

# **Introduction to Computer Organization**

**with x86-64 Assembly Language  
& GNU/Linux**

**Robert G. Plantz**

# Introduction to Computer Organization

## with x86-64 Assembly Language & GNU/Linux

Robert G. Plantz  
Sonoma State University  
`bob.cs.sonoma.edu`

September 2012

# Copyright Notice

Copyright ©2008, ©2009, ©2010, ©2011, ©2012 by Robert G. Plantz. All rights reserved.

The author has used his best efforts in preparing this book. The author makes no warranty of any kind, expressed or implied, with regard to the programs or the documentation contained in this book. The author shall not be liable in any event from incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

All products or services mentioned in this book are the trademarks or service marks of their respective companies or organizations. Eclipse is a trademark of Eclipse Foundation, Inc.

## Making Copies of This File

This is the electronic version of the book. I have not used DRM and am selling it through channels that do not require copy protection so that you may make multiple copies *for your own personal use*. You may also print all or portions of the book *for your own personal use*.

I believe that my pricing for the book is fair to you while providing me with some payment for my work. Please honor my work by not providing copies to others. They can visit my website and download a preview version to determine if they wish to buy their own copy.

Electronic reading is a new paradigm, and I hope we can find ways to sell electronic copies that are fair to the reader, the author, and the distribution channels. I welcome your thoughts about my approach to selling this book.

# Contents

<b>Preface</b>	<b>xiv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Computer Subsystems . . . . .	3
1.2 How the Subsystems Interact . . . . .	4
<b>2 Data Storage Formats</b>	<b>6</b>
2.1 Bits and Groups of Bits . . . . .	6
2.2 Mathematical Equivalence of Binary and Decimal . . . . .	8
2.3 Unsigned Decimal to Binary Conversion . . . . .	9
2.4 Memory — A Place to Store Data (and Other Things) . . . . .	11
2.5 Using C Programs to Explore Data Formats . . . . .	13
2.6 Examining Memory With <code>gdb</code> . . . . .	17
2.7 ASCII Character Code . . . . .	21
2.8 <code>write</code> and <code>read</code> Functions . . . . .	23
2.9 Exercises . . . . .	26
<b>3 Computer Arithmetic</b>	<b>30</b>
3.1 Addition and Subtraction . . . . .	30
3.2 Arithmetic Errors — Unsigned Integers . . . . .	36
3.3 Arithmetic Errors — Signed Integers . . . . .	37
3.4 Overflow and Signed Decimal Integers . . . . .	42
3.4.1 The Meaning of CF and OF . . . . .	45
3.5 C/C++ Basic Data Types . . . . .	48
3.5.1 C/C++ Shift Operations . . . . .	50
3.5.2 C/C++ Bit Operations . . . . .	51
3.5.3 C/C++ Data Type Conversions . . . . .	53
3.6 Other Codes . . . . .	55
3.6.1 BCD Code . . . . .	55
3.6.2 Gray Code . . . . .	56
3.7 Exercises . . . . .	58
<b>4 Logic Gates</b>	<b>61</b>
4.1 Boolean Algebra . . . . .	61
4.2 Canonical (Standard) Forms . . . . .	65
4.3 Boolean Function Minimization . . . . .	67
4.3.1 Minimization Using Algebraic Manipulations . . . . .	68
4.3.2 Minimization Using Graphic Tools . . . . .	70
4.4 Crash Course in Electronics . . . . .	77
4.4.1 Power Supplies and Batteries . . . . .	78
4.4.2 Resistors, Capacitors, and Inductors . . . . .	78

4.4.3 CMOS Transistors . . . . .	83
4.5 NAND and NOR Gates . . . . .	86
4.6 Exercises . . . . .	89
<b>5 Logic Circuits</b>	<b>91</b>
5.1 Combinational Logic Circuits . . . . .	91
5.1.1 Adder Circuits . . . . .	92
5.1.2 Ripple-Carry Addition/Subtraction Circuits . . . . .	94
5.1.3 Decoders . . . . .	95
5.1.4 Multiplexers . . . . .	99
5.2 Programmable Logic Devices . . . . .	100
5.2.1 Programmable Logic Array (PLA) . . . . .	101
5.2.2 Read Only Memory (ROM) . . . . .	102
5.2.3 Programmable Array Logic (PAL) . . . . .	103
5.3 Sequential Logic Circuits . . . . .	103
5.3.1 Clock Pulses . . . . .	105
5.3.2 Latches . . . . .	106
5.3.3 Flip-Flops . . . . .	111
5.4 Designing Sequential Circuits . . . . .	115
5.5 Memory Organization . . . . .	121
5.5.1 Registers . . . . .	121
5.5.2 Shift Registers . . . . .	122
5.5.3 Static Random Access Memory (SRAM) . . . . .	124
5.5.4 Dynamic Random Access Memory (DRAM) . . . . .	126
5.6 Exercises . . . . .	128
<b>6 Central Processing Unit</b>	<b>129</b>
6.1 CPU Overview . . . . .	130
6.2 CPU Registers . . . . .	131
6.3 CPU Interaction with Memory and I/O . . . . .	135
6.4 Program Execution in the CPU . . . . .	136
6.5 Using gdb to View the CPU Registers . . . . .	139
6.6 Exercises . . . . .	145
<b>7 Programming in Assembly Language</b>	<b>147</b>
7.1 Creating a New Program . . . . .	147
7.2 Program Organization . . . . .	148
7.2.1 First instructions . . . . .	156
7.2.2 A Note About Syntax . . . . .	157
7.2.3 The Additional Assembly Language Generated by the Compiler . . . . .	158
7.2.4 Viewing Both the Assembly Language and C Source Code . . . . .	160
7.2.5 Minimum Program in 32-bit Mode . . . . .	162
7.3 Assemblers and Linkers . . . . .	163
7.3.1 Assemblers . . . . .	163
7.3.2 Linkers . . . . .	165
7.4 Creating a Program in Assembly Language . . . . .	166
7.5 Instructions Introduced Thus Far . . . . .	168
7.5.1 Instructions . . . . .	168
7.6 Exercises . . . . .	169

<b>8 Program Data - Input, Store, Output</b>	<b>171</b>
8.1 Calling <code>write</code> in 64-bit Mode . . . . .	171
8.2 Introduction to the Call Stack . . . . .	176
8.3 Viewing the Call Stack . . . . .	183
8.4 Local Variables on the Call Stack . . . . .	188
8.4.1 Calling <code>printf</code> and <code>scanf</code> in 64-bit Mode . . . . .	195
8.5 Designing the Local Variable Portion of the Call Stack . . . . .	197
8.6 Using <code>syscall</code> to Perform I/O . . . . .	201
8.7 Calling Functions, 32-Bit Mode . . . . .	204
8.8 Instructions Introduced Thus Far . . . . .	205
8.8.1 Instructions . . . . .	206
8.8.2 Addressing Modes . . . . .	206
8.9 Exercises . . . . .	207
<b>9 Computer Operations</b>	<b>208</b>
9.1 The Assignment Operator . . . . .	208
9.2 Addition and Subtraction Operators . . . . .	214
9.3 Introduction to Machine Code . . . . .	220
9.3.1 Assembler Listings . . . . .	222
9.3.2 General Format of Instructions . . . . .	224
9.3.3 REX Prefix Byte . . . . .	225
9.3.4 ModRM Byte . . . . .	226
9.3.5 SIB Byte . . . . .	226
9.3.6 The <code>mov</code> Instruction . . . . .	227
9.3.7 The <code>add</code> Instruction . . . . .	229
9.4 Instructions Introduced Thus Far . . . . .	230
9.4.1 Instructions . . . . .	231
9.4.2 Addressing Modes . . . . .	231
9.5 Exercises . . . . .	232
<b>10 Program Flow Constructs</b>	<b>235</b>
10.1 Repetition . . . . .	235
10.1.1 Comparison Instructions . . . . .	237
10.1.2 Conditional Jumps . . . . .	238
10.1.3 Unconditional Jump . . . . .	241
10.1.4 <code>while</code> Loop . . . . .	242
10.2 Binary Decisions . . . . .	250
10.2.1 Short-Circuit Evaluation . . . . .	259
10.2.2 Conditional Move . . . . .	259
10.3 Instructions Introduced Thus Far . . . . .	261
10.3.1 Instructions . . . . .	261
10.3.2 Addressing Modes . . . . .	262
10.4 Exercises . . . . .	262
<b>11 Writing Your Own Functions</b>	<b>266</b>
11.1 Overview of Passing Arguments . . . . .	266
11.2 More Than Six Arguments, 64-Bit Mode . . . . .	273
11.3 Interface Between Functions, 32-Bit Mode . . . . .	283
11.4 Instructions Introduced Thus Far . . . . .	286
11.4.1 Instructions . . . . .	287
11.4.2 Addressing Modes . . . . .	288
11.5 Exercises . . . . .	288

<b>12 Bit Operations; Multiplication and Division</b>	<b>290</b>
12.1 Logical Operators . . . . .	290
12.2 Shifting Bits . . . . .	301
12.3 Multiplication . . . . .	307
12.4 Division . . . . .	315
12.5 Negating Signed ints . . . . .	322
12.6 Instructions Introduced Thus Far . . . . .	322
12.6.1 Instructions . . . . .	322
12.6.2 Addressing Modes . . . . .	324
12.7 Exercises . . . . .	324
<b>13 Data Structures</b>	<b>327</b>
13.1 Arrays . . . . .	327
13.2 structs (Records) . . . . .	333
13.3 structs as Function Arguments . . . . .	337
13.4 Structs as C++ Objects . . . . .	343
13.5 Instructions Introduced Thus Far . . . . .	353
13.5.1 Instructions . . . . .	353
13.5.2 Addressing Modes . . . . .	355
13.6 Exercises . . . . .	355
<b>14 Fractional Numbers</b>	<b>358</b>
14.1 Fractions in Binary . . . . .	358
14.2 Fixed Point ints . . . . .	359
14.3 Floating Point Format . . . . .	360
14.4 IEEE 754 . . . . .	363
14.5 Floating Point Hardware . . . . .	365
14.5.1 SSE2 Floating Point . . . . .	366
14.5.2 x87 Floating Point Unit . . . . .	371
14.5.3 3DNow! Floating Point . . . . .	376
14.6 Comments About Numerical Accuracy . . . . .	376
14.7 Instructions Introduced Thus Far . . . . .	377
14.7.1 Instructions . . . . .	377
14.7.2 Addressing Modes . . . . .	379
14.8 Exercises . . . . .	379
<b>15 Interrupts and Exceptions</b>	<b>384</b>
15.1 Hardware Interrupts . . . . .	385
15.2 Exceptions . . . . .	386
15.3 Software Interrupts . . . . .	386
15.4 CPU Response to an Interrupt or Exception . . . . .	387
15.5 Return from Interrupt/Exception . . . . .	388
15.6 The <code>syscall</code> and <code>sysret</code> Instructions . . . . .	388
15.7 Summary . . . . .	391
15.8 Instructions Introduced Thus Far . . . . .	391
15.8.1 Instructions . . . . .	391
15.8.2 Addressing Modes . . . . .	394
15.9 Exercises . . . . .	394

<b>16 Input/Output</b>	<b>395</b>
16.1 Memory Timing . . . . .	395
16.2 I/O Device Timing . . . . .	396
16.3 Bus Timing . . . . .	396
16.4 I/O Interfacing . . . . .	397
16.5 I/O Ports . . . . .	398
16.6 Programming Issues . . . . .	399
16.7 Interrupt-Driven I/O . . . . .	410
16.8 I/O Instructions . . . . .	410
16.9 Exercises . . . . .	411
<b>A Reference Material</b>	<b>412</b>
A.1 Basic Logic Gates . . . . .	412
A.2 Register Names . . . . .	413
A.3 Argument Order in Registers . . . . .	413
A.4 Register Usage . . . . .	414
A.5 Assembly Language Instructions Used in This Book . . . . .	414
A.6 Addressing Modes . . . . .	417
<b>B Using GNU make to Build Programs</b>	<b>418</b>
<b>C Using the gdb Debugger for Assembly Language</b>	<b>424</b>
<b>D Embedding Assembly Code in a C Function</b>	<b>430</b>
<b>E Exercise Solutions</b>	<b>435</b>
E.2 Data Storage Formats . . . . .	435
E.3 Computer Arithmetic . . . . .	443
E.4 Logic Gates . . . . .	454
E.5 Logic Circuits . . . . .	457
E.6 Central Processing Unit . . . . .	458
E.7 Programming in Assembly Language . . . . .	460
E.8 Program Data - Input, Store, Output . . . . .	465
E.9 Computer Operations . . . . .	468
E.10 Program Flow Constructs . . . . .	476
E.11 Writing Your Own Functions . . . . .	488
E.12 Bit Operations; Multiplication and Division . . . . .	495
E.13 Data Structures . . . . .	510
E.14 Fractional Numbers . . . . .	534
E.15 Interrupts and Exceptions . . . . .	538
<b>Bibliography</b>	<b>540</b>
<b>Index</b>	<b>541</b>

# List of Figures

1.1 Subsystems of a computer. . . . .	3
2.1 Possible contents of the first sixteen bytes of memory . . . . .	11
2.2 Repeat of Figure 2.1 with contents shown in hex. . . . .	12
2.3 A text string stored in memory . . . . .	23
3.1 “Decoder Ring” for three-bit signed and unsigned integers. . . . .	46
3.2 Relationship of I/O libraries to application and operating system. . . . .	49
3.3 Truth table for adding two bits with carry from a previous bit addition. . .	52
3.4 Truth tables showing bitwise C/C++ operations. . . . .	52
3.5 Truth tables showing C/C++ logical operations. . . . .	53
4.1 The AND gate acting on two variables, $x$ and $y$ . . . . .	61
4.2 The OR gate acting on two variables, $x$ and $y$ . . . . .	62
4.3 The NOT gate acting on one variable, $x$ . . . . .	62
4.4 Hardware implementation of the function in Equation 4.22. . . . .	68
4.5 Hardware implementation of the function in Equation 4.28. . . . .	69
4.6 Mapping of two-variable minterms on a Karnaugh map. . . . .	70
4.7 Karnaugh map for $F_1(x, y) = x \cdot y' + x' \cdot y + x \cdot y$ . . . . .	71
4.8 A Karnaugh map grouping showing that $F_1(x, y) = x \cdot y' + x' \cdot y + x \cdot y = x \cdot y' + y$ . .	71
4.9 Two-variable Karnaugh map showing the groupings $x$ and $y$ . . . . .	72
4.10 Mapping of three-variable minterms on a Karnaugh map. . . . .	72
4.11 Mapping of four-variable minterms on a Karnaugh map. . . . .	72
4.12 Comparison of one minterm (a) versus one maxterm (b) on a Karnaugh map. .	75
4.13 Mapping of three-variable maxterms on a Karnaugh map. . . . .	75
4.14 Mapping of four-variable minterms on a Karnaugh map. . . . .	76
4.15 The XOR gate acting on two variables, $x$ and $y$ . . . . .	77
4.16A “don’t care” cell on a Karnaugh map. . . . .	77
4.17 Karnaugh map for $\text{xor}$ function if we know $x = y = 1$ cannot occur. . . . .	77
4.18 AC/DC power supply. . . . .	78
4.19 Two resistors in series. . . . .	79
4.20 Two resistors in parallel. . . . .	80
4.21 Capacitor in series with a resistor. . . . .	81
4.22 Capacitor charging over time. . . . .	82
4.23 Inductor in series with a resistor. . . . .	82
4.24 Inductor building a magnetic field over time. . . . .	83
4.25 A single n-type MOSFET transistor switch. . . . .	84
4.26 Single transistor switch equivalent circuit. . . . .	84
4.27 CMOS inverter (NOT) circuit. . . . .	85
4.28 CMOS inverter equivalent circuit. . . . .	86
4.29 CMOS AND circuit. . . . .	86

4.30 The NAND gate acting on two variables, $x$ and $y$ . . . . .	87
4.31 The NOR gate acting on two variables, $x$ and $y$ . . . . .	87
4.32 An alternate way to draw a NAND gate. . . . .	87
4.33 A NOT gate built from a NAND gate. . . . .	88
4.34 An AND gate built from two NAND gates. . . . .	88
4.35 An OR gate built from three NAND gates. . . . .	88
4.36 The function in Equation 4.55 using two AND gates and one OR gate. . . . .	88
4.37 The function in Equation 4.55 using two AND gates, one OR gate and four NOT gates. . . . .	89
4.38 The function in Equation 4.55 using only three NAND gates. . . . .	89
5.1 An adder circuit. . . . .	93
5.2 A half adder circuit. . . . .	93
5.3 Full adder using two half adders. . . . .	94
5.4 Four-bit adder. . . . .	95
5.5 Four-bit adder/subtractor. . . . .	96
5.6 Circuit for a $3 \times 8$ decoder with enable. . . . .	98
5.7 Full adder implemented with $3 \times 8$ decoder. . . . .	98
5.8 A 2-way multiplexer. . . . .	99
5.9 A 4-way multiplexer. . . . .	100
5.10 Symbol for a 4-way multiplexer. . . . .	100
5.11 Simplified circuit for a programmable logic array. . . . .	101
5.12 Programmable logic array schematic. . . . .	102
5.13 Eight-byte Read Only Memory (ROM). . . . .	103
5.14 Two-function Programmable Array Logic (PAL). . . . .	104
5.15 Clock signals. . . . .	105
5.16 NOR gate implementation of an SR latch. . . . .	106
5.17 State diagram for an SR latch. . . . .	107
5.18 NAND gate implementation of an S'R' latch. . . . .	108
5.19 State table and state diagram for an S'R' latch. . . . .	109
5.20 SR latch with <i>Control</i> input. . . . .	109
5.21 D latch constructed from an SR latch. . . . .	110
5.22 D flip-flop, positive-edge triggering. . . . .	111
5.23 D flip-flop, positive-edge triggering with asynchronous preset. . . . .	112
5.24 Symbols for D flip-flops. . . . .	112
5.25 T flip-flop state table and state diagram. . . . .	113
5.26 T flip-flop. . . . .	113
5.27 JK flip-flop state table and state diagram. . . . .	114
5.28 JK flip-flop. . . . .	115
5.29 A 4-bit register. . . . .	122
5.30 A 4-bit register with load. . . . .	123
5.31 8-way mux to select output of register file. . . . .	123
5.32 Four-bit serial-to-parallel shift register. . . . .	124
5.33 Tri-state buffer. . . . .	125
5.34 Four-way multiplexer built from tri-state buffers. . . . .	125
5.35 4-bit memory cell. . . . .	126
5.36 Addressing 1 MB of memory with one $20 \times 2^{20}$ address decoder. . . . .	127
5.37 Addressing 1 MB of memory with two $10 \times 2^{10}$ address decoders. . . . .	127
5.38 Bit storage in DRAM. . . . .	128
6.1 CPU block diagram. . . . .	130
6.2 Graphical representation of general purpose registers. . . . .	133

