Binary Numbers

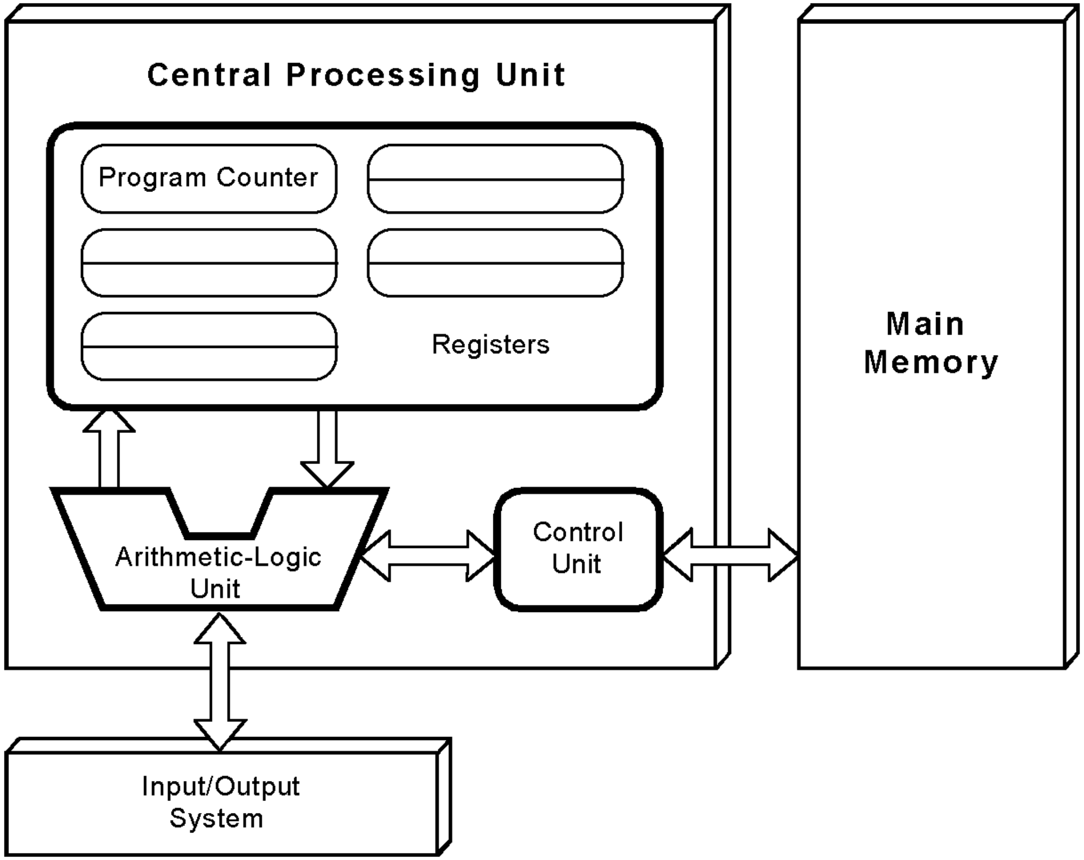
Homework #1

- **Assigned today!**
  - http://ecs-network.serv.pacific.edu/ecpe-170
- **Due Next Class Period (i.e. Wednesday)**
  - Class design: Smaller but more frequent assignments
- **Topics**
  - Number conversion, 50-word sentence problem
- Turn in homework via **Sakai**
  - Either an attachment or inline on the web form

Upcoming Classes

- **Monday**
  - No class – MLK day
- **Wednesday**
  - Floating-point numbers
  - Floating-point errors
  - Range, precision, and accuracy
- **Friday**
  - Characters
  - **Homework #2 Assigned**

Recap - von Neumann Model



- **How does this run a stored program?**
- **What is the *von Neumann Bottleneck*?**

Converting Between Bases

Converting Between Bases

- The following methods work for converting between *arbitrary* bases
  - We'll focus on converting to/from **binary** because it is the basis for digital computer systems
- Two methods for radix conversion
  - Subtraction method
    - Easy to follow but tedious!
  - Division remainder method
    - Much faster

Subtraction Method: Decimal to Binary

2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024
2^{11}	2048

Convert 789_{10} to binary (base 2)

