Information Representation (level ISA-3)

Signed numbers and bit-level operations

Representing signed numbers

- Unsigned binary numbers are limited in their usefulness; to be able to perform a full set of arithmetic operations, signed numbers are necessary
- The most obvious way to store a binary signed number is to reserve a bit to indicate the sign

![Possible representation for a 5-bit signed number](image)

Representing signed numbers

- At first glance, it seems that the most obvious representation would store positive and negative values using identical bit patterns, except for the most significant bit:
  - 10010010₂ would be $-18$ while
  - 00010010₂ would be $+18$
- However, there are problems with this scheme:
  - The sum of the two values should be 0; but the sum of the values above would be 10100100₂, or $-36$
  - If 00000000 represents 0, what does 10000000 represent?

Negative numbers and 2’s complement

- 2’s complement notation solves both of the problems cited on the previous slide
- Notation uses sign bit, with 1 representing negative and 0 representing positive, but similarity ends there
- To take the 2’s complement of a binary number, you first take the 1’s complement, then add 1 to the result

Obtaining 2’s complement

- The 1’s complement of a binary number is the same number with all of the 1 digits converted to 0s and all the 0 digits converted to 1s
- Example: 01001100, becomes 10110011₂
- The 2’s complement is obtained by adding 1:
  \[
  \begin{array}{c}
  \text{10110011} \\
  + \quad 1 \\
  \hline \\
  \text{10110100}
  \end{array}
  \]

Problems solved?

- The first problem with our original scheme was that the sum of a number and its negative should add up to 0; using the previous example:
  \[
  \begin{array}{c}
  \text{01001100} \\
  + \quad 10110100 \\
  \hline \\
  \text{10000000}
  \end{array}
  \]
- Since only the rightmost 8 bits count here, we can see that the sum of the two numbers is indeed 0
Problems solved?

- The second problem was with 0 itself; since 0 is always considered positive, the 8-bit representation of 0 is 00000000
- The 1’s complement of that value is 11111111
- Adding 1 to obtain the 2’s complement, we get: 1 0000 0000
- Again, since the carry doesn’t fit into 8 bits, it doesn’t count – so positive and negative 0 are the same value, eliminating the problem

2’s complement of a negative number

- If the scheme works correctly, then the 2’s complement of a negative value should be its positive equivalent
- For example, -93 is 1010 0011:
  - The 1’s complement is 0101 1100
  - Adding 1, we obtain: 0101 1101, which is 93

Range of 2’s complement numbers

- No matter how many bits are assigned to represent an integer, the largest integer that can be represented will be an odd number
- For example, in 8 bits, the largest positive value is 0111 1111, or 127
- The 2’s complement of this number, representing \(-127_{10}\), is 1000 0001
- Subtracting 1 from this value, we get the smallest negative number that can be represented in 8 bits: 1000 0000, or \(-128_{10}\)

Base conversions with signed numbers

- To convert a negative decimal number to binary:
  - Convert its absolute value to binary notation
  - Take the 2’s complement of the result
- Example: Convert \(-29_{10}\) to signed binary
  - The binary absolute value is: 0001 1101
  - The 1’s complement is: 1110 0010
  - The 2’s complement is: 1110 0011

Base conversions with signed numbers

- To convert a signed binary number to decimal:
  - If the sign bit is 0, convert as you would an unsigned integer
  - If the sign bit is 1, you have 2 choices, presented on the next 2 slides

Method 1: use 2’s complement

- Convert 1010 00112 to its decimal equivalent:
  - The 1’s complement is 0101 1100
  - The 2’s complement is 0101 1101
  - The decimal equivalent of the 2’s complement is 93, so the number is –93 (you can verify this by looking back at slide 8
Method 2: convert directly

- When you take the 2’s complement of a number, you start by taking the 1’s complement, that is, by inverting the bits.
- This means that the bits that are 0s in the negative number were 1s in the positive number, and contributed to its magnitude.
- To convert the binary negative to a decimal negative, take the sum of the place values for all the 0 digits in the 2’s complement version, then add 1.