6.3 Condition codes portion of the rflags register. . . . .	135
6.4 Subsystems of a computer. . . . .	136
6.5 The instruction execution cycle. . . . .	138
7.1 Screen shot of the creation of a program in assembly language. . . . .	167
8.1 The stack in Listing 8.3 when it is first initialized. . . . .	179
8.2 The stack with one data item on it. . . . .	180
8.3 The stack with three data items on it. . . . .	180
8.4 The stack after all three data items have been popped off. . . . .	181
8.5 Local variables in the program from Listing 8.5 are allocated on the stack. . . . .	190
8.6 Local variable stack area in the program from Listing 8.5. . . . .	192
9.1 Assembler listing file for the function shown in Listing 9.7. . . . .	223
9.2 General format of instructions. . . . .	225
9.3 REX prefix byte. . . . .	225
9.4 ModRM byte. . . . .	226
9.5 SIB byte. . . . .	226
9.6 Machine code for the mov from a register to a register instruction. . . . .	227
9.7 Machine code for the mov immediate data to a register instruction. . . . .	228
9.8 Machine code for the add immediate data to the A register . . . . .	229
9.9 Machine code for the add immediate data to a register . . . . .	229
9.10 Machine code for the add immediate data to a register instruction. . . . .	230
9.11 Machine code for the add register to register instruction. . . . .	230
10.1 Flow chart of a while loop. . . . .	237
10.2 Flow chart of if-else construct. . . . .	252
11.1 Arguments and local variables in the stack frame, sumInts function. . . . .	271
11.2 Arguments 7 - 9 are passed on the stack to the sumNine function. . . . .	276
11.3 Arguments and local variables in the stack frame, sumNine function. . . . .	277
11.4 Overall layout of the stack frame. . . . .	282
11.5 Calling function's stack frame, 32-bit mode. . . . .	286
13.1 Memory allocation for the variables x and y from the C program in Listing 13.6. . . . .	335
14.1 IEEE 754 bit patterns. . . . .	363
14.2 x87 floating point register stack. . . . .	372
16.1 Typical bus controllers in a modern PC. . . . .	397

# List of Tables

2.1 Hexadecimal representation of four bits. . . . .	7
2.2 C/C++ syntax for specifying literal numbers. . . . .	8
2.3 ASCII code for representing characters. . . . .	22
3.1 Correspondence between binary, hexadecimal, and unsigned decimal values for the hexadecimal digits. . . . .	33
3.2 Four-bit signed integers, two's complement notation. . . . .	38
3.3 Sizes of some C/C++ data types in 32-bit and 64-bit modes. . . . .	48
3.4 Hexadecimal characters and corresponding int. . . . .	54
3.5 BCD code for the decimal digits. . . . .	55
3.6 Sign codes for packed BCD. . . . .	56
3.7 Gray code for 4 bits. . . . .	57
4.1 Minterms for three variables. . . . .	66
4.2 Maxterms for three variables. . . . .	67
5.1 BCD decoder. . . . .	96
5.2 Truth table for a $3 \times 8$ decoder with <i>enable</i> . . . . .	97
5.3 NOR-based SR latch state table. . . . .	107
5.4 SR latch with Control state table. . . . .	110
5.5 D latch with Control state table. . . . .	110
5.6 T flip-flop state table with D flip-flop inputs. . . . .	113
5.7 JK flip-flop state table with D flip-flop inputs. . . . .	114
6.1 X86-64 operating modes. . . . .	129
6.2 The x86-64 registers. . . . .	132
6.3 Assembly language names for portions of the general-purpose CPU registers. . . . .	133
6.4 General purpose registers. . . . .	134
7.1 Effect on other bits in a register when less than 64 bits are changed. . . . .	157
8.1 Common assembler directives for allocating memory. . . . .	173
8.2 Order of passing arguments in general purpose registers. . . . .	174
8.3 Register set up for using <i>syscall</i> instruction to read, write, or exit. . . . .	202
9.1 Walking through the code in Listing 9.4. . . . .	219
9.2 The mm field in the ModRM byte. . . . .	226
9.3 Machine code of general purpose registers. . . . .	227
10.1 Conditional jump instructions. . . . .	239
10.2 Conditional jump instructions for unsigned values. . . . .	239
10.3 Conditional jump instructions for signed values. . . . .	240

10.4 Machine code for the <code>je</code> instruction. . . . .	241
11.1 Argument register save area in stack frame. . . . .	271
12.1 Bit patterns (in binary) of the ASCII numerals and the corresponding 32-bit ints. . . . .	308
12.2 Register usage for the <code>mul</code> instruction. . . . .	309
12.3 Register usage for the <code>div</code> instruction. . . . .	315
14.1 MXCSR status register. . . . .	367
14.2 SSE scalar floating point conversion instructions. . . . .	368
14.3 Some SSE floating point arithmetic and data movement instructions. . . . .	368
14.4 x87 Status Word. . . . .	372
14.5 A sampling of x87 floating point instructions. . . . .	373
15.1 Some system call codes for the <code>syscall</code> instruction. . . . .	390

# Listings

2.1	Using <code>printf</code> to display numbers. . . . .	14
2.2	C program showing the mathematical equivalence of the decimal and hexadecimal number systems. . . . .	15
2.3	Displaying a single character using C. . . . .	24
2.4	Echoing characters entered from the keyboard. . . . .	24
3.1	Shifting to multiply and divide by powers of two. . . . .	51
3.2	Reading hexadecimal values from keyboard. . . . .	54
6.1	Simple program to illustrate the use of <code>gdb</code> to view CPU registers. . . . .	140
7.1	A “null” program (C). . . . .	149
7.2	A “null” program ( <code>gcc</code> assembly language). . . . .	149
7.3	A “null” program (programmer assembly language). . . . .	150
7.4	A “null” program ( <code>gcc</code> assembly language without exception handler frame). . . . .	159
7.5	The “null” program rewritten to show a label placed on its own line. . . . .	160
7.6	Assembly language embedded in C source code listing. . . . .	160
7.7	A “null” program ( <code>gcc</code> assembly language in 32-bit mode). . . . .	162
7.8	A “null” program (programmer assembly language in 32-bit mode). . . . .	162
8.1	“Hello world” program using the <code>write</code> system call function (C). . . . .	172
8.2	“Hello world” program using the <code>write</code> system call function ( <code>gcc</code> assembly language). . . . .	172
8.3	A C implementation of a stack. . . . .	177
8.4	Save and restore the contents of the <code>rbx</code> and <code>r12 - r15</code> registers. . . . .	182
8.5	Echoing characters entered from the keyboard ( <code>gcc</code> assembly language). . . . .	189
8.6	Echoing characters entered from the keyboard (programmer assembly language). . . . .	193
8.7	Calling <code>printf</code> and <code>scanf</code> to write and read formatted I/O (C). . . . .	195
8.8	Calling <code>printf</code> and <code>scanf</code> to write and read formatted I/O ( <code>gcc</code> assembly language). . . . .	195
8.9	Calling <code>printf</code> and <code>scanf</code> to write and read formatted I/O (programmer assembly language). . . . .	196
8.10	Some local variables (C). . . . .	198
8.11	Some local variables ( <code>gcc</code> assembly language). . . . .	198
8.12	Some local variables (programmer assembly language). . . . .	199
8.13	General format of a function written in assembly language. . . . .	201
8.14	Echo character program using the <code>syscall</code> instruction. . . . .	202
8.15	Displaying four characters on the screen using the <code>write</code> system call function in assembly language. . . . .	204
9.1	Assignment to a register variable (C). . . . .	209
9.2	Assignment to a register variable ( <code>gcc</code> assembly language). . . . .	210
9.3	Assignment to a register variable (programmer assembly language). . . . .	211
9.4	Addition and subtraction (C). . . . .	217
9.5	Addition and subtraction ( <code>gcc</code> assembly language). . . . .	217

9.6	Addition and subtraction (programmer assembly language) . . . . .	219
9.7	Some instructions for us to assemble. . . . .	222
10.1	Displaying a string one character at a time (C). . . . .	235
10.2	Unconditional jumps. . . . .	241
10.3	Displaying a string one character at a time (gcc assembly language). . . . .	243
10.4	General structure of a count-controlled while loop. . . . .	246
10.5	Displaying a string one character at a time (programmer assembly language). . . . .	246
10.6	A do-while loop to print 10 characters. . . . .	249
10.7	Get yes/no response from user (C). . . . .	250
10.8	Get yes/no response from user (gcc assembly language). . . . .	251
10.9	General structure of an if-else construct. . . . .	253
10.10	Get yes/no response from user (programmer assembly language). . . . .	254
10.11	Compound boolean expression in an if-else construct (C). . . . .	256
10.12	Compound boolean expression in an if-else construct (gcc assembly language). . . . .	257
10.13	Simple for loop to perform multiplication. . . . .	263
11.1	Passing arguments to a function (C). . . . .	268
11.2	Accessing arguments in the sumInts function from Listing 11.1 (gcc assembly language). . . . .	269
11.3	Accessing arguments in the sumInts function from Listing 11.1 (programmer assembly language) . . . . .	272
11.4	Passing more than six arguments to a function (C). . . . .	273
11.5	Passing more than six arguments to a function (gcc assembly language). . . . .	275
11.6	Passing more than six arguments to a function (programmer assembly language). . . . .	280
11.7	Passing more than six arguments to a function (gcc assembly language, 32-bit). . . . .	284
12.1	Convert letters to upper/lower case (C). . . . .	293
12.2	Convert letters to upper/lower case (gcc assembly language). . . . .	294
12.3	Convert letters to upper/lower case (programmer assembly language). . . . .	299
12.4	Shifting bits (C). . . . .	303
12.5	Shifting bits (gcc assembly language). . . . .	304
12.6	Shifting bits (programmer assembly language). . . . .	306
12.7	Convert decimal text string to int (C). . . . .	312
12.8	Convert decimal text string to int (gcc assembly language). . . . .	312
12.9	Convert decimal text string to int (programmer assembly language). . . . .	313
12.10	Convert unsigned int to decimal text string (C). . . . .	318
12.11	Convert unsigned int to decimal text string (gcc assembly language). . . . .	319
12.12	Convert unsigned int to decimal text string (programmer assembly language). . . . .	320
13.1	Storing a value in one element of an array (C). . . . .	327
13.2	Storing a value in one element of an array (gcc assembly language). . . . .	328
13.3	Clear an array (C). . . . .	329
13.4	Clear an array (gcc assembly language). . . . .	330
13.5	Clear an array (programmer assembly language). . . . .	331
13.6	Two struct variables (C). . . . .	333
13.7	Two struct variables (gcc assembly language). . . . .	335
13.8	Two struct variables (programmer assembly language). . . . .	336
13.9	Passing struct variables (C). . . . .	339
13.10	Passing struct variables (gcc assembly language). . . . .	340
13.11	Passing struct variables — assembly language version. . . . .	342
13.12	Add 1 to user's fraction (C++) . . . . .	344

---

13.13	Add 1 to user's fraction (C). . . . .	348
13.14	Add 1 to user's fraction (programmer assembly language). . . . .	352
14.1	Fixed point addition. . . . .	359
14.2	Converting a fraction to a float. . . . .	369
14.3	Converting a fraction to a float (gcc assembly language, 64-bit). . . . .	369
14.4	Converting a fraction to a float (gcc assembly language, 32-bit). . . . .	374
14.5	Use float for Loop Control Variable? . . . . .	380
14.6	Are floats accurate? . . . . .	380
14.7	Casting integer to float in C. . . . .	381
14.8	Casting integer to float in assembly language. . . . .	382
15.1	Using <code>syscall</code> to cat a file. . . . .	388
16.1	Sketch of basic I/O functions using memory-mapped I/O — C version. . . . .	399
16.2	Memory-mapped I/O in assembly language. . . . .	402
16.3	Sketch of basic I/O functions, isolated I/O — C version. . . . .	405
16.4	Isolated I/O in assembly language. . . . .	407
B.1	An example of a Makefile for an assembly language program with one source file. . . . .	419
B.2	An example of a Makefile for a program with both C and assembly language source files. . . . .	420
B.3	Makefile variables. . . . .	420
B.4	Incomplete Makefile. . . . .	422
D.1	Embedding an assembly language instruction in a C function (C). . . . .	430
D.2	Embedding an assembly language instruction in a C function gcc assembly language. . . . .	431
D.3	Embedding more than one assembly language instruction in a C function and specifying a register (C). . . . .	432
D.4	Embedding more than one assembly language instruction in a C function and specifying a register (gcc assembly language). . . . .	433

# Preface

This book introduces the concepts of how computer hardware works from a programmer's point of view. A programmer's job is to design a sequence of instructions that will cause the hardware to perform operations that solve a problem. This book looks at these instructions by exploring how C/C++ language constructs are implemented at the instruction set architecture level.

The specific architecture presented in this book is the x86-64 that has evolved over the years from the Intel 8086 processor. The GNU programming environment is used, and the operating system kernel is Linux.

The basic guidelines I followed in creating this book are:

- One should avoid writing in assembly language except when absolutely necessary.
- Learning is easier if it builds upon concepts you already know.
- "Real world" hardware and software make a more interesting platform for learning theoretical concepts.
- The tools used for teaching should be inexpensive and readily available.

It may seem strange that I would recommend against assembly language programming in a book largely devoted to the subject. Well, C was introduced in 1978 specifically for low-level programming. C code is much easier to write and to maintain than assembly language. C compilers have evolved to a point where they produce better machine code than all but the best assembly language programmers can. In addition, the hardware technology has increased such that there is seldom any significant advantage in writing the most efficient machine code. In short, it is hardly ever worth the effort to write in assembly language.

You might well ask why you should study assembly language, given that I think you should avoid writing in it. I believe very strongly that the best programmers have a good understanding of how computer hardware works. I think this principle holds in most fields: the best drivers understand how automobiles work; the best musicians understand how their instrument works; etc.

So this is *not* a book on how to write programs in assembly language. Most of the programs you will be asked to write will be in assembly language, but they are very simple programs intended to illustrate the concepts. I believe that this book will help you to become a better programmer in any programming language, even if you never write another line of assembly language.

Two issues arise immediately when studying assembly language:

- I/O interaction with a user through even the keyboard and screen is a very complex problem, well beyond the programming expertise of a beginner.
- There is an almost endless variety of instructions that can be used.

There are several ways to deal with these problems in a textbook. Some books use a simple operating system for I/O, e.g., MS-DOS. Others provide libraries of I/O functions that are specific for the examples in the book. Several textbooks deal with the instruction set issue by presenting a simplified “idealized” architecture with a small number of instructions that is intended to illustrate the concepts.

In keeping with the “real world” criterion of this book, it deals with these two issues by:

1. showing you how to call the I/O functions already available in the C Standard Library, and
2. presenting only a small subset of the available instructions.

This has the additional advantage of not requiring additional software to be installed. In general, all the programming discussed in the book can be done on any of the common Linux distributions that have been set up for software development with few or no changes.

Readers who wish to write assembly language programs that do not use the C runtime environment should read Sections 8.6 (page 201) and 15.6 (page 388).

If you do decide to write more complex programs in assembly language there are several other excellent books on that topic; see the Bibliography on page 540. And, of course, you would want the manufacturer’s programming manuals; see for example [2] – [6] and [14] – [18]. The goal here is to provide you with an introductory “look under the hood” of a high-level language at the hardware that lies below.

This book also provides an introduction to computer hardware architecture. The view is from a programmer’s eye. Other excellent books provide implementation details. You need to understand many of the implementation details, e.g., pipelining, caches, in order to write highly optimized programs. This book provides the introduction that prepares you for learning about more advanced architectural concepts.

This is not the place to argue about operating systems. I could rationalize my choice of GNU/Linux, but I could also rationalize using others. Therefore, I will simply state that I believe that GNU/Linux provides an excellent environment for studying programming in an academic setting. One of the more important features of the GNU programming environment with respect to the goals of this book is the close integration of C/C++ and assembly language. In addition, I like GNU/Linux.

I wish to comment on my use of “GNU/Linux” instead of the simpler “Linux.” Much has been written about these names. A good source of the various arguments can be found at [www.wikipedia.org](http://www.wikipedia.org). The two main points are that (a) Linux is only the kernel, and (b) all general-purpose distributions rely on many GNU components for the remaining systems software. Although “Linux” has become essentially a synonym for “GNU/Linux,” this book could not exist without the GNU components, e.g., the assembler (`as`), the link editor (`ld`), the make program, etc. Therefore, I wish to acknowledge the importance of the GNU project by using the full “GNU/Linux” name.

In some ways, the x86-64 instruction set architecture is not the best choice for studying computer architecture. It maintains backwards compatibility and is thus somewhat more complicated at the instruction set level. However, it is by far the most widely deployed architecture on the desktop and one of the least expensive way to set up a system where these concepts can be studied.

Assembly language is my favorite subject in computer science, but I have taught the subject to enough students to know that, realistically, it probably will not be the same for you. However, please keep your eye on the long term. I am confident that material presented in this book will help you to become a better programmer, and if you do enjoy assembly language, you will have a good introduction to a more advanced study of it.

## Assumed Background

You should have taken an introductory class in programming, preferably in C, C++, or Java. The high-level language used in this book is C, however all the C programming is simple. I am confident that the C programming examples in Chapters 2 and 3 will provide sufficient C programming concepts to make the rest of the book very usable, regardless of the language you learned in your introductory class.

