UMBC TRAINING CENTERS

x86 Assembly Programming

TCPRG5000-2021-08-30

Table of Contents

Chapter 1. Introduction to x86

Why Learn Assembly?

Assembly code is the lowest level of computation that most programmers can reasonably reach.

Understanding the cost of low-level instructions helps shape the programmer's use of high-level programming concepts.

Knowledge of assembly language also yields deep knowledge of how processes execute.

Familiarity with assembly language allows for analysis and detection of exploits.

Prerequisites

This material assumes an intermediate-or-better knowledge of C, and some familiarity with simple data structures. Required tools are enumerated in Tools. Use of the Linux command-line tools for compilation will also be required.

History

Assembly language is the human-readable version of machine code, which is just zeroes and ones.

Table 1. The Original Original Program, 1948

`0000 0000 1001 0010`	`ldn 18`
`0000 0000 1001 1010`	`ldn 19`
`0000 0000 1010 0001`	`sub 20`
`0000 0000 0000 0011`	`cmp`
`0000 0000 1010 1100`	`jrp 21`
`0000 0000 1011 0001`	`sub 22`
`0000 0000 1100 0110`	`sto 24`
`0000 0000 1011 0010`	`ldn 22`
`0000 0000 1011 1001`	`sub 23`
`0000 0000 1010 0110`	`sto 20`
`0000 0000 1010 0010`	`ldn 20`
`0000 0000 1011 0110`	`sto 22`
`0000 0000 1100 0010`	`ldn 24`

`0000 0000 0000 0011`	`cmp`
`0000 0000 1100 1000`	`jmp 25`
`0000 0000 1011 1000`	`jmp 23`
`0000 0000 0000 0111`	`stp`
`0000 0000 0000 0000`	`0`
`1000 0000 0000 0001`	`-a`
`0111 1111 1111 1110`	`b`
`1111 1111 1111 1101`	`-3`
`1000 0000 0000 0010`	`-b`
`0000 0000 0000 0001`	`1`
`0000 0000 0000 0000`	`0`
`0000 0000 0001 0000`	`16`

When programmable computers were first made, they were hand-programmed directly in machine code. Assembly made it easier for programmers to confirm what a program was doing, by associating mnemonics like “subtract” instead of “101”.

The oldest computer program was written for the Manchester Small-Scale Experimental Machine\footnote{Tootill, Geoff, The Original Original Program, Computer Conservation Society, Number 20, Summer 1998} back in 1948. Compare the machine code version with its assembly code version.

The program calculates the largest factor of a. A C version may look like:

```c
int gfact(int a)
{
    int b = a - 1;
    while(a / b) {
        --b;
    }
    return b;
}
```

Just as high-level code is more readable than assembly language, assembly language is more readable than machine code.

High-level languages are dependent upon compilers or interpreters to be transformed into machine code. Assembly language has a one-to-one mapping with machine code, which means that assembly code is the ultimate “what you see is what you get” language.

Every chip that runs a program has machine code, and every machine code instruction will have an assembly language mnemonic.

Terminology

x86 refers to the family of compatible machine instructions dating back to the 8086 (June 1978), a 16-bit processor, all the way up to the current day. While there is no abbreviation to refer to them like "x16", Intel chips up to the 286 (February 1982) were 16-bit architectures.

IA-32, sometimes x32, specifically refers to 32-bit architecture, the Intel 386 (October 1985) through the Pentium 4E (February 2004).

IA-64 was an attempt by Intel to make a 64-bit architecture that was less backwards compatible with IA-32, the Itanium chip (May 2001). This failed to take on in popularity in the desktop realm, but is still used in some high-end HP servers. It is only supported in some versions of UNIX.

x64 or x86-64 refers to the 64-bit extensions made by AMD, now industry standard for 64-bit x86 chips. This is sometimes also referred to as AMD64, as AMD released the first such chip, the Opteron (April 2003). Modern x86 architecture is x64.

Because these chips are backwards compatible across nearly 30 years, some instructions and registers are applicable across the entire family of chips.

Optimization & Intuition

Humans are no longer better than compilers when it comes to optimizing programs for speed. Humans can do a better job optimizing for space, because the human can better see how to change the structure without changing the meaning.

The compiler can make very different choices when it comes to writing assembly code than a human might make. Sometimes, these choices make no sense at first, but that is because the compiler has much greater insight into performance than a human programmer. For instance, a human may write the following pseudocode:

```
value = 0;
```

A compiler would write the following pseudocode:

```
value ^= value;
```

Both lines have the same outcome, but the performance metrics at the CPU level are drastically different.

Since this assembly code is largely hidden from the programmer, readability is not the same concern that efficiency is.

Exercises

Exercise 1

Read **The Story of Mel** in The Story of Mel

Exercise 2

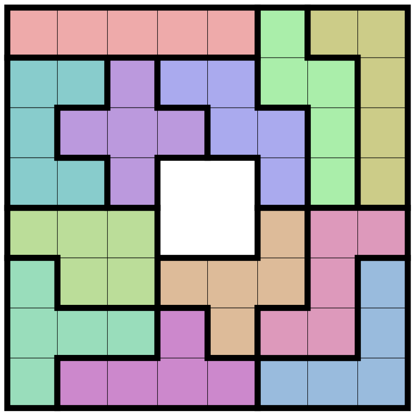
Write four different statements that set an integer variable `value` to 0 using C.

Exercise 3

Write two different statements in C that evaluate to true if the integer variables `alpha` and `beta` are equal.

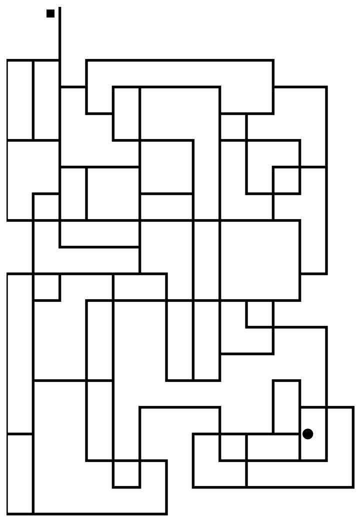
Exercise 4

As seen in the picture, the twelve **pentomino** shapes can be used to create an 8×8 grid, but it leaves a hole in the middle. Make a rectangular shape that contains no holes, using the pentominoes. The tiles may be flipped over if needed. It is possible to make rectangles from 5×4 all the way up to 5×12.



Exercise 5

Starting at the square, give someone clear and unambiguous driving directions how to get to the house marked with a circle.



Chapter 2. Executable Programs

Every program must be loaded into memory to be run, and must be built in such way that it can be loaded first.

There are many programming languages that can built such programs. This course will focus on the C language, the C11 standard.

When languages are interpreted, the interpreter itself is the binary program that is loaded into memory, and must have been built prior.

End-to-end Process

Every program must first start as written source code before it can be run. Each step of the program from one state to the next requires a program to be run.

1. Writing
2. Preprocessing
3. Compiling
4. Assembling
5. Linking
6. Loading

Writing

Code is written. It may be written directly by a programmer, or perhaps as the output of another program.

Preprocessing

The **Preprocessor**, `cpp` performs textual expansion on source code. It has very restricted abilities, usually no more than conditional expansion based on variables or simple expressions.

Compiling

The **Compiler** program, `cc`, is the most complex piece of building software programs. It takes a source code instruction and transforms it into assembly language instructions. Details of compiler design (parsing, lexing, optimization) are outside the scope of this course.

Assembling

The program as, the **Assembler**, takes assembly language code and transforms it into machine code, usually in object files. In general, this program is very straightforward, just making one-for-one substitutions.

Linking

The final step of building a program is done by the **Linker**, ld. This program merges and resolves multiple object files that have been assembled.

Additionally, the linker will organize the code and wrap it in a format that can be executed by the operating system. This could be PE, COFF, ELF, DWARF, a.out, or any other executable binary format recognized by the operating system.

The linker will fail if it cannot determine an entry point for the program, a place for execution to start. By default, this is the main function.

Loading

Now that the program is built, it must be loaded into memory and executed. When a program is started, a section of memory is set aside for the process. The code is loaded into an executable portion of that memory, and the computer begins executing instructions from the program, one at a time.

When the operating system tries to run a program, it must first open the file that is the program. It reads the first few bytes to determine how to run it. These first few bytes are referred to as the **magic number** for that filetype. For instance, the magic number for packet captures is 0xA1B2C3D4.

If the first two bytes are #! (magic number 0x2321), then the rest of the line is used as an interpreter for the file.

The program starts execution wherever it finds the symbol _start. main is for C programmers, _start is for the OS. This can be manipulated with ld --entry=symbol

_start is an OS function that calls main with the appropriate arguments. It will then exit the program. The _start function does not return. Instead, it sends a signal to the Operating System that the process should exit, via the exit(2) call.

```
void _start(void)
{
    ... // Build the argc and argv variables
    _exit( main(argc, argv) );
}
```

One can write a `_start` function manually, with the appropriate `-nostartfiles` argument to the compiler. Note that the function must call `_exit` or `exit` to terminate the program.

Executable Formats

A program file on-disk needs to be structured in such a way that the OS can load the program's code and data to memory, and then begin execution. The program file thus has a specific format.

There are many executable formats, the most common ones today are ELF and PE COFF.

The file must contain some simple metadata, such as the target platform and necessary features. It also must describe its own contents: which sections should be loaded as executable code, which should be loaded as writable data, how much additional memory is needed for BSS storage, etc.

ELF

ELF executables start with the byte 0x7F, followed by the ASCII characters "ELF". This makes their magic number 0x7F454C46.

UNIX operating systems understand the ELF format, which describes the layout of the program in the file.

ELF Header
Program Header Table: Describes the segments
Section Header Table: Describes the sections
Data: Referred to by the tables

`readelf` can be used to extract data from an ELF binary.

PE COFF

The PE COFF executable format is used on Windows. All windows binaries start with the ASCII `MZ` as their first two bytes, which is actually an MS-DOS magic number of 0x5A4D.

DOS Header
PE Header
COFF Header
PE Optional Header
Section Table
Code Sections
Data Sections

In Windows, the programmer may code a separate entry point, depending on the kind of program:

`/SUBSYSTEM:CONSOLE`

`main`

`/SUBSYSTEM:WINDOWS`

`WinMain`

`/SUBSYSTEM:NATIVE`

`NTProcessSStartup`

But Windows still begins at `_start`.

Sections

All binaries share a layout of sections. These sections may be composed of data or code. Sections marked as containing executable code will form part of the process image when the program is loaded. Sections marked as containing data may also loaded into the process image.

There are so many potential sections and labels that a carefully-linked program may omit some sections from the loaded image to be run, to be loaded on demand by the program.

All these sections must be marked and described, which is where the **Section Table** in a binary comes into play. This is a table that describes each section: how large it is, what kind of data it contains, and how it should be loaded to form a process image.

Exercises

Exercise 1

The `ld` program requires all libraries to be explicitly passed to it. An all-in-one compiler like `gcc` will provide some “free” libraries, such as the standard C library (`libc`) as well as an OS-specific loader. Determine how to invoke `ld` directly on a simple `hello.o` file.

Exercise 2

Write a program with a different entry point (use the compiler flag `--entry=<func>`). What happens when the program is started in a debugger?

Chapter 3. Tools

`hexdump` or `xxd`

`hexdump` is an application that displays file contents in hex. It can be extremely useful to view binary files. `xxd` serves a similar function, and can also be used to read in hex descriptions of files that are converted to bytes.

`strings`

The `strings` program will extract all ASCII strings embedded in a binary. This is useful for reverse engineering programs. This only works on plain strings; it is not too difficult to have a string evade detection by this program: Splitting the string up into non-adjacent cells in memory, rotating its bits, bitwise-xor on the characters, etc.

`nm`

Extracts the symbol table from a program or object file. Every function call or static variable can be found. Even debugging symbols can be extracted, if present.

`objdump`

`objdump` (On OS X, `otool`) provides information about a program or object file. This is used to examine assembly code, see section information, function calls, and runtime libraries. It has many options for extracting or printing specific output, but the most common one for this curriculum will be `-D -M intel` for disassembling the machine code into assembly language.

When working with a trusted binary, `ldd` is a reasonable program to get a list of needed shared libraries. However, `ldd` works by **running the program**, and thus must be considered a security risk on any binary that you did not build yourself.

`objdump` can provide the list of shared libraries with `objdump -x binary | grep NEEDED`. This is much safer than running `ldd`.

`gdb`

`gdb` is the GNU Debugger. `gdb` is programmable to a large extent, and can have custom functions written for it in a simple language. Many options can be set using the file `~/.gdbinit`. Ensure that the following options are set for this course:

`.gdbinit` *Additions*

```
set disassembly-flavor intel
set history save on
```

In addition, it can be helpful to use the windowed interface to `gdb`, called the TUI.

1. Start `gdb`
2. Enable the TUI with `tui enable` or `C-x a`
3. Show the disassembled source with `layout asm`
4. If desired, the registers can be shown with `layout regs`
5. Use the `C-x o` key combination to switch between window focus
6. Disable the TUI with `tui disable` or `C-x a`

To see other sets of registers available, issue the commands \cmd{tui reg vector} or `tui reg float`.

The TUI will cause mangled output when the debugged program prints to standard output or error. Refreshing the screen (via `C-L`) **usually** fixes the issue.

Other executable debuggers exist; some graphical, some are console-based. `gdb` is distributed with GCC, so it is highly available, and also happens to be extremely powerful.

`disassemble func`

Dumps the assembler code for `func`

`x func`

Dumps the assembler code for `func`

`p reg`

Prints the value of the `reg` register.

For printing register values, `gdb` provides four “generically” named registers, (which usually shadow platform-specific registers) in addition to the platform-specific registers.

`pc`	Program Counter
`sp`	Stack Pointer
`fp`	Frame Pointer
`ps`	Processor Status

When a process is running, it is possible to attach `gdb` to it.

`$ ./process & [1] 75309 $`	`$ gdb process 75309 (gdb) p $pc 0x400408`

`ltrace` and `strace`

Without resorting to a debugger, in can be useful to see what functions a program is calling. The `ltrace` program can run a program and will provide a listing of all standard C library function calls it makes, along with arguments when possible. `strace` does the same, but for system calls.

`make`

If the compiler is clang or GCC, make sure that the following items are set in the Makefile:

```
ASFLAGS += -W
CFLAGS += -O1 -masm=intel -fno-asynchronous-unwind-tables

%.s: %.c
	$(CC) $(CPPFLAGS) $(CFLAGS) -S -o $@ $^
```

The `ASFLAGS` variable is an implicit variable used by `make` when building .s files into object or executables. The only one set is `-W`, which enables warnings.

The `\%.s: \%.c` pattern rule says that a .s target can be built from a .c file of the same name: `minpath.s` could be made from `minpath.c`.

The `-S` flag is what tells the compiler to generate assembly code output, as opposed to object or executable files.

`-O1`
: Set the optimization level to 1

`-masm=intel`
: Set assembly language syntax to be Intel rather than AT&T

`-fno-asynchronous-unwind-tables`
: Remove call frame information from output

`as`

Finally, an **assembler** is needed. There are many assemblers that will turn human-readable assembly code into machine code or object files. The GNU Compiler Collection distributes `gas`, the GNU Assembler, and thus is a very common assembler. `nasm`, the Netwide Assembler, is extremely popular as well. In Visual Studio, the Microsoft Assembler `masm` is available.

This course will use the GNU Assembler, `gas`.

- It is available as part of GCC, with no separate install.
- It is ported to a large variety of platforms.
- `gas` has built-in unwind directives for x64 architecture.
- The drawbacks of `gas` (poor 16-bit executable segmentation, certain output formats) will not be encountered.

Each assembler may have different directives, or ways of controlling their output. The directives covered here are applicable to other assemblers, but may go by different names. Similarly, certain parts of syntax may be slightly different: comments, constants, labels.

Almost Hello World

Open a file named `returnval.s` and enter the following program:

```
.intel_syntax noprefix

.globl main
main:
    mov eax, 42
    ret
```

The program should be compiled and run with:

```
$ make returnval
$ ./returnval ; echo $?
42
$
```

The `$?` shell variable is used to extract the program's return code. Compare this to the following C program:

```
int main(void)
{
    return 42;
}
```

```
$ make returnval-c
cc -O1 -masm=intel -fno-asynchronous-unwind-tables returnval-c.c -o returnval-c
$ ./returnval-c ; echo $?
42
$
```

The assembly code generated by the C can be seen with the following compilation:

```
$ make returnval-c.s
$ cat returnval-c.s
```

The compiler may add some additional parts, but the core `mov` and `ret` instructions will be there.

Explanation

The first line, `.intel_syntax noprefix`, tells the assembler that Intel Syntax (not AT&T syntax) will be used, and that no prefix sigils are required for registers or constants. This will be a standard line for all code in this course.

The `.globl main` directive tells the assembler that the symbol `main` exists, and should be made visible to the linker. When the linker looks for the `main` function as an entry point, the object file will have a reference to that symbol.

`main:` with a colon tells the assembler the location of the `main` symbol; where that symbol lives in memory. This will be the equivalent of the `main` function.

The heart of the program is the line `mov eax, 42`. This moves the value 42 into the `eax` register, the **accumulator**. This sets the value to be returned from a function, as the line `ret` returns from a function.

NOTE For x86 assembly programming, the return value from a function is in the accumulator.

Hello World

To invoke an actual "Hello World" program, a little bit more is needed.

```
.intel_syntax   noprefix

Greeting:
    .asciz  "Hello World!"

.globl  main
main:
    lea rdi, [rip + Greeting]
    call    puts
    mov eax, 0
    ret
```

A null-terminated string is created and stored in memory using `.asciz "Hello, World!\n"`. The `Greeting:` label indicates where this string is stored in memory. Note that it is not marked as `.globl`, it does not need to be visible to the linker.

`call puts` calls a function called `puts`. The linker automatically links the `libc` library which contains that function.

Parameters must be passed to the function; the first argument to any function will be found in the `rdi` register. `puts` expects that first argument to be of type `char *`. So, the address of `Greeting` needs to be placed into `rdi`.

The `lea` instruction will place an address into its destination. The `Greeting` cannot be referred to directly; instead the assembler is going to calculate where it is in relation to the currently executing instruction (`rip`). So, the `[rip + Greeting]` will determine the final location in the binary of that string.

NOTE In x64 assembly programming, the first argument to any function is found in the destination index, `rdi`.

Exercises

Exercise 1

Write a program whose return code is 97 in assembly.

Exercise 2

Implement the `true` program in assembly code.

Exercise 3

Write a program in assembly whose return code is the number of parameters on the command line (i.e., `argc`).

Exercise 4

Read up on `ldd` arbitrary code execution.

Write a C program whose return value is the number of characters in the name of the program. Compile this program to assembly code. What instructions and directives can still be removed such that the program still assembles?

Chapter 4. CPU

The CPU is the part of the computer that executes all code. There are a few specific varieties of CPU that one should be aware of.

RISC Architecture

The most straightforward CPU is known as a **Reduced Instruction Set Computer**, or RISC CPU. A RISC architecture is characterized by a set of CPU instructions that use a small, often fixed number of cycles to execute each instruction. The word “Reduced” does **not** refer to the number of possible instructions, but rather to the amount of work executed by any given instruction.

Such CPUs are fairly easy to target for compilers, and tend to have low power requirements. It is also easier to run real-time operating systems on a RISC CPU.

CISC Architecture

The x86 architecture is a **Complex Instruction Set Computer**, or CISC. Some instructions may take ten or more cycles to complete, while some only take one. As a result, it is harder to predict the performance of a given series of instructions. While the original Intel chips started out with a RISC profile, the additions and backwards compatibility have evolved it into a very CISCy architecture.

The fastest CPUs as of this writing are CISCs, largely due to the investment of Intel and AMD.

GPU

A **Graphics Processing Unit** or GPU is a CPU specialized for 3D graphics tasks. This involves large amounts of floating-point multiplication and division, and so is heavily optimized for these operations. Many of these operations are parallelizable, so GPUs also tend to be built with parallelization in mind. Certain APIs are available for direct interaction with the GPU specifically for offloading computation, notably CUDA, which allows for general-purpose computation on GPUs.

Pipelining

One of the more important concepts in modern CPU architecture is that of the instruction pipeline. Rather than execute exactly one instruction per clock cycle, the CPU makes progress on multiple instructions simultaneously. This is because each instruction has five steps that it must progress through before being complete.

Fetch

Read the instruction from memory

Decode-1

Figure out which instruction is to be executed

Decode-2

Determine arguments to the instruction

Execute

Execute the instruction

Retire

Flush results to registers/memory

The pipeline design allows for multiple instructions to be progressing through these stages simultaneously. While one instruction is being retired, another one can be decoded, for example.

This can lead to some conflicts when it comes to predicting which instruction to be executed next. In a long series of instructions, the next instructions to fetch are obvious. However, in a branching program, it can be hard to determine which path of a branch will be executed.

There are a number of possibilities to solve the problem, such stochastic models of previous times the code has been encountered, following both branches simultaneously (then deciding afterward which one to retire), and some rule-of-thumb guesses.

Modes

Due to backwards compatibility concerns, a x86 CPU has a number of modes of execution, with different capabilities and privilege levels. By the time an operating system is loaded and running, one of these modes has already been entered.

Real Mode

Real Mode for a x86 CPU means that only 1MiB of addressable memory is available (20 bits). This was the original mode of the 8086. All memory is available to all processes. There is no memory protection, privilege, or multitasking abilities. Only the 16-bit registers are available.

All x86 CPUs start in real mode when reset, and must transition to higher addressing modes.

Protected Mode

Protected Mode allows for 4GiB of addressable memory. This mode was first available for the 286 (but was not as usable until the 386 added some key features). In this mode, the 32-bit registers are available for use, along with instructions that manipulate them.

On a x64 CPU, protected mode is referred to as **compatibility mode**.

Rings

In protected mode, there are four privilege levels, also known as **rings** or **protection rings**. Ring 0 is the most privileged, able to execute any instruction. Ring 3 is the least privileged, restricted from executing certain instructions.

The intent of rings is that the OS kernel operates at Ring 0, device drivers operate at Ring 1, I/O operates on Ring 2, and user processes operate at Ring 3. In practice, for both Linux and Windows, only Ring 0 and Ring 3 are used.

When rings were introduced in x86 architecture, UNIX needed to be portable to chips that did not have more than two rings, so it (and later Linux) did not take advantage of rings other than Ring 0 and Ring 3.

Windows came from DOS, which only had Ring 0 permission levels for many years. As such, Windows only uses Ring 0 and Ring 3.

With virtualization, ring usage has expanded slightly. Some virtualization platforms operate at Ring 1, so as to have high privileges, but not compromise their host OS. The host operating system therefore “operates” at Ring -1, compared to the guest OS.

Switching rings is expensive in terms of latency, an order of magnitude slower than a memory access.

Long Mode

In **Long Mode**, 64-bit registers are available to processes. Long mode can only be enabled at Ring-0 permissions; unless an operating system is running in long mode, processes may not be started in long mode.

The theoretical limit for x64 is 16EiB of addressable memory. At the time of this writing, CPUs have access to 256TiB of RAM, or 48 bits. The x64 architecture is designed so that the addressable amount of memory will be easily scalable without side-effects with new microarchitectures, up to the eventual limit of 16EiB of addressable memory.

Chapter 5. Registers

Registers are word-sized pieces of storage for a CPU. They can be accessed faster than memory. Each instruction executed by a CPU will generally use one or more registers as part of the instruction.

Some registers are only writable by the CPU itself, though the executing program may be able to read them. These are often called **Internal Registers**. It may be dangerous to modify these directly, so the program is prevented from doing so. Examples include the `ip`, or Instruction Pointer, which is the location of the next instruction to be executed. There may even be some registers that are completely inaccessible to the running program.

The majority of registers that programmers are concerned with are **User-Accessible Registers**. These are ones that can be written to or read from during the execution of a program.

