Chapter 5 :: Digital Building Blocks

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris

Copyright © 2007 Elsevier

Chapter 5 :: Topics

- **Introduction**
- **Arithmetic Circuits**
- **Number Systems**
- **Sequential Building Blocks**
- **Memory Arrays**
- **Logic Arrays**

Copyright © 2007 Elsevier

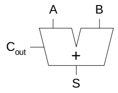
Introduction

- Digital building blocks:
  - Gates, multiplexers, decoders, registers, arithmetic circuits, counters, memory arrays, logic arrays
- Building blocks demonstrate hierarchy, modularity, and regularity:
  - Hierarchy of simpler components
  - Well-defined interfaces and functions
  - Regular structure easily extended to different sizes
- Will use many of these building blocks to build a microprocessor in Chapter 7

Copyright © 2007 Elsevier

1-Bit Adders

Half Adder

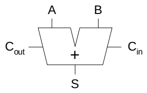


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

S =

C_{out} =

Full Adder

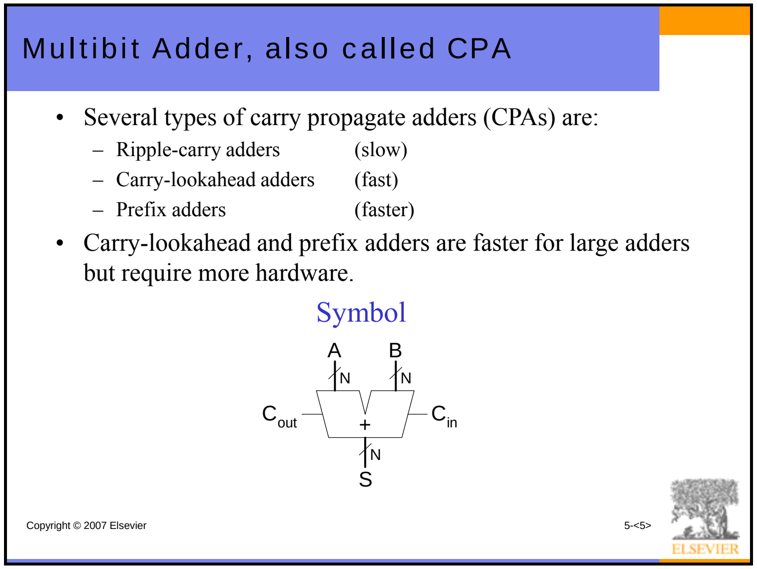


C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

S =

C_{out} =

Copyright © 2007 Elsevier



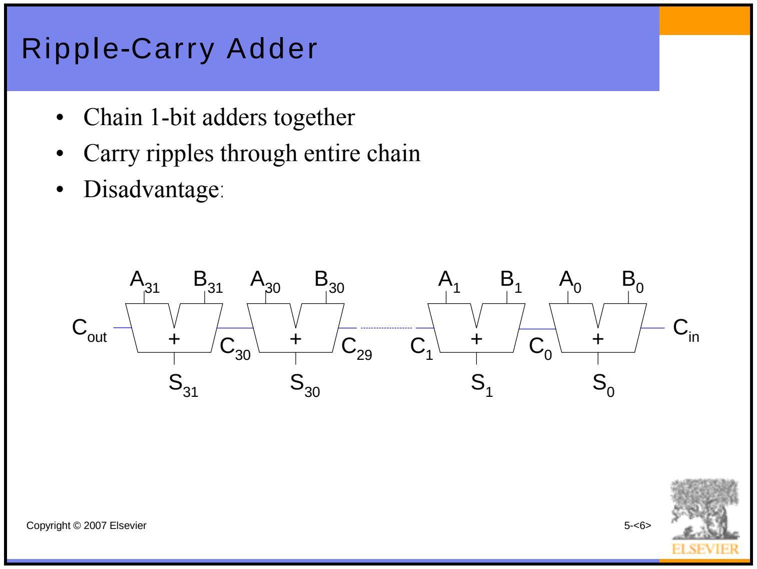
Multibit Adder, also called CPA

- Several types of carry propagate adders (CPAs) are:
  - Ripple-carry adders (slow)
  - Carry-lookahead adders (fast)
  - Prefix adders (faster)
- Carry-lookahead and prefix adders are faster for large adders but require more hardware.

Symbol

A
B
N
N
C_{out}
+
C_{in}
N
S

Copyright © 2007 Elsevier

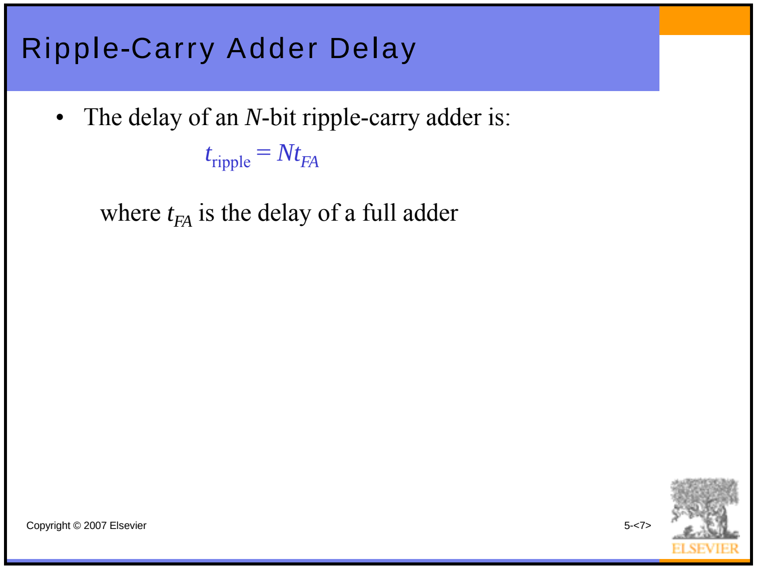


Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage:

A_{31}
B_{31}
A_{30}
B_{30}
A_1
B_1
A_0
B_0
C_{out}
+
C_{30}
+
C_{29}
C_1
+
C_0
+
C_{in}
S_{31}
S_{30}
S_1
S_0

Copyright © 2007 Elsevier



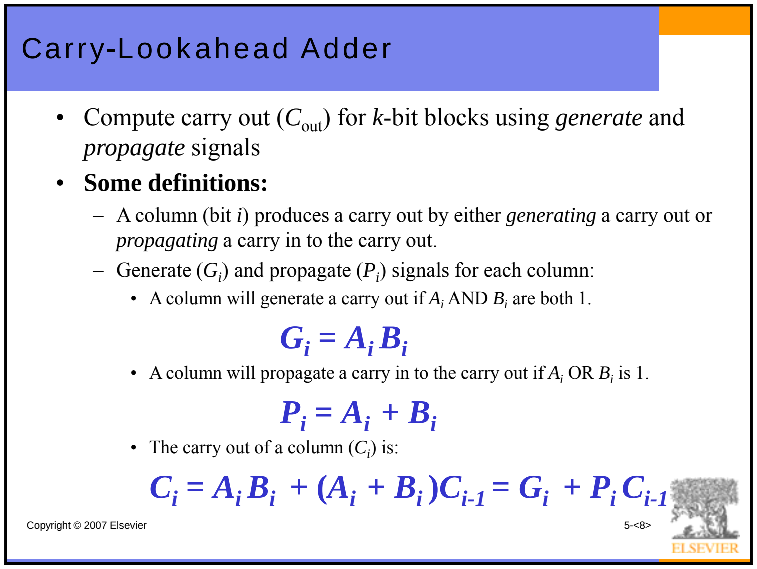
Ripple-Carry Adder Delay

- The delay of an *N*-bit ripple-carry adder is:

$$t_{\text{ripple}} = N t_{FA}$$

where t_{FA} is the delay of a full adder

Copyright © 2007 Elsevier



Carry-Lookahead Adder

- Compute carry out (C_{out}) for *k*-bit blocks using *generate* and *propagate* signals
- **Some definitions:**
  - A column (bit *i*) produces a carry out by either *generating* a carry out or *propagating* a carry in to the carry out.
  - Generate (G_i) and propagate (P_i) signals for each column:
    - A column will generate a carry out if A_i AND B_i are both 1.

$$G_i = A_i B_i$$

    - A column will propagate a carry in to the carry out if A_i OR B_i is 1.

$$P_i = A_i + B_i$$

    - The carry out of a column (C_i) is:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

Copyright © 2007 Elsevier

Carry-Lookahead Addition

- Step 1: compute *generate* (G) and *propagate* (P) signals for columns (single bits)
- Step 2: compute G and P for k-bit blocks
- Step 3: C_{in} propagates through each k-bit propagate/generate block

Copyright © 2007 Elsevier

Carry-Lookahead Adder

- For example, we can calculate generate and propagate signals for a 4-bit block ($G_{3:0}$ and $P_{3:0}$) :
  - A 4-bit block will generate a carry out if column 3 generates a carry (G_3) or if column 3 propagates a carry (P_3) that was generated or propagated in a previous column as described by the following equation:

$$G_{3:0} = G_3 + P_3\,(G_2 + P_2\,(G_1 + P_1G_0\,)$$

  - A 4-bit block will propagate a carry in to the carry out if all of the columns propagate the carry:

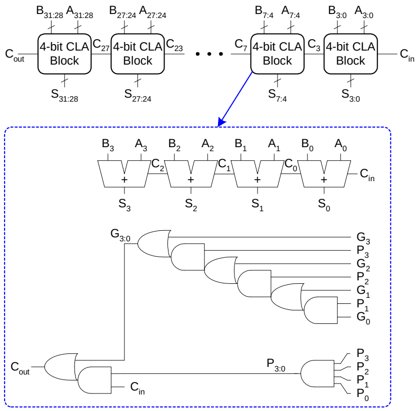
$$P_{3:0} = P_3P_2\,P_1P_0$$

  - The carry out of the 4-bit block (C_i) is:

$$C_i = G_{i:j}\; + P_{i:j}\,C_{i-1}$$

Copyright © 2007 Elsevier

32-bit CLA with 4-bit blocks



Copyright © 2007 Elsevier

Carry-Lookahead Adder Delay

- Delay of an N-bit carry-lookahead adder with k-bit blocks:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

where

- t_{pg} : delay of the column generate and propagate gates
- t_{pg_block} : delay of the block generate and propagate gates
- t_{AND_OR} : delay from C_{in} to C_{out} of the final AND/OR gate in the k-bit CLA block

- An N-bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$

Copyright © 2007 Elsevier

Prefix Adder

- Computes the carry in (C_{i-1}) for each of the columns as fast as possible and then computes the sum:

$$S_i = (A_i \oplus B_i) \oplus C_i$$

- Computes *G* and *P* for 1-bit, then 2-bit blocks, then 4-bit blocks, then 8-bit blocks, etc. until the carry in (generate signal) is known for each column
- Has $\log_2 N$ stages

Prefix Adder

- A carry in is produced by being either *generated* in a column or *propagated* from a previous column.
- Define column -1 to hold C_{in}, so

$$G_{-1} = C_{\text{in}},\ P_{-1} = 0$$

- Then, the carry in to column *i* = the carry out of column *i-1*:

$$C_{i-1} = G_{i-1:-1}$$

$G_{i-1:-1}$ is the generate signal spanning columns *i*-1 to -1.

There will be a carry out of column *i*-1 (C_{i-1}) if the block spanning columns *i*-1 through -1 generates a carry.

- Thus, we can rewrite the sum equation as:

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

- **Goal:** Quickly compute $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, (These are called the *prefixes)*

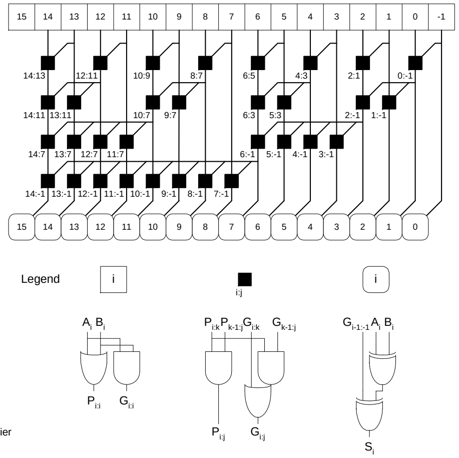
Prefix Adder

- The generate and propagate signals for a block spanning bits *i*:*j* are:

$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$
$$P_{i:j} = P_{i:k}P_{k-1:j}$$

- In words, these prefixes describe that:
  - A block will generate a carry if the upper part (*i*:*k*) generates a carry or if the upper part propagates a carry generated in the lower part (*k*-1:*j*)
  - A block will propagate a carry if both the upper and lower parts propagate the carry.

