Winter 2021 CS 33 Lecture Notes

Kyle Chui

August 30, 2021

Contents

1 Bits and Bytes 3

1.1 Representing Data in Bits and Bytes . . . 3

1.1.1 Everything is Bits . . . 3

1.1.2 Encoding Byte Values . . . 3

1.2 Bit Manipulation . . . 3

1.2.1 Boolean Algebra . . . 3

1.2.2 Bit-Level Operations in C . . . 4

1.2.3 Logic Operations in C . . . 4

1.2.4 Shift Operations . . . 4

2 Integers 5

2.1 Encoding Integers . . . 5

2.1.1 Bits to Integers . . . 5

2.1.2 Numeric Ranges . . . 5

2.1.3 Values for Different Word Sizes . . . 5

2.1.4 Unsigned and Signed Numeric Values . . . 5

2.2 Converting and Casting Integers . . . 6

2.2.1 Mapping Between Signed and Unsigned . . . 6

2.2.2 Relation Between Signed and Unsigned . . . 6

2.2.3 Signed vs. Unsigned in C . . . 6

2.2.4 Casting Surprises . . . 6

2.2.5 Summary . . . 6

2.3 Expanding and Truncating Integers . . . 7

2.3.1 Sign Extension . . . 7

2.3.2 Truncation . . . 7

2.4 Adding, Negating, Multiplying, and Shifting . . . 7

2.4.1 Unsigned Addition . . . 7

2.4.2 Two's Complement Addition . . . 7

2.4.3 Multiplication . . . 7

2.4.4 Unsigned Multiplication in C . . . 8

2.4.5 Signed Multiplication in C . . . 8

2.4.6 Power-of-2 Multiplication Using Shifts . . . 8

2.4.7 Unsigned Power-of-2 Division Using Shifts . . . 8

2.4.8 When To Use Unsigned . . . 8

3 Representations in Memory, Pointers, and Strings 8

3.1 Machine Words . . . 9

3.2 Byte Ordering . . . 9

3.3 Examining Data Representations . . . 9

3.4 Representing Strings . . . 9

4 Machine-Level Programming I: Basics 10

4.1 History of Intel processors and architectures . . . 10

4.2 C, assembly, machine code . . . 10

4.2.1 Assembly/Machine Code View . . . 10

4.2.2 Turning C into Object Code . . . 11

4.2.3 Assembly Characteristics: Data Types . . . 11

4.2.4 Assembly Characteristics: Operations . . . 11

4.2.5 Object Code . . . 11

4.3 Analysing Object Code . . . 11

4.3.1 Using the Disassembler . . . 11

4.3.2 Alternate Disassembly Using gdb . . . 12

4.4 x86-64 Integer Registers . . . 12

4.5 Some History: IA32 Registers . . . 12
4.6 Moving Data . . . 12
4.6.1 Operand Types . . . 12
4.7 `movq` Operand Combinations . . . 13
4.8 Memory Addressing Modes . . . 13

5 Arithmetic and Logical Operations 13
5.1 Address Computation Instruction . . . 13
5.2 Some Arithmetic Operations . . . 14

6 Machine-Level Programming II: Control 15
6.1 Control: Condition Codes . . . 15
6.1.1 Processor State (x86-64, Partial) . . . 15
6.2 Implicit Setting of Condition Codes . . . 15
6.3 Explicit Setting of Condition Codes . . . 15
6.4 Reading Condition Codes . . . 16

1 Bits and Bytes

1.1 Representing Data in Bits and Bytes

1.1.1 Everything is Bits

- Every bit is either a 0 or a 1.
- We can use sets of bits to not only tell the computer what to do (give it instructions), but also represent data.
- We use bits because they are easy to store, and reliably transmitted on noisy/inaccurate wires.

Example. *Counting in Binary*
We can use bits to represent numbers in base 2, or the binary number system. Thus we can use a set of bits to encode any number we want.

1.1.2 Encoding Byte Values

Definition. *Byte*
A *byte* is 8 bits, and can be thought of as a string of 0's and 1's of length eight.