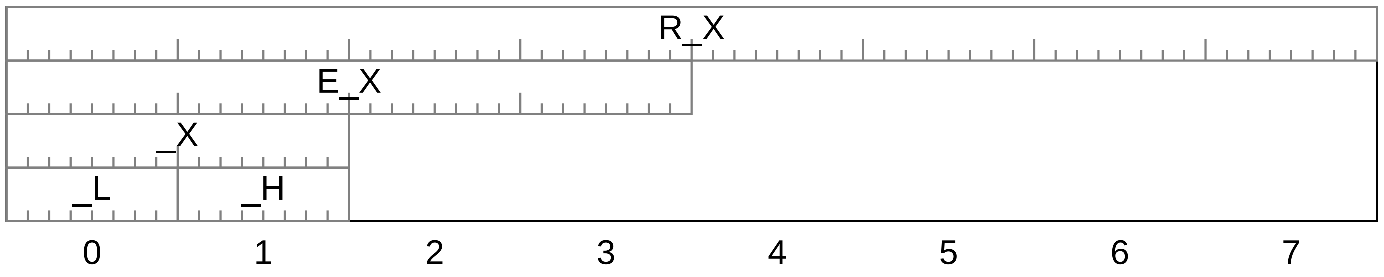
Different CPUs could have different numbers of user-accessible registers. For example, an ARM CPU has 63 user-accessible registers, while a 386/SX has 8.

On certain architectures, these user-accessible registers may be restricted to certain kinds of data; one register might only be used for memory addresses, or a bank of registers might be optimized for floating-point data.

Most CPUs will have **General Purpose Registers**, user-accessible registers that can hold any kind of data: memory addresses, integers, possibly even floating-point data. Instructions that use GPRs normally operate on integers.

NOTE For purposes of this text, the word “register” is intended to mean “general purpose register” unless otherwise specified.

Registers generally hold one CPU “word” of storage, although architectures may have special registers that hold a different amount. They are referred to by their bit-width, such as “64-bit register” or “80-bit register”.



Modern-day x86 registers can be referenced by either their entire amount, or a smaller subset of less-significant bytes. A 64-bit register (usually prefixed with the letter R) also has a 32-bit equivalent that accesses its four least significant bytes. The 32-bit register is prefixed with the letter E, for "`extended'".

That 32-bit register can have the lower 16 bits accessed by not using any prefix. Even the lowest byte may be accessed by suffixing the register in question with the letter L, for “low-order byte”.

Four of the registers (A, C, D, B) allow the second-least-significant byte to be accessed directly with an H, or “high-order byte”, suffix.

Intel Architecture History

The x64 architecture has slowly evolved since 1974, when the 8-bit 8080 chip was produced. Each subsequent design has been so popular that backwards compatibility has been paramount in the design of subsequent chips. By knowing how the architecture has evolved, it is easier to remember how the modern chip operates.

8080

The Intel 8080 8-bit CPU replaced the (binary incompatible) 8008 chip. It had the following 8-bit registers:

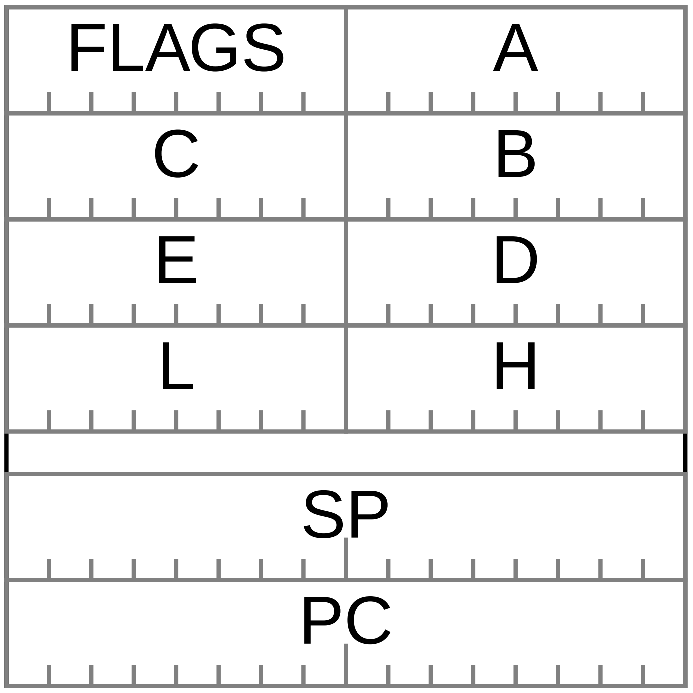


Figure 1. 8080 GP Registers

In addition to the two 16-bit registers, the other registers could be combined into register pairs: BC, DE, and HL. Each register is little-endian: the least significant byte is stored in “front” of the most-significant byte.

8086

The Intel 8086 chip was 16-bit, and contained the following registers:

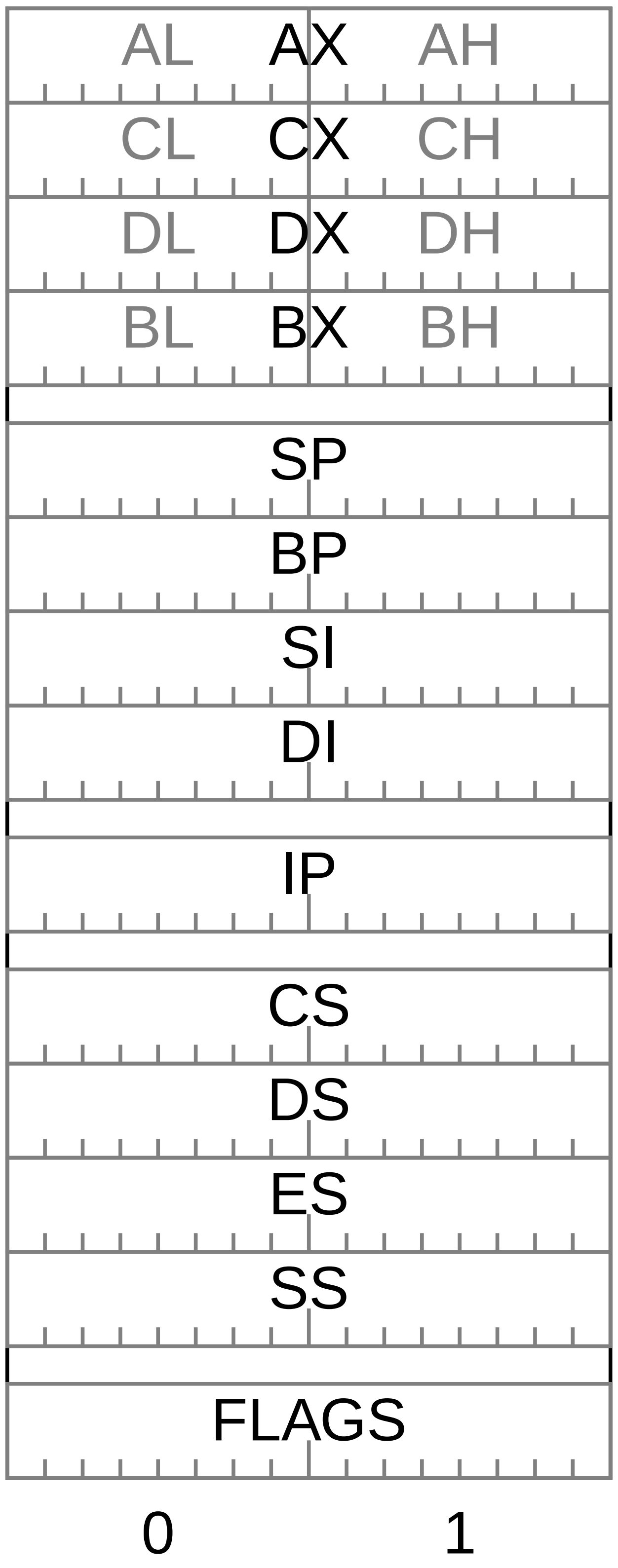


Figure 2. 8086 GP Registers

The data registers could also be accessed as two 8-bit registers, the low and high bits of the given 16-bit register.

This collection of registers persists to this day, even in modern chips.

Segments

With 16 bits, only 64KiB of memory could be addressed. The 8086 allowed a special form of addressing known as **memory segmentation** to address up to 1MiB of memory. One of the **segment registers** (`cs`, `ds`, `es`, or `ss`) is multiplied by 16 and added to a requested memory address.

0xABBA:0xDEAD = 0xABBA × 0x10 + 0xDEAD = 0xB9A4D

```
mov DWORD [es:eax], 99
```

In x86-64 architecture, most of these segment registers are forced to be 0.

386

The Intel 386 chip expanded the word size of the previous 8086 and 286 chips to 32 bits. Intel wanted to still run all programs that were made for its older chips. So, the old registers' names and locations are kept, while also being extended to 32 bits with new names.

The 386/DX also added eight 80-bit registers dedicated for floating-point operation.

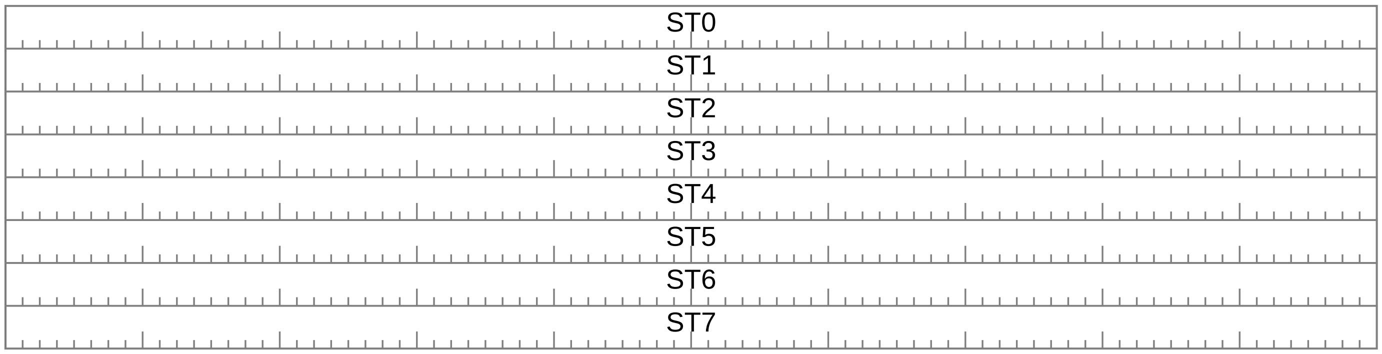


Figure 3. 386 ST Floating-Point Registers

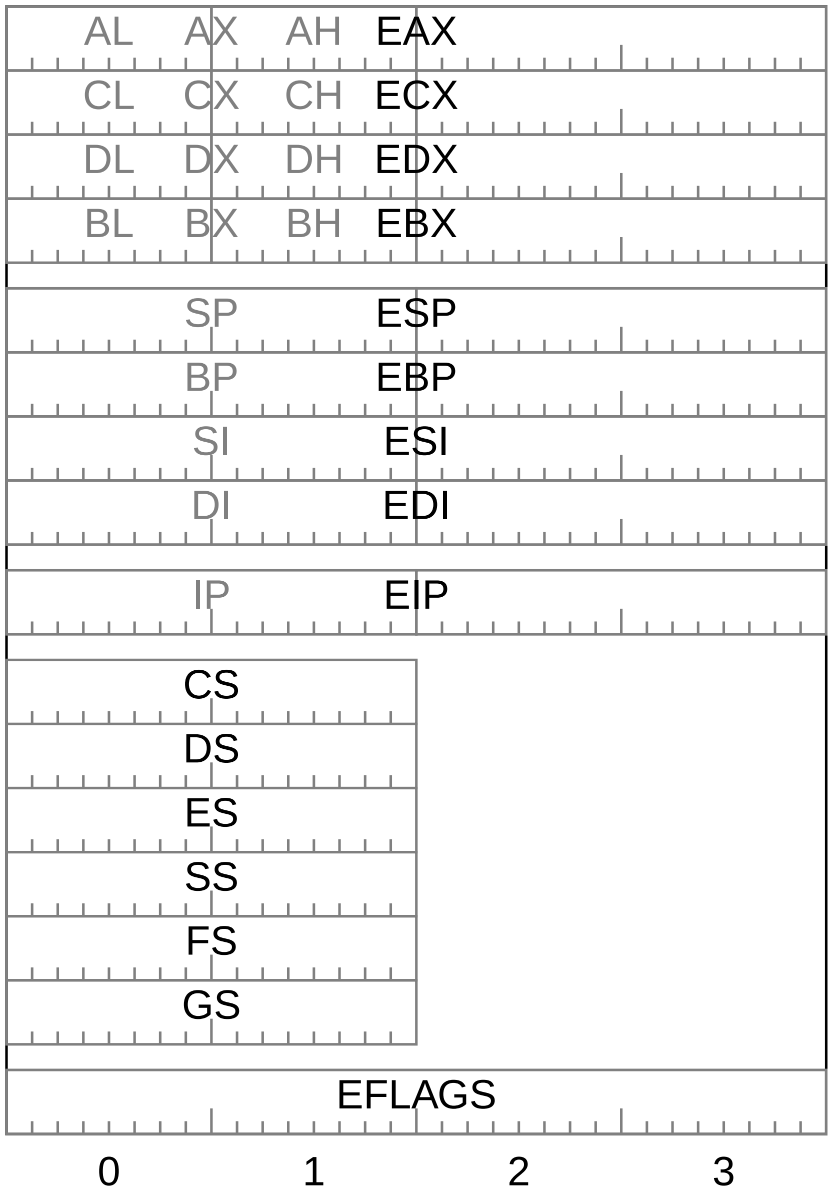


Figure 4. 386 GP Registers

Pentium

The Intel Pentium added parallelizable registers, but these merely overlap with the existing 80-bit ST registers. This means that it was not possible to use both types of registers in the same block of code.

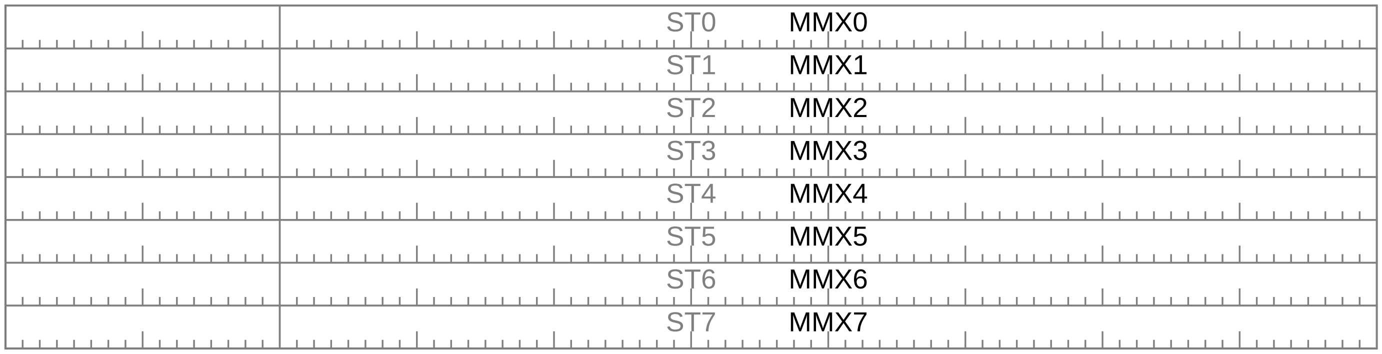


Figure 5. Pentium ST/MMX Registers

These MMX (MMX does not stand for anything, and Intel has gone to court to prove it) registers are actually only 64 bits wide, but originally overlapped the same 80-bit ST registers.

The goal of these registers and their associated instructions is to parallelize common computations. Each register can be used to pack multiple, smaller integers. Then, a single instruction can be applied to 64 bits' worth of integers at once: two 32-bit integers, four 16-bit integers, or eight bytes.

This parallel execution of data manipulation using a single instruction is known as **Single Instruction/Multiple Data**, or SIMD for short.

Pentium III

The Intel Pentium III included additional floating-point registers. These XMM (also does not stand for anything) registers are special SIMD registers that are 128 bits wide, and also have support for SIMD with floating-point values, not just integers. Chip makers have invested a higher proportion of effort into improving performance of these XMM registers, so that at this point they may be more performant than the legacy ST registers.

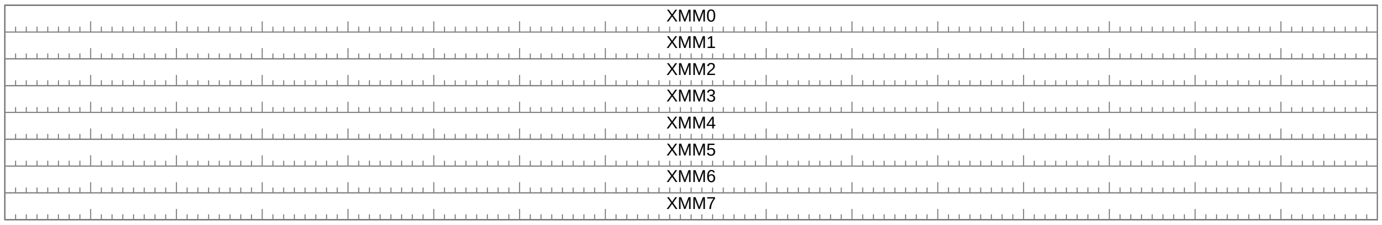


Figure 6. XMM Registers

Itanium

Intel attempted a 64-bit chip known as the Itanium. This was not as backwards-compatible as the AMD Opteron. While some servers might still have IA64 chips, they are not covered in this course.

Opteron

The AMD Opteron was a 64-bit CPU, and further extended the registers in the same manner as the Intel 386.

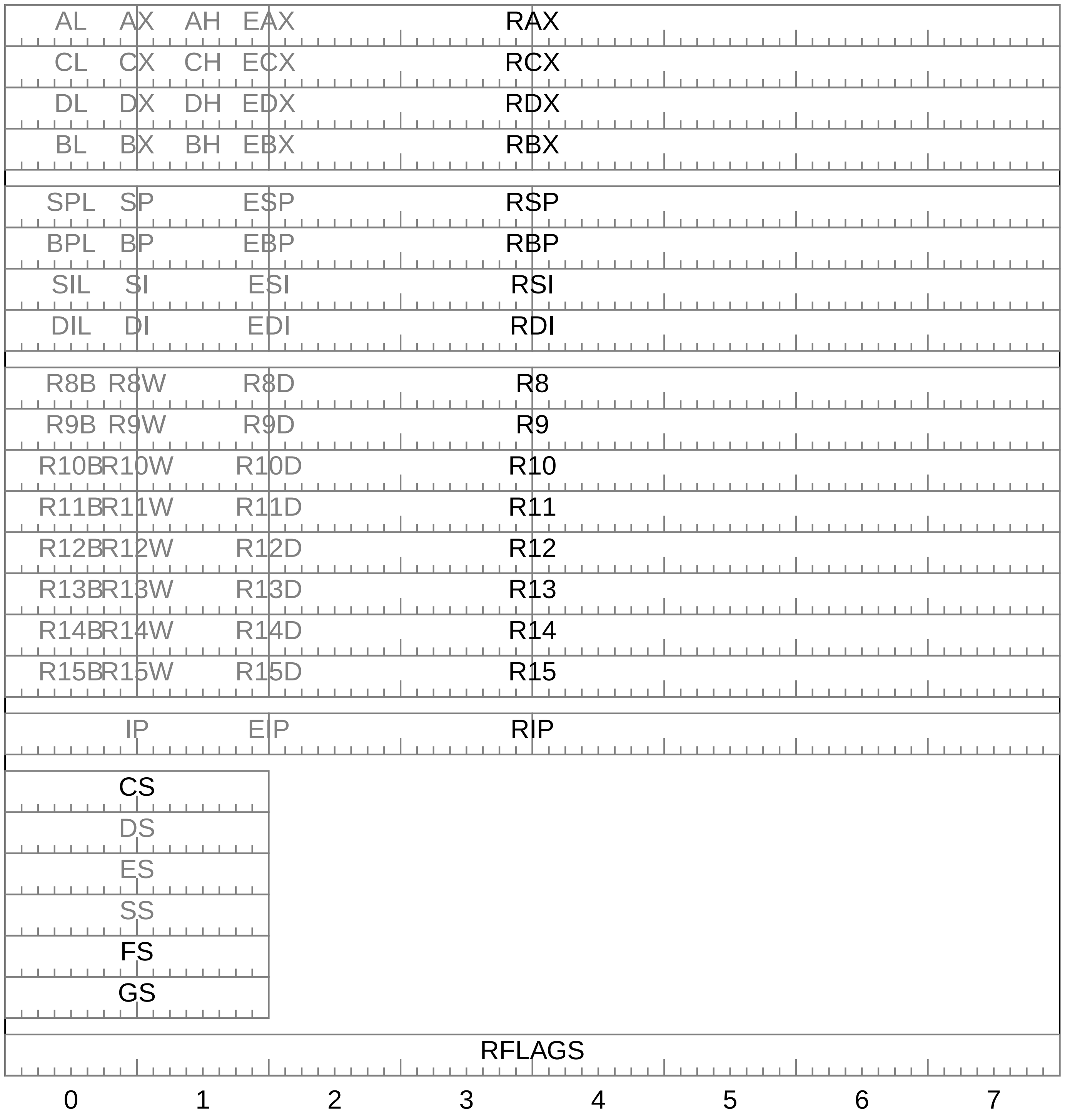


Figure 7. x86-64 Registers

The Opteron also added 8-bit versions of the four "pointer registers". These are rarely used, as they require additional instruction length. It is not unusual for a compiler to copy a 32-bit version of `esi` to `eax`, then use `al`, rather than access `sil` directly!

It also added 8 additional 64-bit registers, the R registers, named R8 through R15. There are also 32-bit, 16-bit, and 8-bit variants, but these variants are not used as often due to increased instruction length.

The Opteron also increased the number of XMM registers to sixteen, following a similar naming system.

The Opteron had enough backwards compatibility that it is now the de facto x64 architecture.

Core i3

For the later Haswell microarchitecture, Intel extended the XMM registers in width, making 256-bit registers known as YMM registers. This was done to extend the SIMD capabilities of modern chips, a technology known as Advanced Vector Extension, or AVX.

Xeon Phi

The most recent expansion has been to extend the YMM registers to a massive 512 bits, known as AVX-512. These ZMM registers allow for a large amount of parallelism via SIMD instructions. Even further, the number of SIMD registers was doubled from sixteen to thirty-two.

Register Descriptions

Instruction Pointer

The **Instruction pointer** register points to the next instruction to be run. This is not a register that is directly writable or readable from most instructions, even if the value can be extracted. Some other architectures may call this the **program counter** or **instruction counter**.

Flags

There are a number of flags that are set after certain operations. While these are in register-like storage in the CPU, they are not accessible in the same way.

Table 2. Flags Register

Bit	Abbr	Name	
0	CF	Carry Flag	unsigned overflow
1			Reserved
2	PF	Parity Flag	same number of 0 and 1 bits
3			Reserved

Bit	Abbr	Name	
4	AF	Adjust Flag	Carry from lower nibble
5			Reserved
6	ZF	Zero Flag	`t == 0`
7	SF	Sign Flag	`t < 0`
8	TF	Trap Flag	Debugging Mode
9	IF	Interrupt Flag	Allow for interrupts
10	DF	Direction Flag	String Processing
11	OF	Overflow Flag	signed overflow
12	IOPL	I/O Privilege Level	I/O Access
13	IOPL		
14	NT	Nested Task Flag	
15			Reserved
16	RF	Resume Flag	
17	VM	Virtual 8086 Mode	
18	AC	Alignment Check	
19	VIF	Virtual Interrupt Flag	
20	VIP	Virtual Interrupt Pending	
21	ID	`cpuid` Permission	
22—63			Reserved

Accumulator

In the oldest computers, the **accumulator** might be the only register at all. All calculations are performed in the accumulator. By convention, all return values are stored in the accumulator.

In x86 assembly language, the accumulator register does have some enhanced capabilities reflecting this heritage. Many basic operations have a special minimal version that works only on the accumulator. Certain special operations must use the accumulator.

Even further, there exist fast operations to move data from memory into the accumulator. The x86 architecture performs best when it runs as many calculations as possible in the accumulator, and the hardware itself is optimized for this kind of use.

