COMP 2280 - Introduction to Computer Systems

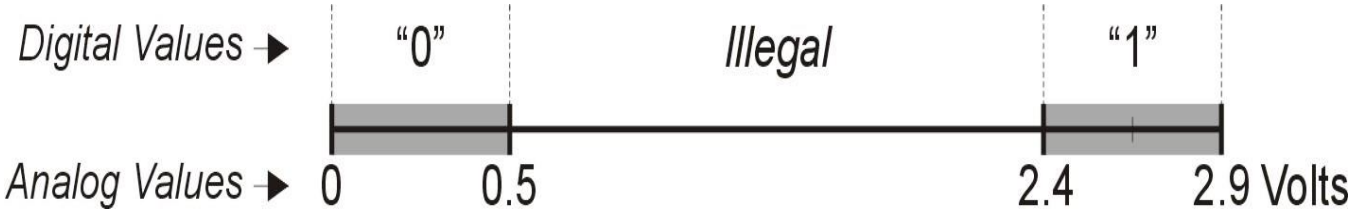
Module 2- Data Representation

Introduction

- **How do we represent data in a computer?**
  - At the lowest level, a computer is an electronic machine.
  - It works by controlling the flow of electrons
  - Easy to recognize two conditions:
    - presence of a voltage – we'll call this state "1"
    - absence of a voltage – we'll call this state "0"
  - A machine could base state on *value* of voltage, but control and detection circuits more complex.
  - It's much easier to build a machine that operates based on the presence or absence of voltage.

Introduction

- A Computer is a binary digital system
- A digital system is based on:
  - finite number of symbols. Usually binary (base two) system with the values 0 and 1.
- Computers don't actually work on the absence or presence of voltage, but instead works on a range of voltage.



Introduction

- The basic unit of information in a computer is a *bit*.
  - The value 0 or 1 can be stored in a bit.
- A bit by itself is not very useful.
- Bits are usually grouped together so that they can be used to represent data.
  - Eg. 8 bits can be group together to form a *byte*, which can store 2^8 different patterns.
- In general, a sequence of k bits can represent one of 2^k distinct k-bit patterns.
- There are lots of varying types of data that a computer needs to store and operate on.

Introduction

- These include text, numbers, images, sound, videos, instructions, etc.
- Data must be encoded in a format that can be operated on by the computer.
- Let us begin by examining the various ways to encode whole numbers (integers).
- Notation: We use subscripts to denote the base of a number. 10_2 means 10 binary, whereas 10_{10} means 10 decimal.
- Notation: When we have a sequence of bits, the highest order bit is called the *most significant bit* (msb) and the lowest order by is called the *least significant bit* (lsb).
- Eg) 10110 – bit 4 is the msb, bit 0 is the lsb.

Unsigned Numbers - Representation

- *Unsigned numbers* are non-negative numbers.
- Typically the size (# of bits) is fixed
  - unsigned int - 32 bits (machine dependent)
    - 64 bit becoming common, sometimes called longs
  - unsigned short - 16 bits
  - unsigned char - 8 bits
- The encoding for unsigned numbers works as follows:
  - The i^{th} bit represents the value 2^i_{10}
  - Therefore if 3 bits were used, 000,001,010,011,100,101,110,111 would represent the numbers 0,1,2,3,4,5,6,7 respectively.

Unsigned Numbers - Representation

Eg) Assume 4-bit numbers.

Then $10_{10} = 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$.

$8 + 0 + 2 + 0$

Therefore $10_{10} = 1010_2$.

- From the example above, it's easy to encode and decode an unsigned number.
- Given n bits, it can encode the numbers in the range between 0 to 2^n-1.

Unsigned Numbers - Arithmetic

- **Unsigned Binary Arithmetic**
- Base-2 addition – just like base-10!
- add from right to left, propagating carry
- Note: The # of bits are fixed for each example (5 bits for first 2 examples, 4 bits for last example)

```
 10010        10010        1111
+1001       +  1011      +   1
 11011        11101       10000
```

*carry on 2nd bit

*carry on all bits

try:

```
 10111
+  111
```

- We shall say more about arithmetic operations after we talk about signed numbers.

Signed Numbers - Representation

- Let us suppose we have n bits to work with.
- There are several established methods for representing signed numbers:
  - Sign-magnitude
  - 1's complement
  - 2's complement
- Computers these days use 2's complement to represent signed numbers, but we shall take a look at all three.