Prefix Adder Schematic



Prefix Adder Delay

- The delay of an *N*-bit prefix adder is:

  $t_{PA} = t_{pg} + \log_2 N(t_{pg\_\mathrm{prefix}}) + t_{\mathrm{XOR}}$

  where
  - t_{pg} is the delay of the column generate and propagate gates (AND or OR gate)
  - t_{pg_prefix} is the delay of the black prefix cell (AND-OR gate)

Copyright © 2007 Elsevier

Adder Delay Comparisons

- Compare the delay of 32-bit ripple-carry, carry-lookahead, and prefix adders. The carry-lookahead adder has 4-bit blocks. Assume that each two-input gate delay is 100 ps and the full adder delay is 300 ps.

t_{ripple} =

=

t_{CLA} =

=

=

t_{PA} =

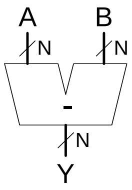
=

=

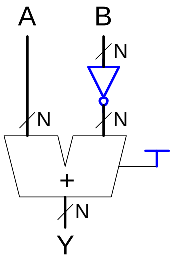
Copyright © 2007 Elsevier

Subtracter

Symbol



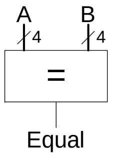
Implementation



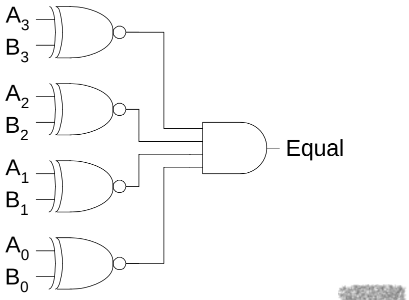
Copyright © 2007 Elsevier

Comparator: Equality

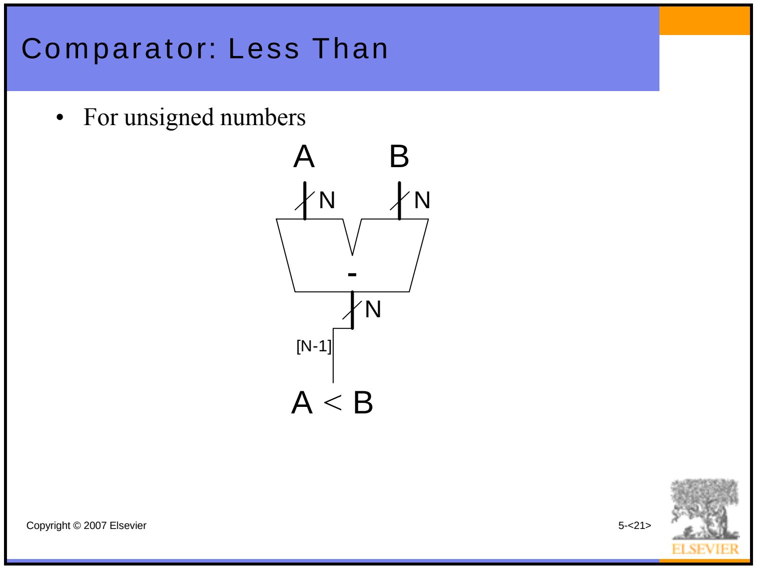
Symbol



Implementation



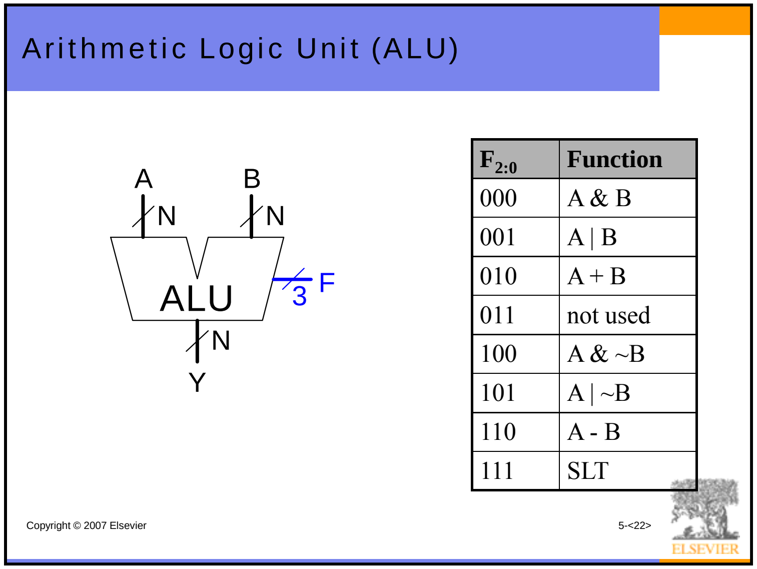
Copyright © 2007 Elsevier



Comparator: Less Than

- For unsigned numbers

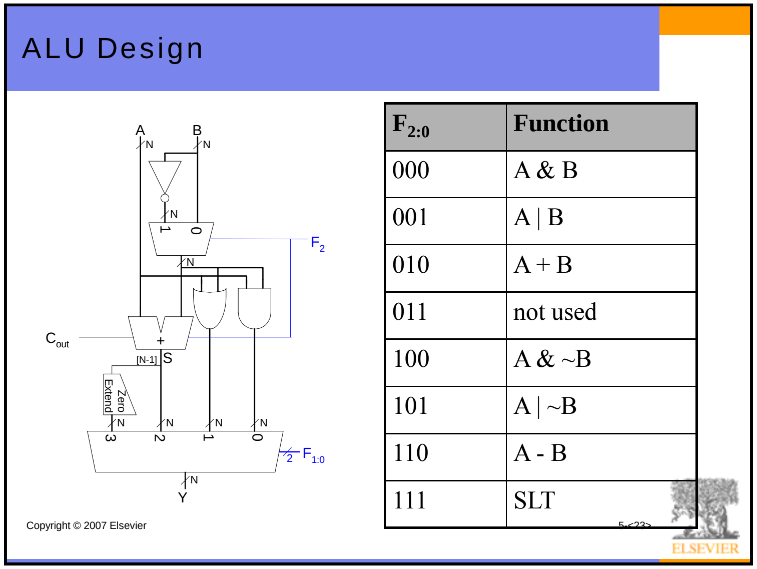
Copyright © 2007 Elsevier



Arithmetic Logic Unit (ALU)

$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

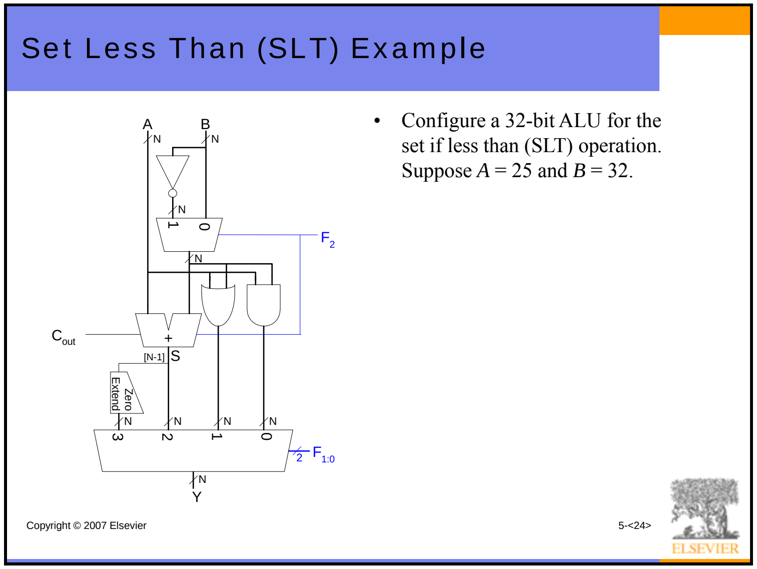
Copyright © 2007 Elsevier



ALU Design

$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Copyright © 2007 Elsevier

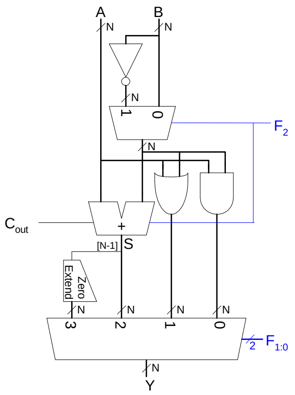


Set Less Than (SLT) Example

- Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose *A* = 25 and *B* = 32.

Copyright © 2007 Elsevier

Set Less Than (SLT) Example



- Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose $A = 25$ and $B = 32$.
  - A is less than B, so we expect Y to be the 32-bit representation of 1 (0x00000001).
  - For SLT, $F_{2:0} = 111$.
  - $F_2 = 1$ configures the adder unit as a subtracter. So 25 - 32 = -7.
  - The two's complement representation of -7 has a 1 in the most significant bit, so $S_{31} = 1$.
  - With $F_{1:0} = 11$, the final multiplexer selects $Y = S_{31}$ (zero extended) = 0x00000001.

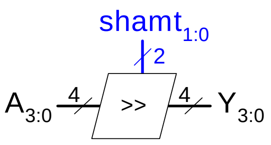
Copyright © 2007 Elsevier

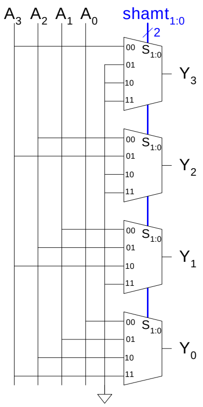
Shifters

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
  - Ex: 11001 >> 2 =
  - Ex: 11001 << 2 =
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: 11001 >>> 2 =
  - Ex: 11001 <<< 2 =
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 =
  - Ex: 11001 ROL 2 =

Copyright © 2007 Elsevier

Shifter Design



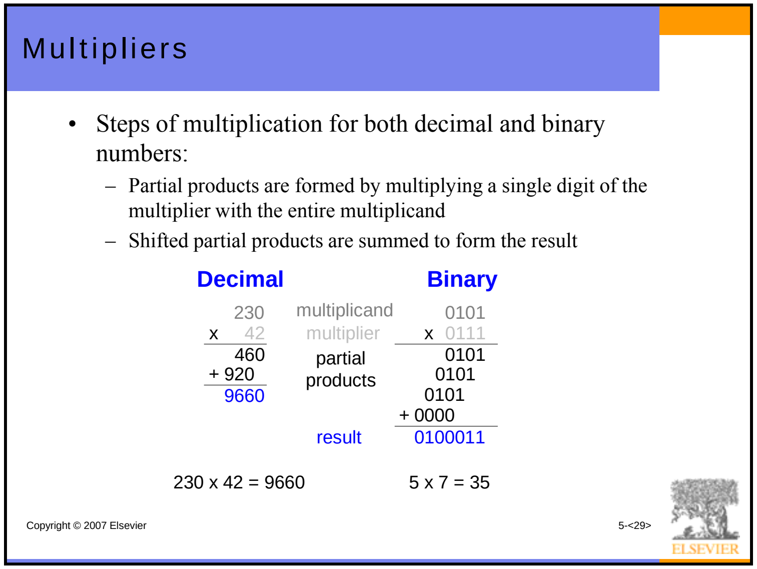


Copyright © 2007 Elsevier

Shifters as Multipliers and Dividers

- A left shift by N bits multiplies a number by 2^N
  - Ex: 00001 << 2 = 00100 ($1 \times 2^2 = 4$)
  - Ex: 11101 << 2 = 10100 ($-3 \times 2^2 = -12$)
- The arithmetic right shift by N divides a number by 2^N
  - Ex: 01000 >>> 2 = 00010 ($8 \div 2^2 = 2$)
  - Ex: 10000 >>> 2 = 11100 ($-16 \div 2^2 = -4$)