Counter

The **count** register closely maps to index variables in loops.

Some highly compact instructions make use of the counter. These instructions deal with loops, iteration, and a number of string manipulation operations. Any instruction or logic that repeats an operation some number of times is best done using the counter.

Data

Calculations in the accumulator may extend themselves into the **data** register. It may also be used to return data structures larger than a single register, but this is platform-specific.

The data register is the most-used general register after the accumulator. It functions as overflow from the accumulator for certain instructions, holding that additional data. It is also optimized for certain I/O operations. Any temporary storage that is not the accumulator should try to use the data register.

Base

The **base** register no longer has a dedicated purpose. In the past, it was the only register that could be used to look up memory addresses. It is now just a general-purpose register.

Stack Pointer

As the program executes, the **stack pointer** holds the address of the top of the executing stack. Instructions that manipulate the stack also manipulate this register.

In x64 assembly, this also functions as the frame pointer.

Base Pointer

In x64 assembly, the **base pointer** is a general-purpose register.

In x86 chips prior to x64, the base pointer served as a current frame pointer; holding the address of the stack where the current function call starts.

Source Index

The **source index** register is an implied source for all memory reads, especially for strings. Most of the time, it is used as a general-purpose register, but there are some customized instructions that work specifically on it.

Destination Index

The **destination index** register is an implied destination for all memory writes, especially for strings. Most of the time, it is used as a general-purpose register, but there are some customized instructions that work specifically on it.

`r8`–`r15`

x64 architecture also has 8 general-purpose registers that are identified by number. They have no special built-in optimizations like the named registers do.

Callee-Saved Registers

For now, try to avoid writing to the following registers: `rbx`, `rsp`, `rbp`, `r11`–`r15`. These registers may be storing values used by the function calling the current function. Overwriting those values may result in crashes or unusual behavior.

Contrariwise, calling a function may result in all other registers being overwritten with new and undesirable values.

Exercises

Exercise 1

Disassemble a C program. Which registers (ignore width differences) are used? Make a histogram of register use.

Exercise 2

A limited number of registers means that certain tricks may sometimes be necessary to perform certain calculations. Determine how to swap two unsigned integer values `alpha` and `bravo` in C without using any other variables.

Chapter 6. Memory and Latency

Main memory is where both the instructions to run the program, and the data it needs, are located. The operating system may move other data in and out of memory, but the program itself generally only has access to memory directly.

```c
#include <stdio.h>
#include <stdlib.h>

int data = 99;
int bss;

int main(void)
{
    int stack = 86;
    int *heap = malloc(sizeof(*heap));

    printf("stack: %p\n", &stack);
    printf(" heap: %p\n", heap);
    printf("  bss: %p\n", &bss);
    printf(" data: %p\n", &data);
    printf(" text: %p\n", main);

    free(heap);
}
```

Structure

For this course, the following offsets will generally imply what lies at a given memory address.

Table 3. Sample Starting Addresses for Process Memory

Stack	0xEFF8
Heap	0x8000
BSS	0x7000
Data	0x6000
Text	0x4000

Addresses beginning with digits 0xC through 0xE will imply data on the stack. Addresses beginning with the digits 8 or 9 will imply data on the heap. If the address begins with a 6 or a 7, it implies static data storage. Addresses beginning with 4 or 5 imply executable code in the text section.

If an address begins with some other digit (0xA, 0xB, etc.), then it is a generic address that may be any one of these sections. Addresses will always be at least 4 hexadecimal digits long.

Stack

The **stack** is the area of memory that keeps track of the currently-executing process. The stack can determine which function is being executed, as well as the parent function. Data and variables local to a function are stored on the stack.

The stack is readable, writable, and may grow.

The lowest-numbered address in the stack is the top of the stack. The stack grows downward from the top of memory. This address of the top of the stack is kept in the register `rsp`, and will change based on the instructions executed.

Heap

Dynamically allocated memory is stored on the **heap**. `malloc(3)` calls return addresses in the heap.

The heap is readable, writable, and may grow.

The heap can be increased via the `brk(2)` system call. This syscall sets the **program break**, the highest address in the data segment. `malloc(3)` maintains a table of memory usage that is initially created by the `brk(2)` call. (In Windows, this is approximateable via the `VirtualAlloc` function.)

Block Started by Symbol

The **BSS** section of a process is an area of static storage. All bits in the BSS section are initialized to 0 and the start of the process. Static variables without an explicit initial value are located within the BSS section. In an object file or binary, this takes up no space, which can be a space savings on disk.

The BSS section is readable, writable, but fixed in size.

Data

In the **data** section of a process, also known as the **static** section, any global or static values that start the process with a specific value are stored.

The data section is readable, writable, but fixed in size.

Text

Also known as the **code** section, the **text** section is where the executable instructions are stored in memory. Each instruction is extracted from this area of memory, and executed by the CPU.

The text section is readable, executable, and fixed in size.

The text section is not writable by default. Having a writable text section means that code could be modified while it is being run. Not only does this make debugging difficult, it is extremely dangerous and difficult to secure. (That said, the `mprotect()` function can modify the permissions for a code page.)

Frame

In x64, frame information is kept outside of the executing stack, and instead managed in extra header sections in the binary.

Red Zone

In the UNIX world for x64 assembly, the 128 bytes beyond `rsp` is known as the **Red Zone**. This area of the stack is scratch space for the current function to use, guaranteed not to be overwritten by asynchronous activity.

Note that subsequent function calls may overwrite this area! The red zone is useful for leaf functions, but care must be used when calling other functions.

Memory Pages

Independent processes are barred from reading each others' memory, let alone overwrite data. However, each process needs to know what memory is available to it.

If a process desires to ``Execute the function at 0x4408", the memory location may already be in use by a different process. Further, there would be no way to tell by examining a program which memory addresses would need to be adjusted.

Multitasking operating systems solve this problem by having all processes execute in **virtual memory**. Every process ``sees" roughly the same labelling of memory, but that memory is mapped differently for every process. The association between the physical memory and the process’s logical memory is managed by the operating system, invisible to the process.

A process that executes `call 0x4408` may actually be executing a function which is located at the physical memory address of 0x1010AF20F808.

The operating system maintains a lookup table, associating process logical memory with machine physical memory. Rather than do this for every byte, it breaks these chunk of memory into **pages**. While these are configurable, both UNIX and Windows have a default pagesize of 4KiB.

4000	0x1010AF20F400
8000	0x1010F00E2300
EF00	0x101000E77000

When a process wants to read or write from memory, the OS first determines if the process in question has access to that segment of memory, by checking the table. A lack of access would result in a **segmentation fault**. When the requested memory is valid for the process, the OS invisibly translates from the virtual address that the program knows (0x8080) to what the physical address would be in memory (0x1010F00E2380).

When more memory is required (from additional function calls on the stack, or heap allocations), the OS may allocate more physical memory to the process, add it to the page table, and then let the process continue. Alternatively, the OS may choose to not allocate more memory (`malloc(3)` returns `NULL`), or end the process (stack overflow).

Modern operating systems generally do their utmost to allow a process to use as much memory as possible.

This page table is only accessible from within Ring-0 permissions.

Caches

Most modern CPUs make use of one or more **caches**, or smaller, high-speed memory storages. A modern x64 CPU is likely to have three caches, referred to as the L1, L2, and L3 caches, although more are possible. The higher the number, the larger and slower the cache is.

Caches are used to hide the latency between memory and the CPU. A CPU may be clocked at 4-5 times the clock speed of memory, and the actual speed of moving data from memory to the CPU may be two orders of magnitude different.

When an instruction requests a location in memory, the CPU queries its closest cache, the L1 cache. If that area of memory exists in the L1 cache, then the value is returned directly from that cache. If the location does not exist in the L1 cache, then the L2 cache is queried. If found, it is loaded into the L1 cache, and from there into the CPU. This works similarly for the L3 or L4 cache and eventually main memory.

When data is to be overwritten in the cache, then the cache is flushed back to main memory. Therefore, while a value may be set in the logical understanding of ``memory'', the actualy RAM chip may not reflect it! Logical memory is distributed between main memory and the various caches.

Caches may be shared between cores or CPUs. L3 cache is commonly shared between all cores, and L2 cache may be shared between just a pair of cores. Increasing the amount of cores that share the cache increases the latency of accessing the cache; so faster caches are generally local to just one core, as is L1 cache.

On the x86 architecture, there is no direct access to any of the caches. While some other chips may allow access, the x86 caches are designed to prefetch areas of memory according to common usage patterns. It is possible that this can lead to ``cache thrashing", where the data loaded is too big to fit in the cache, and thus is constantly loaded and stored to main memory.

Data structures should endeavor to be as cache-friendly as possible. The cache works best when operating over data that reside near each other in memory, so arrays are generally the most cache-friendly data structure. Linked lists, hashmaps, and adjacency list graphs are less performant than arrays, bitwise tries, or heaps, even when factoring in the occasional reallocation and copying of array memory.

Cache Advising

It is possible to advise the CPU to load certain portions of memory into caches, but the CPU is free to ignore these suggestions.

`prefetchtn dst`

Load a value that will be used once, and then never again (i.e., will not store the value in caches). This loads the ***dst*** to all caches **greater than** the given tier. `prefetcht1` loads the value to L2 and L3 cache, for example, and any slower caches on the system.

Using these instructions is probably wrong.

In 2011, Linus Torvalds profiled the kernel, and found the the worst performance offender was executing cache prefetches like these when traversing a linked list, a place that arguably should have been perfect for such prefetches. [1]

It is unlikely that one can achieve a performance boost through these cache prefetch instructions.

Latency Numbers Every Programmer Should Know

The traditional model of the computer in programming is that the CPU is fastest, memory is fast, the disk is slow, and networking is slowest. When it comes to assembly programming, a more fine-grained picture is needed, one that shows the difference in magnitude of low-level operations.

Table 4. Latency Numbers Every Programmer Should Know[2]

Operation	Multiplier
Register	1
L1 cache reference	2
Branch mispredict	20

Operation	Multiplier
L2 cache reference	30
Mutex lock/unlock	100
L3 cache reference, unshared	120
L3 cache reference, shared	200
L3 cache reference, modified	300
Main memory reference	400
Context Switch	3 200
Send 1KiB over Gbit network	40 000
SSD seek	200 000
Read 1MiB from sequential memory	1 000 000
Round trip in datacenter	2 000 000
Read 1MiB from sequential SSD	4 000 000
HDD seek	40 000 000
Read 1MiB from sequential disk	80 000 000
Round trip across globe	600 000 000

These are all measured (roughly) in terms of clock speed of a given high-end x86 CPU. Executing a single-cycle instruction is twenty times faster than backing up after mispredicting a branch path.

Note that simpler CPUs that lack caching may have much more predictable access speed. For computers running real-time operating systems, this predictable (albeit slower) speed may be a requirement for timing of industrial process controllers.

Exercises

Exercise 1

Assume that a single register reference takes 3ns. How long does a round-trip packet that goes across the world and back take to complete?

Exercise 2

Write a program in C that allows the user to enter movie titles into a linked list, and then prints them out. Then, replace the use of `malloc(3)` with appropriate use of `sbrk(2)`.

[1] https://lwn.net/Articles/444344/

[2] Peter Norvig: http://norvig.com/21-days.html

Chapter 7. The Two Major Syntaxes

Each assembler program is welcome to define its own syntax for referring to values, registers, memory locations, and instructions. An assembler may also define custom extensions, macros, and the like.

There are two major forms of assembler syntax for Intel architecture: Intel syntax, and AT&T syntax. Note that the assembler translates the assembly code into machine language: Neither syntax is ``faster'', so the choice of one or the other more often comes down to tooling and personal preference.

Intel

Intel syntax was designed by Intel for the microcomputer market. While it hides some technical detail, it is very easy to read. It is erroneously referred to sometimes as NASM. NASM is actually a popular assembler that uses Intel syntax by default.

In Intel syntax, every two-element operation has the following form:

```
INST    dst, src
```

The destination is first, and the source is second.

Move contents of rcx to rax

```
mov rax, rcx
```

If a constant value needs to be referenced, it is written literally. Hexadecimal, decimal, octal, and even binary are all valid ways of writing a value.

Ways of writing literal values

```
mov cx, 0x15
mov cx, 21
mov cx, 025
mov cx, 0b10101
```

Values in memory are referred to using square brackets. The expression in the square brackets is translated into a memory address.

Move the four bytes at memory address 0x1000 to eax

```
mov eax, [0x1000]
```

Arguments

There are three kinds of arguments to x86 instructions: Immediate values, register values, and memory references. The size of such an argument (when relevant) is referred to in bit-width.

Immediate values are literal values, like 12 or 0xABBA. These may be values to copy or manipulate, but may never be destinations. When referred to in descriptions, either `i` or `imm` might be used, such as `imm32` or `i16`, though `imm` is far more common.

Register arguments are any allowable register for that instruction. Some instructions may use any general-purpose register; some are more restricted; and some allow for other registers. Instruction descriptions just use `r` as an indicator, such as `r32`.

Memory locations may also be arguments to instructions. Their syntax is more complex when it comes to specifying a memory location, known as Scale-Index-Base, or SIB addressing. When being noted in an instruction description, a `m` prefix is used to note the possible width, such as `m16`.

This course material takes a more descriptive approach to describing arguments, but most references will use terse abbreviations. Since a given instruction might allow for multiple possible widths or argument types, an instruction description may look like any of the following:

```
sar r/m16, imm8
```

This `sar` instruction takes a register or memory first argument that is 16 bits wide, and an immediate argument that is 8 bits wide.

```
shl r8/16/32/64
```

The `shl` instruction takes one register argument, which can be 8, 16, 32, or 64 bits wide.

AT&T

AT&T syntax is somewhat more verbose than Intel sytax, but is a closer match to the underlying machine code. This syntax is seen more often in POSIX environments. It is also called GAS Syntax, for GNU Assembler.

For most instructions in AT&T syntax, the order of the operands is source first, destination second.

```
INST    src, dst
```

Move contents of rcx to rax

```
movq    %rcx, %rax
```

AT&T syntax has all of its registers prefixed with a % sign.

All instructions in AT&T syntax are suffixed with the size of the destination operation.

`b`	byte
`w`	word (16 bits)
`l`	long/doubleword (32 bits)
`q`	quadword (64 bits)

In the previous code listing, the destination `%rax` is 64 bits wide, so the instruction is `movq`.

Any literal values in AT&T syntax are prefixed with a $ sign.

Move immediate values in AT&T syntax

```
movw    $0x15, %cx
movd    $10, %eax
```

If a number appears without a dollar sign in AT&T syntax, it actually refers to a memory address.

Move the four bytes at memory address 0x1000 to eax

```
movl    0x1000, %eax
```

Current Use

Intel style is far more prevalant in Windows culture. AT&T syntax is the default output of GCC, so it tends to come up more often on POSIX systems.

The difference between these two syntaxes is purely stylistic. Assemblers building the assembly code will produce identical machine code; some assemblers can even take either syntax.

This course uses Intel syntax for the following reasons:

- Intel syntax always has the destination as the first argument, which is more consistent
- Intel syntax minimizes the use of sigils, which are more distracting than helpful
- Intel syntax automatically determines destination size without needing suffixes
- Intel syntax for indirect addressing follows the logical process, rather than the physical one
- Intel syntax is more consistent with `jcc` instruction mnemonics

Every piece of code that can be written in Intel syntax is also writable in AT&T syntax.

Comments

Code comments are either done with C-style multiline comments `/* … */` or by using a hash sign (`#`). Other single-line comments may be used by different assemblers, such as `;`, `@`, and `!`.

Directives

Writing assembly is not just producing a series of instructions; these instructions must also be organized into an executable program file. The operating system's loader must know how to extract the relevant parts of the executable to build the process in memory.

Assemblers allow the programmer to use special commands to organize or mark parts of the executable; these commands are known as **Assembler Directives**.

Directives may emit bytes, move bytes, perform macro expansion, provide hints to the linker, or define which functions the linker may use elsewhere.

The directives available differ based on the assembler, the target architecture, and the target executable format.

The directives covered here are for the GNU assembler `gas`, with a heavy emphasis on Linux targets.

`.`

`.` is a special symbol that always refers to the current address. This is only valid in directive expressions, not in the assembly code itself. Some assemblers may also use `$`.

`.section name[, flags[, type[, size]]]`

The `.section` directive indicates that the next bytes should be placed into the *name* section of the binary. Different sections are available based on the type of binary, but the most common ones are `.text` for code, `.data` for data, and `.bss` for 0-initialized data.

Differently-named sections may also be created, but then the ***flags*** must be set appropriately; different executable formats have different flags available.

COFF Predefined Sections

- `.arch`
- `.bss`
- `.data`
- `.edata`
- `.idata`
- `.pdata`
- `.rdata`
- `.reloc`
- `.rsrc`
- `.text`
- `.tls`
- `.xdata`
- `.debug`

COFF Flags for Custom Sections

`b`	BSS
`n`	section not loaded
`w`	writable
`d`	data
`r`	read-only

`x`
: executable

`s`
: shared section (PE target)

ELF Predefined Sections

- `.bss`
- `.comment`
- `.data`
- `.data1`
- `.debug`
- `.fini`
- `.init`
- `.rodata`
- `.rodata1`
- `.text`
- `.line`
- `.note`

ELF Flags for custom sections

`a`
: allocatable

`w`
: writable

`x`
: executable

`M`
: mergeable

`S`
: Null-terminated strings

Alignment Directives

`.subsection name`

Replace the current subsection with *name*.

`.zerofill count, size, value`

`.fill count, size, value`

Emits *value* a total of *count* times. *size* is the number of bytes of *value* to use. If *size* is more than 8, it is truncated to 8. Additionally, only the lower 4 bytes of *value* are used, and any higher-order bytes are filled with 0.

`.space size, fill`

Emit *size* bytes of value *fill* (default 0).

`.skip size, fill`

Execute *.space* directive.

`.file filename`

Start a new logical file; this directive is for debugging.

`.balign boundary[, fill[, max]]`

Fills up to the next multiple-of-*boundary* bytes with the value of *fill* (which defaults to NOP instructions in executable sections, 0 otherwise). Optionally, *max* may specify that if more than *max* bytes would be written, then nothing is written.

`.p2align boundary[, fill[, max]]`

Fills up to the next multiple-of-$2^{boundary}$ bytes with the value of *fill* (which defaults to NOP instructions in executable sections, 0 otherwise). Optionally, *max* may specify that if more than *max* bytes would be written, then nothing is written.

`.align boundary[, fill[, max]]`

Either execute the `.balign` or `.p2align` directive, depending on the platform (usually `.balign`).

Symbol Manipulation Directives

`.set symbol, expression`

Set the assembler variable *symbol* to the value of *expression*.

`.equ symbol, expression`

Execute `.set` directive.

`symbol = expression`

Execute `.set` directive.

`.size symbol, expression`

Set the size of *symbol* to the value of *expression*. This is not always necessary, but may be helpful to the linker.

Symbol Visibility Directives

`.globl symbol`

Declares *symbol* so that other programs have visibility to it.

`.global symbol`

Execute `.globl` directive.

`.comm symbol, length[, align]`

Like an `extern` declaration of *symbol*, except that the assembler will build such a symbol if no definition is found, using the largest *length* of any `.comm` directive. May optionally be aligned at a multiple of *align*.

`.lcomm symbol, length[, align]`

Declare *symbol} to be a local symbol of \var{length* bytes in the BSS section.

`.protected symbol…`

The *symbol* in question cannot be overridden in other modules.

`.hidden symbol…`

Set *symbol* to be hidden from other components. It will not be directly referenced from other modules.

`.internal symbol…`

This *symbol* will not be referenced from **any** other module, even indirectly.

Integers and Strings

The following directives may be used to emit integer values or ASCII strings. For integers, the byte-ordering is automatically adjusted for the target architecture.

`.byte num…`

A single byte. This can be very useful for emitting precise bytes, such as for making unusual data sections or headers in an executable.

`.short num…`
: Two-byte integer.

`.long num…`
: Four-byte integer.

`.quad num…`
: Eight-byte integer.

`.octa num…`
: Sixteen-byte integer.

`.ascii string…`
: Emit ASCII characters, but without a NUL terminator.

`.asciz string…`
: Emit a NUL-terminated string.

`.string string…`
: Execute `.asciz` directive.

Avoid the following directives. They are platform-dependent.

`.hword num…`
: Half-word

`.word num…`
: Word

`.int num…`
: Integer

Floating-Point Numbers

The following directives may be used to emit IEEE floating-point numbers' bytes.

`.float num…`

`.single num…`
: Emits single-precision floating point numbers.

`.double num…`
: Emits double-precision floating point numbers.

`.tfloat num⋯`

Emits ten-byte-precision (`long double`) floating point numbers.

Miscellaneous Directives

`.err`

Signal error and stop assembly.

`.print string`

Print *string* during assembly.

`.version string`

Create a `.note` section in ELF with the name *string*.

Disrecommended Directives

The following directives should be avoided on x86 architecture.

`.abort`

Stop assembly immediately.

`.line`

Change logical line number.

`.ln`

Change logical line number.

`.ident`

Places tags in output.

Macros

Machine instructions are always explicit. However, assembly code can contain macros to autogenerate certain chunks of repetetive or obscure code.

Macro Code

```
.macro  iter    from=0, to=5
.long   \from
.if     \to-\from
iter    "(\from+1)",\to
.endif
.endm

iter    1, 7
```

Macro Output

```
.long   1
.long   2
.long   3
.long   4
.long   5
.long   6
.long   7
```

`.irpc symbol, value…`

Iterate, replacing character, from the `.irpc` line to the `.endr` line, assigning the value specified to that symbol each time. The symbol can be referenced with a backslash.

`.irp symbol, char…`

Iterate, for each comma-separated item, from the `.irp` line to the `.endr` line, assigning the item to that symbol each time.

`.endr`

End the `.irpc` directive.

IRPC Code

```
.irpc   temp,89
    test    r\temp, r\temp
    je      found_vowel
.endr
```

IRPC Output

```
    test    r8, r8
    je      found_vowel
    test    r9, r9
    je      found_vowel
```

Chapter 8. Assignment

The **mov** instruction moves (copies) data from *src* to *dst*.

dst may be a register or a location in memory. *src* may be a register, a location in memory, or a constant value. However, it is not allowed for both *src* and *dst* to be memory locations.

```
mov DST, SRC

mov rax, rcx    ;  a = c
mov rax, [rbx]  ;  a = *b
mov rax, 0x100  ;  a = 256
mov [rbx], eax  ; *b = a
mov QWORD PTR [rbx], 0x100  ; *b = 256
```

For non-immediate source values, the size of the destination is implicit based on the source. When a memory location is the destination of an immediate move, the amount of memory must be specified.

When a memory location is used, occasionally the size must be explicitly set, such as with the last `mov` instruction. The assembler may not know how many bytes in memory to use based on the source argument.

1	BYTE PTR
2	WORD PTR
4	DWORD PTR
8	QWORD PTR
10	TBYTE PTR

Since these were based off the historical 16-bit CPU, **word** almost always refers to a 2-byte value, with double-word meaning a 4-byte value, and quad-word indicating an 8-byte value. 10-byte values are for the IEEE `long double` values in memory.

Indirect Addressing

Assembly language for Intel architecture uses the following general structure for accessing memory:

`segment:[base + scale*index + offset]`

This is called **SIB** addressing, which stands for Scale-Index-Base. *base* and *index* are registers, or an omitted 0. *scale* must a constant in {1, 2, 4, 8}, and *offset* is a 32-bit constant. *segment* is a special kind of register known as a segment register. (These registers are completely ignorable in x64 programming, as they are forced to 0. Some tools may show these segments, but that is purely the way an instruction works, not an actual use of the value.)

The registers for *base* and *index* must be general-purpose registers. This is in constrast to RIP-relative addressing.

If a specific memory location is desired, then *offset* may be used as a constant:

```
mov rax, [0x1000]   ; a = *((long long *)0x1000)
```

This loads the 64 bits at 0x1000 into rax.

