

170D WOBC Module R

x86 Assembly Programming

US Government

Table of Contents

1. Introduction.....	1
Why Learn Assembly?	2
Prerequisites	4
Terminology	5
Basic Knowledge.....	6
2. Executable Programs.....	7
End-to-end Process.....	8
Executable Formats.....	14
Sections.....	17
3. Tools to Create Executables.....	19
Introduction	19
gcc	20
make	21
The Gnu Debugger.....	22
Exercises.....	24
4. CPU.....	49
Pipelining.....	50
Modes	51
5. Registers	57
Intel Architecture History	59
Register Descriptions	70
Callee-Saved Registers	78
Exercises.....	78
6. Functions	83
Assembly Language Instructions.....	84
Stack Space	86
Base Pointer, Stack Pointer and Local Variables	87
Caller- and Callee-Saved Registers.....	89
x86_64 ABI	90
Exercises.....	94
7. Assignment	101
Assembly Language Instructions.....	102
The many forms of mov	103
Indirect Addressing.....	105
Load Effective Address.....	108
Segment Offsets	109
Sign Extension and Zero Extension.....	110
Exercises	114

8. Arithmetic	123
Assembly Language Instructions.....	124
No Operation.....	127
Basic Arithmetic Operations.....	128
Bitwise Instructions	132
Exercises	138
9. Conditionals, Jumps and Loops	143
Condition Flags.....	144
Condition Codes	146
Jumps	148
Conditional Moves.....	153
Loops.....	155
Exercises	157
10. Floats.....	163
ST Use	164
XMM Use	166
Exercises	172
11. Week 2, Monday, 7 Hour Encapsulation Lab.....	179
Exercise 1.....	180
Exercise 2.....	181
Exercise 3.....	182
Exercise 4.....	184
Exercise 5.....	185
Exercise 6.....	186
Exercise 7.....	187
Exercise 8.....	188
Exercise 9.....	188
Exercise 10.....	189
12. Review 1.....	193
Topics	194
Objective	195
Assembly Language.....	196
Tools	197
CPU	198
Registers.....	199
Functions.....	200
Assignments	201
Arithmetic	202
Branching	203
Floats.....	204
13. The Two Major Syntaxes	205

Intel	206
AT&T	208
Current Use	209
Comments	210
Directives	211
Macros	217
14. Memory and Latency	219
Structure	219
Memory Pages	220
Caches	221
Latency Numbers Every Programmer Should Know	223
Exercises	225
15. Strings.....	227
lod _s , st _{os}	228
sc _{as}	229
mov _s	230
cmp _s	231
rep, repz, repnz	232
cld, std	233
Exercises	234
16. Interfacing with Linux	243
System Calls.....	244
Command Line Parameters.....	246
Signal and Interrupts	248
Exercises	252
17. Structures.....	255
Bit Encodings	256
Comparison of Structures in C and in Assembly Language.....	260
Exercises	264
18. SIMD — Single Instruction, Multiple Data	269
Introduction	270
Instruction Set	273
Design Considerations	291
Data Organization	293
Exercises	295
19. Primitive Tools for Examining Data Files	307
Motivation	307
Magic Numbers	308
hexdump and xxd	309
strings	312
files	313

Exercises	314
20. Primitive Tools for Examining Executable Files	321
Introduction	321
nm	322
objdump	323
readelf	324
ltrace and strace.....	325
make	325
as	326
Instruction Encoding	327
Exercises	334
21. Week 4, Monday, 7 Hour Lab	339
General Requirements:	339
Exercises	339
22. Review 2.....	343
Topics	344
Objective	345
Syntax.....	346
Memory	347
Strings.....	348
Interfacing with Linux.....	349
Structures	350
SIMD	351
Primitive Tools for Examining Data	352
Primitive Tools for Examining Executable Files	353

Chapter 1. Introduction

This module is about integrating assembly language code with C.

Why Learn Assembly?

Assembly language is the lowest level code that most programmers can reasonably reach. It brings the programmer very close to the bare metal of the CPU. Assembly language offers the potential to write very compact code that executes very quickly.

Assembly language code makes the programmer a very aware of

- CPU architecture
- operating system architecture
- memory layout of data structures

Understanding how computer systems are exploited often requires understanding of:

- CPU architecture
- operating system architecture
- memory layout of data structures
- data file layout
- network communication protocols

Some of these topics are beyond the scope of this course. CPU architecture will be explored as part of this course.

Prerequisites

This course assumes that the student

- understands some common abbreviations such as CPU, RAM and OS
- can write C code that uses the Standard C Library
- knows how to use a Linux command line
- knows that CPUs have registers and flags
- that registers hold values
- that flags convey information about conditions

Terminology

x86 refers to an entire family of Intel CPUs. The first CPU in the family tree was the 16 bit Intel 8086 CPU. The second CPU in the family to receive widespread acceptance was the Intel 80286. The Intel 80386 was Intel's first 32 bit CPU. Because all of these chips were backward compatible, by the early 1980s, programmers started talking about them as a family. Hence the family name, x86. The name stuck even though Intel abandoned the 80x86 naming pattern. x86-64 refers to the 64 bit members of this family.

Basic Knowledge

In order to be able to program in any procedural language, a programmer must be able to:

- use the tools that create and debug executables
- store and retrieve values from variables
- perform simple mathematical operations
- call subroutines and return from them
- make decisions
- perform repetitive actions (loops)

Chapter 2. Executable Programs

Every program must be loaded into memory to be run. In order for that to happen, it must be built in such way that the operating system can load it.

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. Think of the preprocessor as a very limited text editor. It has very restricted abilities. Usually it does no more than conditional expansion based on variables or simple expressions.

Compiling

The **Compiler** program, `cc`, is a complex piece software. 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

All of the preceding steps were performed by developers or development teams. The loader, on the other hand, is part of the operating system. As the name suggests, the loader's job is to load a program into memory and execute it. When a program starts, the operating system loader allocates memory for the process. The code is loaded into an executable portion of that memory while data is loaded into a different—presumably non executable—section of memory. Then the operating system begins executing instructions from the program, one at a time.

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

`_start` is an C runtime library function that calls `main` with the appropriate arguments. When `main` returns, it passes control back to `_start`. 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 operating system can load the program's code and data to memory and 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 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 ASCII characters “ELF”. This makes their magic number 0x7F454C46. UNIX operating systems understand the ELF format.

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. Mark Zbikowski is the Microsoft Architect who designed the Windows executable format. The magic number for Windows executables is 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 contain 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 be 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. This is where the **Section Table** comes into play. This is a table that describes each section:

- how large it is,
- what kind of data it contains,
- how it should be loaded to form a process image.

Chapter 3. Tools to Create Executables

Introduction

Some programs are indispensable to the process of turning source code into an executable file.

- gcc
- make
- gdb

gcc

gcc controls the entire process of turning source code into an executable file. gcc can perform all or part of that process. See the man page for `gcc` for details.

Options are case sensitive. (See the man page for details.) Some commonly used options include:

- `-o <output file name>` Pick a name for the output file. The default name is `a.out`
- `-S` Stop after creating assembler language from the source code. If this option is used, the `-o` option is almost always specified.
- `-c` Compile or assemble the source files, but do not link. If this option is used, the `-o` option is almost always specified.
- `-E` Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code. The output is sent to the standard output unless `-o` specifies a file name.
- `-masm=intel` Generate assembly language in Intel (rather than AT&T) format.
- `-std=c18` Compile C source code using the 2018 version of the ANSI C Standard.
- `-Ox` Controls the amount of optimization. The number that follows the letter "O" can be zero or 1 or 2 or 3. The higher the number, the more optimization. See the man page for other possibilities.
- `-gx` Controls debugging information. The number that follows the letter "g" can be zero or 1 or 2 or 3. The higher the number, the more debugging information. See the man page for other possibilities.
- `-fno-asynchronous-unwind-tables` Remove call frame information from output.

make

A utility that builds an executable. `make` uses a script that is typically called `Makefile` to automate the build process. See the man page for `make` for details.

Options are case sensitive. (See the man page for details.) Some commonly used options include:

- `--trace` Causes `make` to describe the steps as `make` runs.
- `-d` Causes `make` to print voluminous debugging information.
- `-B` Unconditionally make all targets
- `-n` Dry run. Just report what commands would be executed, but do not actually execute them.
- `-r` Don't use any of `make`'s built-in implicit rules.
- `-R` Don't use any of `make`'s built-in variables.

The Gnu Debugger

gdb is the GNU Debugger. gdb is programmable to a large extent, and can have custom functions written for it in a simple language.

Frequently used gdb commands

command	description	example
break	Set a breakpoint	'b main'
run	Start executing from the beginning	r
	Optionally, run with arguments	r -x 27 -b 50
show args	Show the command line arguments that were passed	sho args
continue	Continue executing from current breakpoint	c
next [count]	Execute the current line of source code. Execute across calls.	n
nexti [count]	Execute the current assembler instruction. Execute across calls	ni 6
step [count]	Execute the one line of source code. Descend into calls	s 3
stepi [count]	Execute the current assembler instruction. Descend into calls	si
finish	Execute until the end of the current subroutine	fin
x	Examine memory. The expression that follows x determines what gets examined and how it gets displayed	x /s *SomeAddress
print	print the value of a register or an expression	p \$rax

It is not necessary to type the complete command. There are shortcuts for the most frequently used

commands.

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. The name of the register needs a dollar sign in front of it — \$rcx

For printing register values, gdb provides four “generically” named registers. These registers usually shadow platform-specific registers. In addition, gdb provides access to the platform-specific registers. All of these pseudo registers require a \$ in front of them.

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

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
layout src
layout regs
focus cmd
```

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

```
$ python2 guiTimer.py $
```

```
$ ps -ef ... student 31058 18817 2 13:14 pts/5
00:00:00 python2 guiTimer.py ... $ gdb -p
31058
```

Exercises

Exercise 1

Read the `man` page for `gcc`

Instructions:

Open the `man` page for `gcc`. This is one of the longest `man` pages.

Make sure to read the `Overall Options` section of the `man` page. Make sure you read the description for:

- `-c`
- `-S`
- `-E`
- `-O`

The options above control the stages of compilation. You will see them in many development organizations.

Find the options that control optimization. Notice that there are quite a few. It is not necessary to read all of them. Make sure to read the description for `-O1`. That is the option that the class will use.

Scan through the possibilities and make a note of any you might want to experiment with during our labs.

Find the options that control the dialect of the C programming language. Look up the effect of using:

- `'-std=c11'`
- `'-std=c17'`
- `'-std=c18'`

Exercise 2

Create a Simple Shell of a C Program

Purpose:

Create a simple shell of a C program that can be:

- used to test future assembly language code.
- used as the subject of a `makefile`.

Introduce the student to some of the command line options for 'gcc'.

Instructions:

Create a directory to hold your work from this chapter's exercises. These instructions will call the folder `developmentTools`. The name of the directory is not critical. You may call the directory anything you wish. You will be copying files from this folder for a future exercise.

Create a file called `SeaShell.c`

Write the code you see. Save the file. The exercises that follow will use this file as a test harness for your assembly language code.

SeaShell.c

```
1 /*
2  * SeaShell.c
3  *
4  */
5
6 #include <time.h>
7 #include <string.h>
8 #include <stdio.h>
9
10 int testHarness( char* s01, char* s02, char* s03 ) {
11     int resultCode = 0;
12
13     puts( s01 );
14     puts( s02 );
15     puts( s03 );
16
17     return resultCode;
18 }
19
20 void reportTime(void) {
21     time_t timeStructure;
22     time( &timeStructure );
23     char *s = ctime( &timeStructure );
24     puts( s );
25 }
26
27 int main(int argc, char* argv[] ) {
28     int exitCode = -99;
29     reportTime();
30
31     exitCode = testHarness( argv[0], argv[1], argv[2] );
32
33     reportTime();
34     return exitCode;
35 } // end main() - - -
```

Compile the file. There should be no errors and no warnings.

```
gcc SeaShell.c
```

Execute the code to prove that it works. Make sure that you provide at least three command line parameters when you invoke a.out or the executable will crash.

```
./a.out one Mississippi two Mississippi
```

The output should look something like this:

```
Fri Apr 24 22:13:57 2020 ./a.out one mississippi Fri Apr 24 22:13:57 2020
```

Why do we not see two Mississippi in the output? (Check the source code in the testHarness() function and in main() for the answer.)

Compile the file with a different gcc option. This time, we will give the executable file a different name. There should be no errors and no warnings.

```
gcc -o SeaShell SeaShell.c
```

Execute the code to prove that it works. Our program has a different name: SeaShell

```
./SeaShell one Mississippi two Mississippi
```

The output should look almost the same as before. What changed in the output? Explain the change.

Compile the file with yet another new gcc option. This time, we will optimize the executable. The option is a capital letter 'O'. There should be no errors and no warnings.

```
gcc -O3 -o SeaShell SeaShell.c
```

Execute the code to prove that it works. Our program has a different name: SeaShell

```
./SeaShell one Mississippi two Mississippi
```

The output should look the same as before. The -O3 option causes gcc to optimise the code it produces. You won't see any change in the output from the executable, but the code will run faster.

This time, change two things in the compilation.

- Use the `-o` parameter to change the name of the output file to `SeaShell.s`
- Use the `-S` parameter to tell `gcc` to translate the C code into x86 assembly language code.

There should be no errors and no warnings.

```
gcc -S -O3 -o SeaShell.s SeaShell.c
```

The output from the compilation has a different name: `SeaShell.s`

Examine `SeaShell.s` in a text editor. By default, `gcc` creates AT&T assembly language. This will change in the next step.

Make gcc create Intel flavor assembly language for a 64 bit architecture.

```
gcc -m64 -masm=intel -S -O3 -o SeaShell.s SeaShell.c
```

Examine SeaShell.s in a text editor.

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 assembly language code in this course.

Is there a file name mentioned near the beginning of the file? What is that file name? What does that file name tell you?

What are the names of every publicly visible label in this file?

How can you tell those labels are publicly visible?

Do those publicly visible labels correspond to anything in the SeaShell.c file?

Compile the assembly language file, SeaShell.s

```
gcc SeaShell.s
```

Check the timestamp on a.out in order to make sure that this is the file you just created.

Execute a.out

```
./a.out one Mississippi two Mississippi
```

The output should look something like this:

```
Sat Apr 25 12:13:57 2020 ./a.out one mississippi Sat Apr 25 12:13:57 2020
```

Notice that both the C language source and the assembly language source can be compiled to produce the same executable output.

Summary:

You used `gcc` with several different parameters in order to become familiar with some of the more commonly used parameters. The next step is to use `make` to automate parts of the build process

Exercise 3

Read the `man` page for `make`

Purpose:

Different environments have very slight differences in their implementation of `make`.

Instructions:

Open the `man` page for `make`. Find the acceptable name(s) for a make file. Disregard any name that starts with `gnu`.

Pick an acceptable name for a make file in our lab environments. Use it as the name for every make file in every lab for the rest of the course.

Find the options that control debugging makefiles. Which options have synonyms? At a minimum, there should be:

- `-d`
- `--trace`
- `-r`
- `-R`

Scan through the possibilities and make a note of any you might want to experiment with during our labs.

See if you can find a synonym for the `-B` option. Scan through the other possibilities and make a note of any you might want to experiment with during our labs.

Exercise 4

Create a `makefile` for a Simple C Program

Purpose:

It is obvious gcc can require many options. In a production environment, a command line for gcc can be quite long. Typing a long command line many times can be tedious and error prone. A properly written `makefile` can also make enforcement of certain standards easier to achieve.

Instructions:

Create a makefile by typing the following. Remember what you learned about legal names for makefiles from an earlier exercise.

makefile

```
1 FILENAME = SeaShell
2 CFILE = $(FILENAME).c
3 ASMFILE = $(FILENAME).s
4 OUTFILE = $(FILENAME)
5 CC = gcc
6 ASFLAGS += -m64 -masm=intel
7 CFLAGS += -fno-asynchronous-unwind-tables -std=c11 -mfpmath=sse
8 DIAGNOSTIC += -Wall -Wextra -Wpedantic -Waggregate-return -Wwrite-strings -Wfloat-
equal -Wvla
9 OPTIMIZATION += -O1
10
11 $(OUTFILE) : $(ASMFILE)
12     $(CC) $(ASFLAGS) $(DIAGNOSTIC) $(OPTIMIZATION) -o $(OUTFILE) $(ASMFILE)
13
14 $(ASMFILE) : $(CFILE)
15     $(CC) $(CFLAGS) $(ASFLAGS) $(OPTIMIZATION) $(DIAGNOSTIC) -S -o $(
ASMFILE) $(CFILE)
```

This part of the makefile says that the executable file depends on the assembly language file.

```
$(OUTFILE) : $(ASMFILE)
```

If the assembly language file is newer than the executable file, make will build the executable from the assembly language file.

```
$(CC) $(ASFLAGS) $(DIAGNOSTIC) $(OPTIMIZATION) -o $(OUTFILE) $(ASMFILE)
```

If the C language file is newer than the assembly language file, make will build the assembly language file from the C language file. Make files can get very, very complex.

Test your new `makefile` by typing

```
make --trace -B
```

There will almost certainly be error messages because of the very strict error checking set by `$(DIAGNOSTIC)`. Those errors will be fixed in the next step.

Modify the `main()` function in the C source code as shown below.

part of SeaShell.c

```
1 int main(int argc, char* argv[] ) {
2     int exitCode = -99; // initialize to a dummy value
3     /* ADD THIS CODE v v v */
4     // Make a string buffer long enough to hold a 60 character program name
5     char *numberOfParametersPassed = calloc( 1, 100 );
6
7     // Write some text into our buffer
8     sprintf( numberOfParametersPassed, "%2d parameters were passed by %s", argc,
9     argv[0] );
10    /* END OF CODE TO ADD ^ ^ ^ */
11    reportTime();
12
13    // Display the number of parameters passed to us on the command line
14    puts( numberOfParametersPassed ); /* < - - ADD THIS LINE TOO!!! */
15
16    /* Regardless of how many parameters were passed to us,
17     * we only use the first three parameters.
18     * The exitCode becomes whatever the testHarness() function returns.
19     */
20    exitCode = testHarness( argv[0], argv[1], argv[2] );
21
22    reportTime();
23    return exitCode;
24 } // end main()      -      -
```

Add three lines of code to `main()` as shown in the comments. These three lines use the `argc` parameter. Adding these three lines will require another include file.

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

The error messages that the makefile enabled report unused parameters as a problem. A solution to that problem is to write some code that uses `argc` and `argv`.

Test your modified file by building it.

`make`

There should be no errors or warnings. Execute the code as before.

`./SeaShell one Mississippi two Mississippi`

Exercise 5

Debug an Executable Using gdb

Purpose:

Practice doing several common debugging tasks.

Instructions:

Rebuild the SeaShell executable with debugging information.

makefile

```
1 FILENAME = SeaShell
2 CFILE = $(FILENAME).c
3 ASMFILE = $(FILENAME).s
4 OUTFILE = $(FILENAME)
5 CC = gcc
6 ASFLAGS += -m64 -masm=intel
7 CFLAGS += -fno-asynchronous-unwind-tables -std=c11
8 DIAGNOSTIC += -Wall -Wextra -Wpedantic -Waggregate-return -Wwrite-strings -Wfloat-
equal -Wvla -g3
9 OPTIMIZATION += -O1
10
11 $(OUTFILE) : $(ASMFILE)
12     $(CC) $(ASFLAGS) $(DIAGNOSTIC) $(OPTIMIZATION) -o $(OUTFILE) $(ASMFILE)
13
14 $(ASMFILE) : $(CFILE)
15     $(CC) $(CFLAGS) $(ASFLAGS) $(OPTIMIZATION) $(DIAGNOSTIC) -S -o $(
ASMFILE) $(CFILE)
```

Notice that the line that begins with DIAGNOSTIC has a new option at the very end.

```
DIAGNOSTIC += -Wall -Wextra -Wpedantic -Waggregate-return -Wwrite-strings -Wfloat-equal -Wvla
-g3
```

Add `-g3` to the `DIAGNOSTIC` line of your `makefile`. That option add a lot of information to the executable file that will make using a debugger easier. There is no requirement that the options in this part of the `makefile` appear in any special order.

Rebuild the SeaShell executable. There should be no errors and no warnings.

Start the debugger.

```
gdb SeaShell
```

Set a breakpoint at the start of `main` and at the start of `testHarness`.

```
break main break testHarness
```

Begin executing SeaShell by typing the command `r`. The debugger should tell you that you are

stopped in main and reference a line number in your .c file. Now type the following line:

```
r 25 62 Mary had a little lamb
```

The debugger will ask you if you want to start execution from the beginning. Answer *y*. The debugger should tell you that you have some parameters. You can see them by typing:

```
show args
```

If you type *n* once or twice, execution will advance by one line of C source code for each time you hit *n*. If there are any calls to subroutines, execution will step over them. If you type *si* once, execution will advance to the next assembly language line. If there are any calls to subroutines, execution will step into them.

Notice that any registers that have new values in them are highlighted.

Type *c* to run to the next breakpoint. It is possible that the debugger screen has gotten a little messy. Type *refresh* to clean that up.

Type *finish* to run to the end of *testHarness*. Feel free to experiment with other commands. Type *c* to run to the end of *SeaShell*.

Type *q* to quit the debugger.

Summary:

You have performed several common gbd actions.

Exercise 6

Modify the Makefile to Compile and Link Two Source Files

Purpose:

To create a makefile that will use SeaShell.c as the test harness for assembly language code. You will use a modified version of this makefile in many future exercises.

Instructions:

Modify your makefile as shown below.

makefile

```
OUTFILE := SeaShell
TESTCODE := SeaShell
ASMFILE := returnval

CC := gcc
ASFLAGS += -m64 -masm=intel
CFLAGS += -fno-asynchronous-unwind-tables -std=c11
DIAGNOSTIC += -Wall -Wextra -Wpedantic -Waggregate-return -Wwrite-strings -Wfloat-equal -Wvla -g3
OPTIMIZATION += -O1

$(OUTFILE) : $(TESTCODE).o $(ASMFILE).o
    $(CC) $(ASFLAGS) $(DIAGNOSTIC) $(OPTIMIZATION) -o $(OUTFILE) $(TESTCODE).o
    $(ASMFILE).o

$(ASMFILE).o : $(ASMFILE).s
    $(CC) $(ASFLAGS) $(DIAGNOSTIC) $(OPTIMIZATION) -c -o $(ASMFILE).o $(ASMFILE).s

$(TESTCODE).o : $(TESTCODE).c
    $(CC) $(CFLAGS) $(ASFLAGS) $(OPTIMIZATION) $(DIAGNOSTIC) -c -o $(TESTCODE).o
    $(TESTCODE).c
```

Notice that:

- \$(OUTFILE) now depends on two object files.
- There is a rule for each source file.
- The rules for each source file stop after creating an object file. That is what the -c option does in each of those two rules.

Summary:

This will be the pattern for the makefiles that you use in many exercises for the rest of this module.

Exercise 7

Modify the SeaShell Test Harness to Make It Drive Assembly Language Code

Purpose:

The SeaShell test harness will be modified to provide testing for each exercise. The purpose of the main() function is to accept any command line parameters and, if necessary, convert those parameters from text to numeric values. The purpose of the testHarness() function is to test any assembly language code you write by passing values to that code and evaluating the results.

Instructions:

Change the header files that SeaShell.c imports.

header files

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

extern int myAnswer(void);
```

The extern line tells the C compiler that some other file contains a subroutine that:

- is called myAnswer
- does not expect to receive any parameters
- will return an int

An extern is needed for every assembly language subroutine that you intend to call from C.

Change the testHarness() method as shown below.

modified testHarness() function

```
int testHarness( void ) {
    int resultCode = 0;

    printf( "The answer to everything is %d\n", myAnswer() );

    return resultCode;
}
```

NOTE

In future exercises, testHarness() will hold code that will test the assembly language code you write.

Main will not be processing command line parameters in this exercise. Change the code in main() as shown below.

modified main() function

```
int main(void) {
    int exitCode = -99;
    reportTime();

    exitCode = testHarness();

    reportTime();
    return exitCode;
}
```

NOTE In future exercises, `main()` will contain code that captures command line parameters, parses and transforms them as necessary, and passes values to `testHarness()`.

Summary:

The code in `SeaShell.c` is ready to test the assembly language code.

Exercise 8

Write a Simple Assembly Language subroutine

Purpose: To demonstrate how most assembly language subroutines return values to their callers.

Instructions:

Write the following code. Save it in a file named `returnval.s` in the same folder where you put the makefile and the `SeaShell.c` test driver.

Contents of returnval.s

```
.intel_syntax noprefix  
  
.globl myAnswer  
myAnswer:  
    mov eax, 42  
    ret
```

The first line tells the assembler that the source code in this file will obey the Intel syntax rules. The `.globl` directive tells the linker that code outside of `returnval.s` will be able to call this subroutine. `myAnswer:` with a colon is a label. Labels mark locations. In this case, the location is the beginning of a function. In future exercises, a label could mark a location where data is stored or a location where code should jump.

The heart of the program is the line `mov eax, 42`. Not surprisingly, this moves a value into the `eax` register, also known as the **accumulator**. The `ret` instruction returns to whatever code called this subroutine.

Summary:

NOTE

For x86 assembly programming, most of the time, the return value from a function is the value in the accumulator.

Most assembly language subroutines return values to their callers by `mov`ing those values to the accumulator and `ret`urning.

Exercise 9

Examine the Executable in a Debugger

Purpose:

To explore the interface between C and assembly language code.

Instructions:

Build this small project. Make sure the makefile compiles all code with the -g3 compiler option.

Execute the executable from the command line to make sure it runs.

Start the executable in a debugger. Set two breakpoints.

```
break testHarness  
break myAnswer
```

Run the code.

↴

When execution stops at the testHarness() breakpoint, the code display should look something like this.

```
#include <time.h>  
#include <stdio.h>  
  
extern int myAnswer(void);  
  
int testHarness( void ) {  
    int resultCode = 0;  
    printf( "The answer to everything is %d\n\n", myAnswer() );  
    return resultCode;  
} /* end testHarness() - - - */
```

If C language source code does not show, try typing:

```
layout src
```

If the register group does not show or it disappears at any time, try typing:

```
layout regs
```

Examine the assembly language for testHarness(). Type:

```
layout asm layout regs
```

Execute the code by stepping into each line. Type 's' as often as you have to in order to step into myAnswer(), execute each line of assembly language code and return back to 'testHarness(). Each

time you type `s`, notice which registers change value. Notice that the `rax` register holds 42 when you get back to `testHarness()`. That integer is the return value from `myAnswer()`.

Don't worry about the call to `printf()` for now. Type '`fin`' to return to `main()`. You might have to type `refresh` in order to clean up your screen.

Quit the debugger by typing `q`.

Summary:

Modern assembly language code uses registers to pass values from one function to another. The `rax` register will hold integer return values and address return values.

In the `gdb` debugger, if the executable was compiled and linked with debugging information, it is possible to switch back and forth between C source and the assembly language that the compiler created by using various forms of the `layout` command.

Chapter 4. CPU

The central processing unit (CPU) is the part of the computer that executes all code. It is the brain of any computer.

The x86 architecture is a **Complex Instruction Set Computer**, or CISC. Some instructions may take ten or more cycles to complete. Others might only take one. As a result, it is sometimes hard to predict the performance of a given series of instructions.

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 states 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 issues when it comes to predicting which instruction to be executed next. In a long series of instructions that have no conditions and no loops, the next instruction to fetch is 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 approaches to evaluate the problem. Some are based on advanced statistical math and years of experience. Some are not.

- stochastic models of previous times the code has been encountered,
- following both branches simultaneously (then deciding afterward which one to retire),
- some rule-of-thumb guesses.

Modes

Due to backwards compatibility concerns, a x86 CPU has a number of modes of execution, each with different capabilities and different 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 1 MiB 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 and revert to real mode during reset. Part of any modern operating system initialization almost always involves a transition to higher addressing modes.

Protected Mode

Protected Mode allows for 4 GiB of addressable memory. This mode first appeared in the Intel 80286. 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**.

Protection Rings

In protected mode, there are four privilege levels, also known as **rings** or **protection rings**. Ring 0 is the most privileged. Code that executes at Ring 0 can execute any instruction.

Ring 3 is the least privileged. Code that executes at Ring 3 is restricted from executing certain instructions. Any attempt to execute a privileged instruction will trigger an exception.

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
- 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. For this reason, UNIX (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. Therefore, 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 extremely 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.

Any CPU has **user accessible registers** and **internal registers**. As the name implies, internal registers are not documented. They are reserved for manufacturers use only. Assembly language code uses the user accessible registers.

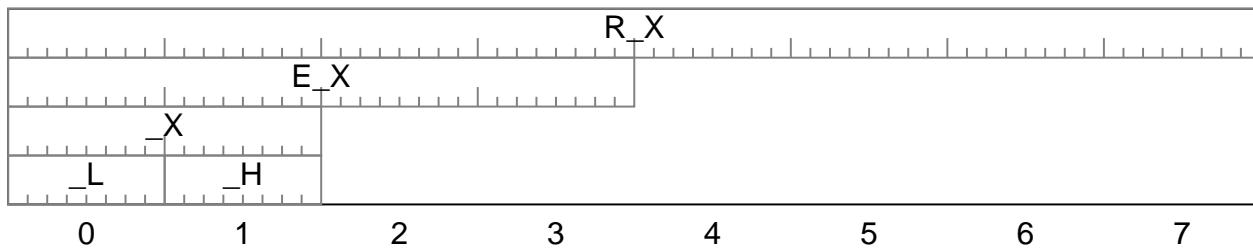
Some registers are read only by your code. An example would be the instruction pionter—rip on Intel chips — and the various status flags registers.

Different CPUs have different numbers of user-accessible registers. For example, an ARM CPU has 63 user-accessible registers, while an Intel 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. Other 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.

Registers generally hold one CPU word of storage. Depending on the CPU, a word can be anything from 8 bits to 512 bits. Some architectures — Intel x86_64 CPUs are an example — may have special registers that hold a different amount. They are referred to by their bit-width, such as 128 bit register or 80-bit register.



