Chapter 5 :: Topics

• Introduction
• Arithmetic Circuits
• Number Systems
• Sequential Building Blocks
• Memory Arrays
• Logic Arrays
• 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 extends to different sizes

• Will use these building blocks in Chapter 7 to build microprocessor
1-Bit Adders

### Half Adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>(C_{\text{out}})</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Equations:**

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

### Full Adder

<table>
<thead>
<tr>
<th>(C_{\text{in}})</th>
<th>A</th>
<th>B</th>
<th>(C_{\text{out}})</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Equations:**

\[ S = A \oplus B \oplus C_{\text{in}} \]
\[ C_{\text{out}} = A \cdot B + A \cdot C_{\text{in}} + B \cdot C_{\text{in}} \]
# 1-Bit Adders

## Half Adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C\text{_out}</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ S = C\text{\_out} = \]

## Full Adder

<table>
<thead>
<tr>
<th>C\text{_in}</th>
<th>A</th>
<th>B</th>
<th>C\text{_out}</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ S = C\text{\_out} = \]
1-Bit Adders

**Half Adder**

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C_{out}</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Full Adder**

\[ S = A \oplus B \oplus C_{in} \]
\[ C_{out} = AB + AC_{in} + BC_{in} \]

<table>
<thead>
<tr>
<th>C_{in}</th>
<th>A</th>
<th>B</th>
<th>C_{out}</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Multibit Adders (CPAs)

- Types of carry propagate adders (CPAs):
  - Ripple-carry (slow)
  - Carry-lookahead (fast)
  - Prefix (faster)

- Carry-lookahead and prefix adders faster for large adders but require more hardware.
Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: slow
Ripple-Carry Adder Delay

\[ t_{\text{ripple}} = N t_{FA} \]

where \( t_{FA} \) is the delay of a 1-bit full adder
• Compute carry out \( (C_{\text{out}}) \) for \( k \)-bit blocks using generate and propagate signals

• Some definitions:
  – Column \( i \) 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:
    • Column \( i \) will generate a carry out if \( A_i \) AND \( B_i \) are both 1.
    \[
    G_i = A_i B_i
    \]
    • Column \( i \) 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 column \( i \) \( (C_i) \) 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 $G_i$ and $P_i$ for all columns
- **Step 2:** Compute $G$ and $P$ for $k$-bit blocks
- **Step 3:** $C_{in}$ propagates through each $k$-bit propagate/generate block
• Example: 4-bit blocks ($G_{3:0}$ and $P_{3:0}$):

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

$$P_{3:0} = P_3 P_2 P_1 P_0$$

• Generally,

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j))$$

$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$

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

Carry-Lookahead Adder
32-bit CLA with 4-bit Blocks

B_{31:28} A_{31:28} \rightarrow C_{27} 4-bit CLA Block \rightarrow C_{23} 4-bit CLA Block \rightarrow \ldots C_7 4-bit CLA Block \rightarrow C_3 4-bit CLA Block \rightarrow C_{in}

C_{out} S_{31:28} \quad S_{27:24} \quad S_{7:4} \quad S_{3:0}

32-bit CLA with 4-bit Blocks
For $N$-bit CLA with $k$-bit blocks:

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

- $t_{pg}$: delay to generate all $P_i, G_i$
- $t_{pg\_block}$: delay to generate all $P_{i:j}, G_{i:j}$
- $t_{AND\_OR}$: delay from $C_{in}$ to $C_{out}$ of final AND/OR gate in $k$-bit CLA block

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

- Computes carry in \((C_{i-1})\) for each column, then computes sum:

\[ S_i = (A_i \oplus B_i) \oplus C_i \]

- Computes \(G\) and \(P\) for 1-, 2-, 4-, 8-bit blocks, etc. until all \(G_i\) (carry in) known

- \(\log_2 N\) stages
Prefix Adder

- Carry in either *generated* in a column or *propagated* from a previous column.
- Column -1 holds $C_{in}$, so
  
  \[ G_{-1} = C_{in}, \quad P_{-1} = 0 \]