The constant-lookup into memory does not scale well for large programs that require more than 2GiB of memory. Nor does this work for any program that might store data in non-constant places, such as the stack or heap. So, an address may be put into a register, and then the data in that memory location can be referenced from the *base* register:

```
; b = (long long *)0xBAB1E5EA77AC0CA75
mov rbx, 0xBAB1E5EA77AC0CA75
mov eax, [rbx]  ; a = *b
```

This loads the 64 bits at 0xBAB1E5EA77AC0CA75 into `rax`.

When iterating over a collection of data, it can be useful to know where the data starts, and then only increment an index into that data. This is similar to a for-loop iterating over a string.

```
mov rbx, 0xAFFEC7ED
mov rcx, 1
mov al, [rbx + rcx]
mov rcx, 2
mov al, [rbx + rcx]
```

This moves the byte at 0xAFFEC7EE into `al`, and then moves the byte at 0xAFFEC7EF into that same location.

Note how this could be written into a loop as a single instruction:

```
mov rbx, 0xAFFEC7ED
mov rcx, 13
LOOP:
    mov al, [rbx + rcx]
    sub rcx, 1
    ja LOOP
```

This code copies the bytes at 0xAFFEC7ED to 0xAFFEC7E0 into `al` one at a time.

Since the collection of data may involve members larger than 1 byte, a constant scale factor is allowed. This scaling factor must be 1, 2, 4, or 8.

```
mov rbx, 0xB01DFACE
mov rcx, 0x1
mov eax, [rbx + 4*rcx]
mov rcx, 0x2
mov eax, [rbx + 4*rcx]
```

This would copy the 4-byte word at 0xB01DFAD3 into `eax`, followed by the similar word at 0xB01DFAD7 into the same location.

The scale factor allows loops over arrays of bytes, `short`s, `long`s, and `long long`s.

All of these parts may be combined in various ways.

Examples of Addressing

```
mov rax, [8*rcx + 0xABBA]
mov eax, [rbx + 0xABBA]
mov  dx, [rbx + rcx + 0xABBA]
mov ebx, [rbx + 2*rcx + 0xABBA]
```

AT&T Indirect Addressing

AT&T syntax is quite different, but does reflect the underlying bits encoded into the instruction in a more natural way. It is written as the following:

`segment:_offset_(base, index, scale)`

AT&T Indirect Addressing

```
movq    0xABBA(%rcx, 8), %rax
movl    0xABBA(%rbx), %eax
movw    0xABBA(%rbx, %rcx), %dx
movl    0xABBA(%rbx, %rcx, 2), %ebx
```

While this may be less intuitive to the programmer, it more accurately reflects the underlying bits in the instruction. Just as square brackets indicate a SIB lookup in Intel syntax, parentheses indicate a SIB lookup in AT&T syntax.

Load Effective Address

One of the most important instructions in Intel assembly is **Load Effective Address**, `lea`. Its original purpose is to calculate a memory location using the standard SIB syntax, and place that address into a register:

```
lea eax, [ebx + 4*ecx]  ; a = &b[c]
lea eax, [ebx]        ; a = b
```

This instruction is highly optimized, executing in roughly as many cycles as a standard addition instruction. However, the SIB calculation can be used to combine multiple arithmetic instructions, and so is now used for both calculating memory addresses and simple arithmetic expressions:

```
lea edx, [eax + 4*eax + 3]  ; y = 5*x + 3
lea rsi, [eax + 2*eax]  ; z = 3*x
```

Segment Offsets

A segment register may modify a memory location. The value in the segment register is multiplied by 16 (A 4-bit shift), then added to the offset to create the new value.

Segment registers were added when users wanted to address more than 64KiB of memory, but registers were only 16 bits long. The use of segment registers in this manner allow up to 1MiB to be addressed, without building larger registers.

On x64 architecture, the CS, SS, DS, and ES segments are all forced to 0 and may not be used. The FS and GS segments may be used for OS-specific purposes, but will not be used in normal assembly language.

RIP-Relative Addressing

RIP-Relative addressing was introduced with x64 assembly. It is a special form of addressing memory relative to `rip`, the 64-bit instruction pointer. It may only take a constant offset, but ends up being extremely efficient due to its small size as an instruction.

RIP-Relative Addressing

```
mov rax, [rip + 0xBABE]
```

Note that `rip` is **not** a general-purpose register! This is what distinguishes RIP-Relative addressing from Indirect Addressing. The syntax in assembly language looks identical to Indirect Addressing, however.

Sign Extension

When a small register contains a signed value, it may need to be sign-extended into a larger version of that register. The **convert** instructions accomplish this. These functions only work on the accumulator. The instructions that end in `e` extend the value only into the accumulator, otherwise they use the data register to store the upper-order bits. This can be a fast way to zero out the data register.

`cbw`, `cwd`, `cwde`, `cdq`, `cdqe`, `cqo`

```
    ; signed byte a

cbw ; int16_t A = a
cwd ; int32_t D_A = A
cwde    ; int32_t EA = A
cdq ; int64_t D__A = EA
cdqe    ; int64_t RA = EA
cqo ; int128_t D___A = RA
```

`movzx`, `movsx`

When data needs to be moved to a larger register, the default for `mov` is to only overwrite the lower bits that would fit the data in question:

```
; a = -1
mov eax, -1
; a = (a & 0xFFFF0000) | 12
mov ax, 12
```

As such, there exists the `movzx` instruction to **move, zero-extending** the destination if needed. The destination must be a register, but any register or memory lookup may be the source.

```
movzx   DST, SRC

movzx   rax, cx     ;  a = c
movzx   rax, [rbx]  ;  a = *b
movzx   eax, cl     ;  a = c
```

The only exception is for the upper 32 bits of 64-registers destinations. When performing `mov` operations with 32-bit destination registers, the high four bytes will be set to 0. As such, the source will only ever be one or two bytes wide, never four.

```
; DOES NOT ASSEMBLE
; movzx rax, ecx

; Correct
mov eax, ecx
```

For signed values, such an move would need to **move, sign-extending** the most significant bit. The `movsx` instruction takes that role.

```
movsx   DST, SRC

movsx   rax, cx     ;  a = c
movsx   rax, [rbx]  ;  a = *b
movsx   eax, cl     ;  a = c
```

xchg

It can be very useful to **exchange** two values, which can be performed by the `xchg` instruction.

```
xchg    DST, SRC

xchg    rax, rcx    ;  a, c =  c,a
xchg    eax, [rbx]  ;  a,*b = *b,a
```

The instruction is particularly speedy when using the accumulator.

Exercises

Exercise 1

Write a program in assembly whose return code is three times the number of parameters on the command line.

Exercise 2

Implement a program `iam` in assembly code that prints out the name of the program as it is invoked on the command line (i.e., `argv[0]`). The second argument to a function is stored in the register `rsi`.

Exercise 3

Write an assembly program whose return code is the ASCII value of the first letter of the program's name.

Exercise 4

The `lea` instruction can be used to multiply a register by a number of different constant values, rather than using `mul`. Using just two `lea` instructions and one register, which constant multipliers of that register are possible?

Exercise 5

What value is returned by this program? Why?

```
main:
    mov al, [rip]
    ret
```

Chapter 9. Arithmetic

The accumulator is specialized as a destination for any arithmetic operation. It should be used as a destination whenever possible for both short instruction codes and for speed. Many compilers will generate seemingly unecessary moves to the accumulator just to take advantage of its speed in simple arithemetic.

Second only to the accumulator is the data register. Many operations may involve a combination of the accumulator and data registers, where the accumulator forms the high bits of the operation, and the data register the lower-order bits.

No Operation

There exist instructions that do absolutely nothing, called **NOP**s. There multiple NOPs of various lengths, from one to eight bytes.

The various lengths of NOP exist because there is overhead involved with decoding an instruction, even a NOP. A processor **might** be able to decode a single eight-byte NOP faster than eight one-byte NOPs.

Which NOP should be used is a function of both the processor, the offset of the instruction, the current instruction pipeline, and any local jumps.

The assembler will generally emit the best NOP instruction via an alignment directive. It may emit an actual NOP instruction, or it may emit instructions that amount to a NOP:

Example N-byte NOP instructions

```
nop
xchg    eax, eax
lea edi, [edi]
nop rax
nop WORD PTR [eax]
nop DWORD PTR [eax + eax + 1]
nop QWORD PTR [eax + eax + 1]
lea di, [edi+0]
```

The literal `nop` instruction may take a register or SIB argument, which is unaffected by the instruction. This is a way to emit NOPs of the desired length; however, it is generally better to let the assembler's alignment directive determine the most efficient NOP to emit. Different processors may actually be able to execute an effectively-NOP command faster than a literal `nop` instruction.

Basic Operations

Addition

Addition is straightforward. Given a destination (memory or register), a value may be added to it. A memory destination may only take a register addend; a register destination may take another register, a value in memory, or an immediate value.

```
add DST, ADDEND

add rax, rcx    ;  a += c
add rax, [rbx]  ;  a += *b
add rax, 0x100  ;  a += 256
add [rbx], rax  ; *b += a
```

Add with Carry

A useful form of addition when dealing with large numbers is add-with-carry. This will add together two values (just like addition), but also add in the value of the carry flag.

```
adc DST, ADDEND

adc rax, rcx    ;  a += c + cf
adc rax, [rbx]  ;  a += *b + cf
adc rax, 0x100  ;  a += 256 + cf
adc [rbx], rax  ; *b += a + cf
```

Exchange and Add

This instruction swaps its arguments, then stores the sum in the destination.

```
xadd    DST, ADDEND

xadd    rax, rcx    ;  a += c, c = a
xadd    [rbx], rax  ; *b += a, a = *b
```

Increment

There is also an increment instruction, that adds one to its destination. Note that for memory destinations, the pointer size would always be required.

```
inc DST

inc rax       ;   ++a
inc WORD PTR [rbx]  ; ++(*b)
```

Subtraction

```
sub DST, SUBTRAHEND

sub rax, rcx    ;  a -= c
sub rax, [rbx]  ;  a -= *b
sub rax, 0x100  ;  a -= 256
sub [rbx], rax  ; *b -= a
```

Subtract with Borrow

Subtract-with-borrow is the subtraction version of add-with-carry. It subtracts the subtrahend from the minuend, and then also subtracts the current value of CF, the carry flag.

```
sbb DST, SUBTRAHEND

sbb rax, rcx    ;  a -= c + cf
sbb rax, [rbx]  ;  a -= *b + cf
sbb rax, 0x100  ;  a -= 256 + cf
sbb [rbx], rax  ; *b -= a + cf
```

Decrement

There is also an decrement instruction, that subtracts one from its destination. Note that for memory destinations, the pointer size would always be required.

```
dec DST

dec rax       ;   --a
dec WORD PTR [rbx]  ; --(*b)
```

Multiplication

Integer multiplication has three different forms, and is one of the only instructions that supports up to three arguments.

The simplest form is a one-argument instruction, which must be a register or memory location. The accumulator is multiplied by that factor, and the whole result stored in the data and accumulator registers. The size of the factor determines the size of the result; a product is always twice as wide as its factors (at least, in binary).

`imul` *with One Argument*

```
imul    FACTOR

imul    eax            ; edx:eax =  a * a
imul    WORD PTR [rbx] ;   dx:ax =  a * *b
imul    cl             ;      ax = al * cl
```

A one-byte multiply will store the product in `ax`, rather than `dl:al`.

Alternatively, a destination register may be specified. That destination is multiplied by a factor (which must still be a register or memory location).

`imul` *with Two Arguments*

```
imul    DST, FACTOR

imul    eax, ecx   ; a *= c
imul    eax, [rbx] ; a *= *b
imul    cl, cl     ; c *= c
```

Finally, a three-argument version exists. The factor is multiplied by an immediate value, and stored in the supplied destination register.

`imul` *with Three Arguments*

```
imul    DST, FACTOR, CONSTANT

imul    eax, ecx, 0xF   ; a = c * 15
imul    eax, [rbx], 12 ; a = *b * 12
imul    cl, cl, 7       ; c = c * 7
```

Note that for the two- and three-argument versions, the destination register is the same size as the factor argument, rather than twice as wide. The carry and overflow flags will be set if the product could not fit into the destination successfully. The only way to get the full product is to use the one-argument version.

The `imul` instruction is signed multiplication; `mul` is unsigned. Unsigned multiplication is only available using the one-argument version.

Unsigned `mul` Only Allows One Argument

```
mul FACTOR

mul eax      ; edx:eax =  a * a
mul WORD PTR [rbx]  ;   dx:ax =  a * *b
mul cl       ;      ax = al * cl
```

Division

There is only one form of integer division; a single divisor argument. It must be a register or memory location; immediate values are not allowed.

```
idiv    DIVISOR

idiv    ecx      ; a = ((d<<32) + a) / c
             ; d = ((d<<32) + a) % c
idiv    BYTE PTR [rbx]  ; al = ax / *b, ah = ax % *b
```

The dividend is the the combination of data and accumulator registers; exactly double the bit-size of the divisor. For instance, if `ecx` was the divisor, the dividend would be `edx:eax`. The only exception is a one-byte divisor; in that case the dividend would be `ax`.

Both the quotient and the remainder are calculated; the quotient is placed in the accumulator, and the remainder in the data register. Once again, a one-byte divisor is the exception; the quotient would be placed in `al` and the remainder in `ah`.

There exists an `idiv` instruction is signed division; `div` is unsigned. The divisor's value is always sign-extended for signed division.

Bitwise Operators

Except for `not`, these bitwise operations clear the OF and CF flags, and set the SF, ZF, and PF flags appropriately.

`and`, `or`, `xor`

Logical operators are really bitwise operators.

```
and DST, SRC

and rax, rcx          ;  a &= c
and rax, [rbx]        ;  a &= *b
and rax, 0x100        ;  a &= 256
and [rbx], rax        ; *b &= a
and BYTE PTR [rbx], 0x100    ; *b &= 256
```

```
or  DST, SRC

or  rax, rcx          ;  a |= c
or  rax, [rbx]        ;  a |= *b
or  rax, 0x100        ;  a |= 256
or  [rbx], rax        ; *b |= a
or  BYTE PTR [rbx], 0x100    ; *b |= 256
```

```
xor DST, SRC

xor rax, rcx          ;  a ^= c
xor rax, [rbx]        ;  a ^= *b
xor rax, 0x100        ;  a ^= 256
xor [rbx], rax        ; *b ^= a
xor BYTE PTR [rbx], 0x100    ; *b ^= 256
```

`not`, `neg`

There exists both a bitwise not instruction as well as an arithmetic negation, swapping the sign of the number.

```
not DST
neg DST

not rax       ;  a = ~a
not QWORD PTR [rbx] ; *b = ~*b
neg ax        ;  a = -a
neg WORD PTR [rbx]  ; *b = -*b
```

Shifts

When shifting, the amount to shift by may not be longer than the width of the destination. As a result, an argument to a shift must either be the `cl` register, or a single immediate byte. Shifts are masked to 5 bits on IA-32, 6 bits on x64. For memory destinations, the pointer size must always be explicit.

Shifting without an argument is equivalent to shifting by 1.

`sal`, `shl`

These two instructions are exactly equivalent. If one is directly writing assembly, the goal of the operation (arithmetic or logical) could be made clear by using the appropriate instruction.

```
shl DST
shl DST, SRC

shl rax         ;  a <<= 1
shl DWORD PTR [rbx]     ; *b <<= 1
shl rax, cl         ;  a <<= c
shl BYTE PTR [rbx], cl  ; *b <<= c
shl rax, 0x10       ;  a <<= 16
shl QWORD PTR [rbx], 0x10   ; *b <<= 16
```

`shr`

`shr` stands for Shift Right. So, it will shift, filling the leftmost bits with zeroes.

```
shr DST
shr DST, SRC

shr rax         ;  (unsigned)a >>= 1
shr DWORD PTR [rbx]     ; (unsigned)*b >>= 1
shr rax, cl         ;  (unsigned)a >>= c
shr BYTE PTR [rbx], cl  ; (unsigned)*b >>= c
shr rax, 0x10       ;  (unsigned)a >>= 16
shr QWORD PTR [rbx], 0x10   ; (unsigned)*b >>= 16
```

`sar`

`sar` stands for Shift Arithmetic Right. So, it will shift, extending the sign bit.

```
sar DST
sar DST, SRC

sar rax         ;  (signed)a >>= 1
sar DWORD PTR [rbx]     ; (signed)*b >>= 1
sar rax, cl         ;  (signed)a >>= c
sar BYTE PTR [rbx], cl  ; (signed)*b >>= c
sar rax, 0x10       ;  (signed)a >>= 16
sar QWORD PTR [rbx], 0x10   ; (signed)*b >>= 16
```

Exercises

Exercise 1

Write a function `void add42(int *dst)` which adds 42 to the value pointed at by its argument.

Exercise 2

Write a function `int triple_product(int x, int y, int z)` in assembly that multiplies its arguments together and returns the result. **The third parameter to a function is stored in `edx`.**

Exercise 3

Write a function `int quintuple(int x)` in assembly that returns the value of its argument times five.

Exercise 4

Write a function `bool is_odd(unsigned int x)` that determines if the argument is odd.

Exercise 5

Write a function `void product(int x, int y, int *res)` in assembly that multiplies the first two arguments and places the result in the third.

Chapter 10. Conditionals

A condition in a high-level language may be composed of multiple parts, such as:

```
if(p->len < 3 && initialized(p)) {
    ...
}
```

In assembly programming, conditionals must be much simpler to reflect the available instructions. More complex conditionals must be broken down into a series of steps and tests.

In x86 assembly, a specific register known as the **Flags Register** stores the results of tests and conditions.

Condition Flags

Condition Flags are set as a result of the most recent arithmetic operation. The combinations of these flags allow for many possible tests between two values. The full set of flags is in the chapter on Registers.

```
add ecx, eax    ; t := c + a
```

Table 5. Condition Flags

CF	Carry Flag	unsigned overflow
ZF	Zero Flag	`t == 0`
SF	Sign Flag	`t < 0`
OF	Overflow Flag	signed overflow

While most arithmetic instructions will set these flags, the `lea`, `inc`, and `dec` instructions will **not** set them.

It is difficult to distinguish the carry flag and overflow flag. The carry flag is set whenever a calculation must “use” the place **past** the most significant bit. The overflow bit is set when the most significant bits of both inputs are the same, but the output is different from them. While the mental shortcut of “signed/unsigned” math is useful, it is not complete.

	CF	**OF**
0b0000 - 0b1000 = 0b1000	X	
0b1001 - 0b1000 = 0b0001		X
0b1001 + 0b1000 = 0b0001	X	X

0 - 8

0 - -8

The most-significant bit needed to execute a borrow, so the carry flag is set. The inputs have differing MSBs, so the overflow flag is not set.

9 - 8

-7 - -8

The inputs have the same MSBs, and the output has a different MSB, thus the overflow flag is set. However, this calculation did not require the MSB to borrow, so the carry flag is not set.

9 + 8

-7 + -8

The inputs have the same MSBs, and the output has a different MSB, thus the overflow flag is set. Also, the addition causes would require a carry **past** the MSB of the result, so the carry flag is set as well.

cmp

The `cmp` instruction compares the two values and sets the flags appropriately, without altering either value. It is equivalent to a `sub` instruction that does not alter `dst`.

```
cmp DST, SRC

cmp rax, rcx    ; t = a - c
cmp rax, [rbx]  ; t = a - *b
cmp rax, 0x100  ; t = a - 0x100
cmp [rbx], rax  ; t = *b - a
cmp ecx, eax    ; t = c - a
```

test

The `test` instruction compares the two values and sets the flags appropriately, without altering either value. It is equivalent to an `and` instruction that does not alter `dst`.

```
test    DST, SRC

test    rax, rcx    ; t = a & c
test    rax, [rbx]  ; t = a & *b
test    rax, 0x100  ; t = a & 0x100
test    [rbx], rax  ; t = *b & a
test    ecx, eax    ; t = c & a
```

Note that the `test` instruction will set the CF and OF flags to 0, since overflow will never happen. In general, a `test` instruction has a slight performance benefit over `cmp`.

Condition Codes

The condition flags can be manually read into a `dst` using the `setcc` instructions. The `dst` should be a single byte register.

```
cmp ecx, eax    ; t := c - a
setne   dl      ; d := t != 0
```

Table 6. `setcc` Instructions

Instruction	Test	Description
`sete`	`ZF`	Equal/Zero
`setne`	`~ZF`	Not Equal/Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setl`	`SF^OF`	Less (signed)
`setle`	`(SF^OF) | ZF`	Less or Equal (signed)
`setg`	`~(SF^OF) & ~ZF`	Greater (signed)
`setge`	`~(SF^OF)`	Greater or Equal (signed)
`setb`	`CF`	Below (unsigned)
`setbe`	`CF | ZF`	Below or equal (unsigned)
`seta`	`~CF & ~ZF`	Above (unsigned)
`setae`	`~CF`	Above or equal (unsigned)

Note that these mnemonics are much more English-readable under Intel syntax than AT&T syntax.

Intel Condition Syntax

```
; Compare C and A.  Set D if C is below A.

cmp ecx, eax    ; t := c - a
setb    dl      ; d := c < a
```

AT&T Condition Syntax

```
; Compare A and C.  If C - A < 0, set D.
cmpl    %eax, %ecx
setb    %dl
```

Condition Code Assignment in C

```
bool gt(int x, int y)
{
    return x > y;
}
```

Condition Code Assignment

```
gt:
    cmp edi, esi    ; tmp: x - y
    setg    al      ; al = x > y
    movzbl  eax, al     ; a &= 0xFF
    ret
```

Jumps

A `jmp` instruction is just a `goto`. The most basic form of jump will jump locally, no more than 32 signed bits relative to the current position (Jumping up to 8 bits relative is extremely efficient). The value is signed, so the jump may take the flow of execution backwards in the program.

Another form of `jmp` may take a register as its destination. It uses the value in that register as the memory destination to jump to.

Finally, a `jmp` instruction may use the SIB form of addressing a memory location. Using `rip`-Relative addressing is an efficient way of encoding most SIB jumps.

```
jmp DST

jmp L6      ; goto L6
jmp rbx     ; goto *b
jmp [rip + 0x100]   ; goto *(LINE + 0x100)
jmp [rbx + 2*ecx]   ; goto *(b[2*c])
```

Conditional Jumps

Basic jumps are augmented by a whole set of `jcc` instructions that jump conditionally, based on the values of the Condition Code flags. Note how similar these are to the `setcc` instructions.

However, the `jcc` instructions may only take a relative destination of one, two, or four bytes. They may not take registers or SIB addressing.

Table 7. `jcc` Instructions

Instruction	Test	Description
`jmp`	`1`	Unconditional/Zero
`je`	`ZF`	Equal/Zero
`jne`	`~ZF`	Not Equal/Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jl`	`SF^OF`	Less (signed)
`jle`	`(SF^OF) | ZF`	Less or Equal (signed)
`jg`	`~(SF^OF) & ~ZF`	Greater (signed)
`jge`	`~(SF^OF)`	Greater or Equal (signed)
`jb`	`CF`	Below (unsigned)
`jbe`	`CF | ZF`	Below or equal (unsigned)
`ja`	`~CF & ~ZF`	Above (unsigned)
`jae`	`~CF`	Above or equal (unsigned)

Conditions in high level languages are implemented in assembly by a test followed by a jump based on the results of the test.

Table 8. Comparisons and Jumps

```c
int absdiff(int x, int y)
{
    int res;

    if (x > y) {

        res = x - y;


    } else {
        res = y - x;
    }

    return res;
}
```

```asm
absdiff:
    mov eax, edi


    cmp esi, eax
    jle .L6
    sub esi, eax
    mov eax, esi
    jmp .L7
.L6:
    sub eax, esi

.L7:
    ret
```

Labels

A **label** is a unique location in the source code. It is a word, such as `.L6`. The location of the label is indicated with a colon.

A label is a placeholder in the assembly source. In machine code, the label location does not take up any space. All references to the label are replaced with the label's memory address.

The assembler will do the calculations necessary to determine the offset for a jump to a label.