Copyright © 2007 Elsevier

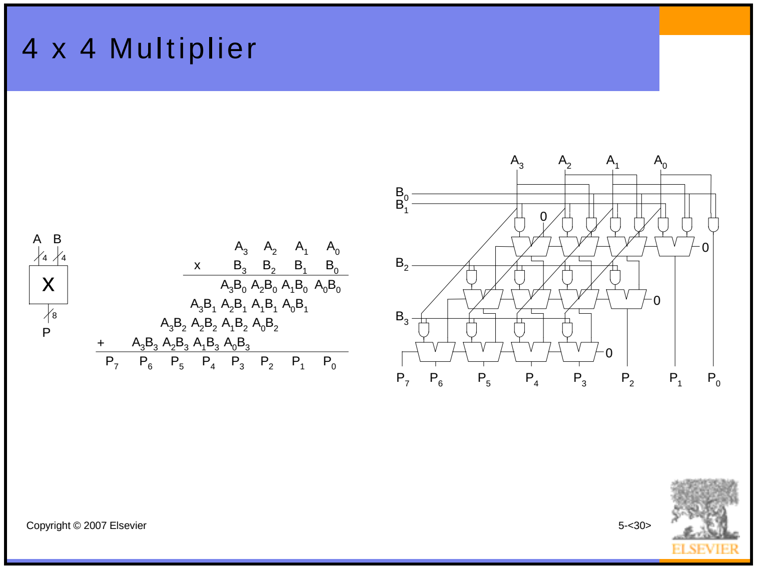


Multipliers

- Steps of multiplication for both decimal and binary numbers:
  - Partial products are formed by multiplying a single digit of the multiplier with the entire multiplicand
  - Shifted partial products are summed to form the result

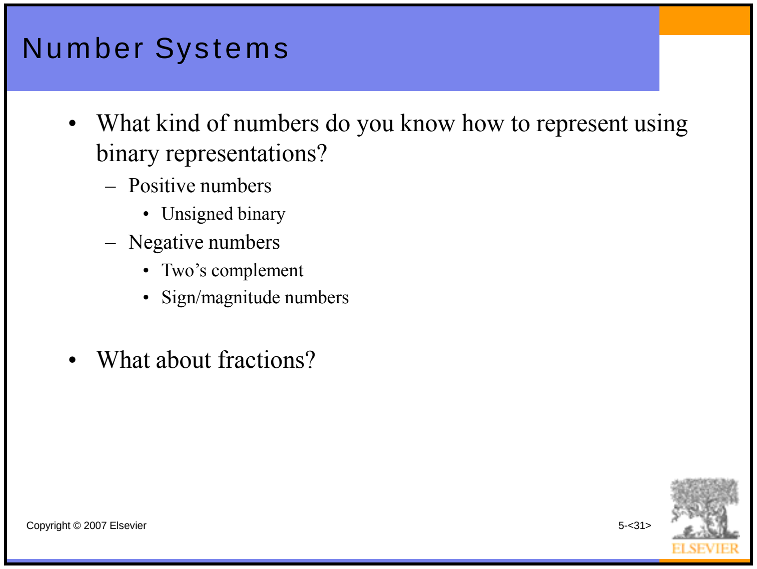
```
   Decimal                       Binary
     230      multiplicand        0101
 x    42       multiplier      x  0111
     460                          0101
 +  920         partial          0101
    9660        products        0101
                              + 0000
                  result        0100011

 230 x 42 = 9660               5 x 7 = 35
```

Copyright © 2007 Elsevier



4 x 4 Multiplier

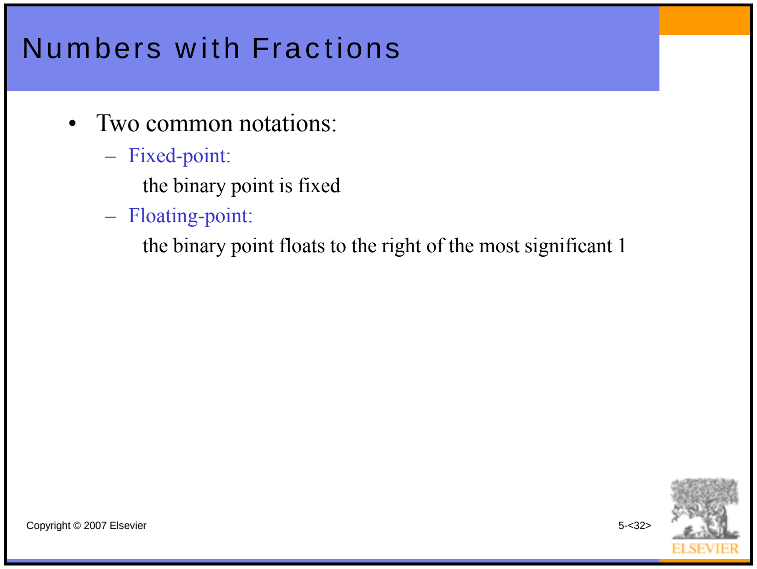
Copyright © 2007 Elsevier



Number Systems

- What kind of numbers do you know how to represent using binary representations?
  - Positive numbers
    - Unsigned binary
  - Negative numbers
    - Two’s complement
    - Sign/magnitude numbers
- What about fractions?

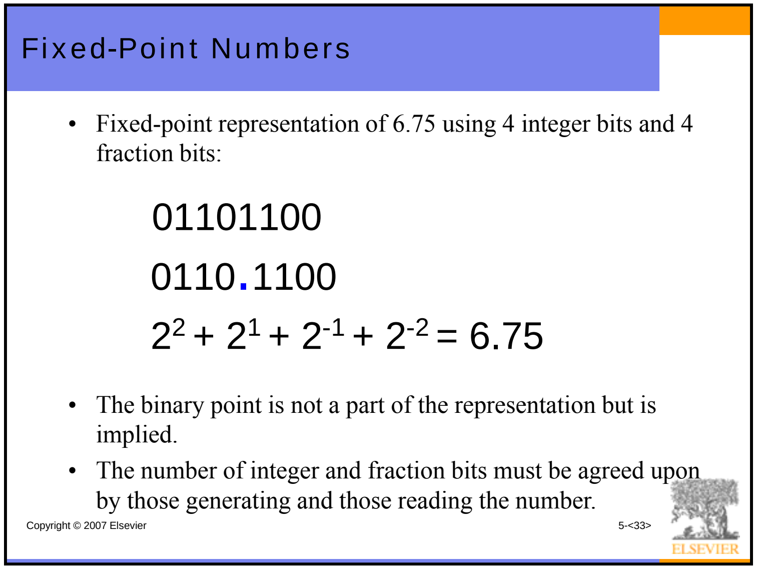
Copyright © 2007 Elsevier



Numbers with Fractions

- Two common notations:
  - Fixed-point:
    the binary point is fixed
  - Floating-point:
    the binary point floats to the right of the most significant 1

Copyright © 2007 Elsevier



Fixed-Point Numbers

- Fixed-point representation of 6.75 using 4 integer bits and 4 fraction bits:

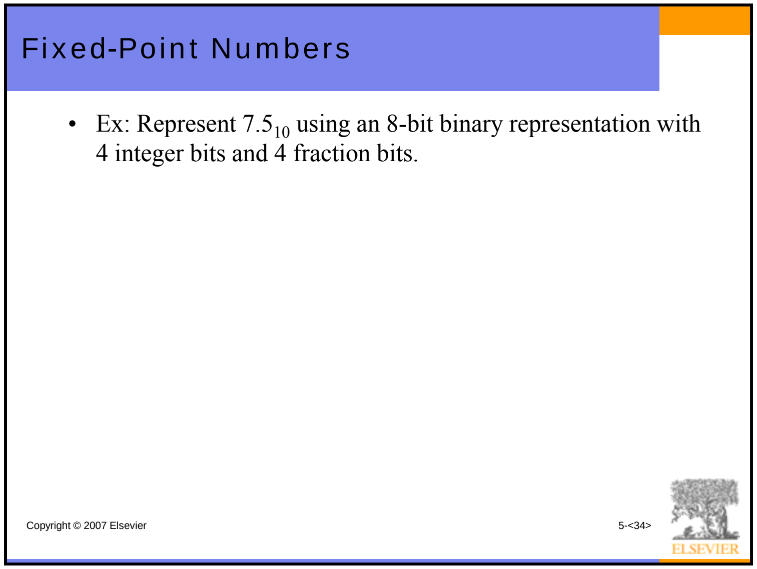
$$01101100$$

$$0110.1100$$

$$2^{2} + 2^{1} + 2^{-1} + 2^{-2} = 6.75$$

- The binary point is not a part of the representation but is implied.
- The number of integer and fraction bits must be agreed upon by those generating and those reading the number.

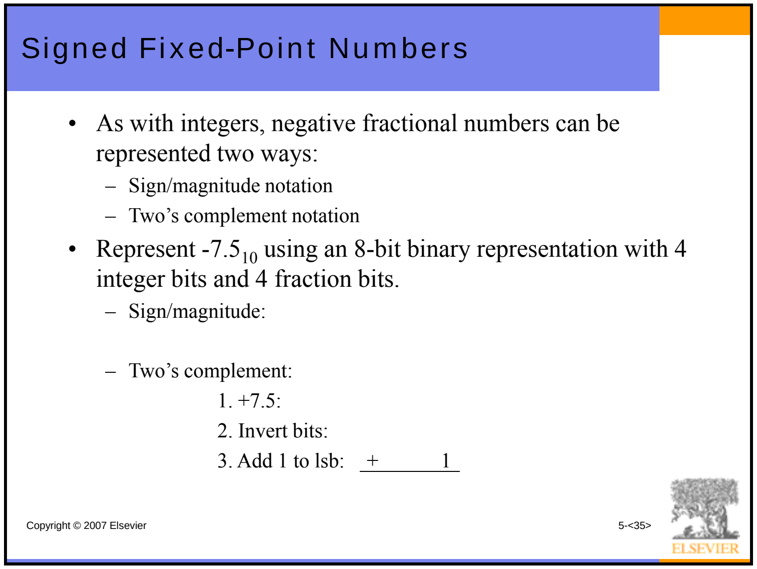
Copyright © 2007 Elsevier



Fixed-Point Numbers

- Ex: Represent 7.5_{10} using an 8-bit binary representation with 4 integer bits and 4 fraction bits.