Sign Magnitude Representation

- Use the first (leftmost) bit for a sign (0 for +, 1 for -)
- Use the remaining n-1 bits for the magnitude (absolute value), exactly like you would for unsigned numbers.
- With n=4, -5 is encoded as 1101 while 5 is encoded as 0101.
- For n bits, $-(2^{(n-1)} - 1)$ to $+(2^{(n-1)} - 1)$ can be represented.
  - Eg. For n=4, the numbers between -7 to 7 can be represented.
- Simple, but arithmetic circuits become awkward (see below).
- Two different zeroes are created (+0 is 000..00 and -0 is 100..00)
- Arithmetic is awkward
  - add 0101 with 1110.
  - Since signs are different, determine the larger whose sign will be kept for answer, subtract smaller from larger.
  - Not pretty and makes circuits awkward.

One's Complement Representation

- Positive numbers begin with a 0, and use the same encoding as for unsigned numbers.
- Negative numbers use the complement (invert all bits) of the representation for the corresponding positive number.
- Eg. Suppose n=4, then 5 is encoded as 0101, while -5 is encoded as 1010.
- For n bits, $-(2^{(n-1)}-1)$ to $+(2^{(n-1)}-1)$ can be represented
- Two different zeroes are created (+0 is 000..00 and -0 is 111..11).
- Simple but has drawbacks when used in arithmetic operations.

Two's Complement Representation

- Non-negative number has msb=0, and encoded as using unsigned number encoding.
  (msb = most significant bit)
- Negative numbers use the "two's complement" of the representation for the corresponding positive number
  - To "two's complement" something:
    - Invert every bit and then add 1 to the result.
- The "two's complement" operation is self-inverting. Applied to any number (positive or negative) it will negate it (exception: $-2^{(n-1)}$ cannot be negated, as there is no $+2^{(n-1)}$).
- The first bit is a sign bit (0 for +, 1 for -)
- Using n bits $-(2^{(n-1)})$ to $+(2^{(n-1)}-1)$ can be represented

Two's Complement Examples

- Assume n=5.
- Numbers that can be represented are between -16 to 15
- Eg) Give the 2's complement n-bit representation for 12.
  - As 12 is non-negative, the encoding is just 01100.
- Eg) Give 2's complement n-bit representation for -12.
  - Take 2's complement of 12 by first flipping all bits to get 10011, and then adding 1 to get 10100.
- Eg) Applying the 2's complement operation to 10100, we get 01011 + 00001 = 01100, which is just 12. This tells you that 10100 encodes the number -12.
- Useful: When given a negative number in 2's complement encoding. You can figure out what that number is by applying the 2's complement operation to it. This gives the absolute value of the number in question. This is exactly what the last example illustrates.
- Useful: In 2's complement representation, a number is negative if and only if the msb is 1

Two's Complement Benefits

- Allow logic circuits to be kept as simple as possible for doing arithmetic.
- All arithmetic & logic operations are performed by the ALU.
- Addition of binary numbers is done the same way as addition for decimal (base 10) numbers.
- 2's complement representation is nice in that
  - Adding A and –A always gives 0….0
  - The representation of (A +1) = Representation of (A) + Representation of (1)
- These properties ensure that addition is done correctly when using 2's complement representation for signed numbers.
- We will consider addition and subtraction of numbers represented using two's complement later.

Converting Binary (2's Complement) to Decimal

- If leading bit is one, take two's complement to get a positive number.
- Add powers of 2 that have "1" in the corresponding bit positions.
- If original number was negative, add a minus sign.

Examples (8 bit 2's compliment number):

X = 01101000_2

= $2^6+2^5+2^3 = 64+32+8$

= 104_{10}

X = 00100111_2

= $2^5+2^2+2^1+2^0 = 32+4+2+1$

= 39_{10}

X = 11100110_2

-X = 00011010

= $2^4+2^3+2^1 = 16+8+2$

= 26_{10}

X = -26_{10}

Converting Decimal to Binary (2's Complement)

- Find magnitude of the decimal number. (Always positive.)
- Divide by two – remainder is least significant bit.
- Keep dividing by two until answer is zero, writing remainders from right to left.
- Append a zero as the msb. Finally, if the original number was negative, take two's complement.

Example (assume n=8 bits):

X =	104_{10}	104/2	=	52 r0	*bit 0*
		52/2	=	26 r0	*bit 1*
		26/2	=	13 r0	*bit 2*
		13/2	=	6 r1	*bit 3*
		6/2	=	3 r0	*bit 4*
		3/2	=	1 r1	*bit 5*
X =	01101000_2	1/2	=	0 r1	*bit 6*

Operations

- We now consider arithmetic and logic operations on signed integers.
- We assume numbers are encoded using 2's complement representation.