Largest number that fits in 789? (512)	789 – 512 = 277	`1xxxxxxxxx`
Does 256 fit in 277? (yes)	277 – 256 = 21	`11xxxxxxxx`
Does 128 fit in 21? (no)	21	`110xxxxxxx`
Does 64 fit in 21? (no)	21	`1100xxxxxx`
Does 32 fit in 21? (no)	21	`11000xxxxx`
Does 16 fit in 21? (yes)	21 – 16 = 5	`110001xxxx`
Does 8 fit in 5? (no)	5	`1100010xxx`
Does 4 fit in 5? (yes)	5-4 = 1	`11000101xx`
Does 2 fit in 1? (no)	1	`110001010x`
Does 1 fit in 1? (yes)	1-1=**0**	**`1100010101`**

Division Method: Decimal to Binary

Convert 789_{10} to binary

789 / 2 = 394.5	Remainder of 1
394 / 2 = 197	Remainder of 0
197 / 2 = 98.5	Remainder of 1
98 / 2 = 49	Remainder of 0
49 / 2 = 24.5	Remainder of 1
24 / 2 = 12	Remainder of 0
12 / 2 = 6	Remainder of 0
6 / 2 = 3	Remainder of 0
3 / 2 = 1.5	Remainder of 1
1 / 2 = 0.5 (stop when <1)	Remainder of 1

Divide by 2 since we're converting to binary (base 2)

Read bottom to top:
$789_{10} = 1100010101_2$

Binary to Decimal

2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024
2^{11}	2048

Convert 1011000100_2 to decimal

$= 1x2^9 + 0x2^8 + 1x2^7 + 1x2^6 + 0x2^5 + 0x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 0x2^0$

= 512 + 128 + 64 + 4

= **708**

Binary to Decimal (Faster!)

Convert 1011000100_2 to decimal

1011000100$_2$	0*2 + 1 = 1
1**0**11000100$_2$	1*2 + 0 = 2
10**1**1000100$_2$	2*2 + 1 = 5
101**1**000100$_2$	5*2 + 1 = 11
1011**0**00100$_2$	11*2 + 0 = 22
10110**0**0100$_2$	22*2 + 0 = 44
101100**0**100$_2$	44*2 + 0 = 88
1011000**1**00$_2$	88*2 + 1 = 177
10110001**0**0$_2$	177*2 + 0 = 354
101100010**0**$_2$	354*2 + 0 = 708

Double your current total and add new digit

Range

- **What is the smallest and largest 8-bit unsigned binary number?**
  - $XXXXXXXX_2$
  - Smallest = 00000000_2 = **0**
  - Largest = 11111111_2 = **255**

Converting Between Bases

- What about **fractional values?**
  - Fractional values can be **approximated** in all base systems
  - No guarantee of finding an exact representations under all radices
- Example of an “impossible” fraction:
  - The quantity ½ is exactly representable in the binary and decimal systems, but is not in the ternary (base 3) numbering system

Converting Between Bases

- Fractional values are shown via nonzero digits to the right of the decimal point ("radix point")
  - These represent negative powers of the radix:

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$

$$\begin{aligned} 0.11_{2} &= 1 \times 2^{-1} + 1 \times 2^{-2} \\ &= \tfrac{1}{2} + \tfrac{1}{4} \\ &= 0.5 + 0.25 = 0.75 \end{aligned}$$

Subtraction Method: Decimal to Binary

Convert 0.8125_{10} to binary

2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625
2^{-5}	0.03125
2^{-6}	0.015625

Does 0.5 fit in 0.8125? (yes)	0.8125-0.5 = 0.3125	`.1`
Does 0.25 fit in 0.3125? (yes)	0.3125-0.25 = 0.0625	`.11`
Does 0.125 fit in 0.0625? (no)	0.0625	`.110`
Does 0.0625 fit in 0.0625? (yes)	0.0625-0.0625 = **0**	**`.1101`**

Stop when you reach 0 fractional parts remaining
(**or you have enough binary digits**)

Multiplication Method: Decimal to Binary

Convert 0.8125_{10} to binary

0.8125 * 2 = 1.625	1 (whole number)
0.625 * 2 = 1.25	1
0.25 * 2 = 0.5	0 (no whole number)
0.5 * 2 = 1.0	1

Stop when you reach 0 fractional parts remaining (or you have enough binary digits)

Read top to bottom:
$0.8125_{10} = .1101_2$

Hexadecimal Numbers

- Computers work in binary internally
- Drawback for humans?
  - Hard to read long strings of numbers!
  - Example: $11010100011011_2 = 13595_{10}$
- For compactness and ease of reading, binary values are usually expressed using the **hexadecimal** (base-16) numbering system