I believe that more experienced programmers who wish to write for the x86-64 architecture can also benefit from reading this book. In principle, these programmers can learn everything they need to know from reading the appropriate manuals. However, I have found that it is usually helpful to have an overview of a new architecture before tackling the manuals. This book should provide that overview. In this sense, I believe that this book can provide a good “introduction” to using the manuals.

## Additional Resources

I maintain additional resources related to this book, including an errata, on my website, [bob.cs.sonoma.edu](http://bob.cs.sonoma.edu). I welcome your feedback ([plantz@sonoma.edu](mailto:plantz@sonoma.edu)), especially any errors or confusing writing that you see in the book. I use such feedback, mostly from students, to constantly improve the book.

## Learning from this Book

This book is intended for a one-semester, four unit course. Our course format at Sonoma State University consists of three hours of lecture and a two - three hour supervised lab session per week. Many of the exercises in each chapter provide good in-lab exercises for supervised labs.

Solutions to almost all the chapter exercises are provided in Appendix E. Students should attempt to solve an exercise *before* looking at the answer for hints. But I think it helps the learning process if a student can see a solution while attempting his or her own solution.

If you have an electronic copy of this book, do *not* copy and paste code. Think about it — typing in the code forces you to read every single character. Yes, it is very tedious, but you will learn much more this way. I’m assuming here that your goal is to learn the material, not simply to get the example programs to work. They are rather silly programs, so just getting them to work is not of much use.

## Development Environment

Most developers use an Integrated Development Environment (IDE), which hides the process of building a program from source code. In this book we use the component programs individually so that you can see what is taking place.

The examples in this book were compiled or assembled on a computer running Ubuntu 12.04. The development programs used were:

- gcc version 4.7.0
- as version 2.22

In most cases compilation was done with no optimization (-O0) because the goal is to study concepts, not create the most efficient code.

The examples should work in any x86\_64 GNU development environment with gcc and as (binutils) installed. However, the machine code generated by the compiler may differ depending on its specific configuration and version. You will begin looking at compiler-generated assembly language in Chapter 7. What you see in your environment may differ from the examples in this book, but the differences should be consistent as you continue through the rest of the book.

You should also keep in mind that the programs used for development may have bugs. Yes, nobody is perfect. For example, when I upgraded my Ubuntu system from 9.04 to 9.10, the GNU assembler was upgraded from 2.19 to 2.20. The newer version had a bug that caused the line numbering in a particular listing file to start from 0 instead of 1. (It affected the C source code in Listing 7.6 on page 160; the numbers have been corrected in this listing.) Fortunately, this bug did not affect the quality of the final program, but it could cause some confusion to the programmer.

## Organization of the Book

Data storage formats are covered in Chapters 2 and 3. Chapter 2 introduces the binary and hexadecimal number systems and presents the ASCII code for storing character data. Decimal integers, both signed and unsigned, are discussed in Chapter 3 along with the code used to store them. We use C programs to explore the concepts in Chapter 3. The C examples also provide an introduction to programming in C for those who have not used it yet. This introduction to C will be sufficient for the rest of the book.

Chapters 4 and 5 get down to the actual hardware level. Chapter 4 introduces the mathematics and electronic circuits used to build computers. There is a section on basic electronic circuit elements for those who are new to electronics. Then Chapter 5 moves on to some of the more common logic circuits used in computers. It ends with a discussion of memory implementations. If the book is being used for a software-only course, the instructor could consider skipping over these two chapters.

Chapter 6 introduces the central processing unit (CPU) and its relationship to memory and I/O. There is a description of how to use the `gdb` debugger to view the registers in the CPU. The basic set of registers used by programmers in the x86-64 architecture is given in this chapter.

Assembly language programming is introduced in Chapter 7. The topic is introduced by showing how to create a file containing the assembly language generated by the `gcc` compiler from C code. The basic assembly language template for a function is introduced, both for 64-bit and 32-bit mode. There is an overall sketch of how assemblers and linkers work.

In Chapter 8 we see how automatic variables are allocated on the stack, how values are assigned to them, and how functions are called. Argument passing, both in registers and on the stack, is discussed. The chapter shows how to call the `write`, `read`, `printf`, and `scanf` C Standard Library functions for user I/O. There is also a section on writing standalone programs that do not use the C environment and use the `syscall` instruction for direct operating system I/O.

Chapter 9 gives an introduction to machine code. There is a discussion of the REX codes used in 64-bit mode. Two instructions, `mov` and `add`, are used as examples.

Program control flow, specifically repetition and binary decision, are covered in Chapter 10. Conditional jumps are discussed in this chapter.

Chapter 11 discusses how to write your own functions and use the arguments passed to it. Both the 64-bit and 32-bit function interface techniques are described.

Bit-level logical and shift operations are covered in Chapter 12. The multiplication and division instructions are also discussed.

Arrays and structs are discussed in Chapter 13. This chapter includes a discussion of how simple C++ objects are implemented at both the C and the assembly language level.

Until this point in the book we have been using integers. In Chapter 14 we introduce formats for storing fractional values, including some IEEE 754 formats. In 64-bit mode the gcc compiler uses SSE2 instructions for floating point, but x87 instructions are used in 32-bit mode. The chapter gives an introduction to both instruction sets.

Exceptions and interrupts are discussed in Chapter 15. Chapter 16 is an introduction to hardware level I/O. Since most students will never do I/O at this level, this is another chapter that could be skipped.

A summary of the instructions used in this book is provided in Appendix A.5. At this point, there is only a list of the instructions. Eventually, there will be a description of each of them.

Appendix B is a highly simplified discussion of the fundamental concepts of the make facility.

Appendix C provides a very brief tutorial on using gdb for assembly language programs.

Appendix D gives a very brief introduction to the gcc syntax for embedding assembly language in a C function.

Almost all the solutions to the chapter exercises are provided in Appendix E. These can be useful for students who wish to use the exercises for self study; if you find yourself getting stuck on a problem, peek at the solution for some hints. Instructors are encouraged to discuss these solutions with their students. There is much to be learned from looking at another person's solution and thinking about how you might do it better.

The Bibliography lists a small fraction of the many books I have consulted when learning this material. I urge you to look at this list of books. I believe that you will want at least some of them in your reference library.

## Suggested Usage

- Our course at Sonoma State University covers each chapter approximately in the book's order. The programming exercises in Chapters 2 and 3 get the students used to using the lab right from the beginning of the course. Hardware simulators are used in the lab for Chapters 4 and 5.
- A pure assembly language course could easily omit Chapters 4 and 5.
- In a curriculum where binary numbers are covered in another course Chapters 2 and 3 could be skimmed. I recommend covering the C coding examples in Chapters 2 and 3 for students who have not programmed in the language. This would provide an introduction to C that should be adequate for the rest of the book.
- Experienced programmers who are using this book to learn x86-64 assembly language on their own should be able to skim the first five chapters. I believe that the remaining chapters would provide a good "primer" for reading the appropriate manuals.

## Production of the Book

I used  $\text{\LaTeX} 2_{\varepsilon}$  to typeset and draw the figures for this book. The fonts are DejaVu ([dejavu-fonts.org](http://dejavu-fonts.org)).

## Acknowledgements

I would like to thank the many students who have taken assembly language from me. They have asked many questions that caused me to think about the subject and how I can better explain it. They are the main reason I have written this book.

Two students deserve special thanks, David Tran and Zack Gold. They used this book in a class taught by Mike Lyle at Santa Rosa Junior College, David in Fall 2010 and Zack in Fall 2011. Both caught many of my typos and errors and gave me many helpful suggestions for clarifying my writing. I am very grateful for their careful reading of the book and the time they spent providing me with comments. It is definitely a better book as a result of their diligence.

I wish to thank Richard Gordon, Lynn Stauffer, Allan B. Cruse, Michael Lyle, Suzanne Rivoire, and Tia Watts for their thorough proofreading and critique of the previous versions of this book. By teaching from this book they have caught many of my errors and provided many excellent suggestions for clarifying the presentation.

I appreciate the work of those who volunteer their time to develop and maintain the software I used to create this book: GNU, Linux,  $\text{\LaTeX} 2_{\varepsilon}$ , etc.

In addition, I would like to thank my partner, João Barretto, for encouraging me to write this book and putting up with my many hours spent at my computer.

Finally, I am sure there are typos and errors left in this book, even with all the feedback I have received from students and colleagues and my efforts to correct what they found. But I hope it is in good enough shape that you will find reading the book relatively comfortable and that it will provide you some insight into how computers are organized.

# Chapter 1

## Introduction

Unlike most assembly language books, this one does not emphasize writing programs in assembly language. Higher-level languages, e.g., C, C++, Java, are much better for that. You should avoid writing in assembly language whenever possible.

You may wonder why you should study assembly language at all. The usual reasons given are:

1. *Assembly language is more efficient.* This does not always hold. Modern compilers are excellent at optimizing the machine code that is generated. Only a very good assembly language programmer can do better, and only in some situations. Assembly language programming is very tedious, even for the best programmers. Hence, it is very expensive. The possible gains in efficiency are seldom worth the added expense.
2. *There are situations where it must be used.* This is more difficult to evaluate. How do you know whether assembly language is required or not?

Both these reasons presuppose that you know the assembly language equivalent of the translation that your compiler does. Otherwise, you would have no way of deciding whether you can write a more efficient program in assembly language, and you would not know the machine level limitations of your higher-level language. So this book begins with the fundamental high-level language concepts and “looks under the hood” to see how they are implemented at the assembly language level.

There is a more important reason for reading this book. The interface to the hardware from a programmer’s view is the *instruction set architecture* (ISA). This book is a description of the ISA of the x86 architecture as it is used by the C/C++ programming languages. Higher-level languages tend to hide the ISA from the programmer, but good programmers need to understand it. This understanding is bound to make you a better programmer, even if you never write a single assembly language statement after reading this book.

Some of you will enjoy assembly language programming and wish to carry on. If your interests take you into systems programming, e.g., writing parts of an operating system, writing a compiler, or even designing another higher-level language, an understanding of assembly language is required. There are many challenging opportunities in programming embedded systems, and much of the work in this area demands at least an understanding of the ISA. This book serves as an introduction to assembly language programming and prepares you to move on to the intermediate and advanced levels.

In his book *The Design and Evolution of C++*[32] Bjarne Stroustrup nicely lists the purposes of a programming language:

- a tool for instructing machines
- a means of communicating between programmers
- a vehicle for expressing high-level designs
- a notation for algorithms
- a way of expressing relationships between concepts
- a tool for experimentation
- a means of controlling computerized devices.

It is assumed that you have had at least an introduction to programming that covered the first five items on the list. This book focuses on the first item — instructing machines — by studying assembly language programming of a 64-bit x86 architecture computer. We will use C as an example higher-level language and study how it instructs the computer at the assembly language level. Since there is a one-to-one correspondence between assembly language and machine language, this amounts to a study of how C is used to instruct a machine (computer).

You have already learned that a compiler (or interpreter) translates a program written in a higher-level language into machine language, which the computer can execute. But what does this mean? For example, you might wonder:

- How is an integer stored in memory?
- How is a computer instructed to implement an if-else construct?
- What happens when one function calls another function?
- How does the computer know how to return to the statement following the function call statement?
- How is a computer instructed to display a simple character string — for example, "Hello, world" — on the screen?

It is the goal of this book to answer these and many other questions. The specific higher-level programming language concepts that are addressed in this book include:

<b>General concept</b>	<b>C/C++ implementation</b>
Program organization	Functions, variables, literals
Allocation of variables for storage of primitive data types — integers, characters	int, char
Program flow control constructs — loops, two-way decision	while and for; if-else
Simple arithmetic and logical operations	+ , - , * , / , % , & ,
Boolean operators	! , && ,
Data organization constructs — arrays, records, objects	Arrays, structs, classes (C++ only)
Passing data to/from named procedures	Function parameter lists; return values
Object operations	Invoking a member function (C++ only)

This book assumes that you are familiar with these programming concepts in C, C++, and/or Java.

## 1.1 Computer Subsystems

We begin with a very brief overview of computer hardware. The presentation here is intended to provide you with a rough context of how things fit together. In subsequent chapters we will delve into more details of the hardware and how it is controlled by software.

We can think of computer hardware as consisting of three separate subsystems as shown in Fig. 1.1.

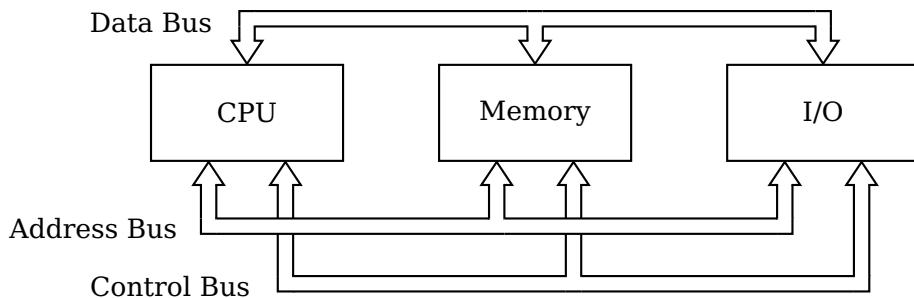


Figure 1.1: Subsystems of a computer. The CPU, Memory, and I/O subsystems communicate with one another via the three buses.

**Central Processing Unit (CPU)** controls most of the activities of the computer, performs the arithmetic and logical operations, and contains a small amount of very fast memory.

**Memory** provides storage for the instructions for the CPU and the data they manipulate.

**Input/Output (I/O)** communicates with the outside world and with mass storage devices (e.g., disks).

When you create a new program, you use an editor program to write your new program in a high-level language, for example, C, C++, or Java. The editor program sees the source code for your new program as data, which is typically stored in a file on the disk. Then you use a compiler program to translate the high-level language statements into machine instructions that are stored in a disk file. Just as with the editor program, the compiler program sees both your source code and the resulting machine code as data.

When it comes time to execute the program, the instructions are read from the machine code disk file into memory. At this point, the program is a sequence of instructions stored in memory. Most programs include some constant data that are also stored in memory. The CPU executes the program by fetching each instruction from memory and executing it. The data are also fetched as needed by the program.

This computer model — both the program instructions and data are stored in a memory unit that is separate from the processing unit — is referred to as the *von Neumann architecture*. It was described in 1945 by John von Neumann [35], although other computer science pioneers of the day were working with the same concepts. This is in contrast to a fixed-program computer, e.g., a calculator. A compiler illustrates one of the benefits of the von Neumann architecture. It is a program that treats the source file as data, which it translates into an executable binary file that is also treated as data. But the executable binary file can also be run as a program.

A downside of the von Neumann architecture is that a program can be written to view itself as data, thus enabling a self-modifying program. GNU/Linux, like most modern, general purpose operating systems, prohibits applications from modifying themselves.

Most programs also access I/O devices, and each access must also be programmed. I/O devices vary widely. Some are meant to interact with humans, for example, a keyboard, a mouse, a screen. Others are meant for machine readable I/O. For example, a program can store a file on a disk or read a file from a network. These devices all have very different behavior, and their timing characteristics differ drastically from one another. Since I/O device programming is difficult, and every program makes use of them, the software to handle I/O devices is included in the operating system. GNU/Linux provides a rich set of functions that an applications programmer can use to perform I/O actions, and we will call upon these services of GNU/Linux to perform our I/O operations. Before tackling I/O programming, you need to gain a thorough understanding of how the CPU executes programs and interacts with memory.

The goal of this book is study how programs are executed by the computer. We will focus on how the program and data are stored in memory and how the CPU executes instructions. We leave I/O programming to more advanced books.

## 1.2 How the Subsystems Interact

The subsystems in Figure 1.1 communicate with one another via buses. You can think of a *bus* as a communication pathway with a protocol specifying exactly how the pathway is used. The buses shown here are logical groupings of the signals that must pass between the three subsystems. A given bus implementation may not have physically separate paths for each of the three types of signals. For example, the PCI bus standard uses the same physical pathway for the address and the data, but at different times. Control signals indicate whether there is an address or data on the lines at any given time.

A program consists of a sequence of instructions that is stored in memory. When the CPU is ready to execute the next instruction in the program, the location of that instruction in memory is placed on the *address bus*. The CPU also places a “read” signal on the *control bus*. The memory subsystem responds by placing the instruction on the *data bus*, where the CPU can then read it. If the CPU is instructed to read data from memory, the same sequence of events takes place.

If the CPU is instructed to store data in memory, it places the data on the *data bus*, places the location in memory where the data is to be stored on the *address bus*, and places a “write” signal on the *control bus*. The memory subsystem responds by copying the data on the *data bus* into the specified memory location.

If an instruction calls for reading or writing data from memory or to memory, the next instruction in the program sequence cannot be read from memory over the same bus until the current instruction has completed the data transfer. This conflict has given rise to another stored-program architecture. In the *Harvard architecture* the program and data are stored in different memories, each with its own bus connected to the CPU. This makes it possible for the CPU to access both program instructions and data simultaneously. The issues should become clearer to you in Chapter 6.

In modern computers the bus connecting the CPU to external memory modules cannot keep up with the execution speed of the CPU. The slowdown of the bus is called the **von Neumann bottleneck**. Almost all modern CPU chips include some cache memory, which is connected to the other CPU components with much faster internal buses. The cache memory closest to the CPU commonly has a Harvard architecture configuration to achieve higher throughput of data processing.

CPU interaction with I/O devices is essentially the same as with memory. If the CPU is

instructed to read a piece of data from an input device, the particular device is specified on the address bus and a “read” signal is placed on the control bus. The device responds by placing the data item on the data bus. And the CPU can send data to an output device by placing the data item on the data bus, specifying the device on the address bus, and placing a “write” signal on the control bus. Since the timing of various I/O devices varies drastically from CPU and memory timing, special programming techniques must be used. Chapter 16 provides an introduction to I/O programming techniques.

These few paragraphs are intended to provide you a very general overall view of how computer hardware works. The rest of the book will explore many of these concepts in more depth. Most of the discussion is at the ISA level, but we will also take a peek at the hardware implementation. In Chapter 4 we will even look at some transistor circuits. The goal of the book is to provide you with an introduction to computer architecture as seen from a software point of view.

