4 Binary Integer Arithmetic

4.1 Binary Integer Arithmetic

- The **ALU** (Arithmetic and Logic Unit) in a CPU is dedicated to perform binary integer arithmetic operations.
  - Floating-point arithmetic—most commonly based on the IEEE 754 number representation—is usually performed:
    - in software, or
    - in a separate processor, the **FPU** (Floating-Point Unit).

- Here, we will focus on the implementation of integer arithmetic: addition, subtraction, and multiplication.
A half adder

- Addition is the most basic and the most frequently used operation in an ALU
  - Note that addition is not only used for arithmetic operations but, e.g., also every time the CPU’s IP register is advanced during the fetch–execute cycle

- The rules for the bit-wise addition of two bits X and Y are readily translated into a truth table
  - the two-bit adder needs two outputs: the **sum** and the **carry**:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C</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>

\[
\begin{array}{c@{}c@{}c@{}c@{}c}
\text{Add} & \text{X} & \text{Y} & \text{C} & \text{S} \\
\hline
0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
\end{array}
\]
A half adder

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C</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>

While the carry $C$ can obviously be implemented via a single AND gate, the SOP form for the sum $S$ leads to this circuit:

$$\text{Carry} = X \cdot Y$$

$$\text{Sum} = (\overline{X} \cdot Y) + (X \cdot \overline{Y})$$
① Use fewer gates:

② Use boolean laws:

\[(\overline{X} \cdot Y) + (X \cdot \overline{Y})\]  

\[=\] Distributivity  
\[=\] Distributivity  
\[=\] Inverse, Identity, DeMorgan

\[\left((\overline{X} \cdot Y) + X\right) \cdot \left((\overline{X} \cdot Y) + \overline{Y}\right)\]

\[\left((\overline{X} + X) \cdot (Y + X) \cdot (\overline{X} + \overline{Y}) \cdot (Y + \overline{Y})\right)\]

\[\left((Y + X) \cdot (\overline{X} \cdot Y)\right)\]
Use an XOR gate:

Why half adder?

- To implement addition for \( n \) bits, \( n \geq 2 \), an adder really is a function of three inputs.

Example (binary 4-bit addition):

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
+ & 1 & 0 & 1 \\
\hline
0 & 0 & 1 & 0 \\
\end{array}
\]
A 3-bit full adder takes the carry bit ($C_{in}$) of a previous addition into account:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>$C_{in}$</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>

Note that the full adder performs the operation $\left( X + Y \right) + C_{in}$; the full adder is thus best assembled from two half adders ($Z = C_{in}$):
Parallel $n$-bit adder (ripple-carry adder)

- Verify the behavior of the full adder: $\text{Carry} = C_1 + C_2$, $C_1 = X \cdot Y$, $C_2 = Z \cdot S_1$, $S_1 = X \text{ XOR } Y$, $S_2 = Z \text{ XOR } S_1$

- A **parallel** $n$-bit adder $C = A + B$ is easily assembled from $n$ full adders:

  - Note: Add0 is implemented as a *full* adder here for a reason to be explained shortly (in principle, a half adder Add0 would be sufficient)
Subtraction and Two’s complement

- The representation of negative numbers in **two’s complement** is very well-suited for the construction of **subtraction** circuits:

  \[ b_{n-1} b_{n-2} \ldots b_1 b_0 \quad \equiv \quad \text{Two’s Complement} \quad -2^{n-1} \cdot b_{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i \]

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>-128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

128 + 8 = -120

**n-bit Two’s complement**

- **Number range:** \(-2^{n-1} \ldots 2^{n-1} - 1\)
- **# representations of 0:** one
- **Negation:** negate each bit, then add 1
- **Subtraction:** \(A - B = A + (-B)\)
• Why does the negation rule work at all?

  – Let $B = b_{n-1}b_{n-2}\ldots b_1b_0$
  – **Test**: compute $B - B = B + (-B) \overset{!}{=} 0$
  – Two’s complement: $-B = \overline{b_{n-1}b_{n-2}\ldots b_1b_0} + 1$

\[
B + (-B)
\]

\[
= -2^{n-1} \cdot b_{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i + -2^{n-1} \cdot \overline{b_{n-1}} + \sum_{i=0}^{n-2} \overline{b_i} \cdot 2^i + 1
\]

\[
\underbrace{B}_B + \underbrace{-B}_B
\]

\[
= -2^{n-1} \cdot (b_{n-1} + \overline{b_{n-1}}) + \sum_{i=0}^{n-2} (b_i + \overline{b_i}) \cdot 2^i + 1
\]

\[
= -2^{n-1} + \sum_{i=0}^{n-2} 2^i + 1
\]

\[
= -2^{n-1} + (2^{n-1} - 1) + 1 = 0
\]
A parallel $n$-bit subtractor is based on a $n$-bit parallel adder and gates to implement the negation rule: ① negate each bit, then ② add 1

