CSE 140 Lecture 13
Standard Combinational Modules

CK Cheng
CSE Dept.
UC San Diego

Some slides from Harris and Harris
Part III. Standard Modules

Interconnect Operators.

Representation of numbers
Adders
1. Full Adder
2. Half Adder
3. Ripple-Carry Adder
4. Carry Look Ahead Adder
5. Prefix Adder

ALU
Comparator
Shifter
Multiplier
Division
Design Flow

• Specification: Data Representations
• Arithmetic: Algorithms
• Logic: Synthesis
• Layout: Placement and Routing
1. Representation of numbers

Negative Numbers

• 2’s complement of n-bit vector
  \(-x: 2^n - x\)

• 1’s complement of n-bit vector
  \(-x: 2^n - x - 1\)
1. Representation
   • 2’s Complement
     \[-x: 2^n - x\]
     e.g. 16-x
   • 1’s Complement
     \[-x: 2^n - x - 1\]
     e.g. 16-x-1

<table>
<thead>
<tr>
<th>Id</th>
<th>2’s comp.</th>
<th>1’s comp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>-1</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>-2</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>-3</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>-4</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>-5</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>-6</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>-7</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>-8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
## 1. Representation

<table>
<thead>
<tr>
<th>Id</th>
<th>-Binary</th>
<th>sign mag</th>
<th>2’s comp</th>
<th>1’s comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>1000</td>
<td>0000</td>
<td>1111</td>
</tr>
<tr>
<td>-1</td>
<td>0001</td>
<td>1001</td>
<td>1111</td>
<td>1110</td>
</tr>
<tr>
<td>-2</td>
<td>0010</td>
<td>1010</td>
<td>1110</td>
<td>1101</td>
</tr>
<tr>
<td>-3</td>
<td>0011</td>
<td>1011</td>
<td>1101</td>
<td>1100</td>
</tr>
<tr>
<td>-4</td>
<td>0100</td>
<td>1100</td>
<td>1100</td>
<td>1011</td>
</tr>
<tr>
<td>-5</td>
<td>0101</td>
<td>1101</td>
<td>1011</td>
<td>1010</td>
</tr>
<tr>
<td>-6</td>
<td>0110</td>
<td>1110</td>
<td>1010</td>
<td>1001</td>
</tr>
<tr>
<td>-7</td>
<td>0111</td>
<td>1111</td>
<td>1001</td>
<td>1000</td>
</tr>
<tr>
<td>-8</td>
<td></td>
<td></td>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>
Representation

1’s Complement

For a negative number, we take the positive number and complement every bit.

2’s Complement

For a negative number, we do 1’s complement and plus one.

\[(b_{n-1}, b_{n-2}, \ldots, b_0): -b_{n-1}2^{n-1} + \sum_{i<n-1} b_i2^i\]

\[b_{n-1}=1 \text{ iff the number is negative}\]
Representation

2’s Complement

- \( x+y \)
- \( x-y: x+2^n-y= 2^n+x-y \)
- \( -x+y: 2^n-x+y \)
- \( -x-y: 2^n-x+2^n-y = 2^n+2^n-x-y \)
- \( -(x)=2^n-(2^n-x)=x \)

1’s Complement

- \( x+y \)
- \( x-y: x+2^n-y-1= 2^n-1+x-y \)
- \( -x+y: 2^n-x-1+y=2^n-1-x+y \)
- \( -x-y: 2^n-x-1+2^n-y-1 = 2^n-1+2^n-x-y-1 \)
- \( -(x)=2^n-(2^n-x-1) -1=x_8 \)
$2 + 3 = 5$
\[
\begin{array}{cccc}
0 & 0 & 1 & 0 \\
+ & 0 & 0 & 1 1 \\
\hline
0 & 1 & 0 & 1 \\
\end{array}
\]

$2 - 3 = -1$ (2’s)
\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
+ & 1 & 1 & 0 1 \\
\hline
1 & 1 & 1 & 1 \\
\end{array}
\]

Examples

$2 - 3 = -1$ (1’s)
\[
\begin{array}{cccc}
0 & 0 & 1 & 0 \\
+ & 1 & 1 & 0 0 \\
\hline
1 & 1 & 0 & 0 \\
\end{array}
\]

Check for overflow (2’s)

$-2 - 3 = -5$ (2’s)
\[
\begin{array}{cccc}
1 & 1 & 0 & 0 \\
+ & 1 & 1 & 0 1 \\
\hline
1 & 0 & 1 & 1 \\
\end{array}
\]