# **Chapter 2**

# **Data Storage Formats**

In this chapter, we begin exploring how data is encoded for storage in memory and write some programs in C to explore these concepts. One way to look at a modern computer is that it is made up of:

- Billions of two-state switches. Each of the switches is always in one state or the other, and it stays in that state until the control unit changes its state or the power is turned off.
- A control unit that can:
  - Detect the state of each switch.
  - Change the state of that switch and/or other switches.

There is also provision for communicating with the world outside the computer — input and output.

## **2.1 Bits and Groups of Bits**

Since nearly everything that takes place in a computer, from the instructions that make up a program to the data these instructions act upon, depends upon two-state switches, we need a good notation to use when talking about the states of the switches. It is clearly very cumbersome to say something like,

“The first switch is on, the second one is also on,  
but the third is off, while the fourth is on.”

We need a more concise notation, which leads us to use numbers. When dealing with numbers, you are most familiar with the decimal system, which is based on ten, and thus uses ten digits.

**Decimal digits:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Two number systems are useful when talking about the states of switches — the binary system, which is based on two,

**Binary digits:** 0, 1

and the hexadecimal system, which is based on sixteen.

**Hexadecimal digits:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f

A less commonly used number system is octal, which is based on eight.

### **Octal digits: 0, 1, 2, 3, 4, 5, 6, 7**

“Binary digit” is commonly shortened to “bit.” It is common to bypass the fact that a bit represents the state of a switch, and simply call the switches “bits.” Using bits (binary digits), we can greatly simplify the previous statement about switches as 1101, which you can think of as representing “on, on, off, on.” It does not matter whether we use 1 to represent “on” and 0 as “off,” or 0 as “on” and 1 as “off.” We simply need to be consistent. You will see that this will occur naturally; it will not be an issue.

Hexadecimal is commonly used as a shorthand notation to specify bit patterns. Since there are sixteen hexadecimal digits, each one can be used to specify uniquely a group of four bits. Table 2.1 shows the correspondence between each possible group of four bits and one hexadecimal digit. Thus, the above English statement specifying the state of four switches can be written with a single hexadecimal digit, d.

<b>Four binary digits (bits)</b>	<b>One hexadecimal digit</b>
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	a
1011	b
1100	c
1101	d
1110	e
1111	f

Table 2.1: Hexadecimal representation of four bits.

When it is not clear from the context, we will indicate the base of a number in this text with a subscript. For example,  $100_{10}$  is written in decimal,  $100_{16}$  is written in hexadecimal, and  $100_2$  is written in binary.

Hexadecimal digits are especially convenient when we need to specify the state of a group of, say, 16 or 32 switches. In place of each group of four bits, we can write one hexadecimal digit. For example,

$$0110 \ 1100 \ 0010 \ 1010_2 = 6c2a_{16} \quad (2.1)$$

and

$$0000 \ 0001 \ 0010 \ 0011 \ 1010 \ 1011 \ 1100 \ 1101_2 = 0123 \ abcd_{16} \quad (2.2)$$

A single bit has limited usefulness when we want to store data. We usually need to use a group of bits to store a data item. This grouping of bits is so common that most modern computers only allow a program to access bits in groups of eight. Each of these groups is called a *byte*.

**byte:** A contiguous group of bits, usually eight.

Historically, the number of bits in a byte has varied depending on the hardware and the operating system. For example, the CDC 6000 series of scientific mainframe computers used a six-bit byte. Nearly everyone uses “byte” to mean eight bits today.

Another important reason to learn hexadecimal is that the programming language may not allow you to specify a value in binary. Prefixing a number with `0x` (zero, lower-case ex) in C/C++ means that the number is expressed in hexadecimal. There is no C/C++ syntax for writing a number in binary. The syntax for specifying bit patterns in C/C++ is shown in Table 2.2. (The 32-bit pattern for the decimal value 123 will become clear after you read Sections 2.2 and 2.3.) Although the GNU assembler, `as`, includes a notation for specifying bit patterns in binary, it is usually more convenient to use the C/C++ notation.

	<b>Prefix</b>	<b>Example</b>	<b>32-bit pattern (binary)</b>
Decimal:	none	123	0000 0000 0000 0000 0000 0000 0111 1011
Hexadecimal:	0x	0x123	0000 0000 0000 0000 0000 0001 0010 0011
Octal:	0	0123	00 000 000 000 000 000 000 001 010 011

Table 2.2: C/C++ syntax for specifying literal numbers. Octal bits grouped by three for readability.

## 2.2 Mathematical Equivalence of Binary and Decimal

We have seen in the previous section that binary digits are the natural way to show the states of switches within the computer and that hexadecimal is a convenient way to show the states of up to four switches with only one character. Now we explore some of the mathematical properties of the *binary number system* and show that it is numerically equivalent to the more familiar decimal (base 10) number system. Showing the mathematical equivalence of the hexadecimal and decimal number systems is left as exercises at the end of this chapter.

We will consider only integers at this point. The mathematical presentation here does, of course, generalize to fractional values. Simply continue the exponents of the radix,  $r$ , on to negative values, i.e.,  $n-1, n-2, \dots, 1, 0, -1, -2, \dots$ . This will be covered in detail in Chapter 14.

By convention, we use a *positional notation* when writing numbers. For example, in the decimal number system, the integer 123 is taken to mean

$$1 \times 100 + 2 \times 10 + 3 \times 1 \quad (2.3)$$

or

$$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 \quad (2.4)$$

The right-most digit (3 in Equation 2.4) is the *least significant digit* because it “counts” the least in the total value of this number. The left-most digit (1 in this example) is the *most significant digit* because it “counts” the most in the total value of this number.

The *base* or *radix* of the decimal number system is ten. There are ten symbols for representing the digits: 0, 1, ..., 9. Moving a digit one place to the left increases its value by a factor of ten, and moving it one place to the right decreases its value by a factor of ten. The positional notation generalizes to any radix,  $r$ :

$$d_{n-1} \times r^{n-1} + d_{n-2} \times r^{n-2} + \dots + d_1 \times r^1 + d_0 \times r^0 \quad (2.5)$$

where there are n digits in the number and each  $d_i = 0, 1, \dots, r-1$ . The radix in the binary number system is 2, so there are only two symbols for representing the digits:  $d_i = 0, 1$ . We can specialize Equation 2.5 for the binary number system as

$$d_{n-1} \times 2^{n-1} + d_{n-2} \times 2^{n-2} + \dots + d_1 \times 2^1 + d_0 \times 2^0 \quad (2.6)$$

where there are n digits in the number and each  $d_i = 0, 1$ .

For example, the eight-digit binary number 1010 0101 is interpreted as

$$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \quad (2.7)$$

If we evaluate this expression in decimal, we get

$$128 + 0 + 32 + 0 + 0 + 4 + 1 + 1 = 165_{10} \quad (2.8)$$

This example illustrates the method for converting a number from the binary number system to the decimal number system. It is stated in Algorithm 2.1.

---

#### **Algorithm 2.1:** Convert binary to unsigned decimal.

---

**input** : An integer expressed in binary.

**output** : Decimal expression of the integer.

- 1 Compute the value of each power of 2 in Equation 2.6 in decimal.
  - 2 Multiply each power of two by its corresponding  $d_i$ .
  - 3 Sum the terms in Equation 2.6.
- 

Be careful to distinguish the binary number system from writing the state of a bit in binary. Each switch in the computer can be represented by a bit (binary digit), but the entity that it represents may not even be a number, much less a number in the binary number system. For example, the bit pattern 0011 0010 represents the character “2” in the ASCII code for characters. But in the binary number system  $0011\ 0010_2 = 50_{10}$ .

See Exercises 2-8 and 2-9 for converting hexadecimal to decimal.

## 2.3 Unsigned Decimal to Binary Conversion

In Section 2.2 (page 8), we covered conversion of a binary number to decimal. In this section we will learn how to convert an unsigned decimal integer to binary. Unsigned numbers have no sign. Signed numbers can be either positive or negative. Say we wish to convert a unsigned decimal integer, N, to binary. We set it equal to the expression in Equation 2.6, giving us:

$$N = d_{n-1} \times 2^{n-1} + d_{n-2} \times 2^{n-2} + \dots + d_1 \times 2^1 + d_0 \times 2^0 \quad (2.9)$$

where  $d_i = 0$  or  $1$ . Dividing both sides by 2,

$$(N/2) + \frac{r_0}{2} = d_{n-1} \times 2^{n-2} + d_{n-2} \times 2^{n-3} + \dots + d_1 \times 2^0 + d_0 \times 2^{-1} \quad (2.10)$$

where / is the div operator and the remainder,  $r_0$ , is 0 or 1. Since  $(N/2)$  is an integer and all the terms except the  $2^{-1}$  term on the right-hand side of Equation 2.10 are integers, we can see that  $d_0 = r_0$ . Subtracting  $r_0/2$  from both sides gives,

$$(N/2) = d_{n-1} \times 2^{n-2} + d_{n-2} \times 2^{n-3} + \dots + d_1 \times 2^0 \quad (2.11)$$

Dividing both sides of Equation 2.11 by two:

$$(N/4) + \frac{r_1}{2} = d_{n-1} \times 2^{n-3} + d_{n-2} \times 2^{n-4} + \dots + d_1 \times 2^{-1} \quad (2.12)$$

From Equation 2.12 we see that  $d_1 = r_1$ . It follows that the binary representation of a number can be produced from right (low-order bit) to left (high-order bit) by applying the algorithm shown in Algorithm 2.2.

---

**Algorithm 2.2:** Convert unsigned decimal to binary.
 

---

**input** : An integer expressed in decimal.  
**output** : Binary expression of the integer, one bit at a time, right-to-left.  
**1** quotient  $\leftarrow$  theInteger;  
**2** **while** quotient  $\neq 0$  **do**  
**3**   | nextBit  $\leftarrow$  quotient % 2;  
**4**   | quotient  $\leftarrow$  quotient / 2;

---

**Example 2-a**


---

Convert  $123_{10}$  to binary.

*Solution:*

$$\begin{aligned} 123 \div 2 &= 61 + 1/2 \Rightarrow d_0 = 1 \\ 61 \div 2 &= 30 + 1/2 \Rightarrow d_1 = 1 \\ 30 \div 2 &= 15 + 0/2 \Rightarrow d_2 = 0 \\ 15 \div 2 &= 7 + 1/2 \Rightarrow d_3 = 1 \\ 7 \div 2 &= 3 + 1/2 \Rightarrow d_4 = 1 \\ 3 \div 2 &= 1 + 1/2 \Rightarrow d_5 = 1 \\ 1 \div 2 &= 0 + 1/2 \Rightarrow d_6 = 1 \\ 0 \div 2 &= 0 + 0/2 \Rightarrow d_7 = 0 \end{aligned}$$

So

$$\begin{aligned} 123_{10} &= d_7d_6d_5d_4d_3d_2d_1d_0 \\ &= 01111011_2 \\ &= 7b_{16} \end{aligned}$$

□

There are times in some programs when it is more natural to specify a bit pattern rather than a decimal number. We have seen that it is possible to easily convert between the number bases, so you could convert the bit pattern to a decimal value, then use that. It is usually much easier to think of the bits in groups of four, then convert the pattern to hexadecimal.

For example, if your algorithm required the use of zeros alternating with ones:

0101 0101 0101 0101 0101 0101 0101 0101

this can be converted to the decimal value

1431655765

or the hexadecimal value (shown here in C/C++ syntax)

0x55555555

Once you have memorized Table 2.1, it is clearly much easier to work with hexadecimal for bit patterns.

The discussion in these two sections has dealt only with unsigned integers. The representation of signed integers depends upon some architectural features of the CPU and will be discussed in Chapter 3 when we discuss computer arithmetic.

## 2.4 Memory — A Place to Store Data (and Other Things)

We now have the language necessary to begin discussing the major components of a computer. We start with the memory.

You can think of memory as a (very long) array of bytes. Each byte has a particular location (or address) within this array. That is, you could think of

```
memory[123]
```

as specifying the 124<sup>th</sup> byte in memory. (Don't forget that array indexing starts with 0.) We generally do not use array notation and simply use the index number, calling it the *address* or *location* of the byte.

**address (or location):** Identifies a specific byte in memory.

The address of a particular byte never changes. That is, the 957<sup>th</sup> byte from the beginning of memory will always remain the 957<sup>th</sup> byte. However, the state of each of the bits — either 0 or 1 — in any given byte can be changed.

Computer scientists typically express the address of each byte in memory in hexadecimal. So we would say that the 957<sup>th</sup> byte is at address 0x3bc.

From the discussion of hexadecimal in Section 2.1 (page 6) we can see that the first sixteen bytes in memory have the addresses 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f. Using the notation

```
address: contents (bit-pattern-at-the-address)
```

we show the (possible) contents (the state of the bits) of each of the first sixteen bytes of memory in Figure 2.1.

Address	Contents	Address	Contents
00000000:	0110 1010	00000008:	1111 0000
00000001:	1111 0000	00000009:	0000 0010
00000002:	0101 1110	0000000a:	0011 0011
00000003:	0000 0000	0000000b:	0011 1100
00000004:	1111 1111	0000000c:	1100 0011
00000005:	0101 0001	0000000d:	0011 1100
00000006:	1100 1111	0000000e:	0101 0101
00000007:	0001 1000	0000000f:	1010 1010

Figure 2.1: Possible contents of the first sixteen bytes of memory; addresses shown in hexadecimal, contents shown in binary. Note that the addresses are shown as 32-bit values. (*The contents shown here are arbitrary.*)

The state of each bit is indicated by a binary digit (bit) and is arbitrary in Figure 2.1. The bits have been grouped by four for readability. The grouping of the memory bits also shows that we can use two hexadecimal digits to indicate the state of the bits in each byte, as shown in Figure 2.2. For example, the contents of memory location 0000000b are 3c. That means the eight bits that make up the twelfth byte in memory are set to the bit pattern 0011 1100.

Once a bit (switch) in memory is set to either zero or one, it stays in that state until the control unit actively changes it or the power is turned off. There is an exception. Computers also contain memory in which the bits are permanently set. Such memory is called *Read Only Memory* or *ROM*.

<b>Address</b>	<b>Contents</b>	<b>Address</b>	<b>Contents</b>
00000000:	6a	00000008:	f0
00000001:	f0	00000009:	02
00000002:	5e	0000000a:	33
00000003:	00	0000000b:	3c
00000004:	ff	0000000c:	c3
00000005:	51	0000000d:	3c
00000006:	cf	0000000e:	55
00000007:	18	0000000f:	aa

Figure 2.2: Repeat of Figure 2.1 with contents shown in hex. Two hexadecimal characters are required to specify *one* byte.

**Read Only Memory (ROM)** : Each bit is permanently set to either zero or one. The control unit can read the state of each bit but cannot change it.

You have probably heard the term “RAM” used for memory that can be changed by the control unit. RAM stands for Random Access Memory. The terminology used here is inconsistent. “Random access” means that it takes the same amount of time to access any byte in the memory. This is in contrast to memory that is sequentially accessible, e.g., tape. The length of time it takes to access a byte on tape depends upon the physical location of the byte with respect to the current tape position.

**Random Access Memory (RAM)** : The control unit can read the state of each bit and can change it.

A bit can be used to store data. For example, we could use a single bit to indicate whether a student passes a course or not. We might use 0 for “not passed” and 1 for “passed.” A single bit allows only two possible values of a data item. We cannot for example, use a single bit to store a course letter grade — A, B, C, D, or F.

How many bits would we need to store a letter grade? Consider all possible combinations of two bits:

00  
01  
10  
11

Since there are only four possible bit combinations, we cannot represent all five letter grades with only two bits. Let’s add another bit and look at all possible bit combinations:

000  
001  
010  
011  
100  
101  
110  
111

There are eight possible bit patterns, which is more than sufficient to store any one of the five letter grades. For example, we may choose to use the code

Letter Grade	Bit Pattern
A	000
B	001
C	010
D	011
F	100

This example illustrates two issues that a programmer must consider when storing data in memory in addition to its location(s):

**How many bits are required to store the data?** In order to answer this we need to know how many different values are allowed for the particular data item. Study the two examples above — two bits and three bits — and you can see that adding a bit doubles the number of possible values. Also, notice that we might not use all the possible bit patterns.

**What is the code for storing the data?** Most of the data we deal with in everyday life is not expressed in terms of zeros and ones. In order to store it in computer memory, the programmer must decide upon a code of zeros and ones to use. In the above (three bit) example we used 000 to represent a letter grade of A, 001 to represent B, etc.

Thus, in the grade example, a programmer may choose to store the letter grade at byte number bfffffed0 in memory. If the grade is "A", the programmer would set the bit pattern at location bfffffed0 to 00<sub>16</sub>. If the grade is "C", the programmer would set the bit pattern at location bfffffed0 to 02<sub>16</sub>. In this example, one of the jobs of an assembly language programmer would be to determine how to set the bit pattern at byte number bfffffed0 to the appropriate bit pattern.

High-level languages use *data types* to determine the number of bits and the storage code. For example, in C you may choose to store the letter grades in the above example in a char variable and use the characters 'A', 'B',..., 'F' to indicate the grade. In Section 2.7 you will learn that the compiler would use the following storage formats:

Letter Grade	Bit Pattern
A	0100 0001
B	0100 0010
C	0100 0011
D	0100 0100
F	0100 0101

And programming languages, even assembly language, allow programmers to create symbolic names for memory addresses. The compiler (or assembler) determines the correspondence between the programmer's symbolic name and the numerical address. The programmer can refer to the address by simply using the symbolic name.

## 2.5 Using C Programs to Explore Data Formats

*Before writing any programs, I urge you to read Appendix B on writing Makefiles, even if you are familiar with them. Many of the problems I have helped students solve are due to errors in their Makefile. And many of the Makefile errors go undetected due to the default behavior of the make program.*

We will use the C programming language to illustrate these concepts because it takes care of the memory allocation problem, yet still allows us to get reasonably close to

the hardware. You probably learned to program in the higher-level, object-oriented paradigm using either C++ or Java. C does not support the object-oriented paradigm.