Intel x86_64 registers can be referenced by either their entire amount or a smaller subset of bytes. A 64-bit register (for example `rax`) contains a 32 bit register (`eax`).

The 32 bit `eax` register contains a 16 bit `ax` register. The `ax` register contains two 8 bit registers (`ah` and `al`).

In Intel architecture, the `A` register was the accumulator. The `D` register was the data register. The `C` register was the register that held counts. Over time, these registers became `rax`, `rcx` and `rdx`.

The source index and the destination index became `rsi` and `rdi`. The instruction pointer became `rip`. The base pointer became `rbp`. The stack pointer became `rsp`.

All of these registers will be discussed as we explore Intel's instruction set.

Intel Architecture History

The Intel architecture that evolved to become x86_64 began in 1974 with an 8 bit 8080 chip. The key to Intel's market dominance has been maintaining backwards compatibility across close to half a century of technology.

8080

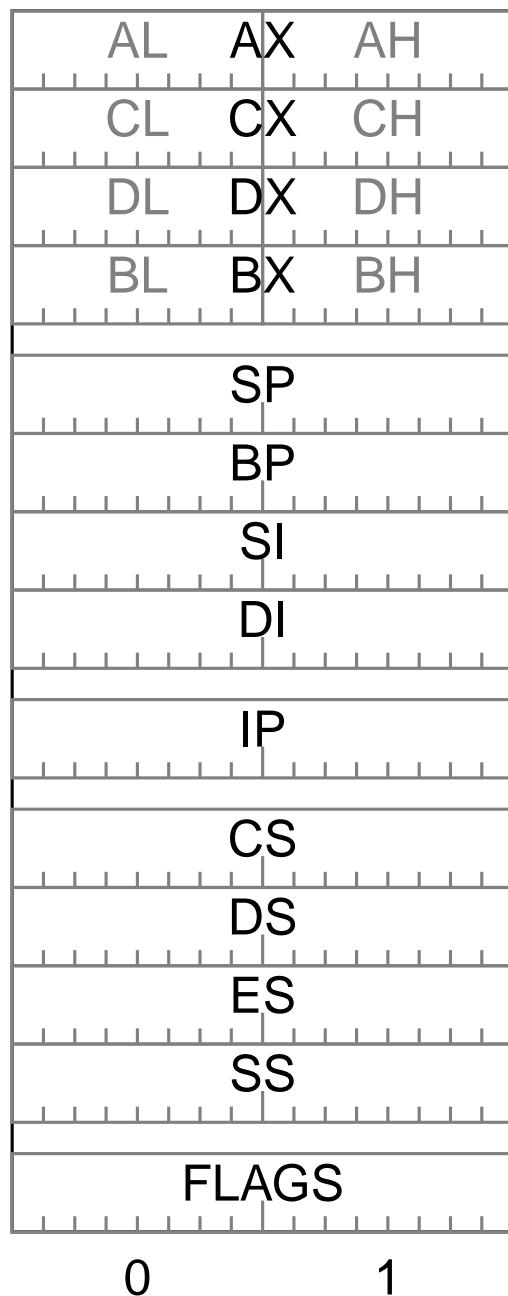
The Intel 8080 8 bit CPU replaced the (binary incompatible) 8008 chip. It had the following 8-bit registers:

FLAGS	A
C	B
E	D
L	H
SP	
PC	

In addition to the two 16 bit registers, 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:



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 64KB of memory could be addressed. The 8086 used a compound form of addressing known as **memory segmentation** to address up to 1MB of memory. The value in a **segment register** (cs, ds, es, or ss) was multiplied by 16 and added to a requested memory address.

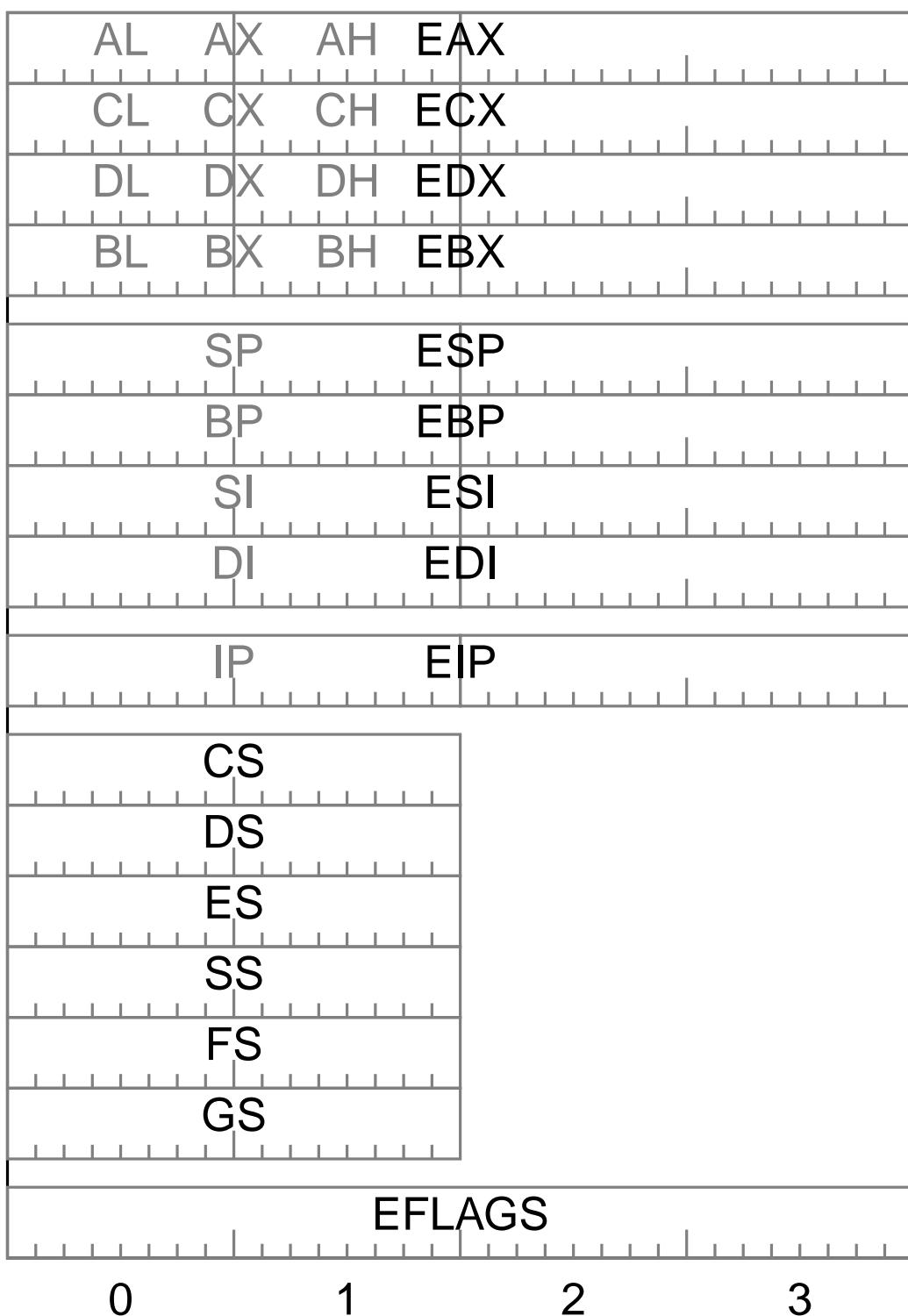
$$0xABBA:0xDEAD = (0xABBA \times 0x10) + 0xDEAD = 0xB9A4D$$

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

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

386

The Intel 386 chip expanded the word size of the previous 8086 and 80286 chips to 32 bits. Intel wanted to be able to 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.



ST0
ST1
ST2
ST3
ST4
ST5
ST6
ST7

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

Pentium

The Intel Pentium added parallelizable registers, but these 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.

	ST0	MMX0
	ST1	MMX1
	ST2	MMX2
	ST3	MMX3
	ST4	MMX4
	ST5	MMX5
	ST6	MMX6
	ST7	MMX7

Figure 1. Pentium ST/MMX Registers

Although the MMX registers are actually only 64 bits wide, they overlapped the 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. A single instruction can be applied to 64 bits worth of integers at once:

- two 32-bit integers or
- 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 registers are special SIMD registers that are 128 bits wide.

XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7

Figure 2. 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 module.

Opteron

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

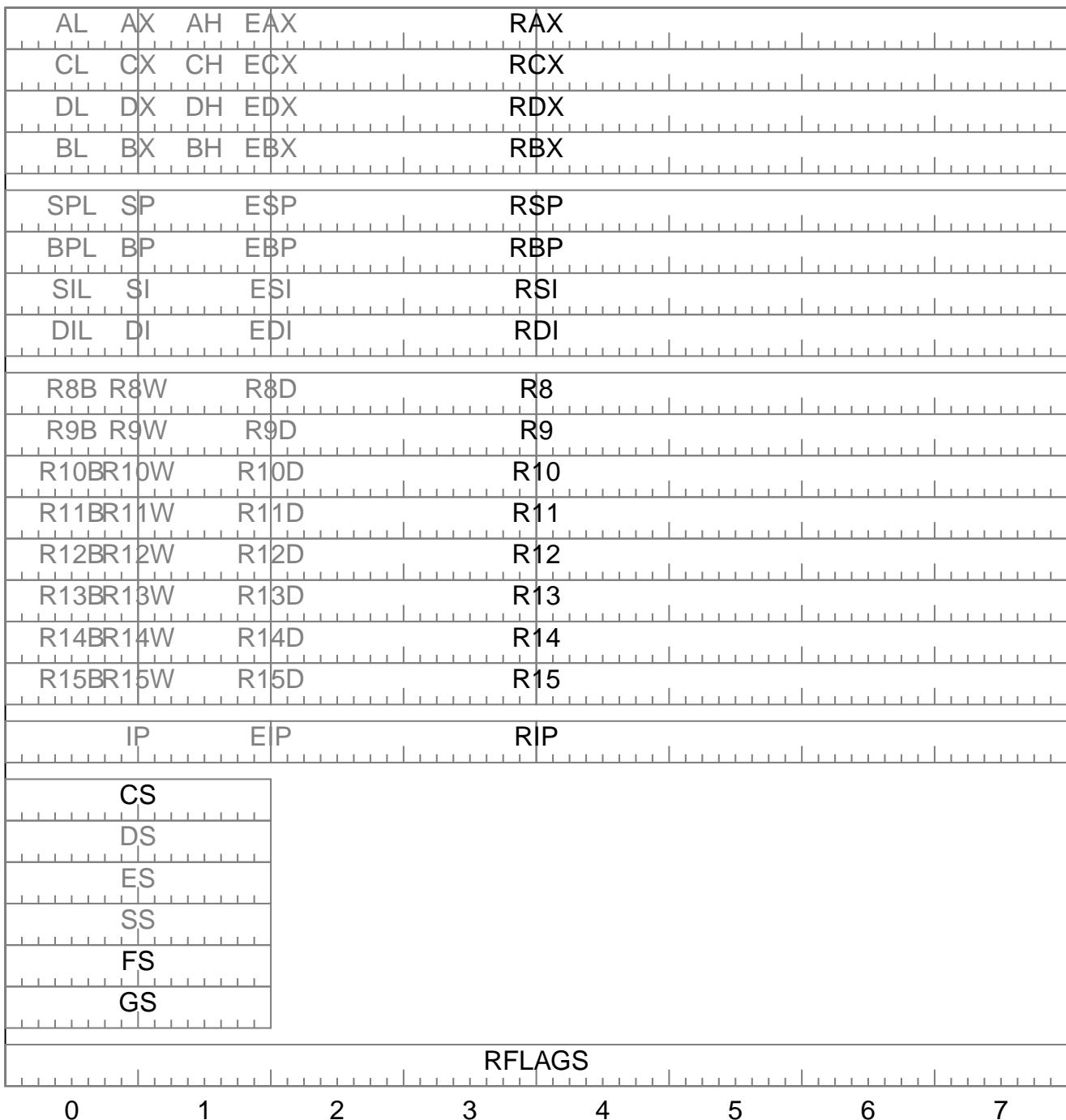


Figure 3. x86-64 Registers

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.

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

Core i3

For the later Haswell microarchitecture, Intel added 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 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**. This is the `rip` register.

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 1. 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
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	

Bit	Abbr	Name	
22–63			Reserved

Accumulator

In the oldest computers, the **accumulator** might be the only register. All calculations are performed in the accumulator. By convention, all return values are stored in the accumulator. Intel's accumulator is the `rax` register.

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. The hardware itself is optimized for this kind of use.

NOTE

In x86_64 assembly programming, the value returned by any function is almost always found in the accumulator, `rax`.

Counter

The **count** register closely maps to index variables in loops. Intel's count register is `rcx`.

Some highly compact instructions make use of `rcx`. 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 `rcx`.

Data

Calculations in the accumulator may extend themselves into the **data** register. Intel's data register is `rdx`. 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. 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 manipulate this register. Intel's stack pointer is `rsp`.

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

Base Pointer

In x64 assembly, the **base pointer** is a general-purpose register. Intel's base pointer register is `rbp`.

In x86 chips prior to x64, the base pointer served as a current frame pointer. It held 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. There are some specialized instructions that require it. Intel's source index is the `rsi` register.

Destination Index

The **destination index** register points to the destination for memory writes by some specialized instructions. Most of the time, it is used as a general-purpose register. Intel's destination index is the `rdi` register.

NOTE

In modern `x86_64` assembly programming, the first argument to any function is found in the destination index, `rdi`.

r8 through r15

x86_64 architecture also has 8 general-purpose registers that are identified by number. Those registers are r9, r10, r11, r12, r13, r14, r15.

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. If you need these registers, push the caller's values on the stack before you use them. Pop the caller's values off the stack before you return.

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

Exercises

Exercise 1

Obtain Intel CPU Information

Purpose:

The purpose of this exercise is to give you an opportunity to learn to explore some Intel registers with a debugger.

Instructions:

It is not necessary to change the makefile for this exercise.

Add another subroutine to `returnvals.s`

modified returnvals.s

```
.intel_syntax noprefix

.globl myAnswer
.globl getCPUInfo

myAnswer:
    mov eax, 42
    ret

getCPUInfo:
    mov rax, 0
    cpuid
    ret
```

Notice a new global label—`getCPUInfo`. This is the name of the subroutine that will be called from `testHarness()`.

The subroutine sets the `rax` register to zero. This is required by the `cpuid` instruction.

The `cpuid` instruction orders an Intel cpu to return some information about itself. If the instruction is called with a zero in the `rax` register, it returns the text "GenuineIntel" spread across three

registers. We will examine this result when we execute this code in a debugger.

Modify the code in `testHarness()` so that it will call `getCPUInfo()`. Add two lines of code where indicated in the listing below.

modified SeaShell.c

```
extern int myAnswer(void);
extern void getCPUInfo( void ); /* ADD THIS LINE!!! */

int testHarness( void ) {
    int resultCode = 0;

    printf( "The answer to everything is %d\n\n", myAnswer() );

    getCPUInfo(); /* ADD THIS LINE!!! */

    return resultCode;
} /* end testHarness() - - */
```

Make the executable. Make sure your `makefile` still uses the `-g3` option for debugging.

Run the executable in a debugger.

Set a breakpoint at the start of the new subroutine.

```
break getCPUInfo
```

Run to that breakpoint. It might be necessary to clean up the debugger display by typing one or more of the following lines:

```
layout regs refresh
```

Make a mental note of the contents of the `rbx`, the `rcx` and the `rdx` registers. Those contents are very likely to change when the `cpuid` instruction executes.

```

Register group: general
rax          0x20          32
rbx          0x555555555230    93824992236080
rcx          0x0           0
rdx          0x0           0
rsi          0x555555559540    93824992253248
rdi          0x7ffff7fb34c0    140737353823424
rbp          0x0           0x0
rsp          0x7fffffffdf38    0x7fffffffdf38
r8           0x0           0
r9           0x20          32
r10          0x555555556026    93824992239654
r11          0x246          582
r12          0x555555555090    93824992235664
r13          0x7fffffff040     140737488347200
r14          0x0           0

returnval.s
 7           mov      eax, 42
 8           ret
 9
10          getCPUInfo:
B+>11         mov      rax, 0
12          cpuid
13          ret
14

```

Figure 4. CPU registers BEFORE cpuid executes

Execute the cpuid instruction and notice any change in rbx, rcx and rdx. (Both next and step work equally well for this.)

```

Register group: general
rax          0xd                  13
rbx          0x756e6547            1970169159
rcx          0x6c65746e            1818588270
rdx          0x49656e69            1231384169
rsi          0x5555555559540        93824992253248
rdi          0x7fffff7fb34c0        140737353823424
rbp          0x0                  0x0
rsp          0x7fffffffdf38        0x7fffffffdf38
r8           0x0                  0
r9           0x20                 32
r10          0x555555556026        93824992239654
r11          0x246                582
r12          0x555555555090        93824992235664
r13          0x7fffffff040         140737488347200
r14          0x0                  0

returnval.s
 7             mov      eax, 42
 8             ret
 9
 10            getCPUInfo:
B+ 11            mov      rax, 0
 12            cpuid
>13            ret
 14

```

Figure 5. CPU registers AFTER cpuid executes

If the cpuid instruction fails, the `rax`, `rbx`, `rcx` and `rdx` registers are all set to zero.

If the cpuid instruction succeeds on a genuine Intel CPU, the `rbx`, `rcx` and `rdx` registers will look like the image above. Treat every pair of hex numbers in `rbx`, `rcx` and `rdx` as ASCII character codes. Look up the characters for each of those ASCII codes. (Google if you have to.) How do you have to arrange the order of the three registers in order to make the characters spell "GenuineIntel"?

The value in `rax` describes the number of different pages of descriptive information your code can recover about this particular Intel cpu.

Quit the debugger. The exercise is finished.

Summary:

In this exercise:

- you used a debugger to set a breakpoint on the first line of an assembly language subroutine
- you stepped through some assembler code

- you examined CPU registers

Chapter 6. Functions

As a reminder, in order to be able to program in any procedural language, a programmer must be able to:

- use the tools that create and debug executables
- store and retrieve values from variables
- perform simple mathematical operations
- call subroutines and return from them
- make decisions
- perform repetitive actions (loops)

This unit is about assembly language instructions that

- call and return from functions
- carve space from the stack for local variables inside functions
- clean up the stack before returning from the function

Assembly Language Instructions

call	calls a function
ret	returns from a function
push	adds an item to the stack
pop	removes an item from the stack
add	in this context, used to create space on the stack for local variables. Called when entering a function (see below)
sub	in this context, used to reclaim space on the stack that was used by local variables. Called when leaving a function (see below)
enter	performs some stack setup upon entering a function to include making space on the stack for local variables (see below)
leave	performs some cleanup before returning from a function (see below)

call and ret

Examples of calling a function

```
call DEST
call puts( "This is an example of calling a function" );
call [rbx]      # So is this
call 0x557766442280 # and this
```

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*.

When a function is complete, the `ret` instruction is used to return control to the point in the program that called the function. The address of the instruction after the call is on the stack. The `ret` instruction pops that address off the stack and into `rip`.

`ret` can optionally take a 16 bit numeric value. This form of the `ret` instruction pops *SIZE* number of **words** from the stack and returns from a function call. It does so by popping the current value from the stack and jumping to that address. The *SIZE* argument is optional.

The execution stack provides a trace of previous saved `rip` locations forming a backtrace of sorts.

Table 2. Call Stack on x86_64 UNIX process

0xEFF8	Calling function local variables	
0xEFF0	Address of instruction after the call	
0xEFEB	Contents of base pointer	
0xEF70	Local variables for the called function	

Stack Space

There are a limited number of registers. Not everything will fit in the available registers. Functions can use memory at the top of the stack to store (sometimes called **spilling**) data.

The stack is a first in last out data structure. The `rsp` register holds the address of the topmost writable location on the stack. 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. The base of the stack is the highest address.

Two special-purpose instructions manipulate the stack—`push` and `pop`. Both of these instructions implicitly adjust the stack pointer register.

push and pop

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

```
push    rbp
```

The `pop` instruction extracts the value at the head of the stack and moves that value into the specified register.

```
pop    rbp
```

Base Pointer, Stack Pointer and Local Variables

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_for_local_variables
...
add    esp, .stack_space_for_local_variables
pop    ebp
ret
```

This structure allows for the entire call stack to be discoverable and tracable with no additional information. Notice the use of `sub esp` to create space on the stack for local variables. Notice the use of `add esp` to clean up space that local variables used. Compare this with the description of `enter` and `leave` below.

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.

On x86_64 systems, this is unnecessary. Debugging information is stored in symbol tables via `.stab*` instructions, and is used to reconstruct the call stack in a debugger.

enter and leave

The `enter` instruction performs setup at the start of a function.

- pushes `rbp`
- saves `rsp` in `rbp`
- adds the requested value to the stack pointer in order to make room for local variables
- saves additional stack frames as requested

The `leave` instruction performs cleanup at the end of a function.

- restores the original stack pointer from `rbp`
- pops the original `rbp` from the stack

Compare the code below with the preamble and epilogue code above.

```
enter  .stack_space_for_local_variables, 0
...
```

`leave`
`ret`

NOTE

Because the stack is writable, overrunning the bounds of local function variables would overwrite the return address. Executing a `ret` under those circumstances would transfer control to an arbitrary location with any set of arguments desired.

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 do not 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 may not be used by the called function. However, when the function returns, these **Callee-Saved Registers** must hold the same value as when the function was called. A function may save the current values, use the registers, and then restore them.

Windows and UNIX 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. Which of the other registers are available for the callee depends on the platform.

The safest choice for a programmer is to store variables on the stack or to use `malloc` to allocate memory.

x86_64 ABI

On x86_64 architecture, the stack must always be aligned to 16 bytes before calling a function. The `call` instruction will push the return location onto the stack, bringing it to 8 modulo 16 on function entry.

Microsoft x86_64 Application Binary Interface

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

Position	Integer Parameter	Floating Point Parameter
1	rcx	xmm0
2	rdx	xmm1
3	r8	xmm2
4	r9	xmm3

Additional arguments are pushed onto 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 the function spills the `rcx`, `rdx`, `r8`, and `r9` registers here.

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

System V AMD64 Application Binary Interface

On non-Windows platforms, the number of register-passed arguments is much greater. The class of the argument and the position of the argument combine to determine the register that will be used to pass that argument.

Class of Argument	Types
INTEGER	char, short, int, long, long long, and pointers <code>_int128</code> require two INTEGER registers
SSE	float, double, <code>_Decimal32</code> , <code>_Decimal64</code> and <code>_m64</code>
SSEUP	<code>_float128</code> , <code>_Decimal128</code> , and <code>_m128</code>
X87	the 64 bit mantissa of type long double
X87UP	the 16 bit exponent (plus 6 bytes of padding) of type long double
MEMORY	anything else

Table 3. Registers specified to pass each type of parameter

Parameter Order	INTEGER	SSE	SSEUP	X87	X87UP	MEMORY
1	rdi	xmm0	next available vector register	stack	stack	stack
2	rsi	xmm1	next available vector register	stack	stack	stack
3	rdx	xmm2	next available vector register	stack	stack	stack
4	rcx	xmm3	next available vector register	stack	stack	stack
5	r8	xmm4	next available vector register	stack	stack	stack
6	r9	xmm5	next available vector register	stack	stack	stack
7		xmm6	next available vector register	stack	stack	stack
8		xmm7	next available vector register	stack	stack	stack

Vector registers could be the xmm registers, the ymm registers or the zmm registers. If there are more arguments in a given class than there are available registers, those excess arguments are passed on the stack.

See System V Application Binary Interface, AMD64 Architecture Processor Suppliment, Version 1.0, January 28, 2018

NOTE

As complicated as the charts above might seem, the vast majority of times calling

functions pass zero to four parameters and those parameters show up in some combination of rdi, rsi, rdx, rcx, xmm0, xmm1, xmm2, xmm3. Unless a function requires a large number of floating point parameters, programmers will not have to concern themselves with the full complexity of the System V Application Binary Interface.

Register Usage by Subroutines

It is never safe to assume that any register other than the stack pointer and the instruction pointer will be preserved across a call to another function.

The C runtime library and Linux kernel calls will preserve rbx, r12, r13, r14 and r15. Your code should preserve any values in these registers from the function that called you.

The only safe places for values are the stack and from allocated memory (i.e. `malloc()`)

Return Values

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

On x64 architecture, floating-point return values are placed in `xmm0`.

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 returned in the accumulator. A structure up to double that size will be returned in the accumulator and data registers on return.

Large literal structures as return values are more complex. A structure of larger than the RDX:RAX register pair will require a pointer to a memory location where the return value may be safely written. The calling function is responsible to set aside sufficient space to hold the structure and pass a pointer to that space.

```
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;
}
```

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

Even though it is not required, an OS may still use `rbp` as a base pointer for the current function.

Red Zone

The System V AMD64 ABI guarantees that the next 128 bytes on the stack will not be overwritten by asynchronous activity. This means that the space can be used for spilling registers, temporary variables, and the like. This **Red Zone** is only guaranteed in the System V ABI. MS-Windows will actively overwrite memory above the top of the stack.

The Red Zone is primarily of interest to "leaf functions"—functions that do not call other functions—and provides them an optimization where they can directly address data in the red zone without having to adjust RSP.

Exercises

Exercise 1

Use a Debugger to Verify the Registers Used to Pass Parameters

Purpose:

To explore the interface between C and assembly language code. To explore the way parameters are passed to assembly language code.

Instructions:

Create a directory to hold your work. The name of the directory is not critical. You may use any directory name you wish. These instructions will name the directory parameterLab.

Copy the SeaShell.c file and the makefile from the developmentTools folder. There will be instructions for modifying these files shortly.

Create a file called function.s Enter the following directives and code:

function.s

```
.intel_syntax noprefix  
  
.globl parameterTest  
.globl registerTest  
  
parameterTest:  
    ret  
  
registerTest:  
    mov rdi, 6633  
    call srand@PLT  
    xor rax, rax  
    push rax  
    call rand@PLT  
    pop rax  
    ret
```

NOTE

This very minimalist assembly language code will not change at all during this exercise. On the other hand, changes made to the extern declarations in the C source file will change the registers used to pass parameters.

Modify the makefile as shown below:

makefile

```
ASMFFileNameRoot = function  
OUTFILE = TestOfFunctionCall
```

DriverFileNameRoot = SeaShell

```
ASMFILE = $(ASMFfileNameRoot).s
OBJFileASM = $(ASMFfileNameRoot).o
CFILE = $(DriverFileNameRoot).c
OBJFileDriver = $(DriverFileNameRoot).o
CC = gcc
ASFLAGS += -m64 -masm=intel
CFLAGS += -fno-asynchronous-unwind-tables -std=c11
DIAGNOSTIC += -Wall -Wextra -Wpedantic -Waggregate-return -Wwrite-strings -Wfloat-equal -Wvla -g3
OPTIMIZATION += -O1

$(OUTFILE) : $(OBJFileDriver) $(OBJFileASM)
    $(CC) $(DIAGNOSTIC) $(OPTIMIZATION) -o $(OUTFILE) $(OBJFileASM) $(OBJFileDriver)

$(OBJFileDriver) : $(CFILE)
    $(CC) $(CFLAGS) $(ASFLAGS) $(OPTIMIZATION) $(DIAGNOSTIC) -c -o $(OBJFileDriver) $(CFILE)

$(OBJFileASM) : $(ASMFfileNameRoot)
    $(CC) $(ASFLAGS) $(OPTIMIZATION) $(DIAGNOSTIC) -c -o $(OBJFileASM) $(ASMFfileNameRoot)
```

A modified version of the above makefile will be useful in many exercises. In many cases, the only lines that will need changing will be ASMFfileNameRoot and OUTFILE.

Modify the include files section of SeaShell.c as shown below.

part of SeaShell.c

```
1 /* SeaShell.c */
2
3 #include <time.h>
4 #include <stdio.h>
5
6 extern double parameterTest( int, double );
7 extern void registerTest( void );
```

In this exercise, you will change the declaration for the parameterTest() function several times.

Modify the code for the testHarness() method as shown below.

The testHarness() function of SeaShell.c

```
1 int testHarness( void ) {
2     int resultCode = 0;
3
4     printf( "parameterTest() returned %f\n\n", parameterTest( 1, 9.876 ) );
```

```
5     registerTest();
6
7
8     return resultCode;
9 } /* end testHarness()      -      - */
```

The `testHarness()` code will display whatever `parameterTest()` returns. It will format that return value as a floating point number because that is what the `extern` declaration told `gcc` to do.

The `main()` function of `SeaShell.c` does nothing but call `testHarness()`.

The main() function of SeaShell.c

```
1 int main( void ) {
2     int exitCode = -99;
3     reportTime(); /* Write something to the screen so that
4     everybody knows that we made it into main() */
5
6     exitCode = testHarness();
7
8     reportTime(); /* Write something to the screen so that
9     everybody knows that we made it
10    out of testHarness() and back into main() */
11
12    return exitCode;
12 } /* end main()      -      - */
```

Make the executable. There should be no errors and no warnings.

Run the executable. The output should be something similar to this:

```
` ./TestOfFunctionCall Thu May 21 17:11:04 2020
```

```
parameterTest() returned 9.876000
```

```
Thu May 21 17:11:04 2020`
```

Can you explain what `parameterTest()` returned? Let us see what happens if we change the declaration for `parameterTest()`.

Make two trivial changes. Change the declaration of `parameterTest()` to

```
extern int parameterTest( int, double );
```

Change the `printf()` format symbol to `%d`.

the testHarness() function of SeaShell.c

```
1 extern int parameterTest( int, double );
2
3 int testHarness( void ) {
4     int resultCode = 0;
```

```
5     printf( "parameterTest() returned %d\n\n", parameterTest( 1, 9.876 ) );
6
7     registerTest();
8
9
10    return resultCode;
11 } /* end testHarness()      -      - */
```

Make the executable. Run the executable.

Your output should look something like:

```
`./TestOfFunctionCall Thu May 21 17:20:51 2020
```

```
parameterTest() returned 0
```

```
Thu May 21 17:20:51 2020`
```

Can you explain the output?

Once again, make two minor changes to the declaration of parameterTest().