Copyright © 2007 Elsevier



Signed Fixed-Point Numbers

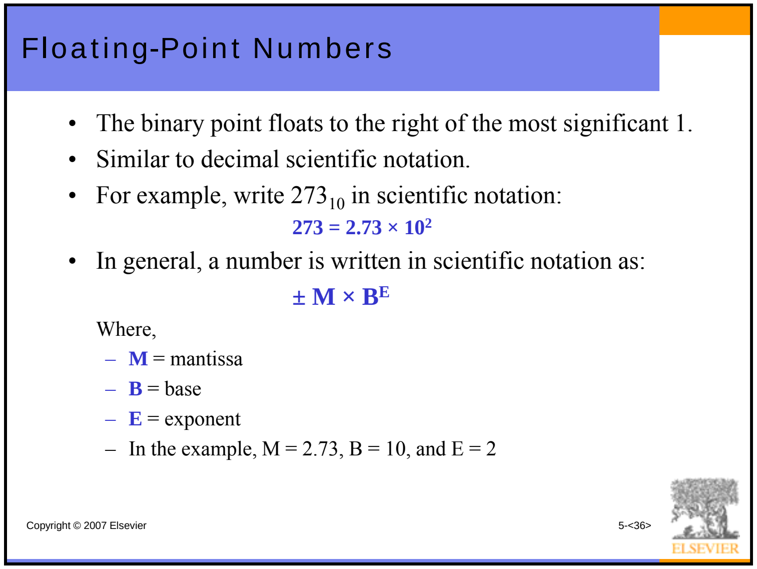
- As with integers, negative fractional numbers can be represented two ways:
  - Sign/magnitude notation
  - Two's complement notation
- Represent -7.5_{10} using an 8-bit binary representation with 4 integer bits and 4 fraction bits.
  - Sign/magnitude:
  - Two's complement:

    1. +7.5:

    2. Invert bits:

    3. Add 1 to lsb: $\underline{+ \quad\quad\quad 1}$

Copyright © 2007 Elsevier



Floating-Point Numbers

- The binary point floats to the right of the most significant 1.
- Similar to decimal scientific notation.
- For example, write 273_{10} in scientific notation:

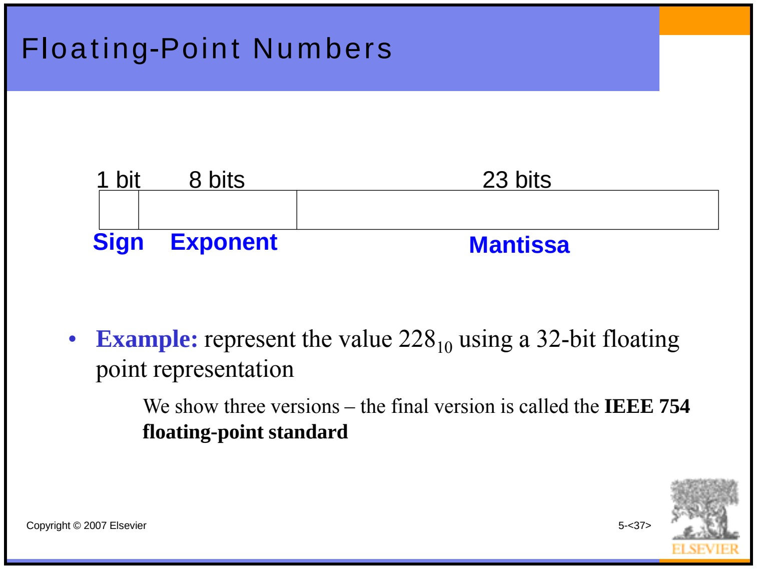
  $273 = 2.73 \times 10^{2}$

- In general, a number is written in scientific notation as:

  $\pm M \times B^{E}$

  Where,
  - **M** = mantissa
  - **B** = base
  - **E** = exponent
  - In the example, M = 2.73, B = 10, and E = 2

Copyright © 2007 Elsevier



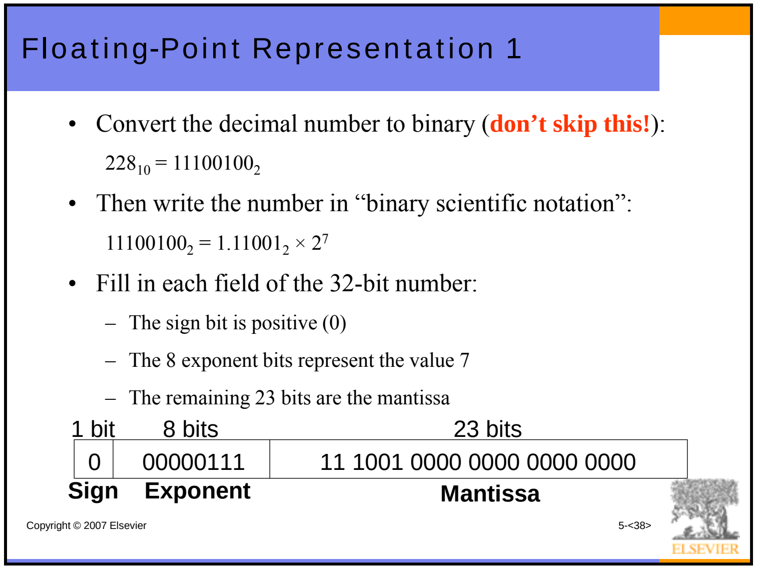
Floating-Point Numbers

1 bit	8 bits	23 bits
Sign	Exponent	Mantissa

- **Example:** represent the value 228_{10} using a 32-bit floating point representation

  We show three versions – the final version is called the **IEEE 754 floating-point standard**

Copyright © 2007 Elsevier



Floating-Point Representation 1

- Convert the decimal number to binary (**don’t skip this!**):

  $228_{10} = 11100100_2$

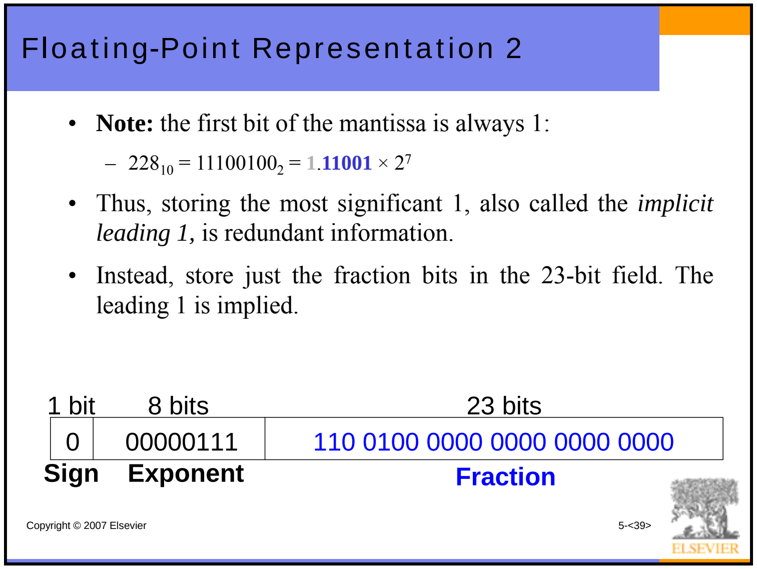
- Then write the number in “binary scientific notation”:

  $11100100_2 = 1.11001_2 \times 2^7$

- Fill in each field of the 32-bit number:
  - The sign bit is positive (0)
  - The 8 exponent bits represent the value 7
  - The remaining 23 bits are the mantissa

1 bit	8 bits	23 bits
0	00000111	11 1001 0000 0000 0000 0000
Sign	**Exponent**	**Mantissa**

Copyright © 2007 Elsevier

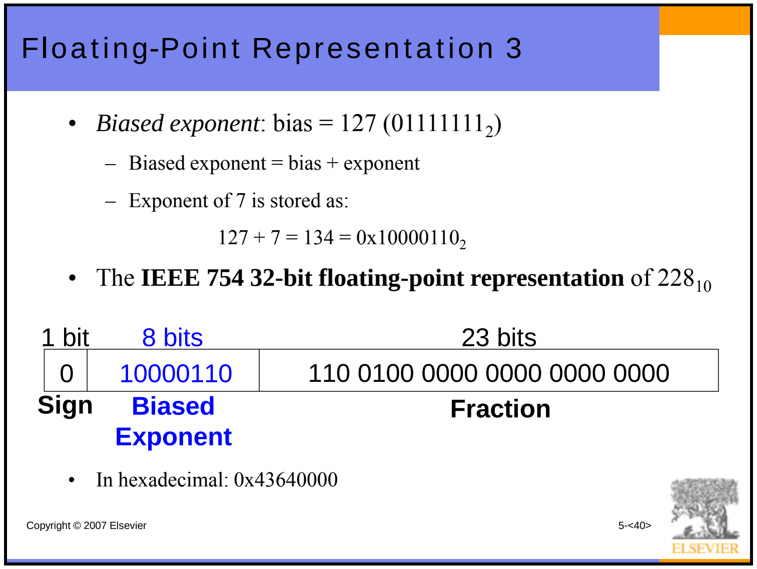


Floating-Point Representation 2

- **Note:** the first bit of the mantissa is always 1:
  - $228_{10} = 11100100_2 = 1.\mathbf{11001} \times 2^7$
- Thus, storing the most significant 1, also called the *implicit leading 1*, is redundant information.
- Instead, store just the fraction bits in the 23-bit field. The leading 1 is implied.

1 bit	8 bits	23 bits
0	00000111	110 0100 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

Copyright © 2007 Elsevier



Floating-Point Representation 3

- *Biased exponent*: bias = 127 (01111111_2)
  - Biased exponent = bias + exponent
  - Exponent of 7 is stored as:

    $127 + 7 = 134 = 0x10000110_2$

- The **IEEE 754 32-bit floating-point representation** of 228_{10}

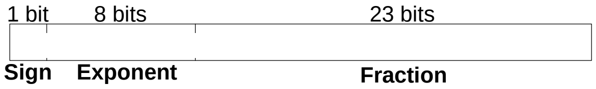
1 bit	8 bits	23 bits
0	10000110	110 0100 0000 0000 0000 0000
Sign	**Biased Exponent**	**Fraction**

- In hexadecimal: 0x43640000

Copyright © 2007 Elsevier

Floating-Point Example

- Write the value -58.25_{10} using the IEEE 754 32-bit floating-point standard.
- First, convert the decimal number to binary:
  - 58.25_{10} =
- Next, fill in each field in the 32-bit number:
  - Sign bit:
  - 8 exponent bits:
  - 23 fraction bits:



1 bit 8 bits 23 bits

Sign Exponent Fraction

- In hexadecimal: 0xC2690000

Copyright © 2007 Elsevier

Floating-Point Numbers: Special Cases

- The IEEE 754 standard includes special cases for numbers that are difficult to represent, such as 0 because it lacks an implicit leading 1.

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

NaN is used for numbers that don't exist, such as √-1 or log(-5).

Copyright © 2007 Elsevier

Floating-Point Number Precision

- Single-Precision:
  - 32-bit notation
  - 1 sign bit, 8 exponent bits, 23 fraction bits
  - bias = 127
- Double-Precision:
  - 64-bit notation
  - 1 sign bit, 11 exponent bits, 52 fraction bits
  - bias = 1023

Copyright © 2007 Elsevier

Floating-Point Numbers: Rounding

- Overflow: number is too large to be represented
- Underflow: number is too small to be represented
- Rounding modes:
  - Down
  - Up
  - Toward zero
  - To nearest
- **Example:** round 1.100101 (1.578125) so that it uses only 3 fraction bits.
  - Down: 1.100
  - Up: 1.101
  - Toward zero: 1.100
  - To nearest: 1.101 (1.625 is closer to 1.578125 than 1.5 is)

Copyright © 2007 Elsevier

Floating-Point Addition

1. Extract exponent and fraction bits
2. Prepend leading 1 to form mantissa
3. Compare exponents
4. Shift smaller mantissa if necessary
5. Add mantissas
6. Normalize mantissa and adjust exponent if necessary
7. Round result
8. Assemble exponent and fraction back into floating-point format

Copyright © 2007 Elsevier

Floating-Point Addition: Example

Add the following floating-point numbers:

0x3FC00000

0x40500000

Copyright © 2007 Elsevier

Floating-Point Addition: Example

1. Extract exponent and fraction bits

1 bit	8 bits	23 bits
0	01111111	100 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

1 bit	8 bits	23 bits
0	10000000	101 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

For first number (N1): S = 0, E = 127, F = .1

For second number (N2): S = 0, E = 128, F = .101

2. Prepend leading 1 to form mantissa

N1: 1.1

N2: 1.101

Copyright © 2007 Elsevier

Floating-Point Addition: Example

3. Compare exponents

   127 – 128 = -1, so shift N1 right by 1 bit

4. Shift smaller mantissa if necessary

   shift N1's mantissa: 1.1 >> 1 = 0.11 $(\times 2^1)$

5. Add mantissas

$$\begin{array}{r} 0.11 \times 2^1 \\ +\ 1.101 \times 2^1 \\ \hline 10.011 \times 2^1 \end{array}$$

Copyright © 2007 Elsevier

Floating-Point Addition: Example

6. Normalize mantissa and adjust exponent if necessary

   $10.011 \times 2^1 = 1.0011 \times 2^2$

7. Round result

   No need (fits in 23 bits)