- In binary, a byte can range from 00000000_2 to 11111111_2.
- In decimal, a byte can range from 0_{10} to 255_{10}.
- In hexadecimal (base 16), a byte can range from 00_{16} to FF_{16}.

> **Note.** In hexadecimal, once you finish using the digits 1 through 9, you use the letters A through F to represent values of 10_{10} through 15_{10}. In C, we append "0x" before a string to indicate that it is a hexadecimal number (which may either be upper-case or lower-case). For example, $\mathrm{FA1D37B}_{16}$ would be written as "0xFA1D37B" or "0xfa1d37b".

All data is just a long string of bits, so any value that we get out of it depends on the *context* of what we are reading in (a `double`, `char`, `int`, etc).

1.2 Bit Manipulation

1.2.1 Boolean Algebra

Developed by George Boole in the 19th century, *Boolean Algebra* is an algebraic representation of logic, where 1's denote a "true" value and 0's denote a "false" value. Some common logical operations are described in the tables below:

$$\underbrace{\begin{array}{c|cc} \& & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}}_{\text{“and” operator}} \qquad \underbrace{\begin{array}{c|cc} | & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 1 \end{array}}_{\text{“or” operator}} \qquad \underbrace{\begin{array}{c|c} \sim & \\ \hline 0 & 1 \\ 1 & 0 \end{array}}_{\text{“not” operator}} \qquad \underbrace{\begin{array}{c|cc} \hat{} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 1 \end{array}}_{\text{“xor” operator}}$$

> **Note.** The "xor" operator stands for "exclusive-or", meaning "either one or the other, but not both".

All of these operations are applied "bitwise" on bit vectors—the jth bit of the result is obtained by applying the operation on the jth bit of the input(s).

Example. *Representing an Manipulating Sets*

Representation

- A width w bit vector can represent a subset of $\{0, \ldots, w-1\}$.
- We do this by letting $a_j = 1$ if j is in our set.
- For example, the bit vector 01101001 would represent the set $\{0, 3, 5, 6\}$. We read from right to left, and we see that the 0th, 3rd, 5th, and 6th entries contain 1's.

Operations

- There are nice parallels between operations on bit vectors and operations on sets, i.e.
  - & on bit vectors is the same as set intersection.
  - | on bit vectors is the same as set union.
  - ˆ on bit vectors is the same as the symmetric difference.
  - ~ on bit vectors is the same as taking the complement of a set.

1.2.2 Bit-Level Operations in C

The four boolean operations covered thus far are available in C, and can apply to any "integral data type"(long, int, char, etc). The operators view the arguments as bit vectors and are applied bit-wise.

1.2.3 Logic Operations in C

Note. These are different than the bit-level operations, so don't get them confused.

There are also the logical operators in C, namely `&&`, `||`, `!`.

- These view 0 as "false", and anything non-zero as "true".
- They always return 0 or 1, and can terminate early.

1.2.4 Shift Operations

The shift operator allows you to "move" the bits in a bit vector to give them higher or lower significance.

- The left shift($x \ll y$) shifts the bit vector x to the left by y positions (extra bits on the left are tossed out and empty slots are filled with 0's).
- The right shift($x \gg y$) shifts the bit vector x to the right by y positions (extra bits on the right are tossed out).
  - In a logical shift, the empty slots are filled with 0's.
  - In an arithmetic shift, the empty slots are filled with duplicates of the most significant bit on the left.
- It is undefined behaviour if the shift amount is negative or greater than or equal to the string size.

2 Integers

2.1 Encoding Integers

2.1.1 Bits to Integers

For unsigned integers, we can just use a summation of powers of two to express integers, given by

$$B2U(x) = \sum_{i=0}^{w-1} x_i \cdot 2^i.$$

For signed integers, we use the *Two's complement* form, where we treat the first bit as negative (otherwise known as the "sign bit"), and take the complement of the non-negative bit representation of our integer. It is given by the equation

$$B2T(x) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i.$$

2.1.2 Numeric Ranges

There's a limited range of numbers that we can represent using our bit representations, because we only use a finite number of bits. For unsigned values, this ranges from

$$\underbrace{000\ldots0}_{0} \quad \text{to} \quad \underbrace{111\ldots1}_{2^w-1}.$$

For Two's complement values, this ranges from

$$\underbrace{100\ldots0}_{-2^{w-1}} \quad \text{to} \quad \underbrace{011\ldots1}_{2^{w-1}-1}.$$

Another interesting number to remember is that $111\ldots1$ represents -1 in Two's complement form.