$-2 - 3 = -5$ (1’s)
\[
\begin{array}{cccc}
1 & 1 & 0 & 0 \\
+ & 1 & 1 & 0 0 \\
\hline
1 & 0 & 0 & 1 \\
\end{array}
\]

$3 + 5 = 8$
\[
\begin{array}{cccc}
0 & 1 & 1 & 1 \\
+ & 0 & 1 & 0 1 \\
\hline
1 & 0 & 0 & 0 \\
\end{array}
\]

$-3 + -5 = -8$
\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
+ & 1 & 0 & 1 1 \\
\hline
1 & 0 & 0 & 0 \\
\end{array}
\]

$C_4C_3$
Addition: 2’s Complement Overflow

In 2’s complement:
\[
\text{overflow} = c_n \oplus c_{n-1}
\]

Exercise:
1. Demonstrate the overflow with more examples.
2. Prove the condition.
Addition and Subtraction using 2’s Complement

Adder

\[ C_4 \rightarrow \text{overflow} \]

\[ C_3 \]

\[ a \]
\[ b \]
\[ b' \]

\[ D_0 \]
\[ D_1 \]
\[ s_0 \]

\[ C_{\text{in}} \]

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

\[ \text{Minus} \]

\[ \text{Sum} \]
1-Bit Adders

**Half Adder**

<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>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ S = A \oplus B \]

\[ C_{out} = AB \]

**Full Adder**

<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>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>

\[ S = A \oplus B \oplus C_{in} \]

\[ C_{out} = AB + AC_{in} + BC_{in} \]
Half Adder

\[ S_{um} = ab' + a'b = a \oplus b \]
\[ C_{out} = ab \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>C_{out}</th>
<th>S_{um}</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>
Full Adder Composed of Half Adders
Full Adder Composed of Half Adders

\[\begin{array}{c}
a \\
b \\
c_{\text{in}}
\end{array}\xrightarrow{\text{HA}}
\begin{array}{c}
x \\
y \\
z
\end{array}\xrightarrow{\text{HA}}
\begin{array}{c}
s_{\text{out}} \\
c_{\text{out}}
\end{array}\xrightarrow{\text{HA}}
\begin{array}{c}
x \\
y \\
z
\end{array}\xrightarrow{\text{HA}}
\begin{array}{c}
\text{sum}
\end{array}\]

<table>
<thead>
<tr>
<th>Id</th>
<th>a</th>
<th>b</th>
<th>c_{\text{in}}</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>c_{\text{out}}</th>
<th>s_{\text{sum}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>x</th>
<th>z</th>
<th>c_{\text{out}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>
Adder

• 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
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
Carry-Lookahead Adder

• Compress the logic levels of $C_{\text{out}}$

• Some definitions:
  
  – 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:
      
      \[ C_{i+1} = A_i B_i + (A_i + B_i)C_i = G_i + P_i C_i \]
Carry Look Ahead Adder

\[ C_1 = a_0 b_0 + (a_0 + b_0) c_0 = g_0 + p_0 c_0 \]
\[ C_2 = a_1 b_1 + (a_1 + b_1) c_1 = g_1 + p_1 c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0 \]
\[ C_3 = a_2 b_2 + (a_2 + b_2) c_2 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \]
\[ C_4 = a_3 b_3 + (a_3 + b_3) c_3 = g_3 + p_3 c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 \]

\[ q_i = a_i b_i \quad p_i = a_i + b_i \]
Carry-Lookahead Addition

- Step 1: compute \textit{generate} \((G)\) and \textit{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
32-bit CLA with 4-bit blocks

\[ B_{31:28} A_{31:28} \]
\[ B_{27:24} A_{27:24} \]
\[ B_{24} \]
\[ B_{21} A_{21} \]
\[ B_{18} A_{18} \]
\[ B_{15} A_{15} \]
\[ B_{12} A_{12} \]
\[ B_{9} A_{9} \]
\[ B_{6} A_{6} \]
\[ B_{3} A_{3} \]
\[ B_{0} A_{0} \]

\[ S_{31:28} \]
\[ S_{27:24} \]
\[ S_{24} \]
\[ S_{21} \]
\[ S_{18} \]
\[ S_{15} \]
\[ S_{12} \]
\[ S_{9} \]
\[ S_{6} \]
\[ S_{3} \]