Hexadecimal Numbers

A=10
B=11
C=12
D=13
E=14
F=15

- The hexadecimal numbering system uses the numerals 0 through 9 and the letters A through F
  - The decimal number 12 is C_{16}
  - The decimal number 26 is $1A_{16}$
- It is easy to convert between base 16 and base 2, because $16 = 2^4$
- To convert from binary to hexadecimal, group the binary digits into sets of four

Converting Between Bases

- Using groups of 4 bits, the binary number `11010100011011`$_2$ (13595_{10}) in hexadecimal is:

0011	0101	0001	1011
3	5	1	B

Careful!

If the number of bits is not a multiple of 4, pad on the left with zeros.

Thus, safest to start at the right and work towards the left!

Signed Integers

Signed Integer Representation

- To date we have only examined unsigned numbers
- Used in a variety of programs and system functions
  - Memory addresses are always unsigned
  - Hard drive block addresses are always unsigned
- But some (*picky*) programmers wanted to represent negative numbers too!
- **Ideas on how we might do this?**

Signed Integer Representation

- To represent signed integers, computer systems use the high-order bit to indicate the sign
  - `0xxxxxxxx` = Positive number
  - `1xxxxxxxx` = Negative number

Value of the number

High order bit /
Most significant bit

- **What have we given up compared to unsigned numbers?**
  - **Range**! With the same number of bits, unsigned integers can express twice as many "positive" values as signed numbers
- Design challenge – How to interpret the *value* field?

Signed Integer Representation

- There are three ways in which signed binary integers may be expressed:
  - Signed magnitude
  - One's complement
  - Two's complement
- In an 8-bit word, *signed magnitude* representation places the **absolute value** of the number in the 7 bits to the right of the sign bit.

Signed Integer Representation

- Examples of 8-bit *signed magnitude* representation:
  - +3 = 00000011
  - -3 = 10000011

  Sign Bit | Magnitude

What if I wanted 16-bit *signed magnitude* representation?

- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
  - Ignore the signs of the operands while performing a calculation
  - Apply the appropriate sign after calculation is complete

Signed Integer Representation

- Example: using 8-bit *signed magnitude* binary arithmetic, find 75 + 46
- Convert 75 and 46 to binary
- Arrange as a sum, but separate the (positive) sign bits from the magnitude bits

```
0   1 0 0 1 0 1 1
0 + 0 1 0 1 1 1 0
  _______________
```

Signed Integer Representation

- Example: using 8-bit *signed magnitude* binary arithmetic, find 75 + 46
- Just as in decimal arithmetic, we find the sum starting with the rightmost bit and work left.

```
0    1001011
0 +  0101110
  ----------
           1
```

Signed Integer Representation

- Example: using 8-bit *signed magnitude* binary arithmetic, find 75 + 46
- In the second bit, we have a carry, so we note it above the third bit.

```
              1
0     1 0 0 1 0 1 1
0 +   0 1 0 1 1 1 0
  -----------------
                0 1
```

Signed Integer Representation

- Example: using 8-bit *signed magnitude* binary arithmetic, find 75 + 46

- The third and fourth bits also give us carries.

```
          1 1 1
0     1 0 0 1 0 1 1
0 +   0 1 0 1 1 1 0
   ----------------
            1 0 0 1
```

Signed Integer Representation

- Example: using 8-bit *signed magnitude* binary arithmetic, find 75 + 46

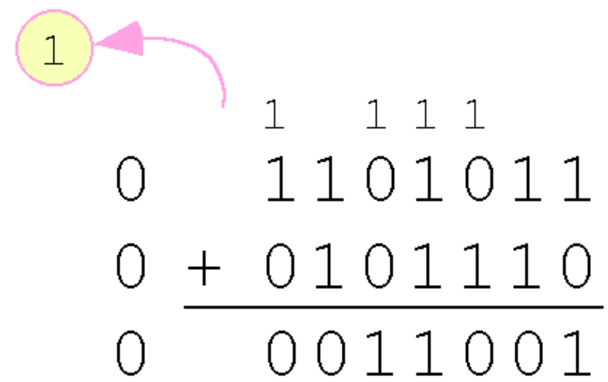
```
           1 1 1
0    1 0 0 1 0 1 1
0  + 0 1 0 1 1 1 0
   ---------------
0    1 1 1 1 0 0 1
```

- Once we have worked our way through all eight bits, we are done.