8. Assemble exponent and fraction back into floating-point format

   $S = 0, E = 2 + 127 = 129 = 10000001_2, F = 001100..$

1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

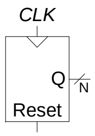
Written in hexadecimal: 0x40980000

Copyright © 2007 Elsevier

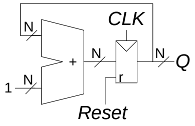
Counters

- Increments on each clock edge.
- Used to cycle through numbers. For example,
  - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001…
- Example uses:
  - Digital clock displays
  - Program counter: keeps track of current instruction executing

Symbol



Implementation

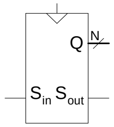


Copyright © 2007 Elsevier

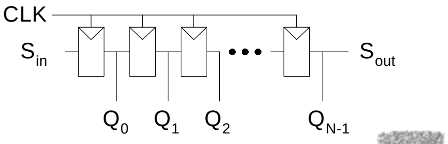
Shift Register

- Shift a new value in on each clock edge
- Shift a value out on each clock edge
- *Serial-to-parallel converter*: converts serial input (S_{in}) to parallel output ($Q_{0:N-1}$)

Symbol:



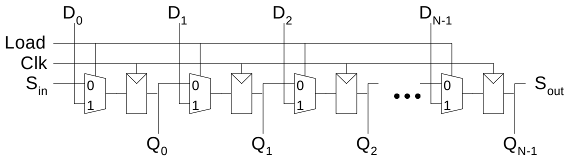
Implementation:



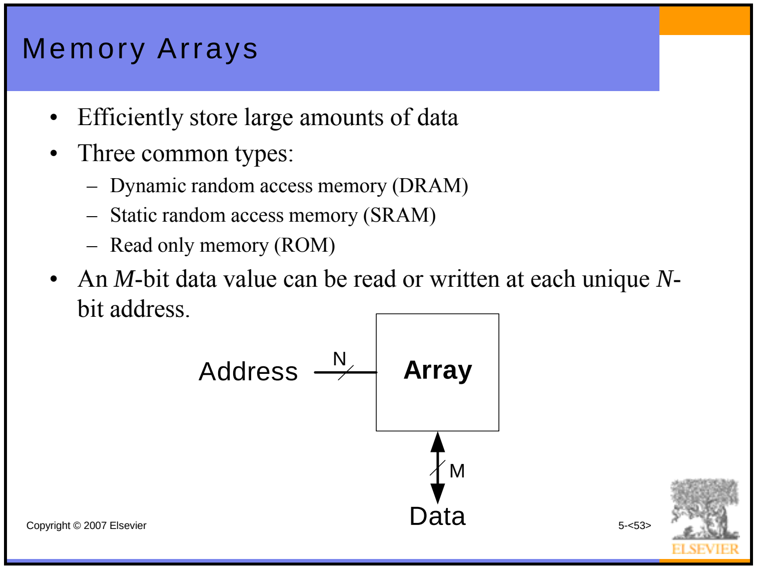
Copyright © 2007 Elsevier

Shift Register with Parallel Load

- When *Load* = 1, acts as a normal *N*-bit register
- When *Load* = 0, acts as a shift register
- Now can act as a *serial-to-parallel converter* (S_{in} to $Q_{0:N-1}$) or a *parallel-to-serial converter* ($D_{0:N-1}$ to S_{out})



Copyright © 2007 Elsevier

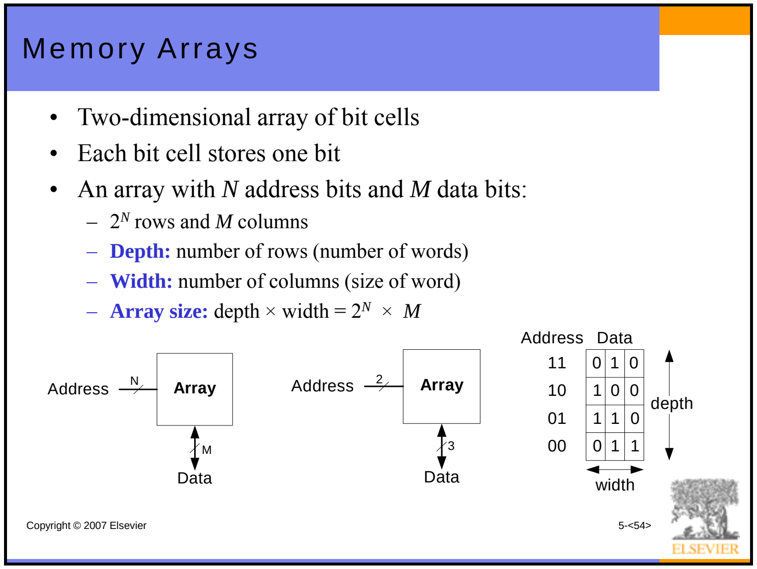


Memory Arrays

- Efficiently store large amounts of data
- Three common types:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)
- An *M*-bit data value can be read or written at each unique *N*-bit address.

Address
N
Array
M
Data

Copyright © 2007 Elsevier



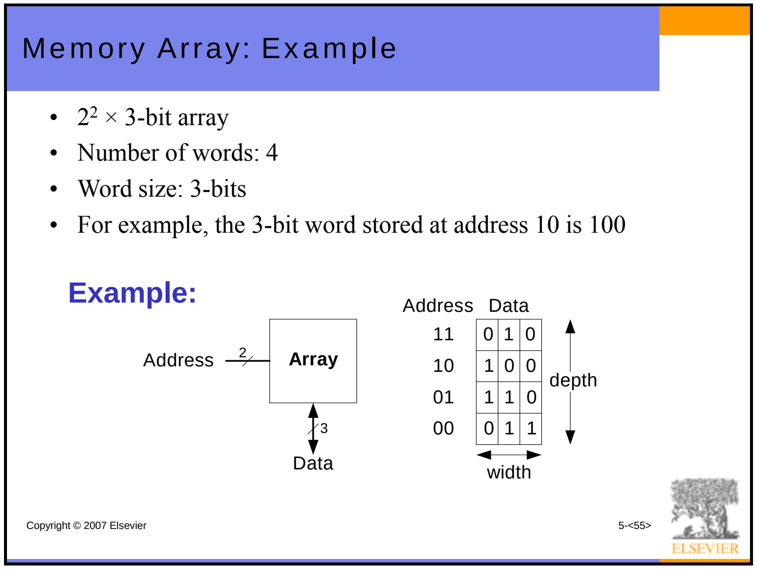
Memory Arrays

- Two-dimensional array of bit cells
- Each bit cell stores one bit
- An array with *N* address bits and *M* data bits:
  - 2^N rows and *M* columns
  - **Depth:** number of rows (number of words)
  - **Width:** number of columns (size of word)
  - **Array size:** depth × width = $2^N \times M$

Address
N
Array
M
Data
Address
2
Array
3
Data
depth
width

Address	Data
11	0 1 0
10	1 0 0
01	1 1 0
00	0 1 1

Copyright © 2007 Elsevier



Memory Array: Example

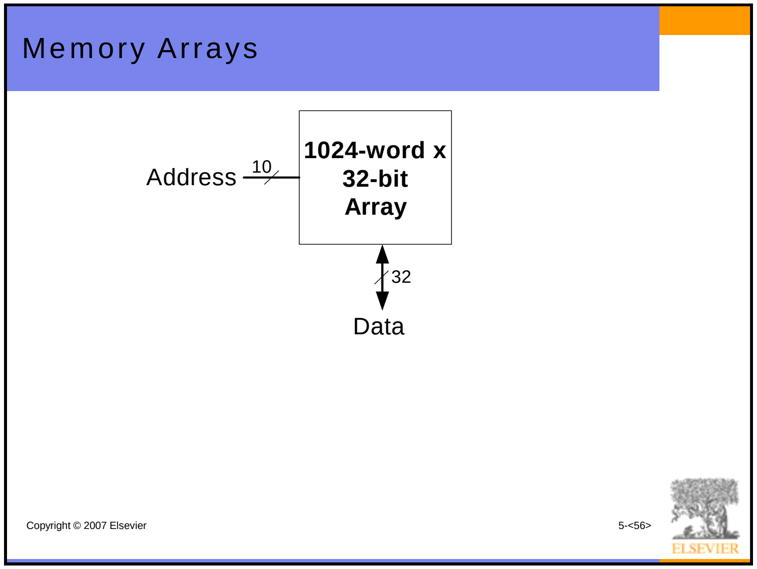
- $2^2 \times 3$-bit array
- Number of words: 4
- Word size: 3-bits
- For example, the 3-bit word stored at address 10 is 100

Example:

Address
2
Array
3
Data
depth
width

Address	Data
11	0 1 0
10	1 0 0
01	1 1 0
00	0 1 1

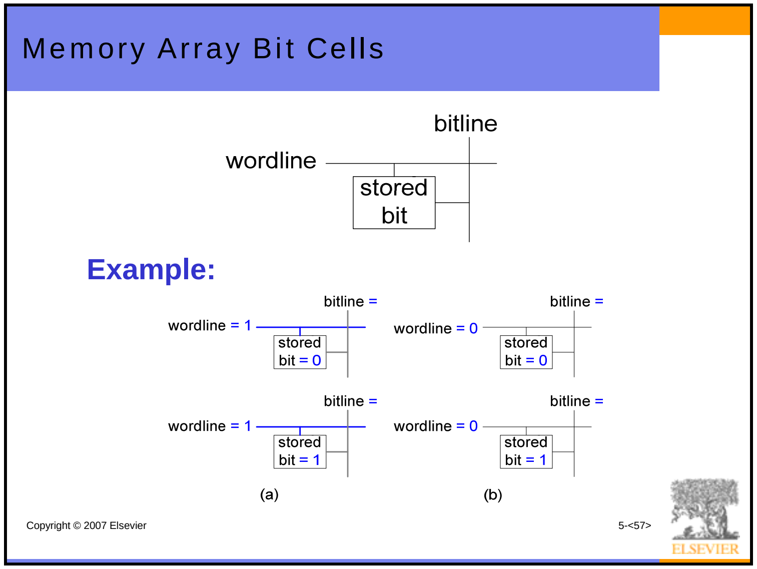
Copyright © 2007 Elsevier



Memory Arrays

Address
10
1024-word x 32-bit Array
32
Data

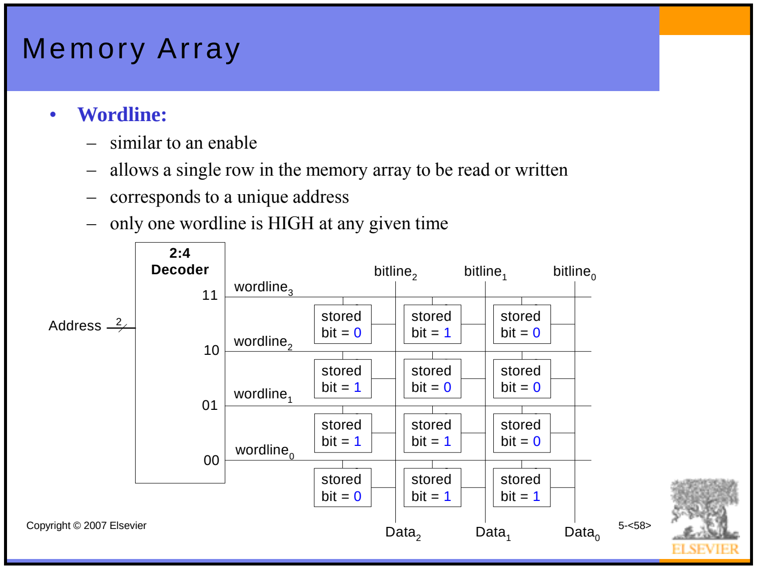
Copyright © 2007 Elsevier



Memory Array Bit Cells

Example:

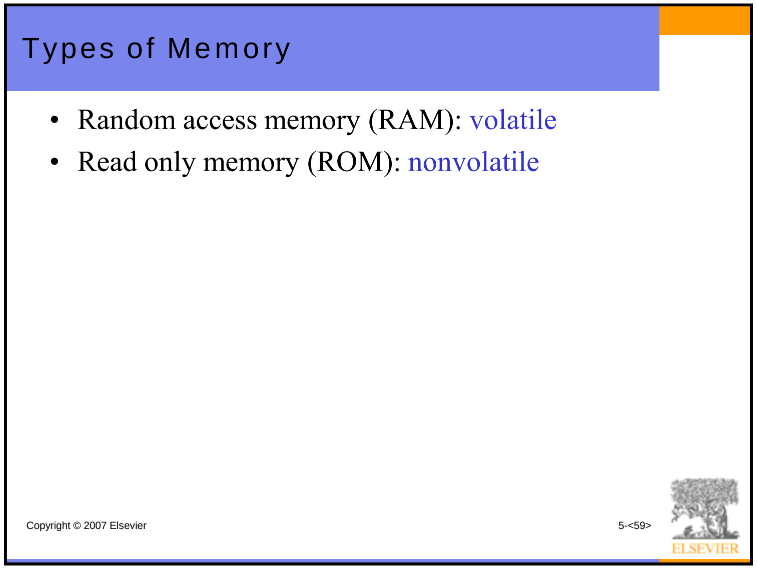
Copyright © 2007 Elsevier



Memory Array

- **Wordline:**
  - similar to an enable
  - allows a single row in the memory array to be read or written
  - corresponds to a unique address
  - only one wordline is HIGH at any given time

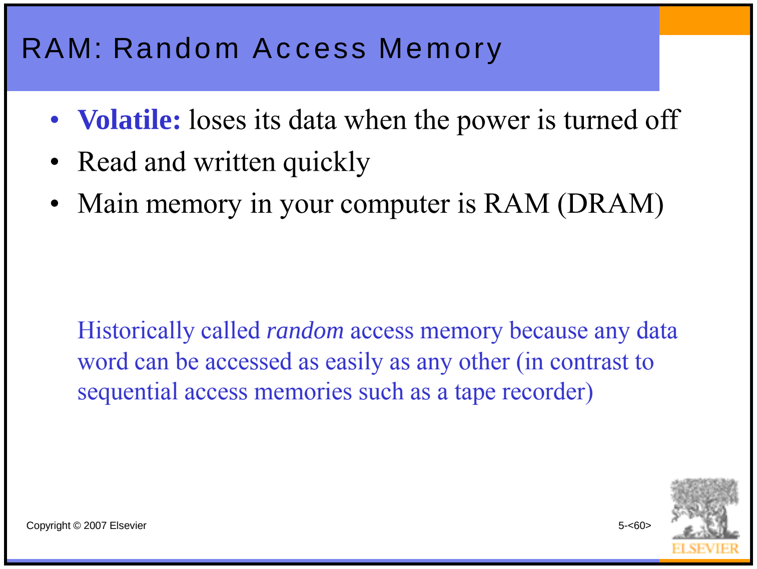
Copyright © 2007 Elsevier



Types of Memory

- Random access memory (RAM): volatile
- Read only memory (ROM): nonvolatile

Copyright © 2007 Elsevier

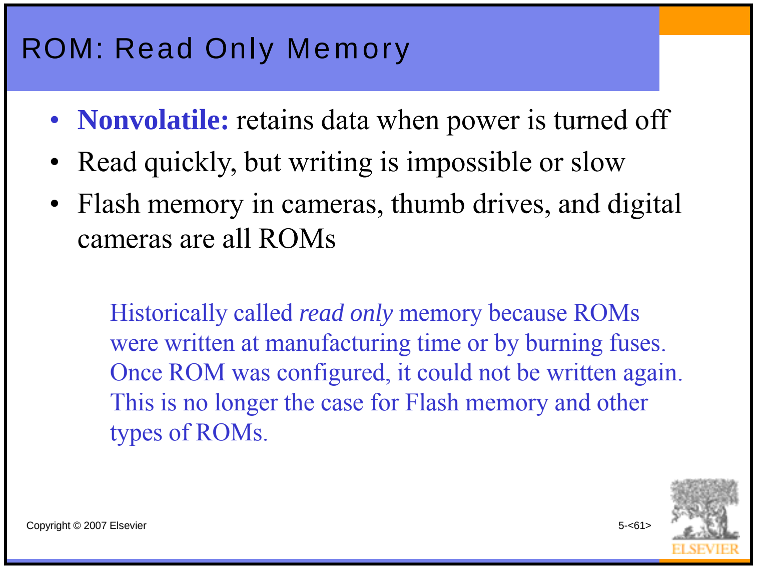


RAM: Random Access Memory

- **Volatile:** loses its data when the power is turned off
- Read and written quickly
- Main memory in your computer is RAM (DRAM)

Historically called *random* access memory because any data word can be accessed as easily as any other (in contrast to sequential access memories such as a tape recorder)

Copyright © 2007 Elsevier

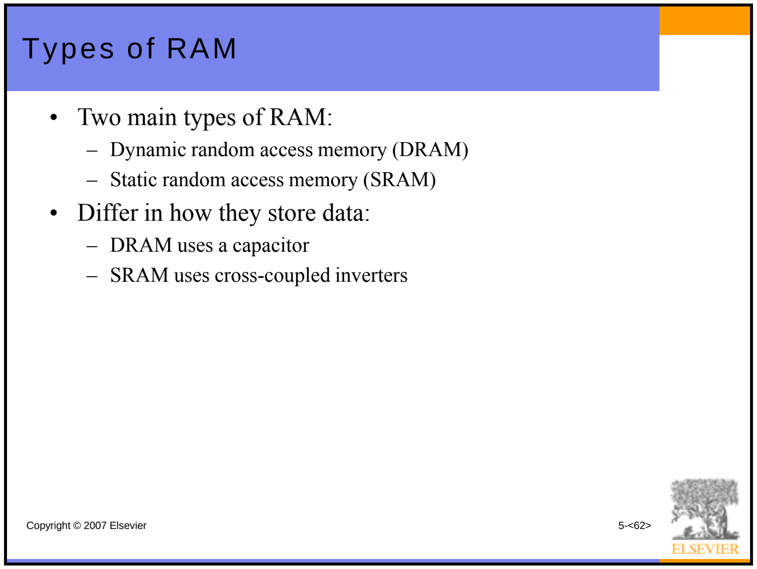


ROM: Read Only Memory

- **Nonvolatile:** retains data when power is turned off
- Read quickly, but writing is impossible or slow
- Flash memory in cameras, thumb drives, and digital cameras are all ROMs

Historically called *read only* memory because ROMs were written at manufacturing time or by burning fuses. Once ROM was configured, it could not be written again. This is no longer the case for Flash memory and other types of ROMs.

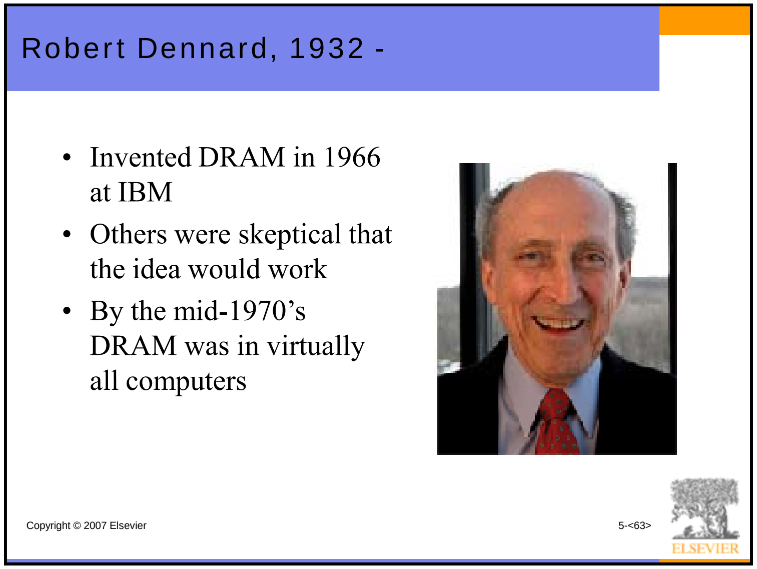
Copyright © 2007 Elsevier



Types of RAM

- Two main types of RAM:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
- Differ in how they store data:
  - DRAM uses a capacitor
  - SRAM uses cross-coupled inverters

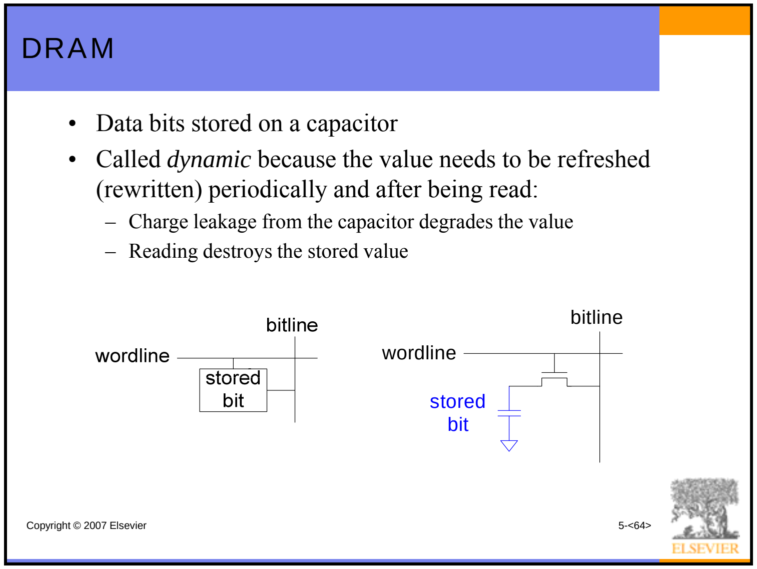
Copyright © 2007 Elsevier



Robert Dennard, 1932 -

- Invented DRAM in 1966 at IBM
- Others were skeptical that the idea would work
- By the mid-1970's DRAM was in virtually all computers

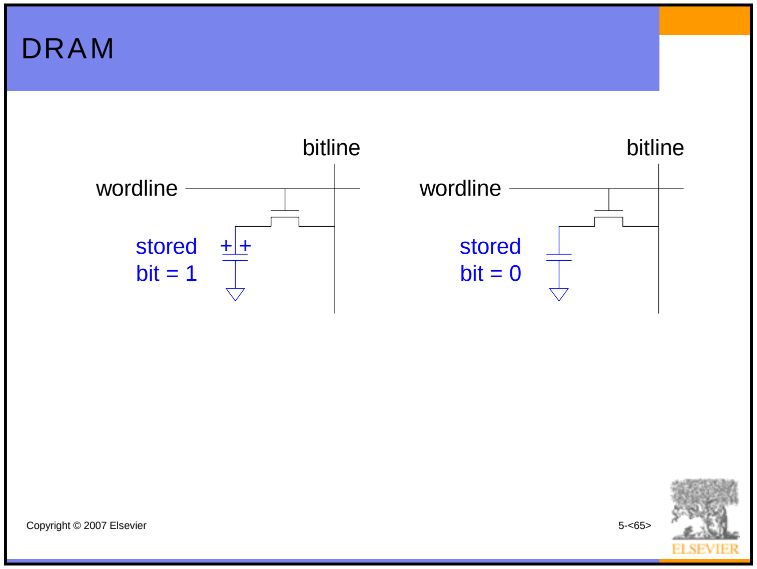
Copyright © 2007 Elsevier



DRAM

- Data bits stored on a capacitor
- Called *dynamic* because the value needs to be refreshed (rewritten) periodically and after being read:
  - Charge leakage from the capacitor degrades the value
  - Reading destroys the stored value

