# CS61, Fall 2011
# Section 1: Binary representations and basic assembly

## Today's topics

- Binary basics
- Signed numbers
- Addressing and byte ordering
- Basic assembly
- Operand specifiers

While we're at it, we'll also get familiar with the course infrastructure and gdb.

## 1 Reminders

Assignment 1 is due Tuesday, Sept. 13, 11:59pm.

Assignment 2 is due Thursday, Sept. 22, 11:59pm.

Please try and attend the section you were assigned to. However, on a trial basis, we are allowing students to attend other sections. There will be short quizzes on recent material at the beginning of some sections. These quizzes will count towards your grade and if you miss a quiz, you will receive a zero for that quiz. Your lowest two quiz scores will be dropped.

## 2 Course infrastructure

If you haven't tried the course infrastructure out yet, please do so soon. You'll need it to complete the labs during the course (including Homework 2, out now). Instructions are on the wiki.

## 3 Binary basics

Some useful terminology:

| | |
|---|---|
| *MSB* | most significant bit—bit position with highest bit value |
| *LSB* | least significant bit—bit position with lowest bit value |
| *Nibble* | a group of four bits |

**Q1:** What's the maximum value of a nibble? *15*

### Converting from binary to decimal

$$101_2 = (2^2 \times 1) + (2^1 \times 0) + (2^0 \times 1) = 5_{10}$$

### Converting from binary to hexadecimal

1. Divide digits into nibbles
2. Convert each nibble into the corresponding hexadecimal digit

$$010011111011 \rightarrow 0100\ \ 1111\ \ 1011 \rightarrow 4\text{FB}_{16} \rightarrow 0\text{x}4\text{FB}$$

To reverse, convert each hexadecimal digit into the corresponding nibble.

**Q2:** Fill in the missing values:

| Binary | Hexadecimal | Decimal |
|---|---|---|
| $101101_2$ | 0x2D | *45* |

**Bit shifting** Bit shifting is a simple way to multiple or divide by $2^n$.

$<< n$, or left shift, is equivalent to multiplication by $2^n$
Compute $12 \times 4$
$1100 << 2 = 110000 = 32_{10} + 16_{10} = 48$

$>> n$, or the *arithmetic* right shift, is equivalent to division by $2^n$
Compute $12/4$
$1100 >> 2 = 11 = 2_{10} + 1_{10} = 3$

## 4 Signed numbers

x86 uses the *two's complement* representation of negative numbers. From lecture and the book, you'll remember there are three steps to representing a negative number in two's complement:

1. Write down the number in binary.
2. Invert the bits (this gives us the *ones' complement*).
3. Add 1.

For example, let's write $-42$ in binary. First, convert to binary (in this case assume we're using one byte): 00101010. Then, invert the bits: 11010101. Finally, add 1: 11010110.

Note that the MSB is '1', so we should interpret this as a negative number. It's a bit misleading to call it the 'sign' bit because it also has weight, unlike in a sign-and-magnitude representation.

As for unsigned representations, we can write a formula for the decimal interpretation of a binary number in two's complement:

$$-b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i.$$

We can use this formula to interpret the number we computed above, 11010110:

$$-1 \times 128 + 1 \times 64 + 1 \times 16 + 1 \times 4 + 1 \times 2 = -42.$$

**Remember:** information is bits + context. The 'value' of a binary representation really depends on how we interpret it. For example, if we interpret the binary representation above as an unsigned integer, we get 214 (instead of -42).

You can try it in C:

```
printf("%d␣(signed)", (char)0xD6);
printf("%d␣(unsigned)", 0xD6);
```

**Q3:** Fill in the missing values:

| Binary | Unsigned byte | Signed byte |
|---|---|---|
| 10000000 | *128* | *-128* |
| 11111111 | *255* | *-1* |
| 1 | *1* | *1* |
| 0 | *0* | *0* |

**Q4:** During lecture and assignment 1, we discussed a few other representations (ones' complement, base negative two, ...). Why is the two's complement so commonly used?

*Because arithmetic operations are the same for both positive and negative numbers and no value is 'wasted' on a second representation of 0.*

**Q5:** Why does the compiler often generate code where memory locations are accessed using large offsets? For example, in `mov fffffffc(%ecx), %eax`. What is this equivalent to?

*Treated as a signed number, the offset is $-4$.*

## 5 GDB interlude

Some gdb commands you should know:

| | |
|---|---|
| break | set a breakpoint |
| disas | disassemble a function |
| si | step forward one (or more) assembly instructions at a time |
| print | print a register value |
| x | print memory value(s) |

## 6 Addressing and byte ordering

In virtually all machines, a multi-byte object is stored as a contiguous sequence of bytes, with the address of the object given by the smallest address of the bytes used.

When we write a hexadecimal value, like 0x1a2b3c4d, we write it with the most significant byte (0x1a) on the left and the least significant byte (0x4d) on the right. That's just because humans have a convention of writing numbers in this way. But when such a value is stored in memory, the order in which the bytes are stored depends on the *endianness* (**end**-ian-ness) of the CPU architecture. The x86 is a "little-endian" machine, which means it stores the low-order byte of a word in the LOWEST MEMORY ADDRESS. So, the value 0x1a2b3c4d would be stored in memory as:

| **memory address** | M | M+1 | M+2 | M+3 |
|---|---|---|---|---|
| **content** | 0x4d | 0x3c | 0x2b | 0x1a |

A "big-endian" architecture (like the Sun SPARC processor) stores the highest-order byte of a word in memory first.