Method 2 example

- Find the decimal equivalent of 10011001₂
  - The 0s are in the 6th, 5th, 2nd and 1st positions (in terms of powers of 2).
  - So we have 64 + 32 + 4 + 2 = 102
  - Adding 1 (and the negative sign), our final result is -103₁₀.

Binary numbers and the number line

- Suppose we had a computer system with a 3-bit byte; with such a system, we could represent the decimal values 0 through 7 as unsigned integers.
- Laid out on a number line, the binary values look like this:

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

We can think of addition as moving to the right from the first operand the number of spots specified by the second operand; for example, 2 + 5 would mean, starting from 2, move 5 spaces to 7.

If we attempt addition of 3 and 7, we fall off the end of the number line; this is the kind of error that causes the carry bit to be set when performing addition (or subtraction) on unsigned numbers.

Binary numbers and the number line

- If we wish to represent binary signed numbers, we use the same symbols as in the unsigned set, but we split the number line in half and move the right side to the left of the left side:

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

With the revised number line, addition of the same binary values produces the same results; for example, what was 2 + 5 when the values were considered unsigned will now be 2 + -3, but it will produce the same result as before which will now be interpreted as -1 instead of 7.

Addition of 2 (010) and 3 (011), which would have given us 5 (101) when the numbers were considered unsigned, will still give us the same result, but now it will be interpreted as -3.
Signed arithmetic & the carry bit

- The carry bit is not used to detect an error condition with signed arithmetic; the carry bit may be set by an operation, but that doesn’t necessarily indicate an error
- For example, with a 3-bit byte, the addition of -2 (110) and 3 (011) will produce 1001 (binary 1, the correct answer, and a 1 in the carry bit)

Integer overflow

- With 2’s complement notation, an arithmetic calculation can still result in a value that is out of range; this is what happened in the 2+3 example a couple of slides back, which produced a result of -3
- This type of error condition is flagged by setting the CPU’s overflow bit to 1
- The hardware detects an overflow by comparing the carry into the sign bit with the carry bit; if they don’t match, an overflow has occurred and the overflow bit is set

Bit-level operations

- We have already seen some examples of operations that occur at the bit level:
  - Addition and subtraction of binary numbers (both signed and unsigned)
  - Taking 1’s complement, also known as a NOT operation (inverting all the bits)
  - Taking 2’s complement, also known as negation
  - The NOT operation is logical; all the others described above are arithmetic operations

More logical bit-level operations

- At the ISA3 level, a 1 represents true and a 0 represents false (just like C++)
- As we have previously seen, the bit-level effect of a logical NOT (a unary operation) is to change all 1s to 0s and 0s to 1s
- Two other logical operations, AND and OR, are described on the next several slides

Logical binary operations

- At the bit level, AND and OR are binary in both senses of the word; they each take two operands, and, in this case, they are operating on binary data
- The truth tables below illustrate the effects of AND and OR operations

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p AND q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p OR q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Examples of logical operations

Suppose you have 2 binary values, 10011101 and 0111010

An AND operation on the two values:

```
  10011101
AND  0111010
--------
  00011000
```

An OR operation on the two values:

```
  10011101
OR   0111010
--------
11111111
```
Exclusive OR

- Another common logical operation is XOR (exclusive or)
- XOR differs from OR in that a 1-1 combination produces a 0 in XOR, unlike OR which produces a 1 in that case
- In other words, XOR produces a 1 if and only if the bit operands have different values

XOR example

An XOR operation on two binary values:

10011101
XOR 01110100
------------
11101111

A value XOR’d with itself produces only 0s
A value XOR’s with its negation produces only 1s

Arithmetic shift operations

- Unary operators ASL (arithmetic shift left) and
  ASR (arithmetic shift right) can be used to alter
  the value of binary numbers
- In a shift left, all the bits are shifted one place to
  the left; the most significant bit is shifted to the
  carry bit, and the least significant bit gets 0

ASL examples

00101111 shifted left becomes 01011110
11001101 shifted left becomes 10011010
10000000 shifted left becomes 00000000

