Logic Design

Number Representation and Arithmetic Circuits

Number representation
- Numbers that are positive only are called unsigned
- Numbers that can be positive or negative are called signed
- Numbers could be integer or real
- Simplest: unsigned integer
- A decimal integer:

\[
D = d_n d_{n-1} \ldots d_1 d_0
\]

\[
V(D) = d_n \times 10^n + d_{n-1} \times 10^{n-1} + \ldots + d_1 \times 10^1 + d_0 \times 10^0
\]

Number representation
- In a binary number the right-most bit is called the least-significant bit (LSB) and the left-most bit is called the most significant bit (MSB)
- A group of 4 bits is called a nibble
- A group of 8 bits is called a byte

Number representation
- Binary numbers:

\[
B = b_n b_{n-1} \ldots b_1 b_0
\]

\[
V(B) = b_n \times 2^n + b_{n-1} \times 2^{n-1} + \ldots + b_1 \times 2^1 + b_0 \times 2^0
\]

\[
1101
\]

\[
V = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13
\]

\[
(1101)_2 = (13)_{10}
\]

Number representation
- Conversion from decimal to binary: successively divide by 2
- In each step the remainder is the next binary digit
- The process continue until the quotient becomes zero

\[
V = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \ldots + b_1 \times 2^1 + b_0 \times 2^0
\]

\[
\frac{V}{2} = b_{n-1} \times 2^{n-2} + b_{n-2} \times 2^{n-3} + \ldots + b_1 \times 2^0 + \frac{b_0}{2}
\]

Number representation
- Numbers that are positive only are called unsigned
- Numbers that can be positive or negative are called signed
- Numbers could be integer or real
- Simplest: unsigned integer
- A decimal integer:

\[
D = d_n, d_{n-1}, \ldots, d_1, d_0
\]

\[
V(D) = d_n \times 10^n + d_{n-1} \times 10^{n-1} + \ldots + d_1 \times 10^1 + d_0 \times 10^0
\]

Number representation
- Binary numbers:

\[
B = b_n, b_{n-1}, \ldots, b_1, b_0
\]

\[
V(B) = b_n \times 2^n + b_{n-1} \times 2^{n-1} + \ldots + b_1 \times 2^1 + b_0 \times 2^0
\]

<table>
<thead>
<tr>
<th>Convert (857)_{10}</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>857 \div 2 = 428\ 1</td>
<td>LSB</td>
</tr>
<tr>
<td>428 \div 2 = 214\ 0</td>
<td></td>
</tr>
<tr>
<td>214 \div 2 = 107\ 0</td>
<td></td>
</tr>
<tr>
<td>107 \div 2 = 53\ 1</td>
<td></td>
</tr>
<tr>
<td>53 \div 2 = 26\ 1</td>
<td></td>
</tr>
<tr>
<td>26 \div 2 = 13\ 0</td>
<td></td>
</tr>
<tr>
<td>13 \div 2 = 6\ 1</td>
<td></td>
</tr>
<tr>
<td>6 \div 2 = 3\ 0</td>
<td></td>
</tr>
<tr>
<td>3 \div 2 = 1\ 1</td>
<td></td>
</tr>
<tr>
<td>1 \div 2 = 0\ 1</td>
<td></td>
</tr>
</tbody>
</table>

Result is (1101011001)_2
Number representation

- The most common bases in addition to decimal are:
  - base 2 (binary) \{ 0, 1 \}
  - base 8 (octal) \{ 0, 1, \ldots, 7 \}
  - base 16 (hexadecimal) \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F \}
- Reason for using octal and hexadecimal systems: useful shorthand notation for binary numbers

Addition of Unsigned Numbers

\[
\begin{align*}
X &= x_4x_3x_2x_1x_0 = 01111_{10} \\
Y &= y_4y_3y_2y_1y_0 = 01010_{10} \\
S &= s_4s_3s_2s_1s_0 = 11001_{10}
\end{align*}
\]