C is a *procedural programming* language. The program is divided into functions. Since there are no classes in C, there is no such thing as a member function. The programmer focuses on the algorithms used in each function, and all data items are explicitly passed to the functions.

We can see how this works by exploring the C Standard Library functions, `printf` and `scanf`, which are used to write to the screen and read from the keyboard. We will develop a program in C using `printf` and `scanf` to illustrate the concepts discussed in the previous sections. The header file required by either of these functions is:

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

which includes the prototype statements for the `printf` and `scanf` functions:

```
int printf(const char *format, ...);
int scanf(const char *format, ...);
```

`printf` is used to display text on the screen. The first argument, `format`, controls the text display. At its simplest, `format` is simply an explicit text string in double quotes.<sup>1</sup> For example,

```
printf("Hello, world.\n");
```

would display

```
Hello, world.
```

If there are additional arguments, the format string must specify how each of these arguments is to be converted for display. This is accomplished by inserting a conversion code within the format string at the point where the argument value is to be displayed. Each conversion code is introduced by the '%' character. For example, Listing 2.1 shows how to display both an `int` variable and a `float` variable.

---

```

1 /*
2  * intAndFloat.c
3  * Using printf to display an integer and a float.
4  * Bob Plantz - 4 June 2009
5 */
6 #include <stdio.h>
7
8 int main(void)
9 {
10     int anInt = 19088743;
11     float aFloat = 19088.743;
12
13     printf("The integer is %i and the float is %f\n", anInt, aFloat);
14
15     return 0;
16 }
```

Listing 2.1: Using `printf` to display numbers.

Compiling and running the program in Listing 2.1 on my computer gave (user input is **boldface**):

---

<sup>1</sup>The text string is a null-terminated array of characters as described in Section 2.7 (page 21). This is not the C++ string class.

```
bob$ gcc -Wall -o intAndFloat intAndFloat.c
bob$ ./intAndFloat
The integer is 19088743 and the float is 19088.742188
bob$
```

Yes, the float really is that far off. This will be explained in Chapter 14.

This is not a book about how to use the GNU development environment, so I usually do not show the compile command. I am showing it here to help get you started. You should use the `man gcc` command to learn about the command line options.

Some common conversion codes are `d` or `i` for integer, `f` for float, and `x` for hexadecimal. The conversion codes may include other characters to specify properties like the field width of the display, whether the value is left or right justified within the field, etc. We will not cover the details here. You should read `man` page 3 for `printf` to learn more.

`scanf` is used to read from the keyboard. The format string typically includes only conversion codes that specify how to convert each value as it is entered from the keyboard and stored in the following arguments. Since the values will be stored in variables, it is necessary to pass the address of the variable to `scanf`. For example, we can store keyboard-entered values in `x` (an `int` variable) and `y` (a `float` variable) thusly

```
scanf("%i %f", &x, &y);
```

The use of `printf` and `scanf` are illustrated in the C program in Listing 2.2, which will allow us to explore the mathematical equivalence of the decimal and hexadecimal number systems.

```
1 /*
2  * echoDecHex.c
3  * Asks user to enter a number in decimal and one
4  * in hexadecimal then echoes both in both bases
5  * Bob Plantz - 4 June 2009
6 */
7
8 #include <stdio.h>
9
10 int main(void)
11 {
12     int x;
13     unsigned int y;
14
15     while(1)
16     {
17         printf("Enter a decimal integer (0 to quit): ");
18         scanf("%i", &x);
19         if (x == 0) break;
20
21         printf("Enter a bit pattern in hexadecimal (0 to quit): ");
22         scanf("%x", &y);
23         if (y == 0) break;
24
25         printf("%i is stored as %#010x, and\n", x, x);
26         printf("%#010x represents the decimal integer %i\n\n", y, y);
27     }
28 }
```

```

29     printf("End of program.\n");
30
31     return 0;
32 }
```

Listing 2.2: C program showing the mathematical equivalence of the decimal and hexadecimal number systems.

Here is an example run of this program (user input is **boldface**):

```

bob$ ./echoDecHex
Enter a decimal integer: 123
Enter a bit pattern in hexadecimal: 7b
123 is stored as 0x0000007b, and
0x0000007b represents the decimal integer 123

Enter a decimal integer: 0
End of program.
bob$
```

Let us walk through the program in Listing 2.2.

- The program declares two `ints`, `x` and `y`.
- The user is prompted to enter an integer in decimal, and the user's response is read from the keyboard and stored in the memory allocated for `x`. The conversion code text string passed to `scanf`, “`%i`”, causes `scanf` to interpret the user's keystrokes as representing a decimal integer. Note that the address of `x`, `&x`, must be passed to `scanf` so that it can store the integer at the memory location named `x`.
- The program next prompts the user to enter a bit pattern in hexadecimal. In this case the conversion code text string passed to `scanf` is “`%x`”, which causes `scanf` to interpret the user's keystrokes as representing hexadecimal digits. Note that the address of `y`, `&y`, must be passed to `scanf` so that it can store the integer at the memory location named `y`.
- Now let us examine the two `printf` function calls that display the results. The “`%i`” conversion code is straightforward. The value of the corresponding variable is displayed in decimal at that point in the text string.
- The “`%#010x`” conversion factor is more interesting. (If you are at a computer read section 3 of the man page for `printf` as you follow through this description.) The basic conversion is specified by the “`x`” character; it causes the value to be displayed in hexadecimal. The “`#`” character causes an “alternate form” to be used for the display, which is the C syntax for hexadecimal numbers; that is, the value is prefaced by `0x` when it is displayed. The ‘`0`’ character immediately after the ‘`#`’ character causes ‘`0`’ to be used as the fill character. The number “`10`” causes the display to occupy at least ten characters (the field width).
- Look carefully at the output from this program above. The bit patterns used to store the data input by the user, shown in hexadecimal, show that the unsigned `ints` are stored in the binary number system (see Section 2.2, page 8 and Section 2.3, page 9). That is,  $123_{10}$  is stored as  $0000007b_{16}$ .

The program in Listing 2.2 demonstrates a very important concept — hexadecimal is used as a human convenience for stating bit patterns. A number is not inherently

binary, decimal, or hexadecimal. A particular value can be expressed in a precisely equivalent way in each of these three number bases. For that matter, it can be expressed equivalently in any number base.

## 2.6 Examining Memory With gdb

Now that we have started writing programs, you need to learn how to use the GNU debugger, gdb. It may seem premature at this point. The programs are so simple, they hardly require debugging. Well, it is better to learn how to use the debugger on a simple example than on a complicated program that does not work. In other words, tackle one problem at a time.

There is a better reason for learning how to use gdb now. You will find that it is a very valuable tool for learning the material in this book, even when you write bug-free programs.

gdb has a large number of commands, but the following are the ones that will be used in this section:

- `li lineNumber` — lists ten lines of the source code, centered at the specified line number.
- `break sourceFilename:lineNumber` — sets a breakpoint at the specified line in the source file. Control will return to gdb when the line number is encountered.
- `run` — begins execution of a program that has been loaded under control of gdb.
- `cont` — continues execution of a program that has been running.
- `print expression` — evaluate expression and display its value.
- `printf "format", var1, var2, ...` — displays the values of the vars, using the format specified in the `format` string.<sup>2</sup>
- `x/nfs memoryAddress` — displays (examine) *n* values in memory in format *f* of size *s* starting at *memoryAddress*.

We will use the program in Listing 2.1 to see how gdb can be used to explore the concepts in more depth. Here is a screen shot of how I compiled the program then used gdb to control the execution of the program and observe the memory contents. My typing is **boldface** and the session is annotated in *italics*. Note that you will probably see different addresses if you replicate this example on your own (Exercise 2-27).

```
bob$ gcc -g -Wall -o intAndFloat intAndFloat.c
```

*The “-g” option is required. It tells the compiler to include debugger information in the executable program.*

```
bob$ gdb ./intAndFloat
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
```

<sup>2</sup>Follows the same pattern as the C Standard Library `printf`.

```
and "show warranty" for details.  
This GDB was configured as "x86_64-linux-gnu".  
For bug reporting instructions, please see:  
<http://bugs.launchpad.net/gdb-linaro/>...  
Reading symbols from /home/bob/my_book_working/progs/chap02/intAndFloat...  
(gdb) li  
1 /*  
2 * intAndFloat.c  
3 * Using printf to display an integer and a float.  
4 * Bob Plantz - 4 Jun 2009  
5 */  
6 #include <stdio.h>  
7  
8 int main(void)  
9 {  
10     int anInt = 19088743;  
(gdb) 11     float aFloat = 19088.743;  
12  
13     printf("The integer is %i and the float is %f\n", anInt, aFloat);  
14  
15     return 0;  
16 }  
(gdb)
```

*The li command lists ten lines of source code. The display ends with the (gdb) prompt. Pushing the return key will repeat the previous command, and li is smart enough to display the next (up to) ten lines.*

```
(gdb) br 13  
Breakpoint 1 at 0x40050b: file intAndFloat.c, line 13.
```

*I set a breakpoint at line 13. When the program is executing, if it ever gets to this statement, execution will pause before the statement is executed, and control will return to gdb.*

```
(gdb) run  
Starting program: /home/bob/intAndFloat  
  
Breakpoint 1, main () at intAndFloat.c:13  
13     printf("The integer is %i and the float is %f\n", anInt, aFloat)
```

*The run command causes the program to start execution from the beginning. When it reaches our breakpoint, control returns to gdb.*

```
(gdb) print anInt  
$1 = 19088743  
(gdb) print aFloat  
$2 = 19088.7422
```

*The print command displays the value currently stored in the named variable. There is a round off error in the float value. As mentioned above, this will be explained in Chapter 14.*

```
(gdb) printf "anInt = %i and aFloat = %f\n", anInt, aFloat
anInt = 19088743 and aFloat = 19088.742188
(gdb) printf "anInt = %#010x and aFloat = %#010x\n", anInt, aFloat
and in hex, anInt = 0x01234567 and aFloat = 0x00004a90
```

*The printf command can be used to format the displayed values. The formatting string is essentially the same as for the printf function in the C Standard Library.*

*Take a moment and convert the hexadecimal values to decimal. The value of anInt is correct, but the value of aFloat is  $19088_{10}$ . The reason for this odd behavior is that the x formatting character in the printf function first converts the value to an int, then displays that int in hexadecimal. In C/C++, conversion from float to int truncates the fractional part.*

*Fortunately, gdb provides another command for examining the contents of memory directly—that is, the actual bit patterns. In order to use this command, we need to determine the actual memory addresses where the anInt and aFloat variables are stored.*

```
(gdb) print &anInt
$3 = (int *) 0x7fffffff058
(gdb) print &aFloat
$4 = (float *) 0x7fffffff05c
```

*The address-of operator (&) can be used to print the address of a variable. Notice that the addresses are very large. The system is in 64-bit mode, which uses 64-bit addresses. (gdb does not display leading zeros.)*

```
(gdb) help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
t(binary), f(float), a(address), i(instruction), c(char) and s(string).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format.
```

Defaults for format and size letters are those previously used.  
Default count is 1. Default address is following last thing printed  
with this command or "print".

*The x command is used to examine memory. Its help message is very brief, but it tells you everything you need to know.*

```
(gdb) x/1dw 0x7fffffff058
0x7fffffff058: 19088743
(gdb) x/1fw 0x7fffffff05c
0x7fffffff05c: 19088.7422
```

*The x command can be used to display the values in their stored data type.*

```
(gdb) x/1xw 0x7fffffff058
0x7fffffff058: 0x01234567
(gdb) x/4xb 0x7fffffff058
0x7fffffff058: 0x67 0x45 0x23 0x01
```

The display of the `anInt` variable in hexadecimal, which is located at memory address `0x7fffffff058`, also looks good. However, when displaying these same four bytes as separate values, the least significant byte appears first in memory.

Notice that in the multiple byte display, the first byte (the one that contains `0x67`) is located at the address shown on the left of the row. The next byte in the row is at the subsequent address (`0x7fffffff059`). So this row displays each of the bytes stored at the four memory addresses `0x7fffffff058`, `0x7fffffff059`, `0x7fffffff05a`, and `0x7fffffff05b`.

```
(gdb) x/1fw 0x7fffffff05c
0x7fffffff05c:    19088.7422
(gdb) x/1xw 0x7fffffff05c
0x7fffffff05c: 0x4695217c
(gdb) x/4xb 0x7fffffff05c
0x7fffffff05c: 0x7c      0x21      0x95      0x46
```

The display of the `aFloat` variable in hexadecimal simply looks wrong. This is due to the storage format of floats, which is very different from ints. It will be explained in Chapter 14.

The byte by byte display of the `aFloat` variable in hexadecimal also shows that it is stored in little endian order.

```
(gdb) cont
Continuing.
The integer is 19088743 and the float is 19088.742188
[Inferior 1 (process 3221) exited normally]
(gdb) q
bob$
```

Finally, I continue to the end of the program. Notice that gdb is still running and I have to quit the gdb program.

This example illustrates a property of the x86 processors. Data is stored in memory with the least significant byte in the lowest-numbered address. This is called *little endian* storage. Look again at the display of the four bytes beginning at `0x7fffffff058` above. We can rearrange this display to show the bit patterns at each of the four locations:

```
7fffffff058: 67
7fffffff059: 45
7fffffff05a: 23
7fffffff05b: 01
```

Yet when we look at the entire 32-bit value in hexadecimal the bytes seem to be arranged in the proper order:

```
7fffffff058: 01234567
```

When we examine memory one byte at a time, each byte is displayed in numerically ascending addresses. At first glance, the value appears to be stored backwards.

We should note here that many processors, e.g., the PowerPC architecture, use *big endian* storage. As the name suggests, the most significant ("biggest") byte is stored in the first (lowest-numbered) memory address. If we ran the program above on a big endian computer, we would see (assuming the variable is located at the same address):

```
(gdb) x/1xw 0x7fffffff058
0x7fffffff058:      0x01234567
(gdb) x/4xb 0x7fffffff058          [BIG ENDIAN COMPUTER, NOT OURS!]
0x7fffffff058:      0x01      0x23      0x45      0x67
```

Generally, you do not need to worry about endianess in a program. It becomes a concern when data is stored as one data type, then accessed as another.

## 2.7 ASCII Character Code

Almost all programs perform a great deal of text string manipulation. Text strings are made up of groups of characters. The first program you wrote was probably a “Hello world” program. If you wrote it in C, you used a statement like:

```
printf("Hello world\n");
```

and in C++:

```
cout << "Hello world\n";
```

When translating either of these statements into machine code, the compiler must do two things:

- store each of the characters in a location in memory where the control unit can access them, and
- generate the machine instructions to write the characters on the screen.

We start by considering how a single character is stored in memory. There are many codes for representing characters, but the most common one is the American Standard Code for Information Interchange — *ASCII* (pronounced “ask’ e”). It uses seven bits to represent each character. Table 2.3 shows the bit patterns for each character in hexadecimal.

It is not the sort of table that you would memorize. However, you should become familiar with some of its general characteristics. In particular, notice that the numerical characters, ‘0’ … ‘9’, are in a contiguous sequence in the code,  $0x30 \dots 0x39$ . The same is true of the lower case alphabetic characters, ‘a’ … ‘z’, and of the upper case characters, ‘A’ … ‘Z’. Notice that the lower case alphabetic characters are numerically higher than the upper case.

The codes in the left-hand column of Table 2.3 (00 through 1f) define *control characters*. The ASCII code was developed in the 1960s for transmitting data from a sender to a receiver. If you read some of names of the control characters, you can imagine how they could be used to control the “dialog” between the sender and receiver. They are generated on a keyboard by holding the control key down while pressing an alphabetic key. For example, `ctrl-d` generates an EOT (End of Transmission) character.

ASCII codes are usually stored in the rightmost seven bits of an eight-bit byte. The eighth bit (the highest-order bit) is called the parity bit. It can be used for error detection in the following way. The sender and receiver would agree ahead of time whether to use even parity or odd parity. Even parity means that an even number of ones is always transmitted in each characters; odd means that an odd number of ones is transmitted. Before transmitting a character in the ASCII code, the sender would adjust the eighth bit such that the total number of ones matched the even or odd agreement. When the code was received, the receiver would count the ones in each eight-bit byte. If the sum did not match the agreement, the receiver knew that one of the bits in the byte

<b>bit</b>	<b>bit</b>	<b>bit</b>	<b>bit</b>		
<b>pat.</b>	<b>char</b>	<b>pat.</b>	<b>char</b>	<b>pat.</b>	<b>char</b>
00	NUL (Null)	20	(Space)	40	@
01	SOH (Start of Hdng)	21	!	41	A
02	STX (Start of Text)	22	"	42	B
03	ETX (End of Text)	23	#	43	C
04	EOT (End of Transmit)	24	\$	44	D
05	ENQ (Enquiry)	25	%	45	E
06	ACK (Acknowledge)	26	&	46	F
07	BEL (Bell)	27	'	47	G
08	BS (Backspace)	28	(	48	H
09	HT (Horizontal Tab)	29	)	49	I
0a	LF (Line Feed)	2a	*	4a	J
0b	VT (Vertical Tab)	2b	+	4b	K
0c	FF (Form Feed)	2c	,	4c	L
0d	CR (Carriage Return)	2d	-	4d	M
0e	SO (Shift Out)	2e	.	4e	N
0f	SI (Shift In)	2f	/	4f	O
10	DLE (Data-Link Escape)	30	0	50	P
11	DC1 (Device Control 1)	31	1	51	Q
12	DC2 (Device Control 2)	32	2	52	R
13	DC3 (Device Control 3)	33	3	53	S
14	DC4 (Device Control 4)	34	4	54	T
15	NAK (Negative ACK)	35	5	55	U
16	SYN (Synchronous idle)	36	6	56	V
17	ETB (End of Trans. Block)	37	7	57	W
18	CAN (Cancel)	38	8	58	X
19	EM (End of Medium)	39	9	59	Y
1a	SUB (Substitute)	3a	:	5a	Z
1b	ESC (Escape)	3b	;	5b	[
1c	FS (File Separator)	3c	<	5c	\
1d	GS (Group Separator)	3d	=	5d	]
1e	RS (Record Separator)	3e	>	5e	^
1f	US (Unit Separator)	3f	?	5f	_
				7f	DEL (Delete)

Table 2.3: ASCII code for representing characters. The bit patterns (bit pat.) are shown in hexadecimal.

had been received incorrectly. Of course, if two bits had been incorrectly received, the error would pass undetected, but the chances of this double error are remarkably small. Modern communication systems are much more reliable, and parity is seldom used when sending individual bytes.

