Chapter 5 :: Digital Building Blocks

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris
Chapter 5 :: Topics

• Introduction
• Arithmetic Circuits
• Number Systems
• Sequential Building Blocks
• Memory Arrays
• Logic Arrays
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
1-Bit Adders

**Half Adder**

\[
\begin{array}{c|c|c}
A & B & C_{out} \quad S \\
0 & 0 & \text{---} \\
0 & 1 & \text{---} \\
1 & 0 & \text{---} \\
1 & 1 & \text{---} \\
\end{array}
\]

\[
S = \\
C_{out} = 
\]

**Full Adder**

\[
\begin{array}{c|c|c|c}
C_{in} & A & B & C_{out} \quad S \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

\[
S = \\
C_{out} = 
\]
1-Bit Adders

Half Adder

\[
\begin{array}{cccc}
A & B & C_{\text{out}} & S \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 \\
\end{array}
\]

\[S = \]
\[C_{\text{out}} = \]

Full Adder

\[
\begin{array}{cccc}
C_{\text{in}} & A & B & C_{\text{out}} & S \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[S = \]
\[C_{\text{out}} = \]
1-Bit Adders

**Half Adder**

\[ S = A \oplus B \]
\[ C_{out} = AB \]

**Full Adder**

\[ S = A \oplus B \oplus C_{in} \]
\[ C_{out} = AB + AC_{in} + BC_{in} \]
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.
Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: slow
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
• Compute carry out \( (C_{\text{out}}) \) 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:
      \[
      C_i = A_i B_i + (A_i + B_i)C_{i-1} = G_i + P_i C_{i-1}
      \]
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
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:

\[
G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))
\]

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

\[
P_{3:0} = P_3 P_2 P_1 P_0
\]

- The carry out of the $(i-j+1)$-bit block ($C_i$) is:

\[
C_i = G_{i:j} + P_{i:j} C_j
\]
32-bit CLA with 4-bit blocks

\[ t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA} \]
• 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 $N > 16$
Prefix Adder

- Computes the carry in \((C_{i-1})\) for each of the columns as fast as possible and then computes the sum:
  \[ S_i = (A_i \oplus B_i) \oplus C_{i-1} \]
- 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
  
  \[ G_{-1} = C_{in}, \quad P_{-1} = 0 \]

• 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:

  \[ G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j} \]

  \[ P_{i:j} = P_{i:k} \cdot P_{k-1:j} \]

- 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

\[ t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR} \]
Prefix Adder Delay

- The delay of an $N$-bit prefix adder is:
  \[ t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR} \]

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)
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.
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.

\[
t_{\text{ripple}} = Nt_{FA} = 32(300 \text{ ps})
\]
\[
= 9.6 \text{ ns}
\]

\[
t_{\text{CLA}} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA}
\]
\[
= [100 + 600 + (7)200 + 4(300)] \text{ ps}
\]
\[
= 3.3 \text{ ns}
\]

\[
t_{\text{PA}} = t_{pg} + \log_2N(t_{pg\_prefix}) + t_{\text{XOR}}
\]
\[
= [100 + \log_232(200) + 100] \text{ ps}
\]
\[
= 1.2 \text{ ns}
\]
Subtractor

Symbol

Implementation
Comparator: Equality

Symbol

Implementation

Equal
Comparator: Less Than

• For unsigned numbers

A < B

\[ A \lessdot B \]

\[ [N-1] \]

\[ N \]
Arithmetic Logic Unit (ALU)

<table>
<thead>
<tr>
<th>F_{2:0}</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>001</td>
<td>A</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A &amp; \neg B</td>
</tr>
<tr>
<td>101</td>
<td>A</td>
</tr>
<tr>
<td>110</td>
<td>A - B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
ALU Design

<table>
<thead>
<tr>
<th>( F_{2:0} )</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>001</td>
<td>A</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A &amp; \sim B</td>
</tr>
<tr>
<td>101</td>
<td>A</td>
</tr>
<tr>
<td>110</td>
<td>A - B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>

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 $B = 32$. 

