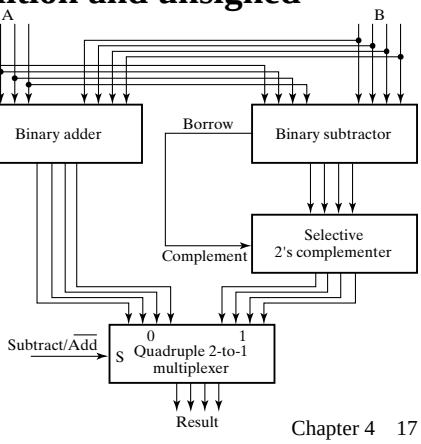

# Unsigned Subtraction

- **Algorithm:**
  - **Subtract the subtrahend N from the minuend M**
  - **If no end borrow occurs, then M ≥ N, and the result is a non-negative number and correct.**
  - **If an end borrow occurs, the N > M and the difference M – N + 2n is subtracted from 2n, and a minus sign is appended to the result.**
- **Examples:**

| | |
|---|---|
| **0** | **1** |
| **1001** | **0100** |
| **– 0111** | **– 0111** |
| **0010** | **1101** |
| | **10000** |
| | **– 1101** |
| | **(–) 0011** |

# Unsigned Subtraction (continued)

- The subtraction, $2^n - N$, is taking the 2's complement of N
- To do both unsigned addition and unsigned subtraction requires:
- Quite complex!
- Goal: Shared simpler logic for both addition and subtraction
- Introduce complements as an approach








# Complements

- Two complements:
  - Diminished Radix Complement of N
    - (r – 1)'s complement for radix r
    - 1's complement for radix 2
    - Defined as $(r^n - 1) - N$
  - Radix Complement
    - r's complement for radix r
    - 2's complement in binary
    - Defined as $r^n - N$
- Subtraction is done by adding the complement of the subtrahend
- If the result is negative, takes its 2's complement

## 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 $01110011_2$ is then:

$$\begin{array}{r} 11111111 \\ -\ \underline{01110011} \\ 10001100 \end{array}$$

- 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 <u>complementing each individual bit</u> (bitwise NOT).





## 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:

$$\begin{array}{r} 100000000 \\ -\ \underline{01110011} \\ 10001101 \end{array}$$

- Note the result is the 1's complement plus 1, a fact that can be used in designing hardware

## Alternate 2's Complement Method

- **Given: an *n*-bit binary number, beginning at the least significant bit and proceeding upward:**
  - **Copy all least significant 0's**
  - **Copy the first 1**
  - **Complement all bits thereafter.**
- **2's Complement Example:**

  **10010<u>100</u>**
  - **Copy underlined bits:**

    **<u>100</u>**
  - **and complement bits to the left:**

    **<u>01101</u>100**





## Subtraction with 2's Complement

- **For n-digit, <u>unsigned</u> numbers M and N, find M – N in base 2:**
  - **Add the 2's complement of the subtrahend N to the minuend M:**

    $$M + (2^n - N) = M - N + 2^n$$
  - **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 $2^n - (N - M)$, the 2's complement of ( N – M ).**
  - **To obtain the result – (N – M) , take the 2's complement of the sum and place a – to its left.**

## Unsigned 2's Complement Subtraction Example 1

- Find $01010100_2 - 01000011_2$

$$\begin{array}{rl} & 01010100 \\ - & \underline{01000011} \end{array} \xrightarrow{\text{2's comp}} \begin{array}{rl} & 1\,01010100 \\ + & \underline{10111101} \\ & 00010001 \end{array}$$

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





## Unsigned 2's Complement Subtraction Example 2

- Find $01000011_2 - 01010100_2$

$$\begin{array}{rl} & 01000011 \\ - & \underline{01010100} \end{array} \xrightarrow{\text{2's comp}} \begin{array}{rl} & 0\,01000011 \\ + & \underline{10101100} \\ & 11101111 \end{array} \xrightarrow{\text{2's comp}} 00010001$$

- 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, the most significant bit is interpreted as a sign bit:**

$$s\ a_{n-2} \ldots a_2 a_1 a_0$$

**where:**

**s = 0 for Positive numbers**

**s = 1 for Negative numbers**

**and $a_i$ = 0 or 1 represent the magnitude in some form.**





## 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 1's Complement***
    - **Uses 1's Complement Arithmetic**
  - ***Signed 2's Complement***
    - **Uses 2's Complement Arithmetic**