2.1.3 Values for Different Word Sizes

We have that $|T_{\min}| = T_{\max} + 1$ and $U_{\max} = 2 \cdot T_{\max} + 1$.

In C, the header file `limits.h` declares constant such as:

- `ULONG_MAX`
- `LONG_MAX`
- `LONG_MIN`

> **Note.** The values provided by this header file are platform specific.

2.1.4 Unsigned and Signed Numeric Values

- For non-negative values, unsigned and signed values have the same bit representations.
- Every bit pattern represents a unique integer value, and every representable integer has a unique bit encoding.
- These mappings are invertible, which is to say that $U2B(x) = B2U^{-1}(x)$ and $T2B(x) = B2T^{-1}(x)$.

2.2 Converting and Casting Integers

2.2.1 Mapping Between Signed and Unsigned

To convert between signed and unsigned values, we first convert to the bit representation of the value and then to the other form. In other words,

$$T2U = B2U \circ T2B \quad \text{and} \quad U2T = B2T \circ U2B.$$

We are maintaining the same *bit pattern* when we are doing our mappings.

> **Note.** This does not necessarily preserve the value of the bit pattern, as the same bit pattern does not always mean the same value in Two's complement and unsigned forms.

2.2.2 Relation Between Signed and Unsigned

The main difference when going from unsigned to signed is that the most significant bit goes from being a large positive weight to a large negative weight. Thus the value difference between unsigned and Two's complement is $2^{w-1} + 2^{w-1} = 2^w$ when the most significant digit is a one.

2.2.3 Signed vs. Unsigned in C

- Constants are by default considered to be signed integers, and are only unsigned if explicitly written that way (using a "U" as a suffix, i.e. `4294967259U`).
- Casting
  - Explicit casting between signed and unsigned values is the same as our previously defined functions $U2T$ and $T2U$.

    ```
    int tx, ty;
    unsigned ux, uy;
    tx = (int) ux;
    uy = (unsigned) ty;
    ```

  - Implicit casting also occurs via assignments and procedure calls.

    ```
    tx = ux;
    uy = ty;
    ```

2.2.4 Casting Surprises

- Expression evaluation
  - If there is a mix of unsigned an signed values in a single expression, *signed values will be implicitly cast to unsigned values.*
  - This includes the comparison operators: `<`, `>`, `==`, `<=`, `>=`

2.2.5 Summary

- When converting, the bit pattern is maintained, but reinterpreted.
- This can have unexpected effects, such as being off by 2^w.

> **Note.** In an expression containing signed and unsigned integers, `int` is cast to `unsigned`.

2.3 Expanding and Truncating Integers

2.3.1 Sign Extension

Given a w-bit signed integer x, we want to convert it to a $w+k$-bit integer with the same value. To do this, we make k copies of the sign bit:

$$X' = \underbrace{x_{w-1}\dots x_{w-1}}_{k \text{ copies of the most significant bit}} x_{w-1}x_{w-2}\dots x_0$$

When converting from a smaller to larger integer data type, C will automatically perform the sign extension.

2.3.2 Truncation

In truncation, bits are removed and the result is reinterpreted. For unsigned integers, this is the same as just performing the modulo operator on the value, and for signed integers it is *similar* to the modulo operator. For small numbers this yields the expected behaviour.

2.4 Adding, Negating, Multiplying, and Shifting

2.4.1 Unsigned Addition

If both of the operands have w bits, then the "true sum" would have to contain $w+1$ bits. The discard carry, denoted as $\text{UAdd}_w(u,v)$, contains w bits (by tossing out the most significant bit).

The standard addition function ignores the carry output, and this is the same as performing modular arithmetic:

$$\text{UAdd}_w(u,v) = u + v \pmod{2^w}$$

2.4.2 Two's Complement Addition

If the sum gets to be too positive or too negative, the sum will overflow in both directions. In other words, if a number gets to be too negative, it will overflow and become a large positive integer. If a number gets to be too positive, it will overflow and become a large (in magnitude) negative integer.

