

Assembly Language Programming for Microcomputers

Tak Auyeung, Ph.D.

April 16, 2009

- 20050829 0051 TA: Added section 4.4, and modified 4.5
- 20050130 2101 TA: Added section 4.5 for more methods to allocate memory.
- 20041129 0113 TA: project 4 is finally assigned!
- 20041028 1026 TA: inserted .text into the provided code of project 3, updated makefile to handle multiple dependency file
- 20041027 0045 TA: Project 3 is now available. Read the chapter on project 3.
- 20041022 1000 TA: updated the pseudocode and fixed some errors, postponed the due date of project 2 by three days
- 20041010 2239 TA: changed wordlen to be 32-bit instead of 8-bit (easier to use for index register, not 100% necessary). Also, added a new clue/hint section at the end of the description of project 2.
- 20041008 1118 TA: added more help at the end of the description of project 2. Fixed the pseudocode to handle spaces, fixed the template code (use # and not \$ for comment, line 10 of the template code). Added a section about end-of-file detection. Added another section to use file redirection in gdb.
- 20041008 0209 TA: added project 2 (after subroutine chapter), includes very simple way to submit programs directly from linux3!
- 20041001 1014 TA: added command line method to submit project1 (at the end of the project 1 description)
- 20040926 2330 TA: updated project 1 for 2004 Fall class
- 20040815 2210 TA: get ready for Fall semester! Remove Cygwin stuff (it doesn't work well enough) and move online account stuff up
- 20040122 2136 TA: fixed installation procedure to work in the lab
- 20040121 2356 TA: fixed sample code end sequence to properly exit a program
- 20040121 2314 TA: add new chapter for basic x86 information (registers, addressing modes, the mov instruction)
- 20040119 2348 TA: add exercise to use the tool chain
- 20040119 0118 TA: add chapter to install Cygwin
- 20040114 1241 TA: first chapter is now written!

Contents

I	Architecture and Tools	9
1	Tools	11
1.1	Linux, is it wise?	11
1.2	Installing Linux	11
1.2.1	Knoppix	11
1.2.2	Debian	12
1.3	School Account	12
1.4	But I don't know Unix!	12
1.5	Assembling your first program	13
1.5.1	Assembling	13
1.5.2	Linking	14
1.5.3	Running/Debugging	14
1.6	Explanation	15
2	Accounts	17
2.1	Account Info	17
2.2	Server Info	17
2.3	Remote Access	17
3	Introduction	19
3.1	Components in a Computer	19
3.1.1	Processor	19
3.1.2	Memory	20
3.1.3	Input/output	20
3.1.4	Memory Bus	20
3.2	Assembly Programming	23
3.2.1	What's that?	23
3.2.2	Isn't that pointless?	23
3.3	Exercise	24
3.3.1	Embedded Applications	24
II	Basic Instructions	25
4	x86 Basics	27
4.1	Registers	27
4.1.1	In General	27
4.1.2	i386 Specific	27
4.2	Data Transfer Instructions	28
4.2.1	Size does matter	29
4.2.2	Addressing Modes	29
4.3	Memory allocation	31
4.4	Data versus Code Memory	31
4.5	Static Memory Allocation	31

4.6	Endian-ness	33
5	Arithmetic Operations	35
5.1	Binary Operators (A Review)	35
5.1.1	Base conversion	35
5.1.2	Addition	35
5.1.3	Subtraction	36
5.1.4	Negation	36
5.1.5	Signed versus Unsigned <i>Interpretations</i>	37
5.2	Status Flags	37
5.2.1	ZF: zero flag	37
5.2.2	CF: carry flag	37
5.2.3	SF: negative flag	38
5.2.4	OF: overflow flag	38
5.3	The use of status flags	38
5.3.1	Common instructions that affect the status flags	38
5.3.2	sub and sbb	39
5.3.3	cmp	39
5.4	Interpreting the flags	39
5.4.1	ZF	39
5.4.2	CF	39
5.4.3	SF	40
5.4.4	OF	40
6	Jump Instructions	41
6.1	Unconditional Jump (Transfers)	41
6.2	Conditional Jump (Transfers)	41
6.2.1	What about the others?	42
6.3	Combining cmp and Conditional Jumps	43
6.4	An Example of Control Structure	43
7	Control Structures	45
7.1	An Example	45
7.2	Statement Implementation	45
7.2.1	Conditional Statement	45
7.2.2	While-do (Prechecking)	46
7.2.3	Repeat-until (Postchecking)	46
7.3	Macro Conditional Branches	46
7.3.1	Notation	46
7.3.2	Logical Not	47
7.3.3	Logical Or	47
7.3.4	Logical And	47
7.3.5	An Example	47
8	Exercises (don't turn in)	51
8.1	Initialize	51
8.2	Base Conversion	51
III	Input/output and Operating System Interface	53
9	Requesting OS Services	55
9.1	What is an OS?	55
9.1.1	Resources Managed by an OS	55
9.2	Interface to the OS	55
9.2.1	Privileges	55

9.2.2	Interrupts	56
9.2.3	Privilege Switching	56
9.3	Common Services	56
9.3.1	File Output	56
9.3.2	File Input	58
10	Homework Assignment	59
10.1	Your Program's Task	59
10.2	Input File Specification	59
10.3	Output File Specification	59
10.4	Extra Instructions that will be Helpful	59
10.4.1	Multiplication	59
10.4.2	Division	60
10.4.3	Specifying ASCII Code	60
10.5	Some Additional Hints	60
IV	Subroutines	61
11	The Stack	63
11.1	The Stack Pointer	63
11.2	Push and Pop	63
11.2.1	Push	63
11.2.2	Pop	64
11.2.3	A Simple Example	64
11.2.4	A Tricky Example	64
11.2.5	A Sample Debug Session	64
11.2.6	Another Way to Push and Pop	65
12	Calling and Returning	67
12.1	Subroutines	67
12.2	The Conceptual Behavior of Calling and Returning	67
12.2.1	Exercise	68
12.3	How Call and Return are Implemented	68
12.3.1	Under the Hood	69
12.3.2	A Trace	69
13	The Frame, Parameters and Other Stuff	71
13.1	Passing Parameters Via the Stack	71
13.1.1	An Example	71
13.1.2	How about Registers?	72
13.1.3	Through the Stack?	72
13.2	The Frame Pointer	73
14	Project 3 (200 points)	75
14.1	Objective of the Program	75
14.2	What you are given with	75
14.3	More pseudocode	76
14.4	Constraints	76
14.5	Getting started	76
14.6	How to turn this in?	77

15 Local Variables and Return Value	79
15.1 Properties of Local Variables	79
15.1.1 Local Scope	79
15.1.2 Local Lifespan	79
15.2 Changes to the Work Flow	79
15.2.1 A Subroutine A File	79
15.2.2 Streamlining the Process	80
15.3 Local Variable in Assembly	81
15.3.1 Flash Back: Parameters	81
15.3.2 Local Variables	82
15.3.3 Accessing Parameters and Local Variables	82
15.4 Return Value	83
V Data Structures	85
16 Data Structures in Assembly	87
16.1 Pointers	87
16.1.1 Concept	87
16.1.2 Dereference	87
16.2 Arrays	87
16.2.1 Concept	87
16.2.2 Allocation	87
16.2.3 Indexing	88
16.2.4 Optimizing Sequential Access	89
16.3 Structures (Records)	89
16.3.1 Concepts	89
16.3.2 Allocation	89
16.3.3 Accessing fields	90
16.4 More complex examples	90
16.4.1 auto pointer to struct	90
16.4.2 array of structs	91
17 Project 4	93
17.1 Program Objectives	93
17.2 The Algorithm(s)	93
17.3 Input format	95
17.4 Output format	95
17.5 Constraints	95
17.6 Suggestions	95
17.6.1 New Directory	95
17.6.2 Partitioning the Program	95
17.6.3 Makefile	95
17.7 Testing	96
17.8 What to turn in?	96
VI Advanced Concepts	97
18 Interrupts	99
18.1 Rationale	99
18.1.1 Example: the UART without Interrupts	99
18.1.2 Interrupts to the Rescue	99
18.2 ISRs and Assembly Programming	100

Copyright Notice

All materials in this document are copyrighted. The author reserves all rights. Infringements will be prosecuted at the maximum extent allowed by law.

You are permitted to do the following:

1. add a link to the source of this document at www.drtak.org
2. view the materials online at www.drtak.org
3. make copies (electronic or paper) for *personal* use only, given that:
 - (a) copies are not distributed by *any* means, you can always refer someone else to the source
 - (b) copyright notice and author information be preserved, you cannot cut and paste portions of this document without also copying the copyright notice

Part I

Architecture and Tools

Chapter 1

Tools

The programming tools we use for this class are all free. This chapter describes how to download, install and test these tools.

1.1 Linux, is it wise?

I chose Linux as the environment for this class. Many question the soundness of this decision. Well, I have the following defense:

- It is free. Get an old computer, even a 486, and you can get Linux installed on it for free. Sure, you probably have access to “free” versions of Windows, but why risk getting sued by Microsoft for products that are proven to be, well, buggy?
- Linux is a 32-bit operating system. In other words, the processor runs in protected mode and instructions are by default 32-bit instructions. All tools in Linux are designed to utilize 32-bit mode instructions. On the other hand, classic tools for Windows and DOS are real-mode tools, by default.
- Linux is a POSIX-compliant OS. This means if you know how to interface to Linux, you know how to interface to most U*ix OSes. What you learn can be applied to Solaris, FreeBSD and etc.
- Linux can be frugal. If you strip all the fancy stuff, Linux is quite content with a 486 system with less than 1GB of storage, less than 256MB of RAM and just an SVGA display. All the tools we use in this course run just fine in text mode.

1.2 Installing Linux

1.2.1 Knoppix

The Knoppix live-CD is a bootable CD that allows practically any PC to run Linux without installation! All you have to do is to pop in the CD, reboot the computer, and it will run Linux! Best of all, Knoppix includes a GUI and all the tools that you need for this class. You do need a bit of RAM (512MB) and a fairly fast machine (Pentium class) to use the GUI.

The Knoppix distribution includes drivers to read/write FAT and FAT32 partitions. This means that if you have a FAT or FAT32 partition on your hard disk, you can save files to those partitions from Knoppix.

If you want to use Knoppix, please give me a blank 700MB CD (CD-R or CD-RW). I'll burn the latest Knoppix distribution and return it to you.

Note: the PCs in the lab (room 152) cannot boot from CD. This option is disabled due to security reasons. As a result, the Knoppix CD is a good solution only for doing the homework assignments off-campus, or on your own notebook computer.

One of the main drawback of using Knoppix is that the default account directory is in a RAM drive. This means you lose all the files saved in this directory as soon as you reset the computer. You *can* access a FAT or FAT32 partition (floppy disk, hard disk, USB drive, etc.) and save/restore your files, but that is an extra step to forget and mess up.

Knoppix does offer options to save and restore system configurations and user data using an existing hard disk partition or removable partitions (such as USB flash drives).

1.2.2 Debian

This is relevant if you decide to install Linux permanently. The good thing about installing Linux permanently is speed. The Knoppix distro is a little slow because it needs to load everything from the CD. Furthermore, with a permanent installation, the default home directory is non-volatile, so you don't need an extra step to save/restore your homework files.

My favorite Linux distro is Debian. In fact, Knoppix is based on Debian. Without starting a distro war, let's just say that Debian is a little harder to install, but easy to update and maintain. Because I personally use it, I can also share what I know with you.

Debian is easy to install if you have broadband connection. Download four start-up floppy disks to boot a PC, and everything else can be downloaded. This is cool because you don't have to buy anything! (Except the four floppy disks, but you have that already.) Without broadband, I suggest that you use the Knoppix CD instead. You *can* install Linux permanently from a Knoppix CD.

1.3 School Account

The college provides a Linux account for this class. If you decide not to use it, that's fine. This account is accessible from the internet, which means you can do your homework assignments wherever you have internet access. The bandwidth requirement is minimal, as you only need text terminals.

You can use the school account as a repository of your homework files. In other words, you can use the Knoppix CD offline to work on assignments at home, and then go online and sync the files with the remote account when you want to shutdown the PC. This way, you can continue to work on the assignments wherever you go.

1.4 But I don't know Unix!

Well, this is something that you have to learn on-the-fly. Even if I were to use DOS tools, there is no OS class for DOS! Besides, there is plenty of documents online for your reference.

First thing first. For the most detailed (and longest) document of the assembler itself, go to http://www.gnu.org/software/binutils/2.9.1/html_chapter/as.toc.html. We will only use a fraction of the features, and a fraction of its support for the x86 architecture. Nonetheless, this is the best reference material as far as the tool is concerned.

Most Unix tools have online manuals. To access the online manuals, use the `man` command. The following is a command to view the manual of the `ls` command:

```
man ls
```

That's fairly painless, isn't it? In `man`, use 'q' to quit. The home, end, page-up and page-down keys do what they should. To search for a pattern forward, use '/', followed by the pattern to search for, then ENTER. To search for a pattern backward, use '?' instead.

The following is a list of commonly used commands in Unix/Linux. You can use `man` to find out more about these commands:

- `vim`: a visual editor, difficult to learn, but very efficient once you get used to it. Visit <http://www.vim.org> for more information.
- `nano`: another visual editor, less difficult to learn, most beginners prefer `nano` to `vim`. Visit <http://www.nano-editor.org> for more information.
- `ls`: list the contents of a directory.
- `cp`: copy a file or directory to some place else.
- `mv`: move a file or directory to some place else. `ls` renames files or directories.

- **rm**: be careful with this one, it deletes files or even directories! Read the manual first before trying anything daring.
- **mkdir**: make a new directory.
- **man**: the manual viewer. Yes, you can read the manual of **man** using itself.
- **as**: our assembler. The manual does not include the syntax definition, just the command line options.
- **ld**: our linker.
- **make**: a utility program to automatically execute programs to create targets based on predefined dependencies.
- **ps**: lists processes.
- **cat**: concatenate files to an output. You can use it to view contents of a text file. Does not scroll!
- **less**: a text file viewer that allows paging and searching. **man** uses **less** as a frontend. If you know how to use **man**, you already know how to use **less**.

1.5 Assembling your first program

Type in, or copy and paste the following program into a file that has a `.s` extension. I recommend the name `test.s` for its obvious creativity.

```
.text
.globl _start
_start:
push %ax
pop %ax
movl $1,%eax
movl $0,%ebx
int $0x80
```

This is a fairly harmless program that doesn't do anything. However, we can still use to test our tool chain.

1.5.1 Assembling

To assemble this program, you only need to invoke **as** with the filename of the assembly source code file name. The following command should do it:

```
as --gstabs -o test.o test.s
```

In this command, the `--gstabs` option indicates that the object file should retain debugger information (for each line). This allows us to use the debugger later to verify the execution of each instruction later.

The `-o test.o` option indicates that the output file is an object file with the name `test.o`. Without this option, the assembler attempts to generate an executable file, which generally is not a good idea.

The last part of the command, `test.s` is the name of the assembly source code file.

After you assemble the program, you can use the following command to verify the existence and size of the object file:

```
ls -l test.o
```

1.5.2 Linking

Strictly speaking, the assembler outputs binary code that can be understood by the processor. However, the file format of the output of the assembler is not suitable for the operating system to load into memory and start execution. In addition, there may be multiple source files, each referring to labels defined in other files.

A linker, `ld`, is used to resolve references of labels and output a single file that is suitable for the operating system to execute. For our purpose, the following command is sufficient:

```
ld -o test.out test.o
```

In this command, the option `-o test.out` tells the linker to output an executable file called `test.out`. The debugging information in `test.o` is automatically incorporated into the executable file.

After this step, you can use the following command to verify the existence and size of the executable file:

```
ls -l test.out
```

1.5.3 Running/Debugging

Once you have the executable file, you can run the executable using the following command:

```
./test.out
```

The dot-slash (`./`) is needed because in Cygwin, the current directory is *not* one of the paths in which the operating system searches for executable files. Dot (`.`) is the current directory, and slash (`/`) is a separator for directories.

Of course, since this program doesn't do anything, running the program this way isn't very satisfying.

You can also execute a program using the debugger `gdb`. To start the debugger with our freshly linked executable, use the following command:

```
gdb test.out
```

Note that dot-slash is no longer needed here, because we are not executing the file directly.

Once you are in `gdb`, the prompt changes to `(gdb)` (including the parentheses). Now, you can use `gdb` commands. As usual, the first command you need to learn how to find more help. In `gdb`, `help` is the command to find more help.

To read the source code of the program, use the `list` command. You should see the source code of the program with line numbers next to each line.

Next, we are ready to execute the program. You can run the entire program at once using the `run` command. But that wouldn't be very exciting. Let us run the program instruction by instruction.

In order to do this, we need to set up a break point. When the program is executed to a break point, the debugger stops and waits for further instructions. For our simple program, let's put a break point on the first line of instruction. You can specify where to put a break point by line number. Here, we should put a break point on line 5, using the following command:

```
break 5
```

Now, we can use `run` to execute the program. The program stops *before* the line containing the break point executes. At this point, we can examine all sorts of data. At this point, let us simply look at the values of registers. This is done by the following command:

```
info registers
```

The leftmost column lists the registers, the middle column lists the value of each register in hexadecimal notation, whereas the right column lists the value of each register in decimal representation.