A label may be any alphanumeric symbol. Local labels (i.e., not external or exported symbols) are prefixed with a period. As such, almost all function-local labels that need to be jumped to are prefixed with periods to make them local.

Referring to Labels

When a label is an argument to an instruction, it is implicitly treated as a SIB argument.

To refer to the address of the label in an instruction, it must be referred to as `OFFSET`.

```
.greeting:
    .asciz  "Hello, World!"
    ...
    ; char a = 'H'
    mov al, [.greeting]
    ; char c = 'H'
    mov cl, .greeting
    ; char *b = "Hello, World!"
    mov rbx, OFFSET .greeting
```

However, most executables must be written in a position-independent style. Since the `.greeting` location may be located anywhere in the text section of the resultant program, the assembler may be unable to work out what the actual address may be. To combat this, all references to memory locations should use `rip`-relative addressing.

```
.greeting:
    .asciz  "Hello, World!"
    ...
    ; char *b = "Hello, World!"
    lea rbx, [rip + .greeting]
```

Labels as arguments to directives are usually treated as the address of that label.

```
.greeting:
    .asciz  "Hello, World!"
.greet_ptr:
    .quad   .greeting
```

Certain instruction/argument combinations may require a more explicit description of the label. The `OFFSET` modifier may take an appropriate `type PTR` modifier:

- `OFFSET BYTE PTR .L1`
- `OFFSET WORD PTR .L2`
- `OFFSET DWORD PTR .L3`
- `OFFSET QWORD PTR .L4`

Numeric Labels

Local symbols can be used by using numeric-only labels. This is extremely convenient when writing code, as they can be referred to forward and backward:

```
1:  branch 1f
2:  branch 1b
1:  branch 2f
2:  branch 1b
```

```
.L1:    branch .L3
.L2:    branch .L1
.L3:    branch .L4
.L4:    branch .L3
```

When the number is followed by an `f`, it searches **forward** in the code until it finds the next label of that number. A `b` suffix searches **backward** until it finds the numbered label.

Most compilers will not generate numeric labels as output, opting for dotted label names instead.

Unusual returns

Normally, `ret` is enough to return from a function. However, branch prediction can be foiled in certain situations:

- When jumping to a `ret`
- The line immediately after a jump or call instruction

Because these jumps are difficult to predict by the CPU, a common optimization is to extend the `ret` instruction to be two bytes, which is `repz ret`. This is used in K8 AMD architecture.

Another common variation is `ret 0`, used by K10 architecture as a three-byte fastpath.

Some architectures do not require this kind of optimization. As always, check the output of a compiler for the target platform for the best guess.

Conditional Moves

A **conditional move** instruction is a prediction-friendly instruction for special kinds of jumps. The `cmovcc` family of instructions follow the same condition codes asm `setcc` instructions. They move the *dst* argument into *src* if the `cc` condition is true.

```
cmp     ecx, eax    ; if(c > a)
cmova   ecx, eax    ;     c = a
```

The benefit of these instructions is that they are pipelined very efficiently compared to jumps: A jump involves branch prediction, and the cost of misprediction is an order of magnitude difference in speed. A `cmovcc` instruction, however, does not require any prediction, and is comparable in speed to a regular `mov` instruction.

As such, `cmovcc` instructions should generally be used whenever possible.

Table 9. Conditional Moves

```c
int absdiff(int x, int y)
{
    int res;

    if (x > y) {
        res = x - y;
    } else {
        res = y - x;
    }

    return res;
}
```

```asm
absdiff:

    mov eax, edi

    sub eax, esi

    sub esi, edi

    cmova   eax, esi

    ret
```

Not only does the conditional-move code result in less code overall, the CPU can better pipeline the instructions for increased speed. `cmovcc` instructions destinations may only be registers, and the source arguments may not be immediate values.

Table 10. `cmovcc` Instructions

Instruction	Test	Description
`cmove`	`ZF`	Equal/Zero
`cmovne`	`~ZF`	Not Equal/Not Zero
`cmovs`	`SF`	Negative
`cmovns`	`~SF`	Nonnegative
`cmovl`	`SF^OF`	Less (signed)
`cmovle`	`(SF^OF) | ZF`	Less or Equal (signed)
`cmovg`	`~(SF^OF) & ~ZF`	Greater (signed)
`cmovge`	`~(SF^OF)`	Greater or Equal (signed)
`cmovb`	`CF`	Below (unsigned)
`cmovbe`	`CF | ZF`	Below or equal (unsigned)
`cmova`	`~CF & ~ZF`	Above (unsigned)
`cmovae`	`~CF`	Above or equal (unsigned)

Exercises

Exercise 1

Write an assembly function `bool is_vowel(char c)` that returns a true value if the character passed in is an ASCII vowel (upper or lower case).

Exercise 2

Write an assembly function `bool is_leap_year(int year)` that returns a true value if the year passed in is a leap year.

Exercise 3

`const char *high_low(int a, int b)`

Implement an assembly function that returns the string `"High/Low"` if the first number is greater than the second number, `"Low/High"` if the reverse is true, and `"Equal"` otherwise.

Exercise 4

Write an assembly function `char letter_grade(int score)` that returns an appropriate letter grade given a 100-point scale numeric score. Assume that 'A' is 90—100 points, 'B' is 80-89, etc., down to an 'F' score being under 60.

Chapter 11. Loops

A loop in assembly language is a jump backwards in the program.

```
    xor eax, eax
.L4:
    add eax, ecx
    sub ecx, 1
    ja  .L4
```

When the code shown is run, it will add the contents of `ecx` repeatedly to `eax`, decrementing `ecx` each time, until `ecx` is 0. It is similar to the following C code:

```
    int sum = 0;
    do {
        sum += count;
        count -= 1;
    } while (count);
```

The do-while loop is the simplest in assembly language: a single jump that goes back in the program.

A while loop contains an extra test:

```
int sum = 0;
while (count) {


    sum += count;
    count -= 1;

}
```

```
    xor eax, eax
1:
    test    ecx, ecx
    je  2f
    add eax, ecx
    sub ecx, 1
    jmp 1b
2:
```

Jump and Loop Optimizations

CPUs will predict that jumps backward in the code are more likely than continuing: Essentially, that a loop is executed at least twice.

The CPU also predicts that jumps forward are less likely than continuing: These tend to be tests and checks for validity, which are unlikely to fail.

Any form of jump still involves disruptions to the instruction pipeline on a CPU. A common optimization for extremely tight loops is known as **loop unrolling**.

Given a simple loop that is executed multiple times, every time though the loop involves a point of CPU prediction as to whether the jump will be executed or not. This is a complex prediction involving current context, bandwidth, and many other factors. If a CPU context switch happens during the execution of the loop, even more overhead is required to be stored and reloaded.

A loop can be unrolled by explicitly executing each iteration of the loop.

```
for (int n = 0; n < 5; ++n) {
    print(n)


}
```

```
print(0)
print(1)
print(2)
print(3)
print(4)
```

Loop unrolling involves a larger binary, since more code must be written. However, the lack of branching often leads to faster performance at the cost of binary size and code readability.

```
    xor ebx, ebx

1:
    mov edi, ebx
    call    print

    add ebx, 1
    cmp ebx, 5

    jl  1b
```

```
xor ebx, ebx
call    print
mov ebx, 1
call    print
mov ebx, 2
call    print
mov ebx, 3
call    print
mov ebx, 4
call    print
```

A small loop could be unrolled entirely. Larger loops could be partially unrolled, to reduce the number of branches.

```
for(int n = 0;
    n < 100;
    ++n) {

    print(n)



}
```

```
for(int n = 0;
    n < 100;
    n+=4) {

    print(n)
    print(n+1)
    print(n+2)
    print(n+3)
}
```

For the example shown, the tight loop involves 100 branches, while the unrolled loop makes 25.

```
    xor ebx, ebx
1:
    call    print
    add ebx, 1






    cmp ebx, 100
    jl  1b
```

```
    xor ebx, ebx
1:
    call    print
    add ebx, 1
    call    print
    add ebx, 1
    call    print
    add ebx, 1
    call    print
    cmp ebx, 100
    jl  1b
```

Loop unrolling is risky. There is no guarantee of measurable performance improvement. And, worse, it is extremely likely that a bug can be introduced via copy and pasting of code.

MySQL Loop Unrolling Bug

```
static int rr_cmp(uchar *a,uchar *b)
{
    if (a[0] != b[0])
        return (int) a[0] - (int) b[0];
    if (a[1] != b[1])
        return (int) a[1] - (int) b[1];
    if (a[2] != b[2])
        return (int) a[2] - (int) b[2];
    if (a[3] != b[3])
        return (int) a[3] - (int) b[3];
    if (a[4] != b[4])
        return (int) a[4] - (int) b[4];
    if (a[5] != b[5])
        return (int) a[1] - (int) b[5];
    if (a[6] != b[6])
        return (int) a[6] - (int) b[6];
    return (int) a[7] - (int) b[7];
}
```

This unrolled loop (taken from the MySQL project) has a bug on line 14. These kinds of bugs are extremely easy to introduce when unrolling a loop or copy and pasting code.

The most famous form of loop unrolling is known as Duff's Device, which combines unrolling a loop interleaved with a switch statement.

Duff's Device

```
send(to, from, count)
    register short *to, *from;
    register count;
{
    register n = (count + 7) / 8;
    switch (count % 8) {
        case 0: do { *to = *from++;
        case 7:      *to = *from++;
        case 6:      *to = *from++;
        case 5:      *to = *from++;
        case 4:      *to = *from++;
        case 3:      *to = *from++;
        case 2:      *to = *from++;
        case 1:      *to = *from++;
                } while (--n > 0);
    }
}
```

Loop instructions

In the interest of smaller (but not faster) code, there exist specific loop instructions for assembly. These instructions assume that the counter register is tracking the loop index, and will jump to a nearby label after adjusting the counter and testing it.

`loop`

`loop` by itself only checks the result of the count register. While the count is nonzero, it will jump to the provided label.

Equivalent code for `loop HEAD`

```
HEAD:
    ...
    dec c
    jne HEAD
```

`loope`, `loopz`

The two `loopcc` instructions check both the count register and the ZF. Both the count being nonzero and the condition code must be true for the jump to occur.

Approximate code for `loope HEAD`

```
HEAD:
    ...
    je  HEAD
    dec c
    jne HEAD
```

`loopne`, `loopnz`

Approximate code for `loopne HEAD`

```
HEAD:
    ...
    jne HEAD
    dec c
    jne HEAD
```

```
    mov ecx, 99
1:
    mov ebx, ecx
    lea rdi, [rip + .bottles]
    mov esi, ebx
    call    printf
    mov ecx, ebx
    loop    1b
    ret
.bottles:
    .asciz "%d bottles of beer on the wall\n"
```

As these instructions tend to be slower than manually testing and jumping, the `loop` instructions tend to not be emitted by compilers. However, their small size makes them an excellent choice for assembly code programmers.

Exercises

Exercise 1

Write a function `unsigned factorial(unsigned top)` that multiplies all numbers from 1 to `top` and returns their product.

Exercise 2

In mathematics, the Collatz sequence is generated by starting with any positive integer and repeatedly applying the following functions until the value 1 is reached:

Collatz(N_{even}) = N/2
Collatz(N_{odd}) = 3N + 1

Although no exceptions have been found, there is no formal mathematical proof as to whether or not this sequence always ends in 1. Write a function that counts and returns the number of steps to 1 from a given starting number, `int collatz_steps(int N)`.

Exercise 3

Write a function `int sum(short *arr, size_t sz)` that returns the sum of all numbers in the array `arr`.

Chapter 12. Functions

A function is called via the `call` instruction:

```
call    DEST
```

DEST is a memory location or a register containing a location. The location of the next instruction (just after the `call`) is pushed onto the stack, and control is passed to *DEST*. So, a `call` is equivalent to the following pseudocode:

Illustrative demo of `call DEST`

```
push    rip
jmp DEST
```

Note that `rip` cannot be an argument to `push`; this is not actually valid code.

When a function is complete, the `ret` instruction is used to return control to the point in the program that called the function. It determines the location of the call from the former value of `rip`, still on the stack. The `ret` instruction is equivalent to the following code:

Illustrative demo of `ret`

```
pop rip
jmp rip
```

Note that `rip` cannot be an argument to `pop` or `jmp`; this is not actually valid code. (Further, notice that `jmp rip` is conceptually a `nop`.)

The executing stack provides a trace of previous `rip` locations, forming a backtrace of sorts.

Table 11. Stack on x64 POSIX process

0xEFF8	Old Function Saved Data
0xEFF0	Old `rip` value
0xEFE8	Current Function Red Zone
⋮	
0xEF70	Current Function Red Zone

Stack Space

There are a limited number of registers, but the data used in a function may be greater than the amount of register space available. Memory, however, is practically unlimited for well-made functions. Functions can use memory at the top of the stack to store (sometimes called **spilling**) data.

The topmost writable location in the stack is pointed at by the `rsp` register. Data on the stack is referred to by offsets from the stack pointer.

The stack in memory grows “downward”: the top of the stack is closer to the lowest memory address, and the base of the stack is the highest address.

Just like a stack data structure, the stack in an executing program has two special-purpose instructions, `push` and `pop`. Both of these instructions implicitly adjust the stack pointer register.

`push`

The `push` instruction pushes the given register value onto the stack, adjusting the stack pointer downward.

Equivalent to `push VALUE` (assuming width of 4 bytes)

```
sub rsp, 4
mov [rsp], VALUE
```

`pop`

The `pop` instruction extracts the value at the head of the stack (the lowest memory address; remember that the stack “grows” downward), and moves that value into the provided register.

Equivalent to `pop DEST` (assuming width of 4 bytes)

```
mov DEST, [rsp]
add rsp, 4
```

`leave`

The `leave` instruction is a shortcut for the following:

Equivalent to `leave`

```
mov rsp, rbp
pop rbp
```

Base Pointer and Stack Pointer

For a long time, the Base Pointer register `rbp` would store the value of the start of the current function's frame. Upon entering a new function, the old value of the base pointer would be stored on the stack, and the new value would be derived from the current value of the stack pointer `rsp`.

Most 32-bit ABIs require the base and stack pointer registers to be set this way. The preamble and epilogue for almost all functions look like this:

32-bit function outline

```
push    ebp
mov ebp, esp
sub esp, .stack_space
...
add esp, .stack_space
pop ebp
ret
```

This structure allows for the entire call stack to be discoverable and tracable with no additional information. The current function can look up `[ebp + 4]` to find the calling function, and `[ebp]` indicates the base pointer in the stack of that calling function!

Table 12. Stack on IA-32 POSIX process

Address	Value	Contents
0xEFF8	0x4000	`_start` `eip` value
0xEFFC	0xEFF8	`_start` `ebp` value
0xEFF4		`main` Function Saved Data
0xEFF0		`main` Function Saved Data
0xEFEC	0x4040	`main` `eip` value
0xEFE8	0xEFFC	`main` `ebp` value
0xEFE4		Current Function Data
0xEFE0		Current Function Data
0xEFDC		Current Function Data

Table 13. Register Values

`ebp`	0xEFE8
`esp`	0xEFDC

`eip`	0x4080

On x64 systems, this is unnecessary. Debugging information is stored in symbol tables via `.stab*` instructions, and is used to reconstruct the call stack in case of debugging. Some compilers will still emit this preamble, and use the `rbp` register as a base pointer rather than a generic register, but will happily reclaim the register with additional optimizations enabled.

`ret SIZE`

As discussed, the `ret` instruction returns from a function call. It does so by popping the current value from the stack and jumping to that address.

Because the stack is writable, this is common way to commandeer a process. If a local array exists in a function’s stack, then overrunning the bounds of that array would allow nefarious input to overwrite the return address, jumping to any point in the code with any set of arguments desired.

The `ret` instruction may take an optional argument ***SIZE***, which is the number of words' worth of memory to pop from the stack after the function returns. This can be used to clean the stack of arguments to the called function, which is required by certain calling conventions.

Caller- and Callee-Saved Registers

Functions and variable scope are high-level constructs. In assembly language, all data are global. So, some level of coordination is needed so that different functions to not clobber or overwrite important data in registers.

A given platform’s ABI dictates which registers are allowed to be used by a function, and which registers must be restored by the time the function returns.

Registers that may be modified freely are called **Caller-Saved Registers**. The calling function, if it wishes to use their values after a called function is complete, must save them to memory.

Some registers are expected to be in the same state once the called function returns. That does not mean they must be unchanged; just that when the function returns, these **Callee-Saved Registers** must hold the same value as before the function was called. A function may save the current values to memory, use the registers, and then restore them.

Windows and the standard ABI disagree on which registers are caller-saved, and which are callee-saved. Both agree that the accumulator, data, and count registers are caller-saved. Both agree that the base register is callee-saved. The other registers are negotiable, depending on the platform.

x64 ABI

On x64 architecture, the stack must always be aligned to 16 bytes at the start of a function. Before any `call` instruction, the `rsp` must be adjusted to be a multiple of 16, as the `call` will push the return location onto the stack, bringing it to 8 modulo 16 on function entry.

Microsoft x64

In Windows, the first four arguments are passed through specific registers, depending on whether the argument is an integer or a float.

Nth Argument	If Integer	If Float
1	rcx	xmm0
2	rdx	xmm1
3	r8	xmm2
4	r9	xmm3

Additional arguments are passed on the stack.

On Windows, it is the responsibility of the calling function to allocate 32 bytes on the stack prior to the call, called **shadow space**. This space exists between any arguments after the fourth, and the return address. It is used as a kind of scratch space by the function; usually as spill locations for the `rcx`, `rdx`, `r8`, and `r9` registers.

All other registers are callee-saved registers. If a function wishes to use them, they must restore their original value before returning.

Microsoft `__vectorcall`

Microsoft introduced a secondary calling convention, __vectorcall, in 2013. It is identical to the standard Microsoft x64 ABI, except that it allows for two extra registers, `xmm4` and `xmm5`, to be used for additional floating-point or SIMD vector arguments.

System V AMD64

On non-Windows platforms, the number of register-passed arguments is much greater.

Nth Integer	Register
1	rdi

Nth Integer	Register
2	rsi
3	rdx
4	rcx (`r10` in system calls)
5	r8
6	r9

Nth Float	Register
1	xmm0
2	xmm1
3	xmm2
4	xmm3
5	xmm4
6	xmm5
7	xmm6
8	xmm7

Note that the System V ABI can allow for up to 14 arguments being passed by register, assuming that six of them are integers (or pointers) and eight of them are floats. Any additional arguments are passed on the stack.

The `r10` and `r11` are caller-saved. Naturally, if one wants to maintain the value of these or the register-passed arguments between function calls, they must be spilled to the stack.

All other registers are callee-saved registers. If a function wishes to use them, they must restore their original value before returning.

Red Zone

When a function is entered, the next 128 bytes on the stack are guaranteed not to be overwritten by asynchronous activity. This means that the space can be used for spilling registers, temporary variables, and the like. The space will be clobbered by any function calls made. This **Red Zone** is only guaranteed in the System V ABI; Windows will actively overwrite memory above the logical top of the stack.

Table 14. System V x64 Calling Convention Stack

0xEFF8	⋮
0xEFF0	old local vars

0xEFE8	Parameter 8
0xEFE0	Parameter 7
0xEFD8	old `rip`
0xEFD0	local vars
⋮	Red Zone
0xEF50	

Table 15. System V x64 Calling Convention Register Values

`rsp`	0xEFD0

Even though it is not required, an OS may still use `rbp` as a base pointer for the current function, pushing it onto the stack on function entry, just as `cdecl` convention does.

IA-32 ABI

For 32-bit architecture, there is no alignment requirement; in practice, the stack is 4-byte aligned.

Some compilers (GCC 3.x+) will still align the stack to sixteen bytes for function calls, so the stack pointer will always be 12 modulo 16 at function entry.

cdecl

In the standard 32-bit ABI, known as **cdecl**, all arguments are passed on the stack section of memory.

Integer or pointer results are returned in `eax`, floating-point results are returned in `st0`.

Table 16. `cdecl` Convention Stack

0xEFFC	⋮
0xEFF8	old local vars
0xEFF4	old local vars
0xEFF0	Parameter 3
0xEFEC	Parameter 2
0xEFE8	Parameter 1
0xEFE4	old `eip`
0xEFE0	old `ebp`

0xEFDC	local vars
0xEFD8	local vars
0xEFD4	local vars

Table 17. `cdecl` Convention Register Values

`ebp`	0xEFE0
`esp`	0xEFD4

fastcall

The first two integer or pointer arguments are passed in via `ecx` and `edx`. Additional arguments are passed on the stack.

stdcall

stdcall functions pass all arguments on the stack, just like `cdecl` calls. The difference is that every `stdcall` function must perform its own stack cleanup, rather than the function calling it. This is achieved by passing the number of arguments to clean off the stack in the `ret` instruction. Note that this does not support variable-length argument lists.

thiscall

For object-oriented languages, such as C++, the `this` object is passed in via the `ecx` register. All other parameters are on the stack in RTL order.

Return Types

Normally, any integer or pointer return value is in the accumulator.

On x64 architecture, floating-point return values are placed in `xmm0`. Floating-point values on IA-32 architecture are in `ST0`. An IA-32 calling function **must** pop the `ST0` value, even it makes no use of the value.

A structure return may be returned differently based on size. If the size of the structure is no larger than the size of a word, then the structure will be stored and returned in the accumulator. A structure up to double that size will be stored in the accumulator and data registers on return, saving a trip to memory.

Large literal structures as return values are more complex. Generally, a structure of sufficient size (larger than the D:A register pair) will, as part of the calling convention, require a pointer to an area of the stack where the return value may be written. Therefore, it is the responsibility of the calling function to both set aside sufficient space **and** pass a pointer to that space.

On x64 architecture, the pointer to the large structure takes the place of the first argument, and all other arguments are shifted down.

For IA-32 architecture, the space for the structure is the “first” argument on the stack.

```
struct bubble_tea {
    int tea_ounces;
    int ice_ounces;
    int tapioca_ounces;
    int cup_size;
    int flavor;
};

struct bubble_tea make_cup(int cup_size)
{
    struct bubble_tea empty = { 0 };
    empty.cup_size = cup_size;

    return empty;
}
```

System Calls

`int`

On IA-32 systems, any system call is invoked by raising an **interrupt**. A specific interrupt code signals to the kernel that a system call will be executed. All system-level calls are done via this interrupt. The desired function call is in `eax`.

`eax` will not be a pointer to the desired function! Instead, it is a value into the complete lookup table of system calls. On Linux, these system call numbers are usually found in `sys/syscall.h`.

Determining the system call number can be difficult, as they are often includes within includes, the final number usually living inside `unistd.h`. Try `for call in write read; do printf "#include <sys/syscall.h>\\n$call SYS_$call" | cc -E -P - ; done`, substituting in the names of the desired calls.

```
mov eax, 0xEF
int 0x80
```

In Windows, the desired function is also in `eax`, but a different interrupt value is used. Windows does not in any way guarantee consistent syscall numbers between versions or even service packs. `ntdll.dll` is the source of these syscall numbers.

```
mov eax, 0x3D
int 0x2E
```

Prior to Windows XP, a 32-bit Windows system would use `int` to transition to Ring 0.

Windows XP and higher used a Pentium II instruction pair, `sysenter` and `sysexit`, to execute system calls. While this does require more code to be written, it is faster to transition to Ring 0. A chip without the instruction would still use the old `int` instruction. Windows got around this issue by having a special subroutine for system calls, `KiFastSystemCall`.

```
...
    mov eax, 0x3D
    mov edx, OFFSET KiFastSystemCall
    call    [edx]
    ret 8

; Pentium II
KiFastSystemCall:
    mov edx, esp
    sysenter
    ret

; Older Chips
KiFastSystemCall:
    lea edx, [esp + 8]
    int 0x2E
    ret
```

`KiFastSystemCall` is a location in memory, 0x7FFE0300, that executes the actual system call. It could still be using `int 0x2E` instead of `sysenter`.

Windows expects `edx` to contain a reference to array of arguments (for `int 0x2E`), or the return address followed by the array of arguments in the case of `sysenter`.