Generated carries

Addition of Unsigned Numbers

- One octal digit represents three bits
- Conversion from binary to octal: starting from the LSB replace every group of three digits with their corresponding octal digit
- Conversion from binary to hexadecimal: starting from the LSB replace every group of four digits with their corresponding hexadecimal digit
- Conversion from octal to binary: substitute each octal digit by corresponding three bits
- Conversion from hexadecimal to binary: substitute each hex digit by four bits

Decomposed Full Adder

\[
\begin{align*}
&y_0 \quad y_1 \quad y_2 \\
&x_0 \quad x_1 \quad x_2 \\
&z_0 \quad z_1 \quad z_2
\end{align*}
\]

(a) Block diagram

(b) Detailed diagram

Figure 5.4: Full adder.

(b) Detailed diagram

(a) Block diagram
Ripple Carry Adder

- When operands X and Y are applied as inputs to the adder, it takes some time before output sum S is valid.
- Each full-adder has a delay before its s_i and c_{i+1} are valid
- If this delay is $\Delta t$ the complete sum will be valid after a delay of $n \Delta t$
- Because of the way the carry signal “ripple” through the full-adder, this circuit is called a ripple-carry adder

Example

- One of the bits (usually the left-most bit) is reserved for the sign of the number.
- Usually a 1 indicates negative and 0 indicates positive.

Signed Numbers

- Extending the 'natural' binary representation of positive integers to negative integers can be done in at least 3 different schemes: sign-magnitude, one's complement and two's complement.
- Sign-and-magnitude: The most significant bit (MSB) is reserved to the sign, 0 is positive, 1 is negative. All other bits are used to store the magnitude in the natural representation.
- Addition and subtraction are complicated.
- There are two representations for zero!
Signed Numbers
• One’s complement Positive integers are like in the natural representation, negative numbers are obtained by complementing each bit of the corresponding positive number (i.e. the absolute value).
• There are two representations for zero! Bitwise addition of N and -N gives -0.
• Positive integers still have MSB = 0, and negative integers have MSB=1.
• 1’s complement of an n-bit negative number K is obtained by subtracting its equivalent positive number P from 2^n-1
  \(K_1 = (2^n - 1) - P\)

Signed Numbers
• Two’s complement Like one’s complement, but negative numbers are having 1 added after complementation.
• Bitwise addition of N and -N gives 0 if you ignore the carry out of the MSB.
• Positive integers still have MSB = 0, and negative integers have MSB=1. Only one representation for zero!
• 2’s complement of an n-bit negative number K is obtained by subtracting its equivalent positive number P from 2^n
  \(K_2 = 2^n - P\)

2’s complement signed numbers
\[ B-b_n b_{n-1} \ldots b_2 b_1 \]
\[ V = (-b_{n-1} \times 2^{n-1}) + b_{n-2} \times 2^{n-2} + \ldots + b_2 \times 2^2 + b_1 \times 2^1 \]
Largest negative number: \(-2^{n-1}\)
Largest positive number: \(2^{n-1} - 1\)

Signed Numbers
• Relationship between 2’s complement and 1’s complement
  \(K_2 = K_1 + 1\)

1’s complement addition
\[ (+5) + (+2) = 0 \ 1 \ 0 \ 1 \ 0 + 0 \ 0 \ 1 \ 0 \ 0 = 0 \ 1 \ 1 \ 1 \ 1 \]
\[ (+5) + (+2) = 0 \ 1 \ 0 \ 1 \ 0 + 0 \ 0 \ 1 \ 0 \ 0 = 0 \ 1 \ 1 \ 1 \ 1 \]

Table: Signed Binary Integers (word length \(n = 4\))