We can use the `continue` command to continue execution at full speed. However, we can also ask the debugger execute one line at a time. This is accomplished by the command `step` or `next` command. The difference between these two commands is not clear from our current example.

When you are ready to exit the debugging, use the command `quit`.

1.6 Explanation

In this section, we look back at the steps taken in the previous section, and explain what each step does.

The source code is written in plain text. The processor (a 386 or better processor in this case) does not know how to execute code written in ASCII. The assembler (`as`) is used to translate the text code (somewhat human friendly) to binary code (understood by the processor).

In the simplest case, this one step should be sufficient. However, for more complex programs, the source is typically broken into multiple files for better maintenance. As a result, a single source file may not contain all the necessary components for a program.

The following command

```
as --gstab -o test.o test.s
```

outputs an *object* file, `test.o`. An object file contains instructions in binary code, but it also contains other types of information. For an object file that corresponds to source code with unresolved symbolic references, the object file has a non-empty section that lists all of these unresolved reference. Each entry includes the symbolic name as well as offsets in the code segment that refers to the symbolic name.

Each object file also contains other book keeping information. As mentioned before, the `--gstab` option means that the generated object file includes extra debugging information. This information is represented by a table that maps addresses in the object file to line numbers in the source file.

The linker (`ld`) is responsible to take a look at all the supplied object files and library files, and try to resolve all the symbolic references. A library file is kind of like a concatenation of many object files. Each library file groups related object files together for easier handling. If the linker is successful in resolving all the unresolved symbolic references, it concatenates the binary code from all the object files as well as the needed portions of library files. Then, the linker “fixes up” the unresolved references.

The linker outputs an executable file that can be invoked by the operating system. Note that the linker can only resolve symbolic references among the object files and libraries. However, it does not know exactly where in memory the program will be loaded when it is run. Some instructions are sensitive to the actual location when the program is run. The executable file contains a table that lists offsets of all locations that need to be fixed up when the program is loaded.

When the operating system invokes a program, the image of binary code in the executable file is loaded into memory of the computer. A special component of the OS, called the *loader*, is responsible for this. The loader is also responsible for fixing up relocation sensitive references.

All of this may sound abstract and difficult to understand. Once we start to talk about addressing modes and instructions, some of this confusion will go away.

Chapter 2

Accounts

Due to some student (most likely not in this class) trashing the Win2000 partitions in the lab using live CDs, the lab does not allow bootable CDs or floppies anymore. There is no easy way around this because we cannot monitor all students who use live CDs.

As a result, the college has set up accounts on one of the servers. This should be a relief to most of you because now you can store files online, and there is no need to bring your files on a floppy, USB flash disk or email.

2.1 Account Info

Your account name is as follows *lastname_firstname*.

This means the account name for “George W. Bush” is going to be `bush_george`.

The initial password is your student ID (*not social security number!*) in 0000000 format. Leading zeros must be typed in. I suggest that you change the password as soon as you get into the account. The command to change password is `passwd`.

2.2 Server Info

The server name is `power.arc.losrios.edu`. You can access it from the internet (i.e., not restricted to the lab).

2.3 Remote Access

If you don’t want to fuss much, you can access the server using `telnet` in Windows. Be warned, however, that the Microsoft implementation of `telnet` has incorrect escape code interpretation for terminal emulation. This means your editor may look funny (be it `vi` or `pico`).

You can use `PuTTY` to remotely log in. A copy of this program is located at <http://www.drta.org/util/putty.exe>. It is GPLed, which means it is free to download and free to install. I personally prefer to launch it from the web so I don’t have to save it anywhere.

To transfer files, you can use `ftp`. The CLI `ftp` included in Microsoft Windows (regardless of version) should work just fine. If you prefer a GUI-type, try `FileZilla`. Download `FileZilla` from http://sourceforge.net/project/showfiles.php?group_id=2. You want to download the `...setup.exe` file, then run it to install the program. Note that `FileZilla` is kind of bloated, it takes up a bit of disk space.

The following part is a contribution from Michael Miller (thanks!). Alternatively, you can use internet explorer (IE) to move files. In IE, specify the following in the address bar:

```
ftp:adoej@linux3.arc.losrios.edu
```

Of course, you need to specify your own log in name.

You will need to provide your password. Once you log in, you can drag files in and out of the IE window.

Chapter 3

Introduction

The term “computer architecture” refers to the subject area that deals with the design and engineering of the various components of a computer. The scope of “computer architecture” varies. Some universities include core level operating system concepts as a part of computer architecture, others only include hardware (electronics) aspects.

For the purpose of this class, “computer architecture” includes all the components up to the *interface* to the operating system. In other words, we shall not discuss the internals of an operating system, but merely how to interface to it. On the other end, we will discuss, very lightly, how major hardware components of a computer interact to perform operations.

The main focus of this class is on assembly programming. Assembly language is the lowest level symbolic language that permits the programmer fully control *exactly* how a computer performs operations in software. In other words, with assembly programming, a programmer can choose not to leave any degree of freedom to the development tool.

3.1 Components in a Computer

3.1.1 Processor

The processor is where computations are performed. The term “processor” has other synonyms. For example, the term CPU (Central Processing Unit) was commonly used in the 60’s and 70’s in large mainframe and minicomputers. This was because in such computers, the CPU was a standalone box about the size of at least a small refrigerator! As electronic computers continue to miniaturize, the processor eventually become small enough to fit into the same case as other major components of a computer. For “microcomputers”, the processor is also called an MPU (microprocessing unit) due to the “micro” size.

Due to the shrinking size of transistors, more and more components are packed into the same physical package that we call “processor”. In addition to just the logic to perform computations, most modern processor ICs (integrated circuits) also include peripheral components (such as timers, direct memory access (DMA) units and etc.). Even the cache, which used to be external to processors, are now on the same silicon die as the computational core.

In this class, we’ll use the following to define core processor elements. This definition is appropriate for programmers, but it is in-depth enough for computer engineers:

- ALU (arithmetic and logic unit): this component, as the name implies, performs all arithmetic and logical operations. It has the electronic circuits to perform addition, subtraction and other operations. It also has the electronic circuits to make simple decisions based on results of arithmetic operations. In general, an ALU can get values to operate from registers and memory locations (see below).
- Registers: these components provides extremely fast data storage for the ALU. Registers can keep up with the full-speed operation of an ALU. In other words, as quickly as an ALU can perform operations, the registers can supply or store data just as quickly. The number of registers is limited, however. On even the most modern processors, up to hundreds of bytes are available as registers. As a result, registers are only suitable for storing data that needs to be utilized “soon”.
- Status flags: each flag is a single binary digit (bit). Some consider status flags as a part of the ALU. However, for programmers, status flags should be viewed independently because instructions can read or write to these status

flags. Status flags are used commonly to indicate attributes of results of arithmetic operations. Some status flags have important, system wide, meanings.

3.1.2 Memory

As mentioned earlier, even though registers can keep up with the operations of the ALU, they lack in quantity. For the storage of data that is not needed “soon”, memory is used.

Memory, in general, is implemented by electronic circuits (as compared to mechanical components). Furthermore, a processor has direct access to memory via the “memory bus”. Memory has multiple locations for storing data. The data stored in a location is completely independent to data stored in other locations. Each location is uniquely identified by an “address”, which is an integer value. Memory locations are numbered from 0, then contiguous integers 1, 2, 3, and etc.

The size of each memory location varies depending on the processor. The most commonly used location size is a byte (8 bits). This means most processors can access data by 8-bit quantities. Most modern processors actually read 32 bits or 64 bits at once in hardware. However, in software, a programmer can consider memory locations can be read byte-by-byte individually.

There are many different types of memory. The two main “traditional” types are RAM (random-access memory) and ROM (read-only memory). RAM provides read and write access, while ROM only provides read access to the processor. This distinction is now blurred because “flash” memory is read-only *most of the time*, but it can be written to occasionally.

If anyone is interested in the history of ROM, PROM, EPROM, EEPROM and etc., we can discuss these off-the-scope topics in the class.

3.1.3 Input/output

A computer is useless with just the processor and memory. This is because there is no way to provide information to process, and there is no way to read the results. The I/O (input/output) of a computer provides the interface to interact with devices that can receive or transmit data.

For example, on a personal computer (PC), there needs to be an interface to read keyboard and mouse events. There also needs to be an interface to send information for printing to a printer. Likewise, most computers are connected to a computer network, and there needs to be some kind of interface that is accessible in software to transmit and receive data via the network.

Regardless of the actual I/O standard (USB, firewire, IDE, SCSI, etc.), all devices are interfaced via I/O locations from the perspective of low-level software. A computer often has many I/O locations. Each individual I/O location is usually a byte in width. Note that each I/O device can utilize to multiple I/O locations.

For example, each serial port of a computer utilizes many I/O locations. One location is used to reflect the status of the serial port, another one is used to control the serial port. Yet another location is used to read a byte received or write a byte to transmit.

Note that a serial port is relatively simple, especially when compared to more complex devices such as an IDE controller, or a sound card.

Most architectures are “I/O mapped”. In an I/O mapped computer, I/O locations are identified by addresses that are independent of memory addresses. This means the first I/O location is 0, then followed by contiguous integers 1, 2, 3, and etc. However, I/O location 0 has *nothing* to do with memory location 0! In fact, I/O mapped processors use one set of instructions to access memory, and another set of instructions to access I/O locations.

Some other architectures, called memory mapped I/O, reserve ranges of memory locations for I/O interface. In other words, there is only one set of instructions for accessing memory *and* I/O locations on these architectures.

Generally speaking, there is no particular advantages or disadvantages to each scheme, especially from the perspectives of programming.

3.1.4 Memory Bus

Theory

A memory bus connects a processor to memory. Note that memory is often implemented by multiple physical integrated circuits (ICs). Furthermore, each IC has its own location 0, 1, 2 and etc. In other words, all the memory ICs have

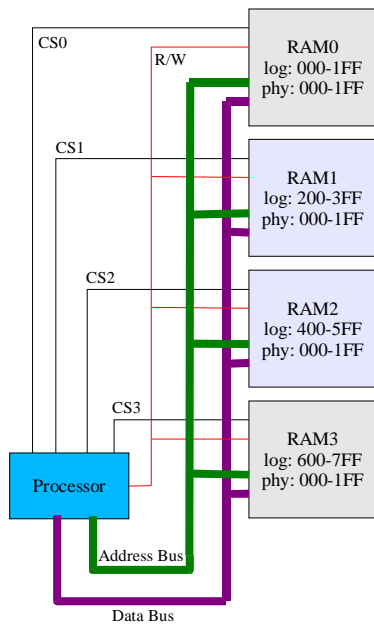


Figure 3.1: A schematic diagram for a memory bus.

location 0, location 1 and etc. As a result, a processor needs to translate memory location addresses into the proper IC and address on that IC.

The processor also needs to indicate whether the access is a read or a write operation. This is indicated by a single signal. When this signal is asserted, it is a read operation, when it is not asserted, it is a write operation. The polarity of this signal (asserted for read or asserted for write) is not important. At the hardware level, the electronics interprets the voltage level on this signal to see if the operation is read or write. Note that the processor is the one controlling the signal, memory ICs are merely listening and interpreting it.

In order for the processor to indicate which IC it wants to access, a *chip select* (CS) signal is allocated to each memory IC. This way, the processor asserts one (and only one) CS for each memory access to make it clear which IC it wants to access.

For the particular memory chip containing the location to access, the processor indicates the address (on the selected IC) to access via the address bus. The address bus has multiple signals, each for one bit in the address. A 32-bit wide address bus can specify more than 4 billion individual locations on a single memory chip. The actual state of the address bus is simply the binary representation of the specified address.

Once the memory IC is selected, address on the IC is specified, and the type of operation is indicated, the *data bus* is used to transmit data between the processor and the selected IC. A data bus has multiple signals. The number of signals in a data bus determines the number of bits that can be accessed by the processor in a single memory operation. Modern processors such as Pentium class processors have 32-bit data busses (which translates to 32 signals on the data bus, one signal for each bit).

For a read operation, the memory IC responds to the processor by specifying data of the accessed location on the data bus. For a write operation, the processor specifies the data to transmit to the selected memory IC on the data bus. Note that for a write operation, the processor writes to the data bus early in the memory bus operation cycle.

Note that this subsection only discusses the interface between a processor and its memory ICs. However, it does not imply the order of events.

Event Ordering

In the previous subsection, we discussed the interface between a processor and its memory ICs. However, the events of a “memory cycle” is not discussed. A “memory cycle” includes all the events necessary for a processor to access (read or write) a memory location.

Note that this discussion is general. Actual processors may alter the sequences.

An Read Example

Let's look at an example of reading from memory. Make the following assumptions:

- each memory location is one byte wide
- each memory chip has 4KB (2^{12} bytes) of storage
- the system has a total of 64KB (2^{16} bytes) of storage

In this example, assume the program wants to read from location 9828. First, we convert the decimal number 9828 to binary. Use a calculator if you need to, but you should know how to do this by hand. Anyway, the binary representation is 0010011001100100_2 .

First, we need to determine which memory IC should be selected. Each memory IC is responsible for 4096 bytes. The first chip (let's call this chip 0) has locations 0 to 4095, the second chip (let's call this chip 1) has locations 4096 to 8191, and etc. In general, chip n has locations $n \times 4096$ to $(n + 1) \times 4096 - 1$.

In our case, $\lfloor \frac{9828}{4096} \rfloor = 2$. The $\lfloor x \rfloor$ takes the "floor" of x . This means it returns the largest integer that is less than or equal to x . Anyway, chip 2 should be selected in our example. As a result, the signal chip select 2 is asserted, while all other chip select signals are negated.

Next, we need to determine the value of the address bus. Recall that the purpose of the address bus is to specify a byte amongst the 4096 bytes contained in each memory IC. Since chip 2 contains locations 8192 to 12287, we need to specify the displacement from 8192 to 9828. This is rather easy: $9828 - 8192 = 1636$. The binary representation of 1636 is 011001100100_2 . As a result, this bit pattern is specified on the address bus to select our byte from memory IC 2.

General memory cycles

For a read cycle:

1. Processor drives the address bus to specify the physical address of the location in the memory module
2. Processor drives the read/write signal to indicate this is a read operation.
3. Processor asserts the chip select signal of the proper memory module.
4. While the processor waits, the selected memory module internally looks up the location and retrieve the content.
5. The selected memory module drives the data bus according to the retrieved content.
6. The processor finishes waiting, samples the data bus and read the content.
7. The processor stops driving the chip select signal. The memory module stops driving the data bus.

For a write cycle:

1. Processor drives the address bus to specify the physical address of the location in the memory module
2. Processor drives the read/write signal to indicate this is a write operation.
3. Processor drives the data bus to indicate the new content of memory location.
4. Processor asserts the chip select signal of the proper memory module, then the processor waits.
5. The selected memory module samples the address bus to find out which location to change, and samples the data bus for the new value. Then the memory module internally writes to the location.
6. The processor finishes waiting, stops driving the chip select signal and the data bus.

3.2 Assembly Programming

3.2.1 What's that?

At the core, a processor fetches instruction opcode (operation code) that is binary. In other words, the actual code that instructs a processor what to do is simply a sequence of zeros and ones. In the early days, programmers actually need to write programs this way (called machine code) and enter programs using banks of toggle switches. Obviously, at this level of programming, the programmer has complete control over the instructions in a program. At the same time, the programmer also needs to track all kinds of details.

As punch cards and terminals became available, it was possible for a programmer to write programs in a semi-symbolic fashion. Instead of entering programs in binary code, instructions can now be entered as mnemonics. It was still difficult to enter long symbolic names, so instructions were replaced by *short and cryptic* symbolic names. For example, BALR stood for “branch and link with register”. At the same time, a programmer can no rely on the tool to track details such as counting locations and associating symbolic names to addresses.

A tool called an assembler is used to translate mnemonics and symbols into the actual binary code to be executed by the processor.

This is assembly programming. In other words, assembly programming is only one level on top of machine code programming. A programmer still has complete control over the instructions contained in a program, but instructions are now specified by short mnemonics. Compared to machine code programming, this is a big leap because an assembler makes it impossible to make common mistakes in machine code programming, such as miscounting locations.

3.2.2 Isn't that pointless?

Yes and no.

If you are referring to the fact that compilers have matured over the past 20 years, the answer is yes. In fact, some compilers, such as the free GNU C Compiler (gcc) can compile programs to code that amazes even experienced programmers. From the perspective of a application or even systems programmer, there is little need to program in assembly.

However, learning how to program in assembly is not pointless. By understanding the kind of instructions a processor executes and low-level operations, a programmer gain insights of how different bugs (in a high level programming language) manifests as symptoms. This understanding of assembly programming, as a result, makes it easier for a programmer to fix bugs even though programs are written completely in a high level programming language.

For professionals who specialize in compilers and interpreters, a thorough understanding of assembly programming is essential. Afterall, a compiler is a translator that converts programs written in a high level programming language into assembly code!