- Carry in to column $i = \text{carry out of column } i-1$:
  
  \[ C_{i-1} = G_{i-1: -1} \]

  $G_{i-1: -1}$: generate signal spanning columns $i-1$ to -1

- Sum equation:
  
  \[ S_i = (A_i \oplus B_i) \oplus G_{i-1: -1} \]

- **Goal:** Quickly compute $G_{0: -1}$, $G_{1: -1}$, $G_{2: -1}$, $G_{3: -1}$, $G_{4: -1}$, $G_{5: -1}$, … (called *prefixes*)
Prefix Adder

• Generate and propagate signals for a block spanning bits $i:j$:

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

• In words:
  – **Generate**: block $i:j$ will generate a carry if:
    • upper part ($i:k$) generates a carry or
    • upper part propagates a carry generated in lower part ($k-1:j$)
  – **Propagate**: block $i:j$ will propagate a carry if *both* the upper and lower parts propagate the carry
Prefix Adder Schematic

Legend

\[ A_i, B_i \]
\[ P_{ij}, G_{ij} \]
\[ P_{ik}, P_{k-1j}, G_{ik}, G_{k-1j} \]
\[ G_{j-1,i-1}, A_i, B_i \]
Prefix Adder Delay

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

- \( t_{pg} \): delay to produce \( P_i G_i \) (AND or OR gate)
- \( t_{pg\_prefix} \): delay of black prefix cell (AND-OR gate)
Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps
Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps

\[
\begin{align*}
    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}
\end{align*}
\]
Subtractor

Symbol

Implementation

\[ A - B = Y \]
Comparator: Equality

Symbol

\[ A \quad B \]

= 

Equal

Implementation

\[ \begin{align*}
A_3 & \quad B_3 \\
A_2 & \quad B_2 \\
A_1 & \quad B_1 \\
A_0 & \quad B_0 \\
\end{align*} \]

Equal
Comparator: Less Than

A < B
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; ~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

\[ F_{2:0} \times \begin{array}{c|c} \text{Function} \\ \hline 000 & A \& B \\ 001 & A \mid B \\ 010 & A + B \\ 011 & \text{not used} \\ 100 & A \& \neg B \\ 101 & A \mid \neg B \\ 110 & A - B \\ 111 & \text{SLT} \end{array} \]
Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps

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

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

\[
t_{\text{PA}} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{\text{XOR}}
\]
\[
= [100 + \log_2 32(200) + 100] \, \text{ps} = 1.2 \, \text{ns}
\]
Set Less Than (SLT) Example

- Configure 32-bit ALU for SLT operation: $A = 25$ and $B = 32$
Set Less Than (SLT) Example

- Configure 32-bit ALU for SLT operation: \( A = 25 \) and \( B = 32 \)
  - \( A < B \), so \( Y \) should be 32-bit representation of 1 (0x00000001)
  - \( F_{2:0} = 111 \)
    - \( F_2 = 1 \) (adder acts as subtracter), so \( 25 - 32 = -7 \)
    - \(-7\) has 1 in the most significant bit \((S_{31} = 1)\)
    - \( F_{1:0} = 11 \) 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:**
  - Ex: $11001 \gg 2 = 00110$
  - Ex: $11001 \ll 2 = 00100$

- **Arithmetic shifter:**
  - Ex: $11001 \gg> 2 = 11110$
  - Ex: $11001 \ll< 2 = 00100$

- **Rotator:**
  - Ex: $11001 \text{ ROR } 2 = 01110$
  - Ex: $11001 \text{ ROL } 2 = 00111$
Shifter Design

\[ A_{3:0} \rightarrow 4 \quad >> \quad 4 \rightarrow Y_{3:0} \]
• \( A << N = A \times 2^N \)
  - Example: \( 00001 << 2 = 00100 \) \((1 \times 2^2 = 4)\)
  - Example: \( 11101 << 2 = 10100 \) \((-3 \times 2^2 = -12)\)

• \( A >>> N = A \div 2^N \)
  - Example: \( 01000 >>> 2 = 00010 \) \((8 \div 2^2 = 2)\)
  - Example: \( 10000 >>> 2 = 11100 \) \((-16 \div 2^2 = -4)\)
• **Partial products** formed by multiplying a single digit of the multiplier with multiplicand