Suppose $A = 25$ and $B = 32$. 

Configurations:

1. **Zero Extend**: If $A < B$, set $C_{out} = 0$.
2. **Set**: If $A < B$, set $F_2 = 1$.
3. **Zero**: If $A < B$, set $F_1 = 0$.
4. **Negation**: If $A < B$, set $S = N$.

Diagram:

- A and B are inputs.
- A and B are connected to an input selection circuit.
- The output of the selection circuit is connected to an adder.
- The adder output is connected to the $C_{out}$ output.
- The $F_2$ and $F_1$ outputs are generated based on the comparison between A and B.
- The $S$ output is set to $N$ if $A < B$.

**Additional Note:**

- The diagram shows a partial representation of an ALU configuration for the set less than operation. The $F_2$ and $F_1$ outputs are generated based on the comparison between A and B, with $F_2$ being 1 if $A < B$. The $S$ output is also set to $N$ if $A < B$. 

**Conclusion:**

The diagram illustrates how a 32-bit ALU can be configured for the set less than operation, demonstrating the process of selecting and comparing inputs to generate the necessary outputs.
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.
Shifters

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0’s
  - Ex: $11001 \gg 2 =$
  - Ex: $11001 \ll 2 =$

- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: $11001 \gggg 2 =$
  - Ex: $11001 \llll 2 =$

- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: $11001 \text{ ROR} 2 =$
  - Ex: $11001 \text{ ROL} 2 =$
Shifters

- **Logical shifter**: shifts value to left or right and fills empty spaces with 0’s
  - Ex: 11001 >> 2 = 00110
  - Ex: 11001 << 2 = 00100

- **Arithmetic shifter**: same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: 11001 >>> 2 = 11110
  - Ex: 11001 >>>< 2 = 00100

- **Rotator**: rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 = 01110
  - Ex: 11001 ROL 2 = 00111
Shifter Design

A \_3:0 \rightarrow 0 \quad \rightarrow 0 \quad \rightarrow 0 Y \_3:0

shamt_{1:0} \quad \downarrow \quad 2

A \_3:0 \quad \rightarrow 0 \quad \rightarrow 0 \quad \rightarrow 0 Y \_3:0
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 (\( -16 \div 2^2 = -4 \))
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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>230 \times 42</td>
<td>0101 \times 0111</td>
</tr>
<tr>
<td>460</td>
<td>0101</td>
</tr>
<tr>
<td>+ 920</td>
<td>0101</td>
</tr>
<tr>
<td>9660</td>
<td>+ 0000</td>
</tr>
<tr>
<td>result</td>
<td>0100011</td>
</tr>
</tbody>
</table>

230 \times 42 = 9660  
5 \times 7 = 35
4 x 4 Multiplier

\[ \begin{array}{ccccccc}
A_3 & A_2 & A_1 & A_0 \\
B_3 & B_2 & B_1 & B_0 \\
\times & & & & \downarrow \\
A_3B_3 & A_2B_2 & A_1B_1 & A_0B_0 \\
A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\
A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\
+ A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 \\
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array} \]
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?
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
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.
Fixed-Point Numbers

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

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

01111000
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:
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:
    1111000
  - Two’s complement:
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:
    \[
    11111000
    \]
  – Two’s complement:
    1. \(+7.5:\) \[01111000\]
    2. Invert bits: \[10000111\]
    3. Add 1 to lsb: \[+\boxed{1}0001000\]
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 $E = 2$
**Floating-Point Numbers**

- **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**

<table>
<thead>
<tr>
<th>1 bit</th>
<th>8 bits</th>
<th>23 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign</td>
<td>Exponent</td>
<td>Mantissa</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000111</td>
<td>11 1001 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

1 bit  8 bits  23 bits
Floating-Point Representation 2

• **Note:** the first bit of the mantissa is always 1:
  - \( 228_{10} = 11100100_2 = 1.11001 \times 2^7 \)

• 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.