In some environments the high-order bit is used to provide a code for special characters. A little thought will show you that even all eight bits will not support all languages, e.g., Greek, Russian, Chinese. The Unicode character coding has recently been adopted to support documents that use other characters. Java uses Unicode, and C libraries that support Unicode are also available.

A computer system that uses an ASCII video system (most modern computers) can be programmed to send a byte to the screen. The video system interprets the bit pattern as an ASCII code (from Table 2.3) and displays the corresponding character on the screen.

Getting back to the text string, “Hello world\n”, the compiler would store this as a constant `char` array. There must be a way to specify the length of the array. In a *C-style string* this is accomplished by using the sentinel character `NUL` at the end of the string. So the compiler must allocate thirteen bytes for this string. An example of how this string is stored in memory is shown in Figure 2.3. Notice that C uses the `LF` character as a single newline character even though the C syntax requires that the programmer write two characters — ‘\n’. The area of memory shown includes the three bytes immediately following the text string.

<b>Address</b>	<b>Contents</b>
4004a1:	48
4004a2:	65
4004a3:	6c
4004a4:	6c
4004a5:	6f
4004a6:	20
4004a7:	77
4004a8:	6f
4004a9:	72
4004aa:	6c
4004ab:	64
4004ac:	0a
4004ad:	00
4004ae:	25
4004af:	73
4004b0:	00

Figure 2.3: A text string stored in memory by a C compiler, including three “garbage” bytes after the string. Values are shown in hexadecimal. A different compilation will likely place the string in a different memory location.

In Pascal the length of the string is specified by the first byte in the string. It is taken to be an 8-bit unsigned integer. So C-style strings are typically processed by sentinel-controlled loops, and count-controlled string processing loops are more common in Pascal.

The C++ string class has additional features, but the actual text string is stored as a C-style text string within the C++ string instance.

## 2.8 write and read Functions

In Section 2.5 (page 13) we used the `printf` and `scanf` functions to convert between C data types and single characters written on the screen or read from the keyboard. In this section, we introduce the two *system call* functions `write` and `read`. We will use the `write` function to send bytes to the screen and the `read` function to get bytes from the keyboard.

When these low-level functions are used, it is the programmer’s responsibility to convert between the individual characters and the C/C++ data type storage formats. Although this clearly requires more programming effort, we will use them instead of `printf` and `scanf` in order to better illustrate data storage formats.

The C program in Listing 2.3 shows how to display the character ‘A’ on the screen. This program allocates one byte of memory as a `char` variable and names it “`aLetter`.”

This byte is initialized to the bit pattern  $41_{16}$  ('A' from Table 2.3). The `write` function is invoked to display the character on the screen. The arguments to `write` are:

1. `STDOUT_FILENO` is defined in the system header file, `unistd.h`.<sup>3</sup> It is the GNU/Linux file descriptor for standard out (usually the screen). GNU/Linux sees all devices as files. When a program is started the operating system opens a path to standard out and assigns it as file descriptor number 1.
2. `&aLetter` is a memory address. The sequence of one-byte bit patterns starting at this address will be sent to standard out.
3. 1 (one) is the number of bytes that will be sent (to standard out) as a result of this call to `write`.

The program returns a 0 to the operating system.

---

```

1  /*
2   * oneChar.c
3   * Writes a single character on the screen.
4   * Bob Plantz - 4 June 2009
5   */
6
7 #include <unistd.h>
8
9 int main(void)
10 {
11     char aLetter = 'A';
12     write(STDOUT_FILENO, &aLetter, 1); // STDOUT_FILENO is
13                                     // defined in unistd.h
14     return 0;
15 }
```

Listing 2.3: Displaying a single character using C.

Now let's consider a program that echoes each character entered from the keyboard. We will allocate a single `char` variable, read one character into the variable, and then echo the character for the user with a message. The program will repeat this sequence one character at a time until the user hits the return key. The program is shown in Listing 2.4.

---

```

1 /*
2  * echoChar1.c
3  * Echoes a character entered by the user.
4  * Bob Plantz - 4 June 2009
5  */
6
7 #include <unistd.h>
8
9 int main(void)
10 {
11     char aLetter;
12
13     write(STDOUT_FILENO, "Enter one character: ", 21); // prompt user
```

<sup>3</sup>It is generally better to use symbolic names instead of plain numbers. The names provide implicit documentation, and the value may be redefined in some future version.

```

14     read(STDIN_FILENO, &aLetter, 1);           // one character
15     write(STDOUT_FILENO, "You entered: ", 13); // message
16     write(STDOUT_FILENO, &aLetter, 1);
17
18     return 0;
19 }
```

Listing 2.4: Echoing characters entered from the keyboard.

A run of this program gave:

```

bob$ ./echoChar1
Enter one character: a
You entered: abob$
bob$
```

which probably looks like the program is not working correctly to you.

Look more carefully at the program behavior. It illustrates some important issues when using the `read` function. First, how many keys did the user hit? There were actually two keystrokes, the “a” key and the return key. In fact, the program waits until the user hits the return key. The user could have used the delete key to change the character before hitting the return key.

This shows that keyboard input is *line buffered*. Even though the application program is requesting only one character, the operating system does not honor this request until the user hits the return key, thus entering the entire line. Since the line is buffered, the user can edit the line before entering it.

Next, the program correctly echoes the first key hit then terminates. Upon program termination the shell prompt, `bob$`, is displayed. But the return character is still in the input buffer, and the shell program reads it. The result is the same as if the user had simply pressed the return key in response to the shell prompt.

Here is another run where I entered three characters before hitting the return key:

```

bob$ ./echoChar1
Enter one character: abc You entered: abob$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation
Inc. This is free software with ABSOLUTELY NO WARRANTY. For details
type 'warranty'.
```

Again, the program correctly echoes the first character, but the two characters `bc` remain in the input line buffer. When `echoChar1` terminates the shell program reads the remaining characters from the line buffer and interprets them as a command. In this case, `bc` is a program, so the shell executes that program.

An important point of the program in Listing 2.4 is to illustrate the simplistic behavior of the `write` and `read` functions. They work at a very low level. It is your responsibility to design your program to interpret each byte that is written to the screen or read from the keyboard.

## 2.9 Exercises

**2-1** (§2.1) Express the following bit patterns in hexadecimal.

- |                        |                        |
|------------------------|------------------------|
| a) 0100 0101 0110 0111 | c) 1111 1110 1101 1100 |
| b) 1000 1001 1010 1011 | d) 0000 0010 0101 0000 |

**2-2** (§2.1) Express the following bit patterns in binary.

- |         |         |
|---------|---------|
| a) 83af | c) aaaa |
| b) 9001 | d) 5555 |

**2-3** (§2.1) How many bits are represented by each of the following?

- |                      |                           |
|----------------------|---------------------------|
| a) ffffffff          | d) 1111 <sub>16</sub>     |
| b) 7fff58b7def0      | e) 00000000 <sub>2</sub>  |
| c) 1111 <sub>2</sub> | f) 00000000 <sub>16</sub> |

**2-4** (§2.1) How many hexadecimal digits are required to represent each of the following?

- |                    |                |
|--------------------|----------------|
| a) eight bits      | d) ten bits    |
| b) thirty-two bits | e) twenty bits |
| c) sixty-four bits | f) seven bits  |

**2-5** (§2.2) Referring to Equation 2.5, what are the values of r, n and each  $d_i$  for the decimal number 29458254? The hexadecimal number 29458254?

**2-6** (§2.2) Convert the following 8-bit numbers to decimal by hand:

- |             |             |
|-------------|-------------|
| a) 10101010 | e) 10000000 |
| b) 01010101 | f) 01100011 |
| c) 11110000 | g) 01111011 |
| d) 00001111 | h) 11111111 |

**2-7** (§2.2) Convert the following 16-bit numbers to decimal by hand:

- |                     |                     |
|---------------------|---------------------|
| a) 1010101111001101 | e) 1000000000000000 |
| b) 0001001000110100 | f) 0000010000000000 |
| c) 1111111011011100 | g) 1111111111111111 |
| d) 0000011111010000 | h) 0011000000111001 |

**2-8** (§2.2) In Section 2.2 we developed an algorithm for converting from binary to decimal. Develop a similar algorithm for converting from hexadecimal to decimal. Use your new algorithm to convert the following 8-bit numbers to decimal by hand:

- |       |       |
|-------|-------|
| a) a0 | e) 64 |
| b) 50 | f) 0c |
| c) ff | g) 11 |
| d) 89 | h) c8 |

**2-9** (§2.2) In Section 2.2 we developed an algorithm for converting from binary to decimal. Develop a similar algorithm for converting from hexadecimal to decimal. Use your new algorithm to convert the following 16-bit numbers to decimal by hand:

- |         |         |
|---------|---------|
| a) a000 | e) 8888 |
| b) ffff | f) 0190 |
| c) 0400 | g) abcd |
| d) 1111 | h) 5555 |

**2-10** (§2.3) Convert the following unsigned, decimal integers to 8-bit hexadecimal representation.

- |        |        |
|--------|--------|
| a) 100 | e) 255 |
| b) 123 | f) 16  |
| c) 10  | g) 32  |
| d) 88  | h) 128 |

**2-11** (§2.3) Convert the following unsigned, decimal integers to 16-bit hexadecimal representation.

- |          |          |
|----------|----------|
| a) 1024  | e) 256   |
| b) 1000  | f) 65635 |
| c) 32768 | g) 2005  |
| d) 32767 | h) 43981 |

**2-12** (§2.3) Invent a code that would allow us to store letter grades with plus or minus. That is, the grades A, A-, B+, B, B-, ..., D, D-, F. How many bits are required for your code?

**2-13** (§2.3) We have shown how to write only the first sixteen addresses in hexadecimal in Figure 2.1. How would you write the address of the seventeenth byte (byte number sixteen) in hexadecimal? Hint: If we started with zero in the decimal number system we would use a '9' to represent the tenth item. How would you represent the eleventh item in the decimal system?

**2-14** (§2.3) Redo the table in Figure 2.2 such that it shows the memory contents in decimal.

- 2-15** (§2.3) Redo the table in Figure 2.2 such that it shows each of the sixteen bytes containing its byte number. That is, byte number 0 contains zero, number 1 contains one, etc. Show the contents in binary.
- 2-16** (§2.3) Redo the table in Figure 2.2 such that it shows each of the sixteen bytes containing its byte number. That is, byte number 0 contains zero, number 1 contains one, etc. Show the contents in hexadecimal.
- 2-17** (§2.4) You want to allocate an area in memory for storing any number between 0 and 4,000,000,000. This memory area will start at location `0x2ffffeb96`. Give the addresses of each byte of memory that will be required.
- 2-18** (§2.4) You want to allocate an area in memory for storing an array of 30 bytes. The first byte will have the value `0x00` stored in it, the second `0x01`, the third `0x02`, etc. This memory area will start at location `0x001000`. Show what this area of memory looks like.
- 2-19** (§2.4) In Section 2.4 we invented a binary code for representing letter grades. Referring to that code, express each of the grades as an 8-bit unsigned decimal integer.
- 2-20** (§2.5) Enter the program in Listing 2.2 and check your answers for Exercise 2-6. Note that `printf` and `scanf` do not have a conversion for binary. Check the answers in hexadecimal.
- 2-21** (§2.5) Enter the program in Listing 2.2 and check your answers for Exercise 2-7. Note that `printf` and `scanf` do not have a conversion for binary. Check the answers in hexadecimal.
- 2-22** (§2.5) Enter the program in Listing 2.2 and check your answers for Exercise 2-8.
- 2-23** (§2.5) Enter the program in Listing 2.2 and check your answers for Exercise 2-9.
- 2-24** (§2.5) Enter the program in Listing 2.2 and check your answers for Exercise 2-10.
- 2-25** (§2.5) Enter the program in Listing 2.2 and check your answers for Exercise 2-11.
- 2-26** (§2.5) Modify the program in Listing 2.2 so that it also displays the addresses of the `x` and `y` variables. Note that addresses are typically displayed in hexadecimal. How many bytes does the compiler allocate for each of the `ints`?
- 2-27** (§2.6) Enter the program in Listing 2.1. Follow through the program with `gdb` as in the example in Section 2.6. Using the numbers you get, explain where the variables `aInt` and `aFloat` are stored in memory and what is stored in each location.
- 2-28** (§2.7) Write a program in C that creates a display similar to Figure 2.3. Hints: use a `char*` variable to process the string one character at a time; use `%08x` to format the display of the address.
- 2-29** (§2.6) Enter the program in Listing 2.4. Explain why there seems to be an extra prompt in the program. Set breakpoints at both the `read` statement and at the following `write` statement. Examine the contents of the `aLetter` variable before the `read` and after it. Notice that the behavior of `gdb` seems very strange when dealing with the `read` statement. Explain the behavior. Hint: Both `gdb` and the program you are debugging use the same keyboard for input.

**2-30** (§2.8) Modify the program in Listing 2.4 so that it prompts the user to enter an entire line, reads the line, then echoes the entire line. Read only one byte at a time from the keyboard.

**2-31** (§2.8) This is similar to Exercise 2-30 except that when the newline character is read from the keyboard (and stored in memory), the program replaces the newline character with a NUL character. The program has now read a line from the keyboard and stored it as a C-style text string. If your algorithm is correct, you will be able to read the text string using the read low-level function and display it with the `printf` library function thusly (assuming the variable where the string is stored is named `theString`),

```
printf("%s\n", theString);
```

and have only one newline. Notice that this program discards the newline generated when the user hits the return key. This is the same behavior you would see if you used

```
scanf("\%s", theString);
```

in C, or

```
cin >> theString;
```

in C++ to read the input text from the keyboard.

**2-32** (§2.8) Write a C program that prompts the user to enter a line of text on the keyboard then echoes the entire line. The program should continue echoing each line until the user responds to the prompt by not entering any text and hitting the return key. Your program should have two functions, `writeStr` and `readLn`, in addition to the `main` function. The text string itself should be stored in a `char` array in `main`. Both functions should operate on NUL-terminated text strings.

- `writeStr` takes one argument, a pointer to the string to be displayed and it returns the number of characters actually displayed. It uses the `write` system call function to write characters to the screen.
- `readLn` takes two arguments, one that points to the `char` array where the characters are to be stored and one that specifies the maximum number of characters to store in the `char` array. Additional keystrokes entered by the user should be read from the OS input buffer and discarded. `readLn` should return the number of characters actually stored in the `char` array. `readLn` should not store the newline character ('\n'). It uses the `read` system call function to read characters from the keyboard.

# Chapter 3

# Computer Arithmetic

We next turn our attention to a code for storing decimal integers. Since all storage in a computer is by means of on/off switches, we cannot simply store integers as decimal digits. Exercises 3-1 and 3-2 should convince you that it will take some thought to come up with a good code that uses simple on/off switches to represent decimal numbers.

Another very important issue when talking about computer arithmetic was pointed out in Section 2.3 (page 9). Namely, the programmer must decide how many bits will be used for storing the numbers before performing any arithmetic operations. This raises the possibility that some results will not fit into the allocated number of bits. As you will see in Section 9.2 (page 214), the computer hardware provides for this possibility with the Carry Flag (CF) and Overflow Flag (OF) in the `rflags` register located in the CPU. Depending on what you intend the bit patterns to represent, either the Carry Flag or the Overflow Flag (not both) will indicate the correctness of the result. However, most high level languages, including C and C++, do not check the CF and OF after performing arithmetic operations.

## 3.1 Addition and Subtraction

Computers perform addition in the binary number system.<sup>1</sup> The operation is really quite easy to understand if you recall all the details of performing addition in the decimal number system by hand. Since most people perform addition on a calculator these days, let us review all the steps required when doing it by hand. Consider two two-digit numbers,  $x = 67$  and  $y = 79$ . Adding these by hand on paper would look something like:

$$\begin{array}{r} 1 \quad 1 & \leftarrow \text{carries} \\ 67 & \leftarrow x \\ + \quad 79 & \leftarrow y \\ \hline 46 & \leftarrow \text{sum} \end{array}$$

We start by working from the right, adding the two decimal digits in the ones place.  $7 + 9$  exceeds 10 by 6. We show this by placing a 6 in the ones place in the sum and carrying a 1 to the tens place. Next we add the three decimal digits in the tens place, 1 (the carry into the tens place from the ones place) + 6 + 7. The sum of these three digits exceeds 10 by 4, which we show by placing a 4 in the tens place in the sum and recording the fact that there is an ultimate carry of one. Recall that we had decided to use only two

---

<sup>1</sup>Most computer architectures provide arithmetic operations in other number systems, but these are somewhat specialized. We will not consider them in this book.

digits, so there is no hundreds place. Using the notation of Equation 2.5 (page 8), we describe addition of two decimal integers in Algorithm 3.1.

**Algorithm 3.1:** Add fixed-width decimal integers.

**given** : N, number of digits.

Starting in the ones place:

1 **for**  $i=0$  **to**  $(N-1)$  **do**

2 | sum<sub>i</sub> ⇐ (x<sub>i</sub> + y<sub>i</sub>) % 10 ;

3 | carry  $\Leftarrow (x_i + y_i) / 10$  ;

**4** |  $i \Leftarrow i + 1;$

// mod operation

// div operation

Notice that:

- Algorithm 3.1 works because we use a positional notation when writing numbers — a digit one place to the left counts ten times more.
  - Carry from the current position one place to the left is always 0 or 1.
  - The reason we use 10 in the / and % operations is that there are exactly ten digits in the decimal number system : 0, 1, 2, ..., 9.
  - Since we are working in an N-digit system, we must restrict our result to N digits. The final carry (0 or 1) must be stated in addition to the N-digit result.