• **Shifted** partial products **summed** to form result

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>0101</td>
</tr>
<tr>
<td>x 42</td>
<td>x 0111</td>
</tr>
<tr>
<td>460</td>
<td>0101</td>
</tr>
<tr>
<td>+ 920</td>
<td>0101</td>
</tr>
<tr>
<td>9660</td>
<td>0101</td>
</tr>
</tbody>
</table>

- **230 x 42 = 9660**

- **5 x 7 = 35**
4 x 4 Multiplier

\[
\begin{array}{c}
x \\
\hline
A_3 & A_2 & A_1 & A_0 \\
\hline
B_3 & B_2 & B_1 & B_0 \\
\hline
A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\
A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
\]

Diagram showing the 4 x 4 multiplier circuit with inputs A, B and outputs P.
A/B = Q + R/B

**Algorithm:**

\[ R' = 0 \]

for \( i = N-1 \) to 0

\[ R = \{ R' << 1, A_i \} \]

\[ D = R - B \]

if \( D < 0 \), \( Q_i = 0 \), \( R' = R \)

else \( Q_i = 1 \), \( R' = D \)

\( R' = R \)
Number Systems

- Numbers we can represent using binary representations
  - Positive numbers
    - Unsigned binary
  - Negative numbers
    - Two’s complement
    - Sign/magnitude numbers

- What about fractions?
• Two common notations:
  – **Fixed-point**: binary point fixed
  – **Floating-point**: binary point floats to the right of the most significant 1
Fixed-Point Numbers

- 6.75 using 4 integer bits and 4 fraction bits:

\[ \text{01101100} \]
\[ \text{0110.1100} \]
\[ 2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75 \]

- Binary point is implied
- The number of integer and fraction bits must be agreed upon beforehand
• Represent $7.5_{10}$ using 4 integer bits and 4 fraction bits.
• Represent $7.5_{10}$ using 4 integer bits and 4 fraction bits.

01111000
Signed Fixed-Point Numbers

• **Representations:**
  – Sign/magnitude
  – Two’s complement

• **Example:** Represent -7.5\textsubscript{10} using 4 integer and 4 fraction bits
  – **Sign/magnitude:**
  – **Two’s complement:**
Signed Fixed-Point Numbers

• **Representations:**
  – Sign/magnitude
  – Two’s complement

• **Example:** Represent $-7.5_{10}$ using 4 integer and 4 fraction bits
  – **Sign/magnitude:**
    
    \[
    11111000
    \]
  – **Two’s complement:**
    
    1. $+7.5$: $01111000$
    2. Invert bits: $10000111$
    3. Add 1 to lsb: $+1$
    \[
    10001000
    \]
Floating-Point Numbers

- 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:
  \[ \pm M \times B^E \]
  - $M$ = mantissa
  - $B$ = base
  - $E$ = exponent
  - In the example, $M = 2.73$, $B = 10$, and $E = 2$
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 – final version is called the **IEEE 754 floating-point standard**
1. Convert decimal to binary (**don’t reverse steps 1 & 2!**):

\[ 228_{10} = 11100100_2 \]

2. Write the number in “binary scientific notation”:

\[ 11100100_2 = 1.11001_2 \times 2^7 \]

3. Fill in each field of the 32-bit floating point 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>1 bit</th>
<th>8 bits</th>
<th>23 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000111</td>
<td>11 1001 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

**Sign**  **Exponent**  **Mantissa**
Floating-Point Representation 2

- First bit of the mantissa is always 1:
  - \(228_{10} = 11100100_2 = 1.11001 \times 2^7\)
- So, no need to store it: *implicit leading 1*
- Store just fraction bits in 23-bit field

<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>
• **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\text{_{10}}

<table>
<thead>
<tr>
<th>1 bit</th>
<th>8 bits</th>
<th>23 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000110</td>
<td>110 0100 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

**Sign**  **Biased Exponent**

in hexadecimal: 0x43640000
Write \(-58.25_{10}\) in floating point (IEEE 754)
Write \(-58.25_{10}\) in floating point (IEEE 754)