The arithmetic effect of ASL is multiplication by 2; in the
first example, the value (47)_{10} was doubled; new value is 94_{10}
In the second instance, the number is -51; after ASL, the
value is -102_{10}
The third example illustrates an overflow condition; the
original value is -128, which is the minimum possible value in
8 bits. When this value is multiplied by 2, the value “rolls
over” to 0

Arithmetic Shift Right

- An ASR operation is almost the opposite of
  ASL:
  – Bits are shifted right instead of left
  – The rightmost (least significant) bit is shifted
    into the carry bit
  – ASR does not affect the sign bit

Other bit-level arithmetic
operations

- We have already seen binary addition; with 2’s
  complement notation, subtraction is just addition
  with a negative number
- Multiplication and division present more
  significant challenges
  – In real systems, special dedicated hardware is used
  – Operations may be carried out in parallel
- The next several slides illustrate a simple
  multiplication algorithm that could be used on the
  machine level
Bit-level multiplication algorithm

- We begin by writing the 2 operands (multiplier and multiplicand) to separate memory locations; a third location (initially set to 0) will store the results, both final and intermediate (partial products)
- Starting with the least significant bit of the of the multiplier, if the current bit of the multiplier is 1, we add the value of multiplicand to the partial product, then shift the multiplicand left
- We continue this process, moving one bit left through the multiplier at each step, until we run out of bits
- Note that the product requires double the working space of either of the operands

Example

Find the product of 6 and 11 using bit-level multiplication

Multiplicand (6) is 0000110
Multiplier (11) is 00001011
Four shifts will be required to find the product

Step 1: product is 00000000
Since rightmost bit of multiplier is 1, we add the multiplicand to the product; partial product result is now 00000110
ASL on multiplicand; value is now 00011000

Example – step 2

Starting values:
Multiplicand: 0001100
Multiplier: 00001011
Partial Product: 00001010
Going to digit: 2

Since the current multiplier digit is 1, we add the multiplicand value to the partial product, then we perform another ASL on the multiplicand

Ending values:
Multiplicand: 00011000
Multiplier: 00001011
Partial Product: 00010010
Going to digit: 3

Example – step 3

Starting values:
Multiplicand: 00011000
Multiplier: 00001011
Partial Product: 00010010
Going to digit: 3

We perform another ASL on the multiplicand, but since the current multiplier digit is 0, the partial product is unchanged at this step

Ending values:
Multiplicand: 00110000
Multiplier: 00001011
Partial Product: 00010010
Going to digit: 4

Example – final step

Starting values:
Multiplicand: 00110000
Multiplier: 00001011
Partial Product: 00010010
Going to digit: 4

Since the 4th (and last) digit of the multiplier is 1, we add the multiplicand’s value to the product, achieving the final result

Ending values:
Multiplicand: 00110000
Multiplier: 00001011
Final Product: 01000010 (or 66₁₀)

Division

- There are a couple of simple approaches to binary division:
  - Iterative subtraction of the denominator from the divisor
  - Trial-and-error long division (like what you learned in grade school)
- Division can easily cause a crash:
  - Division by 0
  - Division of operands whose magnitudes are enormously different: a divide underflow occurs when the divisor is much smaller than the dividend
Representation of alphanumeric data

- The most widespread binary code for the representation of character data is ASCII
  - Based on code used on teletype machines
  - Uses 7 bits to represent each character, plus a parity (error checking) bit
  - PCs use parity bit to extend the character set (extended ASCII)

More fun ASCII facts

- ASCII defines codes for:
  - 32 control characters
  - 10 digits
  - 52 letters (upper and lowercase alpha)
  - 32 “special” characters (such as $, #, and punctuation)
  - 1 space character
- ASCII stands for American Standard Code for Information Interchange
  - Aren’t you glad you know that?
  - OK, now you can forget it