```
extern double parameterTest( double, int );
```

Change the return declaration of parameterTest() to double. Reverse the order of the parameters.

Make two trivial changes to printf(). Change the format string back to %f and reverse the order of the two parameters.

```
printf( "parameterTest() returned %f\n\n", parameterTest( 9.876, 1 ) );
```

Build and run the executable. Try to explain the output.

If you have trouble explaining the output, review the lecture pages for System V AMD64 ABI. In particular, review the discussion of Return Types.

Run the executable in a debugger.

```
gdb TestFunctionCall
```

Set breakpoints at testHarness and parameterTest. Run to the first breakpoint.

```

<testHarness>    endbr64
<testHarness+4>  sub    rsp,0x8
<testHarness+8>  mov    edi,0x1
<testHarness+13> movsd  xmm0,WORD PTR [rip+0xe99]      # 0x555555556028
<testHarness+21>  call   0x55555555179 <parameterTest>
<testHarness+26>  lea    rsi,[rip+0xe69]      # 0x555555556004
<testHarness+33>  mov    edi,0x1
<testHarness+38>  mov    eax,0x1
<testHarness+43>  call   0x555555555070 <__printf_chk@plt>
<testHarness+48>  mov    eax,0x0
<testHarness+53>  add    rsp,0x8
<testHarness+57>  ret

```

Look at the assembly language that the compiler generated for testHarness. Examine the two registers that get values just before the call to parameterTest(). Compare those registers to the lecture material for System V AMD64 ABI.

Type c in order to advance to the next breakpoint at the start of parameterTest(). Examine the contents of the `xmm0` register.

```
p /f $xmm0
```

You will see the floating point parameter you programmed in `xmm0` as a v2_double.

Given that `parameterTask()` does nothing except return, the `xmm0` register has the same value when `printf()` is called that it had when `parameterTask()` was called. That is how `printf()` gets the value it displays.

Modify the declaration for `parameterTest()` again. This time, give it nine integer parameters valued from 1 to 9. Keep the floating point parameter unchanged.

changes to the testHarness() function of SeaShell.c

```

1 extern int parameterTest( int, int, int, int, int, int, int, int, double );
2
3 int testHarness( void ) {
4     int resultCode = 0;
5
6     printf( "parameterTest() returned %d\n\n",
7            parameterTest( 1, 2, 3, 4, 5, 6, 7, 8, 9, 9.876 ) );
8
9     registerTest();
10
11    return resultCode;
12 } /* end testHarness() - - */

```

Build and run the executable. There should be no errors or warnings and the output should be the same as before.

Run the executable in a debugger.

```
gdb TestFunctionCall
```

Set breakpoints at `testHarness` and `parameterTest`. Run to the `parameterTest` breakpoint.

The only code in `parameterTest()` is a rather boring `ret` instruction. Six of the registers, on the other hand, will have the parameters that were passed by the calling function. The other three parameters were passed on the stack. To see those parameters, type:

```
x  $rsp+8  
x  $rsp+16  
x  $rsp+24
```

To see the floating point parameter:

```
p $xmm0
```

Look for the `v2_double`

Set a breakpoint on `registerTest()`.

```
break registerTest
```

Run to the breakpoint.

Source code to registerTest()

```
registerTest:  
    mov rdi, 666  
    call srand@PLT  
    xor rax, rax  
    push rax  
    call rand@PLT  
    pop rax  
    ret
```

Step across the call to `srand`.

```
n 2
```

You should be stopped on the following line:

```
xor rax, rax
```

Every register that is highlighted had its value changed in `srand()`.

Step across the call to `rand`.

```
n 3
```

You should be stopped on the following line:

```
pop rax
```

If your code had stored any values in the highlighted registers, those values would have been

destroyed.

Quit the debugger.

Summary:

Never assume that a function you call will respect the registers you use. If you want a value to survive a call to another function, store it on the stack or store it in allocated memory.

On Linux systems, the System V AMD64 ABI specification dedicates CPU registers to pass parameters to functions.

Windows x86_64 systems dedicate four CPU registers for integer parameters and four for floating point numbers. No more than four parameters can be passed in CPU registers. The four parameters can be any combination of integers and floating point numbers as long as the total of both does not exceed four.

In both systems, when the number of parameters exceed the CPU registers dedicated to passing parameters, the excess parameters are passed on the stack.

Chapter 7. Assignment

As a reminder, in order to be able to program in any procedural language, a programmer must be able to:

- use the tools that create and debug executables
- store and retrieve values from variables
- perform simple mathematical operations
- call subroutines and return from them
- make decisions
- perform repetitive actions (loops)

This unit is about assembly language instructions that perform assignments.

Assembly Language Instructions

One of the most commonly used assembly language instructions is `mov`—an instruction that performs assignment. However, `mov` is not the only instruction that performs assignment.

Table 4. Most common instructions that assign a value to a destination

<code>mov</code>	Copy a value to a destination from a source
<code>movzx</code>	Copy a value to a destination from a smaller source. Set all the upper bits to zero
<code>movsx</code>	Copy a value to a destination from a smaller source. Set the upper bits to the value of the sign bit in the smaller source
<code>cmov</code>	Copy a value to a destination from a source if and only if some condition is met. Comes in many flavors
<code>lea</code>	Load effective address. Calculate an value and move it to a register
<code>xchg</code>	Exchange values between two registers or between a register and memory

In addition to the assignment instructions above, there are instructions that assign values to the various floating point registers on the CPU.

While not really an assignment, there are three instructions that each double the width of a value in the accumulator.

`cbw` converts a byte in the `al` register to a word in the `ax` register.

`cwde` converts a word in the `ax` register to a double word in the `eax` register.

`cdqe` converts a double word in `eax` to a quad word in `rax`.

In all cases, the sign bit is extended in the destination register. In assembly language, "variables" are either CPU registers or memory locations of some kind.

The many forms of mov

The **mov** instruction assigns (copies) a value to a destination from a source. The destination may be a register or a location in memory. The source may be a register, a location in memory, or a constant value. However, the source and the destination cannot both be memory locations.

Table 5. Examples of assignment using mov

Example	C language equivalent	Assignment type	Comment
mov rax, 0x100	a = 256	Immediate assignment to a register	The 64 bit destination register holds the value 256.
mov rax, rcx	a = c	register to register assignment	64 bits of data are copied.
mov rax, [rbx]	a = *b	memory to register assignment	64 bits of data are copied from memory to the 64 bit destination register. Note the square brackets around the source register.
mov [rbx], eax	*b = a	register to memory assignment	64 bits of data are copied from the source register to memory. Note the brackets around the destination register.
mov QWORD PTR [rbx], 0x100	*b = 256	Immediate assignment to memory location	The assembler has no way to know how the data area at [rbx] might be. Data size must be explicitly specified.

Immediate assignments to memory locations are inherently ambiguous. When a memory location is the destination of an immediate move, the size of the destination must be specified. The various PTR qualifiers tell the assembler the size of the memory location.

Descriptor	Number of Bytes at Memory Location
BYTE PTR	1
WORD PTR	2
DWORD PTR	4
QWORD PTR	8
TBYTE PTR	10

When the terms above were first specified, a typical CPU word was 16 bits. Therefore, WORD PTR refers to a 2-byte memory location. DWORD PTR (double word) means a 4 byte memory location. QWORD

PTR (quadruple word) means an 8 byte memory location. TBYTE PTR (ten byte pointer) means a memory location that is 10 bytes long. Very high precision floating point numbers used by Intel's 80 bit (ten byte) floating point processor occupy these locations.

Memory locations may be addressed either directly or indirectly. It is rare for code outside of operating system code to directly address specific memory locations.

```
mov rax, [0x1000] # a = *((void*)0x1000)
```

This loads the 64 bits from fixed memory address 0x1000 into rax.

There are problems with application code that relies on data being stored at specific locations that are known at compile time:

- Operating systems load programs into locations that are not known until execution time.
- Operating systems often move applications from one memory location to another.
- Fixed memory locations do not scale well.

Indirect Addressing

Intel provides several ways to indirectly address memory:

- Treat a value in a CPU register as an address
- Scale Index Base (SIB) addressing
- RIP relative addressing

Indirect register addressing interprets the value in a general purpose register as an address.

Treat an address in a register as a pointer

```
1 mov eax, [rbx] # a = *b
```

Notice the square brackets around the name of the register.

The formula for SIB addressing is:

segment:[base register + (scale * index register) + offset]

Base and index must be general purpose registers. Registers like the stack pointer or the instruction pointer may not be used.

Scale must one of the following constants: 1, 2, 4, 8. Internally, the CPU will shift the value in the index register either zero, one, two or three bits to the left. Hence the reason for those specific multipliers.

Offset is a 32-bit signed constant.

Segment is a value in a segment register (ds, es, ss, fs, gs). In 64 bit Intel assembly language programming, the segment register is almost always zero. Segment registers were very important in 16 and 32 bit programming.

Contrast the explanation of SIB addressing with RIP relative addressing below. Instead of calculating a base address from one register plus a second register and adding a constant to that, RIP relative addressing adds a constant to the current value in the instruction pointer.

Two examples of RIP addressing

```
1 lea rdx, Ethnicity[rip]
2      # several lines of code not shown
3 lea rax, [rip + EthnicityError]
4 ret
5
6 EthnicityError:
7 .string "Bad data!"
8
9 Ethnicity:
10 .quad FloridaEthnicity00
11 .quad FloridaEthnicity01
```

Programmers have no idea at compile time where the operating system loader will ultimately choose to place data. If a programmer specifies an address as an offset from the instruction pointer, the loader will be able to figure out where the address is.

RIP addressing is **addressing relative to the instruction pointer**. In the example above, `Ethnicity[rip]` and `[rip + EthnicityError]` both mean

- calculate the distance from some label to program counter.
- load that effective address into a register, in this case `rdx`

In contrast to RIP relative addressing, sometimes it is convenient to build a table of values in memory and pick the correct entry from that table at runtime. SIB addressing is very convenient for this.

As an example, Florida voter registration records represent a voter's ethnicity as a digit from zero through nine. Not all digits are used. When processing Florida voter records, it is convenient to build a table of strings describing ethnicities and use the number in the voter's record as an index into that table.

Example of SIB addressing

```

1      # the rdi register holds an ethnicity number from a voter record
2      # rdx already holds the address Ethnicity
3      # each address in the Ethnicity data table is 8 bytes long
4      # the addresses in the table
5      #   are the addresses of appropriate text
6      #   to describe ethnicity
7      lea rdx, [rdx + 8*rdi]
8      mov rax, [rdx]
9      ret
10
11 FloridaEthnicity00:    .string "Bad data!"
12 FloridaEthnicity01:    .string "American Indian or Alaskan Native"
13 FloridaEthnicity02:    .string "Asian or Pacific Islander"
14 FloridaEthnicity03:    .string "Black, not Hispanic"
15 FloridaEthnicity04:    .string "Hispanic"
16 FloridaEthnicity05:    .string "White, not Hispanic"
17 FloridaEthnicity06:    .string "Other ethnicity"
18 FloridaEthnicity07:    .string "Multi-racial"
19 FloridaEthnicity09:    .string "Unknown ethnicity"
20
21 Ethnicity:
22 .quad FloridaEthnicity00
23 .quad FloridaEthnicity01
24 .quad FloridaEthnicity02
25 .quad FloridaEthnicity03
26 .quad FloridaEthnicity04
27 .quad FloridaEthnicity05

```

```
28 .quad FloridaEthnicity06
29 .quad FloridaEthnicity07
30 .quad FloridaEthnicity00
31 .quad FloridaEthnicity09
```

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. There are complex forms of `mov` that can do this in one instruction.

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, shorts, longs, and long longs.

All of these parts may be combined in various ways.

Intel Indirect Addressing Examples

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

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 Examples

```
movq 0xABBA(%rcx, 8), %rax
movq 0xABBA(%rbx), %rax
movq 0xABBA(%rbx, %rcx), %rax
movq 0xABBA(%rbx, %rcx, 2), %rax
```

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 was 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 so highly optimized that the SIB calculation can be used to perform simple arithmetic instructions very quickly. Very often, compilers will optimize simple arithmetic expressions into lea instructions.

Examples of using lea to perform simple arithmetic

```
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 actual address.

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.

Sign Extension and Zero Extension

Normally, when a CPU copies a value from a small register to a larger register, it just copies bits. That can present multiple problem if the small register contains a signed value.

```
Register group: general
rax          0x555555555545ff
rcx          0x55555555554610
rsi          0x7fffffffdf08
rbp          0x55555555554610
r8           0x7fffff7dd0d80
r10          0x2          2
r12          0x555555555544f0
r14          0x0          0
rip          0x555555555545fc
cs           0x33         51
ds           0x0          0

B+ 5      mov al, -1
> 6      cbw
7      movsx rcx, ax
8      ret
```

The code in this debugger screenshot moves a negative number to an 8 bit register—al. The al register is the lowest eight bits of the `rax` register.

- The lower eight bits of `rax` became 0xFF as expected.
- All other bits retained the values that they had. In this example, the high 56 bits of `rax` now contain garbage.
- Under certain circumstances, comparisons might not work as expected.
- Under certain circumstances, multiplication and division might not work as expected.

The **convert** instructions address this problem.

cbw, cwd, cwde, cdq, cdqe, cqo

These functions only operate on values in the accumulator.

- cbw, cwd and cdqe extend the value to higher bits in the accumulator.
- cwd, cdq and cqo extend the accumulator values into the data register.

Register group: general		
rax	0x55555555ffff	
rcx	0x555555554610	
rsi	0x7fffffffdf08	
rbp	0x555555554610	
r8	0x7ffff7dd0d80	
r10	0x2	2
r12	0x5555555544f0	
r14	0x0	0
rip	0x5555555545fe	
cs	0x33	51
ds	0x0	0

B+	5	mov al, -1
	6	cbw
>	7	movsx rcx, ax
	8	ret

movzx, movsx

As was shown earlier, when a signed number needs to be copied from a smaller register to a larger register, `mov` has issues with the sign bit. (See earlier screenshots.) The various convert instructions solve the problem if the signed number is already in the accumulator. The `movsx` instruction solves the problem when a register other than the accumulator is the destination or when the source is in memory. The destination for `movsx` must be a register. Any register or memory location may be the source.

`movsx` propagates the sign bit to the larger register. `movzx` fills the higher order bits with zeros.

Register group: general			
rax	0x55555555ffff	93824992280575	
rcx	0xfffffffffffffff	-1	
rsi	0x7fffffffffdf08	140737488346888	
rbp	0x555555554610	0x555555554610	
r8	0x7fffff7dd0d80	140737351847296	
r10	0x2 2		
r12	0x5555555544f0	93824992232688	
r14	0x0 0		
rip	0x555555554602	0x555555554602	
cs	0x33 51		
ds	0x0 0		

B+	5	mov al, -1
	6	cbw
	7	movsx rcx, ax
>	8	ret

Compare the `rax` and `rcx` registers in the last three screenshots.

Additional examples using movsx

```
movzx DST, SRC

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

NOTE

`mov` behaves one way when assigning values to 8 or 16 bit registers and another way when assigning values to 32 bit registers.

When assigning a value to an 8 or a 16 bit register, `mov` leaves any higher bits unchanged. This is for historical reasons so that code written in the early 1980s would still compile and execute the way the authors expected.

When assigning a value to a 32 bit register, the upper 32 bits of the 64 bit registers will be set to 0. Therefore, There is no need for a special version of `mov`. The source for these special `mov` instructions will only be one or two bytes wide. See the examples below.

```
# THESE TWO LINES DO NOT ASSEMBLE
# movzx rax, ecx
# movsx rax, ecx

# Correct -- no need for movsx or movzx
mov eax, ecx # automatically clears the high 32 bits of rax
```

```
mov ecx, 57    # automatically clears the high 32 bits of rcx
```

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

Create a Random Number Generator

Purpose:

Practice assigning values to registers. Practice calling functions. Practice returning values from functions. Demonstrate the design principle of encapsulation. Practice using an environment variable to control part of the build process.

Background:

You will add features to this package of functions in future exercises as you learn the necessary concepts and assembly language instructions.

Instructions:

Create a directory to for this exercise. Call the directory random.

Copy the SeaShell.c file and the makefile from the previous exercise to the directory you just created.

If you followed the instructions from the earlier exercises, your makefile will require only three very trivial modifications.

- Change the value of ASMFileNameRoot to random
- Change the value of OUTFILE to TestRandom
- Remove the -g3 debugging option from the DIAGNOSTIC variable.

In the future, when we want to debug, we will export an environmental variable on the command line.

NOTE Any of the makefile variables that use += rather than = can be modified without editing the makefile. This is very useful for compiling with or without debugging information.

The complete makefile after modifications

```
ASMFileNameRoot = random
OUTFILE = TestRandom
DriverFileNameRoot = SeaShell

ASMFILe = $(ASMFileNameRoot).s
OBJFileASM = $(ASMFileNameRoot).o
CFILE = $(DriverFileNameRoot).c
OBJFileDriver = $(DriverFileNameRoot).o
CC = gcc
ASFLAGS += -m64 -masm=intel
```

```

CFLAGS += -fno-asynchronous-unwind-tables -std=c11
DIAGNOSTIC += -Wall -Wextra -Wpedantic -Waggregate-return -Wwrite-strings -Wfloat-
equal -Wvla
OPTIMIZATION += -O1

$(OUTFILE) : $(OBJFileDriver) $(OBJFileASM)
    $(CC) $(DIAGNOSTIC) $(OPTIMIZATION) -o $(OUTFILE) $(OBJFileASM) $(
OBJFileDriver)

$(OBJFileDriver) : $(CFILE)
    $(CC) $(CFLAGS) $(ASFLAGS) $(OPTIMIZATION) $(DIAGNOSTIC) -c -o $(
OBJFileDriver) $(CFILE)

$(OBJFileASM) : $(ASMFILE)
    $(CC) $(ASFLAGS) $(OPTIMIZATION) $(DIAGNOSTIC) -c -o $(OBJFileASM) $(
ASMFILE)

```

The SeaShell.c test driver requires minor modification to the `testHarness()` function.

The modified `testHarness()` function

```

1 extern void seedGenerator( void );
2 extern unsigned getRandomNumber( unsigned );
3
4 int testHarness( void ) {
5     int resultCode = 0;
6
7     seedGenerator();
8     printf( "Today's random number is %u\n", getRandomNumber( 100 ) );
9
10    return resultCode;
11 } /* end testHarness() */
```

NOTE For the time being, the `getRandomNumber()` function will ignore any parameter it receives. A future version of `getRandomNumber()` will clip its output to the range of 1 to whatever parameter you pass.

Create a file called `random.s`. This file will hold two closely related functions:

- `getRandomNumber` will return a random number between 0 and 3.2 billion.
- `seedGenerator` will provide a value that will serve as a basis for generating a series of random numbers.

NOTE: It is this reliance for entropy on a predictable resource like the system clock that makes this random number generator cryptographically insecure. A future exercise will modify this code to use hardware as a source of entropy.

The assembly language code for `seedGenerator` will call two C runtime library functions—`time()`

and `srand()`. `time()` accepts zero as a parameter and returns a long integer. That long integer becomes the input parameter for `srand()`. This function returns nothing. All it does is seed the random number generator of the C runtime library.

All `getRandomNumber` does is call a C runtime library function that returns a random number.

The source code for random.s

```
1 .intel_syntax noprefix
2
3 .globl seedGenerator
4 .globl getRandomNumber
5
6 seedGenerator:
7     mov    rdi, 0
8     call   time@PLT
9     mov    rdi, rax
10    call   srand@PLT
11    ret
12
13 getRandomNumber:
14    call   rand@PLT
15    ret
```

Build the executable. Run the executable. The output from two tests of a working executable will look something like:

```
./TestRandom

Sat May 23 00:25:42 2020
Today's random number is 127419968
Sat May 23 00:25:42 2020

./TestRandom

Sat May 23 00:27:41 2020
Today's random number is 1513345461
Sat May 23 00:27:41 2020
```

Each time the executable runs, it will produce a different number.

If you need to debug your work, type the following command lines:

```
export DIAGNOSTIC=-g3 make -B gdb ./TestRandom
```

To rebuild your work without debugging information, type the following command lines

```
export DIAGNOSTIC= make -B ./TestRandom
```

Summary:

Random number generators are very useful test tools. While this exercise will not produce a cryptographically secure pseudorandom number generator, it will produce a generator good enough to be used to test the code from these exercises.

Exercise 2

Convert a Number from 1 to 12 into the Name of a Month

Purpose: Practice creating a lookup table. Practice using the load effective address instruction.

Instructions:

Create a directory for this exercise. Call the directory `month`.

Copy the `makefile` and the `SeaShell.c` test driver file from the `random` directory to `month`.

Copy the `random.o` file from the `random` directory to `month`. Do not copy the source file. Copy the object file. For now, `random.o` will become a library artifact that we will use. A future exercise will add features to the random code, but for now, `random.o` has all the functionality we need.

Modify the `makefile` you just copied. As before, the modifications are trivial. We will show you the lines that need changing. Then we will show you the entire `makefile`.

makefile lines that require modification

```
ASMFileNameRoot = month
OUTFILE = TestMonth
LibraryCode=random.o

DriverFileNameRoot = SeaShell

ASMFILER = $(ASMFILER).s
```

Only the top three lines require modification. Notice that well designed makefiles use variables so that things that might require change appear near the top of the file.

The final makefile entry that needs to be changed

```
$(OUTFILE) : $(OBJFileDriver) $(OBJFileASM) $(LibraryCode)
    $(CC) $(DIAGNOSTIC) $(OPTIMIZATION) -o $(OUTFILE) $(OBJFileASM) \
    $(OBJFileDriver) $(LibraryCode)
```

Notice that the output executable file now depends on three things:

- two object files — `SeaShell.o` and `month.o`
- the `random.o` file that you copied from the previous exercise

Also, those same three object files combine to build the executable. Here is the complete `makefile` so that you can see everything in context.

The complete makefile

```
ASMFfileNameRoot = month
OUTFILE = TestMonth
LibraryCode= random.o

DriverFileNameRoot = SeaShell

ASMFILE = $(ASMFfileNameRoot).s
OBJfileASM = $(ASMFfileNameRoot).o
CFILE = $(DriverFileNameRoot).c
OBJfileDriver = $(DriverFileNameRoot).o

CC = gcc
ASFLAGS += -m64 -masm=intel
CFLAGS += -fno-asynchronous-unwind-tables -std=c11
DIAGNOSTIC += -Wall -Wextra -Wpedantic -Waggregate-return -Wwrite-strings -Wfloat-equal -Wvla
OPTIMIZATION += -O1

$(OUTFILE) : $(OBJfileDriver) $(OBJfileASM) $(LibraryCode)
    $(CC) $(DIAGNOSTIC) $(OPTIMIZATION) -o $(OUTFILE) $(OBJfileASM) \
    $(OBJfileDriver) $(LibraryCode)

$(OBJfileDriver) : $(CFILE)
    $(CC) $(CFLAGS) $(ASFLAGS) $(OPTIMIZATION) $(DIAGNOSTIC) -c -o $ \
    $(OBJfileDriver) $(CFILE)

$(OBJfileASM) : $(ASMFILE)
    $(CC) $(ASFLAGS) $(OPTIMIZATION) $(DIAGNOSTIC) -c -o $(OBJfileASM) \
    $(ASMFILE)
```

Modify the testHarness function to generate a random number from 1 to 12 inclusive. Do not worry about testing with bad data. You will add error checking in a future exercise.

The modified testHarness() function

```
1 extern void seedGenerator( void );
2 extern unsigned getRandomNumber( unsigned );
3 extern char * getMonthText( unsigned );
4
5 int testHarness( void ) {
6     int resultCode = 0;
7
8     seedGenerator();
9     unsigned randomMonthNumber = getRandomNumber( 100 ) % 12;
10    randomMonthNumber++;
11    printf( "\nToday's random number is %u\n", randomMonthNumber );
12    printf( "Today's random month is %s\n\n", getMonthText( randomMonthNumber ) );
13}
```

```
14     return resultCode;
15 } /* end testHarness() */ - - */
```

NOTE

At the moment, the value passed to `getRandomNumber()` is ignored. That will change in a future exercise.

The expression `getRandomNumber(100) % 12` will produce a number from 0 to 11 inclusive. The expression `randomMonthNumber++` makes the lowest possible number 1 and the highest possible number 12.

Create a file called `month.s`.

The only public label in this file will be `getMonthText`.

The first few lines of month.s

```
.intel_syntax noprefix  
.globl getMonthText
```

To create the text that will be returned, you will need some strings and some private labels that will not be visible outside of `month.s`. Each label is a pointer to the text for the name of a month. It is the address where the text begins. The `.string` directive is an assembler directive that creates null terminated text. The linker will pick an appropriate location in the executable to store that text.

These are the names of the months plus an error message that we will not use for now

```
January: .string "January"  
February: .string "February"  
March: .string "March"  
April: .string "April"  
May: .string "May"  
June: .string "June"  
July: .string "July"  
August: .string "August"  
September: .string "September"  
October: .string "October"  
November: .string "November"  
December: .string "December"  
  
BadParameter: .string "Only numbers from 1 to 12 are legal"
```

Once you have created the text that you will return to the caller, you can build the lookup table. Think of the lookup table as an array of pointers—addresses. Each address points to the address of a string. The lookup is an array of pointers to pointers just like `argv`.

```
NameOfMonthLoopupTable:  
.quad BadParameter
```

```
.quad January
.quad February
.quad March
.quad April
.quad May
.quad June
.quad July
.quad August
.quad September
.quad October
.quad November
.quad December
```

The `.quad` assembler directive tells the assembler to reserve four words—eight bytes of storage—for each address. Because of an interface design decision, any number less than one is illegal.

The code for the `getMonthText` function is short, simple and fast.