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000111</td>
<td>110 0100 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>
Floating-Point Representation 3

- **Biased exponent**: bias = 127 (01111111₂)
  - Biased exponent = bias + exponent
  - Exponent of 7 is stored as:
    \[ 127 + 7 = 134 = 0x10000110₂ \]

- The **IEEE 754 32-bit floating-point representation** of \( 2^{28} \)₁₀

<table>
<thead>
<tr>
<th>Sign</th>
<th>Biased Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000110</td>
<td>110 0100 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

- In hexadecimal: 0x43640000
Floating-Point Example

- Write the value \(-58.25_{10}\) using the IEEE 754 32-bit floating-point standard.
Floating-Point Example

• Write the value $-58.25_{10}$ using the IEEE 754 32-bit floating-point standard.

• Convert the decimal number to binary:
  – $58.25_{10} =$

• Fill in each field in the 32-bit number:
  – Sign bit:
  – 8 Exponent bits:
  – 23 fraction bits:

<table>
<thead>
<tr>
<th>1 bit</th>
<th>8 bits</th>
<th>23 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  Sign  Exponent  Fraction

• In hexadecimal:
### 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:
  - $58.25_{10} = 111010.01_2 = 1.1101001 \times 2^5$
- Next, fill in each field in the 32-bit number:
  - Sign bit: 1 (negative)
  - 8 exponent bits: $(127 + 5) = 132 = 10000100_2$
  - 23 fraction bits: 110 1001 0000 0000 0000 0000

<table>
<thead>
<tr>
<th>1 bit</th>
<th>8 bits</th>
<th>23 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100 0010 0</td>
<td>110 1001 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

- In hexadecimal: 0xC2690000
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.

<table>
<thead>
<tr>
<th>Number</th>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>00000000</td>
<td>00000000000000000000000000000000</td>
</tr>
<tr>
<td>∞</td>
<td>0</td>
<td>11111111</td>
<td>00000000000000000000000000000000</td>
</tr>
<tr>
<td>- ∞</td>
<td>1</td>
<td>11111111</td>
<td>00000000000000000000000000000000</td>
</tr>
<tr>
<td>NaN</td>
<td>X</td>
<td>11111111</td>
<td>non-zero</td>
</tr>
</tbody>
</table>

NaN is used for numbers that don’t exist, such as √-1 or log(-5).
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
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)
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
Floating-Point Addition: Example

Add the following floating-point numbers:

0x3FC00000
0x40500000
Floating-Point Addition: Example

1. Extract exponent and fraction bits

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01111111</td>
<td>100 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

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
Floating-Point Addition: Example

3. Compare exponents
   \[127 - 128 = -1, \text{ so shift } N1 \text{ right by 1 bit}\]

4. Shift smaller mantissa if necessary
   shift \(N1\)'s mantissa: \(1.1 \gg 1 = 0.11 \times 2^1\)

5. Add mantissas
   \[
   \begin{align*}
   0.11 & \times 2^1 \\
   + 1.101 & \times 2^1 \\
   \hline
   10.011 & \times 2^1
   \end{align*}
   \]
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
   \[S = 0, \ E = 2 + 127 = 129 = 10000001_2, \ F = 001100.\ldots\]

\[
\begin{array}{c|cccc|ccc|c}
   \text{1 bit} & \text{8 bits} & \text{23 bits} \\
   \hline
   0 & 10000001 & 001 1000 0000 0000 0000 0000 \\
\end{array}
\]

Written in hexadecimal: \(0x40980000\)
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**

![Diagram of counter symbol and implementation](image)
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:**
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 $S_{out}$)
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.
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 $\times$ width $= 2^N \times M$
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:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>010</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
</tr>
<tr>
<td>00</td>
<td>011</td>
</tr>
</tbody>
</table>

Array

Address  $\rightarrow^2$

Data  $\downarrow^3$
Memory Arrays

1024-word x 32-bit Array

Address → 10

Data → 32
Memory Array Bit Cells

Example:

- For wordline = 1:
  - bitline = stored bit = 0
  - bitline = stored bit = 1

- For wordline = 0:
  - bitline = stored bit = 0
  - bitline = stored bit = 1

(a) and (b) illustrate the different scenarios for wordline and bitline conditions.
Memory Array Bit Cells