> **Note.** To negate a number in Two's complement form, first take it's bit complement and add one. This is because we know that
>
> $$x + {\sim}x = 11\dots1_2 = -1,$$
>
> so $-x = {\sim}x + 1$.

2.4.3 Multiplication

We want to compute the product of two w-bit numbers x and y (which can either be signed or unsigned). However, exact results can be larger than w bits:

- Unsigned multiplication can yield results of up to $2w$ bits, where

$$0 \le x \cdot y \le \left(2^w - 1\right)^2 = 2^{2w} - 2^{w+1} + 1.$$

- The minimum for Two's complement multiplication can have up to $2w-1$ bits, where

$$x \cdot y \ge \left(-2^{w-1}\right) \cdot \left(2^{w-1} - 1\right) = -2^{2w-2} + 2^{w-1}.$$

- The maximum for Two's complement multiplication can have up to $2w$ bits, where

$$x \cdot y \le \left(-2^{w-1}\right)^2 = 2^{2w-2}.$$

Thus to maintain exact results, we would need to keep expanding the string size every time we take a product. This can be done in software using "arbitrary precision" packages.

2.4.4 Unsigned Multiplication in C

As mentioned before, the true product of two w-bit strings will (potentially) have $2w$ bits, so C discards the first w bits. We denote this as $\mathrm{UMult_w(u, v)}$. Again this is modular arithmetic, so

$$\mathrm{UMult_w(u, v)} = u \cdot v \pmod{2^w}.$$

2.4.5 Signed Multiplication in C

We still ignore the first w bits, but we could have overflowed in either the positive or negative directions. The lower bits are the same.

2.4.6 Power-of-2 Multiplication Using Shifts

Notice that left shifting an bit vector k times yields the same result as if you had multiplied it by 2^k. In other words, $u \ll k = u \cdot 2^k$.

> **Note.** This only works if there is no overflow in the multiplication process.

Example. *Shifting to Multiply*

- `u ≪ 3 == u * 8`
- `(u ≪ 5) - (u ≪ 3) == u * 24`

Most machines shift and add faster than they can multiply, and the compiler generates this code automatically.

2.4.7 Unsigned Power-of-2 Division Using Shifts

This is done by performing a logical right shift on a vector (replace empty slots with zeroes). We have $u \gg k = \left\lfloor \frac{u}{2^k} \right\rfloor$.

2.4.8 When To Use Unsigned

- Use it when performing modular arithmetic.
- Use it when using bits to represent sets.

> **Note.** Don't use it without fully understanding the consequences, otherwise you can (read: will) have unintended behaviour.

3 Representations in Memory, Pointers, and Strings

Remember that the way that memory is organised in a computer is byte-oriented.

Programs refer to data by address:

- Conceptually, envision memory as a very large array of bytes (not actually true).
- An address is like an index into that array, and a pointer variable stores an address.

> **Note.** The system provides private address spaces to each "process". Thus a program can interact with its own data, but not that of other programs.

3.1 Machine Words

Any given computer has a given "word size", which is the nominal size of integer-valued data. Up until recently, most machines used 32 bits as their word size (limits addresses to 4GB), but nowadays machines have 64-bit word size (up to 18 exabytes of addressable memory). Machines still support multiple data formats, using fractions or multiples of their word size to keep the number of bytes an integer.

3.2 Byte Ordering

There are two main conventions: Big Endian and Little Endian. In the former, the least significant byte has the highest address (more intuitive), and in the latter, the least significant byte has the lowest address.

> **Note.** Big Endian—most to least significant. Little Endian—least to most significant.

3.3 Examining Data Representations

Here is some code that will print out the hexadecimal value at a given pointer location:

```
typedef unsigned char *pointer;
void show_bytes(pointer start, size_t len) {
  size_t i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n", start+i, start[i]);
  printf("\n");
}
```

In the above code, `%p` prints a pointer and `%x` prints a hexadecimal.

> **Note.** Different compilers and machines assign different locations to objects. You might even get different results each time the program is run.