```
getMonthText:
# RDX gets the address where the lookup table begins
lea rdx, NameOfMonthLoopupTable[rip]
# RDX gets the address of the address
# where the text we want lives
# RDI is the month number that the calling function passed to us.
lea rdx, [rdx + 8*rdi]
# dereference that address
# return the address of the text we want
mov rax, [rdx]
ret
```

If you need to debug your work, type the following command lines:

```
export DIAGNOSTIC=-g3 make -B gdb ./TestMonth
```

To build your work without debugging information, type the following command lines

```
export DIAGNOSTIC= make -B ./TestMonth
```

Summary:

Lookup tables and their close cousins jump tables are very useful in assembly language programming. Lookup tables and jump tables are very similar to switch statements in C.

Exercise 3

Convert a Number from 1 to 7 into the Name of a Day

Purpose: Unstructured practice creating a lookup table. Unstructured practice using the load effective address instruction. In the previous exercise, we guided you through the process of creating a lookup table. In this exercise, you will demonstrate that you know how to implement the

pattern

Instructions:

Create a directory to hold your work. Copy the files you need from the previous exercise.

Modify the `testHarness` function to generate a random number from 1 to 7 inclusive. Do not worry about testing with bad data. You will add error checking in a future exercise.

Rename the assembly language file to `day.s`. Change the name of the assembly language function to `getDayText`. Modify the assembly language code as necessary. For this exercise, you may assume that you will only be passed good parameters. You will add error checking in a future exercise.

Modify the `makefile` to compile `day.s` Instead of `month.s`. Call your executable `TestDay`.

Build and run the executable.

If you need to debug your work, type the following command lines:

```
export DIAGNOSTIC=-g3 make -B gdb ./TestDay
```

To build your work without debugging information, type the following command lines

```
export DIAGNOSTIC= make -B ./TestDay
```

Summary:

Lookup tables and their close cousins jump tables are very useful in assembly language programming. Lookup tables and jump tables are very similar to switch statements in C. In this exercise, you did all the work yourself.

Chapter 8. Arithmetic

As a reminder, in order to be able to program in any procedural language, a programmer must be able to:

- use the tools that create and debug executables
- store and retrieve values from variables
- perform simple mathematical operations
- call subroutines and return from them
- make decisions
- perform repetitive actions (loops)

Assembly Language Instructions

This unit is about the 64 bit assembly language instructions that perform simple arithmetic and some bit manipulation on integers.

Table 6. The many forms of add, subtract, multiply and divide

Mnemonic	Name	What it does	Flags
inc	Increment	Add one to a general purpose register or the contents of a memory location	OF, SF, ZF, AF, PF
add	Add	Add the source to the destination. Destination can be memory or register. Source can be register, memory or immediate value.	OF, SF, ZF, AF, CF, PF
adc	Add with Carry	Add the source plus the carry flag to the destination. Destination can be memory or register. Source can be register, memory or immediate value.	OF, SF, ZF, AF, CF, PF
lea	Load Effective Address	This instruction was created to speed up the process of calculating the address of an element of an array or a member of a structure. Its ability to do some limited math very, very quickly has made it a popular choice for optimizers. The math has to fit the formula [register + register shifted left either zero or one or two or three bits + a constant]	none
dec	Decrement	Subtract one from a general purpose register or the contents of a memory location	OF, SF, ZF, AF, PF

Mnemonic	Name	What it does	Flags
sub	Subtract	Subtract the source from the destination.	OF, SF, ZF, AF, CF, PF
sbb	Subtract with borrow	Add the carry flag to the source. Subtract the result from the destination.	OF, SF, ZF, AF, CF, PF
mul	Multiply	Unsigned multiply of the accumulator by a value located in either a general purpose register or in memory. The result is placed in the accumulator. If the result does not fit in the accumulator, the high order bits are placed in the data register.	OF, CF
imul	Signed Multiply	signed multiply of the accumulator by a value located in either a general purpose register or in memory. The result is placed in the accumulator. If the result does not fit in the accumulator, the high order bits are placed in the data register.	OF, CF
div	Divide	Divide the unsigned value in the accumulator by a value located in either a general purpose register or in memory. The result is placed in the accumulator. The remainder is placed in the data register.	undefined

Mnemonic	Name	What it does	Flags
idiv	Signed Divide	Divide the signed value in the accumulator by a value located in either a general purpose register or in memory. The result is placed in the accumulator. The remainder is placed in the data register.	undefined

From the earliest days of microprocessors, the accumulator—AX, EAX, and RAX—has always been the preferred destination for any arithmetic operation. In the early days, the accumulator was the only possible destination for arithmetic. That is how it got its name. Even today, it should be used as a destination of arithmetic operations whenever possible. The op codes for arithmetic that uses the accumulator as a destination are both short and fast. Many compilers will generate seemingly unnecessary moves to the accumulator just to take advantage of its speed in simple arithmetic.

Second only to the accumulator is the data register—DX, EDX and RDX. Many operations may involve a combination of the accumulator and data registers. Three bits of the opcode indicate the destination register. Three bits indicate source register.

No Operation

NOP is an instruction that does absolutely nothing. NOP exists:

- to force memory alignment.
- for timing purposes in real time software.
- for other reasons.

For a much more detailed discussion, see [https://en.wikipedia.org/wiki/NOP_\(code\)](https://en.wikipedia.org/wiki/NOP_(code))

There are multiple NOP's of various lengths, from one to eight bytes to deal with opcode memory alignment issues. 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 NOP's. Which 'NOP' should be used is a function of the processor, the offset of the instruction, the current instruction pipeline, and any local jumps.

An .align directive in the source code asks the assembler to choose the NOP instruction. The assembler will generally emit the best NOP instruction. It may emit an actual NOP instruction, or it may emit instructions that amount to a NOP.

You will see compiler generated NOP instructions that take a register or SIB argument. Regardless of what you see, the processor still does nothing. It is generally better to let the assembler's alignment directive determine the most efficient NOP to emit.

Basic Arithmetic Operations

Addition

Addition is straightforward. Add a value to a given a destination. The destination can be a memory location or a register. 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, 0      # a += cf  
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 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

The decrement instruction 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. It is one of the few instructions that support up to three arguments.

The simplest form is a one-operand instruction. The operand must be a register or a memory location. The accumulator is multiplied by that factor. The 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.

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. The carry and overflow flags will be set if the product could not fit into the destination. The only way to get the full product is to use the one-argument version.

The `imul` instruction is signed multiplication. The `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

The `idiv` instruction takes 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. 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. The remainder is placed 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.

The idiv instruction is signed division. div is unsigned. The divisor's value is always sign-extended for signed division.

Bitwise Instructions

Mnemonic	Name	What it does	Flags
BSF	Bit Scan Forward	Find the least significant 1 bit in the source operand. Reports the index in destination. Destination content is undefined if source is zero. Sets the zero flag if source is zero.	ZF
BSR	Bit Scan Reverse	Find the most significant 1 bit in the source operand. Reports the index in destination. Sets the zero flag if source is zero	ZF
LZCNT	Count the Number of Leading Zeros	Find the index of the first 1 bit in source. Store that number in destination. Sets carry flag if most significant bit is set. Sets zero flag if most significant bit is set. Sets carry flag if source is too big for destination.	ZF, CF
BZHI	Zero High Bits Starting with Specified Bit Position	Change all the bits in destination register or memory location to zero starting with the bit specified in source.	CF
TZCNT	Count the Number of Trailing Zero Bits	Same as BSF except that destination is <i>not</i> undefined if source is zero. Destination holds operand size if source is zero.	ZF, CF
BT	Bit Test	Test one bit. Set or clear the carry flag	CF

Mnemonic	Name	What it does	Flags
BTC	Bit Test and Compliment	Test one bit in the source operand. Make the carry flag look like that bit. Make the corresponding bit in the destination have the opposite value	CF
BTR	Bit Test and Reset	Selects a bit. Gives the carry flag the same value as that bit. Sets the selected bit to zero.	CF
AND	Bitwise AND	If source and destination have a 1 bit in the same position, then destination will have that same bit set. Overflow and carry flags are always cleared by this instruction.	OF, SF, ZF, CF, PF
ANDN	Bitwise AND NOT	Inverts the value of the second operand then performs a bitwise AND with the third operand. AF and PF flags are undefined. Overflow and carry flags are always cleared.	OF, SF, ZF, CF
OR	Bitwise OR	If either source or destination have a 1 bit in the any position, then destination will have that same bit set. Overflow and carry flags are always cleared by this instruction.	OF, SF, ZF, CF, PF

Mnemonic	Name	What it does	Flags
XOR	Bitwise Exclusive OR	Destination will have a bit set if source and destination have different values for that position. Overflow and carry flags are always cleared by this instruction.	OF, SF, ZF, CF, PF
NOT	One's Complement Negation	Each 1 bit in destination becomes 0. Each 0 bit becomes 1.	none
NEG	Two's Complement Negation	Subtract the destination from zero. Store the result in destination	OF, SF, ZF, AF, CF, PF
SHL	Shift Left (one operand)	Shift the bits in destination left one bit. The highest bit gets shifted into the carry flag. Effectively multiplies by two. The AF flag is undefined.	OF, SF, ZF, CF, PF
SHL	Shift Left (two operands)	Shift the bits in destination left by source number of bits. Effectively multiplies by a power of two. The AF and OF flags are undefined.	SF, ZF, CF, PF
SHR	Shift Logical Right	Shift destination right by source number of bits. Fill the high bits with 0. Carry flag contains the last bit shifted out. Overflow flag is undefined except for one bit shifts. The AF flag is undefined.	SF, ZF, PF

Mnemonic	Name	What it does	Flags
SAR	Shift Arithmetic Right	Shift destination right by source number of bits. Fill the high bits with the value of the sign bit. Carry flag contains the last bit shifted out. Overflow flag is undefined except for one bit shifts. The AF flag is undefined.	SF, ZF, PF

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

and, or and 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 and 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 and shl

These two instructions are exactly equivalent.

```
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

Modify the random number generator to always produce positive numbers within a caller specified range.

Purpose:

Provide practice using simple arithmetic instructions to find the remainder from integer division.
Provide an opportunity to explore bit manipulation instructions.

Background:

Experienced C programmers know that the `rand()` function from the C runtime library produces positive and negative random numbers. Those experienced C programmers may have noticed that the assembly language code in this course stored the result from `rand()` in a 64 bit register without performing sign extension. This guaranteed that `rand()` would always produce a positive number.

There will be four short functions in `random.s` instead of just the two that were written in an earlier exercise. The two new functions use bit manipulation instructions to explore CPU architecture and learn something about the way `'rand()'` behaves.

Because you have now created several assembly language files, the exercise instructions for the assembly language code will be a lot less specific.

Instructions:

The files you need are in the `random` directory you created earlier.

A standard way to clamp output to a desired range is the following formula:

```
randomNumber % desiredLimit
```

This divides the number, ignores the result, and keeps the remainder. The remainder can never be equal or greater than `desiredLimit`. We will apply this approach to the `getRandomNumber()` function we created in an earlier exercise.

First, we will modify the test code in `SeaShell.c` so that it will test the code we are about to write.

Modified testHarness() code from SeaShell.c

```
/* SeaShell.c */

#include <time.h>
#include <stdio.h>

extern void seedGenerator( void );
extern unsigned getRandomNumber( unsigned );
extern unsigned getRandomNumberSize( void );
extern unsigned getArchitectureSize( void );
```

```

int testHarness( void ) {
    int resultCode = 0;
    unsigned range = 10000;

    seedGenerator();

    printf( "Architecture size: %u\n", getArchitectureSize() );

    for( int i = 12; i > 1; i -= 3 ) {
        printf( "%15d %15d %15d\n", getRandomNumber( range ), getRandomNumber( range ),
        getRandomNumber( range ) );
    }

    for( int i = 12; i > 1; i -= 3 ) {
        printf( "%5d bits %5d bits %5d bits\n", getRandomNumberSize(),
        getRandomNumberSize(), getRandomNumberSize() );
    }

    return resultCode;
} /* end testHarness() */ - - -

```

The choice of value for `range` is arbitrary. Choosing a power of 10 makes the math easier to predict.

Make the necessary additional global declarations in `random.s`

```

.globl seedGenerator
.globl getRandomNumber
.globl getRandomNumberSize
.globl getArchitectureSize

```

Modify the `getRandomNumber()` function.

- Find the register that holds the `range` parameter
- Preserve that register across the call to `rand()`
- Perform integer division on the random number in `RAX`
- Remember that the answer that the calling function needs—the remainder from the division—is in the `RDX` register

Create the code for a new function called `getRandomNumberSize()`. This function will report the number of bits used by the random number that `rand()` returns.

- Call `rand()`
- Use `lzcnt` or some other combination of instructions to find the highest one bit in the `RAX` register.
- Consider whether the result from the above bit count might be off by one

Create the code for a new function called `getArchitectureSize()`. This function should report how many bits wide the CPU registers are in the CPU in the lab computers. It should discover that answer using a method that will work whether the CPU is a 64 bit architecture, a 32 bit architecture, a 16 bit architecture or an 8 bit architecture.

- Consider that every architecture has a very lowest bit. Set that bit to 1 and clear all the other bits.
- Consider how the Rotate Right instruction works.
- Consider the documentation for an instruction that might count leading zeros. Pay particular attention to any special case situations when the highest bit in a register are 1 or all bits are zero.
- Consider any possible off by one errors

Summary:

This exercise provided practice with simple arithmetic and bit manipulation.

Exercise 2

Geometry

Purpose:

Provide practice with integer arithmetic.

Instructions:

Create a directory called `integerGeometry`. Copy the following files to that directory:

- The `random.o` file you created in the previous exercise.
- The `makefile` you used in the previous exercise
- The `SeaShell.c` test driver that you used in the previous exercise.

Modify the makefile as shown. The modifications are trivial.

Modified makefile

```
ASMFfileNameRoot = geometry
OUTFILE = TestGeometry
DriverfileNameRoot = SeaShell

ASMFILE = $(ASMFfileNameRoot).s
OBJfileASM = $(ASMFfileNameRoot).o
CFILE = $(DriverfileNameRoot).c
OBJfileDriver = $(DriverfileNameRoot).o
LibraryObjectFiles = random.o
CC = gcc
ASFLAGS += -m64 -masm=intel
CFLAGS += -fno-asynchronous-unwind-tables -std=c11
```

```

DIAGNOSTIC += -Wall -Wextra -Wpedantic -Waggregate-return -Wwrite-strings -Wfloat-
equal -Wvla
OPTIMIZATION += -O1

$(OUTFILE) : $(OBJFileDriver) $(OBJFileASM) $(LibraryObjectFiles)
    $(CC) $(DIAGNOSTIC) $(OPTIMIZATION) -o $(OUTFILE) $(OBJFileASM) \
    $(OBJFileDriver) $(LibraryObjectFiles)

$(OBJFileDriver) : $(CFILE)
    $(CC) $(CFLAGS) $(ASFLAGS) $(OPTIMIZATION) $(DIAGNOSTIC) -c -o \
    $(OBJFileDriver) $(CFILE)

$(OBJFileASM) : $(ASMFILE)
    $(CC) $(ASFLAGS) $(OPTIMIZATION) $(DIAGNOSTIC) -c -o $(OBJFileASM) \
    $(ASMFILE)

```

Modify SeaShell.c to test some functions that calculate:

- The perimeter of a rectangle: perimeter = (2 * height) + (2 * width)
- The area of a rectangle : area = height * width
- The perimeter or a triangle: perimeter = side01 + side02 + side03

External declarations for SeaShell.c

```

extern void seedGenerator( void );
extern unsigned getRandomNumber( unsigned );

extern unsigned perimeterRectangle( unsigned, unsigned );
extern unsigned areaRectangle( unsigned, unsigned );
extern unsigned perimeterTriangle( unsigned, unsigned, unsigned );

```

Modify the testHarness() function so that it will test the assembly language code you are about to write. The modified code will get three random between 1 and 100 inclusive. It will use those three numbers to test the assembly language code that you are about to write.

testHarness() code for SeaShell.c

```

int testHarness( void ) {
    int resultCode = 0;
    unsigned range = 100;
    unsigned answer = -99;

    seedGenerator();

    unsigned side01 = getRandomNumber( range );
    unsigned side02 = getRandomNumber( range );
    unsigned side03 = getRandomNumber( range );

    side01++; side02++; side03++;

```

```

    answer = perimeterRectangle( side01, side02 );
    printf( "Rectangle height: %u, width: %u, perimeter: %u\n", side01, side02,
answer );

    answer = areaRectangle( side01, side02 );
    printf( "Rectangle height: %u, width: %u, area: %u\n", side01, side02,
answer );

    answer = areaTriangle( side01, side02, side03 );
    printf( "Triangle side 1: %u, side 2: %u, side 3: %u, perimeter: %u\n",
side01, side02, side03, answer );

    return resultCode;
} /* end testHarness()      -      - */

```

Create a file called `geometry.s`

Write the necessary assembler declaration so that you can use Intel syntax.

Write the necessary assembler directives so that C code can call the three functions you are about to write.

Write the assembly language code for three functions that perform the required math.

Build and test your code.

Summary:

This exercise provided practice using the integer arithmetic instructions.

Chapter 9. Conditionals, Jumps and Loops

In order to be able to program in any procedural language, a programmer must be able to:

- use the tools that create and debug executables
- store and retrieve values from variables
- perform simple mathematical operations
- call subroutines and return from them
- make decisions
- perform repetitive actions (loops)

This lecture is about decisions and loops. By the end of this lecture, you will have covered the bare minimum knowledge needed to write procedural code in assembly language.

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

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

In assembly programming, conditions must be much simpler to reflect the available instructions and the available data types. More complex conditions 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.

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

Table 7. Condition Flags

Mnemonic	Name	Result that caused this flag to be set
CF	Carry Flag	operation caused unsigned overflow
ZF	Zero Flag	operation produced zero
SF	Sign Flag	operation produced a negative signed number
OF	Overflow Flag	operation produced a result too large for the register

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.

Example 1		Example 2		Example 3	
	0b0000		0b1001		0b1001
-	0b1000	-	0b1000	+	0b1000
	0b0001		0b0001		0b0001
Flags set	CF		OF		CF, OF

Example 1: The most-significant bit needed to execute a borrow, so the carry flag is set. The inputs have differing most significant bits (MSB), so the overflow flag is not set.

Example 2: The inputs have the same MSBs, and the output has a different MSB. The overflow flag is set. This calculation did not require the MSB to borrow, so the carry flag is not set.

Example 3: The inputs have the same MSBs. The output has a different MSB. The overflow flag is set. The addition requires a carry past the MSB of the result. The carry flag is set.

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 clear the CF and OF flags, since overflow will not 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 8. setcc Instructions

Instruction	Test	Meaning of Test
sete	ZF	logically equal or arithmetically zero
setne	\sim ZF	logically not equal or arithmetically not zero
sets	SF	negative
setns	\sim SF	not negative
setl	SF \wedge OF	(signed values) less
setle	(SF \wedge OF) or ZF	(signed values) less or the same
setg	\sim (SF \wedge OF) & \sim ZF	(signed values) greater
setge	\sim (SF \wedge OF)	(signed values) greater or the same
setb	CF	(unsigned values) below
setbe	CG or ZF	(unsigned values) below or the same
seta	\sim CF & \sim ZF	(unsigned values) above
setae	\sim CF	(unsigned values) above or the same

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

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

Listing 1. Condition Code Assignment in C

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

Listing 2. Condition Code Assignment in Assembler

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

ret

Jumps

A `jmp` instruction is just a `goto`. The most basic form of jump will jump within the same code segment. It is possible to jump to

- a specific address
- an address relative to the instruction pointer
- an address in a general purpose register

It is possible to jump as much as 2,147,483,648 bytes forward or back relative to the current position of the instruction pointer. Jumping up to 128 bytes forward or back relative to the instruction pointer is extremely efficient.

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])
```

Jump Tables

The last two jump examples bridge to the idea of jump tables.

In a high-level language, choosing the 1 of N possible code branches might typically be done using a switch statement. On the other hand, Python lacks a switch statement. Why mention Python, and why doesn't it have a switch statement? Because Python provides a high level representation of the same approach used in assembly language.

In Python, there is no need for a switch statement because functions are first class objects. They can be stored in an array or dictionary, and accessed by looking them up and calling them.

In Assembly, we can create a table (array) of addresses, and jump through the table, accomplishing the same goal.

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

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

The code checks to ensure that the index in EDI is 0-3. If not, the code branches to .L5, which presumably handles the error. Otherwise, the address is loaded from the table into ECX, and the code jumps to wherever ECX is pointing.

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.

A jump table is most efficient when it is compact and contiguous. 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.

Note that although the discussion has focused on the `jmp` instruction family, this form of addressing applies to `call` as well.

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.

The `jcc` instructions may only take a relative destination. They may not take registers or SIB addressing.

Table 9. Conditional Jump Instructions

Instruction	Test	Meaning of Test
<code>je</code>	<code>ZF</code>	logically equal or arithmetically zero
<code>jne</code>	<code>~ZF</code>	logically not equal or arithmetically not zero
<code>js</code>	<code>SF</code>	negative
<code>jns</code>	<code>~SF</code>	not negative
<code>jl</code>	<code>SF ^ OF</code>	(signed values) less
<code>jle</code>	<code>(SF ^ OF) or ZF</code>	(signed values) less or the same
<code>jg</code>	<code>~(SF ^ OF) & ~ZF</code>	(signed values) greater
<code>jge</code>	<code>~(SF ^ OF)</code>	(signed values) greater or the same
<code>jb</code>	<code>CF</code>	(unsigned values) below
<code>jbe</code>	<code>CG or ZF</code>	(unsigned values) below or the same
<code>ja</code>	<code>~CF & ~ZF</code>	(unsigned values) above
<code>jae</code>	<code>~CF</code>	(unsigned values) above or the same

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

```
absdiff:  
    mov eax, edi  
    cmp esi, eax    # in C, this would be an _if_  
    jle    .L6  
    sub esi, eax  
    mov eax, esi  
    jmp AllDone  
.L6:   sub eax, esi    # in C, this would be an _else_  
AllDone: ret
```

Labels

A **label** is a unique location in the source code. It is alphanumeric text, such as `.L6` or `AllDone`. 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. Some assemblers prefix local labels (i.e., not external or exported symbols) with a period. As such, almost all function-local labels created by those assemblers that need to be jumped to are prefixed with periods. You are free to use any text you wish for your labels.

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: jmp 1f
2: jmp 1b
1: jmp 2f
2: jmp 1b

.L1: jmp .L3
.L2: jmp .L1
.L3: jmp .L4
.L4: jmp .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 are difficult to predict, 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`. K10 architecture uses this 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 `cmove` family of instructions follow the same condition codes as `setcc` instructions. They move the `dst` argument into `src` if the `cc` condition is true.

```
cmp ecx, eax    # if(c > a)
cmove  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 `cmove` instruction, however, does not require any prediction and is comparable in speed to a regular `mov` instruction.

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

Listing 3. Conditional Moves

```
maxValue:
    mov eax, edi
    mov ecx, esi
    cmp eax, ecx  # Which register has the bigger value
    cmovb eax, ecx # If ecx is bigger, move it to the accumulator
    ret            # Return the larger of the two
```

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

Table 10. Conditional Move Instructions

Instruction	Test	Meaning of Test
<code>cmove</code>	<code>ZF</code>	logically equal or arithmetically zero
<code>cmove</code>	<code>~ZF</code>	logically not equal or arithmetically not zero
<code>cmovs</code>	<code>SF</code>	negative
<code>cmovns</code>	<code>~SF</code>	not negative
<code>cmovl</code>	<code>SF ^ OF</code>	(signed values) less
<code>cmovle</code>	<code>(SF ^ OF) or ZF</code>	(signed values) less or the same
<code>cmovg</code>	<code>~(SF ^ OF) & ~ZF</code>	(signed values) greater
<code>cmovge</code>	<code>~(SF ^ OF)</code>	(signed values) greater or the same
<code>cmovb</code>	<code>CF</code>	(unsigned values) below
<code>cmovbe</code>	<code>CG or ZF</code>	(unsigned values) below or the same
<code>cmova</code>	<code>~CF & ~ZF</code>	(unsigned values) above

Instruction	Test	Meaning of Test
cmove	~CF	(unsigned values) above or the same

Loops

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

Loop instructions

The loop instructions are specialized `jmp` instructions. They assume that the counter register holds the loop index, and they jump to a nearby label after adjusting the counter and testing it.

`loop`

The unconditional `loop` instruction decrements the value of the count register and compares that updated count to zero. While the count is nonzero, it will jump to the provided label.

Listing 4. Equivalent code for loop HEAD

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

`loope, loopz`

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

Listing 5. Approximate code for loope HEAD

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

`loopne, loopnz`

These two loop instructions require count to be nonzero and the ZF to be clear in order to jump.

Listing 6. Approximate code for loopne HEAD

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

High Level Loops vs ASM

Intel assembly language loop instructions most naturally map to what high level languages may

provide in terms of a do-while or repeat-until structure, where the body of the loop is executed at least once, and then a decision is made to repeat the loop or not.

Listing 7. A familiar lyric

```
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"
```

When that 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:

Listing 8. The C code for 99 bottles of beer using do-while

```
void singAboutBeer( unsigned bottles ) {
    do {
        printf( "%d bottles of beer on the wall\n", bottles );
        bottles -= 1;
    } while(bottles);
```

This is why we say that the do-while loop is the simplest in assembly language. A single jump that goes back in the program.

By contrast, a while loop contains an extra test to determine if the loop will be executed at all.

Listing 9. A sample while loop

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

If count were 0 and interpreted as false, the while loop would not run at all.

Exercises

Exercise 1

Hardware Support for Random

Background:

Some Intel chips have hardware support for cryptographic functions. There are three possible outcomes to any attempt to use hardware support.

- The hardware support works as specified.
- The hardware support does not exist and the attempt to use said nonexistent support produces a documented error indication.
- The attempt to use the hardware support produces an unsupported instruction exception.

Purpose:

Practice using the tests and the conditional jumps from the lecture. Write code that can handle different levels of hardware support.

Instructions:

Before changing any code, rebuild the executable and run it.

Make a note of the CPU architecture size and the size of the random number that this code generates.

Modify the random number generator to use the hardware capabilities of the CPU if available. Revert to using the C runtime library functions if the hardware support is missing.

In this exercise, the only file that will require modification is `random.s`

The `makefile` and the test driver can remain unchanged.

Consider one of the three functions in `random.s`

```
/*void getRandomNumberSize( void )
Get a random number. The highest one bit
indicates the size of the number.
*/
getRandomNumberSize:
    call    rand@PLT
    lzcnt  rax, rax
    inc    rax
    ret
```

If random numbers come from the C runtime library, this function is correct. If the random numbers come from CPU hardware, this function needs to be modified. This function must return

the correct answer regardless of the source of the random number. One way to achieve that would be to try to obtain a random number from the CPU. If that attempt is successful, skip the call to the C runtime library.

If available, the `rdrand` instruction places a random number in a general purpose register of your choice and sets the carry flag to 1. If the carry flag is 0, the attempt failed and the function needs to call the C runtime library function.

Modify `getRandomNumberSize`.

Build and run the code.

If the code generates an unsupported instruction exception, revert to the original code (see above) and the exercise is finished.

If the code produces a result, run the code in a debugger and confirm that the result measures the value returned by `rdrand` and not the value returned by the C runtime library function. Confirm that the carry flag is set after the `rdrand` instruction executes.

Once you prove that you can get a random number from CPU hardware, modify the `getRandomNumber` function.

```
/*unsigned getRandomNumber( unsigned )
input register:
    rdi - limit range of random numbers
        to zero to one less than this number
output register:
    rax - a range limited random number
scratch registers used:
    rdx, rax - used to hold results
        when the output of rand()
        is divided by the input parameter in rdi
*/
getRandomNumber:
    # don't let RDI get clobbered by rand()
    push rdi
    call    rand@PLT
    pop    rdi

    # make sure nothing is in rdx prior to division
    xor     rdx, rdx
    # divide random number in RAX
    # by the integer provided by caller
    div    rdi
    # RDX has the remainder.
    # The remainder is what we want
    mov    rax, rdx
    ret
```

Modify this function so that it uses CPU hardware when that feature is available and the C runtime

library function `rand()` when hardware support is not available. As before, the carry flag is set if the `rdrand` instruction worked and reset otherwise.

Make sure that the sign bit in the random number is cleared before you do any division. Also, make sure that `rdx` is zero before you do division.

Build and execute the code.

Run the code in a debugger to make sure the random number came from CPU hardware rather than the C runtime library function `rand()`

Summary:

This exercise practiced using CPU hardware to provide a more cryptographically secure random number. Doing that in a portable way involved testing flags and using conditional jumps.

Exercise 2

Letter Grades

Purpose:

Data formatting and data conversion are common problems. In this exercise you will have the opportunity to use a combination of arithmetic, conditionals and lookup tables to convert a numeric grade into a letter grade.

In this exercise you will create your own test harness, your own make file and your own assembly language solution.

Background:

Table 11. Number grade to letter grade translation table

A	90 and above
B	80 to 89 inclusive
C	70 to 79 inclusive
D	60 to 69 inclusive
F	anything less than 60

Instructions:

Create a directory called `letterGrades`.

Copy the `makefile`, the `SeaShell.c` file and the `random.o` file from the previous exercise.

Create an empty text file called `letterGrades.s` to hold the assembly language source code for the `getLetterGrade` function.

Modify the `makefile` so that it creates an executable called `TestLetterGrades` from the `random.o` and `letterGrades.s` and `SeaShell.c` files. Keep all the standard settings that were used in previous

exercises.

Modify the `testHarness()` function in `SeaShell.c` so that

- it has the necessary `extern` declaration for `getLetterGrade` — a function that takes one unsigned integer as a parameter and returns a char.
- it generates a random number between 55 and 125 inclusive.
- it calls `getLetterGrade()` at least six times using a different randomly generated number each time.
- it displays the random number and the letter grade so that you can visually determine whether your assembly language code works.

Write assembly language code in '`letterGrades.s`' that

- has the correct assembler directives so that the file uses Intel syntax and so that `getLetterGrade` is a public label.
- returns x — lower case letter 'x' — if the calling function passes a parameter higher than 100.
- returns the correct letter grade for any parameter 100 or lower.

NOTE

Make sure `getLetterGrade` validates the input parameter before trying to convert it into a letter grade.

Run the executable at least three times to be sure it is working correctly. Use a debugger as needed.

Summary:

In this exercise, you proved that you could generate all artifacts for a simple project. The exercise required at least some of the comparisons from the lecture material.

Exercise 3

Leap Year

Purpose:

Provide practice performing a non-trivial evaluation in assembler language code.

Background:

The rules for leap years are more complicated than one might suspect. A year is a leap year if it is evenly divisible by four *except* for century years. Century years must be evenly divisible by 400 in order to be leap years. Thus, the year 2000 was a leap year, but 1900 was not. The years 1996 and 2004 were leap years, but 1998 and 2002 were not.

Instructions:

Create a directory called `leapYear`.

Copy the `makefile`, the `SeaShell.c` file and the `random.o` file from the previous exercise.

Create an empty text file called `leapYear.s` to hold the assembly language source code for the `isLeapYear` function.

Modify the `makefile` so that it creates an executable called `TestLeapYear` from the `random.o` and `leapYear.s` and `SeaShell.c` files. Keep all the standard settings that were used in previous exercises.

Modify the `testHarness()` function in `SeaShell.c` so that

- it has the necessary `extern` declaration for `isLeapYear` — a function that takes one unsigned integer as a parameter and returns a boolean value.
- it generates a random number between 1995 and 2020 inclusive.
- it calls `isLeapYear()` at least 10 times with a different randomly generated year each time.
- it displays the random year and whether or not that year is a leap year so that you can visually determine whether your assembly language code works.

Write assembly language code in '`leapYear.s`' that

- has the correct assembler directives so that the file uses Intel syntax and so that `isLeapYear` is a public label.
- correctly evaluates whether a given year is a leap year and returns a boolean indication of that evaluation.

NOTE It is not necessary to range check the input.

Summary:

This exercise provided practice using comparisons.

Chapter 10. Floats

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

When CPUs added Single Instruction, Multiple Data (SIMD) support, the original intention was to process multiple integer data items 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.

ST Use

x87 Floating-point calculations use ST registers. They are a stack of registers. The top of the stack is 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 than 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.

fld and fld

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 fld instruction will perform integer conversion on the loaded data.

```
fld    SRC  
  
fld    QWORD PTR [ebx] # push(st, *b)
```

fst and 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

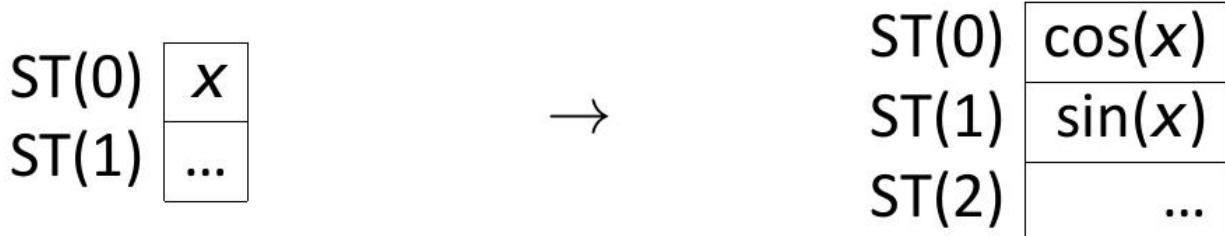
Takes the absolute value by clearing the sign bit of ST(0).