① XOR can act as controllable negator:

<table>
<thead>
<tr>
<th>$b$</th>
<th>negate</th>
<th>$b$ XOR negate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$b$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$\overline{b}$</td>
</tr>
</tbody>
</table>

② Full adder Add0: set $C_{in} = 1$ for subtraction, $C_{in} = 0$ for addition
Overflow

- Given a fixed number $n$ of bits, an addition (or subtraction) may lead to **overflow**: the result lies outside the $-2^{n-1} \ldots 2^{n-1} - 1$ range

**Examples** ($n = 4$, range $-8 \ldots 7$, no overflow):

\[
\begin{align*}
0010 \ (2) & \quad 0101 \ (5) & \quad 1011 \ (-5) \\
+ \ 1001 \ (-7) & \quad + \ 1110 \ (-2) & \quad + \ 1110 \ (-2) \\
1011 \ (-5) & \quad 0011 \ (3) & \quad 1001 \ (-7)
\end{align*}
\]

**Examples** (overflow, result unusable):

\[
\begin{align*}
0111 \ (7) & \quad 1010 \ (-6) \\
+ \ 0111 \ (7) & \quad + \ 1100 \ (-4) \\
1110 \ (-2) & \quad 0110 \ (6)
\end{align*}
\]

**Overflow rule**: overflow occurs, if result has negative (positive) sign but both operands have a positive (negative) sign
Binary multiplication

- In principle, the multiplication $A \times B$ could be implemented via $(B - 1)$-fold addition of $A$ to itself; this would be way too slow.

- Instead, ALUs realize multiplication via **bit shifting**, much like the *pencil and paper* multiplication in base 10:

  **Example** (unsigned 4-bit multiplication [no two’s complement]):

  \[
  \begin{array}{c}
  1011 \quad (11) \\
  \times 1101 \quad (13) \\
  \hline
  1011 \\
  0000 \\
  1011 \\
  +1011 \\
  \hline
  10001111 \quad (143)
  \end{array}
  \]

  - Each 0-bit in the multiplier leads to a left bit shift and an addition of 0.
  
  $\Rightarrow$ shift only
Multiplication algorithm

• In general, the multiplication of two \( n \)-bit numbers leads to a \( 2n \)-bit result

• Java: multiplication \( a \times c \) of two 16-bit numbers (32-bit result) based on bit shifting (\(<<, >>\)), bit testing (\&), and addition (+):

```java
public class Multiply {
    static int multiply (int a, int c) {
        int i;
        c = c << 16;
        for (i = 0; i < 16; i++) {
            if ((a & 1) != 0)
                a = a + c;
            a = a >> 1;
        }
        return a;
    }

    public static void main (String[] args) {
        System.out.println (multiply (11, 13));
    }
}
```
8-bit multiplication (unsigned)

- no temporary variable is needed to store the partial products
- the final product $A \times C$ is available in $A$
8-bit multiplication (complete run)
Bit shifting may be implemented using **crosspoint switches**, a matrix circuit of $n \times n$ switches which may be selectively connected.

**Example** ($n = 8$, shift 8-bit word $x_7x_6 \ldots x_1x_0$):
Multiplication with two’s complement?

- Two’s complement works well for addition and subtraction—how about multiplication?

**Example** (numbers interpreted in two’s complement):

\[
\begin{array}{c}
1011 \quad (-5) \\
x 0111 \quad (7) \\
\hline \\
1011 \\
1011 \\
1011 \\
1011 \\
+ 0000 \\
\hline 01001101 \quad (77_{10})
\end{array}
\]

- This multiplication scheme fails if either operand is negative
Multiplication with two’s complement?

- Why does multiplication fail?

- Remember: the multiplication of two $n$-bit numbers requires $2n$-bit additions to add the partial products
  - This is what actually happens:

\[
\begin{array}{c}
1011 \quad (-5) \\
\times \quad 0111 \quad (7) \\
\hline
00001011 \\
00010110 \\
00101100 \\
+00000000 \\
\hline
01001101 \\
\end{array}
\]

- Note: interpreted as 8-bit numbers, the shifted numbers in the ◆ lines are positive (and not negative as expected)
Multiplication with two’s complement?

