Arithmetic

COMP375 Computer Architecture and Organization

Goal for Today

• Create logic gates that perform arithmetic

Elementary School

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>010001</td>
<td></td>
</tr>
<tr>
<td>+ 7</td>
<td>+000111</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>011000</td>
<td></td>
</tr>
</tbody>
</table>

You add two numbers together. If the sum is greater than the number base, you add one to the next column. When you add two numbers, you may also have to add the carry from the column to the right.

1 Bit Adder

• A one bit adder has three inputs, numbers A and B and Carry in. There are two outputs, the Sum and Carry out.
Multiple Bit Adder

A  B
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cout</td>
<td>Sum</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>one bit adder</td>
<td>one bit adder</td>
</tr>
<tr>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>Sum</td>
<td>Sum</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

1 bit Adder Truth Table

<p>| Inputs | Outputs |
|---|---|---|---|</p>
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Cin</th>
<th>CarryIn</th>
<th>CarryOut</th>
<th>Sum</th>
<th>Comments</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 + 0 + 0 = 00_{10}</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0 + 0 + 1 = 01_{10}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 + 1 + 0 = 01_{10}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 + 1 + 1 = 10_{10}</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1 + 0 + 0 = 01_{10}</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1 + 0 + 1 = 10_{10}</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1 + 1 + 0 = 10_{10}</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1 + 1 + 1 = 11_{10}</td>
</tr>
</tbody>
</table>

Addition Sum

\[
\text{Sum} = A'B'C' + AB'C' + A'B'C + ABC
\]

Addition Carry Out

\[
\text{Cout} = AB + BC + AC
\]
One bit adder

Overflow
- If operands are too big, then sum cannot be represented as an $n$-bit 2’s complement number.

- We have overflow if:
  - signs of both operands are the same, and
  - sign of sum is different.

- Another test -- easy for hardware:
  - carry into left most bit does not equal carry out

Detecting Overflow
- When the carry into the sign bit does not match the carry out, there is an overflow

Intel Status Register

- ID Flag (ID)
- Virtual Interrupt Pending (VIP)
- Virtual Interrupt Flag (VIF)
- Alignment Check (AC)
- Virtual-8086 Mode (VM)
- Resume Flag (RF)
- Nested Task (NT)
- I/O Privilege Level (IOPL)
- Overflow Flag (OF)
- Direction Flag (DF)
- Trap Flag (TF)
- Sign Flag (SF)
- Zero Flag (ZF)
- Auxiliary Carry Flag (AF)
- Parity Flag (PF)
- Carry Flag (CF)
Saving Status

<table>
<thead>
<tr>
<th>Status flag</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overflow</td>
<td>XOR of carry into and out of the sign bit</td>
</tr>
<tr>
<td>Sign</td>
<td>Copy of the sign bit</td>
</tr>
<tr>
<td>Zero</td>
<td>NOR of all result bits</td>
</tr>
<tr>
<td>Carry</td>
<td>Carry into the sign bit</td>
</tr>
</tbody>
</table>

Intel Status Register

- The status register records the results of executing the instruction.
- Performing arithmetic sets the status register.
- The compare instruction does a subtraction, but doesn’t store the results. It just sets the status flags.
- All jump instructions are based on the status register.

1 Bit Subtractor

- A one bit subtracter has three inputs, numbers X and Y and Borrow in. There are two outputs, the Difference and Borrow out.

X-Y Subtractor Truth Table

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Bin</th>
<th>Bout</th>
<th>Diff</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>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</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>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\text{Diff} = \overline{X}'Y'B_{\text{in}}' + X\overline{Y}'B_{\text{in}}' + \overline{X}'Y'B_{\text{in}} + XYB_{\text{in}}
\]
What is the equation for $B_{out}$?

1. $B_{out} = XY + B_{in}$
2. $B_{out} = YB_{in} + XB_{in}$
3. $B_{out} = XB_{in}' + YB_{in}' + XY$
4. $B_{out} = XY + X'B_{in}$

The equation for $B_{out}$

$B_{out} = XB_{in}' + YB_{in}' + XY$

Subtraction

- Instead of building a separate subtraction circuit, you can add the negative of the operand.
- To make a twos complement number negative, you must invert the bits and add one.
- A NOT gate can be used to invert all the bits.
- Setting the Carry In on the rightmost bit will add one to the result.

$$A + B' + 1 = A + (B' + 1) = A + (-B) = A - B$$

Draw a Subtraction Circuit

- Using one bit adder boxes and OR, AND or NOT gates, draw the logic diagram for a four bit twos complement $A - B$ subtraction circuit.
Propagation Delay

- To make a 32 bit adder, you can use 32 one bit adders.
- The left most bit cannot be computed until all of the other bits are computed so that the Carry In value will be known.
- Each one bit adder requires the signal to go through two gates. Each gate takes a small amount of time to react.
- This limits the speed of the adder.