<table>
<thead>
<tr>
<th>Positive Integers (all systems)</th>
<th>-N</th>
<th>Sign and Magnitude</th>
<th>2’s Complement</th>
<th>1’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0 0000</td>
<td>-0</td>
<td>0000</td>
<td>0000</td>
<td>1111</td>
</tr>
<tr>
<td>+1 0001</td>
<td>-1</td>
<td>0100</td>
<td>1100</td>
<td>1101</td>
</tr>
<tr>
<td>+2 0010</td>
<td>-2</td>
<td>1010</td>
<td>1110</td>
<td>1101</td>
</tr>
<tr>
<td>+3 0011</td>
<td>-3</td>
<td>1101</td>
<td>1111</td>
<td>1110</td>
</tr>
<tr>
<td>+4 0100</td>
<td>-4</td>
<td>1000</td>
<td>1000</td>
<td>1011</td>
</tr>
<tr>
<td>+5 0101</td>
<td>-5</td>
<td>1010</td>
<td>1011</td>
<td>1010</td>
</tr>
<tr>
<td>+6 0110</td>
<td>-6</td>
<td>1100</td>
<td>1001</td>
<td>1001</td>
</tr>
<tr>
<td>+7 0111</td>
<td>-7</td>
<td>1111</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>
Addition and Subtraction

- Addition of 1’s complement numbers might need a correction
- Time needed to add two 1’s complement numbers may be twice as long as time needed to add two unsigned numbers

2’s complement addition

<table>
<thead>
<tr>
<th>(+5)</th>
<th>0 1 0 1</th>
<th>(+2)</th>
<th>0 0 1 0</th>
<th>(+7)</th>
<th>0 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-6)</td>
<td>1 0 1 1</td>
<td>(-3)</td>
<td>1 1 0 1</td>
<td>(-5)</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>(+3)</td>
<td>1 0 0 1</td>
<td>(-7)</td>
<td>1 1 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ignore</td>
<td></td>
<td>ignore</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2’s complement subtraction

<table>
<thead>
<tr>
<th>(+5)</th>
<th>0 1 0 1</th>
<th>(-2)</th>
<th>1 1 1 0</th>
<th>(+7)</th>
<th>0 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-6)</td>
<td>1 0 1 1</td>
<td>(-3)</td>
<td>1 1 0 1</td>
<td>(-5)</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>(+3)</td>
<td>1 0 0 1</td>
<td>(-7)</td>
<td>1 1 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ignore</td>
<td></td>
<td>ignore</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adder and Subtractor Unit

Radix-complement schemes

- Complements – general theory
  - The r’s complement of an n-digit number N in base r is:
    \[ K_r = r^n - N \quad \text{for } N \neq 0 \]
    \[ (0 \text{ for } N=0) \]
  - The (r-1)’s complement, \( K_{r-1} \) is defined as:
    \[ K_{r-1} = (r^{n-1}) - N \]
  - The concept of subtracting a number by adding its radix-complement is general

Arithmetic Overflow

- If n bits are used to represent signed numbers, result must be in the range \(-2^{n-1} \text{ to } 2^{n-1}-1\)
- If the result does not fit in this range, we say that arithmetic overflow has happened
- We should be able to detect overflow
- The key to determining the overflow is carry-out from MSB position and carry-out from the sign bit
- If they are the same no overflow has happened.
  \[ \text{overflow} = c_{n-1} \oplus c_n \]
Arithmetic Overflow

| (+7) | 0111 |
| + (+2) | 0010 |
| (+5) | 1001 |
| + (+4) | 1001 |

\[ q_i = 0 \]
\[ s_i = 1 \]
\[ c_i = 0 \]

\[ q_i = 0 \]
\[ s_i = 1 \]
\[ c_i = 0 \]

Performance Issue

- Speed of any circuit is limited by the longest delay along the paths through the circuit
- This is called the critical path delay
- Critical path for the ripple adder is from input y, through the XOR gate and through the carry circuit of each stage.