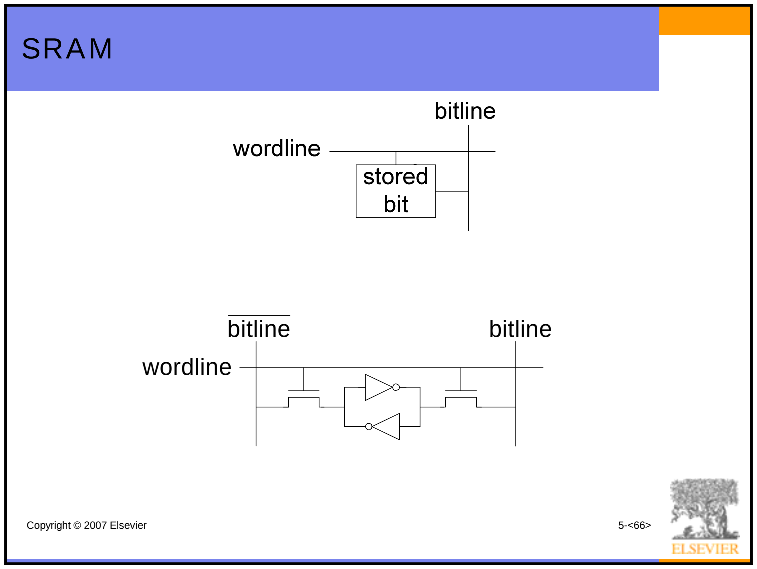
Copyright © 2007 Elsevier



DRAM

Copyright © 2007 Elsevier

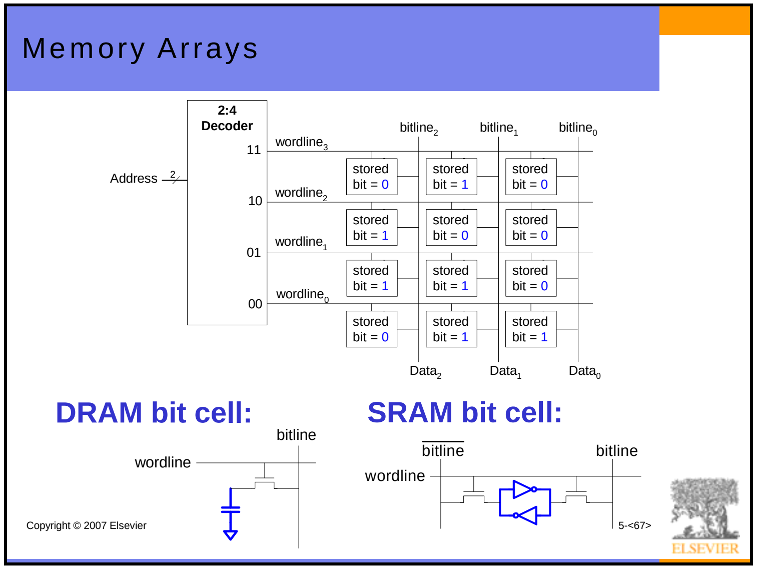
5-<65>



SRAM

Copyright © 2007 Elsevier

5-<66>



Memory Arrays

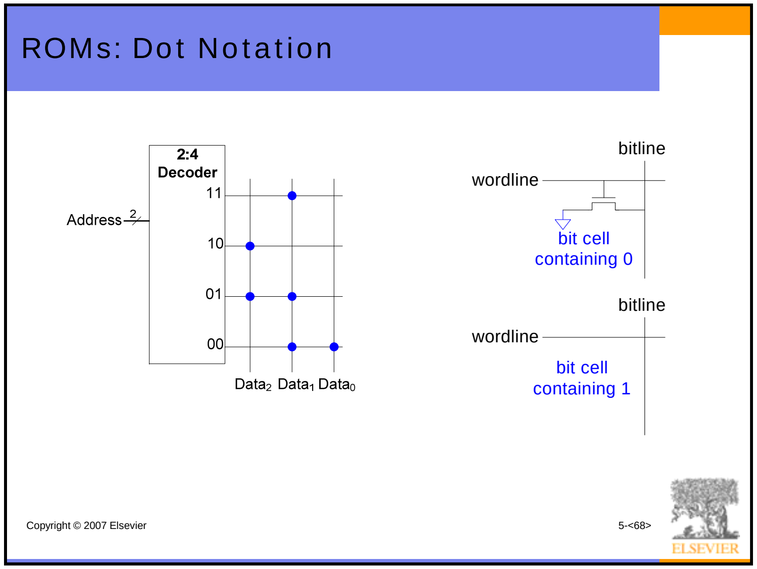
	bitline$_2$	bitline$_1$	bitline$_0$
wordline$_3$ (11)	stored bit = 0	stored bit = 1	stored bit = 0
wordline$_2$ (10)	stored bit = 1	stored bit = 0	stored bit = 0
wordline$_1$ (01)	stored bit = 1	stored bit = 1	stored bit = 0
wordline$_0$ (00)	stored bit = 0	stored bit = 1	stored bit = 1

DRAM bit cell:

SRAM bit cell:

Copyright © 2007 Elsevier

5-<67>



ROMs: Dot Notation

Copyright © 2007 Elsevier

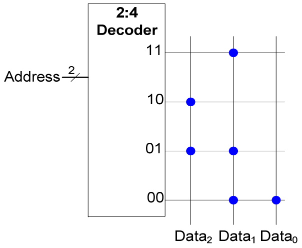
5-<68>

Fujio Masuoka, 1944-

- Developed memories and high speed circuits at Toshiba from 1971-1994.
- Invented Flash memory as an unauthorized project pursued during nights and weekends in the late 1970's.
- The process of erasing the memory reminded him of the flash of a camera
- Toshiba slow to commercialize the idea; Intel was first to market in 1988
- Flash has grown into a $25 billion per year market.

Copyright © 2007 Elsevier

ROM Storage



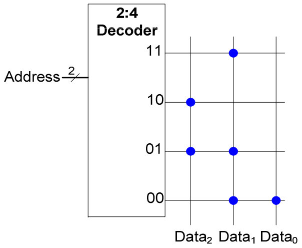
Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

depth

width

Copyright © 2007 Elsevier

ROM Logic



$$Data_2 = A_1 \oplus A_0$$

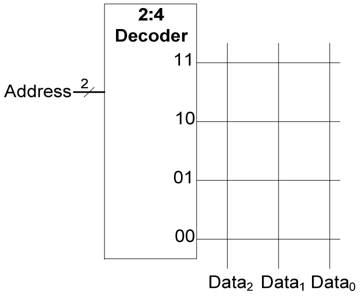
$$Data_1 = \overline{A_1} + A_0$$

$$Data_0 = \overline{A_1}\,\overline{A_0}$$

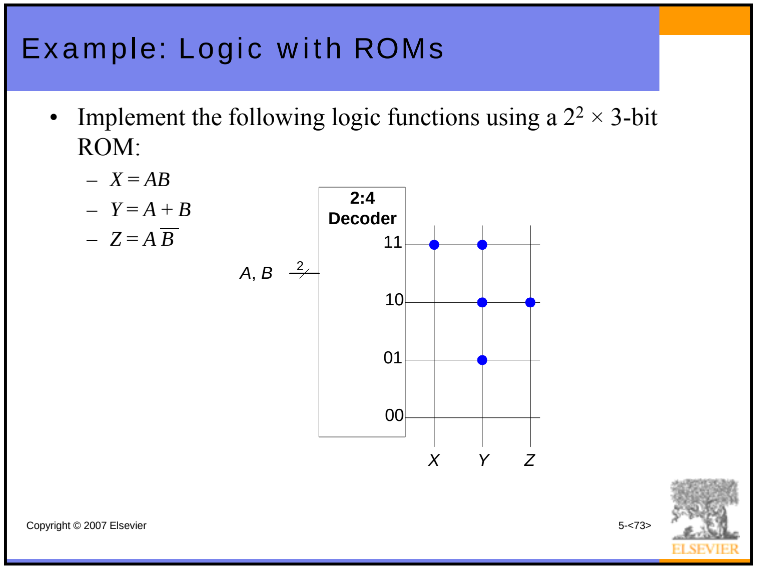
Copyright © 2007 Elsevier

Example: Logic with ROMs

- Implement the following logic functions using a $2^2 \times 3$-bit ROM:
  - $X = AB$
  - $Y = A + B$
  - $Z = A\overline{B}$



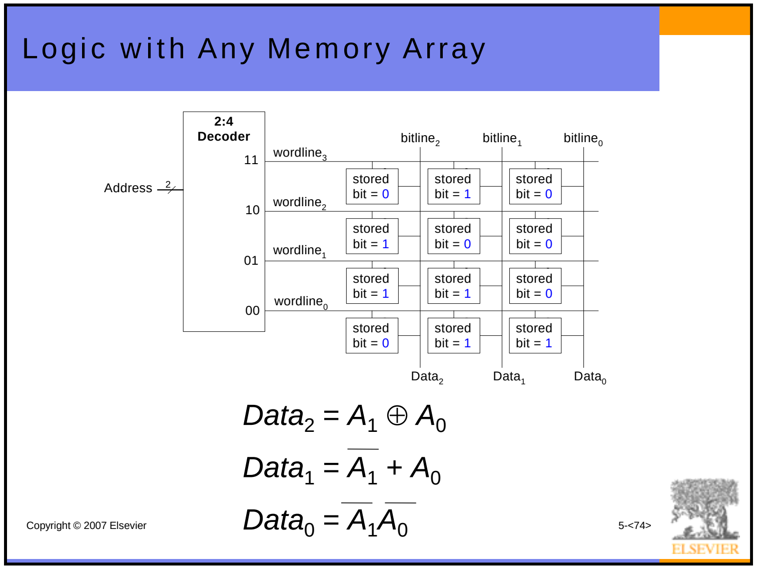
Copyright © 2007 Elsevier



Example: Logic with ROMs

- Implement the following logic functions using a $2^2 \times 3$-bit ROM:
  - $X = AB$
  - $Y = A + B$
  - $Z = A\overline{B}$

Copyright © 2007 Elsevier



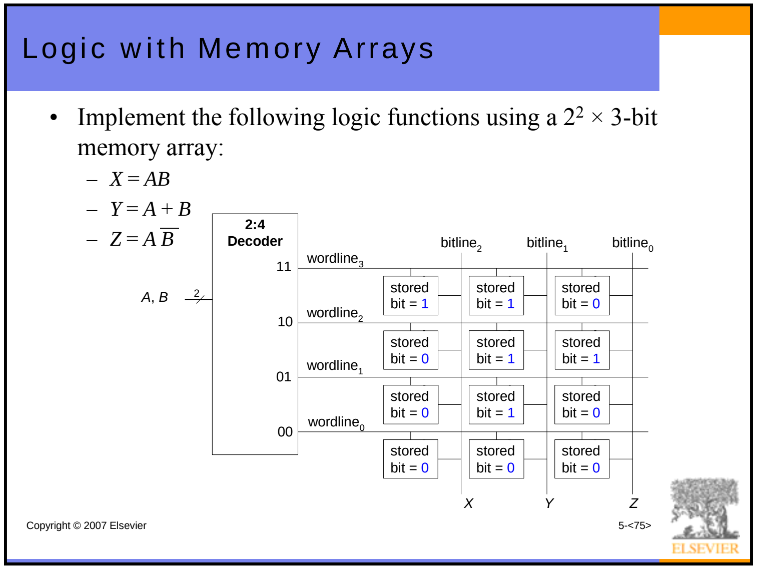
Logic with Any Memory Array

$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \overline{A_1} + A_0$$

$$Data_0 = \overline{A_1}\,\overline{A_0}$$

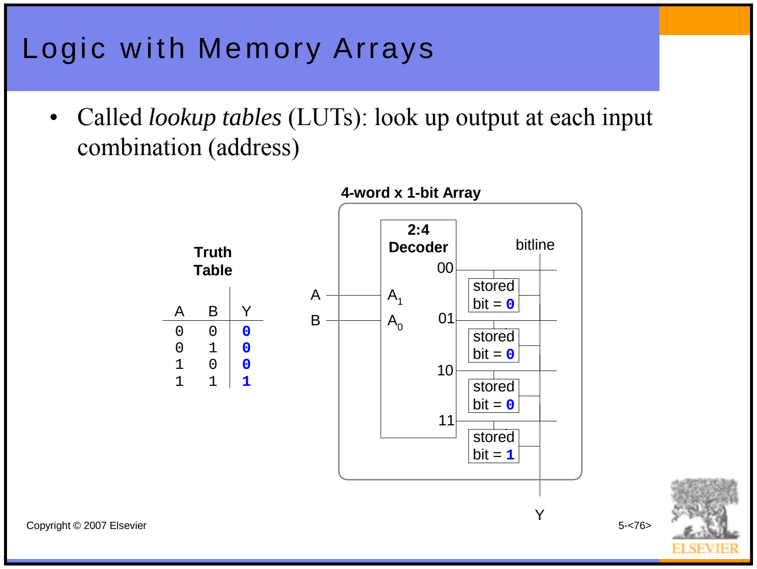
Copyright © 2007 Elsevier



Logic with Memory Arrays

- Implement the following logic functions using a $2^2 \times 3$-bit memory array:
  - $X = AB$
  - $Y = A + B$
  - $Z = A\overline{B}$