`syscall`

For x64 architecture, `syscall` is used instead of `int` or `sysenter`.

With a `syscall`, the Counter register temporarily stores the instruction pointer, and `r11` stores a copy of the flags. Since `rcx` is used for the instruction pointer, it is not available for use as a parameter. Instead, `r10` is used in its place.

```
mov eax, 0x3D
syscall
```

Exercises

Exercise 1

Given the following IA-32 stack, reconstruct the call chain of functions. Assume `cdecl` calling convention.

Table 18. Exercise 1 Call Stack

Address	Value
0xEFFC	???
0xEFF8	0x0003
0xEFF4	0x4040
0xEFF0	0xEFFC
0xEFEC	0x0003
0xEFE8	0x4080
0xEFE4	0xEFF0
0xEFE0	0x0002
0xEFDC	0x4100
0xEFD8	0xEFE4
0xEFD4	0x0002
0xEFD0	0x4080
0xEFCC	0xEFD8
0xEFC8	0x0001
0xEFC4	0x40C0
0xEFC0	0xEFCC
0xEFBC	0x0001

Table 19. Exercise 1 Register Values

`ebp`	0xEFC0
`esp`	0xEFBC
`eip`	0x4100

Exercise 2

Write a function `void times10(char *s)` that prints out the given string ten times.

Exercise 3

Write a program that inputs two strings from the user and outputs the strings with the first two characters of each word having been swapped. You may assume that the words entered are at least two letters long.

Exercise 4

Write a program that inputs three numbers from the user and prints out their product. Include error-checking.

Chapter 13. Structures

Assembly language has no concept of structures in the same sense that higher-level languages do. All it understands is blocks of memory. Thus, all structures in assembly are just a slighly larger amount of memory, and the fields within are calculated offsets into that memory.

Different chips and ABIs may treat structures differently based on size. A structure only four bytes wide may be just treated as a four-byte integer.

```c
struct coord {
    short x;
    short y;
} origin = {0, 0};
center(origin);
```

```asm
 ;


xor     edi, edi
call    center
```

There is little higher-level support for groups of bytes in structured format. Extracting a specific field from a structure requires knowing the byte offset of that field. This is why all C code requires complete definitions of `struct` objects before code manipulating them can be written.

Chapter 14. Strings

Since assembly was intended to be written directly early in its life, it has a number of instructions that are easy to program, but hard to run efficiently. While these instructions may be less efficient than a series of `mov` and `test` instructions, these compact instructions take up far less space, and may be more efficient in unusual circumstances.

The most common data structure is a string of characters. Assembly language has some support for “string operations”, common operations to be performed on a string in memory. Using the accumulator, count, and source/destination registers, these operations may copy or scan a series of bytes in memory.

While these instructions involve implicit segment registers, remember that x64 requires those registers to be set to 0.

`cld`, `std`

These instructions are all dependent upon a special flag known as the **Direction Flag**. This flag defaults to 0, and may be set with the `std` instruction, or cleared to 0 with `cld`.

While the flag is 0, all string instructions involve increases to the index registers. When the flag is set, all string instructions will decrease the index registers.

Any real-world code should consider explicitly setting DF to what it needs; there is no guarantee what state it may be in when a function is executed. The following examples assume that DF is cleared to 0.

`lods`, `stos`

These operations are for loading or storing strings. The register in question is the accumulator, and the memory location is loaded from the source index, and stored in the destination index.

After such an operation, the source or destination is increased by the size of the operand, so it is ready to be executed in a tight loop.

Equivalent code for `lodsb`

```
mov al, es:[rsi]
inc rsi
```

Equivalent code for `stosw`

```
mov es:[rdi], ax
inc rdi
inc rdi
```

`scas`

The **scan string** instruction compares a value in memory with the value in the accumulator, like a `cmp` instruction. It is extremely useful when searching an array for a specific value. In fact, it is one of the few string operations that can be faster than generalized moves and tests. The destination index is increased by the size of the comparison.

Equivalent code for `scasq`

```
cmp rax, es:[rdi]
lea rdi, [rdi + 8]
```

`movs`

The **move string** operation moves bytes between two memory locations. Unintuitively, this operation is slower than just using two `mov` instructions, to transfer one memory location to a register, and then another to transfer the register to the destination memory location. Both index registers are increased by the size of the data moved.

Approximate code for `movsb` (Note invalid `mov`)

```
mov es:[rdi], es:[rsi]
inc rsi
inc rdi
```

`cmps`

Compare String compares the values at two memory locations, like a `cmp` instruction. Like `movs`, it is much slower than a `mov`/`cmp` combination. It will increase both the index registers by an appropriate amount.

Approximate code for `cmpsd` (Note invalid `cmp`)

```
cmp ds:[rsi], es:[rdi]
lea rsi, [rsi + 4]
lea rdi, [rdi + 4]
```

`rep`, `repz`, `repnz`

A powerful asset to these string instructions is **Repeat** modifier. This prefix instruction will execute the specified string instruction until either the required conditions are met:

`rep`

The count register is 0

`repz`

The count register is 0 OR ZF is 0

`repnz`

The count register is 0 OR ZF is set

Equivalent code for `repz scasb`

```
1:
    test    rcx, rcx
    je  2f
    cmp al, es:[rdi]
    inc rdi
    dec rcx
    jne 1b
2:
```

Note that this is an extremely compact way to express common high-level constructs. By compacting this set of instructions into only a few bytes, it can be very easy to align code to word boundaries, with fewer branches.

Exercises

Exercise 1

Write `int exists(short needle, short \*haystack, int sz)` that will return the first index in array `haystack` whose value is equal to `needle`. If the value does not exist in the array, -1 should be returned.

Challenge: Use at most 32 bytes.

Exercise 2

Implement `strrchr(3)` in assembly.

Exercise 3

Implement `strlen(3)` in assembly, in under 32 bytes.

Challenge: Use at most 24 bytes.

Chapter 15. Floats

Floating-point values are handled in a specific set of registers. The original set of register were ST registers, ST(0) through ST(7). Data would be loaded into and out of these registers to have floating-point calculations be done.

When CPUs added SIMD (Single Instruction, Multiple Data) support, they originally intended for it to handle multiple integer data with a single instruction. However, the register area used was the same as the ST stack! So SIMD and floating-point calculations could not be carried out in the same section of code. Modern-day chips have removed this overlap.

When SIMD space and instructions were expanded, having their calculations also work on floating-point numbers was a natural progression. Now, most calculations involving floating point will be done in the SIMD registers (also known as AVX registers), to take advantage of SIMD use for floating-point numbers.

```
float missle_time = 0.001;
```

ST Use

x87 Floating-point calculations use ST registers. They are a stack of registers, with the top of the stack being ST(0). Most operations involve manipulating ST registers like a stack, pushing and popping data as instructions are executed.

ST registers do not have instructions that interact with non-ST registers, only memory. As such, loading data into these registers may be a little bit slower that integer arithmetic using general-purpose registers.

All x87 FPU instructions are prefixed with the letter `f`.

Any operation carried out on 80-bit floating-point numbers (`long double`) will only take place in these ST registers. So while these may allow for slightly greater precision than 64-bit floating-point numbers (`double`), there is a performance cost. To refer to an 80-bit value in memory, the keyword is `TBYTE`, short for “ten bytes”.

Finally, there are certain operations that are only available as single instructions in the ST registers, which may have a performance benefit.

`fld`, `fild`

`fld` will **load data** from either memory or another ST register and push it onto the stack, at position ST(0). The data is assumed to already be in floating-point format.

```
fld SRC

fld st(3)           ; push(st, get(st, 3))
fld QWORD PTR [ebx] ; push(st, *b)
```

Alternatively, the `fild` instruction will perform integer conversion on the loaded data.

```
fild    SRC

fild    QWORD PTR [ebx] ; push(st, *b)
```

`fst`, `fstp`

Store data from the top of the stack to a memory location. `fstp` also pops that value from the stack.

`fadd`, `fiadd`, `faddp`, `fsub`, `fisub`, `fsubp`, `fsubr`, `fsubrp`, `fmul`, `fdiv`

Perform the appropriate arithmetic operation. Instructions with `i` operate with an immediate value as the source operand. Instructions suffixed with `p` implicitly operate on the top two elements of the stack (ST(0) and ST(1)), stores the result in ST(1), and then pops the top element of the stack.

For `fsub` and `fdiv`, the operand is what is subtracted or divided, respectively.

`fld1`, `fldl2t`, `fldl2e`, `fldpi`, `fldlg2`, `fldln2`, `fldz`

Push the given value onto the ST stack.

`fabs`

Takes the absolute value by clearing the sign bit of ST(0).

`fcos`, `fsin`

Calculates the given function of ST(0), storing it back.

`fsincos`

Calculate the sine and cosine of ST(0). Store the sine in ST(0), then push the cosine onto the ST stack, so the final result is

Table 20. Results of Executing `fsincos`

ST(0)	*x*	→	ST(0)	cos(*x*)
ST(1)	…		ST(1)	sin(*x*)
			ST(2)	…

Comparisons

The ST registers also hold a set of x87 condition code flags. However, there exists no branch instruction that can directly test these flags. They must first be transferred to the CPU’s flags register, and then tested there. This is done using a two-instruction combination:

```
fstsw   ax ; Store x87 status flags into AX
sahf       ; Set RFLAGS from contents of AH
```

`fcom`

Compare ST(0) with another ST register or memory location, setting the x87 condition codes.

XMM Use

XMM registers are the first major set of Streaming SIMD Extension (SSE) registers, and are the default for most modern floating-point operations.

There were originally eight such registers (XMM0-XMM7), which were expanded to sixteen registers (XMM0-XMM15) in x64 architecture. The XMM registers are all 128 bits long, because the intent for these registers is to be used in SIMD applications. This chapter focuses on the floating-point application of these registers using one value at a time (scalar, rather than packed).

Instructions that use `ss` operate on single-precision floats, instruction that use `sd` operate on double-precision floats.

SSE instructions can generally be determined from the use of `xmm` registers.

`movss`, `movsd`

Moves data between two XMM registers, or an XMM register and a memory location. Any such memory location **must** be 16-byte aligned, otherwise it is likely to crash the program (With the i7 and later CPUs, this has become less of an issue, but would still result in very poor performance). This includes values in the running process' stack frames, which must be aligned at run-time.

```
movss   xmm0, [rbx] ; x = *b
movss   xmm2, xmm1  ; z = y
```

```
.balign 16  ; Ensure 16-byte alignment in data section
.euler:
    .single 0.5772156649
.balign 16
    .double 3.1415926535
    .tfloat 2.7182818283
.
.
func:
    push    rbp     ; Ensure 16-byte alignment on stack
    mov rax, [rsp+0x18] ; Move
    mov [rsp], rax  ;
    movss   xmm0, [rsp] ; x = *arg7
```

Conversion Instructions

These conversion functions are used to turn signed integers into floats, or vice-versa. The conversion function will adjust appropriately based on the size of the destination integer. The destination of these conversions must be an appropriate register, but the source may be a register or memory location.

`cvtt` will truncate the floating-point source value, while a normal conversion will attempt to round.

`cvtsi2ss`, `cvtsi2sd`

`cvttss2si`, `cvttsd2si`

`cvtss2si`, `cvtsd2si`

`cvtss2sd`, `cvtsd2ss`

```
.balign 16
.euler:
    .double 0.57721566490153286060651209
.
.
.
movsd       xmm0, .euler
cvtsd2si    rax, xmm0
```

Calculations

Each of these perform the specific calculation on its two arguments. The destination must be an XMM register, and the source argument may be an XMM register or memory location.

`addss`, `addsd`

`subss`, `subsd`

`mulss`, `mulsd`

`divss`, `divsd`

`maxss`, `maxsd`

`minss`, `minsd`

```
triple:
    mulsd   xmm0, [rip + .three]
    ret
.balign 16
.three
    .double 3
```

The following instructions take two arguments, a source and a destination.

`roundss`, `roundsd`

`sqrtss`, `sqrtsd`

`rcpss`, `rcpsd`

`rsqrtss`, `rsqrtsd`

Fast Inverse Square Root, from Quake

```c
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    // evil floating point bit level hacking
    i  = * ( long * ) &y;
    // what the fuck?
    i  = 0x5f3759df - ( i >> 1 );
    y  = * ( float * ) &i;
    // 1st iteration
    y  = y * ( threehalfs - ( x2 * y * y ) );
    // 2nd iteration, this can be removed
//  y  = y * ( threehalfs - ( x2 * y * y ) );

    return y;
}
```

Comparisons

It is possible to compare to scalar floating-point values in assembly.

`comiss`, `comisd`

`ucomiss`, `ucomisd`

`cmpss`, `cmpsd`

The `comis*` and `ucomis*` instructions are identical, except that `comis*` will signal an exception if the source is any NaN, as poosed to a signalling NaN. These set the RFLAGS register appropriately.

IEEE Floating-point values may encode a possible Not-A-Number value. Comparing this value with any other value will set the parity flag in the list of flags. It is common to see code along the lines of:

```
ucomisd xmm0, xmm0
jp .error
```

The `cmps*` instructions require three arguments: the two locations being compared, and an immediate 8-

bit value that indicates the comparison to be performed, with the result going into the destination rather than `RFLAGS`.

Value	Comparison
0	==
1	<
2	<=
3	is either NaN
4	!=
5	>
6	>=
7	neither are NaN

Exercises

Exercise 1

Write a function `double hypoteneuse(double a, double b)` that calculates and returns the length of a right triangle's hypoteneuse for sides *a* and *b*.

Exercise 2

Write a function `int quadratic(double a, double b, double c, double *x, double *y)` that implements the quadratic formula. It should return the number of real roots that it finds, and encode the values into `*x` and `*y`.

Exercise 3

Measure the performance of `FastInvSqrt` against `rsqrtss`.

Chapter 16. Jump Tables

Choosing between multiple options in a C program can be done with a switch/case block. In assembly code, these are encoded as **jump tables**. A jump table an array of read-only values in the program containing jump destinations.

An index into the jump table is calculated into a register, and then a computed jmp instruction is used to jump to the correct instruction.

```
func:
    cmp edi, 3
    ja  .L5
    lea ecx, [8*edi + .jump_table]
    jmp [ecx]
...

.jump_table:
    .quad   .L1
    .quad   .L2
    .quad   .L3
    .quad   .L4
```

It is important that the computed jump does not escape the bounds of the jump table. The index must be tested against the bounds of the jump table prior to the jmp.

Making the contents of the jump table writable is extremely risky. While a writable array might be useful to store function pointers, it should not be used to store arbitrary destinations in the program.

C code is a very close mapping of the underlying assembly code. The switch/case block closely maps to assembly's structure. In a case block, exiting the switch statement must be done with an explicit break. So too in assembly, an explicit jmp is required to move elsewhere in the program.

```
case 'y':
    c = true;
case 'A':
    a = true;
    break;
case '!':
    d = true;
    break;
```

```
.L1:
    mov ecx, 1
.L2:
    mov eax, 1
    jmp .L5
.L3:
    mov edx, 1
    jmp .L5
```

A jump table is most efficient when it is compact. A compiler will do its best to reduce the range of cases before executing a jump table calculation. It may do so by turning highly-different cases into explicit jumps, and leaving the more compact cases for a jump table.

```
switch(c) {
case 'a':
    ...
case 'b':
    ...
case 'c':
    ...
case 'z':
    ...
```

```
cmp dl, 'z'

jmp .L4



sub edi, 'a'

jmp [8*edi + .L10]
```

Exercises

Exercise 1

Implement `unsigned scrabble_score(const char *word)`, which calculates the score in Scrabble for a given word.

Exercise 2

Implement `int phone_keypad_button(char letter)` that returns the number on a phone keypad for the given letter.

Chapter 17. Signal and Interrupts

At some point, a program must process exceptions or be controlled by outside processes. These are generally controlled by signals or interrupts.

While code generally concerns itself with signals, the CPU itself must coordinate any interrupts.

Interrupts

An **Interrupt** is the method through which the CPU and OS communicate. Hardware actions will generate interrupts. For instance, a hardware interrupt occurs when a key is pressed on a keyboard. An arithmetic interrupt occurs when the CPU divides by 0. Each interrupt has a specific integer identifier.

Upon receiving an interrupt, the CPU looks up a jump table, offset by the specific interrupt number. This jump table is created and managed by the OS. Each entry in this table is an interrupt handler.

The specific handler may exit the current process, put a keystroke in a buffer, halt the system, or any other number of actions.

Signals

Signals are how a userland process may interact with interrupts. The OS manages signals that are sent to each process, and may maintain a table of signal handlers for each process.

When the OS "sends" a signal to a process, it saves the current state of the process on the stack, then jumps to the handler in that process's signal table. The previous `rip` location was pushed onto the stack, and is popped back to when the handler exits.

Exceptions

Certain operations may trigger exceptions or faults. Dividing by zero is the most common example. It is possible to trap these exceptions and recover from them.

```c
int main(void)
{
    int divisor = 0;

    return 15 / divisor;
}
```

Any such fault or exception during a process is translated into a signal in the OS sense. Therefore, it is possible to write signal handlers for these exceptions.

```
#include <signal.h>
#include <stdio.h>

void recover(int sig)
{
    puts("Recovered");
}

struct sigaction floater;

int main(void)
{
    floater.sa_handler = recover;

    if(sigaction(SIGFPE, &floater, NULL)) {
        perror("Could not FPE handler");
        return 1;
    }

    int divisor = 0;

    return 15 / divisor;
}
```

Running the FPE exception handler, however, generates unusual behavior.

The current value of `rip` is pushed onto the stack, to be returned to. However, that line is what generates an exception, so when the line gets executed again, another exception occurs, in an infinite loop.

It can be extremely difficult to escape this situation gracefully. If an exception happened, what should the correct result have been? An exception at this level means that the program in question is in an unknown state at its current point, and continued exection at that point may generate highly deviant behavior.

Some high-level languages solve this problem through exception handling: an exception is thrown, and may be processed by an exception handling block (such as `try`/`catch` in Java, or `try`/`except` in Python).

In this case, the signal handler slowly unwinds the stack, looking for code that can handle the exception. In assembly language or C, this unwinding of the stack can be done more manually.

`gdb` with System Calls

The debugger `gdb` can halt on interrupts like system calls or signals via its `catch` command.

Chapter 18. Digital Circuitry

While assembly programming in x86 architecture is fairly high-level, understanding the operation of the lower level of circuitry can be an aid to the developer. All non-trivial abstractions, to some degree, are leaky[3], and assembly language manages to abstract away the low-level details of circuitry.

Digital Logic

Digital Logic is merely the functions that are encoded into circuitry. These generally involve the implementation of Boolean functions or operators in transistors.

The presence of current or voltage in a line is often treated as binary 1, and its absence as 0. This is not required; some circuits may define them as reversed! For this text, presence of voltage is treated as 1 and absence as 0. This text also assumes that **voltage** is the signal, rather than current, but that distinction is not required for understanding this content.

Transistors

A **transistor** is a device used in circuitry to control voltage or current. There are many different types of transistors, in a variety of shapes, sizes, functionality, and use. A transistor is special in that it can use a small presence of voltage to emit or block a larger amount of voltage, similar to how power steering can allow one to turn a car’s wheels with only one finger.

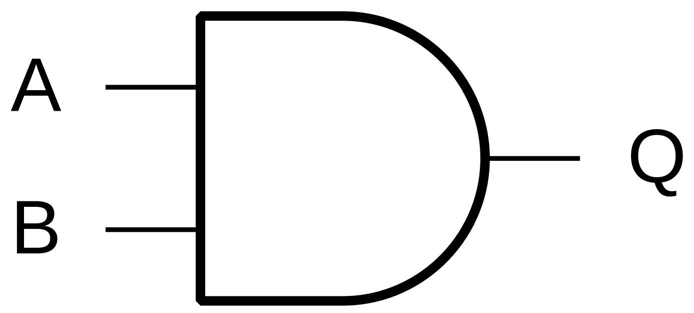
This results in two useful effects: firstly, a low-powered signal can be amplified, so a transistor functions as a signal booster. Secondly, as a transistor requires both the signal and a power source to produce an output, they function as a kind of logical AND gate.

Due to the wide variety of transistors, there are wide varieties of their operation. Some are normally off, some normally on. Some may be safely allowed to “float” a signal, some cannot. Vaccuum tubes, MOSFETs, PNP, NPN, and relays are all different logical forms of the transistor, but may reverse the logic or inputs for the sake of production.

Logic Gates

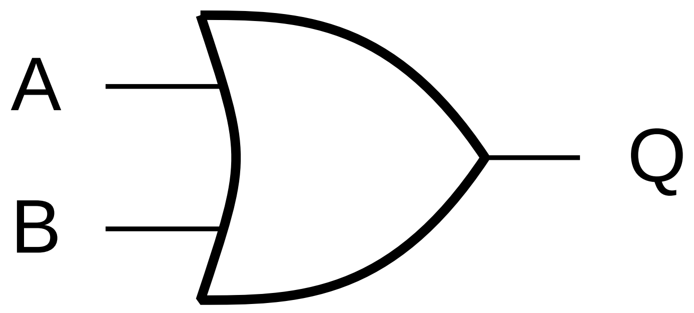
By combining together transistors, different logical gates may be created that mirror binary logic functions. Unintuitively, even though a transistor may seemingly operate like an AND gate, the actual circuitry may involve six transistors. This is from a number of factors, mostly having to do with ensuring that sufficient gain is enacted upon the input signals.

AND Gate



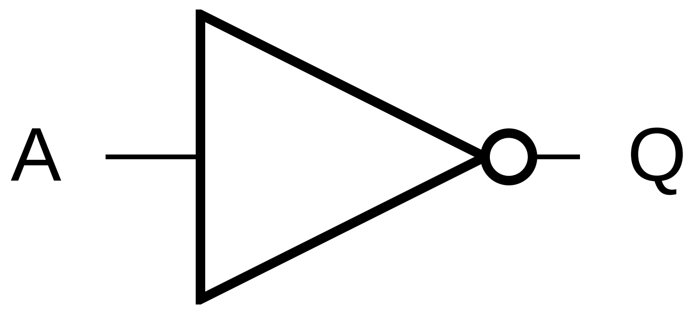
An AND gate functions like the Boolean AND expression of two inputs. If both inputs are 1, then the output is 1. Otherwise, the output is 0.

OR Gate



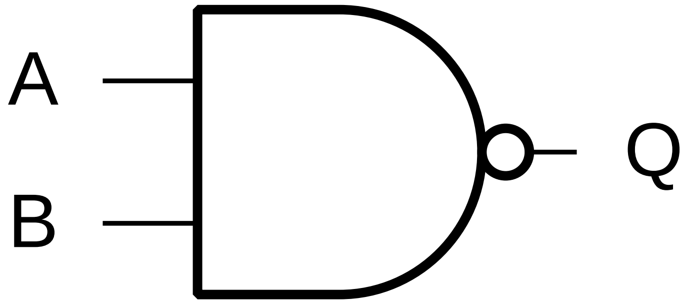
An OR gate functions like the Boolean OR expression of two inputs. If any inputs are 1, then the output is 1. Otherwise, the output is 0.

NOT Gate



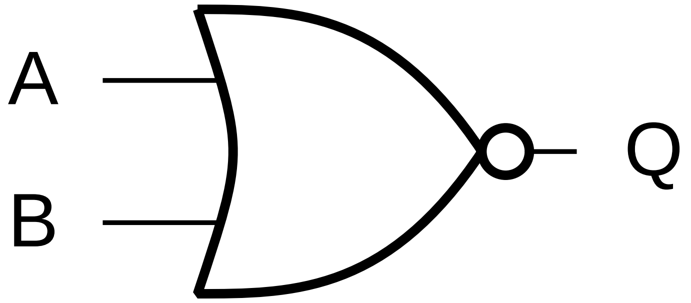
A NOT gate functions like the Boolean NOT expression of an input. If the input is 1, then the output is 0. Otherwise, the output is 1.