- To **extend** a $n$-bit two’s complement number to $m$ bits ($m > n$), pad the number on the left; the new $m - n$ bits have the **same value as the sign bit**

**Example** (extend 4-bit to 8-bit number, two’s complement):

\[-5_{10} = 1011_2 = \underbrace{11110111}_\text{new}\]

- 4-bit multiplication (corrected 8-bit extension):

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & \times & 0 & 1 & 1 & 1 \\
\hline
& 1 & 1 & 1 & 1 & 0 & 1 & 1 & \\
& 1 & 1 & 1 & 0 & 1 & 1 & 0 & \\
& 1 & 1 & 1 & 0 & 1 & 1 & 0 & \\
+ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
\hline
1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & (-35)
\end{array}
\]
Booth’s algorithm

- **Booth’s algorithm** is capable of multiplying two’s complement numbers; multiplicand and/or multiplier may be negative
  - As before, two $n$-bit numbers are multiplied to yield a $2n$-bit result
  - Input: $A = A_{n-1}A_{n-2} \ldots A_1A_0$, $C$ ($n$-bit numbers)
  - Output: $RA \equiv A \times C$
  - Booth’s algorithm uses an operation ASR (**arithmetic shift right**) which behaves *almost* like a right bit shift: the sign bit is preserved

**Example** (right shift and ASR of 4-bit number):

\[
\begin{align*}
1011 & \rightarrow 0101 \\
\text{right shift} & \rightarrow 0101
\end{align*}
\]

\[
\begin{align*}
1011 & \rightarrow 1101 \\
\text{ASR} R,A,A_{-1} & \rightarrow 1101
\end{align*}
\]

\[
\begin{align*}
0011 & \rightarrow 0001 \\
\text{ASR} R,A,A_{-1} & \rightarrow 0001
\end{align*}
\]

- In the algorithm, ASR is applied to 3 “connected” registers:
Booth’s algorithm

\[ A = 1011_2 = -5_{10} \quad C = 0111_2 = 7_{10} \]

<table>
<thead>
<tr>
<th>( i )</th>
<th>( R )</th>
<th>( A )</th>
<th>( A_{-1} )</th>
<th>( C )</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0000</td>
<td>1011</td>
<td>0</td>
<td>0111</td>
<td>( A_0A_{-1} = 10 )</td>
</tr>
<tr>
<td>4</td>
<td>1001</td>
<td>1011</td>
<td>0</td>
<td>0111</td>
<td>( R \leftarrow R - C )</td>
</tr>
<tr>
<td>4</td>
<td>1100</td>
<td>1101</td>
<td>1</td>
<td>0111</td>
<td>ASR</td>
</tr>
<tr>
<td>3</td>
<td>1100</td>
<td>1101</td>
<td>1</td>
<td>0111</td>
<td>( A_0A_{-1} = 11 )</td>
</tr>
<tr>
<td>3</td>
<td>1110</td>
<td>0110</td>
<td>1</td>
<td>0111</td>
<td>ASR</td>
</tr>
<tr>
<td>2</td>
<td>1110</td>
<td>0110</td>
<td>1</td>
<td>0111</td>
<td>( A_0A_{-1} = 01 )</td>
</tr>
<tr>
<td>2</td>
<td>0101</td>
<td>0110</td>
<td>1</td>
<td>0111</td>
<td>( R \leftarrow R + C )</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1011</td>
<td>0</td>
<td>0111</td>
<td>ASR</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1011</td>
<td>0</td>
<td>0111</td>
<td>( A_0A_{-1} = 10 )</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>1011</td>
<td>0</td>
<td>0111</td>
<td>( R \leftarrow R - C )</td>
</tr>
<tr>
<td>1</td>
<td>1101</td>
<td>1101</td>
<td>1</td>
<td>0111</td>
<td>ASR</td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td>1101</td>
<td>1</td>
<td>0111</td>
<td>END</td>
</tr>
</tbody>
</table>

\[ RA = 11011101_2 = -35_{10} \]
Optimization: carry-select adder

• Note that in the ripple-carry adder scheme, full adder \( \text{Add}(k + 1) \) can only start to compute its output once \( \text{Add}k \) has completed its job (computed the carry)
  
  – Although adder circuits are simple and thus incur small delays only, addition is so important that any method for speedup is welcome

• In an \( n \)-bit **carry-select adder**, the additions of the *lower halfword* (bits \( 0 \ldots \frac{n}{2} - 1 \)) and the *upper halfword* (bits \( \frac{n}{2} \ldots n \)) are done in **parallel**

  1. Since the value of the carry bit is still unknown when the upper halfword addition starts, **two full adder circuits** perform this addition in parallel, one assuming \( c_{\text{in}} = 0 \), the other assuming \( c_{\text{in}} = 1 \)

  2. Once the \( c_{\text{out}} \) of the lower halfword addition is known, a **multiplexer** selects the correct upper halfword sum
Carry-select adder

**Example** (8-bit carry-select adder):

![Diagram of a 4-bit full adder and a carry-select adder setup with inputs and outputs labeled.](image)
• The carry-select adder performs twice as fast as a ripple-carry adder while using 50% more circuitry.

• The two-bit multiplexer selects input A or B, depending on control line X (in the carry-select adder, $X = C_{\text{out}}$ of the lower halfword sum):

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

- Karnaugh map optimization yields:

$$O = (B \cdot X) + (A \cdot \overline{X})$$