**Class 1**

**Syllabus**

See slides

**Slide set A-1**

Main components of computer system: Anything missing from the diagram?

Smallest unit of storage on computing systems: bit

Abstract the value as 0 or 1, but stored in a different way (we don’t need to worry about exactly how it’s stored)

Bit string. For example, 0011 or 1100: Are they different? If so, why?

What kinds of data are stored as bit strings in computers?

How are bits trings interpreted? In other words, how do we get a value or meaning out of the bit string?

Encoding: A method of storing value or meaning in a bit string.

Hardware has circuits for getting the meaning out for certain types of data which we’re familiar with (but we’ll say more about them): Integers, Floats, Characters, Booleans.

Primitive (“Built-in” types); Every other type requires software to get the meaning or value out.

How is memory in a system organized? [Note: There are various types of memory, but they’re all organized in the same way fundamentally, as we’ll see.]

Are the definitions of these universal (The same in all systems)?

Byte: What is it?

Word: What is it?

**Integers**

Unsigned: How are they stored? B2U

What ***range of values*** (smallest to largest) can we store in an unsigned number bit string? Know the formula for the maximum number, based on the number of bits.

**Signed integers:** Three different methods have been used in various ways.

1. Binary-to-Sign and Magnitude (B2S): Understand how it works; Simple and intuitive but horrible in machines. [More discussion later]
2. Two’s Complement (B2T): Understand how it works; Not as simple or intuitive, but works optimally in machines. [More discussion later]
3. 1’s Complement (B2O): Understand how it works; More intuitive than Two’s Complement but does not work as well in machines (See the disadvantages in the slides). [More discussion later]

**Arithmetic Operations**

**Addition**

Unsigned integers: Binary numbers are added by processors using the same algorithm which we use for decimal, but the base is different, so that makes it seem a little different:

|  |  |  |  |
| --- | --- | --- | --- |
| Bit 1 | Bit 2 | Sum | Carry |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

**Note:** When the sum to be calculated is 1 + 1, the sum in decimal is 2, of course, but 2 cannot be stored in one digit (bit) in binary; it requires 2 bits (2 is 10 in binary), so this is why the carry is 1.

Also, we will see later that the processor can only add two bits at a time, so to add two numbers, the processor adds the two bits from the numbers to get a sum, and then adds the carry bit to the first sum to get the final sum of the three bits (we usually shorten this by adding all three bits at once, and if all three bits are 1, then the sum is 3 in decimal, but 11 in binary, so the sum is 1 with a carry of 1). The carry bit which is produced is passed through to the next column of bits to be added. The last carry produced from the addition of the last pair of bits (the most significant bits, or msbs) is also stored by the processor. This allows for the determination of overflow (an incorrect result).

Be sure to understand the points about real CPUs on slide 28 (Please ask if you don’t!)

See explanation of the carry flag (C flag) on slide 29. Every processor has this kind of flag.

How do we know there is overflow (incorrect result) from the C flag?

**Example**

Let’s add these two 4-bit unsigned binary numbers:

0110 (decimal 6)

0111 (decimal 7)

What is the 4-bit sum?

Is there overflow?

**Another example**

Let’s add these two 4-bit unsigned binary numbers:

1011 (decimal 11)

0111 (decimal 7)

What is the 4-bit sum?

Is there overflow?

**More on Signed Integers**

**2’s complement (B2T)**

Non-negative integers: The msb is 0 (These have the same value as a B2U number with the same bits)

Negative integers: The msb is 1

How can we get the value of a negative B2T integer? We need 2 steps:

1. Invert (“flip”) each bit
2. Add 1

The original number is the negation of the non-negative (B2U) number we get.

We negate a B2T number (whether it’s non-negative or negative) by following the 2 steps above also.

**1’s complement (B2O)**

Non-negative integers: The msb is 0 (These have the same value as a B2U number with the same bits)

Negative integers: The msb is 1

How can we get the value of a negative B2O integer? We need only 1 step:

Invert (“flip”) each bit

The original number is the negation of the non-negative (B2U) number we get.