Programming in assembly is actually done in many specialized areas. For example, the source code of an operating system always contains a portion of assembly code. This is because even though the C programming language is flexible and powerful, it still relies on resources already initialized by the operating system. The core of an operating system or kernel always includes assembly to specify operations that cannot be represented by high level programming language code.

An active and increasingly important field that requires assembly programming is embedded systems programming. Some applications in this field utilize microcontroller units (MCUs) that cost less than US\$2 (close to US\$1) each. An MCU is essentially a computer-on-a-chip. It includes the processor, ROM to store the program, input/output abilities, and optionally RAM to maintain run-time data. A US\$2 MCU includes little resources, insufficient for programs written in a high level programming language. The only option to program such MCUs is assembly programming.

Note that embedded systems is a growing field because the dramatic drop of MCU cost (from US\$10 to US\$1) over the past 5 to 10 years has opened new possibilities and new applications. Even low price consumer products, such as toasters and personal heaters, can now incorporate programmable MCUs for control purposes. As the applications of low-cost MCUs grow in the near future, so will the demand for programmers who can program these MCUs.

From this perspective, assembly programming is not only useful, it is an essential element for a job market that will expand in the near future.

3.3 Exercise

3.3.1 Embedded Applications

You don't have to turn this in so no one can say that I steal ideas from my students. However, I do want you to explore the possibilities of embedded systems in consumer products. Find three products with street prices of US\$25 to US\$50 that currently do not have "digital control" features. Then, think of how an MCU (remember, an MCU is a computer-on-a-chip) can enhance the operation.

I'll give you a few examples.

- **Backpack:** A normal backpack can be enhanced with blinking bright LEDs for night visibility. The LEDs can be controlled by an MCU that periodically monitors ambient light conditions. This enables the backpack turn on the LEDs automatically, but at the same time extends battery life. A low-battery alarm alerts you when the battery runs low. The alert level is fully programmable.
- **Personal Space Alarm:** Want to doze off at the library, but fear that someone may "borrow" your stuff without your permission? A scanning proximity device can monitor your personal space up to 1 meter from you. If there is any intrusion in this space, the alarm goes off to let you know some one is in your space. An optional voice recording/replay mechanism lets the alarm replay a pre-recorded warning message.
- **Wireless Toddler Leash:** Afraid that your toddler may walk out of your sight or walk too far away from you? Have your child wear an infrared beacon (on a hat or cap) that emits a signal in 360-degrees. A receiver that you carry checks the reception of the beacon signal and its strength. If the signal has been cut off for more than a second, or the strength is reduced, an alarm lets you know you should look for your toddler, and which direction he/she was last detected. Works great on grounded teenagers, too.

See, it is fun! Of course, we can make these products 10 years ago. However, it is only recently that we can make these products and sell them for less than US\$50!

Part II

Basic Instructions

Chapter 4

x86 Basics

This chapter is *very* specific to the x86 family of processors. However, some concepts are easily portable to other architectures.

4.1 Registers

4.1.1 In General

Registers are high speed data storage that can keep up with the ALUs. In other words, operations of an ALU is never stalled because registers cannot supply or store data fast enough. Because registers are “hard wired” to the ALU(s) in a processor, they consume silicon real estate on a processor chip. This also means there cannot be a lot of registers.

In some architectures, most notably the 8088/86 family, certain registers are wired to the ALUs in certain ways. This means all registers are not created equal. Some instructions can only use certain registers. This type of arrangement does not have name, but it is *not* “general purpose registers”.

In other architectures, dating back to the PDP-11 and motorola 68000 series, registers are generalized. This means all registers go through the same switch(es) to the ALU(s). As a result, most, if not all, instructions can use any registers in the register set. This type of arrangement is called “general purpose registers”.

Although the old 8088/86 architecture does not have general purpose registers, the newer versions do have general purpose registers (well, for the most part).

4.1.2 i386 Specific

This section discusses i386 specific topics. This is also applicable to all x86 architectures after the i386 (such as the Pentiums) running in 32-bit enhanced mode.

Due to its heritage, the i386 register set is quite interesting. Although the ALU of the i386 is capable of 32-bit operations, Some registers can be access as 8-bit registers, while other can be access can be access in 16-bit registers.

We are limiting our scope here to just the data and address registers. Other special registers will be introduced later as necessary.

8-bit General Registers

These registers can be access independently, and they serve as an 8-bit data provider to many instructions.

- **al** (a-low)
- **ah** (a-high)
- **bl** (b-low)
- **bh** (b-high)
- **cl** (c-low)
- **ch** (c-high)

- **dl** (d-low)
- **dh** (d-high)

Instructions accessing these registers must specify that these data providers are “byte-mode” using the **b** suffix. We’ll see this later when we talk about instructions.

16-bit Data and Address Registers

These 16-bit registers can be access independently. They can be used in most instructions to provide 16-bit of data.

- **ax** (a-extended): this is a 16-bit quantity in which the lower 8 bits (bits 0 to 7) are provided by bits 0 to 7 of **al**, and the higher 8 bits (bits 8 to 15) are provided by bits 0 to 7 of **ah**.
- **bx** (b-extended): similar to **ax**, except its components are **bl** and **bh**.
- **cx** (c-extended): similar to **ax**, except its components are **cl** and **ch**.
- **dx** (d-extended): similar to **ax**, except its components are **dl** and **dh**.
- **si**
- **di**
- **bp**
- **sp**

32-bit Data and Address Registers

These 32-bit registers are *enhanced* versions of the 16-bit ones. The 32-bit version have 16 additional bits to the “left hand side” (more significant side) of the 16-bit versions.

- **eax**: this is the 32-bit version of **ax**. Bits 0 to 15 of **ax** becomes bits 0 to 15 of **eax**. **eax** has 16 additional bits (bits 16 to 31).
- **ebx**: enhanced version of **bx**
- **ecx**: enhanced version of **cx**
- **edx**: enhanced version of **dx**
- **esi**: enhanced version of **si**
- **edi**: enhanced version of **di**
- **esp**: enhanced version of **sp**

4.2 Data Transfer Instructions

For now, we’ll only discuss *one* instruction: **mov**. This mnemonic means “move”. Of course, in a computer, data is not really moved. Consequently, **mov** is a misnomer. It should have been called **copy**.

An operand is a data-specifier. It tells the processor how to get the data necessary for the instruction. **mov** has two operands. The first operand specifies the source, and the second one specifies the destination of the action.

How can this instruction be complicated? Although the concept of moving data is quite simple, complication comes from the various ways to specify the origin and destination of the copying. This section discusses most of the useful uses of this instruction.

4.2.1 Size does matter

But it is not the bigger the better. As an assembly programming, you need to explicitly tell the assembler the size of the data copied. In general, a `mov` instruction becomes `movs`. In this notation, *s* represents the size of the data copied. One of the following suffices (also called modifiers) *must* be used:

- `b` stands for byte (8-bit)
- `w` stands for word (16-bit)
- `l` stands for long word (32-bit)

4.2.2 Addressing Modes

This subsection will quickly become hairy. We'll start with the obvious and simple cases, but we'll also get to the more complicated ones.

Register

This is one of the simplest addressing modes. A register data item simply specifies the content of a register. For example,

```
movb %ah, %b1
```

specifies that the 8-bit contents of `ah` should be copied to the 8-bit destination `b1`.

Note that when you specify a register operand, the name of the register must be prefixed by the percent (%) symbol.

Literal

This is also a simple addressing mode, but it is only useful as a source operand (first operand of a `mov` instruction). A literal operand is a constant value that is fixed by the time a program executes. This means the operand *always* supply the same value. The value of a literal operand is a part of the instruction itself. As a result, this operand does not take much time to get.

For example,

```
movl $523, %eax
```

always copies the value 523 (base-10) to the 32-bit register `eax`.

Direct

A direct (memory) operand specifies that the processor should use a location in memory. For example,

```
movw 2532, %bx
```

tells the processor to go to location 2532 (base-10), retrieve two bytes (location 2532 as the least significant, 2533 as the most significant), and store those two bytes in `bx`.

There is where you need to be very careful. The `$` symbol makes all the difference! With the dollar sign, the operand is a constant. *Without* the dollar sign, the operand is the content of some location(s) in memory.

Indirect

This is where things get a little bit complicated. The contents of a register can be treated as an address (memory location), and the contents at that address becomes the operand in indirect addressing mode.

Let us consider the following example:

```
movl $523, %eax
movb (%eax), %d1
```

This is what the two instructions do:

- the first instruction, as mentioned before, copies the constant value 523 into `eax`
- the second instruction treats the contents of `eax` as an address. As a result, the processor goes to memory location 523, retrieves its content, and copy that to `d1`.

Yes, we could have just specified the following instruction, and leave `eax` alone:

```
movb 523, %d1
```

Based

This is similar to “indirect”, but a fixed (constant) displacement is added to the indirect register, and the sum becomes the address (memory location) to get the operand.

For example,

```
movb -12(%eax), %d1
```

subtracts the constant 12 from the content of `eax`, the difference is then used as an address. The content at this address is copied to `d1`.

Indexed

This is similar to “direct”, but you can specify a register to be the “index”. Essentially, the content of the index register is added to the address, and the sum serves as the address of data.

For example,

```
movb 5234(,%edx), %bh
```

adds the content of `edx` to the address 5234, then the sum is used as an address. The content at this address is copied to `bh`.

But wait, isn't this the same as “based”, except that we have a comma in front of the register? Yes, you are right! Read on, there is more!

Scaled Indexed

This is where “indexed” get interesting. Instead of just adding the content of the index register to the address, you can “scale” it with a scaling factor. This means the product of the scaling factor and the content of the index register is used to add to the address, then the sum becomes the address of data.

For example,

```
movb 5234(,%edx,4), %bh
```

Is going to first multiply the content of `edx` by 4, then the product is added to 5234, then the sum becomes an address. The content at this address is then copied to `bh`.

Based Scaled Indexed

Can it get any worse? Nope. It gets simpler!

In this general mode, it is best to use the following operand notation: $d(b, i, s)$. d represents the displacement, b represents the base register, i represents the index register, and s represents the scaling factor.

The address of data becomes the result of $d + b + (i \times s)$, in which b is the content of the base register, i is the content of the index register, and s is the scaling constant.

4.3 Memory allocation

This section is now obsoleted by section 4.5. Skip it!

I mean it, skip this section!

Since we have already discussed memory addressing modes, it is only appropriate to start discussing how memory is allocated (reserved) in an assembly program.

The `.comm` and `.lcomm` directives are used to reserve a number of bytes in a program. The following is an example:

```
.lcomm dogBiscuit 20
```

This code reserves 20 bytes and give the address of the first byte a symbolic name of `dogBiscuit`. You can also do the same with `.comm`:

```
.lcomm dogBiscuit 20
```

The difference between `.comm` and `.lcomm` is that `.lcomm` defines local symbols, while `.comm` defines gloabl symbols. In the context of assembly programs, local means local to the source file, and global means global to all other files that will be linked by `ld`.

In general, you should use `.lcomm` by default. Exposing symbolic names without good reason can lead to lots of problems down the line. Not exposing symbols when they should have been simply results in linker errors, which are relatively easy to fix (compared to accessing the wrong chunk of memory when a program runs).

4.4 Data versus Code Memory

There are two distinct “segments” of memory in each process. The data segment refers to memory that is intended to store data. Memory locations in this segment may be initialized when a program is first loaded, but the values of such locations can change as a program executes.

The code segment, on the other hand, is loaded when a program is first started, but the values of locations cannot be altered as a program executes.

As an assembler translates a program, it maintains an independent counter for each segment. In other words, the data segment has a counter (for locations that are allocated), and the code segment has its own. This permits an assembly program to alternate between the two segments.

The directive to switch to the data segment (if not already using it) is `.data`, whereas the directive to switch to the code segment is `.text`.

4.5 Static Memory Allocation

This section is added on 2005/01/30 and modified again on 2005/08/29.

Most assembler offer many different methods to allocate memory. The following are a few methods to statically allocate memory for data.

- `.byte <byte-expr>{, <byte-expr>}*`

This directive allocates *and* initializes memory on a byte-by-byte basis. When you use this, replace `<byte-expr>` with an expression that yields the value of a byte. You can use commas to separate two or more bytes.

For example,

```
.byte 23, 42, 32
```

allocates enough space for three bytes, then initialize the bytes to the bit patterns for 23, 42 and 32.

- `.word <word-expr>{, <word-expr>}*`

This directive is similar to `.byte`, except each item is a 16-bit pattern.

- `.int <int-expr>{, <int-expr>}*`

This directive is similar to `.byte`, except each item is an “integer”. Unfortunately, the width of each integer is platform dependent. It can be 16-bit or 32-bit.

- `.fill <repeat>[, <size>[, <value>]]`

This directive allocates and fills a portion of memory with a certain bit-pattern of a particular width. `<repeat>` specifies the number of items to allocate and fill. Without `<size>`, the default size is 1 byte. `<size>` specifies the size of each repeating pattern. `<value>` has a default of 0. However, you can specify it to be something else.

The simplest use is as follows:

```
.fill 20
```

which allocates 20 bytes, and fill them all with the bit pattern of 0. The following does exactly the same thing:

```
.fill 20, 1, 0
```

- `.ascii <string>`

Allocates memory and initializes the content to a string *that is not null terminated*.

The following is an example:

```
.ascii "Tak"
```

It allocates exactly 3 bytes for the ASCII codes of “T”, “a” and “k”.

- `.asciz <string>`

Allocates memory and initializes the content to a string that is null terminated.

The following is an example that allocates *four* bytes (one for the null terminator):

```
.asciz "Tak"
```

- `. = . + <int-expr>`

This increments the address counter for the current segment. This is a quick-and-dirty way to allocate memory (in number of bytes) without initializing anything. For all practical reasons, however, this should have been replaced by `.fill <int-expr>` to allocate the same space but keep it initialized to 0.

We can combine the use of the memory allocation directives mentioned above with the definition of labels. Observe the following example:

```
.data
dogBiscuit:
.fill 20, 1, 0xfd
```

It first switches to the data segment, then it defines a label, `dogBiscuit` to be the “current” location of the segment. After that, it allocates enough locations for “20 1-byte values, each initialized to a value of `0xfd`”. Effectively, `dogBiscuit` is defined to be the address of the first byte of the 20 bytes allocated by the `.fill` directive.

With this directive, we can now use symbolic name instead of numbers to refer to addresses. For example, the following code loads the address of the first byte of `dogBiscuit` into register `eax`:

```
movl $dogBiscuit,%eax
```

You can also copy the word starting at the 4th byte at `dogBiscuit` to register `dx`:

```
movw dogBiscuit+3, %dx
```

The `+3` is not a typo. Because we count from 0, `+3` really means the fourth byte.

When you are debugging in `gdb`, you can view contents of memory using the `x` command. However, because `gdb` was originally designed to work with high-level languages, it does have some interesting behavior.

For example, if you type the following in `gdb`:

```
x dogBiscuit
```

It treats the four bytes starting at label `dogBiscuit` as an address, and then attempts to display the contents that that address. This is because in a high level programming language, there is no need to look at bytes in memory *uninterpreted*. Programmers only need to use the ‘x’ command when they want to look at a chunk of memory pointed to by a pointer. As a result, ‘x’ treats any symbolic argument as a pointer, and dereference it to display contents of memory.

If you want to display the content at `dogBiscuit` (i.e., not to dereference the first four bytes of `dogBiscuit`), you need to use the following instead:

```
x &dogBiscuit
```

If you want to specify the formatting of the output, you can use a repeat count, followed by one format code. For example, if you want to display the 20 bytes at `dogBiscuit` as 20 bytes, displayed in binary, you can use the following command in gdb:

```
x/20t &dogBiscuit
```

Use “help x” in gdb to find out what else you can do with this useful command.

4.6 Endian-ness

How are bits organized when we need more than one byte? For example, how do we represent the value of `0x2d53` in two bytes?

There are two methods. Little endian processors (such as the Pentium processor) store the least significant byte in the byte with the lowest address, and most significant byte in the byte with the highest address. This means that when you inspect memory, the value will display as `0x53 0x2d` because

- When you inspect memory, bytes are printed from left to right, with the leftmost at the lowest address.
- Little endian means the least significant byte is at the lowest address.

Is this confusing? Hopefully, it is not. `0x2d43` is a value expressed in hexadecimal. Regardless of the base, our convention is that the right most digit is the least significant digit. Each hexadecimal digit is four bits, that means two hexadecimal digits make a bytes. That’s why the most significant byte is `0x2d`, while the least significant byte is `0x43`. `0x43 0x2d` is displaying the value of two consecutive bytes in memory, with the left one at the lowest address.

In other words, the potential confusion stems from the fact that when we display memory content, it is “lowest address leftmost”. However, when we write numbers, it is “least significant rightmost”. Combining these two with the little endian convention causes what is leftmost in memory content become rightmost in numerical representation.

Besides little endian processors, there are big endian processors. Most processors other than the Intel Pentium family are big endian processors. This means when a multi-byte integer is stored in memory, the byte at the lowest address contains the most significant 8 bits of the number.

As a side note, TCP/IP is also big endian. This means the most significant byte of a multibyte value gets transmitted first.

Chapter 5

Arithmetic Operations

This chapter deals with arithmetic operations in a processor. First, we get a quick review of binary arithmetic, then we discuss instructions on the i386 that performs arithmetic operations.