Why does this matter? Because when you look at a memory dump, say using GDB, you will see the contents of memory as they are stored. If you intend to interpret 4 bytes stored in memory as a word, you need to remember to 'swap the bytes' to make up the actual hex value of the word. So a memory dump of 0xab 0x40 0x1b 0x67 would be interpreted as the value 0x671b40ab.

**Q6:** What happens when computers with different endianness try to communicate? What is a solution to this problem?

*Most RFCs for network protocols define a standard network order. For the Internet Protocol, this is big-endian. The Berkeley sockets API (the standard C networking API) provides a family of functions (*htonl, htons, ntohl, ntohs*) to manage the required conversions in a portable way.*

## 7 Operand specifiers

Assembly instructions can take three different types of operands: a constant, or *immediate*, value, a register value, or a memory value.

| Type | Form | Operand value | Example |
|---|---|---|---|
| Immediate | $\$Imm$ | $Imm$ | \$42 |
| Register | $E_a$ | $R[E_a]$ | %**eax** |
| Memory | $Imm(E_b, E_i, s)$ | $M[Imm + R[E_b] + R[E_i] \times s$ | \$42(%**esp**, %**edx**, 4) |

This last operand form is one of many ways of accessing memory, though it is the most general. The full list of memory operand specifiers is given in Figure 3.3 of the text (pp169). This is the most useful form to remember though, because we can derive all of the others from it. Basically, the other forms leave of some of the arguments. One way to think of those forms is as supplying "default" arguments to this specifier, where the defaults are 0 for $Imm$, $R[E_b]$, and $R[E_i]$, and 1 for $s$.

*Things to note:*

1. *b for "base", i for "index", s for "scale"*
2. *scale has to be one of 1, 2, 4, or 8*

**Q7:** Assume the following values are stored at the indicated memory addresses and registers.

| Address | Value | Register | Value |
|---|---|---|---|
| 0x100 | 0xFF | %**eax** | 0x100 |
| 0x104 | 0xAB | %**ecx** | 0x1 |
| 0x108 | 0x13 | %**edx** | 0x3 |
| 0x10C | 0x11 | | |

Fill in the missing value for each operand:

| Operand | Value |
|---|---|
| %**eax** | 0x100 |
| 0x104 | 0xAB |
| \$0x108 | 0x108 |
| (%**eax**) | 0xFF |
| 4(%**eax**) | 0xAB |
| 9(%**eax**,%**edx**) | 0x11 |
| 260(%**ecx**,%**edx**) | 0x13 |
| 0xFC(,%**ecx**,4) | 0xFF |
| (%**eax**,%**edx**,4) | 0x11 |

**Q8:** A function with prototype `int decode(int x, int y, int z);` is compiled into assembly. The body of the code is as follows:

```
1 # x at %ebp+8, y at %ebp+12, z at %ebp+16
2 movl   12(%ebp), %edx
3 subl   16(%ebp), %edx
4 movl   %edx, %eax
5 sall   $31, %eax
6 sarl   $31, %eax
```

```
7  imull   8(%ebp), %edx
8  xorl    %edx, %eax
```

Parameters `x`, `y`, and `z` are stored at memory locations with offsets 8, 12, and 16 relative to the address in register $\%\mathbf{ebp}$. The code stores the return value in register $\%\mathbf{eax}$. Write the C code for `decode` that will have an effect equivalent to our assembly code.

*This week in lecture: why are the variables in memory at offsets from $\%\mathbf{ebp}$? Why does the code return the value in $\%\mathbf{eax}$? and other calling conventions...*

*Breaking things down by line:*

*2* $\%\mathbf{edx} = y$
*3* $\%\mathbf{edx} = \%\mathbf{edx} - z$
*4* $\%\mathbf{eax} = y - z$
*5* $\%\mathbf{eax} = \%\mathbf{eax} << 31$
*6* $\%\mathbf{eax} = \%\mathbf{eax} >> 31$
*7* $\%\mathbf{edx} = \%\mathbf{edx} \times x$
*8* $\%\mathbf{eax} = \%\mathbf{eax}^{\%\mathbf{edx}}$

*Remember: operands are usually ordered Source, Destination, and arithmetic operations are Destination = Destination 'op' Source.*

```
int decode(int x, int y, int z)
{
    int t1 = y - z;
    int t2 = x * t1;
    int t3 = (t1 << 31) >> 31;
    int t4 = t3 ^ t2;

    return t4;
}
```

## 8 Getting a *really* highscore.

**Q9:** How do you get the highest score on the highscore binary?

*Overflow! It turns out that big negative number we can get from score4 is quite handy.*

$10 + 10 - 65556 - 2147418113 = -2147483649$, *but the smallest negative number we can represent is* $-2147483648$, *so we wrap around to the biggest positive number,* $2147483647$.