We negate a B2O number (whether it’s non-negative or negative) by doing the 1 step above also.

Disadvantages of 1’s Complement (B2O): Even though it looks simpler than 2’s complement, B2O makes the hardware for doing arithmetic more complex.

**Adding B2O Numbers**

If we add 2 non-negative numbers, we can do it the same way as B2U numbers.

If we add 2 negative numbers, though, the result is always 1 less than the correct result if we use a 1st carry of 0 (as we do for B2U)

So instead, we can use a first carry of 1, and this will adjust the result to the correct result, if the 2 numbers being added are both negative.

To get the correct 1st carry, we can feed the two msbs from the operands to an AND gate, and use the output of this gate as the 1st carry.

**PROBLEM:** This does not always work if the 2 operands have different signs! When we add 2 B2O numbers with different signs, with a 1st carry of 0, sometimes the result is correct, but sometimes NOT CORRECT.

**HOW CAN THE ADDER DETERMINE IF THE SIGNS ARE DIFFERENT?**

**An EXCLUSIVE-OR (XOR) gate can be used; XOR outputs a 1 only if the two input bits are different (exactly 1 input bit is 1). XOR outputs:**

|  |  |  |
| --- | --- | --- |
| **Bit1 input** | **Bit2 input** | **Output** |
| **0** | **0** | **0** |
| **0** | **1** | **1** |
| **1** | **0** | **1** |
| **1** | **1** | **0** |

**SOLUTION:** When we add 2 B2O numbers with different signs (using a 1st carry of 0), if the result is incorrect, it will always be 1 less than the correct result, but the last carry will be 1, so we can design the adder to do A SECOND ADDITION and add the last carry back to the first sum to get a final sum, which will always be correct!

Although the solution above works, it makes the adder more complicated, and it means addition will be slower when the operands have different signs; this is a significant disadvantage, because addition is a very common operation in programs.

**CONCLUSION:** Although using B2O for signed numbers can be made to work correctly, it increases complexity and reduces performance, so B2O is not used for signed integers anymore (all modern machines use B2T instead).

**Example B2O Addition**

Let’s add these two 4-bit B2O binary numbers (Remember that a 2nd addition is needed, since the numbers have different signs):

0010 (decimal 2)

1101 (decimal -2)

After the 2nd addition of the last carry, what is the final 4-bit sum?

Is the sum correct?

**Another Example B2O Addition**

Let’s add these two 4-bit B2O binary numbers (Remember that a 2nd addition is needed, since the numbers have different signs):

0100 (decimal 4)

1101 (decimal -2)

After the 2nd addition of the last carry, what is the final 4-bit sum?

Is the sum correct?

**B2S HAS SIMILAR DISADVANTAGES TO B2O**, but even worse; it makes hardware extremely complicated to get correct results, so it has never been used as a way to represent signed integers in real machines.

**SIGNIFICANT ADVANTAGE of B2T (2’s Complement):** The same hardware can be used to do operations on unsigned or signed integers (the only difference is how overflow is detected; see below).

**Example B2T Addition**

Let’s add these two 4-bit B2T binary numbers (Remember we do the addition the same way we do for B2U (unsigned) numbers):

0010 (decimal 2)

1101 (decimal -3)

What is the final 4-bit sum?

Is the sum correct?

**OVERFLOW FOR B2T:** Instead of using only the last carry bit, the last 2 carry bits must be compared (using an XOR gate, as described above); if they are the same, there’s no overflow, and the result is correct, but if they are different, there is overflow, and the result is incorrect. The output bit of the XOR gate used for overflow is written to a different flag from the C flag, called **the O (Overflow) flag.**

**Another Example B2T Addition**

Let’s add these two 4-bit B2T binary numbers:

1100 (decimal -4)

1011 (decimal -5)

What is the final 4-bit sum?

What will the O flag be?