5.1 Binary Operators (A Review)

5.1.1 Base conversion

You may want to review your textbook or notes for CIS 3 (CISC 310). Studying the manual of your calculator may be helpful (to perform the conversions quickly, but not necessarily to understand how it works).

5.1.2 Addition

Binary add is much easier than decimal add. Afterall, there are only four cases:

$0_2 + 0_2 = 0_2$ is obvious.

So is $1_2 + 0_2 = 1_2$.

Not to mention $0_2 + 1_2 = 1_2$ (afterall, addition is commutative, which means the order is not significant).

The only one that is a little tricky is $1_2 + 1_2 = 10_2$.

A few quick taps with a calculator should reveal that $10_2 = 2_{10}$. It is starting to make sense. Another way to look at this is $1_2 + 1_2 = 0_2$ with a carry of one to the next bit.

Generally speaking, we can perform multidigit binary addition like we perform multidigit decimal addition. The key to do this by hand is to introduce a row just for the carry from a previous bit position.

We'll use one example here, lets add 00100111_2 to 01001011_2 :

When we begin, we have

```
      00100111
+     01001011
-----
+C           0
-----
```

The first digit (bit 0) has a result of zero, but the addition also yields a carry for bit 1:

```
      00100111
+     01001011
-----
           0
+C         10
-----
          0
```

The sum of bit 1 from the numbers is 0 with a carry, then this zero is added to the carry from bit 0, resulting in a 1 as the final result for bit 1:

$$\begin{array}{r}
 00100111 \\
 + 01001011 \\
 \hline
 00 \\
 +C 110 \\
 \hline
 10
 \end{array}$$

As for bit 2, the sum of the two numbers is a 1, with no carry to bit 3. However, when we add this partial sum to the carry from bit 1, the final result is a 0 with a carry to bit 3:

$$\begin{array}{r}
 00100111 \\
 + 01001011 \\
 \hline
 1100 \\
 +C 11110 \\
 \hline
 0010
 \end{array}$$

This gets boring after a while, the final solution is as follows:

$$\begin{array}{r}
 00100111 \\
 + 01001011 \\
 \hline
 01101100 \\
 +C 00011110 \\
 \hline
 01110010
 \end{array}$$

5.1.3 Subtraction

Subtraction is also quite easy.

$0_2 - 0_2 = 0_2$ is easy.

$1_2 - 1_2 = 0_2$ is also easy.

$1_2 - 0_2 = 1_2$ is obvious.

What is $0_2 - 1_2$? One way to look at this is $0_2 - 1_2 = -1_2$, which is simple. However, another way to look at this is $0_2 - 1_2 = 1_2$ with a borrow from the next bit.

$$\begin{array}{r}
 01110010 \\
 - 01001011 \\
 \hline
 00111001 \\
 -B 00011110 \\
 \hline
 00100111
 \end{array}$$

We'll look at a few other examples in the class.

5.1.4 Negation

While it is easy to negate a number in writing, for example, $-(35) = -35$, it is another story to negate a number if we are only given 0s and 1s. Because all numbers are represented by just 0s and 1s in a computer, there is no negation symbol that we can use for a written negated number.

In binary arithmetic, we use two's complement for negation. But before we can discuss two's complement, let's discuss one's complement.

One's complement, also called bitwise-not, or bitwise-negation, simply flips all bits of 0s to 1s, and all bits of 1s to 0s. It is common to use an overbar to represent bitwise-not. For example, $\overline{0101_2} = 1010_2$. In these operations, it is important to use a fixed number of bits. In other words, leading (more significant) zeros are important.

Two's complement can be defined with addition and one's complement. There is no particular symbol (except the negative symbol) for two's complement. I'll use $C_2(x)$ to represent the two's complement of x .

Two's complement is defined as follows:

$$C_2(x) = \bar{x} + 1$$

For example,

$$C_2(10010011_2) = \overline{10010011_2} + 1 \tag{5.1}$$

$$= 01101100_2 + 1 \tag{5.2}$$

$$= 01101101_2 \tag{5.3}$$

Is two's complete *really* negation? Let's try a few cases (we are not proving it, illustrating).

Let's consider $25 - 11$. 25 has a binary representation of 00011001_2 , whereas 11 has a binary representation of 00001011_2 . The difference is 14, which has a binary representation of 00001110 . If two's complement is negation, then $00011001_2 + C_2(00001011_2) = 00001110_2$

$$00011001_2 + C_2(00001011_2) \tag{5.4}$$

$$= 00011001_2 + (\overline{00001011_2} + 1) \tag{5.5}$$

$$= 00011001_2 + (11110100_2 + 1) \tag{5.6}$$

$$= 00011001_2 + 11110101_2 \tag{5.7}$$

$$= 100001110_2 \tag{5.8}$$

Hmmm. It seems that we have a problem. 100001110_2 is 270!

As it turns out, two's complement is only useful when we use modulo math. In other words, we need to limit the range of values that can be represented. In this case, we limit the range of values to ones that can be represented by 8 bits. As a result, we only pay attention to the least significant 8 bits.

Now the result is 00001110_2 , which is 14!

This is cool, because in modulo-256 mathematics, $C_2(00001011_2) = 11110101_2$ has the same properties as -11! Better yet, two's complement is very inexpensive in terms of gate logic.

In two's complement, a negative value is always represented by a bit pattern that has the most significant bit set to 1. In other words, the 8-bit pattern $1??????_2$ represents a negative value when interpreted signed.

5.1.5 Signed versus Unsigned *Interpretations*

So, does 11110101_2 represent -11 or 245?

The answer depends on how you plan to use it.

Most instructions are insensitive to signed versus unsigned interpretations. For example, there is only one add instruction to add two operands. It works for both signed and unsigned numbers. The only time when signed/unsigned interpretation is important is when you evaluate the flags after an operation. We'll get to the flags next.

5.2 Status Flags

After an arithmetic operation like `add`, the processor changes some status flags to reflect properties of the result. Several flags are particularly useful.

5.2.1 ZF: zero flag

This flag is set (1) iff the result is zero.

5.2.2 CF: carry flag

This flag is set (1) iff the result yield a carry or borrow beyond the width of the operand.

5.2.3 SF: negative flag

This flag is set (1) iff the result has the most significant bit set.

5.2.4 OF: overflow flag

This overflow flag means “the sign of the result makes no sense”.

5.3 The use of status flags

5.3.1 Common instructions that affect the status flags

A few common instructions change the status flags. Many instructions do not affect the status flags.

add and adc

The **add** instruction is used to add two operands. The sum is stored in the second operand (in AT&T syntax).

The **adc** instruction means “add with carry”. This means it adds two operands *and* the carry flag (0 or 1), the sum is stored in the second operand (in AT&T syntax).

Both instructions accept the following operand combinations:

- register to register
- register to memory (direct, indirect, base, indexed, scaled indexed, based scaled indexed)
- memory to register
- immediate to register
- immediate to memory

The **adc** instruction is used normally to handle the addition of integers that exceeds the width of the ALU. Observe the following example:

```
.data
largeNum1: .fill 8
largeNum2: .fill 8
.text
.global _start
_start:
    movl largeNum1, %eax
    addl %eax, largeNum2
    movl largeNum1+4, %eax
    adcl %eax, largeNum2+4

    movl $1, %eax
    int $0x80
```

In this program **largeNum1** is an 64-bit integers, as is **largeNum2**. The first two instructions add the least significant 32 bits of **largeNum1** to the least significant 32 bits of **largeNum2**. The result is stored in the least significant 32 bits of **largeNum2**.

This **addl** instruction sets or clears the carry flag depending on whether there is a carry from bit 31 to bit 32. The second two instructions then add the most significant 32 bits of **largeNum1**, and the carry bit to the most significant 32 bits of **largeNum2**.

Perhaps it is helpful to illustrate this with the following picture:

```

          adc1                add1
    /-----^-----\ /-----^-----\
    largeNum1[63..32] largeNum1[31..0]
+   largeNum2[63..32] largeNum2[31..0]
-----
    ?????????????????? ??????????????????
+   c?????????????????C ??????????????????0
-----
    largeNum2[63..32] largeNum2[31..0]

```

In this picture, **C** is the carry from bit 31 to bit 32. It is the result of the `add1` instruction. The second instruction, `adc1`, take into account the value of the carry flag from the `add1` instruction. `c` (lower case) is the carry from bit 63 to bit 64. This is the carry of the entire addition.

Note that the `movl` instructions do not affect the carry flag. As a result, it is safe to interleave `add1` and `adc1` with the `movl` instructions.

With `adc`, a program can easily add two numbers of any bit width.

5.3.2 sub and sbb

`sub` is subtraction, and `sbb` is subtraction with carry. They work very similarly to `add` and `adc`. Note that with these two instructions, the carry flag reflects borrows. These two instructions have the same permitted operand combinations as the `add` and `adc` instructions.

Also, remember that the first operand is subtracted from the second.

5.3.3 cmp

`cmp` is compare. It is almost the same as `sub`, except that the difference (result) is not stored.

5.4 Interpreting the flags

5.4.1 ZF

The zero flag is easy to interpret. It is set iff the result of an instruction that affects status flags is zero.

If this is the result of addition, it means the sum is zero. There is not much to read into this case.

If this is the result of subtraction, there is also not much to read into it.

However, if this is the result of compare, it means the two operands are the same. Most of the time, the zero flag is examined after compare instructions (rather than addition or subtraction).

5.4.2 CF

The carry flag reflects the carry or borrow from the most significant bit of the operands to the “next digit”.

In addition, this flag is useful for add with carry. In subtraction, this flag is useful for subtract with borrow. However, the carry flag has additional uses.

After an unsigned addition, if the final carry flag is non zero, it means the sum cannot be stored in the second operand. The carry flag usually has no meaning after a signed addition.

Note that we use the same `add` and `adc` instructions for both signed and unsigned operations. It’s only a matter of whether the carry flag is meaningful afterwards.

After an unsigned subtraction or compare instruction, the carry flag (which means borrow) is useful. Iff it is set to 1, the original second operand is less than the first operand. This is important: we use the carry flag to indicate order for unsigned number comparison.

After a signed subtraction or compare instruction, the carry flag has no meaning. It cannot be used to indicate which operand is greater or smaller.

5.4.3 SF

The sign (negative) flag is essentially the most significant bit of the result of an instruction that affects it.

Naturally, the sign flag has no meaning for unsigned operations. In other words, as long as the operands are unsigned, the sign flag should not be used after addition, subtraction or compare.

The sign flag reflects the sign of the sum after adding two signed operands. But that is pretty much it for addition.

After a signed compare, the sign flag alone cannot indicate the order of the operands. Let's consider the following examples.

Let's consider signed 8-bit operations. $00000000_2 - 00000001_2 = 11111111_2$. In this case, the second operand (to the left of the minus symbol) is less than the first operand (to the right of the minus symbol), the the sign flag is set.

Now, let's consider another case: $01111111_2 - 10000000_2 = 11111111_2$. The second operand represents 127, while the first one represents -128. Although the second operand is greater than the first operand, the resulting sign flag is still set!

As a result, we can conclude that the sign flag, at least all by itself, cannot be used to determine ordering in signed compare.

5.4.4 OF

The overflow flag tell you whether the sign of an operation makes sense or not. It is set iff the sign of result makes no sense.

Obviously, this flag has no meaning for unsigned operations.

For example, let us consider $00000000_2 + 00000001_2 = 00000001_2$. A non-negative number added to another non-negative number, and the result is non-negative. In this case, the overflow flag is cleared.

Let us consider another case $01111111_2 + 01111111_2 = 11111110_2$. In this case, we are also adding a non-negative value to another non-negative value, but the result is negative! As a result, the overflow flag is set.

As demonstrated above, the overflow flag is useful because it indicates when a result of out of the range of signed operands in addition and subtraction. However, the overflow also has value in signed compare (not subtraction) because it can *help* to indicate the order of operands.

Consider these four cases: $00000000_2 - 00000001_2 = 11111111_2$, $00000001_2 - 00000000_2 = 00000001_2$, $10000000_2 - 01111111_2 = 00000001_2$ and $01111111_2 - 10000000_2 = 11111111_2$.

In the first case, $00000000_2 - 00000001_2 = 11111111_2$, **OF**=0 because the sign of the result makes sense. Subtracting a non-negative number from another non-negative number can yield a negative value. At the same time **SF**=1.

In the second case, $00000001_2 - 00000000_2 = 00000001_2$ also makes **OF**=0 because the sign of the result still makes sense. However, **SF**=0 in this case.

In the third case, $10000000_2 - 01111111_2 = 00000001_2$ does not make sense. 10000000_2 represents -128, while 01111111_2 represents 127. When we subtract a non-negative value from a negative value, the result should *always* be negative! That's why **OF**=1. At the same time, **SF**=0.

In the last case, $01111111_2 - 10000000_2 = 11111111_2$ also does not make sense. Subtracting a negative value from a non-negative value should always result in a non-negative value. As a result, **OF**=1. **SF**=1 because of the resulting bit pattern.

So, what is the pattern here?

In signed comparison, the second operand is less than the first operand iff **OF** \neq **SF**. This also means that the second operand is greater than or equal to the first operand iff **OF** = **SF**.

This is why some architectures offer an extra signed comparison flag that is **OF** \oplus **SF**. \oplus means exclusive or. This signed comparison flag is set iff the second operand is less than the first.

Chapter 6

Jump Instructions

The i386 architecture (and all subsequent downward compatible processors) has a few jump instructions. This chapter introduces the use of a few basic ones.

6.1 Unconditional Jump (Transfers)

Intel refers to jump instructions as “transfers”. We’ll stick with the “jump” terminology here, just to be consistent with the rest of the world.

The mnemonic name for the unconditional jump instruction is `jmp`. It requires one memory operand, which is the destination address. The memory operand can be “short”, “direct” or “indirect”.

Between “short” and “direct”, the GNU assembler automatically uses the smaller form (“short”) whenever possible. The indirect form allows the destination of a jump to be controlled by values in registers. You can use based, indexed, scaled indexed or based scaled indexed in indirect mode.

6.2 Conditional Jump (Transfers)

After our long discussion of the status flags (C, O, S and Z), we finally have some instructions that make use of those flags!

All the conditional jumps must use one short displacement or full displacement operand. The assembler automatically uses the shortest possible form. There is *no* indirect operand.

Although there are many more conditional jump instructions, the following ones are the most basic ones. Each of these instructions only examines one flag and uses the value of that one flag to determine whether a jump should occur or not.

- `jc/jb`: jump if and only if carry is set
- `jnc/jnb`: jump if and only if carry is cleared
- `jo`: jump if and only if overflow is set
- `jno`: jump if and only if overflow is cleared
- `js`: jump if and only if sign is set
- `jns`: jump if and only if sign is cleared
- `jz`: jump if and only if zero is set
- `jnz`: jump if and only if zero is cleared

In addition to these instructions that inspect one flag, there are some that inspect more than one flag.

- `jl`: jump if and only if “less than”

- `jnl`: jump if and only if “not less than”
- `jg`: jump if and only if “greater than”
- `jng`: jump if and only if “not greater than”
- `ja`: jump if and only if “above”
- `jna`: jump if and only if “not above”

Sounds confusing? “Less” and “Greater” in this case apply only to signed interpretation, while “Above” and “Below” apply only to unsigned interpretation.

In other words, this is *exactly* how the confusing instructions mean:

- `j1`: jump if and only if $S \oplus 0 = 1$
- `jnl`: jump if and only if $S \oplus 0 = 0$
- `jg`: jump if and only if $(Z = 0) \wedge (S \oplus 0 = 0)$
- `jng`: jump if and only if $(Z = 1) \vee (S \oplus 0 = 1)$
- `ja`: jump if and only if $(Z = 0) \wedge (C = 0)$
- `jna`: jump if and only if $(Z = 1) \vee (C = 1)$

CISC vs. RISC: Do we need all of these conditional branch instructions? The answer is clearly no. From a minimalist perspective, the following conditional branches are necessary:

- `jc`
- `js`
- `jz`
- `jo`

An optional but handy one is `j1`. Even though `j1` can be implemented using `js` and `jo`, it is so commonly used that being its own instruction is worth the silicon.

Given `jc`, it is easy to implement `jnc`. The following code does what `jnc label1` does:

```
jc  continue1
   jmp label1
continue1:
```

Most processors, include RISCs, include `jnc`, `jnz`, `jno`, `jns` and `jnl` instructions because they are relatively inexpensive in terms of silicon, and they can be used to optimize execution in a pipelined architecture (but only when programs are written in assembly language).

6.2.1 What about the others?

The i386 architecture has a bit more specialized instructions such as `jcxz`, `jecxz`, `loop`, `loopz` and `loopnz`. All of these instructions are unnecessary and usually not available in a RISC architecture.

- `jcxz`: jump only if `cx` is zero. This instruction combines the testing of `cx` and a condition jump “only if `cx` is zero” into one instruction. Note that this instruction does not alter `ZF`.
- `jecxz`: similar to `jcxz`, but uses `ecx`.
- `loop`: loop to the operand (direct memory operand) `cx` times. This instruction combines the decrement test and `jcxz` into one.
- `loopz/loope`: loop with zero/equal, similar to `loop`, but it branches only if `cx` is non-zero and `ZF` is set.