Addition

- As we've mentioned earlier, 2's complement addition is just binary addition.
- Assume all numbers have the same number of bits (n=8 in our examples)
- Ignore carry out.
- For now, assume that sum fits in n-bit 2's comp. representation.

```
  01101000  (104)
+ 11110000  (-16)
 (1)01011000 (88)
```

```
  11110110 (-10)
+ 11110111  (-9)
 (1)11101101  (-19)
```

Subtraction

- Negate subtrahend (2nd no.) and add.
- assume all integers have the same number of bits
- ignore carry out
- for now, assume that difference fits in n-bit 2's complement representation

```
  01101000   (104)        11110110   (-10)
- 00010000   (16)       - 11110111   (-9)
  01101000   (104)        11110110   (-10)
+ 11110000   (-16)     + 00001001   (9)
 (1)01011000 (88)        (0)11111111 (-1)
```

Sign Extension

- To add two numbers, we must represent them using the same number of bits.
- If we just pad with zeroes on the left:

4-bit		8-bit	
0100	(4)	00000100	(still 4)
1100	(-4)	00001100	(12, not -4)

- Instead, replicate the msb -- the sign bit:

4-bit		8-bit	
0100	(4)	00000100	(still 4)
1100	(-4)	11111100	(still -4)

Overflow

- If operands are too big, then sum cannot be represented as an *n*-bit 2's comp number.

```
    01000   (8)          11000   (-8)
+   01001   (9)      +   10111   (-9)
    10001   (-15)        01111   (+15)
```

- We have overflow if:
  - signs of both operands are the same, and sign of sum is different.
  - Can be used as a test to detect overflow.
- Another test -- easy for hardware:
  - The carry bit into msb does not equal carry out bit.
  - In the first example, the carry in is 1 but the carry out is 0.
  - In the 2nd example, the carry in is 0 but the carry out is 1.

Logical Operations

- Operations on logical TRUE or FALSE.
- two states -- takes one bit to represent: TRUE=1, FALSE=0.
- Recall how AND, OR and NOT works from comp2130.
- View *n*-bit number as a collection of *n* logical values
- operation applied to each bit independently

AND Operator

- Useful for clearing bits.
- AND with zero = 0
- AND with one = no change
- Eg.

```
        11000101
AND     00001111
        --------
        00000101
```

- In the C programming language, the operator & denotes the AND operator.

OR Operator

- Useful for setting bits
- OR with zero = no change
- OR with one = 1
- Eg.

```
          11000101
OR        00001111
          --------
          11001111
```

- In C, the operator | (pipe) is the OR operator.

Not Operator

- unary operation -- one argument
- flips every bit
- Eg.

NOT 11000101

00111010

- In C, the ~ operator is the NOT operator.

Hexadecimal (Hex) Notation

- It is often useful to write numbers in hexadecimal (base 16) form.
- Each digit is one of 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.
- Letters are case insensitive.
- One reason for this notation is that each hex symbol encodes exactly 4 bits.
- It's also less error prone to write out 2 hex digits 0xFE than it is to write out 8 binary bits 11111110.
- Notation: hex numbers are often preceded with 0x or subscripted with 16.
- Eg) 0xFE6FAE or $FE6FAE_{16}$

Converting From Binary To Hexadecimal

- Every four bits is a hex digit.
- start grouping from right-hand side
- E.g.

0011 1010 1000 1111 0100 1101 0111

3 A 8 F 4 D 7

- Note:*This is not a new machine representation, just a convenient way to write a number.*
- To convert from hexadecimal to binary, just reverse the process.
- Eg. 0x3F2E represents the 16 bits 0011 1111 0010 1110.

Dealing with Decimal Points

- Sometimes we have to work with numbers that have decimal points.
- Eg) The number 0.25
- We can represent/encode 0.25 using binary digits if we think of $0.25 = 0x2^{-1} + 1x2^{-2}$.
- So $0.25 = 0.01_2$.
- In general, any decimal number with a fixed # of digits can be represented this way.
- Eg) $5/8 = 1/2 + 0/4 + 1/8 = 0.101_2$.

Dealing with Decimal Points

- You can probably see how we can do this in general.
- Given a decimal part D of a number,
  - Multiply it by 2 and take the digit to left of decimal point as the bit to append to binary number.
  - Remove this bit from the product, and repeat the previous step, until the part to the right of decimal point is 0.
- Sometimes this process repeats indefinitely. In such cases, just stop when we have reached the required # of bits.