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



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

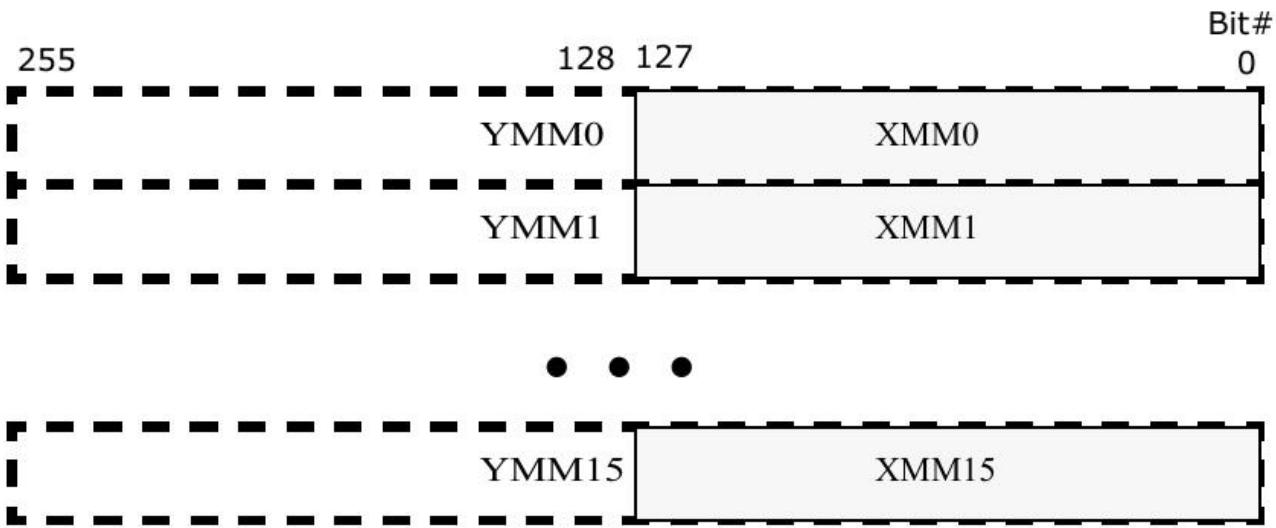
Compare ST(0) with another ST register or memory location, setting the x87 condition codes.

XMM Use

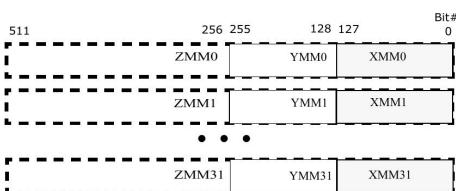
XMM registers are the first major set of Streaming SIMD Extension (SSE) registers and are the default for most modern floating-point operations.

There were originally eight such registers (XMM0-XMM7). The 64 bit architecture expanded that to sixteen registers (XMM0-XMM15). The XMM registers are all 128 bits long. These registers were always intended to hold more than one data item. This chapter focuses on the floating-point application of these registers using one value at a time (scalar, rather than packed).

The next revision of SSE expanded the register set to 32 floating point registers each of which are 256 bits wide. The lower 128 bits are addressed as XMM0 through XMM31. The full register is addressed as YMM0 through YMM31.



A later revision expanded that register set to 512 bit registers. The full register is addressed as ZMM0 through ZMM31. The Z registers overlap the Y registers. The Y registers overlap the X registers.



Instructions that end in ss operate on single-precision floats. Instructions that end in 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. This includes values on the stack. The numbers in memory must already be in either IEEE754 single precision or IEEE754 double precision format.

```
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  
.pi .double 3.1415926535  
.br  
func:  
    movss  xmm0, .euler[rip]    # .euler is already in IEEE754 single precision  
format  
    movsd  xmm1, .pi[rip]       # .pi is already in IEEE754 double precision format
```

Conversion Instructions

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

Conversions become necessary when floating point code has to interface with integer code.

The various `cvt` instructions will truncate the floating-point source value. The various `cvt` instructions will round the integer.

Consider this line of C code which calls a function written in assembly language. The number of bits in the mantissa of a floating point number is an integer. There is no such thing as one tenth of a bit so it makes no sense to pass a floating point number. The number of base 10 digits is an integer. There is no such thing as a fraction of a digit so it makes no sense to return a floating point number.

Listing 10. `digitsOfAccuracy()` is an assembly language function.

```
extern unsigned digitsOfAccuracy( unsigned );  
*  
*  
unsigned ieee754doublePrecisionMantissa = 52;  
  
printf( "IEEE754 double precision floats have about %u base 10 digits of maximum
```

```
precision\n", digitsOfAccuracy(ieee754doublePrecisionMantissa) );
```

There is a need to convert the parameter in EDI to a floating point number in order to enable floating point math. The same need exists in reverse when returning a value in EAX.

Notice that the conversion of the return value from floating point to integer is done by the cvtsd2si instruction.

```
digitsOfAccuracy:  
    cvtsi2sd  xmm1, edi    # convert number of mantissa digits into floating point  
    mulsd     xmm1, Log10of2[rip] # calculate number of base 10 digits of precision  
    cvtsd2si  eax, xmm1    # return value to caller as an integer  
    ret
```

This is the output. Notice the number 16 because the answer was rounded to the nearest integer.

```
IEEE754 double precision floats have about 16 base 10 digits of maximum precision
```

If instead the conversion is performed by the cvttsd2si instruction, the answer will be truncated instead of rounded.

```
digitsOfAccuracy:  
    cvtsi2sd  xmm1, edi    # convert number of mantissa digits into floating point  
    mulsd     xmm1, Log10of2[rip] # calculate number of base 10 digits of precision  
    cvttsd2si  eax, xmm1    # return value to caller as an integer  
    ret
```

This is the output from the exact same calculation as before. Notice the number 15 because the answer was truncated rather than rounded.

```
IEEE754 double precision floats have about 15 base 10 digits of maximum precision
```

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, minsdsd

Notice that each instruction ends in either ss or sd. ss means scalar, single precision. sd means scalar, double precision.

Scalar means that there is only one floating point number in the XMM register and that number is in the lower 64 bits of the register. Single precision floating point numbers are 32 bit IEEE 754 floating point numbers. Double precision floating point numbers are 64 bit IEEE 754 floating point

numbers.

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

Multiply or divide a scalar, double precision value by a constant. Return the result to the caller. Memory locations must be aligned on 16 byte boundaries.

```
milesToKM:  
    mulsd xmm0, Mile[rip]  
    ret  
  
kmToMiles:  
    divsd xmm0, Mile[rip]  
    ret  
  
.balign 16  
Mile:     .double 1.609344    #kilometers
```

The area of a circle is PI times the radius squared. The radius was passed as a parameter in XMM0.

```
areaCircle:  
    mulsd  xmm0, xmm0  
    mulsd  xmm0, PI[rip]  
    ret  
  
.balign 16  
PI:      .double 3.141592653589793
```

The time it takes something to fall to earth is the square root of (the distance / 16)

```
timeToFall:  
    divsd xmm0, Sixteen[rip]  
    sqrtsd xmm0, xmm0  
    ret  
  
.balign 16  
Sixteen: .double 16.0
```

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 opposed to a signalling NaN. These set the RFLAGS register appropriately.

Table 12. Flags and their meanings for each possible comparison outcome

Outcome	ZF	PF	CF
unordered (one or both operands were NaN)	1	1	1
greater than	0	0	0
less than	0	0	1
equivalent	1	0	0

.globl compare

```
/* char * compare( double, double )
Compare two doubles.
Return text indicating which is bigger or if the two numbers are equivalent
```

input registers: XMM0 -- one floating point number
 XMM1 -- another floating point number
return register: RAX -- address of a string

scratch registers: RAX, RCX, RDX */
compare:

```
lea      rax, Greater[rip]
lea      rcx, Equivalent[rip]
lea      rdx, Less[rip]
comisd xmm0, xmm1
cmove  rax, rcx
cmovb  rax, rdx
ret
```

.balign 16

Greater:	.asciz	"is more than"
Equivalent:	.asciz	"is the same as"
Less:	.asciz	"is less than"

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.

Exercises

Exercise 1

Floating Point Accuracy

Purpose:

To illustrate the possible difference between truncation and rounding when converting from floating point numbers to integers. To indicate actual accuracy of single and double precision floating point numbers.

Instructions:

Create a directory called `irrational` and copy the `makefile` and the `SeaShell.c` files from the previous exercise. You will not need the `random.o` file for this exercise.

Create a file called `irrational.s` for this exercise.

Modify your `makefile` to create an executable called `TestIrrational` from the `SeaShell.c` and `irrational.s` files.

Listing 11. Test Harness for the assembly code in irrational.s

```
extern float sqrtTwoSingle( void );
extern double sqrtTwoDouble( void );

extern float oneThirdSingle( void );
extern double oneThirdDouble( void );

extern unsigned digitsOfAccuracy( unsigned );

int testHarness( void ) {
    int resultCode = 0;

    unsigned ieee754singlePrecisionMantissa = 23;
    unsigned ieee754doublePrecisionMantissa = 52;
    unsigned extendedPrecision80bitFloatMantissa = 64;

    printf( "\nIEEE754 single precision floats have about %u base 10 digits of maximum
precision\n", digitsOfAccuracy(ieee754singlePrecisionMantissa) );
    printf( "IEEE754 double precision floats have about %u base 10 digits of maximum
precision\n", digitsOfAccuracy(ieee754doublePrecisionMantissa) );
    printf( "Intel 80 bit extended precision floats have about %u base 10 digits of
maximum precision\n\n", digitsOfAccuracy(extendedPrecision80bitFloatMantissa) );

    puts( "The published calculated square root of 2 is
1.41421356237309504880168872420969807856967" );
    printf( "        Double precision square root of 2 is %.20f\n", sqrtTwoDouble()
);
```

```

    printf( "      Single precision square root of 2 is %.20f\n\n", sqrtTwoSingle()
);

printf( "Double precision one third is %.20f\n", oneThirdDouble() );
printf( "Single precision one third is %.20f\n", oneThirdSingle() );

return resultCode;
} /* end testHarness() - - */

```

In `irrational.s` create the necessary global declarations for the external functions mentioned in the test harness.

It is not possible to move an immediate value into an XMM, YMM or ZMM register. You can move integers into a general purpose register then convert and move it to an XMM register using an appropriate `cvtsi2xx` instruction. Floating point constants can be loaded from memory. Remember to align any memory constants on a 16 byte boundary.

```

.balign 16
Log10f2: .double 0.30102999566398

```

Write the code for the `digitsOfAccuracy` function. The formula is number of digits in the mantissa times the logarithm of 2 in base 10. That is the constant above. The number of digits in the mantissa is an integer parameter that is passed to you in the EDI register. This function has to multiply the parameter by the logarithm. That value has to be converted to an integer and returned to the caller. You can either truncate the answer or round the answer.

Write the code for the `oneThirdSingle` and `oneThirdDouble` functions. They are really the same code. They both divide 1.0 by 3.0 and return the number to their caller. The only difference is that one does this in single precision floating point math while the other does that same math using double precision floating point math. Remember that if you are doing single precision math, your conversions have to be single precision conversions.

Write the code for the `sqrtTwoSingle` and `sqrtTwoDouble` functions. They are really the same code. They both calculate the square root of 2 and return the number to their caller. The only difference is that one does this in single precision floating point math while the other does that same math using double precision floating point math. Remember that if you are doing single precision math, your conversions have to be single precision conversions.

Summary:

There are two ways to convert from floating point numbers to integers. One way involves truncation. That way mimics the way some higher level languages behave. The other way involves rounding. Some applications might work better with truncation. Other applications work better with rounding.

Calculating irrational numbers is a good way to see how many digits of accuracy you can expect from a given floating point format.

Exercise 2

Geometric Functions

Purpose:

To practice using floating point instructions and floating point registers

Background:

You are going to write assembly code that calculates:

- the area of a right triangle. area = length x width / 2.0
- the hypoteneuse of a right triangle. hypoteneuse = square root of (side01 squared + side02 squared)

Instructions:

Create a directory called floatGeometry.

Copy the `makefile`, the `SeaShell.c` test driver and the `random.o` file from the directory you created for the `integerGeometry` exercise.

The `makefile` from `integerGeometry` should not need modification to support this exercise.

Modify the `extern` declarations and the `testHarness()` function in the `SeaShell.c` file.

```
extern void seedGenerator( void );
extern unsigned getRandomNumber( unsigned );

extern double areaTriangle( double, double );
extern double hypoteneuseRightTriangle( double, double );

int testHarness( void ) {
    int resultCode = 0;
    unsigned range = 100;
    double answer = -99;

    seedGenerator();

    double side01 = getRandomNumber( range );
    double side02 = getRandomNumber( range );

    side01++;    side02++;

    answer = areaTriangle( side01, side02 );
    printf( "Triangle height: %f, width: %f, area: %.3f\n", side01, side02,
    answer );

    answer = hypoteneuseRightTriangle( side01, side02 );
```

```

    printf( "Right Triangle height: %f, width: %f, hypoteneuse: %.3f\n", side01,
    side02, answer );

    return resultCode;
} /* end testHarness()      -      - */

```

Create an empty `geometry.s` file. In `geometry.s` write the assembly language code for both equations. Use double precision floating point math.

Consider the `addsd` instruction for adding floating point numbers.

Consider the `mulsd` instruction for multiplying floating point numbers.

Consider the `divsd` instruction for dividing floating point numbers.

Consider the `sqrtsd` instruction for square roots.

Summary:

You just used some very common floating point math instructions and registers to solve two common geometric problems.

Exercise 3

Sorting Floats

Purpose:

Practice calling more complex C runtime library functions from assembly language. Practice encapsulation principles. Practice using read write assembly language data areas. Practice using both direct addressing and indirect addressing. Create and use function pointers.

Background:

Closely related code should be in one place. Memory should never be allowed to leak. A programmer should never be able to predict the test data. Client code should never know anything more than signatures of the functions that client code is allowed to call.

Instructions:

Create a folder called `floatsSorted` and copy `makefile`, `random.o` and `SeaShell.c` from the last exercise.

Create a file called `sortedArrayOfDoubles.s` and adjust your `makefile` accordingly.

The assembly language code in this file will manage all of the very limited functionality.

- Create an array of any length the client code desires.
- Fill that array with random data.

- Sort that data when requested.
- Free the memory allocated for the array automatically no matter how the client program ends.

Modify the `testHarness()` code as shown.

```

extern unsigned getRandomNumber( unsigned );
extern double *createRandomArray( unsigned, unsigned );
extern void sortArray( void );

int testHarness( void ) {
    int resultCode = 0;

    /* Once your code works, increase the range
    and, by implication, the array size until you reach
    at least 100,000,000. */
    unsigned range = 24;

    /* we don't know how big the array will be
       because the array size is randomly generated */
    unsigned arraySize = getRandomNumber( range / 2 );
    arraySize += 6; /* don't allow array size of zero */
    arraySize += arraySize % 6;

    printf( "array size: %u doubles ranging from 1 to %u\n", arraySize, range );
    double *arrayOfDoubles = createRandomArray( arraySize, range );

    for( unsigned i = 0; i < 12; i++ ) {
        printf( "%s%20.10f", ((i % 6 == 0) ? "\n" : "      "), arrayOfDoubles[ i ] );
    }

    puts( "\n+      + Before and after sorting the data +      +\n" );

    sortArray();

    for( unsigned i = 0; i < 12; i++ ) {
        printf( "%s%20.10f", ((i % 6 == 0) ? "\n" : "      "), arrayOfDoubles[ i ] );
    }

    printf( "\n-      -      -      -      -      -\narray size: %u doubles ranging from 1 to
%u\n", arraySize, range );
    return resultCode;
} /* end testHarness()      -      - */

```

This test harness only calls two of the functions in your assembly language file so only those two functions will be global. Any other functions you write will be private to the assembly language file.

The `createRandomArray()` function will take two parameters:

- the number of elements that the array will have and
- what is the largest number you want any double in this array to be.

The function will call a C runtime library function to allocate memory for the array. It will fill that array with randomly generated doubles that range from 1.0 to whatever you chose for an upper limit. Making the array size half of the upper limit for data helps minimize the number of duplicate data items.

Summary:

Read write data needs to be in a data section. Assembly language floating point instructions are *very* fast.

Chapter 11. Week 2, Monday, 7 Hour Encapsulation Lab

This lab requires that you:

- write code as specified
- create test cases and a test plan that reasonably exercises the code you write.
- explain the reasoning behind your choice of test cases
- explain your reasoning behind your choice of code strategies

Exercise 1

Preparation for Lab

Instructions:

Create a directory called week02MondayLab. This lab will require:

- a makefile
- a test harness file to hold your test cases
- a main file to call your test harness and report successes and failures
- a random.o file
- a double.s to hold your assembly language code.

Summary:

The files should compile and execute with no errors and no warnings. The output will probably be nothing more than some trivial text messages.

Exercise 2

No More Leaks

Purpose:

To create memory that never leaks

Instructions:

Write a function called `createDoubleArray` that accepts one parameter — the number of elements that the array has to hold. Dynamically allocate an array at least as large as required. Explain your reasoning for any number higher than requested.

In variables that can be modified, track:

- the address where the array of doubles begins
- the number of elements that the array can hold
- the number of elements that the array currently holds

Register a function with `atexit()` that frees any memory created.

Write a function that reports the capacity — the number of elements it *could* hold — of the array.

Write a function that reports the number of elements of the array that are in use.

Write test cases to exercise the code from this exercise. Write criteria for passing the test cases. Report the number of test cases that passed and the number of test cases that failed.

Record the execution time.

Summary:

No memory should ever leak. There are simple, common sense techniques that can guarantee that memory never leaks.

Exercise 3

Generate Test Data

Purpose:

To guarantee that test data is never predictable. To provide convenient way to generate large amounts of test data.

Background:

No programmer, no matter how diligent, will ever anticipate all the kinds of data that user will throw at an executable. The ability to generate appropriate random data is an important test tool.

Instructions:

Create a globally visible assembly language function that will generate random double precision floating point numbers for all or part of the `arrayOfDoubles` using the following parameters:

- The element where the first random number should go.
- The element where the random numbers should stop. In other words, the element that is *after* the last random number.

This will allow random numbers to:

- fill the entire array, or
- fill X number of elements at the beginning of the array, or
- fill X number of elements in the middle of the array, or
- fill X number of elements at the end of the array.

In this part of the lab, it will also be possible to call this function from the test harness.

Create multiple versions of the random number generator and collect timing information on each. This will:

- demonstrate the cost of Intel's floating point instructions
- suggest several ways to generate random floating point numbers.

Method #1: Use the same random number generator function that has been used in earlier exercises by the test harness. Use an appropriate Intel floating point convert instruction to create a double precision IEEE754 floating point

Method #2: Use the same random number generator function that has been used in earlier exercises by the test harness. Use an appropriate Intel floating point convert instruction to create a double precision IEEE754 floating point. Divide the floating point number by some power of 10 depending on how many decimal places desired.

Method #3: Use the same random number generator function that has been used in earlier exercises by the test harness. Square the highest value random number desired. Get a random

number in that zero to higher limit range. Use an appropriate Intel floating point convert instruction to create a double precision IEEE754 floating point. Take the square root of that random floating point number.

Time the execution of each method of creating a short (less than 10,000 elements), medium (half a million to a million elements) and long (greater than 20 million elements) array. Explain how the time cost of each method changes as the array gets bigger. Explain what this experiment teaches about the relative time cost of each category of different floating point instructions used by the three methods.

Write tests cases for this code. Write criteria for passing or failing the test cases. Record the number of test cases that passed and the number of test cases that failed.

Decide which random number generator you prefer and state your reasons for that choice.

Summary:

If you need random data to test your code, you have it. You had a chance to use some of Intel's floating point instructions. The floating point instructions provided by Intel bring hardware support to a complicated calculation.

Exercise 4

Grow the Array

Purpose:

A data structure should be able to adapt to the size of its data.

Instructions:

Create a non public function that grows the array as needed. Document your choice of event(s) that would cause your code to call this function. Document your choice for determining the new size of the array and your reasons for making this choice.

Develop a test plan for proving that this non public function works as specified. Submit results from executing that test plan.

Summary:

No collection should ever be limited by a fixed length. No client code that uses your collection should ever have to explicitly grow your data structure.

Exercise 5

Get One Element

Purpose:

Retrieving data from a collection is a fundamental necessary behavior.

Background:

In order to test your code, you have to be able to display the elements. Iterating through those elements is the responsibility of the calling code. Your code should provide the building blocks for that usage.

Instructions:

Write a function called `get`. It should take one parameter and return one double.

The parameter shall be the element whose value you want.

The return value shall be the value of that element.

The function shall return the `Not a Number` value if it receives an invalid element parameter.

Write and execute a test plan. Your tests must include measurements of the time it takes for `get` to execute. Record the results.

Does the execution time for `get` vary in any measurable way as the array size grows?

Is that variation meaningful?

Summary:

At this point, you have enough functionality to be able to test your code in a meaningful way.

Exercise 6

Create the Ability to Iterate Through the Collection

Purpose:

Iteration is a fundamentally necessary capability of any collection.

Instructions:

Iteration requires three functions:

- a function to find the first element of the collection
- a function to report whether there is any more data in the collection
- a function to retrieve — get — the next element in the collection

You will almost certainly need additional internal read / write variables to support iteration capability.

Write three functions:

- `getFirstElement` — returns the first value in the collection.
- `hasMoreElements` — reports a true or false indication of whether or not there are any more elements.
- `getNextElement` — returns the next element in the array and updates any relevant internal variables

Write and execute a test plan. Your tests must include measurements of the time it takes for get to execute. At a minimum, your test plan must test:

- a collection of zero elements
- a collection of one element
- a collection of 20 elements

Record the results.

Did the iteration functions return all the data in the array? Does the execution time for get vary in any measurable way as the array size grows?

Is that variation meaningful?

Summary:

Having iteration capability makes some test cases easier to write.

Exercise 7

Change the Value of a Specified Element

Purpose:

Explore the cost of adding user data to the collection

Instructions:

Summary:

Exercise 8

Append Data

Purpose:

Explore the cost of adding user data to the collection

Instructions:

Summary:

Exercise 9

Insert Data

Purpose:

Explore the cost of adding user data to the collection

Instructions:

Summary:

No collection should ever be limited. Once the collection is encapsulated, many strategies become available.

Exercise 10

Sorting Floats

Purpose:

Practice calling more complex C runtime library functions from assembly language. Practice encapsulation principles. Practice using read write assembly language data areas. Practice using both direct addressing and indirect addressing. Create and use function pointers.

Instructions:

Create a folder called floatsSorted and copy `makefile`, `random.o` and `SeaShell.c` from the last exercise.

Create a file called `sortedArrayOfDoubles.s` and adjust your `makefile` accordingly.

The assembly language code in this file will manage all of the very limited functionality.

- Create an array of any length the client code desires.
- Fill that array with random data.
- Sort that data when requested.
- Free the memory allocated for the array automatically no matter how the client program ends.

Modify the `testHarness()` code as shown.

```
extern unsigned getRandomNumber( unsigned );

extern double *createRandomArray( unsigned, unsigned );
extern void sortArray( void );

int testHarness( void ) {
    int resultCode = 0;

    /* Once your code works, increase the range
     * and, by implication, the array size until you reach
     * at least 100,000,000. */
    unsigned range = 24;

    /* we don't know how big the array will be
     * because the array size is randomly generated */
    unsigned arraySize = getRandomNumber( range / 2 );
    arraySize += 6; /* don't allow array size of zero */
    arraySize += arraySize % 6;

    printf( "array size: %u doubles ranging from 1 to %u\n", arraySize, range );
    double *arrayOfDoubles = createRandomArray( arraySize, range );

    for( unsigned i = 0; i < 12; i++ ) {
```

```

printf( "%s%20.10f", ((i % 6 == 0) ? "\n" : " "), arrayOfDoubles[ i ] );
}

puts( "\n+ + Before and after sorting the data + +\n" );

sortArray();

for( unsigned i = 0; i < 12; i++ ) {
printf( "%s%20.10f", ((i % 6 == 0) ? "\n" : " "), arrayOfDoubles[ i ] );
}

printf( "\n- - - - - - -\narray size: %u doubles ranging from 1 to
%u\n", arraySize, range );
return resultCode;
} /* end testHarness() - - - */

```

This test harness only calls two of the functions in your assembly language file so only those two functions will be global. Any other functions you write will be private to the assembly language file.

The `createRandomArray()` function will take two parameters:

- the number of elements that the array will have and
- what is the largest number you want any double in this array to be.

The function will call a C runtime library function to allocate memory for the array. It will fill that array with randomly generated doubles that range from 1.0 to whatever you chose for an upper limit. Making the array size half of the upper limit for data helps minimize the number of duplicate data items.

The `sortArray()` function must call the C runtime library function `qsort()`. That will require a comparison function that can compare two floating point numbers and report which one is larger.

Summary:

Read write data needs to be in a data section. Assembly language floating point instructions are *very* fast.

Exercise 11

Search the Array for a Specific Value

Purpose:

To perform a normal operation needed by any collection of data. The comparisons for floating point numbers are different from the comparisons for text or for integers.

Instructions:

Write a function called `find` that searches for data in the physical array.

Return the element that holds the data, or if the data does not exist in the array, return a negative number.

Record the time it takes to find an item to the array. Does the location make a difference in execution time?

Summary:

Search is a necessary function for any collection that can be modified. The comparisons for floating point numbers are different from the comparisons for text or for integers.

Chapter 12. Review 1

Topics

- Objective
- Assembly Language
- Tools
- CPU
- Registers
- Functions
- Assignments
- Arithmetic
- Branching
- Floats
- Code Review

Objective

This unit is a review of the first half of the course. As such, it is not a repeat of the entire first half. The purpose of the review day is to clarify questions that may have come up when doing exercises, the full-day lab, or reviewing solutions; and to prepare for the written and practical exams to follow.

The topics are not really as divided as it appears. For example, the issue of MUL and DIV with RAX and RDX could as easily be in the Registers section as the Arithmetic section.

The content here is a guide. Please use this time to ask questions.

Assembly Language

- Why do we learn assembly programming?

Based on the experience of having a week of programming in assembly, including a full-day lab, discuss reasons for learning assembly.

- How does programming in assembly language differ from programming in a high level language?

When commenting on that question, did commenting come up? How should commenting be approached with assembly language?

Tools

Review `make`, `gcc` and `as`. What about `ld` and `ldd`? Should either be used, at least directly? If not, why not?

CPU

Make sure that there is fluency reading the instruction set documentation, so that each instruction and the kind(s) of parameters it can accept is understood. No one memorizes the entire instruction set. References are a programmer's constant companion.

Review Long Mode. It is the only one used for 64 bit programming on Intel.

What happens to the segment registers?

What is meant by instruction pipelining? How can that impact performance?

Registers

Review the different registers available, and their normal uses. Make sure that 8, 16, 32 and 64 bit registers are understood.

Functions

Review in detail CALLER and CALLEE saved registers. Bookmark the System V AMD64 ABI specification, and keep the Register Usage table handy!

Functions are vital to allow code to be maintained. When writing assembly code, it is recommended that code be liberally commented. Every line of assembly code should have a comment, generally speaking. Functions should document:

- register usage on entry
- any stack usage on entry
- register usage within the function
- any local stack usage
- any dynamic allocation performed within the function, especially if it is being returned to the caller (who would be responsible for clean-up)
- register usage on exit

Write the documentation as if explaining it to another party.

Assignments

Make sure that there is fluency with the different types of parameters: immediate, register and memory. This ties back to the topic of fluency reading the instruction set documentation.

Make sure that the differences between LEA and MOV are understood.

Make sure that SIB addressing is well understood.

What happens to the higher order bits of a register when moving an 8, 16 or 32 bit value into a register? What happens to the rest of RAX when loading 8 bits into AX? Is there a difference between how the 32 bit and 64 bit register extensions handle these cases? If necessary, review MOV vs MOVZX vs MOVSX.

Arithmetic

Review the A and D registers, especially with respect to MUL and DIV instructions. Why might it be important to clear the D register before doing a DIV?

Review all of the math operations, including ones like ADC and XADD, and how to use them.

Branching

Review and be fluent with all forms of Jcc.

Review jump and call tables, and how those work with the `jmp`, `Jcc` and `call` instructions.

Floats

Review and be fluent, assuming reference to documentation and sample code, with all aspects of the floating point stack, including the ST and YMM register sets.

Chapter 13. 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.

This course focuses almost exclusively on the Intel syntax, with the exception of a sidebar reference related to SIB addressing. However, one is extremely likely to encounter both Intel and AT&T syntax.

This unit also covers important directives, which will be helpful in future units, such as strings and structures.

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.

Listing 12. 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.

Listing 13. 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.

Listing 14. 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`.

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

`sar r/m16, imm8`

This 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 syntax, but is a closer match to the underlying machine code. This syntax is seen more often in UNIX 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
------	----------

Listing 15. 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.

Listing 16. 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.

Listing 17. Move the four bytes at memory address 0x1000 to eax

movl	0x1000, //eax
------	---------------

Current Use

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

.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). Because of this unportability, this directive should be avoided.

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 *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 floating point numbers.

Symbol Table Directives

Debugging information can be very verbose at the assembly level. Debugging information is read from a separate **symbol table**. This Symbol Table can have entries added to it via various 'stab' (Symbol TABLE) directives.

```
' .stabs _string_, _type_
` .stabn _type_
` .stabd _type_'
```

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 repetitive or obscure code.