## Signed Integer Representation Example

- **r =2, n=3**

| Number | Sign -Mag. | 1's Comp. | 2's Comp. |
|---|---|---|---|
| +3 | 011 | 011 | 011 |
| +2 | 010 | 010 | 010 |
| +1 | 001 | 001 | 001 |
| +0 | 000 | 000 | 000 |
| – 0 | 100 | 111 | — |
| – 1 | 101 | 110 | 111 |
| – 2 | 110 | 101 | 110 |
| – 3 | 111 | 100 | 101 |
| – 4 | — | — | 100 |





## Signed-Magnitude Arithmetic

- **If the parity of the three signs is 0:**
  1. **Add the magnitudes.**
  2. **Check for overflow (a carry out of the MSB)**
  3. **The sign of the result is the same as the sign of the first operand.**
- **If the parity of the three signs is 1:**
  1. **Subtract the second magnitude from the first.**
  2. **If a borrow occurs:**
     - **take the two's complement of result**
     - **and make the result sign the complement of the sign of the first operand.**
  3. **Overflow will never occur.**

## Sign-Magnitude Arithmetic Examples

- **Example 1:** 0010 + 0101
- **Example 2:** 0010 + 1101
- **Example 3:** 1010 – 0101





## 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

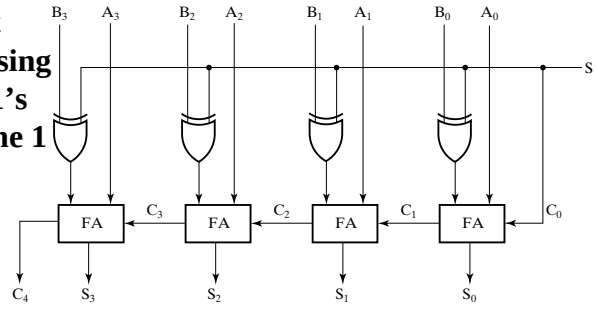
- **Example 1: 1101**
  **+0011**

- **Example 2: 1101**
  **–0011**



## 2's Complement Adder/Subtractor

- **Subtraction can be done by addition of the 2's Complement.**
  1. **Complement each bit (1's Complement.)**
  2. **Add 1 to the result.**
- **The circuit shown computes A + B and A – B:**
- **For S = 1, subtract, the 2's complement of B is formed by using XORs to form the 1's comp and adding the 1 applied to $C_0$.**
- **For S = 0, add, B is passed through unchanged**



# Overflow Detection

- ***Overflow*** **occurs if *n* + 1 bits are required to contain the result from an n-bit addition or subtraction**
- **Overflow can occur for:**
  - **Addition of two operands with the same sign**
  - **Subtraction of operands with different signs**
- **Signed number overflow cases with correct result sign**

$$\begin{array}{rrrr} 0 & 0 & 1 & 1 \\ +\underline{0} & -\underline{1} & -\underline{0} & +\underline{1} \\ 0 & 0 & 1 & 1 \end{array}$$

- **Detection can be performed by examining the result signs which should match the signs of the top operand**





# Overflow Detection

- **Signed number cases with carries $C_n$ and $C_{n-1}$ shown for correct result signs:**

$$\begin{array}{rrrr} 0 & 0\,0 & 0\,1 & 1\,1 \\ 0 & 0 & 1 & 1 \\ +\underline{0} & -\underline{1} & -\underline{0} & +\underline{1} \\ 0 & 0 & 1 & 1 \end{array}$$

- **Signed number cases with carries shown for erroneous result signs (indicating overflow):**

$$\begin{array}{rrrr} 0 & 1\,0 & 1\,1 & 0\,1 \\ 0 & 0 & 1 & 1 \\ +\underline{0} & -\underline{1} & -\underline{0} & +\underline{1} \\ 1 & 1 & 0 & 0 \end{array}$$

- **Simplest way to implement overflow $V = C_n \oplus C_{n-1}$**
- **This works correctly only if 1's complement and the addition of the carry in of 1 is used to implement the complementation! Otherwise fails for – 10 ... 0**

## Other Arithmetic Functions

- **Convenient to design the functional blocks by *contraction* - removal of redundancy from circuit to which input fixing has been applied**
- **Functions**
  - **Incrementing**
  - **Decrementing**
  - **Multiplication by Constant**
  - **Division by Constant**
  - **Zero Fill and Extension**