By changing "10" to "2" we get Algorithm 3.2 for addition in the binary number system. The only difference is that a digit one place to the left counts two times more.

**Algorithm 3.2:** Add fixed-width binary integers.

**given** : N, number of bits.

Starting in the ones place:

**1 for**  $i=0$  **to**  $(N-1)$  **do**

**2** |  $\text{sum}_i \Leftarrow (\mathbf{x}_i + \mathbf{y}_i) \% 2 ;$

—

**Example 3-a** \_\_\_\_\_

Comp

0	0001	111	← carries
	1010	1011	← x
+	0100	1101	← y
	1111	1000	← sum

This is how the algorithm was applied.

ones place.

$$\begin{aligned} \text{sum}_0 &= (1 + 1) \% 2 = 0 \\ \text{carry} &= (1 + 1) / 2 = 1 \end{aligned}$$

twos place:

```
sum1 = (1 + 1 + 0) % 2 = 0
carry = (1 + 1 + 0) / 2 = 1
```

fours place:

```
sum2 = (1 + 0 + 1) % 2 = 0
carry = (1 + 0 + 1) / 2 = 1
```

eights place:

```
sum3 = (1 + 1 + 1) % 2 = 1
carry = (1 + 1 + 1) / 2 = 1
```

sixteens place:

```
sum4 = (1 + 0 + 0) % 2 = 1
carry = (1 + 0 + 0) / 2 = 0
```

thirty-twos place:

```
sum5 = (0 + 1 + 0) % 2 = 1
carry = (0 + 1 + 0) / 2 = 0
```

sixty-fours place:

```
sum6 = (0 + 0 + 0) % 2 = 1
carry = (0 + 0 + 0) / 2 = 0
```

one hundred twenty-eights place:

```
sum7 = (0 + 1 + 0) % 2 = 1
carry = (0 + 1 + 0) / 2 = 0
```

In this eight-bit example the result is 1111 1000, and there is no carry beyond the eight bits. The lack of carry is recorded in the rflags register by setting the CF bit to zero.



It should not surprise you that this algorithm also works for hexadecimal. In fact, it works for any radix, as shown in Algorithm 3.3.

### **Algorithm 3.3:** Add fixed-width integers in any radix.

**given:** N, number of digits.

Starting in the ones place:

```
1 for i=0 to (N-1) do
2   sumi ⇐ (xi + yi) % radix ;           // mod operation
3   carry ⇐ (xi + yi) / radix ;          // div operation
4   i ⇐ i + 1;
```

For hexadecimal:

- A digit one place to the left counts sixteen times more.
- We use 16 in the / and % operations because there are sixteen digits in the hexadecimal number system: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f.

Addition in hexadecimal brings up a notational issue. For example,

d + 9 = ??    Oops, how do we write this?

Although it is certainly possible to perform all the computations using hexadecimal notation, most people find it a little awkward. After you have memorized Table 3.1 it is much easier to :

- convert the (hexadecimal) digit to its equivalent decimal value

- apply our algorithm
- convert the results back to hexadecimal

Actually, we did this when applying the algorithm to binary addition. Since the conversion of binary digits to decimal digits is trivial, you probably did not think about it. But the conversion of hexadecimal digits to decimal is not as trivial. To see how it works, first recall that the conversion from hexadecimal to binary is straightforward. (You should have memorized Table 2.1 by now.) So we will consider conversion from binary to decimal.

As mentioned above, the relative position of each bit has significance. The rightmost bit represents the ones place, the next one to the left the fours place, then the eights place, etc. In other words, each bit represents  $2^n$ , where  $n = 0, 1, 2, 3, \dots$  and we start from the right. So the binary number 1011 represents:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \quad (3.1)$$

This is easily converted to decimal by simply working out the arithmetic in decimal:

$$\begin{aligned} 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 &= 8 + 0 + 2 + 1 \\ &= 11 \end{aligned} \quad (3.2)$$

From Table 2.1 on page 7 we see that  $1011_2 = b_{16}$ , and we conclude that  $b_{16} = 11_{10}$ . We can add a “decimal” column to the table, giving Table 3.1.

Four binary digits (bits)	One hexadecimal digit	Decimal equivalent
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	a	10
1011	b	11
1100	c	12
1101	d	13
1110	e	14
1111	f	15

Table 3.1: Correspondence between binary, hexadecimal, and unsigned decimal values for the hexadecimal digits.

**Example 3-b**

Compute the sum of  $x = 0abcd$  and  $y = 0x6089$ .

*Solution:*

$$\begin{array}{r}
 1 \quad 011 & \leftarrow \text{carries} \\
 abcd & \leftarrow x \\
 + \quad 6089 & \leftarrow y \\
 \hline
 0c56 & \leftarrow \text{sum}
 \end{array}$$

Now we can see how Algorithm 3.3 with radix = 16 was applied in order to add the hexadecimal numbers, abcd and 6089. Having memorized Table 3.1, we will convert between hexadecimal and decimal “in our heads.”

ones place:

$$\begin{aligned}
 \text{sum}_0 &= (d + 9) \% 16 = 6 \\
 \text{carry} &= (d + 9) / 16 = 1
 \end{aligned}$$

sixteens place:

$$\begin{aligned}
 \text{sum}_1 &= (1 + c + 8) \% 16 = 5 \\
 \text{carry} &= (1 + c + 8) / 16 = 1
 \end{aligned}$$

two hundred fifty-sixes place:

$$\begin{aligned}
 \text{sum}_2 &= (1 + b + 0) \% 16 = c \\
 \text{carry} &= (1 + b + 0) / 16 = 0
 \end{aligned}$$

four thousand ninety-sixes place:

$$\begin{aligned}
 \text{sum}_3 &= (0 + a + 6) \% 16 = 0 \\
 \text{carry} &= (0 + a + 6) / 16 = 1
 \end{aligned}$$

This four-digit example has an ultimate carry of 1, which is recorded in the rflags register by setting the CF to one. The arithmetic was performed by first converting each digit to decimal. It is then a simple matter to convert each decimal value back to hexadecimal (see Table 3.1) to express the final answer in hexadecimal.

Let us now turn to the subtraction operation. As you recall from subtraction in the decimal number system, you must sometimes *borrow* from the next higher-order digit in the minuend. This is shown in Algorithm 3.4.

**Algorithm 3.4:** Subtract fixed-width integers in any radix.

---

```

given: N, number of bits.
Starting in the ones place, subtract Y from X:
1 for i=0 to (N-1) do
  2 if  $y_i \leq x_i$  then
    3 differencei  $\leftarrow x_i - y_i;$ 
    4 borrow  $\leftarrow 0;$ 
  5 else
    6 j  $\leftarrow i + 1;$ 
    7 while  $x_j = 0$  do
      8 j  $\leftarrow j + 1;$ 
    9 for j to i do
      10  $x_j \leftarrow x_j - 1;$ 
      11 j  $\leftarrow j - 1;$ 
      12  $x_j \leftarrow x_j + \text{radix};$ 
  13 i  $\leftarrow i + 1;$ 

```

---

This algorithm is not as complicated as it first looks.

**Example 3-c**

Subtract  $y = 10101011$  from  $x = 11001101$ .

*Solution:*

$$\begin{array}{r}
 0 & 0100 & 010 & \leftarrow \text{borrows} \\
 & 1100 & 1101 & \leftarrow x \\
 - & 1010 & 1011 & \leftarrow y \\
 \hline
 0010 & 0010 & & \leftarrow \text{difference}
 \end{array}$$

The bits have been grouped to improve readability. A 1 in the borrow row indicates that 1 was borrowed from the minuend in that place, which becomes 2 in the next place to the right. A 0 indicates that no borrow was required. This is how the algorithm was applied.

ones place:

$$\text{difference}_0 = 1 - 1 = 0$$

twos place:

Borrow from the fours place in the minuend.

The borrow becomes 2 in the twos place.

$$\text{difference}_1 2 - 1 = 1$$

fours place:

Since we borrowed 1 from here, the minuend has a 0 left.

$$\text{difference}_2 = 0 - 0 = 0$$

eights place:

$$\text{difference}_3 = 1 - 1 = 0$$

sixteens place:

$$\text{difference}_4 = 0 - 0 = 0$$

thirty-twos place:

Borrow from the sixty-fours place in the minuend.

The borrow becomes 2 in the thirty-twos place.

$$\text{difference}_5 = 2 - 1 = 1$$

sixty-fours place:

Since we borrowed 1 from here, the minuend has a 0 left.

$$\text{difference}_6 = 0 - 0 = 0$$

one hundred twenty-eights place:

$$\text{difference}_7 = 1 - 1 = 0$$

□

This, of course, also works for hexadecimal, but remember that a digit one place to the left counts sixteen times more. For example, consider  $x = 0x6089$  and  $y = 0xab5d$ :

$$\begin{array}{r}
 1 \quad 101 & \leftarrow \text{borrows} \\
 6089 & \leftarrow x \\
 - ab5d & \leftarrow y \\
 \hline
 b52c & \leftarrow \text{difference}
 \end{array}$$

Notice in this second example that we had to borrow from “beyond the width” of the two values. That is, the two values are each sixteen bits wide, and the result must also be sixteen bits. Whether there is borrow “from outside” to the high-order digit is recorded in the CF of the rflags register whenever a subtract operation is performed:

- no borrow from outside  $\rightarrow \text{CF} = 0$
- borrow from outside  $\rightarrow \text{CF} = 1$

Another way to state this is for unsigned numbers:

- if the subtrahend is equal to or less than the minuend the CF is set to zero
- if the subtrahend is larger than the minuend the CF bit is set to one

## 3.2 Arithmetic Errors — Unsigned Integers

The binary number system was introduced in Section 2.2 (page 8). You undoubtedly realize by now that it probably is a good system for storing unsigned integers. Don’t forget that it does not matter whether we think of the integers as being in decimal, hexadecimal, or binary since they are mathematically equivalent. If we are going to store integers this way, we need to consider the arithmetic properties of addition and subtraction in the binary number system. Since a computer performs arithmetic in binary (see footnote 1 on page 30), we might ask whether addition yields arithmetically correct results when representing decimal numbers in the binary number system. We will use four-bit values to simplify the discussion. Consider addition of the two numbers:

$$\begin{array}{rcl}
 0100_2 & = & 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 4_{10} \\
 + 0010_2 & = & 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = + 2_{10} \\
 \hline
 0110_2 & = & 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6_{10}
 \end{array}$$

and  $\text{CF} = 0$ .

So far, the binary number system looks reasonable. Let’s try two larger four-bit numbers:

$$\begin{array}{rcl}
 0100_2 & = & 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 4_{10} \\
 + 1110_2 & = & 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = +14_{10} \\
 \hline
 0010_2 & = & 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 2_{10}
 \end{array}$$

and CF = 1. The result, 2, is arithmetically incorrect. The problem here is that the addition has produced carry beyond the fourth bit. Since this is not taken into account in the result, the answer is wrong.

Now consider subtraction of the two numbers:

$$\begin{array}{rcl}
 0100_2 & = & 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 4_{10} \\
 - 1110_2 & = & 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -14_{10} \\
 \hline
 0110_2 & = & 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6_{10}
 \end{array}$$

and CF = 1.

The result, 6, is arithmetically incorrect. The problem in this case is that the subtraction has had to borrow from beyond the fourth bit. Since this is not taken into account in the result, the answer is wrong.

From the discussion in Section 3.1 (page 30) you should be able to convince yourself that these four-bit arithmetic examples generalize to any size arithmetic performed by the computer. After adding two numbers, the Carry Flag will always be set to zero if there is no ultimate carry, or it will be set to one if there is ultimate carry. Subtraction will set the Carry Flag to zero if no borrow from the “outside” is required, or one if borrow is required. These examples illustrate the principle:

- When adding or subtracting two unsigned integers, the result is arithmetically correct if and only if the Carry Flag (CF) is set to zero.

It is important to realize that the CF and OF bits in the rflags register are always set to the appropriate value, 0 or 1, each time an addition or subtraction is performed by the CPU. In particular, the CPU will not ignore the CF when there is no carry, it will actively set the CF to zero.

### 3.3 Arithmetic Errors — Signed Integers

When representing signed decimal integers we have to use one bit for the sign. We might be tempted to simply use the highest-order bit for this purpose. Let us say that 0 means + and 1 means -. We will try adding (+2) and (-2):

$$\begin{array}{rcl}
 0010_2 & = & (+2)_{10} \\
 + 1010_2 & = & +(-2)_{10} \\
 \hline
 1100_2 & = & (-4)_{10}
 \end{array}$$

The result, -4, is arithmetically incorrect. We should note here that the problem is the way in which the computer does addition — it performs binary addition on the bit patterns that in themselves have no inherent meaning. There are computers that use this particular code for storing signed decimal integers. They have a special “signed add” instruction. By the way, notice that such computers have both a +0 and a -0!

Most computers, including the x86, use another code for representing signed decimal integers — the *two's complement* code. To see how this code works, we start with an example using the decimal number system.

Say that you have a cassette player and wish to represent both positive and negative positions on the tape. It would make sense to somehow fast-forward the tape to its center and call that point “zero.” Most cassette players have a four decimal digit counter

that represents tape position. The counter, of course, does not give actual tape position, but a “coded” representation of the tape position. Since we wish to call the center of the tape “zero,” we push the counter reset button to set it to 0000.

Now, moving the tape forward — the positive direction — will cause the counter to increment. And moving the tape backward — the negative direction — will cause the counter to decrement. In particular, if we start at zero and move to “+1” the “code” on the tape counter will show 0001. On the other hand, if we start at zero and move to “-1” the “code” on the tape counter will show 9999.

Using our tape code system to perform the arithmetic in the previous example — (+2) + (-2):

1. Move the tape to (+2); the counter shows 0002.
2. Add (-2) by decrementing the tape counter by two.

The counter shows 0000, which is 0 according to our code.

Next we will perform the same arithmetic starting with (-2), then adding (+2):

1. Move the tape to (-2); the counter shows 9998.
2. Add (+2) by incrementing the tape counter by two.

The counter shows 0000, but there is a carry. ( $9998 + 2 = 0000$  with carry = 1.) If we ignore the carry, the answer is correct. This example illustrates the principle:

- When adding two signed integers in the two’s complement notation, carry is irrelevant.

The two’s complement code uses this pattern for representing signed decimal integers in bit patterns. The correspondence between signed decimal (two’s complement), hexadecimal, and binary for four-bit values is shown in Table 3.2.

<b>Four binary digits</b> (bits)	<b>One hexadecimal digit</b>	<b>Decimal equivalent</b>
1000	8	-8
1001	9	-7
1010	a	-6
1011	b	-5
1100	c	-4
1101	d	-3
1110	e	-2
1111	f	-1
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

Table 3.2: Four-bit signed integers, two’s complement notation.

We make the following observations about Table 3.2:

- The high-order bit of each positive number is 0.

- The high-order bit of each negative number is 1.
- However, changing the sign of (negating) a number is more complicated than simply changing the high-order bit.
- The code allows for one more negative number than positive numbers.
- The range of integers,  $x$ , that can be represented in this code (with four bits) is

$$-8_{10} \leq x \leq +7_{10} \quad (3.3)$$

or

$$-2^{(4-1)} \leq x \leq +(2^{(4-1)} - 1) \quad (3.4)$$

The last observation can be generalized for  $n$  bits to:

$$-2^{(n-1)} \leq x \leq +(2^{(n-1)} - 1) \quad (3.5)$$

In the two's complement code, the negative of any integer,  $x$ , is defined as

$$x + (-x) = 2^n \quad (3.6)$$

Notice that  $2^n$  written in binary is "1" followed by  $n$  zeros. That is, it requires  $n+1$  bits to represent. Another way of saying this is, "in the  $n$ -bit two's complement code adding a number to its negative produces  $n$  zeros and carry."

We now derive a method for computing the negative of a number in the two's complement code. Solving Equation 3.6 for  $-x$ , we get:

$$-x = 2^n - x \quad (3.7)$$

For example, if we wish to compute  $-1$  in binary (in the two's complement code) in 8 bits, we perform the arithmetic:

$$-1_{10} = 10000000_2 - 00000001_2 = 11111111_2 \quad (3.8)$$

or in hexadecimal:

$$-1_{16} = 100_{16} - 01_{16} = f_{16} \quad (3.9)$$

This subtraction is error prone, so let's perform a few algebraic manipulations on Equation 3.7, which defines the negation operation. First, we subtract one from both sides:

$$-x - 1 = 2^n - x - 1 \quad (3.10)$$

Rearranging a little:

$$\begin{aligned} -x - 1 &= 2^n - 1 - x \\ &= (2^n - 1) - x \end{aligned} \quad (3.11)$$

Now, consider the quantity  $(2^n - 1)$ . Since  $2^n$  is written in binary as one (1) followed by  $n$  zeros,  $(2^n - 1)$  is written as  $n$  ones. For example, for  $n = 8$ :

$$2^8 - 1 = 11111111_2 \quad (3.12)$$

Thus, we can express the right-hand side of Equation 3.11 as

$$2^n - 1 - x = 111\dots111_2 - x \quad (3.13)$$

where  $111\dots111_2$  designates  $n$  ones.

You can see how easy the subtraction on the right-hand side of Equation 3.13 is if we consider the previous example of computing -1 in binary in eight bits. Let  $x = 1$ , giving:

$$11111111_2 - 00000001_2 = 11111110_2 \quad (3.14)$$

or in hexadecimal:

$$f_{16} - 01_{16} = fe_{16} \quad (3.15)$$

Another (simpler) way to look at this is

$$2^n - 1 - x = \text{"flip all the bits in } x\text{"} \quad (3.16)$$

The value of the right-hand side of Equation 3.16 is called the *reduced radix complement* of  $x$ . Since the radix is two, it is common to call this the *one's complement* of  $x$ . From Equation 3.11 we see that this computation — the reduced radix complement of  $x$  — gives

$$-x - 1 = \text{the reduced radix complement of } x \quad (3.17)$$

Now we can easily compute  $-x$  by adding one to both sides of Equation 3.17:

$$\begin{aligned} -x - 1 + 1 &= (\text{the reduced radix complement of } x) + 1 \\ &= -x \end{aligned} \quad (3.18)$$

This leads us to Algorithm 3.5 for negating any integer stored in the two's complement,  $n$ -bit code.