- `loopnz/loopne`: loop while not zero, similar to `loop`, but branches only if `cx` is non-zero and `ZF` is cleared.

From the perspective of efficient i386 code, these instructions are useful. However, most other architectures, including CISCs, do not have instructions similar to these. You should understand these instructions, but for academic purposes, we will not use these instructions in this course.

By now, you should already know why `cx` gets its name: it is “C” for counter!

6.3 Combining `cmp` and Conditional Jumps

The compare instruction is particularly useful when followed immediately by conditional branches. Consider the following *abstract* assembly code:

```
cmp op1, op2
j? label1
```

This code compares `op2` to `op1`, then perform the jump `j?` based on the flags set/cleared by the `cmp` instruction. Recall that `cmp` changes the flags based on the result of `op2 - op1`. Generally, you can say that this means “jump to `label1` if and only if `op2` is ? `op1`.” Substitute ? with “equal to”, “less than”, “above” and etc.

As an example, let us look at the following code:

```
cmpw %bx,%ax
jng label1
```

These two instructions combine to say “branch to `label1` if and only if `%ax` is not greater than `%bx`.” The bottom line is that the `cmpw` instruction subtracts `%bx` from `%ax`, and then `jng` branches if and only if $(Z = 1) \vee (((S \oplus 0) = 1))$.

6.4 An Example of Control Structure

Let us consider the following pseudocode:

```
Given unsigned numbers num1 and num2, find the product
product ← 0
while num1 > 0 do
  product ← product + num2
  num1 ← num1 - 1
end while
```

Let us assume that `product` is in a register, say `%ax`, while `num1` and `num2` are memory locations. Then the program translates to the following incomplete code:

```
movw $0, %ax
# while num1 > 0 do
  addw num2, %ax
  subw $1,num1
# end while
```

The important part is how we implement the loop. The meaning of a “while-do” loop is that we check the pre-condition first. If and only if the pre-condition is true, we perform an iteration. After each iteration, we reevaluate the pre-condition. As soon as the pre-condition evaluates to false, we continue with whatever code is after “end while”.

This means the line

```
# while num1 > 0 do
```

should be substituted with the code that says “if and only if `num1` is less than or equal to zero, conditional jump out of the loop”. This translates to “if and only if `num1` is not above zero, conditional jump out of the loop”. To do this, we change the code as follows:

```

    movw $0, %ax
#   while num1 > 0 do
    cmpw $0,num1
    jna  exitLoop
    addw num2, %ax
    subw $1,num1
#   end while
exitLoop:

```

We add the `cmpw` and `jna` (jump iff not above) to get out of the loop, and we add the label `exitLoop` to mark where to branch when we exit the loop.

At the same time, the line

```
#   end while
```

should be replaced by “go back and reevaluate the pre-condition”. This is done as follows:

```

    movw $0, %ax
#   while num1 > 0 do
evalCond:
    cmpw $0,num1
    jna  exitLoop
    addw num2, %ax
    subw $1,num1
    jmp  evalCond
#   end while
exitLoop:

```

We added the label `evalCond` to mark the beginning of the code that evaluates the pre-condition, and we add the `jmp` instruction to go back to reevaluate the pre-condition after an iteration.

Can this code be optimized? Yes. Can we use `jz` instead of `jna`? Yes, but only because we compare to 0.

The slightly more optimized version is as follows:

```

    movw $0, %ax
#   while num1 > 0 do
    cmpw $0,num1
    jna  exitLoop
performAdd:
    addw num2, %ax
    subw $1,num1
    jnz  performAdd
#   end while
exitLoop:

```

This optimization moves the beginning-of-loop label (and rename it), and it also changes the `jmp` instruction. It preserves the meaning of the original version, but it is now faster because the `subw` instruction sets the Z flag as soon as `num1` is zero, so there is no need to compare again at the beginning of the loop. The first `cmpw` is still necessary because `num1` can be initialized to 0.

Chapter 7

Control Structures

This chapter deals with the general translation of control structures of a high level programming language to assembly code.

7.1 An Example

Let's begin with an example as follows:

```
if  $x < y$  then
  block1
end if
```

To translate this code, we want to do the following skeletal code:

```
# if  $x < y$  then
#   branch around block1 iff  $x \geq y$ 

# whatever block1 is
```

Assuming x and y can be used in the same compare instruction, we can implement the code as follows:

```
# if  $x < y$  then
#   branch around block1 iff  $x \geq y$ 
  cmp  y,x
  jnl  aroundBlock1

# whatever block1 is
aroundBlock1:
```

This implementation is based on the interpretation of a compare instruction followed immediately by a conditional branch instruction. Review section 6.3 for more information.

7.2 Statement Implementation

7.2.1 Conditional Statement

For a general conditional statement like this:

```
if condition then
  then-block
else
  else-block
```

end if

We can first translate to the following rather ugly high-level code:

```
if not condition then
  goto label1
end if
then-block
goto label2
label1:
else-block
label2:
```

Note how we have translated to a condition branch, goto statements and label definitions. We'll discuss how to translate to the general conditional branch later.

7.2.2 While-do (Prechecking)

Now let us consider the next control structure:

```
while condition do
  while-block
end while
```

This translates to the following ugly code:

```
checkPrecondition:
if not condition then
  goto exitLoop
end if
while-block
goto checkPrecondition
exitLoop:
```

7.2.3 Repeat-until (Postchecking)

This is our last control structure:

```
repeat
  repeat-block
until condition
```

It translate to the following ugly code:

```
beginIteration:
repeat-block
if not condition then
  goto beginIteration
end if
```

7.3 Macro Conditional Branches

Now that we have gotten simple conditions and control structures out of the way, let's focus on the general case for compound/macro conditions.

7.3.1 Notation

We will continue to use the following notation for conditional branches:

```
if condition then
  goto label
end if
```

However, *condition* can now be a compound condition.

As a brief review, here are the symbols for boolean operations:

- \vee is logical or
- \wedge is logical and
- \neg is logical not
- \oplus is exclusive or

7.3.2 Logical Not

```

if  $\neg C$  then
  goto label
end if
is translated to
if  $C$  then
  goto local1
end if
goto label
local1:

```

Although we can always do it this way, most of the time we can change the condition $\neg C$ to a form that can be translated directly.

7.3.3 Logical Or

```

if  $C \vee D$  then
  goto label
end if
is translated to
if  $C$  then
  goto label
end if
if  $D$  then
  goto label
end if

```

Note that this is called “short circuited evaluation” because we do not always have to evaluate D .

7.3.4 Logical And

```

if  $C \wedge D$  then
  goto label
end if
is translated to
if  $\neg C$  then
  goto local1
end if
if  $D$  then
  goto label
end if
local1:

```

7.3.5 An Example

Let’s deal with an example:

```

while  $(x < y) \wedge (\neg(x > 0) \vee (y < z))$  do
  block1
end while

```

We refer to subsection 7.2.2 and get the following code:

```
beginloop:
if  $\neg((x < y) \wedge (\neg(x > 0) \vee (y < z)))$  then
    goto endloop
end if
block1
goto beginloop
endloop:
```

Next, we identify the logical negation, and transform the code as follows (as describe in 7.3.2:

```
beginloop:
if  $(x < y) \wedge (\neg(x > 0) \vee (y < z))$  then
    goto local1
end if
goto endloop
local1:
block1
goto beginloop
endloop:
```

Now we apply the transformation for logical and (7.3.4):

```
beginloop:
if  $\neg(x < y)$  then
    goto local2
end if
if  $(\neg(x > 0) \vee (y < z))$  then
    goto local1
end if
local2:
goto endloop
local1:
block1
goto beginloop
endloop:
```

Now we apply the transformation for logical or (7.3.3):

```
beginloop:
if  $\neg(x < y)$  then
    goto local2
end if
if  $\neg(x > 0)$  then
    goto local1
end if
if  $(y < z)$  then
    goto local1
end if
local2:
goto endloop
local1:
block1
goto beginloop
endloop:
```

We can now stop. How about those negations? Well, we can rewrite our conditions to get rid of them:

```
beginloop:
if  $(x \geq y)$  then
    goto local2
end if
```

```

if ( $x \leq 0$ ) then
    goto local1
end if
if ( $y < z$ ) then
    goto local1
end if
local2:
goto endloop
local1:
block1
goto beginloop
endloop:

```

Isn't this a bit clumsy? After all, we go to label "endloop" after arriving at "local2". We can optimize the code a little by replacing all references to "local2" by "endloop", then remove the definition of "local2" altogether:

```

beginloop:
if ( $x \geq y$ ) then
    goto endloop
end if
if ( $x \leq 0$ ) then
    goto local1
end if
if ( $y < z$ ) then
    goto local1
end if
goto endloop
local1:
block1
goto beginloop
endloop:

```

There may be some more possible optimizations, but the code is correct now. Given that x , y and z are already loaded into registers, it is only a matter of replacing each conditional branch with the proper `cmp` and condition jump instructions.

The final pseudo assembly code as follows:

```

beginloop:
    cmp    y,x      # if x >= y goto endloop
    jnl   endloop
    cmp    $0,x     # if x <= 0 goto local1
    jng   local1
    cmp    z,y     # if y < z goto local 1
    jl    local1
    jmp   endloop # goto endloop
local1:
    # put block1 here
    jmp   beginloop # goto beginloop
endloop:

```


Chapter 8

Exercises (don't turn in)

8.1 Initialize

Write a small program to initialize a number of bytes. Start with the following code:

```
    stringlen = 20
.lcomm stringbuf stringlen
.text
.global _start
_start:
    # put code here
    # initialize all 20 bytes to spaces

    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

8.2 Base Conversion

Change the division program to convert a number from binary (internal representation) to a printable format in base 10.

```
.lcomm stringbuf 20
    stringlen = . - stringbuf
.text
.global _start
_start:
    # initialize to spaces first

    # fill in from last byte to first byte
    # convert to base 10
    # using a loop of division
```


Part III

Input/output and Operating System Interface

Chapter 9

Requesting OS Services

9.1 What is an OS?

In our context, we only consider the “core” part of an operating system. This portion is often called a kernel, since it is the innermost portion of an operating system.

An operating system (OS) is a piece of software that provides common interfaces to resources available on a computer platform to *application* programs.

9.1.1 Resources Managed by an OS

Almost all hardware resources are managed by a good OS. For example, devices like the screen, keyboard, sound card, USB ports, parallel ports, modems, network cards and etc. are all managed by an OS.

Some resources are not as obvious. For example, the processor is one of the devices managed by an OS. Given a number of active programs, an OS is responsible to share processing resources among the programs in a “fair” manner.

Memory is another not-so-obvious resource that is managed by an OS. Although a program can request memory, the OS is responsible to deny requests when system memory is low.

The file system is another resource that is managed by an OS. When an application program needs to write to a file, or to read from a file, it must request such actions via the OS. We’ll expand this topic later in this chapter.

9.2 Interface to the OS

Although it is possible to implement system services as subroutine calls, most “protected” operating system do not use subroutine calls. This is because in a protected operating system (such as Linux, FreeBSD, Windows NT/2000/XP/2003/Longhorn and MacOS X), the OS needs to ensure that system services are only granted properly.

This is important. Without protection, a virus or worm program can easily request the OS to write themselves onto other files, causing widespread damage to the OS and resources managed by the OS. With protection, even if a virus or worm program gets to run, their requests to write to system files will be denied. This helps to limit infections to user-owned files, which helps to contain viruses and worms.

In a multi-user environment, protection from the OS is even more important. An OS needs to ensure that a hacker cannot get system services that can compromise data privacy, security and integrity.

9.2.1 Privileges

Most modern architectures intended to run in a multi-user and multi-process environment has features to protect the system from bugs and abuse. For example, it is common that a processor can define regions in memory that are accessible. The accessibility is granted to programs by an operating system. Obviously, the memory used by an OS kernel needs to protected from user programs.

This means that a user program cannot access memory used by other programs or the kernel itself. In other words, an application program usually has no privilege to access most memory locations. In addition, an application program also has no privilege to access input/output devices.

By comparison, when an OS kernel executes, it has privileges to perform practically all operations.

9.2.2 Interrupts

An interrupt is essentially a request by some hardware device for the attention of the processor. Interrupts occur typically when a hardware device needs to be fed, or when an operation is completed.

In order to service these requests from hardware, an OS includes many interrupt service routines (ISRs). Each ISR handles a particular source of interrupt. Entry points to ISRs are listed in a vector table that must be initialized by an OS.

9.2.3 Privilege Switching

Interrupts occur asynchronous to the execution of programs. In other words, it is impossible to predict when a particular hardware device will require attention. Recall that an application program usually runs with little privileges, certainly not enough to access hardware devices.

When an interrupt occurs, the ISR must have access to hardware devices. This means the hardware implementation must bump up privileges when ISRs are executed.

While most interrupts correspond to some event generated by hardware devices, a special class of interrupts does not connect to any hardware. These interrupts are called “software interrupts”. A software interrupt occurs whenever the processor encounters the software interrupt instruction `int`. When a processor executes an `int` instruction, it goes through the same motion as handling a hardware interrupt. In other words, the processor must increase the privilege level before entering the ISR, when decrease the privilege level right before the ISR completes.

Software interrupts are the perfect gateway to access system services. First, privilege level is escalated so that the corresponding ISRs has access to all the system resources. Second, the actual destination of the ISR is in the vector table. This means an application program never needs to know, and should not know, the actual memory locations containing code for system services. This *also* means that there is no need to link an application program to system services (using `ld`).

9.3 Common Services

In Linux, all system services are accessed via software interrupt 128. In an assembly program, this is implemented using the following instruction:

```
int $0x80
```

To distinguish a system service from others, register `%eax` should be initialized with a number to indicate the type of service requested. Additional parameters are specified in `%ebx`, `%ecx`, `%edx`, `esi` and `edi`.

For example, when `%ax` has a value of 1, it means an application is requesting the Linux kernel to terminate its execution and free up all the resources. When `%ax` has a value of 1, the value of `%bx` is used as an exit code.

9.3.1 File Output

File output, file writing, is accomplished by service number 4. Let us write a program to illustrate how you can print a message to the output.

The whole program is located at `output.s` in the `samples` folder. Here is a line-by-line break down of this program.

```
.data
```

This begins the definition of the initialized data of the program.

```
myMessage: .ascii "This is cool!\n"
```

This line has two elements. First, the label definition `myMessage:` (note the colon) defines a symbolic name for the “current” location. What is the current location? It is the first byte of the ASCII string (not null-terminated) `"This is cool!\n"`. 14 bytes are allocated in this case, one byte for each character. Note that `\n` is the notation for the newline character, which is a single character.

```
myMessageLen = . - myMessage
```

This line defines another symbolic name, `myMessageLen`. This symbol, unlike `myMessage`, is not a memory location label. Instead, it is just a symbolic name defined to be the value of `. - myMessage`.

The dot (`.`) represents “the current address”. At this point, the current address is that of the location immediately *after* the last byte of the ASCII string. In other words, the location right after the newline character.

As a result, the difference `. - myMessage` is actually the length of the ASCII string. This is a nifty definition because we don’t have to manually count the number of characters in the message.

```
.text
```

This is telling the assembler that we are about to specify code.

```
.global _start
```

This is our usual line to tell the assembler to “export” the name and definition of `_start`.

```
_start:
```

This defines the label `_start` at the beginning of our program.

```
movl $myMessageLen, %edx
```

We need to specify the number of characters to output in `%edx`.

```
movl $myMessage, %ecx
```

This specifies the address of the first byte to output. In this program, we use `myMessage` because we have an ASCII string defined at that location.

```
movl $1, %ebx
```

`%ebx` specifies the handle of the output file. 1 is the value for the standard output file (`stdout`).

```
movl $4, %eax
```

`%eax` specifies the system service. 4 means we want the operating system to output a number of characters (`%edx`), starting at a particular location (`%ecx`) to a particular file (`%ebx`).

```
int $0x80
```

Once we set up parameters, we use the software interrupt instruction `int` to request service. Note that *all* kernel services are requested via software interrupt 128 (hexadecimal `0x80`).

```
movl $1, %eax
```

Set up `%eax` for the program exit request.

```
movl $0, %ebx
```

Specifies an exit code of 0 (normal).