3.4 Representing Strings

In C, each string is represented by an array of characters, which are encoded using the ASCII format. In ASCII, the first 7 bits encode the character, while the last bit is 0, which denotes the end of the character. Byte ordering is also not an issue for strings, as endian ordering only matters for individual values, not the array as a whole.

4 Machine-Level Programming I: Basics

4.1 History of Intel processors and architectures

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible (up until 8086, introduced in 1978).
  - More features were added as time went on.
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats.
  - Hard to match the performance of Reduced Instruction Set Computers (RISC).
  - Intel has performance, but not great at power efficiency.

4.2 C, assembly, machine code

> **Definition.** *Instruction Set Architecture (ISA)*
> An *Instruction Set Architecture* is the parts of a processor design that one needs to understand or write assembly/machine code, i.e. instruction set specification, registers.

> **Definition.** *Microarchitecture*
> The *microarchitecture* is the implementation of the architecture, i.e. cache sizes and core frequency.

> **Definition.** *Code Forms*
> There are two code forms—*machine code*, which are the byte-level programs that a processor executes, and *assembly code*, which is a text representation of machine code.

Some examples of ISAs include x86, IA32, x86-64, and ARM.

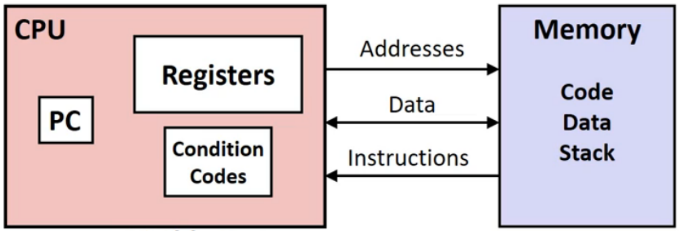
4.2.1 Assembly/Machine Code View

CPU:

- PC: Program Counter
  - Keeps track of which instruction will be run next, and is called the "RIP" for x86-64 architectures.
- The Register file is used for heavily used program data (because it is very fast).
- Condition codes are special registers that store status information about the most recent arithmetic or logical operation, and are used for conditional branching.

Memory:

- We think of memory as a large array where each element is a byte.
- It holds code and user data, and uses a stack to support procedure calls.



4.2.2 Turning C into Object Code

We first code in human-readable files, such as `.c` files, and then compile them into an executable program (which is a binary and not human-readable).

1. We start with a C program, which is written in text.
2. The code gets compiled into an Asm program, which is still in text.
3. The Asm program goes through an assembler, which turns it into an object program (a binary).
4. The object program finally goes through a linker, which turns it into an executable program (also a binary).

4.2.3 Assembly Characteristics: Data Types

- "Integer" data of 1, 2, 4, or 8 bytes (includes data values and addresses).
- Floating point data of 4, 8, or 10 bytes.
- Code: Byte sequences encoding series of instructions.
- No aggregate types such as arrays or structures (just contiguously allocated bytes in memory).

4.2.4 Assembly Characteristics: Operations

- Perform arithmetic functions on register or memory data.
- Transfer data between memory and register.
- Transfer control (changing the instruction pointer of the program counter).

4.2.5 Object Code

Assembler:

- Translates `.s` into `.o`.
- Binary encoding of each instruction.
- Nearly-complete image of executable code.
- Missing linkages between code in different files.

Linker:

- Resolves references between files.
- Combines with static run-time libraries (e.g. code for `malloc`, `printf`).
- Some libraries are *dynamically linked* (but linking occurs when program begins execution).

4.3 Analysing Object Code

In the following examples, we are analysing the function `sumstore`.

4.3.1 Using the Disassembler

The command here would be `objdump -d sumstore`.

- It analyses the bit pattern for the series of instructions, and then products an approximate rendition of the assembly code.
- Can be run on either a `.out` (complete executable) or `.o` file.

4.3.2 Alternate Disassembly Using gdb

The commands here would be:

```
gdb sum
disassemble sumstore
```

This will produce a list of locations and actions done (in assembly code), but not the actual bits inside each address. To see the first 14 bytes of `sumstore`, you can call `x/14xb sumstore`.

> **Note.** Anything that can be interpreted as executable code can be disassembled, but reverse-engineering for many programs is illegal.