For a readable version of this chart, see page 109 in your textbook

<table>
<thead>
<tr>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00</td>
<td>0031</td>
<td>11</td>
<td>0032</td>
<td>12</td>
<td>0033</td>
<td>13</td>
<td>0034</td>
<td>14</td>
</tr>
<tr>
<td>0035</td>
<td>15</td>
<td>0036</td>
<td>16</td>
<td>0037</td>
<td>17</td>
<td>0038</td>
<td>18</td>
<td>0039</td>
<td>19</td>
</tr>
<tr>
<td>003A</td>
<td>1A</td>
<td>003B</td>
<td>1B</td>
<td>003C</td>
<td>1C</td>
<td>003D</td>
<td>1D</td>
<td>003E</td>
<td>1E</td>
</tr>
<tr>
<td>003F</td>
<td>1F</td>
<td>0040</td>
<td>20</td>
<td>0041</td>
<td>21</td>
<td>0042</td>
<td>22</td>
<td>0043</td>
<td>23</td>
</tr>
<tr>
<td>0044</td>
<td>24</td>
<td>0045</td>
<td>25</td>
<td>0046</td>
<td>26</td>
<td>0047</td>
<td>27</td>
<td>0048</td>
<td>28</td>
</tr>
<tr>
<td>0049</td>
<td>29</td>
<td>004A</td>
<td>2A</td>
<td>004B</td>
<td>2B</td>
<td>004C</td>
<td>2C</td>
<td>004D</td>
<td>2D</td>
</tr>
<tr>
<td>004E</td>
<td>2E</td>
<td>004F</td>
<td>2F</td>
<td>0050</td>
<td>30</td>
<td>0051</td>
<td>31</td>
<td>0052</td>
<td>32</td>
</tr>
<tr>
<td>0053</td>
<td>33</td>
<td>0054</td>
<td>34</td>
<td>0055</td>
<td>35</td>
<td>0056</td>
<td>36</td>
<td>0057</td>
<td>37</td>
</tr>
<tr>
<td>0058</td>
<td>38</td>
<td>0059</td>
<td>39</td>
<td>005A</td>
<td>3A</td>
<td>005B</td>
<td>3B</td>
<td>005C</td>
<td>3C</td>
</tr>
<tr>
<td>005D</td>
<td>3D</td>
<td>005E</td>
<td>3E</td>
<td>005F</td>
<td>3F</td>
<td>0060</td>
<td>40</td>
<td>0061</td>
<td>41</td>
</tr>
<tr>
<td>0062</td>
<td>42</td>
<td>0063</td>
<td>43</td>
<td>0064</td>
<td>44</td>
<td>0065</td>
<td>45</td>
<td>0066</td>
<td>46</td>
</tr>
<tr>
<td>0067</td>
<td>47</td>
<td>0068</td>
<td>48</td>
<td>0069</td>
<td>49</td>
<td>006A</td>
<td>4A</td>
<td>006B</td>
<td>4B</td>
</tr>
<tr>
<td>006C</td>
<td>4C</td>
<td>006D</td>
<td>4D</td>
<td>006E</td>
<td>4E</td>
<td>006F</td>
<td>4F</td>
<td>0070</td>
<td>50</td>
</tr>
</tbody>
</table>

Other character codes

- Although ASCII is fairly ubiquitous, it is not the only character code in use today
- IBM mainframes use a rival 8-bit code called EBCDIC
- Java uses a 16-bit character code called Unicode

EBCDIC

- Stands for Extended Binary Coded Decimal Interchange Code
- Originally developed for IBM’s System/360 series
- Based on a 6-bit numeric coding system, Binary-Coded Decimal (BCD) which we’ll revisit tomorrow

Unicode

- Developed in the 1990s by a consortium of industry and government types to address the restrictions inherent in 8-bit codes like ASCII and EBCDIC
  - Both based on Latin alphabet; assumes English language (or something close to it)
  - Majority of the world’s population communicates in languages other than English
  - As more countries began using computers, a proliferation of incompatible language codes were being developed
Unicode

• Unicode provides single standard system with capacity to encode the majority of characters in every language in the world
• Uses 16-bit alphabet, downward compatible with ASCII
• Defines an extension mechanism that allows for coding of an additional million characters
• Default character set for Java programs