Fast Adders

\[ c_i = x_i y_i + x_i y_i c_i \]
\[ c_i = x_i y_i + (x_i + y_i)c_i \]
\[ c_i = g_i + p_i c_i \]
\[ g_i = x_i y_i \]
\[ p_i = x_i + y_i \]

\[ c_i = g_i + p_i c_i + p_i p_i c_i + \cdots + p_i p_i \cdots + p_i p_i \cdots + p_i p_i c_i \]

Fast Adders

Figure 5.15. A ripple-carry adder based on expression 5.3.

Fast Adders

Figure 5.16. The first two stages of a carry-lookahead adder.
Fast Adders

- In an n-bit carry-look-ahead adder the final carry-out signal would be produced after three gate delays.
- The total delay in an n-bit carry-look ahead adder is four gate delays.
- Complexity of an n-bit carry look-ahead adder increases rapidly as n becomes larger.
- We can use a hierarchical approach in designing large adders.

Fast Adders

\[
c_i = s_i + P_i s_i + P_iP_j s_i + P_iP_jP_k s_i + P_iP_jP_kP_l s_i + \ldots
\]

\[
P_i = P_iP_jP_kP_lP_mP_nP_o
\]

\[
G_i = s_i + P_i s_i + P_iP_j s_i + P_iP_jP_k s_i + P_iP_jP_kP_l s_i + P_iP_jP_kP_lP_m s_i + \ldots
\]

\[
c_{i+1} = G_i + P_i c_i
\]

Fast Adders

Figure 5.17. A hierarchical carry-lookahead adder with ripple-carry between blocks.

Fast Adders

- A faster circuit can be designed in which a second-level carry-look-ahead is performed to produce quickly the carry signals between blocks.
- Instead of producing a carry-out signal from the most significant bit of the block, each block produces generate and propagate signals for the entire block.

Fast Adders

Technology Considerations

- So far we assumed gates with any number of inputs can be used.
- Fan-in is limited to a small number.
- More gates should be used to implement the logic.
- Example: max fan-in is four.

\[
c_i = s_i + P_i s_i + P_iP_j s_i + P_iP_jP_k s_i + P_iP_jP_kP_l s_i + P_iP_jP_kP_lP_m s_i + \ldots
\]

\[
c_i = (s_i + P_i s_i + P_iP_j s_i + P_iP_jP_k s_i + P_iP_jP_kP_l s_i + P_iP_jP_kP_lP_m s_i + \ldots)
\]

\[
+ (P_iP_jP_kP_lP_mP_nP_o c_i)
\]
Because fan-in limitation reduces the speed of carry-look-ahead adder, some devices with low fan-in include dedicated circuit for implementing fast adders

Example: FPGA

Multiplication

- A number is multiplied by $2^k$ by shifting it left by $k$ bit positions
- This is true both for unsigned and signed numbers
- Shifting to the right by $k$ bit positions, is equivalent to dividing by $2^k$
- For unsigned numbers the empty bit positions are filled with zero
- For signed numbers, in order to preserve the sign, the empty bit positions are filled with the sign bit

Multiplication of unsigned numbers

Each multiplier bit is examined: if 1, a shifted version of the multiplicand is added to form the partial product; if zero nothing is added

Multiplicand $M$ (14)  1 1 1 0
Multiplier $Q$  (11)  $\times$ 1 0 1 1

| Partial product 0 | 1 1 1 0 + 1 1 1 0 |
| Partial product 1 | 1 0 1 0 1 |
| Partial product 2 | 0 1 0 1 0 + 1 1 1 0 |

Product $P$  (154)  1 0 0 1 1 0 1 0

(a) Multiplication by hand

Multiplication of unsigned numbers

| Multiplicand $M$  (11) | 1 1 1 0 |
| Multiplier $Q$  (14) | $\times$ 1 0 1 1 |

| Partial product 0 | 1 1 1 0 |
| Partial product 1 | 1 0 1 0 1 |
| Partial product 2 | 0 1 0 1 0 + 1 1 1 0 |