Example:

(a) wordline = 1, stored bit = 0, bitline = 0
(b) wordline = 1, stored bit = 1, bitline = 1
(c) wordline = 0, stored bit = 0, bitline = Z
(d) wordline = 0, stored bit = 1, bitline = Z

Copyright © 2007 Elsevier
• **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
Types of Memory

- Random access memory (RAM): volatile
- Read only memory (ROM): nonvolatile
**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).
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.
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
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
• Data bits stored on a capacitor
• Called *dynamic* because the value needs to be refreshed periodically and after being read:
  – Charge leakage from the capacitor degrades the value
  – Reading destroys the stored value
DRAM

Wordline

Bitline

Stored bit = 1

Stored bit = 0
SRAM

wordline

stored bit

bitline

wordline

bitline

bitline
Memory Arrays

2:4 Decoder

Address

Wordline bit = 0

Wordline bit = 1

Wordline bit = 0

Wordline bit = 0

Data

DRAM bit cell:

SRAM bit cell:
ROMs: Dot Notation

Dot represents 1 data.
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.
ROM Storage

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>010</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
</tr>
<tr>
<td>00</td>
<td>011</td>
</tr>
</tbody>
</table>

2:4 Decoder

Address: 00, 01, 10, 11

Data: 0, 1

Depth: 11, 10, 01, 00

Width: 00, 01, 10, 11
Data_2 = A_1 \oplus A_0

Data_1 = \overline{A_1} + A_0

Data_0 = \overline{A_1}A_0
Example: Logic with ROMs

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

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

$$Data_2 = A_1 \oplus A_0$$

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

$$Data_0 = \overline{A_1 A_0}$$
Logic with Memory Arrays

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

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

**Truth Table**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**4-word x 1-bit Array**

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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*
// 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 @(posedge clk)
  if (we)
    RAM[a] <= wd;
endmodule
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
PLAs

- $X = \overline{ABC} + ABC$
- $Y = A\overline{B}$
PLAs: Dot Notation

AND ARRAY

Inputs

Implicants

OR ARRAY

Outputs

AND ARRAY

OR ARRAY

\( \bar{A}B\bar{C} \)

\( A\bar{B}C \)

\( AB\bar{C} \)

\( A\bar{B} \)
FPGAs: Field Programmable Gate Arrays

• Composed of:
  – **CLBs** (Configurable logic blocks): perform logic
  – **IOBs** (Input/output buffers): interface with outside world
  – **Programmable interconnection**: connect CLBs and IOBs
  – Some FPGAs include other building blocks such as multipliers and RAMs
Xilinx Spartan 3 FPGA Schematic
CLBs: Configurable Logic Blocks

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

• The Spartan CLB has:
  – 3 LUTs:
    • F-LUT ($2^4 \times 1$-bit LUT)
    • G-LUT ($2^4 \times 1$-bit LUT)
    • H-LUT ($2^3 \times 1$-bit LUT)
  – 2 registered outputs:
    • $XQ$
    • $YQ$
  – 2 combinational outputs:
    • $X$
    • $Y$
CLB Configuration Example

- Show how to configure the Spartan CLB to perform the following functions:
  - \( X = \overline{ABC} + \overline{AB} \)
  - \( Y = A\overline{B} \)
### CLB Configuration Example

- Show how to configure the Spartan CLB to perform the following functions:
  - \( X = \overline{ABC} + \overline{ABC} \)
  - \( Y = \overline{AB} \)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(X)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F4</td>
<td>F3</td>
<td>F2</td>
<td>F1</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|   |   |   |   |   |   |   |   |   |   | (A) | (B) | (C) | (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 |   |   |   |   |   |   |   |   |   |   |   |

#### Diagram:
- **F**
  - Inputs: \( A, B, C \)
  - Outputs: \( X \)
- **G**
  - Inputs: \( A, B, C \)
  - Outputs: \( Y \)
FPGA Design Flow

• A CAD tool (such as Xilinx 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.