1. Convert decimal to binary:

\[ 58.25_{10} = \text{111010.01}_2 \]

2. Write in binary scientific notation:

\[ 1.1101001 \times 2^5 \]

3. Fill in fields:
   - **Sign bit**: 1 (negative)
   - **8 exponent bits**: \((127 + 5) = 132 = \text{10000100}_2\\)
   - **23 fraction bits**: \text{110 1001 0000 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: Special Cases

<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>00000000000000000000000</td>
</tr>
<tr>
<td>∞</td>
<td>0</td>
<td>11111111</td>
<td>00000000000000000000000</td>
</tr>
<tr>
<td>- ∞</td>
<td>1</td>
<td>11111111</td>
<td>00000000000000000000000</td>
</tr>
<tr>
<td>NaN</td>
<td>X</td>
<td>11111111</td>
<td>non-zero</td>
</tr>
</tbody>
</table>
Floating-Point Precision

• **Single-Precision:**
  – 32-bit
  – 1 sign bit, 8 exponent bits, 23 fraction bits
  – bias = 127

• **Double-Precision:**
  – 64-bit
  – 1 sign bit, 11 exponent bits, 52 fraction bits
  – bias = 1023
Overflow: number too large to be represented
Underflow: number too small to be represented
Rounding modes:
  - Down
  - Up
  - Toward zero
  - To nearest

Example: round 1.100101 (1.578125) to 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
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>
<tr>
<td>0</td>
<td>10000000</td>
<td>101 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
3. Compare exponents
   
   \[ 127 - 128 = -1 \], so shift N1 right by 1 bit

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

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

<table>
<thead>
<tr>
<th>1 bit</th>
<th>8 bits</th>
<th>23 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000001</td>
<td>001 1000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

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**

```
CLK

Q
Reset

CLK

1

N

N

N

N

Q

Reset
```
Shift Registers

- Shift a new bit in on each clock edge
- Shift a bit out on each clock edge
- *Serial-to-parallel converter*: converts serial input \( (S_{\text{in}}) \) 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
- 3 common types:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)
- $M$-bit data value read/ written at each unique $N$-bit address
Memory Arrays

- 2-dimensional array of bit cells
- Each bit cell stores one bit
- $N$ 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
Memory Arrays

1024-word x 32-bit Array

Address \(\rightarrow 10\)

Data \(\leftarrow 32\)
Memory Array Bit Cells

![Diagram showing memory array bit cells with wordline and bitline connections and stored bit values for different conditions.](diagram.png)

- For wordline = 1:
  - Stored bit = 0 (a)
  - Stored bit = 1 (b)

- For wordline = 0:
  - Stored bit = 0
  - Stored bit = 1

Where:
- Memory Array Bit Cells
- Wordline
- Bitline
- Stored Bit

Note: The diagram illustrates the relationship between the wordline, bitline, and stored bit values in a memory array.
Memory Array Bit Cells

![Diagram of Memory Array Bit Cells]

- **Memory Array Bit Cells**
  - **Wordline**
  - **Bitline**
  - **Stored Bit**

- **Wordline Values**:
  - **wordline = 1**
    - **bitline**
      - **0**: Stored bit = 0
      - **1**: Stored bit = 1
  - **wordline = 0**
    - **bitline**
      - **0**: Stored bit = 0
      - **1**: Stored bit = 1

- **Bitline Values**:
  - **bitline = 0**
  - **bitline = 1**
  - **bitline = Z**

- **Example Cases**:
  - (a) Wordline 1, Bitline 0: Stored bit = 0
  - (b) Wordline 0, Bitline 1: Stored bit = 1


Chapter 5 <67>
- **Wordline:**
  - like an enable
  - single row in memory array read/written
  - corresponds to unique address
  - only one wordline HIGH at once
Types of Memory

- Random access memory (RAM): volatile
- Read only memory (ROM): nonvolatile
• **Volatile**: loses its data when power off
• Read and written quickly
• Main memory in your computer is RAM (DRAM)

Historically called *random* access memory because any data word 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 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