4.4 x86-64 Integer Registers

Of the 16 registers listed below for a 64-bit system, the first eight share space with older 32-bit registers. You can reference the first four bytes of the first eight 64-bit registers by replacing the "r" with an "e" (i.e. `%rax` → `%eax`). For the other eight 64-bit registers, you append a `d` at the end (i.e. `%r13` → `%r13d`).

`%rax`	`%rbx`	`%rcx`	`%rdx`	`%rsi`	`%rdi`	`%rsp`	`%rbp`
`%r8`	`%r9`	`%r10`	`%r11`	`%r12`	`%r13`	`%r14`	`%r15`

> **Note.** The `%rsp` register is the stack pointer register, and thus is the only one (for the purposes of this class) that cannot be used as a general purpose register.

4.5 Some History: IA32 Registers

In older IA32 machines, the `%ebp` register was also reserved to point at the top of the stack, in addition to the `%esp` register being reserved as the stack pointer register. Furthermore the 32-bit registers could be decomposed even further by removing the `e` from the beginning of the register name (i.e. `%edi` → `%di`). Finally, the first four 16-bit registers (`%ax`, `%cx`, `%dx`, `%bx`) could all be split into "high" and "low" registers, i.e. `%cx` could be split into `%ch` and `%cl`.

4.6 Moving Data

To move data with assembly code, we have the `movq source, dest` command.

> **Note.** The `q` in `movq` stands for "quad word", or a 64-bit quantity. There is also `movl` for "long word", or a 32-bit quantity, etc.

4.6.1 Operand Types

The `movq` command takes two parameters for its source and destination, which can either be registers, immediates, or memory locations.

- Immediates are constant integer data, just like constants in C, although they are prefixed with a `$`, like `$0x400`.
  - They are encoded with 1, 2, or 4 bytes.
- Registers are one of the 16 integer registers mentioned earlier.
  - We already know that `%rsp` is reserved for special use, but some of the other registers also have special use cases for particular instructions.
- Memory is 8 consecutive bytes of memory at the address given by the register, i.e. `(%rax)`. There are also various other "address modes".

4.7 `movq` Operand Combinations

In the following diagram, the first column indicates the source, the second the destination, the third an example of assembly code, and the final an example of C code.

> **Note.** Where there are no parentheses around a register, i.e. `%rax`, we are referring to writing *directly* to the register. Where there are parentheses around the register, we dereference it and write to where the register points in memory. The only exception to this is the `leaq` command, which does address computation arithmetic when there are parentheses present.

`movq`	Immediate	Register	`movq $0x4, %rax`	`temp = 0x4;`
		Memory	`movq $-147, (%rax)`	`*p = -147;`
	Register	Register	`movq %rax, %rdx`	`temp2 = temp1;`
		Memory	`movq %rax, (%rdx)`	`*p = temp;`
	Memory	Register	`movq (%rax), %rdx`	`temp = *p;`

> **Note.** You *cannot* do memory-memory transfer with a single instruction.

4.8 Memory Addressing Modes

The first two methods below are simple, and the final the most general form of memory addressing.

- Normal: The register `R` specifies the memory address, and `(R) = Mem[Reg[R]]` dereferences the pointer. For example, we have `movq (%rcx), %rax`.
- Displacement: The register `R` specifies the start of the memory region, and the constant displacement `D` specifies the offset. The syntax is `D(R) = Mem[Reg[R] + D]`.
  For example, we have `movq 8(%rbp), %rdx`.
- Most General Form: `D(Rb, Ri, S)` which translates to `Mem[Reg[Rb] + S*Reg[Ri] + D]`.
  - `D` is a constant "displacement" of 1, 2, or 4 bytes.
  - `Rb` is the base register (any of the 16 integer registers).
  - `Ri` is the index register, which can be any register *but* `%rsp`.
  - `S` is the scale, which is 1, 2, 4, or 8.

5 Arithmetic and Logical Operations

5.1 Address Computation Instruction

The `leaq src, dst` command takes in two parameters:

- `src` is an address mode expression that we just covered.
- `dst` is the address denoted by that expression.