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

iter 1, 7
```

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

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

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

Chapter 14. 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.

Structure

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

*Table 13.
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.

Block Started by Symbol (BSS)

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 actual 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.

`prefetchnta _dst`

Load a value that will be used once, and then never again (i.e., will not store the value in caches).

`prefetcht0 _dst`

`prefetcht1 _dst`

`prefetcht2 _dst`

Load 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 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.

ref: <https://lwn.net/Articles/444344/>

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 14. Latency Numbers Every Programmer Should Know

Operations	Multiplier
Register	1
L1 cache reference	2
Branch mispredict	20
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

ref :Peter Norvig: <http://norvig.com/21-days.html>

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.

Branch Misprediction

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.

This is why we have conditional move instructions: to avoid branch misprediction penalties.

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)`.

Exercise 3

Consider the cache-friendliness of a linked list versus an array.

Exercise 4

What is the performance difference between calculating Fibonacci numbers to using 8 registers to hold 256 bit integers (using `adc` and `xadd`) vs using memory to hold 3 256 bit numbers?

Exercise 5

Compare the performance of these alternative implementations. Taking into account the cost of branch misprediction:

```
high_low:  
    cmp rdi, rsi          # compare  
    cmovl  rax, [rip + less]    # rdi < rsi  
    cmove  rax, [rip + same]    # rdi == rsi  
    cmovg  rax, [rip + high]    # rdi > rsi  
    ret
```

```
high_low  
    cmp rdi, rsi          # compare  
    jae .above_or_equal      # rdi >= rsi?  
    mov rax, [rip + less]    # rdi < rsi  
    jmp .done  
.above_or_equal:  
    ja  .above             # rdi > rsi?  
    mov rax, [rip + same]    # rdi == rsi  
    jmp .done  
.above:  
    mov rax, [rip + high]    # rdi > rsi  
.done:  
    ret
```


Chapter 15. Strings

The most common data structure is a block of raw memory. Literature describing the instructions covered here talk about "string instructions". There is, however, no requirement that the data be one byte entities described by the ASCII character set or any other character set. Indeed, Intel's "string instructions" can be used to process binary data. The "string instructions" that examine and move raw bytes of memory use the accumulator, the count register, and the source/destination registers.

The versions of these instructions for 16 bit and 32 bit processors use combinations of the data segment register—DS—and the extra segment register—ES—plus the source index register—ESI—and destination register—EDI—to specify addresses. Application code that uses the version of these instructions for 64 bit processors will find that the operating system sets the ES and DS registers to zero. Intel reference material still references the segment registers.

lod_s, st_{os}

These instructions are called **load string** and **store string**. They work on raw bytes. The register in question is the accumulator. The memory location that contains the data to be loaded into the accumulator is specified by the source index—RSI. The memory location that will receive the accumulator data is specified by the destination index—RDI. That is how those registers got their names. After lod_s or st_{os} execute, the source or destination is automatically incremented by the size of the operand.

Listing 18. Equivalent code for lodsb

```
mov al, [rsi]  
inc rsi  # Assuming the direction flag is 0
```

Listing 19. Equivalent code for stosw

```
mov [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.

Listing 20. Equivalent code for scasq

```
cmp rax, es:[rdi]
lea rdi, [rdi + 8] # Assuming the direction flag is 0
```

MOVVS

The **move string** operation moves bytes between two memory locations. Both index registers are increased by the size of the data moved.

Listing 21. Approximate code for movsb

```
mov [rdi], [rsi]  # Note that these are not valid 'mov' arguments
inc rsi  # Assuming the direction flag is 0
inc rdi  # Assuming the direction flag is 0
```

cmps

Compare String compares the values at two memory locations, like a `cmp` instruction. It will increase both the index registers by an appropriate amount.

Listing 22. Equivalent code for cmpsd

```
cmp [rsi], [rdi]
lea rsi, [rsi + 4] # Assuming the direction flag is 0
lea rdi, [rdi + 4] # Assuming the direction flag is 0
```

rep, repz, repnz

A powerful asset to these string instructions is **Repeat** modifier. This prefix instruction will execute the specified string instruction until any the termination 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

Listing 23. Equivalent code for repz scash

```
1:  
test    rcx, rcx  
je 1f  
cmp al, [rdi]  
je 1f  
inc rdi  # Assuming the direction flag is 0  
dec rcx  # Assuming the direction flag is 0  
jne 1b  
1:  
inc rdi  # Assuming the direction flag is 0  
dec rcx  # Assuming the direction flag is 0  
(continue processing)
```

Note that this is an extremely compact way to express common high-level constructs. Using this set of instructions, fewer branches are needed.

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 to 1 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 to 1, 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.

Exercises

Exercise 1

Preparation for Text String Exercises

Purpose:

To create a directory, a makefile, a test harness and an empty assembly language file that can be used for the next few exercises.

Instructions:

Create a directory called `searchText`.

Create an empty file called `search.s`

Write the following line at the beginning of the file:

```
.intel_syntax noprefix
```

Copy the following files from another exercise directory:

- `SeaShell.c`
- `makefile`
- `random.o`

Correct the '`makefile`' so that it uses '`search.s`' and creates an executable called '`TestSearch`'

Remove any external declarations except '`getRemoteNumber`'

Remove any test code from the `testHarness()` method.

The executable should build with no errors.

Summary:

You are now ready to do the next few exercises.

Exercise 2

Some C Runtime Library String Functions in Assembly Language

Purpose:

To practice using Intel's scan instruction and Intel's repeat prefix.

Instructions:

You are going to use Intel's scasb instruction and an appropriate repeat prefix to write your own version of the C runtime library function, `strlen()`

For simplicity sake, you may assume that every string you process *will* have a null terminating byte.

Your test harness should pass at least these four strings to function you write.

- "" — a zero length string.
- "Short" — a short, simple string
- "My name is John Henry Peter Simson-Boles, III (password — <\$enior>M0n3y{Officia!})"
- "We hold these truths to be self evident:\n\tThat all men are created equal\n\tThat they are endowed by their Creator with certain inalienable rights.\n\t\t[from the United States Declaration of Independence, 1776]"

Feel free to add to that minimal set of test cases, but at a minimum, use that text. The text provides a variety of lengths that will test at least one boundary case, and most but intentionally not all of the characters in the USASCII character set. Compare the result from your function with the result from the C runtime library function. Report success if your function reports the same length as `strlen()`

Write your own version of `strlen()`. Use the same parameters and return value as the C runtime library function. Test your code with at least the test strings above. Print the text string and your calculated length and the C runtime library calculation of the length. You succeeded if your function produces the same output as the C runtime library function.

Write your own version of `strcpy()`. Use the same parameters and return value as the C runtime library function. Test your code with the same test strings that you used for `strlen()`. Print the original and your copy of the original and the C runtime library copy of the original. You succeeded if all versions look the same.

Write your own version of `strncpy()`. Use the same parameters and return value as the C runtime library function. Use a random number between 2 and 40 for the maximum number of characters to copy. Test your code with the same test strings that you used for `strlen()`. Print the original and your copy of the original and the C runtime library copy of the original. You succeeded if your code and the C runtime library function produce the same output.

Write your own version of `strchr()`. Use the same parameters and return value as the C runtime library function. Test your code with the same test strings that you used for `strlen()`. Use the random generator that you created in an earlier exercise to generate random upper and lower case characters. Search the strings for these characters. Print out the random character and the location of that character in the test string (or the failure to find it). You succeeded if your code and the C runtime library function produce the same output.

Summary:

Rewriting standard C runtime library string functions provides practice searching forward and

backward. Each of these functions has its own set of boundary conditions and special cases.

Exercise 3

Preparation for the Binary Scan Exercises

Purpose:

To create a directory, a makefile, a test harness and an empty assembly language file that can be used for the next few exercises.

Background:

Intel's scan and repeat instructions can be used for more than just text searches. They can be used on binary data as well and they can be used to move 64 bits—8 bytes—of data at a time.

Instructions:

Create a directory called searchBinary.

Create an empty file called search.s

Write the following line at the beginning of the file:

```
.intel_syntax noprefix
```

Copy the following files from another exercise directory:

- SeaShell.c
- makefile
- random.o

Correct the 'makefile' so that it uses 'search.s' and creates an executable called 'TestSearch'

Remove any external declarations except 'getRemoteNumber'

Remove any test code from the testHarness() method.

The executable should build with no errors.

Summary:

You are now ready to do the next exercises.

Exercise 4

Create the Skeleton Code

Purpose:

To practice loops and to perform comparisons on binary data

Background:

Searching for one item in a large collection of items is a fairly common task. The Intel instruction set has some instructions that are specialized for searching. In this exercise, you will explore those instructions and the Intel prefixes for repeating.

Copying buffer from one memory location to another is a fairly common task. The Intel instruction set has some instructions that are specialized for block moves. In this exercise, you will explore those instructions and the Intel prefixes for repeating.

As you will see in this exercise, the Intel instructions can search for much more than just text.

Instructions:

Here is a skeleton testHarness() function.

```
extern unsigned getRandomNumber( unsigned );

extern void *createArray( unsigned );
extern unsigned getCapacity( void );
extern unsigned getNumberOfItems( void );

int testHarness( void ) {
    int resultCode = 0;
    int failedToAllocateIntegerArray = -6;
    unsigned numberOfElements = 5 + getRandomNumber( 100 );
    clock_t start = clock();
    printf( "There are %ld clock ticks per second\n", CLOCKS_PER_SEC );
    printf( "A short is %ld bytes. An int is %ld bytes. A long is %ld bytes\n\n",
            sizeof( short ), sizeof( int ), sizeof( long ) );

    void *array = createArray( numberOfElements );
    printf( "Created new integer array at %p with capacity for %u elements\n",
            array, getCapacity() );
    printf( "That array currently holds %u elements\n\n", getNumberOfItems() );
    if( array == 0 ) {
        resultCode = failedToAllocateIntegerArray;
    }

    clock_t end = clock();
    printf( "testHarness took %ld clock ticks\n", end - start );
    return resultCode;
```

```
 } /* end testHarness() - - */
```

Starting with this exercise, you will begin to collect execution time metrics. The standard C runtime `clock()` function is useful for this.

Also new in this exercise, you will report test failures. A non-zero return value from `testHarness()` will indicate failure.

Create the skeleton `testHarness()` function.

Here is a skeleton for `search.s`

This file will manage an array of integers. Ultimately, that array will grow as needed. The `testHarness()` function will measure the cost in time for any code in this file.

The next exercises will add functionality to this file.

```
.intel_syntax noprefix

.globl createArray
.globl getCapacity
.globl getNumberOfItems

/* unsigned createArray( unsigned )
Create an array with this number of elements. All elements are initialized to zero.
Input Register: RDI -- unsigned initial number of elements in array
Output Register: RAX -- success or failure indicator
                  zero indicates failure
                  non-zero indicates success
*/
createArray:
    ret

/* This function releases resources at the end of program execution.
Notice that this function was __NOT__ declared global. */
destroyArray:
    mov    rdi, arrayOfIntegers[rip]
    call   free@PLT
    mov    QWORD PTR [arrayOfIntegers+rip], 0
    lea    rdi, ConfirmationThatFreeGotCalled[rip]
    call   puts@PLT
    ret

/* unsigned getNumberOfItems( void )
Tell caller how many elements are in the array right now
input registers: none
output registers: RAX -- number of elements in array */
getNumberOfItems:
```

```

ret

/* unsigned getCapacity( void )
Tell caller how many elements the array can hold
input registers: none
output registers: RAX -- number of elements array can hold */
getCapacity:
    ret

/* Read only data goes here */
.balign 16
ConfirmationThatFreeGotCalled:
    .asciz "Memory was released by calling free()"

/* Data that might get modified goes in this section */
.section .data
.balign 16
arrayOfIntegers:     .quad 0
itemsInArray:         .quad 0
appendNextItemHere:  .quad 0
endOfPhysicalArray:  .quad 0

temporaryArrayPointer: .quad 0

```

NOTE

Variables that might get modified belong in a data section. Read only variables can go in the text section. The linker marks the .data section differently.

There are several pointers that will become useful in later exercises. For now, be aware that they exist.

arrayOfIntegers holds the address of the memory that holds the array of integers. It is very likely that the memory here will be larger than the data in the array.

endOfPhysicalArray holds the address just after the last byte that was allocated in the array.

appendNextItemHere holds the address of the first empty element in the array. When the array is first allocated, appendNextItemHere and arrayOfIntegers will hold the same address.

itemsInArray should always equal appendNextItemHere minus arrayOfIntegers divided by the size of an integer (4 bytes).

The calculation for capacity is endOfPhysicalArray minus arrayOfIntegers divided by the size of an integer (4 bytes)

It is very important that these variables be properly updated any time memory is allocated or reallocated.

In this exercise,

- complete the code for the `createArray` function. Use `calloc()` to allocate the memory for the array.
- complete the code for the `getCapacity` function.
- complete the code for the `getNumberOfItems` function.

Make sure to update the pointers and variables any time you create a new array. Make sure to register the `destroyArray` function with the standard C runtime function `atexit`. Look up man page for `atexit()` if necessary. Does the man page give you any guidance about how many functions you can register with `atexit()`? What did you do in `testHarness()` to verify that number of functions?

The name of any function is a pointer to that function.

Summary:

There are things you can do to eliminate memory leaks. One is to bury memory allocation and dealocation inside a library of some kind. Another is to use operating system facilities in order to guarantee that allocated memory gets freed.

Exercise 5

Append an Integer to the End of the Array

Purpose:

To perform a very simple way to add data to the array.

Instructions:

Write a function called `append` that takes an unsigned integer and adds it — appends it — to the end of the physical array. If the array is not large enough to hold the new element, grow the array.

How will you know if there is enough room in the array? If the array has to grow, how big should you make the new array?

Remember to adjust any pointers as needed.

Record the time it takes to append an item to the array.

Summary:

Exercise 6

Get One Element from the Array

Purpose:

Provide a way to know what values are in the array.

Instructions:

Write a function called `get` that will return the value of the exement at position X in the physical

array.

Return the largest possible unsigned integer if the caller asks for an element beyond the end of the array.

Time the execution of the function. Does the location of the data make a difference in execution time?

Summary:

This function gives you the ability to test the array to know what elements are in it.

Exercise 7

Insert an Integer into Some Random Location in the Array

Purpose:

To practice using Intel's `movs` and `rep` string instructions on binary data. Insert requires you to move all the data below the insertion point down one element in order to make room for the new element.

Instructions:

Write a function called `insert` that takes an unsigned integer and inserts it at the specified element of the physical array. For this exercise, do not insert more data than the collection can hold.

Fail silently if the caller tries to insert data after the end of the array.

How will you know if there is enough room in the array?

Remember to adjust any pointers as needed.

Record the time it takes to insert an item to the array. Does the insert location make a difference in execution time?

Summary:

Inserting into an array involves moving elements in order to create a hole in the collection for the new data. Intel's `movs` instruction coupled with some form of Intel's `rep` prefix makes this very easy code to write.

Exercise 8

Search for a Particular Data Item

Purpose:

Explore Intel's `scas` instruction.

Instructions:

Use the code you have written so far to create an array holding 10,000 or so random numbers. In search.s write two publicly visible assembly language functions that take the same parameters.

- the unsigned integer input parameter is the value to search for
- the signed integer return value is either the element number that holds that value or a negative number to indicate that the value was not found.

Both functions should search the array for the requested data and return the first element number that holds that data. One function should use some form of the scas instruction to search for the data. The other should solve the problem without using the scas instruction.

Which function was easier to write?

Which function performs better?

Test both functions as described below. Search for data that exists:

- in the first 20% of the array
- in the second 20% of the array
- in the third 20% of the array
- in the fourth 20% of the array
- in the last 20% of the array

Search for data that you know does not exist in the array.

Record the time it takes to find (or not find) an item in the array. Does the location of the data make a difference in execution time? Which takes longer — searching for data that exists or searching for data that does not exist?

Summary:

Various forms of Intel's scas instruction are useful when searching for binary data.

Chapter 16. Interfacing with Linux

Applications need a means to interact with the operating system. Interacting with the operating system might mean:

- invoking an operating system provided service, such as accessing resources (file, network, memory, etc.), launching a new process, or other
- receiving a notice from the operating system that something occurred, such as a ^C from the keyboard or a divide-by-zero error in the CPU
- being launched by the operating system, and receiving parameters

System Calls

A platform needs to provide an application some means to invoke services from the operating system. Different platforms do it differently. For example, the old way with IA-32 was to use the `int` (interrupt) instruction. For System V AMD64, the `syscall` instruction is used.

IA-32 using `int`

On IA-32 systems, any system call is invoked by raising an interrupt. A specific interrupt code signals to the kernel that it is desired for a system call to be executed. All system-level calls are done via this interrupt.

In this example, the desired function call is in `eax`. `eax` is a value into the complete lookup table of system calls. On Linux, these system call numbers are usually found in `sys/syscall.h`.

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

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

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

System V AMD64 ABI `syscall`

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

When using `syscall`, `RCX` temporarily stores `RIP`, and `R11` stores a copy of the `FLAGS` register, as the call transitions from user-space to kernel space. The stack is not used.

The standard System V AMD64 ABI is followed with the following differences:

On entry:

- `RAX` is used to hold the system call number
- `RCX` is used to store the return address (`RIP`)
- `R11` preserves `RFLAGS`
- `RDI` is used for the first parameter, as normal
- `RSI` is used for the second parameter, as normal
- `RDX` is used for the third parameter, as normal
- `R10` is used for the fourth parameter, instead of `RCX`
- `R8` is used for the fifth parameter, as normal

- R0 is used for the sixth parameter, as normal
- The stack is **not** used with `syscall`.

On exit:

- RAX has the return value

```
mov RAX, 0x3D  
syscall
```

CAVEAT: Although user applications can invoke the kernel directly using the System V AMD64 ABI as described, the documentation for the ABI states that:

"User-level applications that like to call system calls should use the functions from the C library."

System V Application Binary Interface AMD64 Architecture Processor Supplement Section: A.2.1

Yes, programmers are recommended to eschew the use of `syscall` directly, and to rely upon the C runtime library. Should one choose to ignore that recommendation, and invoke the kernel directly, one would need the documentation for each kernel service. A commonly cited reference is Ryan A Chapman's from 2012 (https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/), which contains a table documenting the majority of system calls. It is for version 4.7 of the kernel, and is somewhat out of date, but would suffice for most uses. The official list of system call numbers can be found in Linux Torvalds's GITHUB repository (https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl).

Command Line Parameters

Programmers using C, Java, Python, Perl, or pretty much anything else, are used to being able to receive command line arguments in a string array. For C programmers, that leads to the well-known definition of `int main(int argc, char** argv)` or `int main(int argc, char* argv[])`. For Java programmers, it would be `public static void main(String[] args)`, where the String array provides its own length. By convention, the first string is the name of the program, followed by any actual parameters as parsed from the command line, or otherwise provided (to the exec family of functions).

Most examples in this course have relied upon the C runtime to invoke a `main` entry point in programs. On entry to `main`, RDI has `argc` and RSI has `argv`.

However, this is not how Linux launches a process.

_start

The details by which Linux launches a process are described in <https://0xax.gitbooks.io/linux-insides/content/Misc/linux-misc-4.html>, but we can summarize it that when the operating system launches a process, it invokes an entry point normally called `_start`. Eventually, if using the C runtime library, that leads to `main` being invoked.

This program, despite being written in AMD64 assembly code, uses the C runtime library, and provides a `main` to match the above definition.

Listing 24. echo.s

```
.intel_syntax noprefix

# Symbolic references to registers, for convenience
argc=r12
argv=r13
next=r14

.globl main
main:
    mov argc,rdi      # preserve argc
    mov argv,rsi      # preserve argv

    xor next,next     # zero the counter

nextArgument:
    mov rdi,[argv + 8*next] # next string into rdi
    call puts@plt      # call put to print rdi
    inc next          # increment to the next argv index
    cmp next,argc     # less than argv?
    jl nextArgument   # if so, print the next one

    ret
```

The code will print out the parameters provided when starting this executable. In the traditional manner, the parameters passed to main are:

- argc — a count of the number of pointers to string in argv
- argv — an array of pointers to string passed on the command line

This leads some people to believe that the C runtime is responsible for parsing the command line parameters, but as the ABI makes clear, this is not the case. The various forms of exec in the C runtime, which map directly to a kernel system call, take the list as a parameter, and that is passed onto the newly launched program. So who parses the command line? The shell, itself, and no other entity.

Before calling a programs OS-defined entry point, the arguments are actually pushed onto the stack by the kernel:

- [RSP] — this is where argc is stored, and is the last thing pushed onto the stack
- [RSP + 8*i] — a pointer to each string passed to the program

The order in which they are passed onto the stack is the **reverse** order in which we would process them. The last command line argument is pushed first, then the next, until the first command line argument, and finally the argument count. Pushing them in the reservse order means that they will appear to be an array in the correct order.

The OS-defined startup function, _start, retrieves this information from the stack, and puts it into the standard ABI defined registers before calling the language defined entry point; in our case, main().

Listing 25. echo-nomain.s

```
.intel_syntax  noprefix

# Symbolic references to registers, for convenience
argc=r12
argv=r13
next=r14

.globl _start
_start:
    mov argc,[rsp]    # preserve argc
    lea argv,[rsp+8]   # preserve argv

    xor next,next      # zero the counter

nextArgument:
    mov rdi,[argv + 8*next] # next string into rdi
    call  puts@plt    # call put to print rdi
    inc next        # increment to the next argv index
    cmp next,argc    # less than argv?
    jl  nextArgument # if so, print the next one
```

```
mov rax, 60      # call code for exit
mov rdi, 0        # exit with success
syscall
```

In summary, programmers can choose to use `_start` and `syscall`, or follow the advice of the System V AMD64 ABI, and leverage the C runtime library, as most examples in this course have done.

In fact, it is the not too subtle reason why the course is called "Assembly Language Using C": not because we want to use the C language, but because we use the C library.

Because parameters are plain text, they must be value checked and converted as necessary. There are many library routines for managing complex sets of parameters, including `getopt(3)`.

Some good practices:

- Do not depend on arguments being in any special order.
- Do not depend on all defined arguments being provided.
- Provide reasonable default values for all parameters.
- Unexpected and/or invalid parameters should never cause a program to misbehave.

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.

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.

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 execution 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.

Red Zone

As introduced elsewhere in the course, the System V AMD64 ABI guarantees that the 128 bytes below RSP (remember, the stack grows DOWNWARDS in terms of address) will not be overwritten by asynchronous activity. This is where that comes to fruition. When a signal is processed, the current function, whatever it is, need not worry about the 128 bytes below RSP being used by an asynchronously invoked handler. Handler code must stay away from the red zone.

Exercises

Exercise 1

Echo

Purpose:

To understand the interaction between the command line and the startup routines

Instructions:

Create an assembly language file named echo.s. Copy the source code from the lecture into that file. Compile that file into an executable called myEcho using the command line below.

```
gcc -m64 -masm=intel -fno-asynchronous-unwind-tables -std=c11 -o myEcho echo.s
```

Make a symbolic link to the executable.

```
ln -s myEcho Fake
```

Execute your code with the following command lines:

- ./myEcho What goes up must come down
- ./Fake What goes up must come down
- ./myEcho "What goes up must come down"
- ./myEcho What goes up must come down | sort
- ./myEcho "What goes up must come down" | sort

Summary:

The command shell will process your command line before you do.

Exercise 2

Parse and Validate a Command Line

Purpose:

Practice parsing command line parameters

Instructions:

Create the following source files:

- testHarness.c

- main.s

Create a makefile that will create an executable named TestMain from the above files.

In the `testHarness.c` file, create a C language function called `testHarness()` that accepts and displays the following parameters:

- a single character
- an integer
- a double
- a string

In the `main.s` file, create an assembly language globally visible function called `main` that provides default values for:

- a single character
- an integer
- a double
- a string

The `main` function should parse and convert any command line functions it finds. All parameters shall be optional. All parameters shall be case sensitive. There must be whitespace between the parameters and their values.

Table 15. Optional parameters

Parameter Name	Parameter Value
-c	If present, -c will indicate that the value will be a single character. It is an error for the value to be more than one character long.
-i	If present, -i will indicate that the value will be an integer. It is an error if the value contains any non-numeric characters.
-f	If present, -f will indicate that the value will be a number of some kind. It is an error if the value contains any non-numeric characters other than one single period.
-s	If present, -s will indicate that the value will be a string. All characters are legal. There is no limit on the string length.

Build and execute your work. The executable should work correctly with:

- no parameters.
- any single parameter.
- all parameters.

- any combination of parameters in any order.

Any parameters not supplied on the command line should pick up their default values.

Improper use of parameters should be reported as an error. Any parameters not on the above list should be reported as an error.

Summary:

The code from this exercise will serve as a model for a main function in future exercises. Parsing code and error processing can easily consume large amounts of the total quantity of code needed to solve any problem.

Chapter 17. Structures

Assembly language has no concept of structures in the same sense that higher-level languages do. All assembly language understands is blocks of memory. In assembly language, structures are just a block of memory, and the fields within that block are calculated offsets from the beginning of 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 by some chips and some assemblers.

Different high level languages and different compilers for the same language might lay out fields in a structure differently. Compilers that create packed structures leave no gaps between the fields that make up the structure. Other compilers start each field on a word or a double word boundary. Still others offer a choice.

If a structure has to be processed by more than one function, the safest, most portable way to pass that structure between two different functions is to

- pass the address of each individual field in the structure (if the called function is expected to modify values of the fields) or
- pass the values of each individual field in the structure (if the data in the structure is read only)

If an entire structure must be passed, be sure to document any padding between fields. Packed structures have no padding between fields.

Bit Encodings

All values in programming are represented by bits. Whether the value in question is an integer, a character, a floating-point value, or a pointer to something else is purely based on **how those bits are interpreted**. Complex structures are made from these primitive types. Each primitive type has a bit encoding.

It is the responsibility of the programmer to interpret bits correctly. Assembly programming provides no tools to check that an integer isn't being treated as a float or any other form of such checks, and relies on the writer to know what they are doing as data is structured and processed.

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	1	0	1
128	64	32	16	8	4	2	1
85	=	64	+ 16	+ 16	+ 4	+ 1	

One of the useful properties of binary is how simple the arithmetic tables are.

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

Binary formatting also makes multiplication extremely straightforward using the standard algorithm.

		1	0	1	0	1
x				1	1	1
		1	0	1	0	1
		1	0	1	0	0
+	1	0	1	0	1	0
	1	0	0	1	0	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	1	0	1
-128	64	32	16	8	4	2	1
85	=	64	+	16	+	4	+
-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	0	1	0	1	0
-128	64	32	16	8	4	2	1
-86	-128	+	32	+	8	+	2

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 both 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 120,000 are used^{as of Unicode 8.0, published June 2015}.

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

$$6.02214 \times 10^{23}$$

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] or [1, 10). Note that in base-10, 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:

$$1.11111100001100001_2 \times 10_2^{1001110_2}$$

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 given 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. The sign of the mantissa (positive or negative) is encoded as a single bit in its own section.

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	1	0	0	1	1	0	1	1	1	1	1	1	1	1	0	0	0	1	1

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

Comparison of Structures in C and in Assembly Language

The CPUID assembly language instruction returns blocks of information that could be presented in a structure. We shall examine how that information would be organized in assembly language and compare that with the way that a C structure would organize that same information.

The CPUID instruction

Every modern Intel CPU can describe its capabilities in several ways. One of those ways is by responses to the CPUID instruction. Code queries the CPU by passing values in the EAX and ECX registers when it executes the CPUID instruction. The CPU returns processor identification and feature information in the EAX, EBX, ECX and EDX registers.

Intel documents several leafs of information. Each of those leaves can be thought of as a structure.

The most basic leaf is leaf 0x00. Assembly language code interrogates this leaf by putting 0x00 in EAX and 0x00 in ECX.

```
.intel_syntax noprefix

.globl cpuInfo00
.type cpuInfo00, @function

.align 16
cpuInfo00:
    mov eax, 0    #cpuid information request
    mov ecx, 0
    cpuid
    mov MaxCpuInformationLeaf[rip], eax
    mov GenuineOrNot [rip], ebx
    mov [GenuineOrNot + rip + 4], edx
    mov [GenuineOrNot + rip + 8], ecx
    lea    rax, [rip + GenuineOrNot]
    ret

.data
.align 16
GenuineOrNot:
.zero 16
MaxCpuInformationLeaf:
.zero    4
```

The response from the CPUID instruction is in EBX, EDX and ECX.

- EBX holds the ascii codes for "Genu"
- EDX holds the ascii codes for "ineI"

- ECX holds the ascii codes for "ntel"

If you append those three registers together, it spells "GenuineIntel".

The EAX register holds the number of the highest leaf that this particular CPU has. The data area set aside to hold this data looks like this C language structure.

```
struct cpuInfoLeaf00 {
    char cpuBrand[16];
    unsigned highestCpuLeaf;
};
```

In this example, none of the potential problems manifest themselves.

- There are no problems with padding.
- There are no problems with byte order.
- There are no problems with mismatched data sizes.
- The memory that holds the structure has not been released before the calling function tries to use it.
- The calling function makes no attempt to modify any data in the structure.

When the assembly language function creates multiple data items packed into a register, the task requires a little more effort.

Consider the information provided in leaf 01 by the CPUINFO instruction.

- Processor family number is returned in EAX bits 8 through 11
- Model number is returned in EAX bits 4 through 7
- Stepping ID is returned in EAX bits 0 through 3

There are no four bit data types in the C programming language. That data has to be masked out of the registered and stored in a standard C data type.

The data in assembly language.

```
cpuInfoLeaf01:
# extracted from the EAX register
extendedFamily: .zero 1
extendedModel: .zero 1
processorType: .zero 1
family: .zero 1
model: .zero 1
stepping: .zero 1
# extracted from the EBX register
initialAPICid: .zero 1
clflushLineSize: .zero 1
brandIndex: .zero 1
```

```
# registers as returned from CPUID Leaf 01
```

```
leaf01ECX: .zero 4  
leaf01EDX: .zero 4
```

The same data in a C structure

```
struct cpuInfoLeaf01 {  
    /* from EAX */  
    unsigned char extendedFamily;  
    unsigned char extendedModel;  
    unsigned char processorType;  
    unsigned char family;  
    unsigned char model;  
    unsigned char stepping;  
    /* from EBX */  
    unsigned char initialAPICid;  
    unsigned char clflushLineSize;  
    unsigned char brandIndex;  
    /* hold the flag bits returned in ECX and EDX registers */  
    unsigned ecx;  
    unsigned edx;  
};
```

```
struct cpuInfoLeaf01 *leaf01 = cpuInfo01();
```

Excerpts of the assembly language code that loads data into the structure.

```
cpuInfo01:  
    mov eax, 1    #cpuid information request  
    mov ecx, 0  
    cpuid  
    mov [leaf01ECX + rip], ecx  
    mov [leaf01EDX + rip], edx  
    mov edx, eax    # preserve eax  
    and ax, 0x00F    # mask off the stepping ID  
    mov [stepping + rip], al  
    mov ax, dx  
    and ax, 0x00F0    # mask off the model  
    shr ax, 4  
    mov [model + rip], al  
    mov ax, dx  
    and ax, 0x0F00  
    shr ax, 8  
    mov [family + rip], al  
    #  
    # lots of omitted code  
    #  
    # load the address of the data area
```

```
lea      rax, [cpuInfoLeaf01 + rip]
ret
```

Notice that each piece of return information packed into the EAX register is four bits wide. Those values must be stored in at least an 8 bit wide memory location because the smallest data size available in C—a char—is 8 bits wide. The packed data in the EAX register must be unpacked and stored in memory that corresponds to the C language structure.