G_{3:0}

C_{out}

G_{3}
P_{3}
G_{2}
P_{2}
G_{1}
P_{1}
G_{0}

C_{in}

P_{3:0}

C_{in}
Carry-Lookahead Adder Delay

- Delay of an $N$-bit carry-lookahead adder with $k$-bit blocks:
  \[ t_{CLA} = t_{pg} + t_{pg\_block} + \left(\frac{N}{k} - 1\right)t_{AND\_OR} + kt_{FA} \]

  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:

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

• 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 \( G_{-1} = C_{in}, P_{-1} = 0 \)
- Then, the carry in to col. \( i \) = the carry out of col. \( i-1 \):
  \[ C_{i-1} = G_{i-1:-1} \]
  \( G_{i-1:-1} \) 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 rewrite the sum equation:
  \[ S_i = (A_i \oplus B_i) \oplus G_{i-1:-1} \]
- **Goal**: Compute \( G_{0:-1}, G_{1:-1}, G_{2:-1}, G_{3:-1}, G_{4:-1}, G_{5:-1}, \ldots \) (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 Delay

• The delay of an $N$-bit prefix adder is:

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\text{-prefix}}) + t_{\text{XOR}}$$

where

- $t_{pg}$ is the delay of the column generate and propagate gates (AND or OR gate)
- $t_{pg\text{-prefix}}$ 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.

\[
\begin{align*}
    t_{\text{ripple}} &= N t_{\text{FA}} = 32(300 \ \text{ps}) = 9.6 \ \text{ns} \\
    t_{\text{CLA}} &= t_{\text{pg}} + t_{\text{pg_block}} + (N/k - 1)t_{\text{AND_OR}} + k t_{\text{FA}} \\
                   &= [100 + 600 + (7)200 + 4(300)] \ \text{ps} = 3.3 \ \text{ns} \\
    t_{\text{PA}} &= t_{\text{pg}} + \log_2 N(t_{\text{pg_prefix}}) + t_{\text{XOR}} \\
                   &= [100 + \log_2 32(200) + 100] \ \text{ps} = 1.2 \ \text{ns}
\end{align*}
\]
Comparator: Equality

Symbol

Implementation

Equal
Comparator: Less Than

- Compare two numbers

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; \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>
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; ~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>
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 B

Cout

$F_2$

$F_{1:0}$

$S$

$[N-1]$ Zero

 ticks the box for "Set Less Than (SLT) Example" and includes a diagram illustrating an ALU configuration for performing the SLT operation with inputs $A$ and $B$, and outputs $C_{out}$, $F_2$, and $F_{1:0}$. The text explains the setup for the SLT operation and provides an example with $A = 25$ and $B = 32$. The diagram shows the logical components and connections involved in the ALU for this operation.
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: $11001 >> 2 = 00110$
  - 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

A3:0 \rightarrow \text{shamt}1:0 \leftarrow 2 \rightarrow A1:0 \rightarrow \text{shamt}1:0 \leftarrow Y3:0

\text{Diagram showing the shifter design with inputs A3:0 and outputs Y3:0, shifter by 2 positions.}
Shifter

Can be implemented with a mux

\[
y_i = \begin{cases} 
  x_{i-1} & \text{if } En = 1, s = 1, \text{ and } d = L \\
  x_{i+1} & \text{if } En = 1, s = 1, \text{ and } d = R \\
  x_i & \text{if } En = 1, s = 0 \\
  0 & \text{if } En = 0
\end{cases}
\]
Shifters as Multipliers and Dividers

• A left shift by $N$ bits multiplies a number by $2^N$
  – Ex: 00001 $\ll$ 2 = 00100 ($1 \times 2^2 = 4$)
  – Ex: 11101 $\ll$ 2 = 10100 ($-3 \times 2^2 = -12$)

• The arithmetic right shift by $N$ divides a number by $2^N$
  – Ex: 01000 $\gggg$ 2 = 00010 ($8 \div 2^2 = 2$)
  – Ex: 10000 $\gggg$ 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 x 42</td>
<td>0101 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>+ 0000</td>
</tr>
<tr>
<td>result</td>
<td>0100011</td>
</tr>
</tbody>
</table>

230 x 42 = 9660

5 x 7 = 35
4 x 4 Multiplier

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

- \( Q = A/B \)
- \( R \): remainder
- \( D \): difference

\[ R = A \]

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

\[ D = R - B \]

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

else \( Q_i = 1, R' = D \)  // \( R \geq B \)

\[ R = 2R' \]