```
int $0x80
```

Request the operating system to terminate the program

9.3.2 File Input

File input is not much more exciting than file output. Instead of using system service number 4, we use system service number 3. The meanings of registers (before the software interrupt instruction) are almost identical. The only exception is that `%ecx` now contains the address of the first byte to store whatever the system returns.

When you use system service 3, you need to specify a file that you can read from. The `stdout` file is a write-only file. To read from the keyboard, more commonly known as `stdin`, you need to specify 0 as the file handle in `%ebx`.

Just like system service 4, you still need to specify the number of characters to read in `%edx`.

Let us consider a small program:

```
.data
inputString: .fill 30
    inputStringLen = . - inputString
.text
.global _start
_start:
    movl $3,%eax # system service 3, file read
    movl $0,%ebx # file handle 0, stdin
    movl $inputString,%ecx # where to store the characters read
    movl $inputStringLen,%edx # number of characters to read
    int $0x80

    movl $1,%eax # put a break point here so we can inspect vars
    movl $0,%ebx
    int $0x80
```

When you run this program, it prompts you to type in something. It proceeds as soon as you press the ENTER key. Inspecting `inputString` reveals that it stores everything that you type, including the ENTER key. However, it only stores up to the number of characters specified in `%edx`. What you type after the number of bytes specified by `%edx` is kept in the file buffer for the next file read operation.

(Added 2004/10/08) After the instruction `int $0x80` executes, you can check the value of `%eax` for the number of bytes *actually* read. This comes in handy for various reasons. First, if the value of `%eax` is zero, it means you are at the end of a file. Second, if you are reading a file in “cooked” mode, the end-of-line character (linefeed, ASCII code 10) terminates a read operation. The value of `%eax` tells you exactly how many bytes are read, including the linefeed character itself.

Note that there are ways to make the system feed you “raw” characters rather than “cooked” ones. “Raw” in this context means special characters like linefeed, backspace, ENTER are not interpreted. Instead, they are put into your buffer directly. “Raw” also means the system does not interpret lines. In other words, the system no longer interprets what is the end of a line, and use that to terminate the system call to read from a file.

While it is possible to do this in assembly, it generally considered too difficult. You can do this more easily by the system command `stty`.

Chapter 10

Homework Assignment

Now that we have introduced file input and output, let's write a program to utilize this new ability!

10.1 Your Program's Task

Write a program that performs base conversion. You'll need to use the division instruction.

10.2 Input File Specification

- each line has a number.
- each number is a base 7 number.
- each number can be represented *unsigned* in 16 bits.
- each number begins on column 1.
- there may be leading zeros.
- each number is followed immediately by the end-of-line character (linefeed, ASCII code 10).

10.3 Output File Specification

- each line has a number, corresponding to the value of the same line in the input file.
- each number is a base 9 number.
- each number is unsigned and represented in 16 bits.
- each number begins on column 1.
- you must use *exactly* 6 digits for each number, including leading zeros.
- each number must be immediately followed by the end-of-line character.

10.4 Extra Instructions that will be Helpful

10.4.1 Multiplication

Use `mul` to get the unsigned product of two unsigned numbers. This instruction expects only one operand because the implicit operand is `al`, `ax`, or `eax`, depending on the width suffix of the instruction. To multiply the value stored in `bx` to the value stored in `ax` and store the result in `dx:ax` (this notation means the most significant 16 bits are in `dx`, while the least significant 16 bits are in `ax`), do the following:

```
# load something into ax
# load something into bx
mulw %bx
# now ax has the product
```

The operand of `mul` can be a register or a memory location (direct, indirect, based, displacement, etc.).

- `mulb` multiplies `al` to the 16-bit operand, stores the 16 bit product in `ah:al`
- `mulw` multiplies `ax` to the 16-bit operand, stores the 32 bit product in `dx:ax`
- `mull` multiplies `eax` to the 32-bit operand, stores the 64 bit product in `edx:eax`

10.4.2 Division

Use `div` to divide a number by another number. Just like `mul`, `div` uses `ah:al`, `dx:ax` or `edx:eax` as the implicit operand (to be divided). In this notation, `dx:ax` means that the least significant 16 bits come from `ax`, whereas the most significant 16 bits come from `dx`. The only operand specifies the divisor of the operation. The result (quotient) is stored into `al`, `ax` or `eax`.

In addition to storing the quotient into `al/ax/eax`, the remainder is also stored into `ah/dx/edx`, respectively.

The following is an example:

```
# load something into ax
# load something into bx
divw %bx
# now ax has the quotient
# and dx has the remainder
```

Just like `mul`, the operand of `div` can be a register or a memory operand.

10.4.3 Specifying ASCII Code

This is rather simple. To add the ASCII code of 'x' to the value in `%al`, do the following:

```
addb $'x',%al
```

The same technique also works for subtraction and comparison.

10.5 Some Additional Hints

When a file is redirected as input, system service 3 tries to read in the number of bytes as specified. This means it does not return when there is a linefeed in the input. To get around this problem, it may be easier to restrict the program to read a byte at a time, and look for end-of-line (the linefeed character) in your code. You have to do this anyway, even if the system service reads one line at a time.

System service 3 returns a value in `eax`. When the system service reads successfully, `eax` returns the number of bytes actually read. At the end of file, `eax` returns a 0 because it cannot read past the end of a file. Use this to terminate the outer so you can exit the program.

I did not know that in `div`, the register that will eventually store the remainder supplies the most significant portion of the dividend. As a result, you need to initialize `dx` to zero right before the `divw` instruction.

Part IV

Subroutines

Chapter 11

The Stack

Up to this point, all data items are allocated explicitly using some assembler directives. There is a memory storage area that does not require explicit allocation, it is the stack.

A stack is a LIFO structure. LIFO stands for last-in-first-out. This means the last item you put into a stack is the first one to remove. Think of the stack like a U-haul moving truck. The first item you put in the truck is the last one to remove, and the last item you jam into the truck is the first item to fall out.

A stack does not sound very exciting or useful at first. However, as we will see later, the stack is very important in any modern computers.

11.1 The Stack Pointer

The register `esp` is the stack pointer. It contains the address of the “last byte put into the stack”. You can also interpret the value of `esp` as “the address of the next byte to remove”.

Note that the stack grows down to lower addresses. In other words, as you put more items on the stack, the value of `esp` becomes smaller and smaller. This also means that the stack is “used” up to and including the address in `esp`, but all locations below the address in `esp` are “free”.

I know this may sound very strange and counter-intuitive. However, it will be clear that this makes perfect sense once we start to discuss instructions that use the stack pointer implicitly.

The i386 (in 32-bit mode) treats `esp` as a “general purpose register” with one exception. You cannot use `esp` as an index register. In other words, you can practically use `esp` anywhere a general purpose register is expected.

11.2 Push and Pop

To “push a value” is to put a value into the stack, and to “pop a value” is to retrieve and remove a value from the stack. Huh?

To “push” is to load an item to the deepest available space on the U-haul truck called the stack, and to “pop” is to unload the most immediately available item from the same U-haul truck.

Oh.

How can these operations be useful?

Among many other uses, the primary use of “push” and “pop” is to save and restore values in registers and memory locations.

11.2.1 Push

There is a total of three “push” instructions:

- `push` expects one operand that either specifies a register or a memory location. It pushes the value of the specified register or memory location.
- `pushf` has no operand, it pushes the status flag register (which contains the carry, sign, overflow and other bits).
- `pusha` has no operand, it pushes all the registers on the stack.

11.2.2 Pop

There is a total of three “pop” instructions to match the three push instructions:

- `pop` expects one operand that either specifies a register or a memory location. It pops a value from the stack into the specified register or memory location.
- `popf` has no operand, it pops a value from the stack and put that into the status flag register.
- `popa` has no operand, it pops a number of bytes from the stack and copy those values into registers.

11.2.3 A Simple Example

Consider the following code:

```
movl $0x12345678,%eax
push %eax
movl $0,%eax
pop  %eax
```

In this code, the value pushed on the stack is `0x12345678`. Even though we copy a constant value of `0` to `eax`, the value of `0x12345678` is restored when the `pop` instruction is executed.

11.2.4 A Tricky Example

A quick way to make a copy of the status flag register is the following code:

```
pushf
pop  %eax
```

The first instruction pushes the status flag register (a 32-bit register) on the stack. Then this value is popped into `eax`. As a result, we have just made a copy of the status flag register in `eax`.

11.2.5 A Sample Debug Session

Let us consider the following program:

```
.text
.global _start
_start:
    nop # only need this so we can use a break point
    movl $0x12345678,%eax
    push %eax
    movl $0,%eax
    pop  %eax

    movl $1,%eax
    movl $0,%ebx
    int  $0x80
```

When you debug this program in the debugger, put a breakpoint on the first `movl` instruction. Here is what you should observe.

- stops at `movl $0x12345678,%eax`:

Nothing interesting at this point. However, you should at least see the value of the stack pointer. Use `i r esp` in the debugger to see the value of the stack pointer. In my debug session, the value of `esp` was `0xbffffa00`. Single step.

- stops at `push %eax`:

Still nothing interesting yet. You can confirm that the value of `eax` is `0x12345678`, but that is hardly exciting. Single step.

- stops at `movl $0,%eax`:

Now, we have something to talk about. First, re-inspect `esp`. In my debug session, it now has a value of `0xbffff9fc`. In other words, the stack point is decremented by 4. This is because the `push` instruction uses up four bytes to store the value of `eax`, and the stack grows down.

You can also inspect what is at location `0xbffff9fc`. Use the command `x/4xb 0xbffff9fc`, and you should see the value of the four bytes. Do they make sense? Remember, the i386 is a little-endian processor.

Now, single step.

- stops at `pop %eax`:

I know it is boring, but confirm that `eax` now has a value of 0. Single step.

- stops at `movl $1,%eax`:

Inspect `esp` again. This time, it should have a value of `0xbffffa00`. Why? It is because the “pop” operation moves (increments) the stack pointer so that it skips over the value that is retrieved and “removed” from the stack. The value it retrieved and “removed” is `0x12345678`.

Where are we storing this retrieved value? Inspect `eax` and you will see.

The rest of this program is hardly interesting.

11.2.6 Another Way to Push and Pop

Most newer RISC architectures do not have the instructions `push` and `pop`. This is because it is easy to implement these instructions with just regular copy, add and subtraction instructions.

Let us try to push `%ecx` without using `push %ecx`:

```
subl $4,%esp
movl %ecx,(%esp)
```

In the previous code, it is important that the stack pointer be subtracted before we copy the value of `ecx`. In other words, we *cannot* use the following instructions:

```
movl %ecx,-4(%esp)
subl $4,%esp
```

But why? You can try both versions in a debugger, and you cannot see the difference.

The problem is due to interrupts. Interrupts are asynchronous hardware events that “call” interrupt service routines (ISR). Invoking an ISR also needs to use the stack, and the processor assumes the stack is only used up to the address stored in `esp`. This means locations lower than the address stored in `esp` can all potentially be changed by an ISR. Imagine that you have an interrupt between the two instructions. Because `esp` is not decremented when the ISR is invoked, the value of `ecx` will be overwritten by the handling of the interrupt! We’ll talk about this again when we deal with interrupts.

Note that this code is not exactly the same as a `push` instruction because the `subl` instruction modifies the status flags. On most RISC architectures, you can specify a modifier on the equivalent `movl` instruction so that the indirect register is pre-decremented by 4 without changing the status flags.

How about `pop`? The following code implements `pop %ecx`.

```
movl (%esp),%ecx
addl $4,%esp
```

Note that the order is once again important. You cannot use the following sequence:

```
addl $4,%esp
movl -4(%esp),%ecx
```

Once again, the reason is that an interrupt can occur between the two instructions. After we add 4 to `esp`, an interrupt can corrupt the four bytes that *will* be copied to `ecx`. After the interrupt is handled, the `movl` instruction copies the values left behind by the ISR to `ecx`, not the values originally stored at `(%esp)` before the `addl` instruction.

Chapter 12

Calling and Returning

After we discuss the system stack, we are now ready to introduce the call and return instructions.

12.1 Subroutines

Since you have supposed to have taken a programming course before this one, I assume that you are familiar with the concept of a subroutine. A subroutine is also called a function (in C) or a procedure (in Pascal). Regardless of how it is called, a subroutine has the following characteristics:

- An entry point. In a high level programming language, this entry point usually has a symbolic name.
- An exit point. When a subroutine reaches the exit point, it *returns* to whatever code invoked it in the first place. The exit point of a subroutine marks the end of the subroutine.
- A body. This is a sequence of operations to perform between the entry point and exit point.

Subroutines in assembly programs are similar to subroutines in high level languages, but the assembly programming language makes subroutines more flexible. It is also easier to make mistakes in an assembly program that utilizes subroutines. Special care must be taken, or programs can behave very strangely!

12.2 The Conceptual Behavior of Calling and Returning

Let us first discuss conceptually what happens when a subroutine is called. Let us observe the following code:

```
    call sub1
    # block 1
    ...

sub1:
    # block 2
    ret
```

In this code, when the instruction `call sub1` is executed, control is transferred to label `sub1`. As a result, the code in `# block 2` executes. When `# block 2` is completed, the processor executes `ret`. The program then passes control back to the instruction immediately following `call sub1`, which is the first instruction of `# block 1`.

To be more exactly,

- the `call` instruction passes control to a destination specified as the only operand. This operand must be of memory mode (direct, indirect, etc.).
- the `ret` instruction returns to the instruction immediately following the most recent un-returned `call`.

12.2.1 Exercise

1. What happens when the following program executes? What is the value of `%ebx` when the program exits?

```
.text
.global _start
_start:
    movl $0,%ebx
    call sub1
    call sub2
    movl $1,%eax
    int $0x80

sub1:
    addl $1,%ebx
    call sub2
    ret

sub2:
    addl $1,%ebx
    ret
```

2. What happens when the `ret` instruction in `sub1` is removed? Is this program going to crash?

12.3 How Call and Return are Implemented

Let us begin with the equivalent instruction sequences for `call` and `ret`. Note that we ignore the side effects of changing registers and flags in the status register. Let us focus on what is happening as far as the stack, the stack pointer and control is concerned.

The `call X` instruction can be implemented as follows (using `push`):

```
movl $return_address, %eax
pushl %eax
jmp X
return_address:
```

Without the `push` instruction, `call X` can also be implemented as follows:

```
subl $4,%esp
movl $return_address, (%esp)
jmp X
```

The `ret` instruction can be implemented as follows (using `pop`):

```
popl %eax
jmp (%eax)
```

Without the `pop` instruction, `ret` can also be implemented as follows:

```
movl (%esp),%eax
addl $4,%esp
jmp (%eax)
```

12.3.1 Under the Hood

What is really happening? Essentially, the stack is used to keep track of locations to return to for `calls`. When a `call` instruction is executed, the following two operations are done:

- the address of the immediately following instruction is saved on the stack
- control is passed to the location as specified by the operand of `call`

When a `ret` instruction executes, the following is done:

- a four-byte value is retrieved and removed from the stack
- the processor jumps to the address as retrieved by the previous step

12.3.2 A Trace

Let us consider the following program (also available as `callret.s`):

```
.text
.global _start
_start:
    movl  $0,%ebx
    call  sub1

    movl  $1,%eax
    movl  $0,%ebx
    int   $0x80

sub1:
    addl  $1,%ebx
    call  sub2
    ret

sub2:
    addl  $1,%ebx
    ret
```

1. put a breakpoint on the first `call` instruction:

```
b 5
```

2. run the program:

```
run
```

3. stopped at line 5, examine the stack pointer:

```
i r esp
```

The debugger displayed `0xbffffa00`.

4. display the instruction pointer:

```
i r eip
```

This is the address of the `call` instruction. My debugger displayed `0x8048079`.

- single step, the program should stop at the first instruction of `sub1`:

```
s
```

- inspect the stack pointer:

```
i r esp
```

My debugger displayed `0xbffff9fc`, this is exactly 4 fewer than the value before the `call` instruction was executed.

- inspect the stack content at the stack pointer:

```
x/wx 0xbffff9fc
```

My debugger displayed `0x0804807e`. This is the address of the instruction immediately following the first `call` instruction. We'll verify this later.

- single step the `addl` instruction
- single step the second `call` instruction. Inspect the stack pointer and content of the stack. My debugger shows the `%esp` is `0xbffff9f8` and the value is `0x08048092` at the stack pointer address.
- single step the `addl` instruction.
- single step the `ret` instruction.
- show the current address:

```
currentline
```

And the address is `0x8048092`. Haven't we seen this before? It *was* the value at the top of the system stack!

- inspect the stack pointer. It is now `0x9ffff9fc` because one address has just been retrieved.
- single step another `ret`
- show the current address using `currentline`, We are now back on line 7. The address of the `movl` instruction on line 7 is `0x804807e`. This matches the value that we pushed on the stack as shown in step 7.
- inspect the stack pointer, mine show a value of `0xbffffa00`. The stack is now restored to what it was when we first started to run the program.
- the rest of the program is not very interesting.

Chapter 13

The Frame, Parameters and Other Stuff

At this point, we have the basic mechanism to call subroutines and return from subroutines. A few major questions remain. For example, it is important to be able to pass information to/from subroutines. It is also important to create and access local variables.

This chapter discusses how parameters, local variables are all related.

13.1 Passing Parameters Via the Stack

Instead of using registers to pass information into and out of a subroutine, we can utilize the stack to do so. Let us see how this is possible.

13.1.1 An Example

Let us consider an example. Let's say that we want to write a subroutine to print one character to the standard output file. This can be rather handy so that we don't have to set up the system service every time.

Let's call this subroutine `putchar`, it may be implemented as follows:

```
.data
localbuf: .fill 1
.text
putchar:
    push %eax
    push %ebx
    push %ecx
    push %edx
    movl $4,%eax
    movl $1,%ebx
    movl $localbuf,%ecx
    movl $1,%edx
    int $0x80
    pop %edx
    pop %ecx
    pop %ebx
    pop %eax
    ret
```

In this implementation, we use a “global” variable `localbuf` to pass information from the caller to the subroutine. In other words, if we want to print the character J, we can do the following:

```
movb $'J',localbuf
call putchar
```

This works, but it has a few problems.

- The communication variable `localbuf` is allocated statically. Whether or not `putchar` is actually being used. This does not seem wasteful even though `localbuf` is only one character. However, it is the principle that matters. Other subroutines can be a lot more wasteful.
- In order to communicate with the subroutine, we need to know the name of `localbuf`. This is rather cumbersome.

13.1.2 How about Registers?

Well, how about registers?

We can use registers, which are visible to code. This may work! Let us first redefined `putchar`:

```
.data
localbuf: .fill 1
.text
putchar:
    movb  %al,localbuf
    push  %eax
    push  %ebx
    push  %ecx
    push  %edx
    movl  $4,%eax
    movl  $1,%ebx
    movl  $localbuf,%ecx
    movl  $1,%edx
    int   $0x80
    pop   %edx
    pop   %ecx
    pop   %ebx
    pop   %eax
    ret
```

This way, `localbuf` only needs to be known to `putchar`. To print, we can do the following instead:

```
movb  $'J',%al
call  putchar
```

This code is better. But it does have one little problem. We only have so many registers. If a subroutine needs to pass more information than what all of the registers can carry, we have a problem.

13.1.3 Through the Stack?

Yes, we can pass parameters through the stack. This is rather interesting, right?

What we will do from the caller's side is to push the value to print:

```
movb  $'J',%al
push  %ax
call  putchar
addl  $2,%esp
```

In this case, we are pushing two bytes, even though only the least significant byte is useful. This is a convention of many 32-bit or 16-bit architectures. The `addl` instruction after the `call` is merely there to “balance” the stack.

Let's flip our point of view and see how we can access the parameter:

```

putchar:
    push    %eax
    push    %ebx
    push    %ecx
    push    %edx
    movl    $4,%eax
    movl    $1,%ebx
    movl    %esp,%ecx
    addl    $4+4+4+4+4,%ecx
    movl    $1,%edx
    int     $0x80
    pop     %edx
    pop     %ecx
    pop     %ebx
    pop     %eax
    ret

```

The only interesting line is this:

```
addl    $4+4+4+4+4,%ecx
```

Let's track down what has been happening on the stack:

- the value is pushed, -2
- the return address is pushed by `call`, -4
- `eax`, `ebx`, `ecx` and `edx` are pushed, -16

Compared to the value of `esp` at the time of the `movl %esp,%ecx`, the location of the character to print is `20+esp`. This is why we need to `addl $20,%ecx` because `ecx` has the same value of `esp`.

While this method appears to be even more cumbersome and complicated than the other methods, it offers advantages beyond the other two methods:

- it is not limited by the availability of registers
- it does not rely on statically allocated memory
- it is on demand, it is only allocated when a subroutine is to be invoked
- the caller needs not know the name of locations to store parameter values

13.2 The Frame Pointer

So, it *is* really possible to pass parameters via the stack. However, the need to compute the displacement to parameters depend on the number of items pushed within the subroutine. And this is rather difficult to track.

A register that is independent from the stack pointer, called a frame pointer, is used to remember a “reference” point on the stack for the invocation of a subroutine.

`ebp` is the designated frame pointer for the i386 architecture. With the help of the frame pointer, we can change the subroutine as follows:

```

putchar:
    push    %ebp
    movl    %esp,%ebp
    push    %eax
    push    %ebx

```

```
push %ecx
push %edx
movl $4,%eax
movl $1,%ebx
movl %ebp,%ecx
addl $4+4,%ecx
movl $1,%edx
int $0x80
pop %edx
pop %ecx
pop %ebx
pop %eax
pop %ebp
ret
```

In this code, we first push `ebp` itself on the stack. This way, we can easily restore its original value before the subroutine returns. Next, we simply copy `esp` to `ebp`. At this point, the parameter is 8 bytes from the location pointed to by the stack pointer due to the return address (4 bytes) and the saved value of `ebp` (another 4 bytes).

With the help of the frame pointer, we don't need to know what happens to the stack after the `movl %esp,%ebp` instruction. This is because `ebp` is a snapshot of `esp` up to the point immediately after `ebp` is pushed. Whatever happens after this does not affect `ebp`, only `esp`. As a result, we only need to know the displacement of parameters from `ebp` at the entry point of the subroutine.

The use of the frame pointer becomes even more important in the following section, when we introduce local variables.

Chapter 14

Project 3 (200 points)

This project makes sure you understand how parameters are passed in assembly language programs.

14.1 Objective of the Program

You are to complete a program that performs “word count”. In other words, the program, as a whole, reads an input file (from standard input), computes the number of words, then prints the number of words at the standard output.

14.2 What you are given with

I will provide some subroutines. For example, the main program and subroutines to read a single character. You will need to complete the program by writing certain missing subroutines.

The following is the pseudocode of the main program:

```
totalwc ← 0
repeat
  wc ← readword
  if wc > 0 then
    totalwc ← totalwc + 1
  end if
until wc < 0
printdec(totalwc)
```

In addition, I'll give you simple subroutines like the following:

- **int16 readchar(void)**: This subroutine does not take any parameters. It returns a 16-bit integer in **ax**. If end-of-file is encountered, this subroutine returns a negative value. Otherwise, it returns the ASCII code of the read character.
- **void writechar(int16)**: This subroutine takes a 16-bit integer parameter. Only the least significant 8 bits are used. The parameter is written to the standard output file.
- **int16 isspace(int16 ch)**: This subroutine takes a 16-bit integer parameter. Only the least significant 8 bits are used. The function returns a non-zero 16-bit integer if and only if **ch** is one of the following:
 - a space (ASCII 32)
 - a tab (ASCII 9)
 - an end-of-line character (ASCII 10)
 - a formfeed (ASCII 12)

You need to implement “readword” and “printdec”.

14.3 More pseudocode

This is “readword”:

```

state ← 0
repeat
  result ← readchar
  if result ≥ 0 then
    if state = 0 then
      if isspace(result) ≠ 0 then
        state ← 1
      else
        state ← 2
      end if
    else if state = 1 then
      if isspace(result) = 0 then
        state ← 2
      end if
    else if state = 2 then
      if isspace(result) ≠ 0 then
        state ← 3
      end if
    end if
  end if
until (result < 0) ∨ (state = 3)
if state = 0 then
  return -1
else if state = 1 then
  return 0
else
  return 1
end if

```

You can recycle code from the base conversion program to print base 10 numbers.

14.4 Constraints

Your program should output a single 5 digit decimal number. Keep all the leading zeros.

The given subroutines, `isspace.s`, `writchar.s`, and `readchar.s` must be used but not modified. You can add comments if you want, but there should be no changes to the code.

The entire program cannot use *any* static data allocation. In other words, do not use `.lcomm`, `.fill`, `.ascii` and etc. All variables and parameters must be allocated on the stack, and use register `ax` to return results.

Use one file for each subroutine. You need to use `readword.s` for “readword”, and `printdec.s` for “printdec”. The names of the subroutines cannot be changed. In other words, they must link with my code without any alterations.

14.5 Getting started

For organization, I suggest that you use a subdirectory for this project, and put my files along with yours in the subdirectory.

To make your job easier, please follow these steps to set up the files:

```

cd ~
wget http://www.drtak.org/teaches/ARC/cisp310/samples/wc.tar.gz
mkdir wc
cd wc
tar xzvf ../wc.tar.gz

```

This should set up all the necessary files. Note that I put `printdec.s` and `readword.s` in the subdirectory `my`. I suggest you keep the directory structure so it is clear which files are yours, which ones are mine. Obviously, the provided files are merely stubs.

To test the installation, do the following:

```
make wc.out
```

It should succeed without any problem. However, the generated executable is not very useful because `printdec.s` and `readword.s` are just stubs.

Remember to change directory to `my` to work on your files!

14.6 How to turn this in?

The following instructions assume you put the `wc` directory directly under your home directory:

```
cd ~/wc
tar czvf ../mywc.tar.gz Makefile my/*.s *.s
```

Then, you can download the file `mywc.tar.gz` to a PC, then submit it via the Moodle interface.

Chapter 15

Local Variables and Return Value

In a high level programming language, such as C, C++ and Visual Basic, subroutines can define local variables. Local variables are “local” because not only are they local in terms of scope, but also of lifespan.

15.1 Properties of Local Variables

15.1.1 Local Scope

A local variable has a local scope. This means that the local variable is only visible from within the function that it belongs. No other subroutine should have access to it.

Unfortunately, there are no local definitions in assembly programming. As a result, the only way to isolate subroutines from accessing local variable definitions of each other is to define subroutines in different files.

15.1.2 Local Lifespan

The local scope of local variables is fairly easy to understand. However, the local lifespan of local variables is a little more confusing.

“Local lifespan” means a local variable only exists when the subroutine is invoked. Furthermore, *each invocation has its own local variables*. This is what makes recursion possible.

Local variables are created when a subroutine is invoked, and they are reclaimed when a subroutine returns.

15.2 Changes to the Work Flow

15.2.1 A Subroutine A File

We need to limit the scope of local variables. This means we need to define each subroutine in its own file. This is not difficult at all. Let’s consider the following subroutine `sub1`:

```
.text
sub1:
# ... definition of sub1
```

This is fine. However, all labels are, by default, local to the file. This means `sub1` is not visible to other files. This makes this file useless. In order to expose `sub1` to the rest of the program, we need to add the following line in the program (preferably at the beginning):

```
.globl sub1
```

Note that this is not a definition. Instead, it marks `sub1` as a label (symbolic) name that should be visible to other files when the files are linked.

Once we have many files, each defining a subroutine, we need to assemble and link them. Let's assume we have `main.s`, `sub1.s` and `sub2.s` to make up a program. First, we need to assemble the files individually. The following command is used to assemble `sub1.s`:

```
as --gstabs -o sub1.o sub1.s
```

The other two files are assembled similarly.

Next, we need to link the object files to become the executable. The linker is responsible to extract all the `.globl` symbols and resolve all the cross references:

```
ld -o test main.o sub1.o sub2.o
```

15.2.2 Streamlining the Process

Although you *can* type in all the CLI (command line interface) commands, it is far better to do this with a make file. A make file is a file that states the dependency of files. It can get very complicated. I am supplying a sample make file that makes it easier to just add whatever is necessary to it.

The default `Makefile` is as follows:

```
%.o: %.s
as --gstabs -o $@ $?

%.out: %.o
ld -o $@ $?

%.gdb: %.out
gdb $?
```

This is fine for programs that are confined to one single file. For programs that need multiple files, however, we need to make a few changes.

Let us first add the following definition:

```
mainsrc := main.s sub1.s sub2.s
```

This line should be added at the beginning of the `Makefile`. This defines the symbolic name `mainsrc` to be the string `main.s sub1.s sub2.s`. This will become useful later on.

It is also helpful, though not necessary, to define the name of the executable file:

```
main := main.out
```

With the definition of the executable file and the source files, we can now specify a special rule to link the object files:

```
$(main): $(mainsrc:.s=.o)
ld -o $@ $^
```

This rule specifies a target (file to be created) using the previously defined name `main`. The depended file list is `mainsrc`, except with all the `.s` extensions replaced by `.o`. Inside the action, `$@` is the target of the rule, while `$^` is the list of dependent file.

To debug this program, you can type the following command in the CLI:

```
make main.gdb
```

A `Makefile` may take a bit of time to get used to, but it is also very helpful. Particularly, if you type `make main.gdb`, the `make` program automatically makes sure that all the object files are assembled up to date. If you have modified any `.s` file since the last `make`, `make` ensures these source files are reassembled before the object files are linked.

This can potentially save you lots of time. It is frustrating to know that a program doesn't work only because corrected source files are not reassembled, and the same old object files/executable file are used for debugging purposes.

15.3 Local Variable in Assembly

There is no easy way to do this. We will cover the syntax and language related topics as we discuss the concepts.

15.3.1 Flash Back: Parameters

Recall that parameters are pushed before a subroutine is called, and the frame pointer is initialized *after* it is saved. Let us assume the following for `sub1`:

- `p0`: first parameter, a 32-bit integer
- `p1`: second parameter, a 16-bit integer
- `p2`: third parameter, a 32-bit address

We push parameters in reversed order. This means `p2` is pushed first, `p1` second and `p0` last. The `call` itself pushes a 32-bit return address, then the frame pointer is saved, using another 32-bit on the stack. By the time immediately after we execute `movl %esp,%ebp`, we will have the following items on the stack:

- `(%ebp)`: the old `%ebp`, this is the last item pushed, it has the lowest address.
- `4(%ebp)`: the return address
- `4+4(%ebp)`: the first parameter `p0`
- `4+4+4(%ebp)`: the second parameter `p1`
- `4+4+4+2(%ebp)`: the third parameter `p2`

While this works, it is awfully confusing, especially when a programming needs to refer to the parameters within the subroutine. It is much better to define symbolic names as follows:

```
oldbp = 0 # offset from bp to its former value, not very useful
retaddr = oldbp + 4 # offset to return address of subroutine, not useful
p0 = retaddr + 4 # offset to first param
p1 = p0 + 4 # offset to second param
p2 = p1 + 2 # offset to third param
```

Note that each symbol is defined by the previous symbol and the size of the object represented by the previous symbol. This make is possible to readjust values when sizes of parameters are changed. For example, if we decide the `p0` is a 64-bit integer, we only need to change

```
p1 = p0 + 4 # offset to second param
```

to

```
p1 = p0 + 8 # offset to second param
```

The rest of the definitions can remain the same!

15.3.2 Local Variables

As it turns out local variables are also stack items! Unlike parameters, which must be created by the caller, local variables are created by the called subroutine. Most compilers use the convention of “local variables are on the other side of the frame pointer.” This means local variables are reserved on the stack *after* `%ebp` is saved.

Let us resume our example from the previous section. We add the following local variables:

- `strBuffer`: an array of 32 ASCII characters
- `strPtr`: a 32-bit pointer
- `cmpChar`: a single character, but 16 bits are allocated

These local variables will, then, have the following symbolic definitions:

```
strBuffer = oldbp-32 # offset from %ebp to strBuffer
strPtr = strBuffer-4 # offset from %ebp to strPtr
cmpChar = strPtr - 2 # offset from %ebp to cmpChar
```

Unless these local variables should be initialized, the allocation is a simple `addl` instruction. Let’s look at the complete entry code of the subroutine:

```
subl:
    pushl %ebp # save the old value of ebp
    movl %esp,%ebp # initialize this frame
    addl $cmpChar,%esp # adjust stack pointer to reserve for local var
```

Note that `addl` is used instead of `subl` because `cmpChar` is already a negative value.

To clean up the stack right before this subroutine returns, we need the following code:

```
    movl %ebp,%esp # deallocate local var
    popl %ebp # restore old value of ebp
    ret # time to return
```

15.3.3 Accessing Parameters and Local Variables

Within a subroutine that has the frame pointer `ebp` initialized correctly, parameters and local variables are access the same way.

In our example, if we want to initialize the local variable `strPtr` so it points to the first byte of `strBuffer`, we can do the following:

```
movl %ebp,strPtr(%ebp) # start with the ebp
addl $strBuffer,strPtr(%ebp) # then adjust with offset
```

If we want to compare the character pointed to by `strPtr` with `\n`, we can do the following:

```
movl strPtr(%ebp),%eax # eax be the address of the byte
cmpb $'\n',(%eax) # compare the byte to newline character
```

15.4 Return Value

Compared to local variable, the return value is easy. Most compilers, especially C compilers, use a single register to store the return value of a subroutine. This register, in the case of an i386 architecture, is `%eax`.

As a result, in order to return a value, a subroutine must ensure that `%eax` has a meaning return value before a subroutine executes `ret`.

Some languages other than C, for example, Pascal, allows the specification of record or even array return types. This means it is no longer possible to return a value via a register. In these languages, the return value is also a component on the stack. The storage for the return value is reserved *before* parameters are pushed.

Let us consider the following partial C function:

```
struct Huge
sub1(int p1, int p2)
{
    int L1, int L2;
    struct Huge myvar;

    ...
    return myvar;
}
```

The matching definitions of symbols should be as follows:

```
oldBp = 0 # this goes without saying
retAddr = oldBp + 4
retValPtr = retAddr + 4 # because it is reserved last
p1 = retValPtr + 4 # pointer to the return value area
p2 = p1 + 4
myvar = oldBp - Huge_size
L2 = myvar - 4
L1 = L2 - 4
```

In the subroutine, it can access the space reserved for the return value through a pointer to it. This pointer is `retValPtr`. For all practical purposes, the declaration of this pointer is equivalent to `struct Huge *retValPtr`. The return statement, essentially, is copying the entire `myvar` to where `retValPtr` points to.

The code of the caller to set up the invocation frame should look like this:

```
subl $Huge_size,%esp # reserve space
movl %esp,%eax
pushl $253 # this is p2
pushl $3822 # this is p1
pushl %eax
call sub1
# code to utilize the return value
# the return value is at (%esp)
addl $4+4+4,%esp
# esp is now pointing to the return value
# utilize it now
addl $Huge_size,%esp
```


Part V

Data Structures

Chapter 16

Data Structures in Assembly

Just as data structure is important in high-level programming, it is just as important in assembly programming. This chapter discusses how various data structures are implemented in assembly.

16.1 Pointers

16.1.1 Concept

A pointer is a variable that contains the address of something else. For example, `iptr` is a pointer to an integer in the following definition (in C):