Product $P$  (154)  1 0 0 1 1 0 1 0

(b) Multiplication for implementation in hardware

- $B=011000=24$
- $B/2=001100=12$
- $B/4=000110=6$
- $B=101000=24$
- $B/2=110100=12$
- $B/4=111010=6$

Multiplication

$$M = m_0m_1m_2m_3$$

$$Q = q_0q_1q_2q_3$$

$$PP0 = pp0_0\ pp0_1\ pp0_2\ pp0_3$$

$$PP0 = 0\ pp0_1\ pp0_2\ pp0_3\ pp0_0$$

$$+ m_0q_0\ m_1q_1\ m_2q_2\ m_3q_3\ 0$$

$$PP1 = pp1_0\ pp1_1\ pp1_2\ pp1_3\ pp1_4$$

$PP0_0$ and $PP0_1$ are added to partial product of the previous step.
Multiplication of Signed Numbers

- If multiplier is positive essentially the same scheme as unsigned numbers can be used
- Since shifting the multiplicand to the left results in one of the operands having n+1 bits, the addition has to be performed using the second operand represented in n+1 bits
- An n bit signed number is represented as an n+1 bit number by replicating the sign bit
- Replication of the sign bit is called sign extension

Fixed point

- A fixed point number consists of integer and fraction parts.
- The position of radix point is fixed
\[ B = b_n b_{n-1} \ldots b_1 b_0 . b_{-1} b_{-2} \ldots \]
\[ V(B) = \sum_{i=-d}^{n-1} b_i \times 2^i \]

Floating point

- Fixed point numbers: limited range
- Floating point: numbers are represented by a mantissa and an exponent: \[ \text{Mantissa} \times R^{\text{Exponent}} \]
- Normalized: radix point is the right of fist nonzero digit
- Example: 5.234 \times 10^{43}
- For binary R=2
- How mantissa and exponent are represented has been standardized by IEEE
- Single precision (32 bits) and double precision (64 bits)
Binary coded decimal (BCD)

- Each digit in a decimal number is represented by its binary form
- Since there are 10 digits we need 4 bits per digit

<table>
<thead>
<tr>
<th>Decimal digit</th>
<th>BCD code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
</tbody>
</table>

• Single precision
  - Exponent=E-127
  - Value=(+ or -)1.M x 2^E-127

• Double precision
  - Exponent=E-1023
  - Value=(+ or -)1.M x 2^E-1023

BCD

- BCD representation was used in some early computers
- Drawback: complexity of circuits that perform arithmetic operations
- BCD addition:
  - X and Y two BCD digits (each four bits)
  - \( S=X+Y \)
  - If \( X+Y \leq 9 \) the addition is the same as the addition of 2 unsigned binary numbers
  - If \( X+Y > 9 \) the result requires two BCD digits and the four-bit sum may be incorrect.
ASCII code

- ASCII code: the most popular code for representing information in digital systems used for letters, numbers, and some control characters.
- Control characters: those needed in computer systems to handle and transfer data, e.g., return character.
- ASCII representation of numbers is not convenient for arithmetic operations.
- It is best to convert ASCII numbers to binary for arithmetic operations.

<table>
<thead>
<tr>
<th>ASCII Code</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>

ASCII code uses 7-bit, natural size in computer systems is one-byte (8-bits).

Two common ways on going to 8-bits:
- Set the eight bit to 0
- Use the eight-bit to indicate the parity of the other bits

Even parity: the parity bit is given a value such that total number of 1’s is even.
Odd parity: the parity bit is given a value such that total number of 1’s is odd.

- Even parity generator: \( p = x_0 \oplus x_2 \oplus x_4 \oplus \ldots \oplus x_7 \)
- Parity checker: \( c = p \oplus x_0 \oplus x_2 \oplus x_4 \oplus \ldots \oplus x_7 \)