Look Ahead Carry

- The Carry Out is determined by
  \[ C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B \cdot C_i \]
  \[ C_{i+1} = A_i \cdot B_i + C_i \cdot (A_i + B_i) \]
- The \( A \cdot B \) term is true when this bit generates a carry out. Call it \( G_i \)
- The other term is true when a carry propagates from an earlier bit. Call it \( P_i \)
Arithmetic Circuits

Carry Equations

- We can express the carry as:
\[ C_{i+1} = G_i + (P_i \cdot C_i) \]
\[ C_{i+2} = G_{i+1} + (P_{i+1} \cdot G_i) + (P_{i+1} \cdot P_i \cdot C_i) \]
\[ C_{i+4} = G_{i+3} + (P_{i+3} \cdot G_{i+2}) + (P_{i+3} \cdot P_{i+2} \cdot G_{i+1}) + (P_{i+3} \cdot P_{i+2} \cdot P_{i+1} \cdot G_i) + (P_{i+3} \cdot P_{i+2} \cdot P_{i+1} \cdot P_i \cdot C_i) \]

4 bit Adder with Look Ahead Carry

Reduced Propagation

- The simple n bit ripple adder took \( O(2n) \) time to add n bits due to carry propagation.
- With carry look ahead, it takes \( O(3) \) time to propagate the carry. The look ahead requires more circuitry.

Further Simplification

- Creating a big adder out of groups of adders can reduce propagation and circuitry.
### Multiplication Tables

**Decimal**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
<td>36</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>48</td>
<td>54</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>21</td>
<td>28</td>
<td>35</td>
<td>42</td>
<td>49</td>
<td>56</td>
<td>63</td>
<td>70</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
</tr>
<tr>
<td>9</td>
<td>18</td>
<td>27</td>
<td>36</td>
<td>45</td>
<td>54</td>
<td>63</td>
<td>72</td>
<td>81</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td>100</td>
</tr>
</tbody>
</table>

**Binary**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Multiplication

- **Decimal**: 543
- **Binary**: 110

\[ 543 \times 312 = 110543 \]

- **Decimal**: 1086
- **Binary**: 110

\[ 1086 \times 101 = 11110 \]

- **Decimal**: 1629
- **Binary**: 110

\[ 1629 \times 101 = 11110 \]

- **Decimal**: 169416
- **Binary**: 11110

\[ 169416 \times 101 = 11110 \]

### Binary Multiplication

- **Binary**: 110
- **Binary**: 101

\[ 110 \times 101 = 11110 \]

- **Binary**: 000
- **Binary**: 101

\[ 000 \times 101 = 000 \]

- **Binary**: 110
- **Binary**: 101

\[ 110 \times 101 = 11110 \]

### Add and Shift

- Multiplication can be done by a series of adds and left shifts.
- Assume you have operands A & B and Product P (initially zero).

\[
\text{for each bit } i \text{ in } B \{ \\
\quad \text{if } (B_i \text{ is one}) \quad \text{add A to P} \\
\quad \text{shift A left by one bit} \\
\}
\]
Sequential Logic

- The Add and Shift multiplier is not a combinatorial logic circuit.
- **Combinatorial logic** has no memory. The output is determined by the input.
- **Sequential logic** has memory. The output is determined by the input and what happened previously.
- Add and Shift multiplier is sequential logic.

### 5 x 5 initial setup

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101</td>
<td>00101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
</tr>
</tbody>
</table>

### B₀ is 1 so Add

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101</td>
<td>00101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101</td>
</tr>
</tbody>
</table>

### Shift A left one bit

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010</td>
<td>00101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>00101</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
</tbody>
</table>
B₁ is zero so do nothing

A 01010
P 00101

Shift A left one bit

A 10100
P 00101

B₂ is 1 so Add

A 10100
P 11001

Shift A left one bit

A 01000
P 11001
B₃ is zero so do nothing

\[
\begin{array}{ccc}
A & 01000 & B \\
\uparrow & 00101 & B \\
P & 11001 & 11001
\end{array}
\]

Shift A left one bit

\[
\begin{array}{ccc}
A & 10000 & B \\
\uparrow & 00101 & B \\
P & 11001 & 11001
\end{array}
\]

B₄ is zero so do nothing

\[
\begin{array}{ccc}
A & 10000 & B \\
\uparrow & 00101 & B \\
P & 11001 & 11001
\end{array}
\]

Done

\[
\begin{array}{ccc}
A & 10000 & B \\
\uparrow & 00101 & B \\
P & 11001 & 11001
\end{array}
\]
Combinatorial Multiplier

One Bit Multiplier

if B = 0 then Pin
if B = 1 then Pin + A

3 x 3 Example

3 x 3 Example
Division

- Division is difficult. *(Every elementary school child knows that.)*
- Some small computers do not have multiplication or division.
- Some have multiplication but not division.