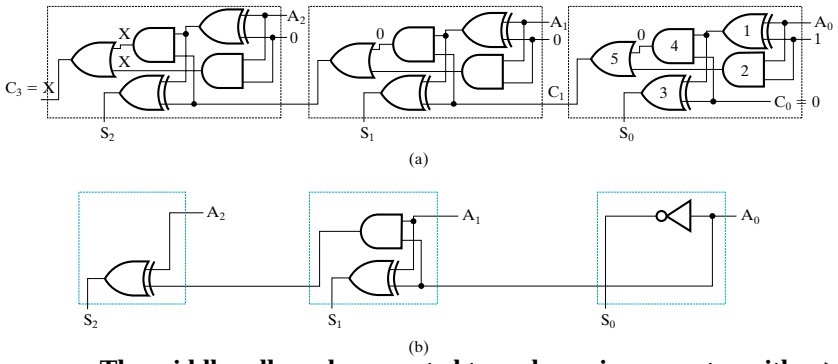


## Design by Contraction

- **Contraction is a technique for simplifying the logic in a functional block to implement a different function**
  - **The new function must be realizable from the original function by applying rudimentary functions to its inputs**
  - **Contraction is treated here only for application of 0s and 1s (not for X and $\overline{X}$)**
  - **After application of 0s and 1s, equations or the logic diagram are simplified by using rules given on pages 224 - 225 of the text.**

# Design by Contraction Example

- **Contraction of a ripple carry adder to incrementer for *n* = 3**
  - **Set B = 001**




  - **The middle cell can be repeated to make an incrementer with *n* > 3.**



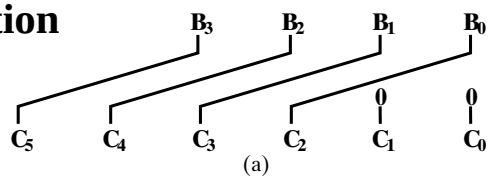


# Incrementing & Decrementing

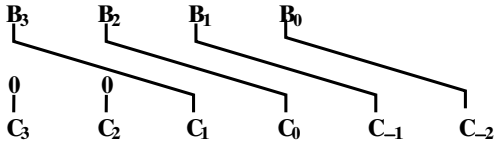
- ***Incrementing***
  - **Adding a fixed value to an arithmetic variable**
  - **Fixed value is often 1, called *counting (up*)**
  - **Examples: A + 1, B + 4**
  - **Functional block is called *incrementer***
- ***Decrementing***
  - **Subtracting a fixed value from an arithmetic variable**
  - **Fixed value is often 1, called *counting (down*)**
  - **Examples: A – 1, B – 4**
  - **Functional block is called *decrementer***

# Multiplication/Division by $2^n$

- **(a) Multiplication by 100**
  - **Shift left by 2**
- **(b) Division by 100**
  - **Shift right by 2**
  - **Remainder preserved**



(a)



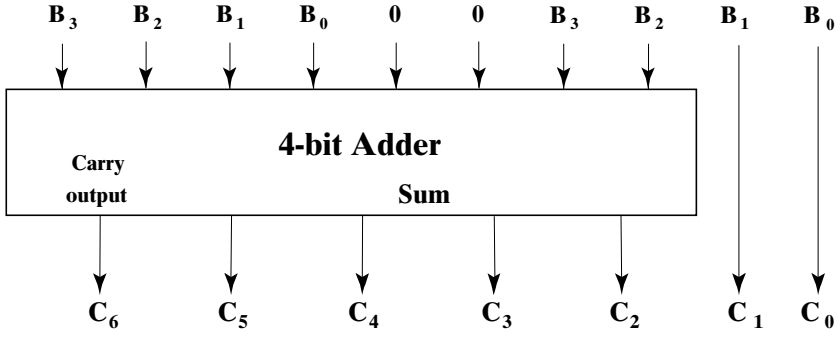
(b)





# Multiplication by a Constant

- **Multiplication of B(3:0) by 101**
- **See text Figure 513 (a) for contraction**



## Zero Fill

- ***Zero fill*** **- filling an *m*-bit operand with 0s to become an *n*-bit operand with *n* > *m***
- **Filling usually is applied to the MSB end of the operand, but can also be done on the LSB end**
- **Example: 11110101 filled to 16 bits**
  - **MSB end: 0000000011110101**
  - **LSB end: 1111010100000000**





## Extension

- ***Extension*** **- increase in the number of bits at the MSB end of an operand by using a complement representation**
  - **Copies the MSB of the operand into the new positions**
  - **Positive operand example - 01110101 extended to 16 bits:**
    **0000000001110101**
  - **Negative operand example - 11110101 extended to 16 bits:**
    **1111111111110101**

# Terms of Use

- **All (or portions) of this material © 2008 by Pearson Education, Inc.**
- **Permission is given to incorporate this material or adaptations thereof into classroom presentations and handouts to instructors in courses adopting the latest edition of Logic and Computer Design Fundamentals as the course textbook.**
- **These materials or adaptations thereof are not to be sold or otherwise offered for consideration.**
- **This Terms of Use slide or page is to be included within the original materials or any adaptations thereof.**