Signed numbers

- Non-negative (unsigned) numbers are well understood
- An n-bit number represents numbers from 0 to $2^n - 1$
- However, negative numbers cause another problem
- In all solutions, one bit is needed to represent the sign, + or -
- MSB (Most Significant Bit) can be used for that purpose, i.e., represent sign (0: +ve 1: -ve)
- Remaining bits can be interpreted differently
  - They can represent magnitude as a positive number
  - They can be complemented (represent 0 by 1 and 1 by 0)
  - Or manipulate in some other way

Non-negative (unsigned) numbers are well understood

An n-bit number represents numbers from 0 to $2^n - 1$

However, negative numbers cause another problem

In all solutions, one bit is needed to represent the sign, + or -

MSB (Most Significant Bit) can be used for that purpose, i.e.,
represent sign (0: +ve 1: -ve)

Remaining bits can be interpreted differently

– They can represent magnitude as a positive number
– They can be complemented (represent 0 by 1 and 1 by 0)
– Or manipulate in some other way

Signed numbers

- Non-negative (unsigned) numbers are well understood
- An n-bit number represents numbers from 0 to $2^n - 1$
- However, negative numbers cause another problem
- In all solutions, one bit is needed to represent the sign, + or -
- MSB (Most Significant Bit) can be used for that purpose, i.e.,
represent sign (0: +ve 1: -ve)
- Remaining bits can be interpreted differently
  - They can represent magnitude as a positive number
  - They can be complemented (represent 0 by 1 and 1 by 0)
  - Or manipulate in some other way

Example

Consider the bit string 1010:

- Sign and Magnitude
  - 1 010
  - ve 2
  - So it represents -2
- 1’s Complement
  - 1 010 → 1 101
  - ve 5
  - So it represents -5
- 2’s Complement
  - 1 010 → 1 101 + 1
  - ve 6
  - So it represents -6

Interpretation

- Sign and Magnitude
  - Out of n bits, one is reserved for sign
  - Remaining bits represent the value of number as positive
  - It is equivalent of representing it as $- (1 - 2 x_{n-1}) \sum_{i=0}^{n-2} 2^i x_i$
- 1’s Complement
  - Convert the magnitude of number as a binary string
  - Then complement every bit (replace 1 by 0 and 0 by 1)
  - This is equivalent of having the weight of MSB as $-(2^{n-1})$
- 2’s Complement
  - Convert the magnitude of number as a binary string
  - Complement every bit (replace 1 by 0 and 0 by 1) and add 1
  - This is equivalent of having the weight of MSB as $-2^{n-1}$

Sign Magnitude, 1’s, and 2’s complement

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

Maximum and Minimum values in n-bits

- We use 2’s complement as it makes arithmetic (add/sub) simple
- n-bits uses only n-1 bits to store the value
- Largest positive value is $2^n - 1$
- Largest negative value is $-2^{n-1}$
- For n=4, these values are +7 and -8
- For n=8, these values are +127 and -128
- If we need larger or smaller values to be stored, we have problem – leads to overflow and underflow
- For MULTI/DIV, sign and magnitude is better
  - But we cannot keep switching

Negation

- To change sign of a number
- In Sign and Magnitude
  - Just complement the sign
- 1’s Complement
  - Complement all bits
- 2’s Complement
  - Complement all bits and add 1
- Adding 1 is expensive operation (Example: Add 1 to 0111)
- Alternate 2’s complement method
  - Scan the string from right
  - Retain all bits up to the first 1
  - Then complement the remaining bits

Example:
6 = 0110
-6 = 1010

Example: 6 = 0110
-6 = 1010

Example: 6 = 0110
-6 = 1010

Example: 6 = 0110
-6 = 1010
Negation Examples

Negate the following 4-bit 2's Complement Binary Values:

- 0011 1111 0111 1010
- 1100+1 0000+1 1000+1 0101+1
  \(\rightarrow\) 1101 \(\rightarrow\) 0001 \(\rightarrow\) 1001 \(\rightarrow\) 0110

- What is the negation of 1000 in 4-bit 2's complement?

Converting negative number to Binary

- Convert a negative decimal number to binary in 2's complement
  - Method 1:
    - Convert the magnitude to an n-bit string
    - Negate the number
    - Example: -5  Magnitude in binary: 0101  Negation: 1011
  - Method 2:
    - The magnitude of number must be less than or equal to 2^n-1
    - Add 2^n to the number
    - Convert this number as an n-bit unsigned integer
    - Example: -4 + (16) = 12 (decimal) = 1100 (binary)
      -7 + (16) = (decimal) = (binary)

ADD/SUB with more than one bit

- Follow rules of decimal arithmetic
- Add carry to/sub borrow from the next digit
- In 2's complement, if we simply add or subtract without regard to sign, we get correct result if there is no overflow/underflow
- Overflow/Underflow occurs when the carry into and the carry out of the sign bit position are different.
- Examples

ADD/SUB revisited

- Understand the examples again
- Overflow
  - When two positive numbers added together or a negative number subtracted from a positive number yields negative
- Underflow
  - When two negative numbers added together or a positive number subtracted from a negative number yields positive

Using 1-bit building blocks to make n-bit circuit

- Design a 1-bit circuit with proper “glue logic” to use it for n-bits
  - It is called a bit slice
  - The basic idea of bit slicing is to design a 1-bit circuit and then piece together n of these to get an n-bit component
  - Example:
    - A half-adder adds two 1-bit inputs
    - Two half adders can be used to add 3 bits
    - A 3-bit adder is a full adder
    - A full adder can be a bit slice to construct an n-bit adder
Two half adders can be used to add 3 bits.

An n-bit adder can be built by full adders, and n can be arbitrary large.

Logic Functions of 2 Variables

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</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>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</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>
</tbody>
</table>

- F1 is called a logical AND, denoted by X.Y.
- F6 is called an XOR (Exclusive-OR), denoted by X ⊕ Y.
- F7 is called OR, denoted by X + Y.
- F8 is NOR, denoted by X + Y.
- F9 is called an XNOR (Exclusive-NOR), denoted by X ⊕ Y.
- F14 is NAND, denoted by X.Y.

Truth Tables for 2 Variable Functions

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>AND</th>
<th>OR</th>
<th>XOR</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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- NAND denoted by X.Y
- NOR denoted by X + Y
- XNOR denoted by X ⊕ Y
- NOT denoted by X' or X