In this example, I picked two values whose sum would fit into 7 bits (leaving the 8th bit for the sign). If the sum *doesn't* fit into 7 bits, we have a problem.

Signed Integer Representation

- Example: using 8-bit *signed magnitude* binary arithmetic, find 107 + 46.
- The carry from the seventh bit **overflows** and is discarded – no room to store it!
- We get an erroneous result: 107 + 46 = ~~25~~.



No magic solution to this overflow problem – you need more bits! (or a smaller number)

Signed Integer Representation

- How do I know what sign to apply to the *signed magnitude* result?
  - Works just like the signs in pencil and paper arithmetic
- **Addition rules**
  - If the **signs are the same**, just add the absolute values together and use the **same sign** for the result
  - If the **signs are different**, use the sign of the **larger number**. Subtract the larger number from the smaller

```
        1 1
1   0101110
1 + 0011001
  ---------
1   1000111
```

- Example: Using *signed magnitude* binary arithmetic, find -46 + -25.
- Because the signs are the same, all we do is add the numbers and supply the negative sign when finished

Signed Integer Representation

- Mixed sign addition (aka **subtraction**) is done the same way
  - Example: Using signed magnitude binary arithmetic, find 46 + -25.
- The sign of the result is the sign of the larger (here: +)
  - Note the “borrows” from the second and sixth bits.

			0	2			0	2
0		0	~~1~~	0	1	1	~~1~~	0
1	+	0	0	1	1	0	0	1
0		0	0	1	0	1	0	1

Signed Integer Representation

- Strengths
  - *Signed magnitude* is easy for people to understand
  - **You'll find that, in low-level computer design, "easy for people to understand" doesn't count for very much!**
- Drawbacks
  - Makes computer **hardware** more **complicated** / slower
    - Have to compare the two numbers first to determine the correct sign and whether to add or subtract
  - Has two different representations for zero
    - Positive zero and negative zero
- We can **simplify computer hardware** by using a *complement* system to represent numbers

Signed Integer Representation

- 8-bit *one's complement* representation:
  - + 3 is: 00000011
  - - 3 is: 11111100 (just invert all the bits!)
- In one's complement representation, as with signed magnitude, negative values are indicated by a 1 in the high order bit
- Complement systems are useful because they eliminate the need for subtraction – just complement one and add them together!

Signed Integer Representation

- One’s complement is simpler to implement in hardware than signed magnitude
  - Don’t need to compare numbers to see which is larger (for mixed signs)
- Still one disadvantage
  - Positive zero and negative zero
- Solution? *Two’s complement* representation
  - **Used by all modern systems**

Signed Integer Representation

- To express a value in *two's complement* representation:
  - If the number is **positive**, just convert it to binary and you're **done**
  - If the number is **negative**, find the **one's complement** of the number (i.e. invert bits) and then **add 1**
- Example:
  - In 8-bit binary, 3 is:
    `00000011` *(notice how nothing has changed!)*
  - -3 using one's complement representation is:
    `11111100`
  - Adding 1 gives us -3 in two's complement form:
    `11111101`

Signed Integer Representation

- With two's complement arithmetic, all we do is add the two binary numbers and **discard any carries** from the high order bit
- Example: Using two's complement binary arithmetic, find 48 + -19 = 29

```
(1) ←  1 1
       0 0 1 1 0 0 0 0
     + 1 1 1 0 1 1 0 1
     -----------------
       0 0 0 1 1 1 0 1
```

48 in binary is: 00110000
19 in binary is: 00010011,
-19 using one's complement is: 11101100,
-19 using two's complement is: 11101101.

Reminders

For positive numbers, the *signed-magnitude*, *one's complement*, and *two's complement* forms are all **the same**!

In *one's complement* / *two's complement* form, you only need to modify the number if it is **negative**!

Range

- **What is the smallest and largest 8-bit two's complement number?**
  - $XXXXXXXX_2$
  - Smallest (negative) # = 10000000_2 = **-128**
  - Largest (positive) # = 01111111_2 = **127**

Overflow

- **Overflow**: The result of a calculation is too large or small to store in the computer
  - We only have a finite number of bits available for each number
- Can we **prevent** overflow?
  - Not without re-writing your program to use values that can fit within computer memory
- Can we **detect** overflow? Yes!
  - Easy to detect in complement arithmetic
  - See book for a set of rules that you could implement in hardware