Dealing with Decimal Points

- Eg) Convert 0.125 to binary representation
  1. 0.125 x 2 = 0.25, so take the bit 0 (msb of fractional part)
  2. 0.25 x 2 = 0.5, so take bit 0
  3. 0.5 x 2 = 1.0 so take bit 1
- Therefore 0.125 = $\mathbf{0.001_2}$.

- Eg) Convert 0.8 to binary representation
  1. 0.8 x 2 = 1.6, so take bit 1 (msb of fractional part)
  2. 0.6 x 2 = 1.2, so take bit 1
  3. 0.2 x 2 = 0.4, so take bit 0
  4. 0.4 x 2 = 0. 8, so take bit 0.
  5. Now, we see a repeated pattern, as we run into 0.8 again.
- Therefore 0.8 = $\mathbf{0.11001100\ldots_2}$, where the number of bits is fixed.

IEEE Floating Point Representation

- Floating point (real) numbers can be represented in a computer.
- Most computers use IEEE-754 encoding.
- Encoding (single-precision numbers) consists of:
  - 1 sign bit (0 if non-negative, 1 if negative)
  - 8 exponent bits (exponent + 127 as unsigned number)
  - 23 fraction bits (for significant digits)

IEEE Floating Point Representation

- The number being represented is first normalized so that exactly 1 non-zero binary digit appears left of the decimal point.
- The layout of the bits is given as:

IEEE Floating Point Representation

- The number needs to be converted to binary representation first, then normalized.
- The bits stored in the exponent field is the value of the exponent + 127. This is encoded as an unsigned number.
- The sign bit is 1 if and only if the number is negative.
- Lets do some examples.

IEEE Floating Point Representation

- Eg) Encode -65.2 in IEEE format.
- $65 = 1000001_2$ $(65 = 64 + 1 = 2^6 + 2^0)$
- 0.2
  - 0.2 x 2 = 0.4
  - 0.4 x 2 = 0.8
  - 0.8 x 2 = 1.6
  - 0.6 x 2 = 1.2
  - and now the pattern repeats
  - So 0.2 = 0.00110011…
- So, 65.2 = 1000001.00110011…
- Normalizing, we get
- $65.2 = 1.00000100110011001100110 \times 2^6$.
- So the exponent is 6, and therefore the exponent field will contain the value 127+6 = 133, which is 10000101.
- The sign bit is 1.
- Therefore, the IEEE encoding of -65.2 is :
  - **1 10000101 00000100110011001100110**

IEEE Floating Point Representation

- Eg) What number is represented by $C2328000_{16}$ when using IEEE floating point encoding?
- Note: C2328000 = 1 10000100 01100101000000000000000$_2$.
- Sign bit = 1
- Exponent is 132 – 127 = 5.
- So the number is -1.01100101000000000000000$_2$ x 2^5.
- Therefore the number is -101100.101$_2$ = **-44.625**.

ASCII Codes

- ASCII: Maps 128 characters to 7-bit code.
- both printable and non-printable (ESC, DEL, …) characters
- See back page of text or type "man ascii" in UNIX for an ASCII table.
- What is relationship between a decimal digit ('0', '1', …,'9') and its ASCII code? 30h, 31h, …, 39h
- What is the difference between an upper-case letter ('A', 'B', …) and its lower-case equivalent ('a', 'b', …)? +20h
- Given two ASCII characters, how do we tell which comes first in alphabetical order? value compare...
- Are 128 characters enough? Unicode using 16 bits to represent a character. See http://www.unicode.org/
- *No new operations -- integer arithmetic and logic.*

LC-3 Data Types

- Which data types are supported directly by the LC-3 instruction set architecture?
- For LC-3, there is only one hardware-supported data type:
  - 16-bit 2's complement signed integer
  - No support for floating point numbers
- Operations: ADD, AND, NOT
  - Notice OR, Subtract, XOR, and other operations are not provided by LC-3.
  - Of course, you can write software to implement them yourself.
- Other data types are supported by interpreting 16-bit values as logical, text, fixed-point, etc., in the software that we write.

Exercises

- Determine the decimal values for the following 16-bit unsigned numbers.
  - 1001001110111100_2
  - $A25F_{16}$
  - $F0F0_{16}$
- Determine the decimal value of the following 2's complement 8-bit signed numbers :
  - FE
  - 7D
- Give the IEEE representation for -4.5.
- What floating point number is encoded by the hex digits 42220000 using IEEE representation?