There are several ways to pass structures back and forth between C and assembly language. All of these ways require the assembly language code and the C language code to know the internals of the data structure. This means that any changes to the layout of the structure will affect two pieces of code written in two different languages.

Structures can be shared by

- returning a pointer to the memory controlled by the assembly language code.
- passing the assembly language code a pointer to memory controlled by the C language code.

Passing copies of entire structures (as opposed to pointers to structures) is strongly discouraged. Compilers have warning settings (`-Waggregate-return`) that flag attempts to pass entire structures.

When Data Is Bigger Than One Register

Sometimes data is too big to fit in one register. A structure can be a useful way to deal with such situations. Consider the following structure.

```
struct bigNumber {
    unsigned long array[2];
    unsigned char arrayLength;
    unsigned char negativeNumber;
};
```

Every additional element in the array adds 64 bits to the length of the number. The arrayLength field allows the assembly language code to know how big the number is. The negativeNumber field is a flag. Zero means false—the number is positive. Non zero means true—the number is negative. That avoids complications from sign extension.

If element zero in the array is the *least* significant integer, it is easier to write math code in a way that adjusts automatically to a larger array. If element zero in the array is the *most* significant integer, it is easier to write display code.

Exercises

Exercise 1

Which ASCII character is 0x61?

Exercise 2

What is 0b10001011 as a signed byte value?

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?

Exercise 5

Return the Address of an Assembly Language Data Area That Matches a C Structure Definition

Purpose:

To explore several ways to pass a structure between C and assembly language.

Instructions:

Create a directory called structures. Create the following files:

- cpuid.s to hold functions that invoke the CPUID instruction
- main.s to hold the main function that processes any command line parameters and calls the testHarness function in TestHarness.c
- TestHarness.c to hold C language code that calls the functions in cpuid.s
- makefile that builds an executable called TestCPUID from the files above

For now, 'main' will not have to deal with any command line parameters.

In 'cpuid.s' write an assembly language function that:

- interrogates leaf 0 of the CPUID instruction
- creates a structure that holds the highest available leaf of CPUID and the text that Intel returns
- returns the address of that structure to the calling function

In the testHarness C language function:

- create a C language structure definition that matches the layout of the memory area created in the assembly language code
- call the assembly language function and capture the address returned by the assembly language function
- display the information in the structure.

Build and test and debug your work.

Summary:

You have proven that:

- your makefile works
- you can create a structure definition in C that matches the memory layout of memory defined in assembly language code

Exercise 6

Modify main to Accept Command Line Parameters

Purpose:

To enable our code to test more than one structure scenario.

Instructions:

Modify the code in main so that it accepts a single integer parameter.

- For now, the only integer parameter will be zero. That will change in a subsequent exercise.
- Convert the parameter into an integer.
- Report any incorrect parameters and exit from main with an error code of your choice.
- Pass the integer to the testHarness function.

Modify the testHarness code you wrote earlier so that it only calls the test code you wrote if it receives a parameter of zero.

Summary:

Your code now responds to a command line parameter.

Exercise 7

Populate a C Structure from Assembly Code

Purpose:

Write assembly language code that populates a structure defined in C code.

Instructions:

Review the description of the information returned in leaf 0x01 of the CPUID instruction. Notice how that information looks in a C structure, in CPU registers, and in assembly language memory.

Modify `main` so that it accepts an integer parameter of either 0 or 1.

Modify the `testHarness` C language code so that when `main` passes an integer parameter of 1 it will:

- define a `cpuInfoLeaf01` structure that holds the information returned in CPUID leaf 0x01. Notice that the ECX and EDX registers contain bit mapped flags that indicate the presence or absence of language features. Those flags will be captured in a 32 bit unsigned integer.
- create a local instance of that structure within `testHarness`.
- pass the address of that structure to an assembly language function that you will create in `cpuid.s`
- display the contents of the C language structure after calling that assembly language function.

Create an assembly language function in `cpuid.s` called `cpuInfo01` that

- accepts the address of the `cpuInfoLeaf01` structure from the C language `testHarness`
- fills the fields in that structure with the data returned by CPUID leaf 0x01.

You will have to modify the sample code to do this. You may find it interesting to look up the Intel documentation for the information returned in the ECX and the EDX registers by leaf 0x01.

You may also find it interesting to compare the settings of the flags in ECX and EDX with the information returned when you type `/proc/cpuinfo` on a Linux command line.

Summary:

You just populated a C language structure from assembly language code.

Exercise 8

Do Some Simple Math on a Very Big Number

Purpose:

To explore the side effects of doing math on a large numeric structure in assembly language.

Instructions:

Modify the assembly language code in `main` to extend the range of the integer parameter it accepts to be either zero or 1 or 2.

Create an assembly language file called `bigNumber.s` that will:

- add two `bigNumber` structures and store the result in the first structure as described in the instructions below, and
- subtract two `bigNumber` structures and store the result in the first structure as described in the

instructions below.

Modify the `makefile` to assemble and link the `bigNumber.s` file.

Modify the C language code in `testHarness` to so that when `main` passes it an integer value of 2, it will:

- define a `bigNumber` structure as described in the lecture notes.
- create three instances of that `bigNumber` structure as described in the instructions below.
- add two of those `bigNumber` structures as described in the instructions below.
- subtract two of those `bigNumber` structures as described in the instructions below.

In this exercise, element zero fo the array will be the *most* significant integer of the array. Thus, if element zero holds the number 1 and element one holds the number zero, the combined big number is 18,446,744,073,709,551,616. In hexadecimal, that is 0x1 0000 0000. The number 18,446,744,073,709,551,615 would be 0x0 FFFF FFFF.

In `testHarness`, create and initialize three instances of the `bigNumber` structure:

- one that has the value of 2 raised to the 65th power. (0x2 0000 0000)
- another that has the value of 1. (0x0 0000 0001)
- yet another that has the value of 3. (0x0 0000 0003)

Test your assembly language subtraction code by asking the subtraction function to subtract one from 0x2 0000 0000. Display the answer. (It should be 0x1 FFFF FFFF.)

Test your assembly language addition code by asking the addition function to add three to the result you just got back from the addition function. Display the answer. (It should be 0x2 0000 0002.)

Summary:

You have just used a structure to extend the ability of the CPU to do math on large numbers.

Chapter 18. SIMD — Single Instruction, Multiple Data

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.

Intel's SIMD instructions have the ability to perform more than one calculation per instruction. They have the ability to pack more than one data item into a register. The technology has existed in Intel and AMD CPUs for several decades.

Depending on the level of technology that a given CPU supports, there could be as many as three new groups of registers. There will be new instructions.

SIMD instructions presume that there will always be more than one pair of operands that require processing. This implies that SIMD assumes one or more arrays of data.

Introduction

Topics include:

- SIMD specific instructions
- SIMD specific registers
- data organization issues specific to SIMD
- design issues specific to SIMD

SIMD specific instructions include:

- data movement instructions
- arithmetic instructions
- logical instructions
- comparison instructions

Registers

Each of the XMM registers hold 128 bits of data.

- two double precision floating point numbers
- four single precision floating point numbers
- two 64 bit integers
- four 32 bit integers
- eight 16 bit integers
- sixteen 8 bit integers

Each of the YMM registers hold 256 bits of data.

- four double precision floating point numbers
- eight single precision floating point numbers
- four 64 bit integers
- eight 32 bit integers
- sixteen 16 bit integers
- thirty two 8 bit integers

Each of the ZMM registers hold 512 bits of data.

- eight double precision floating point numbers
- sixteen single precision floating point numbers
- eight 64 bit integers
- sixteen 32 bit integers
- thirty two 16 bit integers
- sixty four 8 bit integers

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, we say that the register holds a scalar value. When the register holds more than one piece of the same kind of data, we say that the register holds packed data.

Instructions that manipulate scalar data have the letter "S" as part of their mnemonic. Notice the "s" in every mnemonic below. We have seen these instructions in an earlier chapter.

```
movsd xmm0, [rsi]      # move a scalar double precision floating point number
movsd xmm1, [rdi]      # from memory to an XMM register
mulsd xmm0, xmm1       # multiply the two scalar
                      # double precision floating point numbers
                      # in XMM0 and XMM1
```

Instructions that manipulate packed data have a letter "P" as part of their mnemonic. Notice the "p" in the multiply mnemonic below. We will explain all of these instructions in the pages that follow.

```
# load two doubles from each of two arrays
# that were passed as parameters
movupd    xmm1,  XMMWORD PTR [rsi]
movupd    xmm2,  XMMWORD PTR [rdx]

# multiply TWO double precision floating point numbers in XMM1
# by TWO double precision floating point numbers in XMM2

mulpd     xmm1,  xmm2

# at this point, XMM1 holds TWO double precision floating point numbers
```

When the CPU reads packed data from memory or writes packed data to memory, the CPU sometimes requires that the packed data be aligned on 16 byte boundaries. More on that later.

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 opportunity for improved performance.

Instruction Set

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

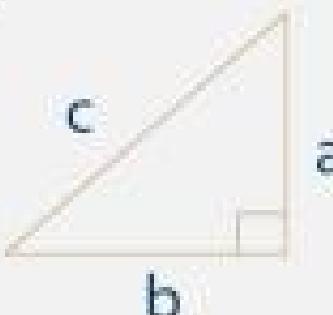
- data movement
- arithmetic
- comparisons
- logic

This is by no means an exhaustive coverage of Intel's many instructions that operate on SIMD data. This sampling is meant to introduce you to

- the categories of SIMD instructions
- the naming conventions used by Intel
- the parameters required by the instructions
- the results that the instructions produce

The instructions are really just common building blocks for calculations that show up in a lot of different formulae. Consider the following formulae.

Pythagoras Theorem:


$$a^2 + b^2 = c^2$$
$$a = \sqrt{c^2 - b^2}$$
$$b = \sqrt{c^2 - a^2}$$
$$c = \sqrt{a^2 + b^2}$$

We will illustrate several of the instructions by calculating the hypotenuse of a right triangle several different ways using different sets of instructions.

Data Movement Instructions

mnemonic	name	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.

mnemonic	name	description
MOVDQA	move aligned double quad word	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 unaligned double quad word	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	replicate double precision floating point values	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]
....      # do something with all four floats

# movaps requires data aligned on a 16 byte boundary.
.align 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. You have no control over the alignment of that data. 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 [rdx]
....  # do something with the two doubles
# move the pointers in RDX and RSI to the next pair of doubles in each array
    add     rdx,  SizeOfXmmRegister
    add     rsi,  SizeOfXmmRegister
....  # loop
```

NOTE

Structures that are aligned on 16 byte boundaries might well have members that are not aligned.

In this example, think of RSI and RDX 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 [rdx]
.... # do something with the eight short unsigned ints
# move the pointers in RDX and RSI to the next eight short unsigned integers in each
array
add      rdx,  SizeOfXmmRegister
add      rsi,  SizeOfXmmRegister
.... # loop
```

NOTE

This version of the mov instruction does not care if it moves two unsigned longs, four signed ints, eight short ints or 16 chars. It moves 128 bits. What those bits mean is up to you. Other instructions (ie arithmetic instructions) *will* care. MOVDQA and MOVDQU will not.

Comparisons

mnemonic	CPUID feature	name	description
MAXPD or MAXPS	SSE2	maximum of packed single or double precision floating point	Depending on the instruction, each packed operand contains either two double precision floating point numbers or four single precision numbers.
MINPD or MINPS	SSE2	minimum of packed single or double precision floating point	Depending on the instruction, each packed operand contains either two double precision floating point numbers or four single precision numbers.
CMPPD	SSE	compare packed double precision floating point values	Compare packed double precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of an immediate 8 bit value as a comparison predicate.
CMPPS	SSE	compare packed single precision floating point values	Compare packed single precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of an immediate 8 bit value as a comparison predicate.

mnemonic	CPUID feature	name	description
PCMPEQB or PCMPEQW or PCMPEQD or PCMPEQQ	SSE2	compare packed bytes or words or double words or quad words for equality	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 location. Sets corresponding data element(s) in destination to all 1. Sets other data element(s) to all 0. See also PAND
PCMPGTB or PCMPGTW or PCMPGTD or PCMPGTQ	SSE2	compare packed bytes or words or double words or quad words for greater than	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 location. Sets corresponding data element(s) in destination to all 1 if destination is greater than source. Sets other data element(s) to all 0. See also PAND

Arithmetic

mnemonic	name	description
ADDPD or SUBPD	add/subtract packed double precision floating-point	add or subtract packed double precision floating-point operands. Each packed operand contains two double precision floating point numbers.
ADDPS or SUBPS	add/subtract packed single precision floating-point	add or subtract packed single precision floating-point operands. Each packed operand contains four single precision floating point numbers.
PADDB or PADDW or PADDD or PADDQ	add packed integers	Add packed unsigned integers. Depending on the instructions, the integers will be 8 bits or 16 bits or 32 bits or 64 bits. Any carry is ignored.
PADDSS or PADDSW	add packed signed integers with signed saturation	Add packed signed integers with signed saturation. Depending on the instruction, the integers will be 8 bits or 16 bits. Carry flag is ignored. Range is limited to upper or lower bounds of the data type.
PSUBSB or PSUBSW	subtract packed signed integers with signed saturation	Depending on the instructions, the integers will be 8 bits or 16 bits. Results are limited — saturated — to the upper or lower bound of the data type.
PSUBUSB or PSUBUSW	subtract packed unsigned integers with unsigned saturation	Depending on the instructions, the integers will be 8 bits or 16 bits. Results are limited — saturated — to the upper or lower bound of the data type.
HADDPD or HADDP S	horizontally add adjacent packed floating point numbers	Depending on the instructions, horizontally add either single or double precision floats. Source one's results go in the low position. Source two's results go in the high position.

mnemonic	name	description
HSUBPD or HSUBPS	horizontally subtract adjacent packed floating point numbers	Depending on the instructions, horizontally subtract either single or double precision floats. Source one's results go in the low position. Source two's results go in the high position.
MULPD or MULPS	multiply packed single or double precision floating point	Depending on the instruction, each packed operand contains either two double precision floating point numbers or four single precision numbers.
DIVPD or DIVPS	divide packed single or double precision floating point	Depending on the instruction, each packed operand contains either two double precision floating point numbers or four single precision numbers.
SQRTPD or SQRTPS	square root of packed single or double precision floating point	Depending on the instruction, each packed operand contains either two double precision floating point numbers or four single precision numbers.

Listing 26. Illustrate 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.

Calculate the hypoteneuse of two different triangles at the same time. Substitute single precision floating point numbers in order to process four different traingles at the same time.

Listing 27. 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 [rdx] # 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 hypoteneuse of two different triangles
```

Combinations

mnemonic	name	description
ADDSUBPD or ADDSUBPS	add/subtract of packed single or double precision floating point	Add the odd numbered single or double precision floating point numbers. Subtract the even numbered numbers.
PMADDWD	multiply and add packed integers	Perform packed integer multiplication followed by addition of the result. Also called "dot product".

PMADDWD

The raw data must consist of whole numbers that fit in a 32 bit integer. PMADDWD performs a packed multiplication of 16 bit integers. The product of each multiplication becomes a 32 bit integer. Adjacent 32 bit integers are added.

SRC	X3	X2	X1	X0
DEST	Y3	Y2	Y1	Y0
TEMP	X3 * Y3	X2 * Y2	X1 * Y1	X0 * Y0
DEST	(X3*Y3) + (X2*Y2)	(X1*Y1) + (X0*Y0)		

Typical usage for dot product calculations would be matrix math. There are, however, additional ways to use this instruction.

Calculate the hypoteneuse of a right triangle using the Dot Product instruction.

Listing 28. Example of Dot Product

```
movdqu      xmm1, XMMWORD PTR [rsi] # load two sides of four different triangles
movdqa      xmm2, xmm1                  # make a copy of xmm1
PMADDWD    xmm1, xmm2    # 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. The second instruction makes XMM2 a copy of XMM1. A packed multiply of XMM1 and XMM2 would square each measurement.

PMADDWD performs a packed multiply of XMM1 and XMM2. When 16 bit integers are multiplied, the result could be as big as 32 bits. PMADDWD changes eight 16 bit registers into four 32 bit registers and stores the sum of the packed multiplications in them. Effectively, that means that each of the 32 bit registers holds the square of the hypoteneuse for four different triangles.

Logical

mnemonic	name	description
POR	bitwise logical OR	Bitwise OR of a pair of XMM or YMM or ZMM registers
PAND	bitwise logical AND	Bitwise AND of a pair of XMM or YMM or ZMM registers
PSLLW or PSLLD or PSLLQ or PSLLDQ	shift packed data left logical	Shift 16 or 32 or 64 or 128 bit packed data items left. Fill with zero.
PSRLW or PSRLD or PSRLQ or PSRLDQ	shift packed data right logical	Shift 16 or 32 or 64 or 128 bit packed data items right. Fill with zero
PSRAW or PSRAD or PSRAQ	shift packed data right logical	Shift 16 or 32 or 64 or 128 bit packed data items right. Fill with whatever value is in high bit. Similar to rotate right in the general purpose registers.

Conversion

mnemonic	name	description
CVTDQ2PD or CVTDQ2PS	convert packed doubleword integers into packed floats	Depending on the instruction, convert a pair of 32 bit integers into a pair of either single precision or double precision floats.
CVTPD2PI or CVTPS2PI	convert packed floats to packed dword integers with rounding	Depending on the instruction, convert packed single or double precision floats to 32 bit signed integers. Results are rounded.
CVTPD2DQ or CVTPS2DQ	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.

Listing 29. Example of floating point conversion and Packed Square Root

```
movdqu    xmm1, XMMWORD PTR [rsi]    # load two sides of four different triangles
movdqa    xmm2, xmm1                  # make a copy of xmm1
PMADDWD  xmm1, xmm2      # A squared + B squared = four 32 bit C squared results in
XMM1
CVTDQ2PS xmm1, xmm1      # they are now four single precision floats
SQRTPS   xmm1, xmm1      # four hypoteneues for four different
triangles
```

Compare this listing with the listing that did not use the dot product instruction.

Reciprocal Instructions

mnemonic	name	description
RCPPS	compute recipricals 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	compute recipricals of the square roots 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.

Single precision floating point numbers only. No integers. To compute recipricals of double precision floats requires AVX512VL or AVX512F.

Design Considerations

The ability to do multiple arithmetic operations in parallel has side effects on software design. One side effect is an inability to depend on flags to interpret results. If four arithmetic operations take place at once, what does the Carry Flag mean? Or the Equals Flag?

Sometimes, there are opportunities for parallel processing within what is otherwise a serial function. Consider the assembly language for the quadratic equation

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The *plus and minus* symbol in the formula offers an opportunity for parallel processing.

Listing 30. From the test harness driver

```
extern void quadraticSIMD( double , double , double , double * );
...
/* x**2 - 4x + 4
   a = 1, b = -4, c = 4 */
double roots[2];
quadraticSIMD( 1.0, -4.0, 4.0, roots );
```

Plan your use of registers.

Listing 31. From the comments in the assembly language code

```
/* void quadraticSIMD( double , double , double , double * );
See quadratic formula above for meaning of a,b,c
Input registers:
  XMM0 -- a
  XMM1 -- b
  XMM2 -- c
  RDI -- the address where an array of at least two doubles begins
Output registers:
  none
Scratch registers:
  XMM7 -- two copies of -B
  XMM6 -- one copy of B squared
  XMM2 -- one copy of 4 * A * C
  XMM0 -- two copies of 2 * A      */
```

Look for any constants in the formula. Consider whether those constants need to be aligned on a 16 byte boundary

Listing 32. Constants in the assembly language code

```
.align 16
```

```

Zero: .double 0
Two: .double 2
Four: .double 4

```

Explore possibilities for parallel processing. The square root calculation in the formula gets used twice. Once it gets added to $-b$ and once it gets subtracted from $-b$. This presents an opportunity to use packed data with the addsubpd instruction. Consider this excerpt from the source code.

<code>movsd</code>	<code>xmm7, Zero[rip]</code>	
<code>subsd</code>	<code>xmm7, xmm1</code>	# XMM7 holds one copy of $-B$
<code>movddup</code>	<code>xmm7, xmm7</code>	# XMM7 now holds two copies of $-B$
<code>subsd</code>	<code>xmm6, xmm2</code>	# XMM6 holds one copy of $(B^2 - 4AC)$
<code>sqrtsd</code>	<code>xmm6, xmm6</code>	# XMM6 holds one copy of $\sqrt{(B^2 - 4AC)}$
<code>movddup</code>	<code>xmm6, xmm6</code>	# XMM6 now holds two copies of $\sqrt{}$
<code>addsubpd</code>	<code>xmm7, xmm6</code>	# XMM7 holds $(-B + \sqrt{})$ and $(-B - \sqrt{})$
<code>divpd</code>	<code>xmm7, xmm0</code>	# XMM7 holds two answers

Parallel processing starts with `movddup xmm6, xmm6`. After that instruction executes, XMM6 holds two copies of the same number. The number in the high half of XMM6 will be added to the number in the high half of XMM7. The number in the low half of XMM6 will be subtracted from the number in the low half of XMM7. The two numbers in XMM7 get divided by the two numbers in XMM0.

The complete listing follows.

Listing 33. Complete listing for quadraticSIMD

```

quadraticSIMD:
# must be done before multiplying A * 2
mulsd      xmm2, xmm0      # XMM2 holds one copy of A * C
mulsd      xmm2, Four[rip] # XMM2 holds one copy of 4 * A * C
# it is now safe to multiply A * 2
mulsd      xmm0, Two[rip]  # XMM0 holds one copy of 2a
movddup   xmm0, xmm0      # XMM0 holds two copies of 2a
movsd      xmm6, xmm1
mulsd      xmm6, xmm1      # XMM6 holds one copy of B squared
movsd      xmm7, Zero[rip]
subsd      xmm7, xmm1      # XMM7 holds one copy of -B
movddup   xmm7, xmm7      # XMM7 holds two copies of -B
subsd      xmm6, xmm2      # XMM6 holds one copy of (B squared - 4AC)
sqrtsd    xmm6, xmm6      # XMM6 holds one copy of sqrt((B squared - 4AC))
movddup   xmm6, xmm6      # XMM6 holds two copies of sqrt
addsubpd  xmm7, xmm6      # XMM7 holds (-B + sqrt) and (-B - sqrt)
divpd     xmm7, xmm0      # XMM7 holds two roots
movupd    [rdi], xmm7     # write the answer to the array
ret

```

Data Organization

Structures of arrays instead of arrays of structures. Business programming deals with arrays of structures—arrays of personnel structures, arrays of inventory items, etc. The various SIMD instructions want to deal with arrays of individual data items. That might well require a layer of code to extract the data from structures and create arrays.

CPU instructions that process packed data consume data in some multiple of a power of two. Typically, the instruction will process two data items or four data items or eight data items at a time. The size and type of the data you choose to process will affect the amount of parallel processing that can be done.

Single precision floating point numbers (as opposed to double precision numbers) allow twice as much data to be processed in parallel. Applications that do not need more than seven significant digits of precision can benefit from the ability to process twice as many data items per instruction.

Applications that do not require decimal points can benefit from using integers as opposed to floating point numbers. If the range of numbers needed by the application is well known, additional opportunities for optimizations arise.

16 bit integers are potentially twice as fast to process as 32 bit integers. 8 bit integers are potentially double that. The tradeoff is the range of the numbers.

Instructions that perform saturated math offer other design possibilities. Saturation arithmetic has numerous practical advantages. The result of saturated arithmetic is often as numerically close to the true answer. For 8-bit binary signed arithmetic, when the correct answer is 130, it is considerably less surprising to get an answer of 127 from saturating arithmetic than to get an answer of -126 from modular arithmetic. Intel instructions that guarantee to limit the results of math to certain well known upper and lower limits have the word "saturated" as part of their descriptions.

Saturation math is only available for integer data types. The points of saturation depend on the data type. Unsigned bytes saturate at 255 and 0. Signed bytes saturate at +127 and -128. Analogous limits apply for words, double words and quad words.

Listing 34. Example of unsigned subtraction of integers with saturation

```
lea      r8, firstIntArray[rip]
lea      r9, secondIntArray[rip]
...
movdqa  xmm1, [r8]
movdqa  xmm2, [r9]
psubusw xmm1, xmm2
movdqu  XMMWORD PTR [rdi], xmm1
...
# data area
.align 16
firstIntArray: .int  1,2,3,4,5,6,7,8,9,66,22,22,66,22,22,66,22,22,66,22,22
.align 16
```

```
secondIntArray: .int  9,8,7,6,5,4,3,2,1,11,22,22,11,22,22,11,22,22  
# - - - - Results - - - - - - - - - - - - - - - - - - - - - - - - - -  
0,0,0,0,0,2,4,6,...
```

Data sets that are not a multiple of the data width of the packed instructions will need special handling at the end of the data set to process the remaining data items.

Exercises

Exercise 1

Modify the Test Harness

Purpose:

Create a test environment for the next few exercises.

Instructions:

Copy the entire floatGeometry folder that you created for an earlier exercise to a directory called simdGeometry.

Rebuild everything (`make -B`) and confirm that the code correctly calculates the area of one triangle and the hypoteneuse of one right triangle.

Modify the `testHarness` function. (See the code below)

- Create three arrays—two to hold measurements of each side of triangles and the other to hold answers. (See code below)
- Create a define to establish the sizes of the arrays. (See code below)
- Create loops to initialize the arrays with random test data. (See code below)
- Create loops to execute the functions that you wrote in the earlier exercise. (See code below)

Build and execute the code and prove that the answers are correct.

Listing 35. Revised code for the `testHarness()` functions

```
#define SidesArraySize ( 2 )

int testHarness( void ) {
    int resultCode = 0;           unsigned i = 0;
    unsigned range = 100;
    double answer[ SidesArraySize ];
    double side01[ SidesArraySize ];
    double side02[ SidesArraySize ];

    /* Initialize the sides array with random numbers */
    for( i = 0; i < sizeof( side01 ) / sizeof( side01[0] ); i++ ) {
        side01[i] = 1 + getRandomNumber( range );
        side02[i] = 1 + getRandomNumber( range );
        answer[i] = -75.0;
    }

    for( i = 0; i < sizeof( side01 ) / sizeof( side01[0] ); i++ ) {
        answer[i] = areaTriangle( side01[i], side02[i] );
    }
}
```

```

}

for( i = 0; i < sizeof( answer) / sizeof( answer[0] ); i++ ) {
    printf( "Triangle height: %5.2f, width: %5.2f, area: %7.3f\n", side01[i],
            side02[i], answer[i] );
}

for( i = 0; i < sizeof( answer) / sizeof( answer[0] ); i++ ) {
    answer[i] = hypoteneuseRightTriangle( side01[i], side02[i] );
}

for( i = 0; i < sizeof( answer) / sizeof( answer[0] ); i++ ) {
    printf( "Triangle side A: %5.2f, side B: %5.2f, hypoteneuse: %7.3f\n",
            side01[i], side02[i], answer[i] );
}

return resultCode;
} /* end of testHarness() */

```

With this trivial modification, test data will come from an array of random numbers. The length of the array will be determined by a `#define` at the beginning of the file. This will become important for executing and testing the code you will write in the next few exercises.

Make sure this code builds without errors or warnings. Make sure it executes correctly.

Summary:

You now have a test harness that you can use to drive the rest of the exercises. You will also be able to compare the code you wrote earlier with the code that executes multiple instructions in parallel.

Exercise 2

Calculate a Small Batch of Areas of Triangles

Purpose:

Provide guided practice writing the code to make two calculations execute in parallel using Intel's packed instructions.

Instructions:

Write one additional assembly language function in `geometry.s` (See sample below.)

Call that function from the `testHarness()` function. (See code fragment below.)

You will need these extern declarations in `TestHarness.c`

Listing 36. Revised list of extern declarations and defines

```
extern double areaTriangle( double, double );
extern double hypoteneuseRightTriangle( double, double );

extern void areaTriangleSIMD( double * , double *, double *, unsigned );

#define SidesArraySize ( 2 )
```

The new extern declaration is for `areaTriangleSIMD`. This function will use packed instructions to process two triangles each time the function is called.

In order to test the assembly language function that you are about to write, call them from the end of the `testHarness` function. This means that you will feed the exact same data to the geometry functions you wrote earlier and to the functions that you are about to write today. That will allow you to verify the accuracy of the assembly language code that you write.

Listing 37. The end of testHarness() function should look like this

```
/* Reinitialize the answer array with dummy numbers */
for( i = 0; i < sizeof( side01 ) / sizeof( side01[0] ); i++ ) {
    answer[i] = -222.0;
}
puts( "\nCalculations using packed registers" );

areaTriangleSIMD( answer, side01, side02, sizeof( answer ) / sizeof( answer[0] ) );
for( i = 0; i < sizeof( answer ) / sizeof( answer[0] ); i++ ) {
    printf( "Triangle height: %f, width: %f, SIMD area: %.3f\n", side01[i],
    side02[i], answer[i] );
}

return resultCode;
```

```
 } /* end testHarness()      -      - */
```

In the `geometry.s` file, add `globl` declarations for the new assembly language function.

Listing 38. New Globals and Equates for geometry.s

```
.globl areaTriangleSIMD  
  
.equ DoublesPerXMMRegister, 2    #number of 32 bit ints that fit in an XMM  
register  
.equ SizeOfXmmRegister,        16   #bytes in XMM register
```

Write an assembly language function that calculates the area of a triangle using SIMD instructions that operate on packed data. In this exercise, you will write code that will calculate areas for two triangles at each time the function is called. In a future exercise, you will modify this code to process larger data sets.