```
int *iptr;
```

16.1.2 Dereference

Let us assume that `iptr` is a global/static variable. `*iptr = 0;` translates to the following assembly code:

```
movl  iptr,%eax
movl  $0, (%eax)
```

If `iptr` is the displacement from `ebp` to an auto local variable, then the code becomes the following:

```
movl  iptr(%ebp),%eax
movl  $0, (%eax)
```

16.2 Arrays

16.2.1 Concept

An array is a group of objects of the same type and size, and each object in the group is identified by an index number. In most languages, the indices must start with 0, then follow consecutive integers 1, 2, 3, and etc.

16.2.2 Allocation

Allocating for an array is no different from allocating for a simple object (like an integer). Care should be taken when an array is allocated from the stack as an auto local variable. This is because available space of the stack can vary greatly from one system to another.

The following code allocates a global array `numbers` that has enough room for 200 integers:

```

numbers_itemsize = 4 # each item in this array requires 4 bytes
numbers_numitems = 200 # a total of 200 items in this array
.data
numbers: .fill numbers_itemsize * numbers_numitems

```

In this definition, it may seem tedious and unnecessary to define the symbolic names `numbers_itemsize` and `numbers_numitems`. However, these two definitions let us easily change the array size either due to the size of each item, the number of items, or both.

To allocate an array on the stack (as a local variable), we can do the following instead:

```

numbers_itemsize = 4 # each item in this array requires 4 bytes
numbers_numitems = 200 # a total of 200 items in this array
numbers = prev_item - numbers_itemize * numbers_numitems
# ... more local variable definitions

```

16.2.3 Indexing

Why do we start counting from 0? The reason is obvious when one programs in assembly.

let us consider an array `numbers` defined as follows:

```
int numbers[10];
```

Let us also assume that an `int` is 32-bit (4 bytes). What is the address of `numbers[3]`?

The address of `numbers[3]` is 12 bytes from the beginning address of `numbers`. This makes sense because there are three integers before `numbers[3]`, and each integer takes four bytes.

In general, given that the size of each item is s , and that we want to access the item with index i , and the base address of the entire array is b , the address of $b[i]$ is $b + i \times s$. Note that in this case, we are not using C semantics.

Using our example, given that `numbers` is a global (static) array, and the index is stored in a 16-bit variable `i`, we can use the following code to access `numbers[i]`. Let's say we want to initialize it to zero:

```

movl  $0,%eax # clears all bits in eax
movw  i,%ax   # the index
movw  $numbers_itemsize,%bx # the size of each item
mulw  %bx     # ax is the product
addl  $numbers, %eax      # add the base address of array
movl  $0,(%eax)

```

Using a displacement, we can also do the following:

```

movl  $0,%eax # clears all bits in eax
movw  i,%ax   # the index
movw  $numbers_itemsize,%bx # the size of each item
mulw  %bx     # ax is the product
movl  $0,numbers(%eax)

```

Using the built-in indexed addressing mode, we can also simplify the code to the following:

```

movl  $0,%eax # clears all bits in eax
movw  i,%ax   # the index
movl  $0,numbers(,%eax,4)

```

The last form, however, is very specific to the i386 architecture. Most RISC processors do not have an indexed and scaled addressing mode.

16.2.4 Optimizing Sequential Access

Most C compilers are smart enough to change the following code:

```
for (i=0; i < n; ++i) numbers[i] = i;
```

to

```
for (i=0, ptr=numbers; i < n; ++i) *ptr++ = i;
```

The second form is fast regardless of the size of each item in the array. This is because pointer addition does not involve multiplication.

In assembly programming, we can do the following (assuming `n` is a static/global variable):

```
    movl    $numbers, %eax
    movl    $0, %ebx
loopBegin:
    cmpl   n, %ebx
    jnb    done
    movl   %ebx, (%eax)
    addl   $numbers_itemsize, %eax
    addl   $1, %ebx
    jmp    loopBegin
done:
```

Note that there is no multiplication in this code!

16.3 Structures (Records)

16.3.1 Concepts

`struct` in C (record in Pascal) is used to group *fields* of different types into one clump. For example, let us consider the following definition:

```
struct X {
    int i;
    char ch;
    char str[10];
};

struct X a;
```

`X` is the name of the structure (which can be treated as a user-defined type), whereas `a` is the name of a variable that is an `X` structure. Inside the structure, we have three fields.

Within a `struct` or `record`, individual fields are accessed by name. For example, to initialize field `i` to zero, we do the following:

```
a.i = 0;
```

16.3.2 Allocation

Records and structs are allocated just like any other objects. However, it is convenient to define certain symbolic names. In our example, we should define the following symbolic names:

```

X_i = 0 # this is the first field, so the displacement is 0
X_ch = X_i + 4 # this is the second field
X_str = X_ch + 1 # this is the third field
X_size = X_str + 10 # this is the total size of a struct X

```

With these definitions, we can allocate for an auto local variable:

```
a = prev_var - X_size # a is a struct X auto local var
```

We can also allocate as a global/static variable:

```

.data
a: .fill X_size

```

16.3.3 Accessing fields

Given that `a` is a global/static definition, we can access `a.ch` relatively easily. Let us see how we can compare that to the newline character:

```
cmpb    $'\n', a+X_ch
```

If `a` is an auto local variable, it is still relatively easy to compare it to the newline character:

```
cmpb    $'\n', a+X_ch(%ebp)
```

16.4 More complex examples

Let us consider some more complicated examples.

16.4.1 auto pointer to struct

Let us consider the following C code:

```

void f(void)
{
    struct X *ptr;

    ptr->ch = '\n';
}

```

Given that `ptr` is properly defined as the displacement from `ebp` to the variable, we can translate the C statement to the following code:

```

movl    ptr(%ebp), %eax # eax points to the base of the struct
movb    $'\n', X_ch(%eax) # we need to add displacement to get to ch

```

16.4.2 array of structs

Let us consider the following C code:

```
struct X stuff[20];

int main(void)
{
    int i,j;

    ...
    stuff[i].str[j] = '\0';
    ...
}
```

First, to allocate for the array `stuff`, we can use the following assembly code:

```
stuff_numitem = 20
stuff_itemsize = X_size
stuff: .fill stuff_numitem * stuff_itemsize
```

Next, the statement can be implemented as follows (assuming proper definitions of `i` and `j`):

```
movl    i(%ebp),%eax
movl    $stuff_itemsize,%ebx
mull    %ebx
addl    $stuff+X_str,%eax # add base of array and field offset
addl    j(%ebp),%eax      # indexed
movb    $'\0',(%eax)
```

Let's make some "minor" changes to the C code:

```
int main(void)
{
    struct X stuff[20];
    int i,j;

    ...
    stuff[i].str[j] = '\0';
    ...
}
```

The corresponding assembly code is as follows:

```
movl    i(%ebp),%eax
movl    $stuff_itemsize,%ebx
mull    %ebx
# all instructions above are preserved
addl    %ebp,%eax      # displaced off the frame pointer
# add instruction below are preserved
addl    $stuff+X_str,%eax # add base of array and field offset
addl    j(%ebp),%eax      # indexed
movb    $'\0',(%eax)
```

As you can see, the only change is to utilize `ebp` as a part of the address. This is because when `stuff` is an auto local variable, it is a displacement from `ebp`, rather than a complete (absolute) address all by itself.

Chapter 17

Project 4

This project involves recursion and arrays. Although recursion often strikes fear in even seasoned programmers, it doesn't have to be like this. Given the proper infrastructure (such as stack-based frames), recursion is no more difficult than calling any subroutine.

17.1 Program Objectives

You are to implement the quick sort algorithm in a recursive fashion.

17.2 The Algorithm(s)

```
define sub Quicksort
  given  $a$  is an address to an array
  given  $b$  is the begin index
  given  $e$  is the end index
  local  $p$  is an integer
  if  $b < e$  then
    call Pivot( $a, b, e, \&p$ ) { $\&p$  means the address of  $p$ }
    call Quicksort( $a, b, p - 1$ )
    call Quicksort( $a, p + 1, e$ )
  end if
end define sub
```

The “meat” of quicksort is Pivot. One algorithm for Pivot is as follows:

```
define sub Pivot
  given  $a$  is an address to an array
  given  $b$  is the begin index
  given  $e$  is the end index
  given  $p$  is a pointer to an integer
  local  $v$  is an integer
  local  $i$  is an integer
   $v \leftarrow a[b]$ 
   $i \leftarrow b$ 
  while  $b < e$  do
    while  $(b < e) \wedge (a[e] \geq v)$  do
       $e \leftarrow e - 1$ 
    end while
    while  $(b < e) \wedge (a[b] \leq v)$  do
       $b \leftarrow b + 1$ 
    end while
    if  $b \neq e$  then
```

```

        swap  $a[b]$  with  $a[e]$   $\{(a[b] \geq v) \wedge (a[e] \leq v)\}$ 
    end if
end while
swap  $a[i]$  with  $a[b]$ 
 $*p \leftarrow b$ 
end define sub

```

Okay, since some of you asked, here is the algorithm to read the numbers:

```

define sub readNumbers
    given  $a$  is the address of an array
    local  $i$  is an integer
    local  $r$  is an integer
     $i \leftarrow 0$ 
    repeat
         $r \leftarrow \text{readDec}(\&a[i])$ 
        if  $r \neq 0$  then
             $i \leftarrow i + 1$ 
        end if
    until  $(i \geq 20) \vee (r = 0)$ 
    return  $i$ 
end define sub

```

The algorithm to read a single number is as follows:

```

define sub readDec
    given  $p$  is a pointer to an integer
    local  $r$  is an integer
     $*p \leftarrow 0$ 
    repeat
         $r \leftarrow \text{readchar}$ 
        if  $(r \geq 0) \wedge (r \neq 10)$  then
             $*p \leftarrow *p \times 10 + (r - 48)$ 
        end if
    until  $(r < 0) \vee (r = 10)$ 
    if  $r < 0$  then
        return 0
    else
        return 1
    end if
end define sub

```

This means that your main program needs to do the following:

```

define sub sort
    local  $a$  is an array of 20 16-bit integers
    local  $n$  is an integer
    local  $i$  is an integer
     $n \leftarrow \text{readNumbers}(\&a)$ 
    call quicksort( $\&a$ , 0,  $n-1$ )
     $i \leftarrow 0$ 
    while  $i < n$  do
        call printdec( $a[i]$ )
        call writechar(10)
         $i \leftarrow i + 1$ 
    end while
end define sub

```

17.3 Input format

The input file is a file of decimal numbers. Each line consists of a single number. Each number is non-negative and up to 65535, and it begins with column 1. Leading zeros may or may not be present. A linefeed (newline) character follows the number immediately.

The input file may contain anywhere from 2 numbers to 20 numbers. Your program must track and handle the actual number of numbers in the file.

I suggest that you reuse code from your base conversion program.

17.4 Output format

The output file is a file of decimal numbers. You can reuse your `printdec` subroutine in project 3. Only output the correct number of integers. In other words, if the input file has 15 numbers, the output file should have exactly 15 numbers.

Numbers in the output file should be sorted in a non-decreasing order.

17.5 Constraints

As with the previous project, you are *only* allowed to use the stack for data storage. This means the array of numbers must be allocated from the stack as well.

17.6 Suggestions

17.6.1 New Directory

Use a directory for this project. Use `mkdir` to create a new directory. I suggest that you use `proj4` or `sort` as the name of the directory.

17.6.2 Partitioning the Program

You'll need to partition the program because there are multiple subroutines. I suggest the following breakdown:

- `sort.s`: make this the main program.
- `quicksort.s`: this implements the quicksort algorithm
- `pivot.s`: this implements the pivoting algorithm (also known as the partitioning algorithm)
- `printdec.s`: you can reuse this from project 3
- `readdec.s`: you can convert from a previous project. I suggest a prototype of `int16 readdec(int16 *p);`, in which `p` is a pointer to an integer (to be read), and the subroutine returns 0 if no integer is read (due to end-of-file).

17.6.3 Makefile

I am assuming that you follow the suggested partitioning from the previous section. Add the following lines to the beginning of the Makefile:

```
sortSrc := sort.s quicksort.s pivot.s printdec.s readdec.s readchar.s writechar.s
sort := sort.out
```

Then add the following to the end of the Makefile:

```
$(sort): $(sortSrc:.s=.o)
ld -o $@ $^
```

To debug the program, use the following command:

```
make sort.gdb
```

17.7 Testing

This is not a program that you want to write in a single session. I suggest that you write the subroutines in the following order.

- `readdec.s`: this is a conversion (to use stack-based local variables and etc.) from a previous project. Test it individually.
- `sort.s`: implement it so that you can at least read the input file (probably from redirection) and store numbers in an array.
- `printdec.s`: this should be the same as in project 3.
- `sort.s`: implement it so that you can print numbers from the array. Verify that the program reads and writes integers correctly.
- `pivot.s`: get the subroutine `pivot` to work. You need to modify `sort.s` to call `pivot`. Just make `sort.s` call `pivot` once and make sure the array is pivotted correctly, and that the index of the pivot is computed properly.
- `sort.s`: modify the main program again to call `quicksort` instead of `pivot`.
- `quicksort.s`: this is the recursive subroutine. Interestingly, if you have done everything correctly up to this point, this subroutine is also one of the simplest!

The key is to do incremental programming and debugging. Version control may be helpful as well. Try to apply the techniques that I discussed in the class when I demonstrated how to implement `readword` in project 3.

17.8 What to turn in?

Since it is more likely than not that you'll need multiple files, you need to put all the files into one archive before submitting it via Moodle.

Assuming you are using `proj4` as the name of the directory, and it is in your home directory, do the following:

```
cd ~
tar czf proj4.tar.gz proj4
ls -l proj4.tar.gz
```

The last command `ls` should display information (size, date, etc.) of `proj4.tar.gz`. Download it to a PC, then upload again via the Moodle interface.

Part VI

Advanced Concepts

Chapter 18

Interrupts

Interrupts are fairly advanced features. This is due to several reasons that this chapter will explain.

18.1 Rationale

18.1.1 Example: the UART without Interrupts

Many devices (implemented either on the processor chip or on supporting chipset chips) occasionally requires the attention of the processor. For example, let us consider the Universal Asynchronous Receiver Transmitter (UART). The UART is a relatively simple device that sends data using the asynchronous protocol. It is typically known as either the serial port or “COM” port on a PC. The Intel chip for this function is called the 16550.

A byte is transmitted in 10 bits because of overhead. At 115,000kbps, this takes approximately $87\mu\text{s}$. In other words, after the processor instructs the UART to transmit a byte, it takes $87\mu\text{s}$ before the UART is ready to accept another bytes.

$87\mu\text{s}$ is eternity for a power processor. At even 1GHz, a PIII processor can execute 87,000 instructions in $87\mu\text{s}$. If we have 30 bytes to transmit, how do we keep the UART “occupied” all the time?

We can do the following:

```
while there are more bytes to transmit do
  get a byte ch to transmit
  send ch to the UART
  while the UART is busy do
    do nothing
  end while
end while
```

This works, it is called busy polling. In other words, this logic cannot perform *any* other operations when it is waiting for the UART to be free again. This is a waste of processor resources.

18.1.2 Interrupts to the Rescue

Most UARTs can also interrupt. This means that an UART can *take control of the processor* when it needs to. In really technical terms, an UART *interrupts* the processor when it is done transmitting a byte, and requires the processor to tell it what to transmit next.

Look at it another way. When a hardware device, such as an UART, has a need for the processing (when it has just transmitted a byte), it can *call* a subroutine that is written just for the hardware device. This subroutine is called an Interrupt Service Routine (ISR). The exact hardware reasons for calling an ISR depends on the hardware device and how it is configured.

In general, an ISR performs the following operations, in general:

- figure out why the hardware device needs attention
- service the hardware device to eliminate the need of attention

- return

In the case of the UART, the ISR may have the following logic:

```

if transmitter is empty then
  if there is at least one more byte to transmit then
    feed the transmitter
  else
    disable the transmitter, no more to transmit
  end if
else if receiver is full then
  remove byte from receiver
  process the byte
else if error is detected then
  log the error
end if

```

So, with the ISR containing all the logic, what is the main program to do?

```

prepare bytes to transmit
set up the UART for transmission
enable interrupts

```

Yep, that's it!

But how do we know it is all done in the main program?

```

while there are bytes to transmit do
  give up control and let other programs run
end while

```

Without going into operating system concepts, it is difficult to explain how a program (in execution) gives up control so other programs can run.

18.2 ISRs and Assembly Programming

Can we write ISRs in C and not in assembly?

The answer is absolutely. For most operating systems, including Linux, ISRs are often written in C.

It is only necessary to write ISRs in assembly for small architectures that have few resources to begin with. Such small architectures are often known as microcontroller units (MCUs), and they range from less than \$1 to about \$10.

On some of these architectures, the stack is only two call-levels deep, and there is no RAM to speak of. For such architectures, assembly programming is the only option.