- **DRAM** (Dynamic random access memory)
- **SRAM** (Static random access memory)
- 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 in virtually all computers
• Data bits stored on capacitor
• *Dynamic* because the value needs to be refreshed (rewritten) periodically and after read:
  – Charge leakage from the capacitor degrades the value
  – Reading destroys the stored value
DRAM

Wordline

Bitline

Stored bit = 1

Stored bit = 0
SRAM

![Diagram of SRAM]
Memory Arrays Review

2:4 Decoder

Address 2

wordline_3

stored bit = 0

wordline_2

stored bit = 1

wordline_1

stored bit = 1

wordline_0

stored bit = 0

Data_2

Data_1

Data_0

DRAM bit cell:

SRAM bit cell:
ROM: Dot Notation

2:4 Decoder

Address

Data₂ Data₁ Data₀

wordline

bitline

bit cell containing 0

bit cell containing 1

• Developed memories and high speed circuits at Toshiba, 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

2:4 Decoder

Address 11 10 01 00
Data  010 100 110 011

Address 2

ROM Storage
\[ Data_2 = A_1 \oplus A_0 \]
\[ Data_1 = \overline{A_1} + A_0 \]
\[ Data_0 = \overline{A_1}A_0 \]
Implement the following logic functions using a $2^2 \times 3$-bit ROM:

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

\[
\begin{align*}
\text{Data}_2 &= A_1 \oplus A_0 \\
\text{Data}_1 &= \overline{A}_1 + A_0 \\
\text{Data}_0 &= \overline{A}_1 \overline{A}_0
\end{align*}
\]
Implement the following logic functions using a $2^2 \times 3$-bit memory array:

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

- $X = AB$
- $Y = A + B$
- $Z = A \overline{B}$
Called *lookup tables* (LUTs): look up output at each input combination (address)
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)
- **Register file**: small multi-ported memory
// 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

• **PLAs (Programmable logic arrays)**
  – AND array followed by OR array
  – Combinational logic only
  – Fixed internal connections

• **FPGAs (Field programmable gate arrays)**
  – Array of Logic Elements (LEs)
  – Combinational and sequential logic
  – Programmable internal connections
PLAs

- \( X = \overline{A}BC + ABC \)
- \( Y = A\overline{B} \)

\[ \begin{align*}
\text{Inputs} & \quad \text{Implicants} & \quad \text{Outputs} \\
M & \quad N & \quad P \\
\end{align*} \]
PLAs: Dot Notation

Inputs

AND ARRAY

Implicants

OR ARRAY

Outputs

AND ARRAY

OR ARRAY

\[ \bar{A}\bar{B}C \]

\[ A\bar{B}C \]

\[ AB\bar{C} \]

\[ AB \]

\[ X \]

\[ Y \]
• Composed of:
  – **LEs** (Logic elements): perform logic
  – **IOEs** (Input/output elements): interface with outside world
  – **Programmable interconnection**: connect LEs and IOEs
  – Some FPGAs include other building blocks such as multipliers and RAMs
General FPGA Layout
• Composed of:
  – **LUTs** (lookup tables): perform combinational logic
  – **Flip-flops**: perform sequential logic
  – **Multiplexers**: connect LUTs and flip-flops
The Altera Cyclone IV LE has:

- 1 four-input LUT
- 1 registered output
- 1 combinational output
Show how to configure a Cyclone IV LE to perform the following functions:

- $X = \overline{ABC} + \overline{AB\bar{C}}$
- $Y = \overline{AB}$
Show how to configure a Cyclone IV LE to perform the following functions:

- \( X = \overline{A} \overline{B} \overline{C} + \overline{A} \overline{B} \overline{C} \)
- \( Y = \overline{A} \overline{B} \)

**LE Configuration Example**

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>data 4</th>
<th>(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>data 4</th>
<th>(Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>
FPGA Design Flow

Using a CAD tool (such as Altera’s Quartus II)

- **Enter the design** using schematic entry or an HDL
- **Simulate** the design
- **Synthesize** design and map it onto FPGA
- **Download the configuration** onto the FPGA
- **Test** the design