The formula for the area of the triangle is: $\text{Area} = (\text{height} * \text{base}) / 2$

Call the function `areaTriangleSIMD`

Each time `areaTriangleSIMD` is called

- RDI will hold the address where the destination array begins. This is where the answer will be stored. Each time this function is called, it will add two areas to this array.
- RSI will hold the address where the array that holds heights of a triangle begins. Each time this function is called, it will load two numbers from this array.
- RDX will hold the address where the array that holds measurements of the base of a triangle begins. Each time this function is called, it will load two numbers from this array.
- RCX will hold the number of elements in the arrays. In this exercise, this value will not be used.

Load two packed doubles (heights) from the height array into the XMM1 array. You can do this with one instruction that operates on *unaligned* memory areas. Look through the course notes if you need an example.

Load two packed doubles (bases) from the height array into the XMM2 array. You can do this with one instruction that operates on *unaligned* memory areas. Look through the course notes if you need an example.

Do a packed multiplication of XMM1 by XMM2. XMM1 should be the destination of this operation. Look through the course notes if you need an example.

Move the constant at Two: into both the high half and the low half of the XMM2 register. Look at the various move instructions for an instruction that moves and duplicates a 64 bit memory value. Look through the course notes if you need an example.

Do a packed double precision floating point divide of XMM1 by XMM2. The destination should be the XMM1 register.

Move all 128 bits of the XMM1 register to the two doubles at the unaligned memory address in the RDI register.

Remember to return.

Build and execute the code you just wrote. Your new functions should produce the same results as the functions you wrote in an earlier exercise.

Once your new functions work, compare the code that does the actual calculation. Ignore the code that reads from the input arrays and writes to the output array. Just compare the code that computes the area of the triangle in both functions. How many instructions in each function? If the instruction count is similar, think about how many triangles you process per function invocation in the function you wrote earlier in your training vs the function you just wrote.

Summary:

You have a side by side comparison of code that uses packed data and code that does not.

Exercise 3

Calculate a Small Batch of Hypoteneueses of Triangles

Purpose:

Use the packed instructions with less guidance.

Instructions:

Write one additional assembly language function in `geometry.s` (See sample below.)

Call that function from the `testHarness()` function. (See code fragment below.)

You will need these extern declarations in `TestHarness.c`

Listing 39. Revised list of additional extern declarations

```
extern void areaTriangleSIMD( double * , double *, double *, unsigned );
extern void hypoteneuseRightTriangleSIMD( double *, double *, double *, unsigned );
```

The new `extern` declaration is for `hypoteneuseRightTriangleSIMD`. This function will use packed instructions to process two triangles each time it is called.

In order to test the assembly language function that you are about to write, call it from the end of the `testHarness` function. This means that you will feed the exact same data to the `geometry` functions you wrote earlier and to the function that you are about to write today. That will allow you to verify the accuracy of the assembly language code that you write.

Listing 40. The revised end of `testHarness()` function should look like this

```
/* Reinitialize the answer array again with dummy numbers */
for( i = 0; i < sizeof( side01 ) / sizeof( side01[0] ); i++ ) {
    answer[i] = -222.0;
}

hypoteneuseRightTriangleSIMD( answer, side01, side02, sizeof( answer ) / sizeof(
answer[0]) );
for( i = 0; i < sizeof( answer ) / sizeof( answer[0] ); i++ ) {
    printf( "Triangle side A: %5.2f, side B: %5.2f, SIMD hypoteneuse:
%7.3f\n", side01[i], side02[i], answer[i] );
}

return resultCode;
} /* end testHarness() */
```

In the `geometry.s` file, add the following `globl` declaration.

Listing 41. New Global for geometry.s

```
.globl hypoteneuseRightTriangleSIMD
```

Hypoteneuse of the triangle = the square root of (side01 squared + side02 squared)

Write an assembly language function that calculates the hypoteneuse of a right triangle.

- Call the function `hypoteneuseRightTriangleSIMD`
- Use the same parameter list and input registers as the `areaTriangleSIMD` used.
- Used instructions that operate on packed data so that you process two triangles each time the function is called.
- Just get the function to work on two data sets. In the next exercise, we will make the function work on arbitrary length arrays of data.

Build and test your work.

Summary:

The semantics of the SIMD instructions that use packed data imply arrays of data rather than singleton data.

Exercise 4

Make the Two New Functions Work with Arbitrary Length Arrays

Purpose:

To show that SIMD instructions that operate on packed data want to work on arrays.

Instructions:

Add a loop to each of the two functions that you just wrote so that they can process any even length array. The number of elements in each array will be passed to you in the RCX register.

Some things you might want to consider:

- What to do if a caller passes you a zero length array
- What to do if a caller passes you an odd length array
- That each pass through the loop will process two data sets

Immediately on entry, check the value in RCX. If that value is less than 2, there is nothing to do.

Given that each pass through the loop will process data for two triangles, shift RCX right by one bit to divide the value in half. We will not deal with the special case of a caller passing an odd number of items.

In an earlier exercise, you added an equate to geometry.s

```
.equ SizeOfXmmRegister, 16 #bytes in XMM register
```

If that equate is missing, add it now.

That amount is the number of bytes to add to registers RDI, RSI and RDX each pass through the loop. Those three registers really are just like C pointers.

Keep looping until RCX hits zero.

Build, execute and debug your work.

Summary:

You just used packed instructions in their most common way. These instructions work very naturally with arrays of data that have some exact multiple of the number of items the instructions pack.

Exercise 5

Write Two New Functions That Use Single Precision Instead of Double Precision Floats.

Purpose:

To learn how different data types affect the design of SIMD code that uses packed data.

Instructions:

The decision to use single precision floating point numbers will affect:

- the test harness code that tests the two functions.
- the `extern` declarations in the test code.
- the number of data items that fit into each XMM register.
- the instructions used to load input data.
- the instructions that perform packed calculations.

Write two new functions:

- `areaTriangleFloatSIMD`
- `hypoteneuseRightTriangleFloatSIMD`

Use the same interface registers as before. Write test code in the `testHarness()` function to verify the correctness of the two new functions.

Write the two new functions.

Document your functions.

Build, execute and debug your work.

Summary:

You have just explored the kinds of changes necessary for different data types.

Chapter 19. Primitive Tools for Examining Data Files

Motivation

Some systems have a graphic user interface. Others don't. Sometimes operational issues prevent you from using a graphic user interface even if one exists. If you have to work on the command line, there are powerful tools at your disposal.

Experienced users who work in typical environments can look at the suffix of a file name and know what kind of file they have. In other environments, a user does not have the luxury of assuming that a file name reveals file characteristics. In those specialized environments, it is useful to know that there are tools that are:

- informative and
- safe to use

Magic Numbers

Magic Numbers are metadata embeded in the file itself. A magic number identified the kind of data a file holds. It can also tell if a file is meant to be executed or not. Many magic numbers appear at the start of a file. Others appear at some offset into the file. https://en.wikipedia.org/wiki/List_of_file_signatures

Some common magic numbers:

- ffd8ffe1 = jpg
- 6d70 3432 6d70 3431 = mp4
- PK = zip or a file format derived from zip
- .ELF = Linux executable
- MZ = Microsoft executable
- 7B 5C 72 74 66 31 = Microsoft Rich Text Format

hexdump and xxd

Both utilities display file contents in hex. Both utilities are extremely useful:

- to view binary files
- to discover whether an unknown file is binary or text
- to attempt to discover the Magic Number of an unknown file

Sometimes the decision to use hexdump vs xxd boil down to personal preference. Other times, there are practical reasons for choosing one over the other. For example, vim can use xxd as a makeshift hex editor via :%!xxd and :%!xxd -r

`hexdump` displays file contents in hex. `hexdump [options] [-e format_string] inputFile` Options are case sensitive. Some commonly used options include:

- `-c` One-byte character display. Display the input offset in hexadecimal, followed by sixteen space-separated, three column, space-filled, characters of input data per line.
- `-C` Canonical hex+ASCII display. Display the input offset in hexadecimal, followed by sixteen space-separated, two column, hexadecimal bytes, followed by the same sixteen bytes in `%_p` format enclosed in | characters
- `-x` Two-byte hexadecimal display. Display the input offset in hexadecimal, followed by eight, space separated, four column, zero-filled, two-byte quantities of input data, in hexadecimal, per line
- `-e` Specify a format string to be used for displaying data.

`xxd` is another application that displays file contents in hex. `xxd [options] [infile [outfile]]`
Options are case sensitive. Some commonly used options include:

- `-l length` Stop after processing length bytes
- `-r` Reverse the process. The input file contains edits used to modify the output file

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. Typical strategies for hiding strings from the `strings` utility include:

- Splitting the string up into non-adjacent cells in memory
- rotating the bits of each character of the string
- bitwise-xor on the characters

Options are case sensitive. Some commonly used options include:

- `-d` Only scan the initialized loaded data sections of the file. There are potential security issues with this option. See the man page for details.
- `-n length` Do not report strings less than this length
- `-w` Include all whitespace, not just tab and space characters

files

The `files` program will reliably describe not just files, but also devices.

Exercises

Exercise 1

Purpose: To provide practice using command line tools to analyze files that

- are not expected to execute.
- have unknown content.

Instructions: You can learn a lot about a file from the simplest of command line tools - ls

Take a directory listing of this directory using the long form of ls

- ls -l
- ll (if your operating system supports this command)

Do any files appear to be duplicates of each other? If so, write their names and file sizes here

Checksums are a quick and common way to determine if two pieces of data are identical. Use your favorite checksum program to produce a hash of every file that you think might be a duplicate. `sha56sum` and `sha512sum` are commonly available checksum utilities. Typical usage: `sha512sum [name of file]`

Feel free to use any other checksum program if you wish. What files are duplicates of each other?

Now, let's discover what kind of files you have. `head` `tail` `cat` will provide the information needed to separate text files from binary files. Use `man` if necessary to learn how to use each of these tools. Identify any text files and write their names and file lengths and checksums

Now use `diff` to compare pairs of text files and identify the location of any differences. Record those differences.

The command line tool strings can be very useful for analyzing binary files. Use the man utility if necessary to learn how to use the tool. Sometimes, just reading the strings inside a file will reveal the file type. Very often, there is interesting information at the beginning of a file or at the end of a file. Pipe the output of strings into head or tail to find stereotypical beginnings or endings. Sometimes, it is necessary to look at as much as 30 lines of text to see what you need. If there is enough information to know the file type with a certainty, go ahead and rename the file with the correct extension.

Sometimes, it is necessary to look a little deeper. The command line tools hexdump and xxd allow you to examine binary files. Use the man utility to learn how to use both tools. Many files start with a "magic number" that identifies the file type. That "magic number" is very often found in the first five to 20 bytes of the file. Pipe the output of either hexdump or xxd into head. Use google to search for "magic numbers". Rename the file to have the correct suffix.

Read the `man` description for the `file` utility.

```
man file
```

Use the `file` utility on every file you examined in this exercise. Use the `file` utility to examine some operating system devices.

- Type `ls -a /dev/` to discover some devices on the lab environment.
- Pick an interesting "folder" under `/dev`
- Do an `ls -a` of that "folder"
- Explore what `file` reveals about that "folder"

Let's review:

`ls`, the simplest of tools, can be used in the simplest of ways to learn some simple details about files.

Checksums identify which files are identicle twins.

`head`, `tail` and `cat` can help differentiate between text files and binary files.

`diff` can identify files that had essentially the same information and reveal where those files might have been modified.

`hexdump` and `xxd` can be used to find "magic numbers" in binary files.

`file` will reveal a lot about files and pseudo files

Chapter 20. Primitive Tools for Examining Executable Files

Introduction

Often it is necessary to examine executable code. Several tools exist.

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. Options are case sensitive. (See the man page for details.) Some commonly used options include:

- **-a** Show debug symbols (if any)
- **-C** Demangle names (very useful when examining C++ executables)
- **-D** Display dynamic names
- **-l** Show line numbers from the source file

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. Options are case sensitive. (See the man page for details.) Some commonly used options include:

- -d Disassemble sections of the object file that are expected to contain code. By default, this option uses AT&T syntax.
- -D Disassemble everything, not just sections that are expected to contain code. By default, this option uses AT&T syntax.
- -M "x86-64,intel" This option specifies the machine architecture to use when examining the object file. In our case, we are using the 64 bit architecture of the x86 family of CPUs. intel tells objdump to use Intel syntax rather than the default AT&T syntax for the disassembled code.
- -i List all the architectures that objdump understands.
- -l Print source code line numbers if this debugging information is included in the object file.
- -x List all headers.

objdump can provide the list of shared libraries.

```
objdump -x binary | grep NEEDED
```

readelf

readelf displays information about one or more ELF format object files. The options control what particular information to display. Options are case sensitive. (See the man page for details.) Some commonly used options include:

- `-W` Don't break output lines to fit into 80 columns.
- `-x <number or name>` Displays the contents of the indicated section as hexadecimal bytes. A number identifies a particular section by index in the section table. Any other string identifies all sections with that name in the object file.
- `--relocated-dump=<number or name>` Displays the contents of the indicated section as a hexadecimal bytes. A number identifies a particular section by index in the section table; any other string identifies all sections with that name in the object file. The contents of the section will be relocated before they are displayed.
- `--string-dump=<number or name>` Displays the contents of the indicated section as printable strings. A number identifies a particular section by index in the section table. Any other string identifies all sections with that name in the object file.
- `-a` Equivalent to specifying `--file-header`, `--program-headers`, `--sections`, `--symbols`, `--relocs`, `--dynamic`, `--notes`, `--version-info`, `--arch-specific`, `--unwind`, `--section-groups` and `--histogram`.
- `--file-header` Displays the information contained in the ELF header at the start of the file.
- `--program-headers` Displays the information contained in the file's segment headers, if it has any.
- `--sections` Displays the information contained in the file's section headers, if it has any.
- `--symbols` Displays the entries in symbol table section of the file, if it has one.
- `--headers` Display all the headers in the file. Equivalent to `--file-header` and `--program-headers` and `--sections`.
- `--relocs` Displays the contents of the file's relocation section, if it has one.
- `--dynamic` Displays the contents of the file's dynamic section, if it has one.
- `--notes` Displays the contents of the NOTE segments and/or sections, if any.
- `--version-info` Displays the contents of the version sections in the file, if they exist.
- `--arch-specific` Displays architecture-specific information in the file, if there is any.
- `--unwind` Displays the contents of the file's unwind section, if it has one.
- `--section-groups`
- `--histogram` Display a histogram of bucket list lengths when displaying the contents of the symbol tables.

To see other sets of registers available, issue the commands `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. These registers usually shadow 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, it 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 -fasm=intel -fno-asynchronous-unwind-tables

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

# -mno-target-align to remove align from most instructions (not loops)
# -fverbose-asm
# -finithibit-size-directive
```

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.

-01

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.

Instruction Encoding

Although tools such as Ghidra, gdb, objdump, and others are capable of disassembly, one might need to manually decode HEX into opcodes, or write a tool to perform such decoding.

An assembly instruction must be encoded into binary bytes.

An instruction can be up to 15 bytes long.

Table 16. Number of bytes per instruction section

Instruction Prefix	SIMD Prefix	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
0-4	0-1	0-1	1-3	0-1	0-1	0-4	0-4

Legacy Prefix 1

0xF0

lock: Only works on add, sub, adc, sbb, dec, inc, neg, not, and, or, xor, xadd, xchg, cmpxchg, cmpxchg8b, cmpxchg16b, btc, btr, bts \\

0xF2

repne: Only with cmpsX, scasX \\

0xF3

repz: Only with cmpsX, scasX \\

0xF3

rep: Only with ins, lods, movs, outs, stos

Legacy Prefix 2

0x2E

CS segment

0x26

ES segment

0x3E

DS segment

0x36

SS segment

0x64

FS segment

0x65

GS segment

0x2E

Branch not taken

0x3E

Branch taken

Legacy Prefix 3

0x66

Operand-size override prefix

Legacy Prefix 4

0x67

Address-size override prefix

SIMD Prefix

0x66: \| 0xF2: \| 0xF3:

REX Prefix

7	6	5	4	3	2	1	0
0	1	0	0	W	R	X	B

The REX prefix byte is a byte that allows for 64-bit registers to be used. Specific bits of the REX byte are set that correspond to registers (or values) in the arguments of the instruction.

These bytes are usually referenced as REX.B, REX.W, REX.RB, etc. If the REX byte is present, but has none of its control bits set, it is referred to as REX.0.

Each bit (W, R, X, B) affect registers in certain parts of the instruction. They function like a shift key, selecting a different set of registers:

W

Specifies 64-bit operands for immediates

R

Selects upper registers for ModR/M register.

X

Selects upper registers for SIB index.

B

Selects upper registers for ModR/M R/M field or SIB base.

REX.reg	8-bit	16-bit	32-bit	64-bit	x87	MMX	XMM	YMM
0.000	al	ax	eax	rax	ST0	MMX0	XMM0	YMM0
0.001	cl	ecx	rcx	ST1	MMX1	XMM1	YMM1	
0.010	dl	dx	edx	rdx	ST2	MMX2	XMM2	YMM2
0.011	bl	bx	ebx	rbx	ST3	MMX3	XMM3	YMM3
0.100	spl	sp	esp	rsp	ST4	MMX4	XMM4	YMM4
0.101	bpl	bp	ebp	rbp	ST5	MMX5	XMM5	YMM5
0.110	sil	si	esi	rsi	ST6	MMX6	XMM6	YMM6
0.111	dil	di	edi	rdi	ST7	MMX7	XMM7	YMM7
1.000	r8l	r8w	r8d	r8		MMX0	XMM8	YMM8
1.001	r9l	r9w	r9d	r9		MMX1	XMM9	YMM9
1.010	r10l	r10w	r10d	r10		MMX2	XMM10	YMM10
1.011	r11l	r11w	r11d	r11		MMX3	XMM11	YMM11
1.100	r12l	r12w	r12d	r12		MMX4	XMM12	YMM12
1.101	r13l	r13w	r13d	r13		MMX5	XMM13	YMM13
1.110	r14l	r14w	r14d	r14		MMX6	XMM14	YMM14
1.111	r15l	r15w	r15d	r15		MMX7	XMM15	YMM15

If there is no REX byte whatsoever, then the one-byte pointer registers are unreachable. Instead, those bit-patterns are references to the second byte (the high byte) of the standard registers.

reg	8-bit
000	al
001	cl
010	dl
011	bl
100	ah
101	ch
110	dh
111	bh

The REX byte implicitly exists for some instructions: call, enter, leave, ret, loop, loopcc, jmp, jcc, pop, push, lXdt, jrcxz.

If any of the high one-byte registers are desired, the REX prefix may **not** be used.

Opcode

The opcode is the actual instruction.

op:

0x0F op:

0x0F 0x38 op:

0x0F 0x3A op:

ModR/M byte



Mod bits	R/M bits							
	000	001	010	011	100	101	110	111
	A	C	D	B	SP	BP	SI	DI
00	[r/m]				[SIB]	[rip + disp32]		[r/m]
01	[r/m + disp8]				[SIB + disp8]	[r/m + disp8]		
10	[r/m + disp32]				[SIB + disp32]	[r/m + disp32]		
11	r/m							

The rip/eip-relative addressing is only available in long mode. In 32-bit mode, this is instead treated as a 32-bit displacement

With the REX.B bit set, the registers change, but the calculations stay the same.

SIB byte



The Mod bits from the ModR/M byte determine how the SIB byte is interpreted. A SIB byte is never present if the ModR/M byte has Mod bits equal to zero.

Table 17. Mod = b00

		Base bits							
		000	001	010	011	100	101	110	111
Index bits		A	C	D	B	SP	BP	SI	DI
000	A	[b + (sxi)]						[sxi] + disp32]	[b + (sxi)]
001	C								
010	D								
011	B								

100	SP	[b]	[disp32]	[b]
101	BP	[b + (s×i)]	[s×i] + disp32]	[b + (s×i)]
110	SI			
111	DI			

REX.X extends index with one more MSB, REX.B extends base with one more MSB. This affects the registers, but the calculations remain the same.

Table 18. Mod = b01

		Base bits							
Index bits		000	001	010	011	100	101	110	111
		A	C	D	B	SP	BP	SI	DI
000	A	$[b + (s \times i) + \text{disp8}]$							
001	C								
010	D								
011	B								
100	SP	$[b + \text{disp8}]$							
101	BP								
110	SI								
111	DI								

Table 19. Mod = b10

		Base bits							
Index bits		000	001	010	011	100	101	110	111
		A	C	D	B	SP	BP	SI	DI
000	A	$[b + (s \times i) + \text{disp32}]$							
001	C								
010	D								
011	B								
100	SP	$[b + \text{disp32}]$							
101	BP								
110	SI								
111	DI								

The scale factor is determined from two bits to create a power of 2.

Table 20. Scale Bits

Bits	Factor
00	1
01	2
10	4
11	8

Displacement

Displacement values are constant offsets in memory from the specified position.

Immediate

An immediate value is just that, a constant value encoded as part of the instruction.

Exercises

Exercise 1

Use `nm` to Analyze an Executable

Instructions:

Use `nm` to examine an executable that is part of the Linux operating system. To find an executable file, type a few exploratory command lines:

```
which vi
```

```
which gcc
```

```
which objdump
```

Pick one or more of these files and use `nm` to discover which standard C runtime libraries that executable uses. Notice the backticks around `which` and the executable name. For example:

```
nm -D `which vi` | grep " U "
nm -D `which vi` | less

nm -D `which gcc` | grep " U "
nm -D `which gcc` | less

nm -D `which objdump` | grep " U "
nm -D `which objdump` | less
```

Read the man page for `nm`. Feel free to experiment.

Exercise 2

Use objdump to Analyze an Executable

Instructions:

Use objdump to disassemble the Java Virtual Machine.

Notice the backticks around which and the executable name.

```
objdump -D -M "x86-64,intel" `which java` | less  
objdump -d -M "x86-64,intel" `which java`
```

Read the man page for objdump. Feel free to experiment.

Exercise 3

Use `readelf` to Analyze an Executable

Exercise 4

Create a C Program That Does Not Start in `main()`

Purpose:

Sometimes mission requirements require custom starting points

Instructions:

Write a program that starts in a function other than `main()`. Create a C file named `dingo.c` and type the following lines of code:

Listing 42. dingo.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int dingo() {
5     printf( "Hello world\n" );
6     exit(0);
7 }
```

Save the file.

Compile and link the file with the following command:

```
gcc --entry=dingo -nostartfiles dingo.c
```

Run the executable to prove that it works.

Try to compile without the `-nostartfiles` compiler option. What error do you get?

Try to compile without the `--entry=dingo` compiler option. What error do you get?

Summary:

These exercise taught you how to start a program from an entry point other than `main()`

Chapter 21. Week 4, Monday, 7 Hour Lab

General Requirements:

- Code must be written as specified
 - If dynamically allocating memory, ensure that there are no memory leaks
 - Appropriately factor into functions and use appropriate data structures
- Test cases and a test plan that reasonably exercises the code
 - Document the reasoning behind the choice of test cases
 - Document the reasoning behind the choice of code strategies
- Handle any and all errors, such as missing parameters, missing files, I/O errors, etc.

Exercises

Exercise 1 - Locate the data

The file `grades.txt` contains grade information for the students in a class. It is structured as follows:

- Line 1: The course title
- Line 2: The course dates in text format
- Line 3: The instructor's name
- Line 4: A blank line
- Line N: A student name, a tab, and a list of grade numbers separated by comma

Listing 43. grades.txt

```
Title: Assembly Language Using C
Dates: April 1, 2020 - May 5, 2020
Instructor: Noel J. Bergman

Liam    100,100,100,100,100,100,100,100,100
Kevin   100,100,100,100,100,100,100,100,100
Tom     72,59,93,67,83,53,51,50,83,81
Jeff    89,85,51,85,87,77,66,55,88,81
Frank   95,84,94,94,100,83,100,92,99,98
Lou     81,78,62,63,73,95,51,72,58,52
Joe     96,81,62,76,79,59,97,62,97,69
Evan    100,100,100,117,100,100,123,100,100,100
```

The first three lines have a label followed by a colon, after which comes the data for that line.

The data file should be present in the lab environment. If not, create one to match the requirements.

Do not make any assumptions regarding either the number of students or the number of graded items per student. The only assumption that you can make is that the number of entries is symmetric, *i.e.*, each student will have the same number of grades.

Exercise 2 - Read the data

Write an assembly language program that accepts a filename from the command line, opens the file, and reads the information into an appropriate structure.

It should be considered an error if the parameter is not provided, is not readable, or is not in the expected format. Report accordingly.

Exercise 3 - Process the grades

- For each student, calculate:
 - Overall grade average
 - A delta for each graded item to that item's average grade
 - A delta from student's grade average to the overall class grade average
- For each graded item (the Nth entry per student), calculate:
 - The average grade
 - The minimum grade
 - The maximum grade
- For entire class, calculate:
 - The overall grade average
 - The minimum grade
 - The maximum grade

Please note: those are the requirements, not the order in which they will need to be performed.

Modify the storage structure(s) as necessary.

Exercise 4 - Generate a report

Modify the program so that it accepts another parameter, which will be the name of a report file. If that parameter not provided, write to stdout.

Generate a text report containing all of the information:

- The course
- The course dates
- The instructor
- The overall grade for the course
- The range of grades for the course

- Student Grades
 - The student name
 - The student's overall grade
 - The student's grade delta to the course average
 - For each of the student's grades, the grade and the difference to that average for that graded item
- For each graded item (column)
 - The average grade
 - The range of grades for that item

Exercise 5 - Add letter grades

Modify the program so that a student receives an overall letter grade. Letter grades are to be calculated according to the following:

- ≤ 90 A
- $80 \leq B < 90$
- $70 \leq C < 80$
- $60 \leq D < 70$
- $50 \leq E < 60$
- $F < 50$

Exercise 6 - Add low grade discard option

Add a command line option such that the program will discard a student's lowest score, before computing that student's average grade (and resulting letter score).

Verify the results.

Exercise 7 - Add high/low grade discard option

Modify the program so that one can specify the number of high and low grades to remove, *e.g.*, throw out the two lowest grades **and** one highest grade.

Verify the results.

Chapter 22. Review 2

Topics

- Objective
- Syntax
- Memory
- Strings
- Interfacing with Linux
- Structures
- SIMD
- Primate Tools for Examining Data
- Primate Tools for Examining Executable Files
- Code Review

Objective

This unit is a review of the second half of the course. As such, it is not a repeat of the entire first half. The purpose of the review day is to clarify questions that may have come up when doing exercises, the full-day lab, or reviewing solutions; and to prepare for the written and practical exams to follow.

The topics are not really as divided as it appears. For example, allocating a structure would overlap with the Syntax, Memory and Structures topics.

The content here is a guide. Please use this time to ask questions.

Syntax

Review, in general, the overall syntax for Intel. AT&T syntax is not used in this course, but is important to know.

Review the directives for exporting names, *e.g.*, function names from a module.

Review the directives for declaring other sections, such as a BSS or DATA section. Review the related directives for placing data into those sections. Make sure that the difference between BSS and DATA is understood.

Review the syntax for defining labels.

The C compiler tends to generate verbose directives, so it can be an excellent reference. For example, if one wants the assembler to calculate the size of a structure that can be referenced in assembly code.

Memory

Review the overall organization of a process. Be clear about what generally goes where, *e.g.*, code starting from the lower addresses, the stack growing down, the heap being created and expanded with `sbrk` or `mmap`. Review the notes for `malloc(3)` to see the comments on when it uses `sbrk` and when it uses `mmap`.

Another fun reference: <https://sploitfun.wordpress.com/2015/02/11/syscalls-used-by-malloc>

Make sure that the direction in which the stack grows and shrinks is absolutely crystal clear.

Review the Red Zone and be clear on what it is, when to use it (or not), and how to use it.

Review the concept of latency, since that can have so much impact on performance, and thus on how one approaches writing code. Not least, the use of `CMOVcc` instructions.

Strings

Review all of the string related instructions.

Review and make sure that one understands the `lod$` and `stos` family of instructions; there are nuances.

What happens to the entire RAX when using `lod$` with various width sources? Be sure.

Review `scas`, especially with REPcc prefixes. Make sure that `strlen`, `strchr`, and `strrchr` are all understood in terms of these instructions.

Understand the role of the DF, STD and CLD with respect to string instructions.

Review `movs`. Make sure that `strcpy` and `memcpy` are understood in terms of `rep` and `movs`. But for the truly curious, see "Moving blocks of data" in Agner Fog's *Optimizing Subroutines in Assembly* for a discussion of just how complicated it can be to optimially move data within memory.

Interfacing with Linux

Review the System V AMD64 ABI and the differences imposed by the SYSCALL instruction.

Review the SYSCALL instruction in some detail.

Review at least some of the Linux Kernel Call table, at least for those deemed likely to use.

Compare the direct use of SYSCALL with the use if the C Library, as recommended.

Review how parameters are passed (on the stack) when Linux starts a process.

Structures

Review the bit encodings for numbers. Be very clear on the binary representation of signed integers in Intel.

Review how to define a structure in Assembly. How would the structure's size be provided to code by the assembler?

How do structures and SIB addressing relate? How does a structure's size get used with SIB addressing? Was that a trick question?

Review the System V AMD64 ABI. How is a **structure** returned by a function? Should functions like that be used, or would it be better to explicitly pass a pointer as a parameter? Which approach appears to be more common, if one reviews the C Library?

SIMD

Review packed vs unpacked data. What is meant by packed data?

What flags are effected by instructions that use packed data?

What flags are effected by instructions that use unpacked data?

Review how to move data into and out of the XMM registers.

Review horizontal add and subtract.

Review data alignment, and its effect on instruction choice.

What is saturated arithmetic, and when is it useful?

Primitive Tools for Examining Data

Review the available tools, such as `string`, `hexdump` and `xxd`.

What are Magic Numbers, and how can they be used?

Primitive Tools for Examining Executable Files

Review the available tools, such as `nm`, `make`, `gcc`, `as`, `objdump`, `ltrace`, `strace`, etc.

What might prevent `ltrace` and `strace` from working on a process, or might prevent a process from running if it is being traced?