> **Note.** You are essentially computing the address referred to by `src`, and then setting `dst` equal to that address. You are *not* dereferencing anything when using `leaq`.

This has several use cases, such as computing addresses without a memory reference (i.e. translation of `p = &x[i];`) or computing arithmetic expressions of the form `x + k*y`, where `k` is a 1, 2, 4, or 8 byte value.

Note. If the command does not specify the size, i.e. just a `mov` command, you can always look at the registers that it is operating on.

Register	`%al`	`%ax`	`%eax`	`%rax`
Syntax Example	`movb`	`movw`	`movl`	`movq`
Space	8 bits	16 bits	32 bits	64 bits

You can actually have multiple extensions after a command, i.e. `movzbl`. The first letter after the `mov` is either a `z` or a `s`, indicating whether we are zero extending or sign extending. The other two letters indicate what size we are moving from and to. For instance, `movzbl` means we are moving from a 8 bit register to a 32 bit quantity, and filling in the gaps with zeroes. Sign extension will duplicate the MSB instead of padding the data with zeroes.

5.2 Some Arithmetic Operations

- Two Operand Instructions

`addq`	Dest = Dest + Src	
`subq`	Dest = Dest - Src	
`imulq`	Dest = Dest * Src	
`salq`	Dest = Dest $\ll$ Src	Also called `shlq`
`sarq`	Dest = Dest $\gg$ Src	Arithmetic Shift
`shrq`	Dest = Dest $\gg$ Src	Logical Shift
`xorq`	Dest = Dest ˆ Src	
`andq`	Dest = Dest & Src	
`orq`	Dest = Dest | Src	

- One Operand Instructions

`incq`	Dest = Dest + 1
`decq`	Dest = Dest $-$ 1
`negq`	Dest = $-$Dest
`notq`	Dest = ~Dest

Note. There is no distinction between signed and unsigned int because these are all bit-level manipulations.

6 Machine-Level Programming II: Control

6.1 Control: Condition Codes

6.1.1 Processor State (x86-64, Partial)

The information about the currently executing program is stored in the 16 integer registers.

- Temporary data can be stored in most of the registers.
- The location of the runtime stack is stored in `%rsp`.
- The location of the current code control point (what command is currently being executed) is stored inside the `%rip` register.
- The status of the recent tests (condition codes)
  - CF—Carry Flag
  - ZF—Zero Flag
  - SF—Sign Flag
  - OF—Overflow Flag

  These four condition codes allow us to create and make more complex comparison operators.

6.2 Implicit Setting of Condition Codes

Condition codes can be implicitly set by arithmetic operations (you can think of this as a side effect of the operations).

- CF is set if there is a carry out from the most significant bit (unsigned overflow).
- ZF is set if the result of the operation is zero.
- SF is set if the result is less than zero (as a signed integer).
- OF is set if there is signed overflow in the MSB.

The condition codes are not set by the `leaq` instruction.

6.3 Explicit Setting of Condition Codes

- Explicit setting by compare instruction
- Explicit setting by test instruction

Neither of the above write to a destination. The zero flag is set when `a&b == 0`, and the signed flag is set when `a&b < 0`.

6.4 Reading Condition Codes

The `setX` sets the lowest-order byte of the destination to a zero r one based on combinations of the other condition codes, and does not change the other 7 bytes.

SetX	Condition	Description
`sete`	`ZF`	Equal/Zero
`setne`	`~ZF`	Not Equal/Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Non-negative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF) | ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

> **Note.** There are different `setX` commands for comparing signed vs unsigned integers.

You can access to lowest-order byte in an x86-64 register:

- For the first four registers, remove the `r` and the `x`, and append an `l`, i.e. `%rax` → `%al`.
- For the next four registers, remove the `r` and append an `l`, i.e. `%rsp` → `%spl`.
- For the final eight registers, just append a `b`, i.e. `%r14` → `%r14b`.

Because the `setX` commands only ever modify the lowest-order bit, these commands usually need to be combined with other commands. For instance, the `movzbl` command will move from one register to another, but if the first is smaller, pad the remaining bits with zeroes. For instance, `movzbl %al, %eax` will pad the other bits of `%rax` with zeroes.