Copyright © 2007 Elsevier



Logic with Memory Arrays

- Called *lookup tables* (LUTs): look up output at each input combination (address)

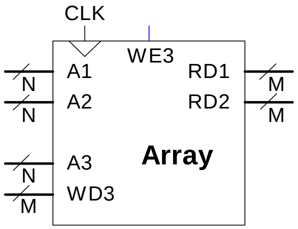
Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Copyright © 2007 Elsevier

Multi-ported Memories

- **Port:** address/data pair
- 3-ported memory
  - 2 read ports (A1/RD1, A2/RD2)
  - 1 write port (A3/WD3, WE3 enables writing)
- Small multi-ported memories are called *register files*



Copyright © 2007 Elsevier

Verilog Memory Arrays

```
// 256 x 3 memory module with one read/write port
module dmem( input  logic          clk, we,
             input  logic [7:0]    a
             input  logic [2:0]    wd,
             output logic [2:0]    rd);

  logic [2:0] RAM[255:0];

  assign rd = RAM[a];

  always_ff @(posedge clk)
      if (we)
          RAM[a] <= wd;
endmodule
```

Copyright © 2007 Elsevier

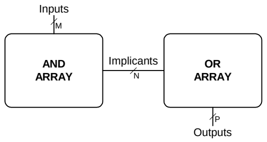
Logic Arrays

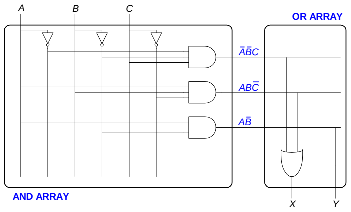
- Programmable logic arrays (PLAs)
  - AND array followed by OR array
  - Perform combinational logic only
  - Fixed internal connections
- Field programmable gate arrays (FPGAs)
  - Array of configurable logic blocks (CLBs)
  - Perform combinational and sequential logic
  - Programmable internal connections

Copyright © 2007 Elsevier

PLAs

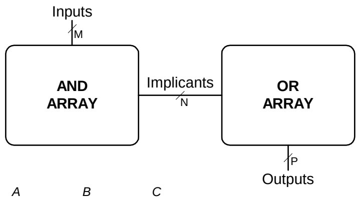
- $X = \overline{A}\overline{B}C + AB\overline{C}$
- $Y = A\overline{B}$

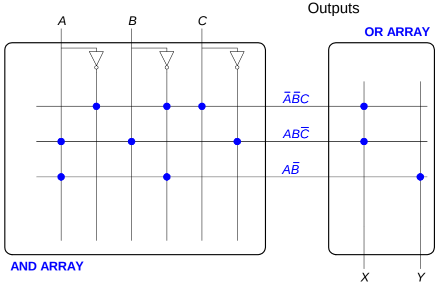




Copyright © 2007 Elsevier

PLAs: Dot Notation





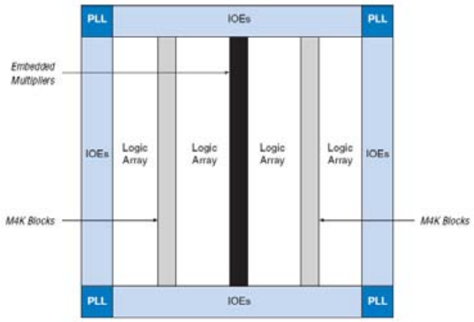
Copyright © 2007 Elsevier

FPGAs: Field Programmable Gate Arrays

- Composed of:
  - LEs (Logic Elements) or CLBs (Configurable Logic Glocks)
    - perform logic
  - IOBs (Input/output blocks)
    - interface with outside world
  - Programmable interconnection:
    - connect CLBs and IOBs
  - Some FPGAs include other building blocks such as multipliers and RAMs

Copyright © 2007 Elsevier

Altera Cyclone II FPGA Organization

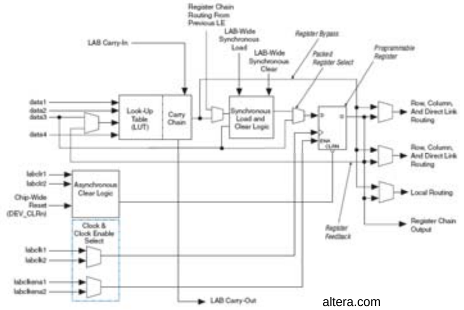


altera.com

Copyright © 2007 Elsevier

Altera Cyclone II Logic Elements

- Logic Array contains Logic Elements (LEs) with
  - LUT (lookup table): perform combinational function of 4 variables
  - Flip-flops: hold one bit of state, enable, async & sync reset
  - Multiplexers: route information to LUT and flip-flop



altera.com

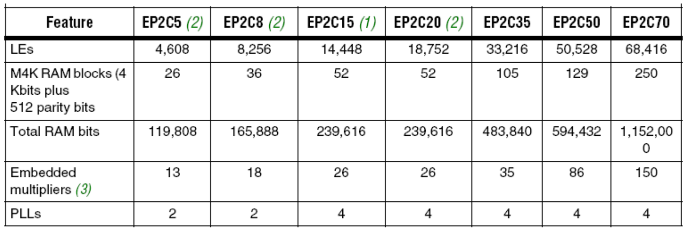
Copyright © 2007 Elsevi

Other Cyclone II Features

- 18 x 18-bit multipliers
- 4 Kb memories
- Clock generation with phase-locked loops (PLLs)
- Configurable I/O pins

Copyright © 2007 Elsevier

Cyclone II Capacities

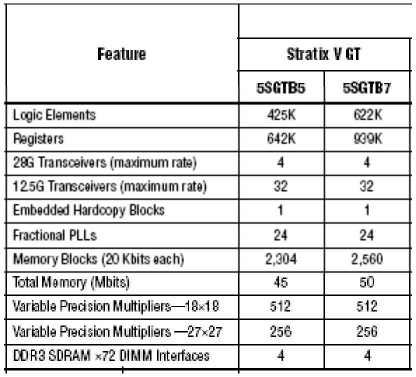


Feature	EP2C5 *(2)*	EP2C8 *(2)*	EP2C15 *(1)*	EP2C20 *(2)*	EP2C35	EP2C50	EP2C70
LEs	4,608	8,256	14,448	18,752	33,216	50,528	68,416
M4K RAM blocks (4 Kbits plus 512 parity bits	26	36	52	52	105	129	250
Total RAM bits	119,808	165,888	239,616	239,616	483,840	594,432	1,152,000
Embedded multipliers *(3)*	13	18	26	26	35	86	150
PLLs	2	2	4	4	4	4	4

altera.com

Copyright © 2007 Elsevier

Stratix V Capacities



Feature	Stratix V GT	
	5SGTB5	5SGTB7
Logic Elements	425K	622K
Registers	642K	939K
28G Transceivers (maximum rate)	4	4
12.5G Transceivers (maximum rate)	32	32
Embedded Hardcopy Blocks	1	1
Fractional PLLs	24	24
Memory Blocks (20 Kbits each)	2,304	2,560
Total Memory (Mbits)	45	50
Variable Precision Multipliers—18×18	512	512
Variable Precision Multipliers —27×27	256	256
DDR3 SDRAM ×72 DIMM Interfaces	4	4

altera.com

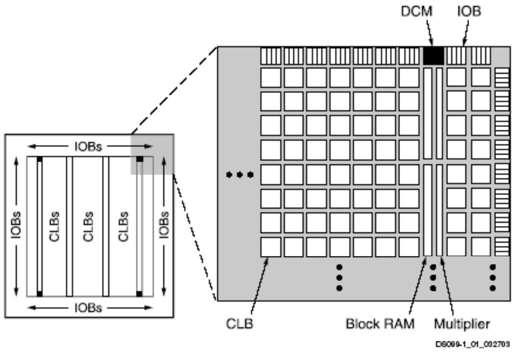
Copyright © 2007 Elsevier

FPGA Design Flow

- A CAD tool (such as Altera's Quartus or Xilinx's Project Navigator) is used to design and implement a digital system. It is usually an iterative process.
- The user **enters the design** using schematic entry or an HDL.
- The user **simulates** the design.
- A **synthesis** tool converts the code into hardware and maps it onto the FPGA.
- The user uses the CAD tool to **download the configuration** onto the FPGA
- This configures the CLBs and the connections between them and the IOBs.

Copyright © 2007 Elsevier

Xilinx Spartan 3 FPGA Schematic



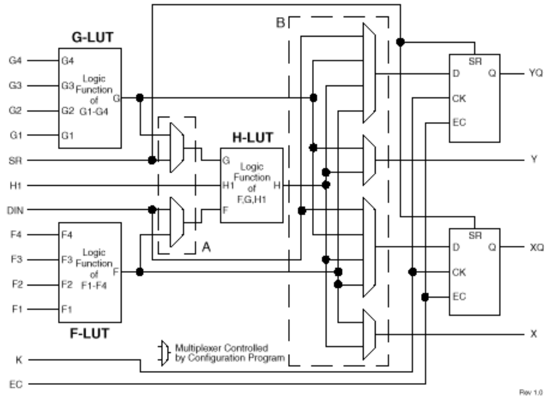
Copyright © 2007 Elsevier

CLBs: Configurable Logic Blocks

- Composed of:
  - LUTs (lookup tables): perform combinational logic
  - Flip-flops: perform sequential functions
  - Multiplexers: connect LUTs and flip-flops

Copyright © 2007 Elsevier

Xilinx Spartan CLB



Copyright © 2007 Elsevier

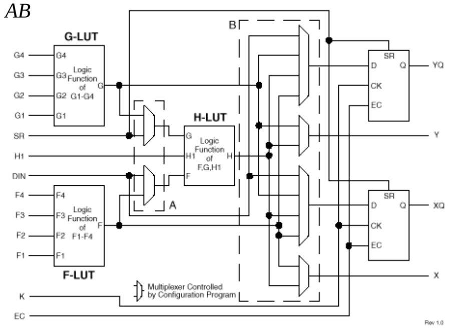
Xilinx Spartan CLB

- The Spartan CLB has:
  - 3 LUTs:
    - F-LUT (2^4 x 1-bit LUT)
    - G-LUT (2^4 x 1-bit LUT)
    - H-LUT (2^3 x 1-bit LUT)
  - 2 registered outputs:
    - *XQ*
    - *YQ*
  - 2 combinational outputs:
    - *X*
    - *Y*

Copyright © 2007 Elsevier

CLB Configuration Example

- Show how to configure the Spartan CLB to perform the following functions:
  - $X = \overline{A}\overline{B}C + AB\overline{C}$
  - $Y = A\overline{B}$



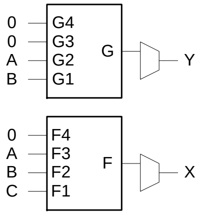
Copyright © 2007 Elsevier

CLB Configuration Example

- Show how to configure the Spartan CLB to perform the following functions:
  - $X = \overline{A}\overline{B}C + AB\overline{C}$
  - $Y = A\overline{B}$

	(A)	(B)	(C)	(X)
F4	F3	F2	F1	F
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
x	1	1	1	0

		(A)	(B)	(Y)
G4	G3	G2	G1	G
X	X	0	0	0
X	X	0	1	0
X	X	1	0	1
X	X	1	1	0



Copyright © 2007 Elsevier