NAND Gate



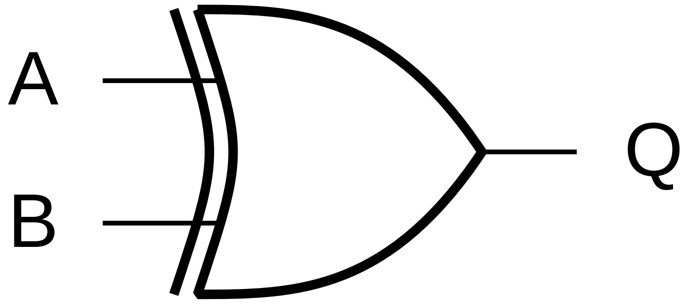
A NAND gate functions like the Boolean AND expression of two inputs, with the result then passed through a NOT. If both inputs are 1, then the output is 0. Otherwise, the output is 1.

NOR Gate



A NOR gate functions like the Boolean OR expression of two inputs, with the result then passed through a NOT. If any inputs are 1, then the output is 0. Otherwise, the output is 1.

XOR Gate



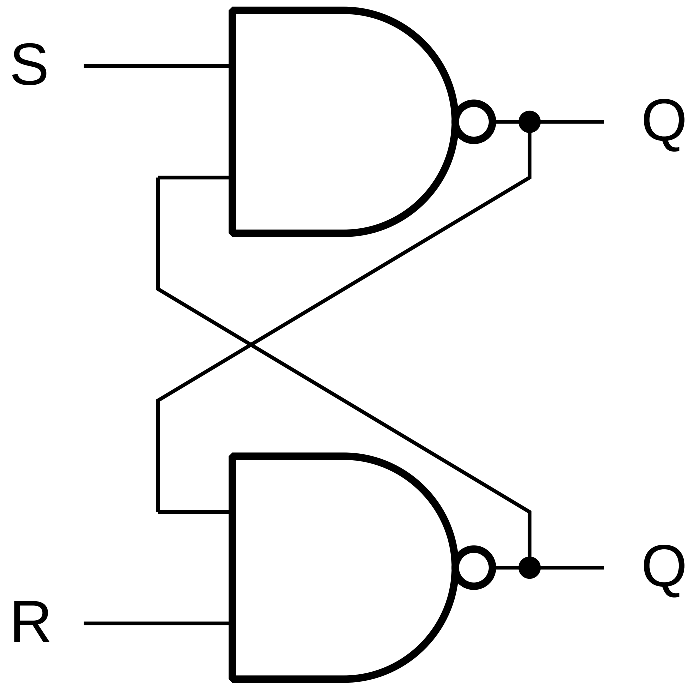
An XOR gate functions like the Boolean exclusive OR expression of two inputs. If the inputs are the same, then the output is 0. Otherwise, the output is 1.

Boolean Functions

The NAND and NOR gates are important because those operations are **universal operators**. That is, using only the given gate, any other Boolean operation can be implemented.

P	Q	¬(P ∧ Q) = R	¬(R ∧ R)	P ∧ Q
T	T	F	T	T
T	F	T	F	F
F	T	T	F	F
F	F	T	F	F

Flip-Flops and Latches

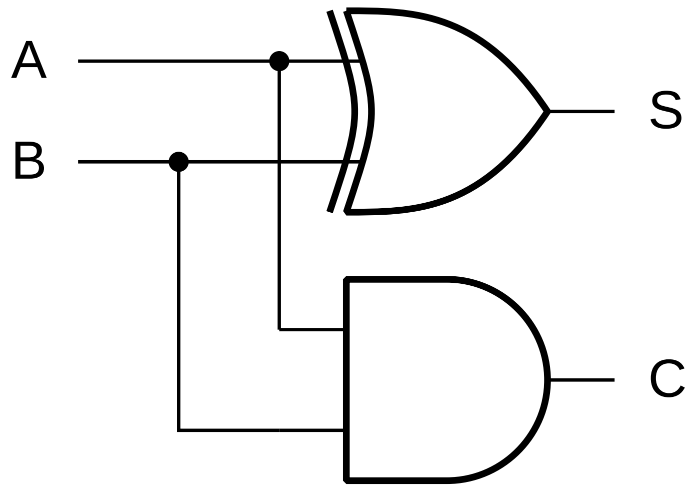


A **flip-flop** is a special circuit capable of storing a value. Memory is made up of many such flip-flops to store gigabytes of data. A simple flip-flop is sometimes called a **latch**. (The technical difference is that a flip-flop involves a clock, while a latch does not.)

By wiring the output back into the inputs of the gates, the value of the output is maintained. Note that the NAND latch shown has a possible illegal state: if both inputs are 0, the results would get locked into Q = ¬Q = 1, which does not make logical sense. A circuit would normally add additional circuitry to make such a set of inputs cause a valid result, either flipping the result (called a **JK Latch**) or holding the result (**E-latch**).

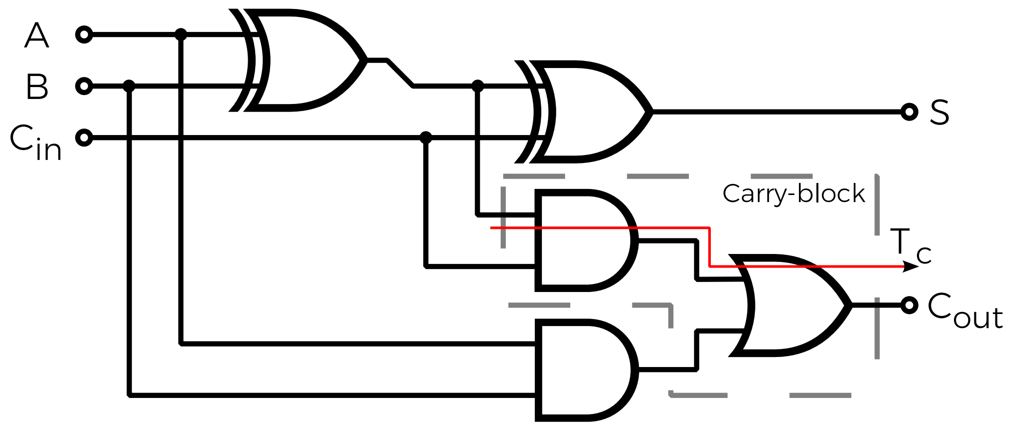
Arithmetic Circuits

Half Adder



It is possible to use the listed gates to perform addition between two signals. The simple form of this is the **half adder**, which produces the sum of two inputs, plus the possible carry. Notice that while the diagram shown uses an XOR and an AND gate, the whole circuit could be built using only universal NAND gates.

Full Adder



Since an addition operation may result in a carry, a true addition engine (adding together multi-bit numbers) requires three inputs for each position: the two bits to add, plus the result of the previous carry. Thus, a **full adder** accounts for a previous addition's carry bit, and produces the sum of the two inputs plus the previous carry.

A **ripple-carry adder** is a chain of these full adders, with the carry output of one adder being hooked to the carry input of the next. While this type of circuit is not normally made due to its slow speed, it is easily understandable as to how circuits can perform mathematical operations using universal gates.

Finite State Machines

A **finite state machine** is a simple abstract machine that conforms to a number of rules.

1. The machine has a finite number of possible states, one of which is a `starting state', and at least one of which is a `final state'.
2. The machine takes input that moves the machine between states according to machine-specific rules.
3. The machine, when done processing input, indicates whether or not it is in a final state (success), or not (failure).

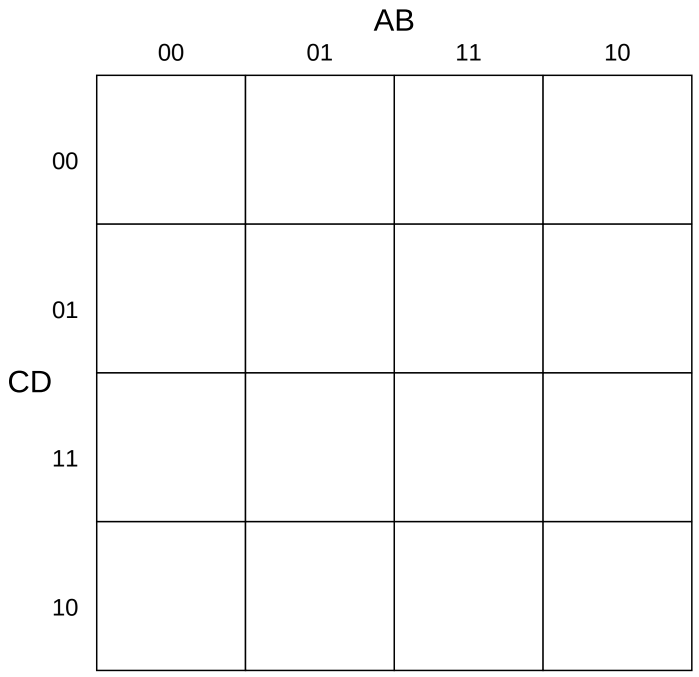
Given logic gates and flip-flops (which are composed of logic gates), circuitry may be built that acts as a finite state machine: the possible states are the permutation of all flip-flop values, and the input is clock-synchronized signals on an input wire.

Circuit Simplification

The smallest possible circuit, in terms of silicon, is usually most desired. Including fewer paths means faster signal throughput, resulting in a faster overall chip. Any technique that can be used to eliminate circuits or pathways is generally beneficial to the resultant chip.

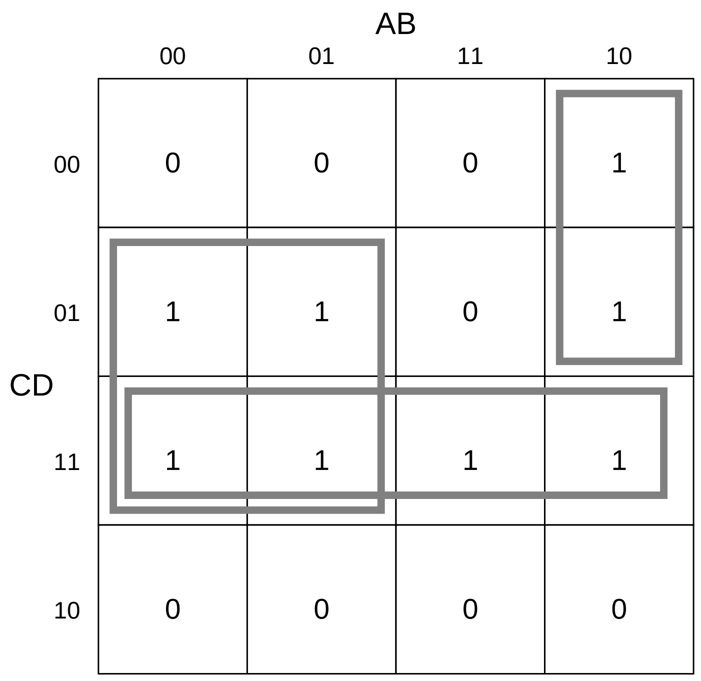
Karnaugh Maps

Karnaugh Maps are a useful technique used to simplify binary circuitry. A Karnaugh Map is a square grid consisting of every possible output combination from the given inputs.



The grid is treated like a torus, in that the left and right edges (as well as the top and bottom) “wrap around” and meet up, adjacent to each other. Notice that the bits are not ordered as 00, 01, 10, 11, but rather as 00, 01, **11, 10**. This means that moving from one cell in the grid to an orthogonally adjacent one involves changing only a single bit. This form of encoding is sometimes referred to as **Gray Code**, where only a single bit is toggled between adjacent values.

For each combination of inputs, the output is determined and entered into the grid. Then, a series of rectangles whose areas are a power of two are drawn over the grid, to cover all possible 1-outputs.



The rectangles allow for describing the system of inputs in the most minimal way possible. Each rectangle has a certain number of set inputs, with the rest allowed to vary. Matching up the set inputs yields a Boolean and-expression, and then all rectangles are joined using a Boolean or-expression.

For instance, in the example shown, the rectangles outline the sub-expressions of CD, A’D, and AB’C'. So, the whole expression would be CD + A’D + AB’C'.

Exercises

Exercise 1

Using only NAND gates, make the equivalant of a NOR gate.

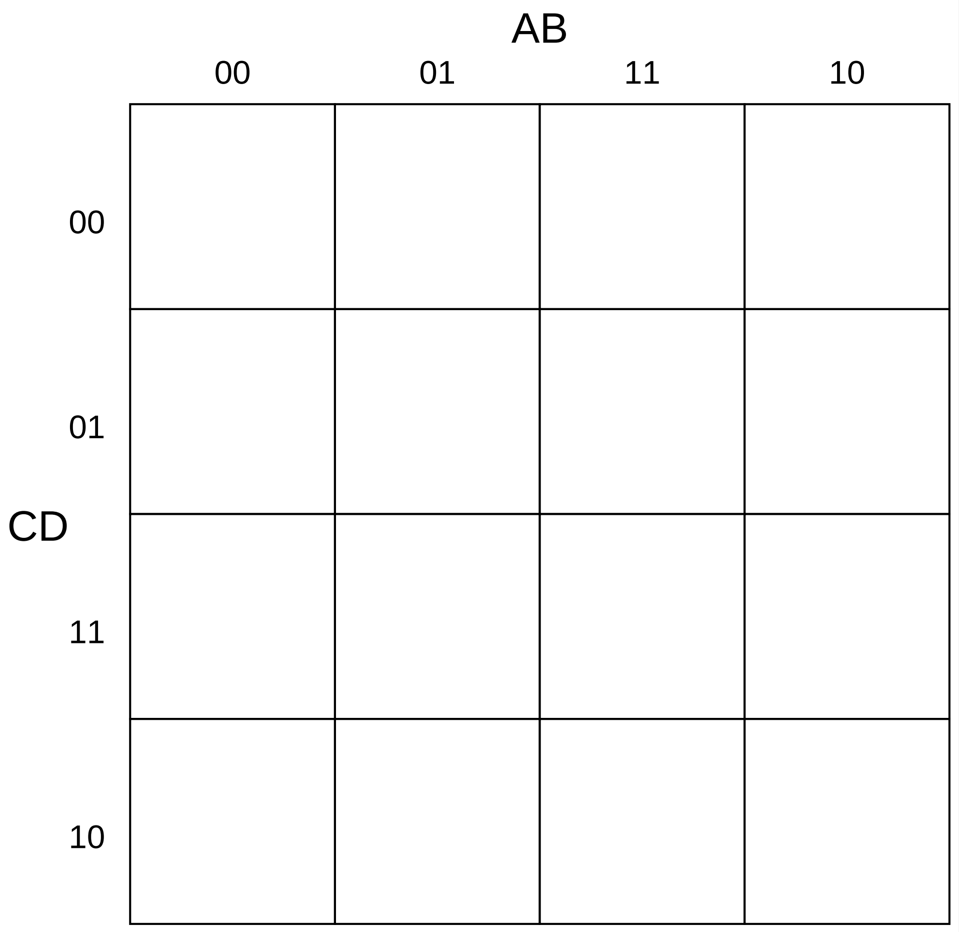
Exercise 2

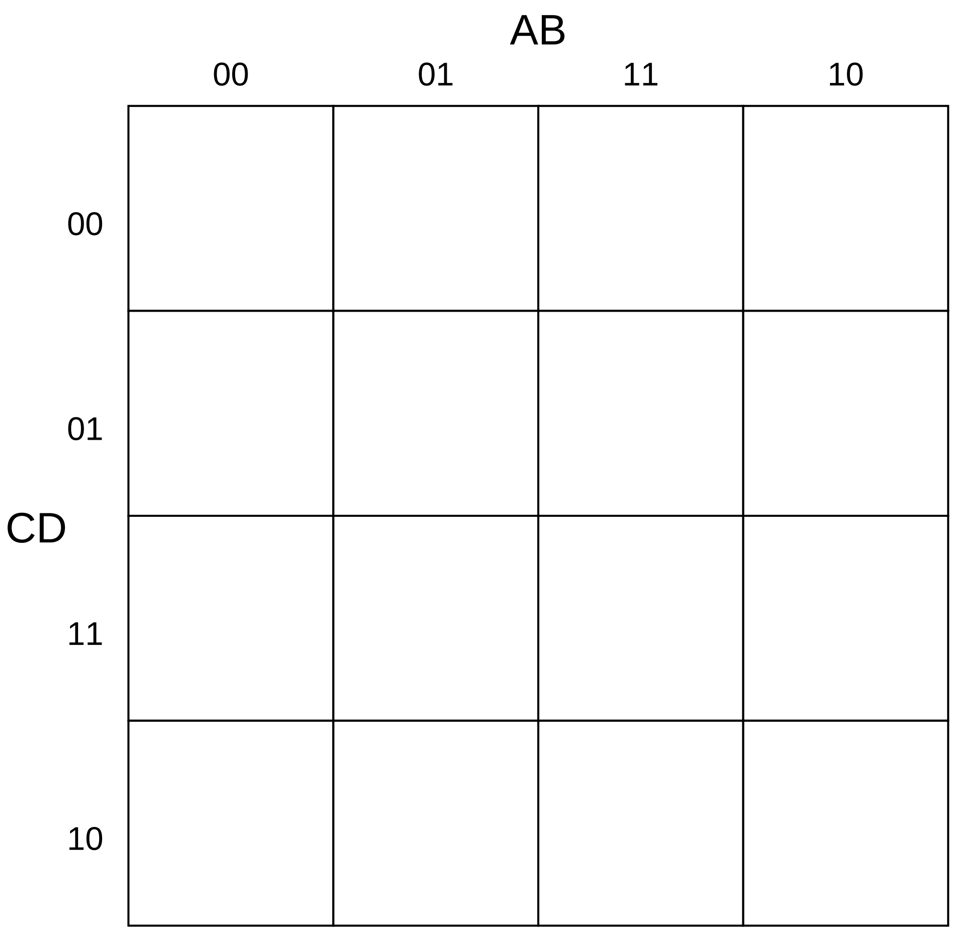
Using only NAND gates, make the equivalant of an XOR gate.

Exercise 3

Draw a Karnaugh map and a simplified Boolean equation for each chart.

ABCD	T
ABCD'	T
ABC’D	T
ABC’D'	T
AB’CD	T
AB’CD'	T
AB’C’D	T
AB’C’D'	T
A’BCD	T
A’BCD'	F
A’BC’D	F
A’BC’D'	F
A’B’CD	F
A’B’CD'	T
A’B’C’D	F
A’B’C’D'	T



ABCD	T	
ABCD'	F	
ABC’D	F	
ABC’D'	T	
AB’CD	F	
AB’CD'	T	
AB’C’D	T	
AB’C’D'	F	
A’BCD	F	
A’BCD'	T	
A’BC’D	T	
A’BC’D'	F	
A’B’CD	T	
A’B’CD'	F	
A’B’C’D	F	
A’B’C’D'	T	

[3] Joel Spolsky, **The Law of Leaky Abstractions**

Chapter 19. SIMD

There is a lower limit to how fast a single CPU can execute an instruction. To execute faster than that rate, multiple calculations must be performed in parallel.

Using multiple threads or processes involves overhead from task switching. Managing data sent to multiple CPUs would also involve overhead from locks and waits.

A large enough register in the CPU could hold multiple items of data, and then have a single instruction that manipulates all that data at once. This is known as **Single Instruction/Multiple Data**, or SIMD.

Concept

A table of binary addition results is extremely simple.

	0	1	0	1
+	0	1	1	0
	1	0	1	1

Because a carry only happens in 25% of the possible operations, doing an add across eight bits, which takes a single clock cycle, is not that different from adding two four-bit values. As long as the possible “boundary carries” are not carried, a 64-bit register could be used to add two 32-bit values, four 16-bit values, or eight 8-bit values in one instruction.

	0 1 0 1	0 1 1 0
+	0 1 1 0	0 0 1 1
	1 0 1 1	1 0 0 0

While this example uses integers, this could also be done using floating-point values (except `long double`s). The XMM, YMM, and possibly ZMM registers are wide enough to hold multiple integers or floating-point values. This is referred to as “packed data”. Effectively, an partial array of data can be stored in these registers.

Packed Data

Each of the XMM, YMM and ZMM registers can hold more than one piece of data. When the register holds only one piece of data, it is referred to as holding a scalar value. **Packed Data** is when the register holds more than one piece of the same kind of data.

Instructions that manipulate scalar data have the letter ‘S’ as part of their mnemonic. These instructions come from an earlier chapter.

```
movsd   xmm0, [rsi]
movsd   xmm1, [rdi]

mulsd   xmm0, xmm1
```

Instructions that manipulate packed data have a letter ‘P’ as part of their mnemonic.

```
movupd  xmm1,  [rsi]
movupd  xmm2,  [rdi]

mulpd   xmm1, xmm2
```

When one instruction performs two distinct operations on two distinct pairs of data at the same time, the CPU is executing those instructions in parallel. This offers oppportunity for improved performance.

Instruction Set

Intel CPUs have a number of specialized instructions that operate on packed data. These specialized instructions deal with:

- data movement
- arithmetic
- comparisons
- logic

This is by no means an exhaustive coverage of Intel’s many instructions that operate on SIMD data.

The instructions are really just common building blocks for calculations that show up in a lot of different formulae. Consider the following formula.

$a^2 + b^2 = c^2$

Several of the instructions will be demonstrated by calculating the hypoteneuese of a right triangle several different ways using different sets of instructions.

Data Movement Instructions

Instruction	Mnemonic	Description
`movapd`	MOVe Aligned Packed Double-precision floating-point	Transfers a 128-bit double-precision floating-point operand between memory and an XMM register, or between XMM registers. The memory address must be aligned to a 16-byte boundary, otherwise a general protection exception (GP#) is generated.
`movupd`	MOVe Unaligned Packed Double-precision floating-point	Transfers a 128-bit double-precision floating-point operand between memory and an XMM register, or between XMM registers without any requirement for alignment of the memory address.
`movhpd`	MOVe High Packed Double-precision floating-point	Transfers a 64-bit double-precision floating-point operand between the high 64 bits of a 128 bit memory location and the high 64 bits of an XMM register or between the high quadword of two XMM registers. The low quadword of the register is left unchanged. Alignment of the memory address is not required.
`movlpd`	MOVe Low Packed Double-precision floating-point	Transfers a 64-bit double-precision floating-point operand between the low 64 bits of a 128 bit memory location and the low 64 bits of an XMM register or between XMM registers. The high quadword of the register is left unchanged. Alignment of the memory address is not required.
`movdqa`	MOVe Double Quad word Aligned	Transfer 128 bits of integers between memory and an XMM register or between two XMM registers. The memory address must be aligned to a 16-byte boundary, otherwise a general protection exception (GP#) is generated.
`movdqu`	MOVe Double Quad word Unaligned	Transfer 128 bits of integers between memory and an XMM register or between two XMM registers. Alignment of the memory address is not required.
`movddup`	MOVe Double precision and DUPlicate	Copies the double precision floating point number in the low half of the source register to both the low half and the high half of the destination register.
`vbroadcast`	load with BROADCAST floating point data	Broadcast the same single precision floating point number to all four parts of an XMM register.

Instructions that work with aligned data are useful when the data is in another register or when the memory area that holds the data is part of the executable. Data loads faster because the CPU does not have to worry about the data being split into different cache segments. If data does not in fact start on a 16 byte boundary, the code will throw a general protection (GP#) exception. Instructions that work with aligned data have the letter ‘A’ in their mnemonic.

```
    ; load four floats into a register with one instruction
    movaps    xmm0,   XMMWORD PTR firstFloatArray[rip]
    ....

; movaps requires data aligned on a 16 byte boundary.
.balign 16
firstFloatArray:
.float  1.0, 2.0, 3.0, 4.0
```

NOTE

Structures that are aligned on 16 byte boundaries might well have members that are not aligned.

Instructions that work with unaligned data are useful when loading data from an address that was passed as parameter. The data lives in the caller's data area, which has little to no control over its alignment. Instructions that work with unaligned data have the letter 'U' in their mnemonic.

```
.equ    SizeOfXmmRegister, 16  ; bytes in XMM register

....

    ; load two doubles from each of two arrays
    ; that were passed as parameters
    movupd  xmm1, XMMWORD PTR [rsi]
    movupd  xmm2, XMMWORD PTR [rdi]
    ....  ; do something with the two doubles

    ; move the pointers to the next pair of doubles in each array
    add rdi, SizeOfXmmRegister
    add rsi, SizeOfXmmRegister
    .... ; loop
```

NOTE

Structures that are aligned on 16 byte boundaries might well have members that are not aligned.

In this exameble, think of `rsi` and `rdi` as being pointers to doubles. Each trip through the loop, move the pointers *two* elements down the array.

To process an array of integers, use the `movdqa` or the `movdqu` instruction depending on whether the data area aligns on a 16-byte boundary or not.

```
.equ    SizeOfXmmRegister, 16  ; bytes in XMM register

....

    ; load eight short unsigned integers (16 bits each)
    ; from each of two arrays that were passed as parameters
    movdqu  xmm1, XMMWORD PTR [rsi]
    movdqu  xmm2, XMMWORD PTR [rdi]
    ....  ; do something with the eight short unsigned ints

    ; move the pointers to the next eight short unsigned integers in each array
    add rdi, SizeOfXmmRegister
    add rsi, SizeOfXmmRegister
    .... ; loop
```

NOTE

This version of the `mov` instruction does not care if it moves two `unsigned long long`s, four `signed int`s, eight `short int`s or sixteen `char`s. It moves **128 bits**. What those bits mean is up to the developer. Other instructions (i.e. arithmetic instructions) *will* care. `movdqa` and `movdqu` will not.

Comparisons

Instruction	Mnemonic	Description
`maxps` `maxpd`	MAXimum of Packed Singles or Doubles	Depending on the instruction, each packed operand contains either two double precision floating point numbers or four single precision numbers.
`minps` `minpd`	MINimum of Packed Singles or Doubles	Depending on the instruction, each packed operand contains either two double precision floating point numbers or four single precision numbers.
`cmpps` `cmppd`	CoMPare Packed Singles or Doubles	Compare packed floating-point values in xmm2/m128 and xmm1 using bits 2:0 of an immediate 8 bit value as a comparison predicate.

`pcmpeqb`
`pcmpeqw`
`pcmpeqd`
`pcmpeqq`
	Packed CoMPare for EQuality: Bytes, Words, Doublewords, Quadwords	Depending on the instruction, compares packed bytes or words or double words or quad words between two XMM registers or between an XMM register and 128 bit memory loacation. Sets corresponding data elements in destination to all 1. Sets other data elements to all 0.

Arithmetic

Instruction	Mnemonic	Description
`addpd`	ADD Packed Doubles	Add packed double precision floating-point operands. Each packed operand contains two double precision floating point numbers.
`subpd`	SUBtract Packed Doubles	Subtract packed double precision floating-point operands. Each packed operand contains two double precision floating point numbers.

`paddb`
`paddw`
`paddd`
`paddq`
	Packed ADD: Bytes, Words, Doublewords, Quadwords	Add packed unsigned integers. Depending on the instructions, the integers will be 8 bits or 16 bits or 32 bits or 64 bits. Carry flags are discarded.

`paddsb` `paddsw`	Packed Add Saturation: Bytes, Words	Add packed signed integers with signed staturation. This means that there is no overflow or underflow, results are truncated to the min or max value for the type. Carry flags are discarded.
`psubsb` `psubsw`	Packed SUBtract Saturation: Bytes, Words	Subtract packed signed integers with signed staturation. This means that there is no overflow or underflow, results are truncated to the min or max value for the type. Carry flags are discarded.
`psubusb` `psubusw`	Packed SUBtract Unsigned Saturation: Bytes, Words	Subtract packed unsigned integers with unsigned staturation. This means that there is no overflow or underflow, results are truncated to 0 or the max value for the type. Carry flags are discarded.
`haddpd`	Horizontally ADD Packed Doubles	Horizontally add doubles. The two floats in the destination operand are added together, as are the two floats in the second operand. The two sums are placed in the destination operand.
`hsubpd`	Horizontally SUBtract Packed Doubles	Horizontally subtract doubles. The two floats in the destination operand are subtracted, as are the two floats in the second operand. The two differences are placed in the destination operand.
`mulpd`	MULtiply Packed Doubles	As these are floating-point multiplications, the products are the same width as the factors.
`divpd`	DIVide Packed Doubles	As these are floating-point divisions, the results are the same width as the operands.

Instruction	Mnemonic	Description
`sqrtpd`	SQuare RooT of Packed Doubles	Calculates the square root of each item in the source operand, and stores the result in the destination.

Illustrated horizontal add

```
horizontalAddDoubles:
    lea     r8, firstDoubleArray[rip]
    lea     r9, secondDoubleArray[rip]
    lea     rax, answerDouble[rip]
    movapd  xmm1, XMMWORD PTR [r8]   ; XMM1 holds 1.0 and 2.0
    movapd  xmm2, XMMWORD PTR [r9]   ; XMM2 holds 9.0 and 8.0
    haddpd  xmm1, xmm2               ; XMM1 holds (1.0 + 2.0) and (9.0 + 8.0)
    movapd  XMMWORD PTR [rax], xmm1  ; the answer array holds 3.0 and 17.0
    ret

.data
.align 16
firstDoubleArray:  .double  1,2,3,4,5,6,7,8,9,50
.align 16
secondDoubleArray: .double  9,8,7,6,5,4,3,2,1,5
.align  16
answerDouble:   .double   48.0,48.0,48.0,48.0,48.0,48.0,48.0,48.0
```

Horizontal add and horizontal subtract operate on adjacent numbers. They operate on single precision and double precision floats and they operate on all sizes of integers.

There is no horizontal multiply and no horizontal divide.

The following can be used to calculate the hypoteneuse of two different triangles at the same time. Substitute single precision floating point numbers to process four different traingles at the same time.

Illustrate packed multiplication and packed addition and packed square root

```
movdqu xmm1, XMMWORD PTR [rsi]  ; load one side of two different triangles
movapd xmm2, xmm1               ; XMM1 and XMM2 have the exact same numbers
movdqu xmm3, XMMWORD PTR [rdi]  ; load the second side of two different triangles
movapd xmm4, xmm3               ; XMM3 and XMM4 have the exact same numbers

mulpd  xmm1, xmm2  ; square side #1 of two triangles
mulpd  xmm3, xmm4  ; square side #2 of two triangles
addpd  xmm1, xmm3  ; XMM1 now has the sum of the squares
                   ; of both sides of two different triangles
sqrtpd xmm1, xmm1  ; XMM1 now has the hypoteneuese of two different triangles
```

Note that the `movapd` instructions could be removed if the `mulpd` instructions operate on the same register twice.

Combinations

Instruction	Mnemonic	Description
`addsubpd`	ADD and SUBtract Packed Doubles	Add the odd-numbered pairs of floating point numbers. Subtract the even-numbered pairs.
`pmaddwd`	Packed Multiply and ADD Words to Doublewords	Perform packed integer multiplication followed by addition of the result. Also called "dot product".

PMADDWD

`pmaddwd` performs a packed multiplication of four pairs of 16-bit integers. The raw data must consist of whole numbers that are adjacent in memory. The product of the each multiplication becomes a 32-bit integer. Adjacent 32-bit integers are added.

Typical usage for dot product calculations would be matrix math. There are, however, additional ways to use this instruction.

The hypoteneuse of a right triangle can be calulated using `pmaddwd`.

Example of Dot Product

```
movdqu  xmm1, XMMWORD PTR [rsi]  ; load two sides of four different triangles
pmaddwd xmm1, xmm1 ; A squared + B squared = four 32 bit C squared in XMM1
```

`pmaddwd` works with 16 bit integers. The first instruction above loads eight 16 bit integers into XMM1. Each pair of 16 bit integers holds the length of one side of a right triangle. There are measurements of 8 triangles in XMM1. A packed multiply of XMM1 with itself would square each measurement.

`pmaddwd` performs a packed multiply of XMM1 with itself. When 16 bit integers are multiplied, the result could be as big as 32 bits. `pmaddwd` treats the eight 16-bit registers as four 32-bit registers and stores the sum of the packed multiplications in them. That means that each of the 32-bit hunks holds the square of the hypoteneuse for four different triangles.

Logical

Instruction	Mnemonic	Description
`por`	Packed bitwise OR	Bitwise OR of a pair of XMM or YMM or ZMM registers
`pand`	Packed bitwise AND	Bitwise AND of a pair of XMM or YMM or ZMM registers
`psllw` `pslld` `psllq` `pslldq`	Packed Shift Left Logical: Words/Doublewords/Quadwords/Double-Quaadwords	Shift 16 or 32 or 64 or 128 bit packed data items left. Fill with zero.
`psrlw` `psrld` `psrlq` `psrldq`	Packed Shift Right Logical: Words/Doublewords/Quadwords/Double-Quaadwords	Shift 16 or 32 or 64 or 128 bit packed data items right. Fill with zero.
`psraw` `psrad` `psraq`	Packed Shift Right Arithmetic: Words/Doublewords/Quadwords	Shift 16 or 32 or 64 bit packed data items right. Fill with whatever value is in high bit. Similar to rotate right in the general purpose registers.

Conversion

Instruction	Mnemonic	Description
`cvtdq2ps` `cvtdq2pd`	ConVerT Doubleword TO Packed Single/Double-precision floats	Convert an array of 32-bit integers to floating-point format.
`cvtps2pi` `cvtpd2pi`	ConVerT Packed Single/Double-precision floats TO Packed Integers	Convert packed single- or double-precision floats to 32 bit signed integers. Results are rounded.
`cvttps2dq` `cvttpd2dq`	convert packed floats to packed dword integers with truncation	Depending on the instruction, convert packed single or double precision floats to 32 bit signed integers. Results are truncated.

`pmaddwd` works on integers, but `sqrtps` works on floating point numbers. `cvtdq2ps` does the conversion from packed integers to packed single precision floating point.

Example of floating point conversion and Packed Square Root

```
movdqu  xmm1, XMMWORD PTR [rsi]  ; load two sides of four different triangles
PMADDWD xmm1, xmm1  ; A squared + B squared = four 32-bit C squared
cvtdq2ps    xmm1, xmm1  ; they are now four single precision floats
sqrtps  xmm1, xmm1  ; four hypoteneueses for four different triangles
```

Compare this listing with the listing that did not use the dot product instruction.

Reciprocal Instructions

Instruction	Mnemonic	Description
`rcpps`	ReCiProcal of Packed Single-precision floats	Divide the packed single precision floats in the source into 1.0 and store the results in the destination register.
`rsqrtps`	Reciprocal SQuare RooT of Packed Single-precision floats	Take the square roots of the packed single precision floats in the source. Divide those floats into 1.0 and store the results in the destination register.

Appendix A: Bits

All values in programming are represented by bits. Whether the value in question is an integer, a letter, a floating-point value, or a pointer to something else is purely based on **how those bits are intepreted**.

It is the responsibility of the programmer to interpret bits correctly. A programming language like C can bring such misinterpretations to light, requiring the enterprising programmer to explicitly cast or convert bits of one interpretation to another.

Assembly programming makes no such checks, and relies on the writer to know what they are doing.

Integer Values

Integer values interpret bits based on binary powers of two.

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	0	1	0	1	0	0	0	1
	128	64	32	16	8	4	2	1
81	=	64	+	16		+		1

One of the useful properties of binary is how simple the arithmetic tables are.

	0	1	0	1	0	0	0	1
+	0	0	0	1	1	1	1	1
	0	1	1	1	0	0	0	0

				1	**0**	**1**	**0**	**1**
×						1	1	1
				1	0	1	0	1
			1	0	1	0	1	0
+		1	0	1	0	1	0	0
	1	0	0	1	0	0	1	1

Unsigned Integers

The easiest way to work with integers is treating them as unsigned integers. This means the value of such an integer ranges between 0 and 2^{N-1}, where N is the number of bits or binary places in the number.

Signed Integers—Two’s Complement

There are many ways of implementing signed integers; integers that may be negative or positive. x86 architecture uses **two’s complement** encoding to interpret signed integers.

In two’s complement, the most significant bit is considered to be negative, and all other bits are still positive.

	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	0	1	0	1	0	0	0	1
	–128	64	32	16	8	4	2	1
81	=	64	+	16		+		1

	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	1	0	1	0	1	0	0	1
	–128	64	32	16	8	4	2	1
–87	–128	+	32	+	8	+		1

Character Values—Encodings

By associating any enumerable set with the integers, that set can be represented in binary. The most common set is that of letters or characters. This mapping of integers to characters is called an **encoding**.

The most well-known encoding is ASCII, the American Standard Code for Information Interchange. ASCII encodes digits, some punctuation, and the English alphabet in bother upper- and lower-case letters. In addition, there are a number of nonprinting entities designed to control devices or organize data.

Since other languages have other letters or glyphs, ASCII is insufficient to represent all possible means of communicating. While there are many other encodings for specific character sets (some written languages may even have multiple competing encodings), the emerging solution is that of Unicode.

Unicode is designed to be a universal encoding for all possible character sets. It currently has capacity for over one million distinct characters, of which about 144,000 are used[4].

Unicode is merely the enumeration of characters, mapping them to integer values called **code points**. The actual encoding of those numbers into binary representation is a bit more fractured. Some encodings require that every glyph or character uses 32 bits, even if most of those bits are 0. Some encodings require up to 48 bits.

Pointers

Pointers in C are explicitly not integer values, but are intended to be abstracted away as black boxes. In reality, they can be treated like unsigned integers, especially at the level of assembly language.

Any register or memory location could hold what is intended to be a pointer to some other location in memory. The `NULL` pointer is defined as being the integer value 0.

Floating-Point Values

IEEE Floating-Point values sacrifice some simplicity, precision, and speed at additions for a larger range of numeric values and increased speed at multiplication and trigonometric calculations.

IEEE Floating-Point Numbers are scientific notation, but in binary. A number in scientific notation is of the form:

6.02214×10^{23}

It is composed of a **mantissa**, the decimal number, multiplied by the number 10 raised to an **exponent**. The exponent may be any integer, and the mantissa is in the range (-10, -1], [1, 10). Note that in base-ten, decimal, the mantissa is limited to being less than the base, and the exponent is applied to the base. A number in this form is more compact than 602,214,000,000,000,000,000,000, and also carries with it the level of precision (in this case, 6 decimal digits).

For binary, therefore, a similar number would look like this:

$1.111111100001100001_2 \times 10_2^{1001110}$

The mantissa is still clamped to [1, 10_2), which means that the leading digit of the mantissa must always be 1. The same amount of precision, one part in a million, requires more binary digits.

A floating-point number encodes the mantissa and exponent in a set of bits, with a few optimizations.

S	Exponent								Mantissa																						

Since the leading digit of the mantissa is always 1, it can be presumed to always be 1, and omitted from the encoding. This means only the digits after the decimal point are encoded in a floating-point format. The sign of the mantissa (positive or negative) is encoded as a single bit in its own section.

S	Exponent								Mantissa																						
0									1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0

To express both positive and negative exponents, a **bias** is added to the scientific notation’s exponent before being stored in the encoding. When the number is retrieved, the bias is subtracted. This ends up being more efficient for calculations than using two’s-complement for signing the exponent. For 32-bit floats, the bias is 127.

S	Exponent								Mantissa																						
0	1	1	0	0	1	1	0	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0

Programmers are not expected to manipulate these bits directly; but it is important to understand that floats are limited in the amount of precision that they allow. Different floating-point formats allow for more bits of precision.

Type	Exponent Bits	Mantissa Bits	Decimal Digits
`float`	8	23	7
`double`	11	52	15
`long double`	15	64	18

Exercises

Exercise 1

Which ASCII character is 0x61?

Exercise 2

What is 0b10001011 as a signed byte integer?

Exercise 3

What is the value of 0b00110101 ?

Exercise 4

Read up on the Patriot Missile Failure at Dhahran. How many hours would it take for the error to be 1 second?

[4] as of Unicode 13.0, published March 2020

Appendix B: Inline Assembly

Occasionally, it can be useful to write assembly code directly in a C file. Most C compilers have a built-in function to do exactly that.

Inline Assembly

```
asm("mov %0, %1" : "=r" (answer) : "r" (operand));
asm("add %0, %1" : "=r" (sum) : "r" (addend) : "cc");
```

`asm` is a keyword in C++, but not officially part of the C standard. If the symbol `asm` is in use as a variable or function name, most C compilers also offer another symbol, such as `_asm or \__asm\_`.

Inline assembly works on a string templating system, like `printf(3)`. The most basic form of inline assembly just involves writing assembly code in strings.

```
asm("mov ecx, eax");
```

If multiple lines of assembly are desired, C's string concatentation can be extremely useful. Each line of assembly must be terminated with a newline.

```
asm(    "mov ecx, eax\n\t"
        "add ecx, eax\n\t"
        "div ecx"
    );
```

However, in the context of larger code, there is no guarantee that a given variable is stored in one register or another. Inline assembly could end up clobbering data or corrupting variables used in the current function.

```
asm(
    template
    : output operands
    : input operands
    : clobbered registers
);
```

```
asm goto(
    template
    :
    : input operands
    : clobbered registers
    : goto labels
);
```

```
asm volatile ( );
```

Appendix C: The Story of Mel

This was posted to Usenet (`net.jokes`) by its author, Ed Nather (<utastro!nather>), on May 21, 1983. The story is set in 1959.

It is one of the most famous stories of programming.

```
A recent article devoted to the macho side of programming
made the bald and unvarnished statement:

    Real Programmers write in FORTRAN.

Maybe they do now,
in this decadent era of
Lite beer, hand calculators, and "user-friendly" software
but back in the Good Old Days,
when the term "software" sounded funny
and Real Computers were made out of drums and vacuum tubes,
Real Programmers wrote in machine code.
Not FORTRAN.  Not RATFOR.  Not, even, assembly language.
Machine Code.
Raw, unadorned, inscrutable hexadecimal numbers.
Directly.

Lest a whole new generation of programmers
grow up in ignorance of this glorious past,
I feel duty-bound to describe,
as best I can through the generation gap,
how a Real Programmer wrote code.
I'll call him Mel,
because that was his name.

I first met Mel when I went to work for Royal McBee Computer Corp.,
a now-defunct subsidiary of the typewriter company.
The firm manufactured the LGP-30,
a small, cheap (by the standards of the day)
drum-memory computer,
and had just started to manufacture
the RPC-4000, a much-improved,
bigger, better, faster -- drum-memory computer.
Cores cost too much,
and weren't here to stay, anyway.
(That's why you haven't heard of the company,
or the computer.)
```

```
I had been hired to write a FORTRAN compiler
for this new marvel and Mel was my guide to its wonders.
Mel didn't approve of compilers.

"If a program can't rewrite its own code",
he asked, "what good is it?"

Mel had written,
in hexadecimal,
the most popular computer program the company owned.
It ran on the LGP-30
and played blackjack with potential customers
at computer shows.
Its effect was always dramatic.
The LGP-30 booth was packed at every show,
and the IBM salesmen stood around
talking to each other.
Whether or not this actually sold computers
was a question we never discussed.

Mel's job was to re-write
the blackjack program for the RPC-4000.
(Port?  What does that mean?)
The new computer had a one-plus-one
addressing scheme,
in which each machine instruction,
in addition to the operation code
and the address of the needed operand,
had a second address that indicated where, on the revolving drum,
the next instruction was located.

In modern parlance,
every single instruction was followed by a GO TO!
Put that in Pascal's pipe and smoke it.

Mel loved the RPC-4000
because he could optimize his code:
that is, locate instructions on the drum
so that just as one finished its job,
the next would be just arriving at the "read head"
and available for immediate execution.
There was a program to do that job,
an "optimizing assembler",
but Mel refused to use it.

"You never know where it's going to put things",
he explained, "so you'd have to use separate constants".
```

```
It was a long time before I understood that remark.
Since Mel knew the numerical value
of every operation code,
and assigned his own drum addresses,
every instruction he wrote could also be considered
a numerical constant.
He could pick up an earlier "add" instruction, say,
and multiply by it,
if it had the right numeric value.
His code was not easy for someone else to modify.

I compared Mel's hand-optimized programs
with the same code massaged by the optimizing assembler program,
and Mel's always ran faster.
That was because the "top-down" method of program design
hadn't been invented yet,
and Mel wouldn't have used it anyway.
He wrote the innermost parts of his program loops first,
so they would get first choice
of the optimum address locations on the drum.
The optimizing assembler wasn't smart enough to do it that way.

Mel never wrote time-delay loops, either,
even when the balky Flexowriter
required a delay between output characters to work right.
He just located instructions on the drum
so each successive one was just past the read head
when it was needed;
the drum had to execute another complete revolution
to find the next instruction.
He coined an unforgettable term for this procedure.
Although "optimum" is an absolute term,
like "unique", it became common verbal practice
to make it relative:
"not quite optimum" or "less optimum"
or "not very optimum".
Mel called the maximum time-delay locations
the "most pessimum".

After he finished the blackjack program
and got it to run
("Even the initializer is optimized",
he said proudly),
he got a Change Request from the sales department.
The program used an elegant (optimized)
random number generator
```

```
to shuffle the "cards" and deal from the "deck",
and some of the salesmen felt it was too fair,
since sometimes the customers lost.
They wanted Mel to modify the program
so, at the setting of a sense switch on the console,
they could change the odds and let the customer win.

Mel balked.
He felt this was patently dishonest,
which it was,
and that it impinged on his personal integrity as a programmer,
which it did,
so he refused to do it.
The Head Salesman talked to Mel,
as did the Big Boss and, at the boss's urging,
a few Fellow Programmers.
Mel finally gave in and wrote the code,
but he got the test backwards,
and, when the sense switch was turned on,
the program would cheat, winning every time.
Mel was delighted with this,
claiming his subconscious was uncontrollably ethical,
and adamantly refused to fix it.

After Mel had left the company for greener pa$ture$,
the Big Boss asked me to look at the code
and see if I could find the test and reverse it.
Somewhat reluctantly, I agreed to look.
Tracking Mel's code was a real adventure.

I have often felt that programming is an art form,
whose real value can only be appreciated
by another versed in the same arcane art;
there are lovely gems and brilliant coups
hidden from human view and admiration, sometimes forever,
by the very nature of the process.
You can learn a lot about an individual
just by reading through his code,
even in hexadecimal.
Mel was, I think, an unsung genius.

Perhaps my greatest shock came
when I found an innocent loop that had no test in it.
No test.  None.
Common sense said it had to be a closed loop,
where the program would circle, forever, endlessly.
Program control passed right through it, however,
and safely out the other side.
It took me two weeks to figure it out.

The RPC-4000 computer had a really modern facility
called an index register.
It allowed the programmer to write a program loop
that used an indexed instruction inside;
each time through,
the number in the index register
was added to the address of that instruction,
so it would refer
to the next datum in a series.
He had only to increment the index register
each time through.
Mel never used it.

Instead, he would pull the instruction into a machine register,
add one to its address,
and store it back.
He would then execute the modified instruction
right from the register.
The loop was written so this additional execution time
was taken into account --
just as this instruction finished,
the next one was right under the drum's read head,
ready to go.
But the loop had no test in it.

The vital clue came when I noticed
the index register bit,
the bit that lay between the address
and the operation code in the instruction word,
was turned on --
yet Mel never used the index register,
leaving it zero all the time.
When the light went on it nearly blinded me.

He had located the data he was working on
near the top of memory --
the largest locations the instructions could address --
so, after the last datum was handled,
incrementing the instruction address
would make it overflow.
The carry would add one to the
operation code, changing it to the next one in the instruction set:
a jump instruction.
Sure enough, the next program instruction was
```

```
in address location zero,
and the program went happily on its way.

I haven't kept in touch with Mel,
so I don't know if he ever gave in to the flood of
change that has washed over programming techniques
since those long-gone days.
I like to think he didn't.
In any event,
I was impressed enough that I quit looking for the
offending test,
telling the Big Boss I couldn't find it.
He didn't seem surprised.

When I left the company,
the blackjack program would still cheat
if you turned on the right sense switch,
and I think that's how it should be.
I didn't feel comfortable
hacking up the code of a Real Programmer.
```

The original version was not in the free verse format; this evolved from forwards and re-forwards of the text. However, the author has noted that he prefers this version.