---

**Algorithm 3.5:** Negate a number in binary (compute 2's complement).

---

We use  $x'$  to denote the complement of  $x$ .

- 1  $x \Leftarrow x';$
  - 2  $x \Leftarrow x + 1;$
- 

This process — computing the one's complement, then adding one — is called computing the two's complement.

**Be Careful!**

- “In two's complement” describes the storage code.
- “Taking the two's complement” is an active computation. If the value the computation is applied to an integer stored in the two's complement notation, this computation is mathematically equivalent to negating the number.

Combining Algorithm 3.5 with observations about Table 3.2 above, we can easily compute the decimal equivalent of any integer stored in the two's complement notation by applying Algorithm 3.6.

---

**Algorithm 3.6:** Signed binary-to-decimal conversion.

---

- 1 **if the high-order bit is zero then**
  - 2   | compute the decimal equivalent of the number;
  - 3 **else**
  - 4   | take the two's complement (negate the number);
  - 5   | compute the decimal equivalent of this result;
  - 6   | place a minus sign in front of the decimal equivalent;
-

**Example 3-d**

The 16-bit integer  $5678_{16}$  is stored in two's complement notation. Convert it to a signed, decimal integer.

*Solution:*

Since the high-order bit is zero, we simply compute the decimal equivalent:

$$\begin{aligned} 5678_{16} &= 5 \times 4096 + 6 \times 256 + 7 \times 16 + 8 \times 1 \\ &= 20480 + 1536 + 112 + 8 \\ &= +22136_{10} \end{aligned}$$


---

**Example 3-e**

The 16-bit integer  $8765_{16}$  is stored in two's complement notation. Convert it to a signed, decimal integer.

*Solution:*

Since the high-order bit is one, we first negate the number in the two's complement format.

$$\begin{aligned} \text{Take the one's complement} &\Rightarrow 789a_{16} \\ \text{Add one} &\Rightarrow 789b_{16} \end{aligned}$$

Compute the decimal equivalent.

$$\begin{aligned} 789b_{16} &= \times 4096 + \times 256 + 9 \times 16 + 11 \times 1 \\ &= 28672 + 2048 + 144 + 11 \\ &= +30875_{10} \end{aligned}$$

Place a minus sign in front of the number (since we negated it in the two's complement domain).

$$8765_{16} = -30875_{10}$$


---



Algorithm 3.7 shows how to convert a signed decimal number to two's complement binary.

**Algorithm 3.7:** Signed decimal-to-binary conversion.

- 1 **if** the number is positive **then**
- 2   | simply convert it to binary;
- 3 **else**
- 4   | negate the number;
- 5   | convert the result to binary;
- 6   | compute the two's complement of result in the binary domain;

**Example 3-f**

Convert the signed, decimal integer +31693 to a 16-bit integer in two's complement notation. Give the answer in hexadecimal.

*Solution:*

Since this is a positive number, we simply convert it. The answer is to be given in hexadecimal, so we will repetitively divide by 16 to get the answer.

$$31693 \div 16 = 1980 \text{ with remainder } 13$$

$$1980 \div 16 = 123 \text{ with remainder } 12$$

$$123 \div 16 = 7 \text{ with remainder } 11$$

$$7 \div 16 = 0 \text{ with remainder } 7$$

So the answer is

$$31693_{10} = 7\text{bcd}_{16}$$

**Example 3-g**

Convert the signed, decimal integer -250 to a 16-bit integer in two's complement notation. Give the answer in hexadecimal.

*Solution:*

Since this is a negative number, we first negate it, giving +250. Then we convert this value. The answer is to be given in hexadecimal, so we will repetitively divide by 16 to get the answer.

$$250 \div 16 = 15 \text{ with remainder } 10$$

$$15 \div 16 = 0 \text{ with remainder } 15$$

This gives us

$$250_{10} = 00\text{fa}_{16}$$

Now we take the one's complement: 00fa  $\Rightarrow$  ff05  
and add one:  $\Rightarrow$  ff06 So the answer is

$$-250_{10} = \text{ff06}_{16}$$



## 3.4 Overflow and Signed Decimal Integers

The number of bits used to represent a value is determined at the time a program is written. So when performing arithmetic operations we cannot simply add more digits (bits) if the result is too large, as we can do on paper. You saw in Section 3.1 (page 30) that the CF indicates when the sum of two unsigned integers exceeds the number of bits allocated to it.

In Section 3.3 (page 37) you saw that carry is irrelevant when working with signed integers. You also saw that adding two signed numbers can produce an incorrect result. That is, the sum may exceed the range of values that can be represented in the allocated number of bits.

The flags register, `rflags`, provides a bit, the Overflow Flag (OF), for detecting whether the sum of two n-bit, signed numbers stored in the two's complement code has exceeded the range allocated for it. Each operation that affects the overflow flag sets the bit equal to the exclusive or of the carry into the highest-order bit of the operands and the ultimate carry. For example, when adding the two 8-bit numbers,  $15_{16}$  and  $6f_{16}$ , we get:

$$\begin{array}{r} \text{carry} \rightarrow 0 \quad 1 \leftarrow \text{penultimate carry} \\ \begin{array}{r} 0001 \quad 0101 \\ + \quad 0110 \quad 1111 \\ \hline 1000 \quad 0100 \end{array} \end{array} \begin{array}{l} \leftarrow x \\ \leftarrow y \\ \leftarrow \text{sum} \end{array}$$

In this example, there is a carry of zero and a *penultimate* (next to last) *carry* of one. The OF flag is equal to the exclusive or of carry and penultimate carry:

$$OF = CF \wedge \text{penultimate carry} \quad (3.19)$$

where ' $\wedge$ ' is the exclusive or operator. In the above example

$$\begin{aligned} OF &= 0 \wedge 1 \\ &= 1 \end{aligned} \quad (3.20)$$

There are three cases when adding two numbers:

**Case 1: The two numbers are of opposite sign.** We will let x be the negative number and y the positive number. Then we can express x and y in binary as:

$$\begin{aligned} x &= 1 \dots \\ y &= 0 \dots \end{aligned}$$

That is, the high-order bit of one number is 1 and the high-order bit of the other is 0, regardless of what the other bits are. Now, if we add x and y, there are two possible results with respect to carry:

1. If the penultimate carry is zero:

$$\begin{array}{r} \text{carry} \rightarrow 0 \quad 0 \leftarrow \text{penultimate carry} \\ \begin{array}{r} 0 \quad \dots \\ + \quad 1 \quad \dots \\ \hline 1 \quad \dots \end{array} \end{array} \begin{array}{l} \leftarrow x \\ \leftarrow y \\ \leftarrow \text{sum} \end{array}$$

this addition would produce  $OF = 0 \wedge 0 = 0$ .

2. If the penultimate carry is one:

$$\begin{array}{r} \text{carry} \rightarrow 1 \quad 1 \leftarrow \text{penultimate carry} \\ \begin{array}{r} 0 \quad \dots \\ + \quad 1 \quad \dots \\ \hline 0 \quad \dots \end{array} \end{array} \begin{array}{l} \leftarrow x \\ \leftarrow y \\ \leftarrow \text{sum} \end{array}$$

this addition would produce  $OF = 1 \wedge 1 = 0$ .

We conclude that adding two integers of opposite sign always yields 0 for the overflow flag.

Next, notice that since y is positive and x negative:

$$0 \leq y \leq +(2^{(n-1)} - 1) \quad (3.21)$$

$$-2^{(n-1)} \leq x < 0 \quad (3.22)$$

Adding inequalities (3.21) and (3.22), we get:

$$-2^{(n-1)} \leq x + y \leq +(2^{(n-1)} - 1) \quad (3.23)$$

Thus, the sum of two integers of opposite sign remains within the range of signed integers, and there is no overflow ( $OF = 0$ ).

**Case 2: Both numbers are positive.** Since both are positive, we can express x and y in binary as:

$$x = 0 \dots$$

$$y = 0 \dots$$

That is, the high-order bit is 0, regardless of what the other bits are. Now, if we add x and y, there are two possible results with respect to carry:

1. If the penultimate carry is zero:

$$\begin{array}{r} \text{carry} \longrightarrow 0 \quad 0 \quad \leftarrow \text{penultimate carry} \\ \quad \quad \quad 0 \quad \dots \quad \leftarrow x \\ + \quad \quad \quad 0 \quad \dots \quad \leftarrow y \\ \hline \quad \quad \quad 0 \quad \dots \quad \leftarrow \text{sum} \end{array}$$

this addition would produce  $OF = 0 \wedge 0 = 0$ . The high-order bit of the sum is zero, so it is a positive number, and the sum is within range.

2. If the penultimate carry is one:

$$\begin{array}{r} \text{carry} \longrightarrow 0 \quad 1 \quad \leftarrow \text{penultimate carry} \\ \quad \quad \quad 0 \quad \dots \quad \leftarrow x \\ + \quad \quad \quad 0 \quad \dots \quad \leftarrow y \\ \hline \quad \quad \quad 1 \quad \dots \quad \leftarrow \text{sum} \end{array}$$

this addition would produce  $OF = 0 \wedge 1 = 1$ . The high-order bit of the sum is one, so it is a negative number. Adding two positive numbers cannot yield a negative sum, so this sum has exceeded the allocated range.

**Case 3: Both numbers are negative.** Since both are negative, we can express x and y in binary as:

$$x = 1 \dots$$

$$y = 1 \dots$$

That is, the high-order bit is 1, regardless of what the other bits are. Now, if we add x and y, there are two possible results with respect to carry:

1. If the penultimate carry is zero:

$$\begin{array}{r} \text{carry} \longrightarrow 1 \quad 0 \quad \leftarrow \text{penultimate carry} \\ \quad \quad \quad 1 \quad \dots \quad \leftarrow x \\ + \quad \quad \quad 1 \quad \dots \quad \leftarrow y \\ \hline 0 \quad \dots \quad \leftarrow \text{sum} \end{array}$$

this addition would produce  $0F = 1 \wedge 0 = 1$ . The high-order bit of the sum is zero, so it is a positive number. Adding two negative numbers cannot yield a negative sum, so this sum has exceeded the allocated range.

2. If the penultimate carry is one:

$$\begin{array}{r} \text{carry} \longrightarrow 1 \quad 1 \quad \leftarrow \text{penultimate carry} \\ \quad \quad \quad 1 \quad \dots \quad \leftarrow x \\ + \quad \quad \quad 1 \quad \dots \quad \leftarrow y \\ \hline 1 \quad \dots \quad \leftarrow \text{sum} \end{array}$$

this addition would produce  $0F = 1 \wedge 1 = 0$ . The high-order bit of the sum is one, so it is a negative number, and the sum is within range.

### 3.4.1 The Meaning of CF and OF

These results, together with the results from Section 3.2 (page 36), yield the following rules when adding or subtraction two n-bit integers:

- If your algorithm treats the result as unsigned, the Carry Flag (CF) is zero if and only if the result is within the n-bit range; OF is irrelevant.
- If your algorithm treats the result as signed (using the two's complement code), the Overflow Flag (OF) is zero if and only if the result is within the n-bit range; CF is irrelevant.

The CPU does not consider integers as either signed or unsigned. Both the CF and OF are set according to the rules of binary arithmetic by each arithmetic operation. The distinction between signed and unsigned is completely determined by the program. After each addition or subtraction operation the program should check the state of the CF for unsigned integers or the OF of signed integers and at least indicate when the sum is in error. Most high-level languages do not perform this check, which can lead to some obscure program bugs.

**Be Careful!** Do not confuse positive signed numbers with unsigned numbers. The range for unsigned 32-bit integers is 0 - 4294967295, and for signed 32-bit integers the range is -2147483648 - +2147483647.

The codes used for both unsigned integers and signed integers are circular in nature. That is, for a given number of bits, each code "wraps around." This can be seen pictorially in the "Decoder Ring" shown in Figure 3.1 for three-bit numbers.

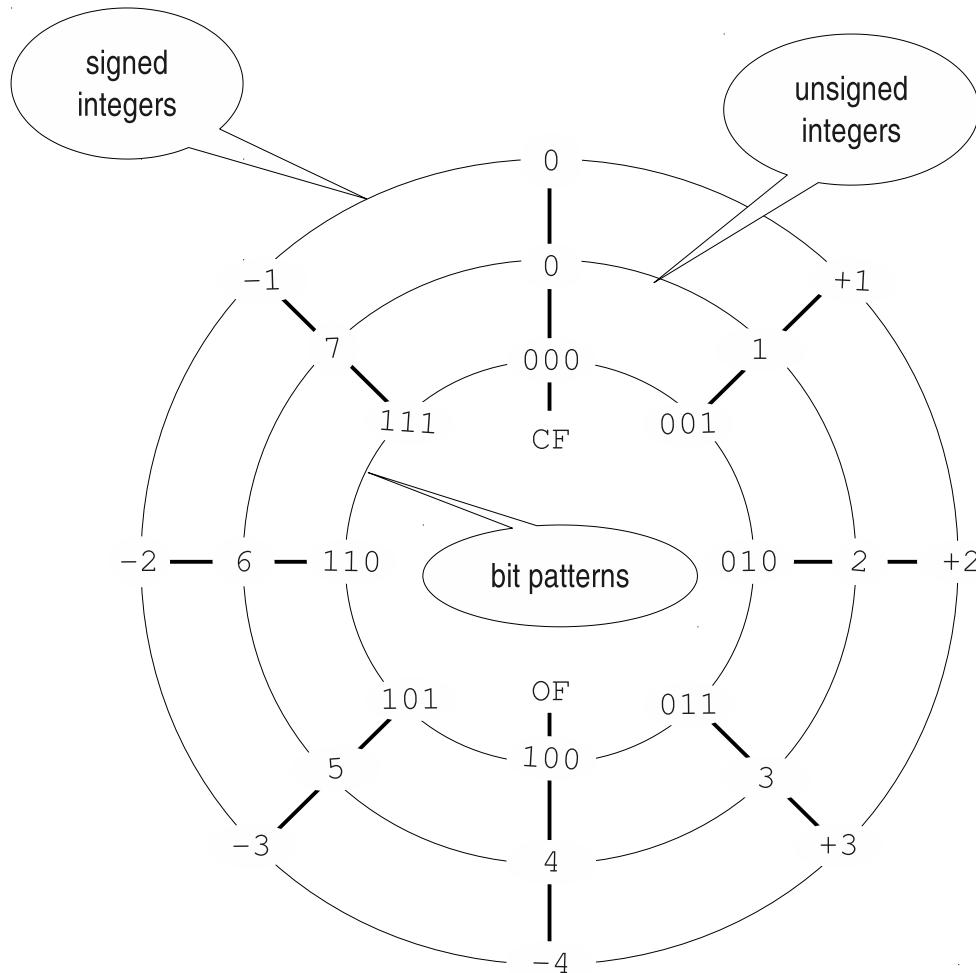


Figure 3.1: "Decoder Ring" for three-bit signed and unsigned integers. Move clockwise when adding numbers, counter-clockwise when subtracting. Crossing over 000 sets the CF to one, indicating an error for unsigned integers. Crossing over 100 sets the OF to one, indicating an error for signed integers.

**Example 3-h**

Using the “Decoder Ring” (Figure 3.1), add the unsigned integers  $3 + 4$ .

*Solution:*

Working only in the inner ring, start at the tic mark for 3, which corresponds to the bit pattern **011**. The bit pattern corresponding to 4 is **100**, which is four tic marks CW from zero. So move four tic marks CW from the 3 tic mark. This places us at the tic mark labeled **111**, which corresponds to 7. Since we did not pass the tic mark at the top of the Decoder Ring, **CF = 0**. Thus, the result is correct.

---

**Example 3-i**

Using the “Decoder Ring” (Figure 3.1), add the unsigned integers  $5 + 6$ .

*Solution:*

Working only in the inner ring, start at the tic mark for 5, which corresponds to the bit pattern **101**. The bit pattern corresponding to 6 is **110**, which is six tic marks CW from zero. So move six tic marks CW from the 5 tic mark. This places us at the tic mark labeled **011**, which corresponds to 3. Since we have crossed the tic mark at the top of the Decoder Ring, the CF becomes 1. Thus, the result is incorrect.

---

**Example 3-j**

Using the “Decoder Ring” (Figure 3.1), add the signed integers  $(+1) + (+2)$ .

*Solution:*

Working only in the outer ring, start at the tic mark for **+1**, which corresponds to the bit pattern **001**. The bit pattern corresponding to **+2** is **010**, which is two tic marks CW from zero. So move two tic marks CW from the **+1** tic mark. This places us at the tic mark labeled **011**, which corresponds to **+3**. Since we did not pass the tic mark at the bottom of the Decoder Ring, **OF = 0**. Thus, the result is correct.

---

**Example 3-k**

Using the “Decoder Ring” (Figure 3.1), add the signed integers  $(+3) + (-4)$ .

*Solution:*

Working only in the outer ring, start at the tic mark for **+3**, which corresponds to the bit pattern **011**. The bit pattern corresponding to **-4** is **100**, which is four tic marks CCW from zero. So move four tic marks CCW from the **+3** tic mark. This places us at the tic mark labeled **111**, which corresponds to **-1**. Since we did not pass the tic mark at the bottom of the Decoder Ring, **OF = 0**. Thus, the result is correct.

---



**Example 3-1**

Using the “Decoder Ring” (Figure 3.1), add the signed integers  $(+3) + (+1)$ .

*Solution:*

Working only in the outer ring, start at the tic mark for  $+3$ , which corresponds to the bit pattern **011**. The bit pattern corresponding to  $+1$  is **001**, which is one tic mark CW from zero. So move one tic mark CW from the  $+3$  tic mark. This places us at the tic mark labeled **100**, which corresponds to  $-4$ . Since we did pass the tic mark at the bottom of the Decoder Ring, **0F = 1**. Thus, the result is incorrect. □

## 3.5 C/C++ Basic Data Types

High-level languages provide some basic data types. For example, C/C++ provides **int**, **char**, **float**, etc. The sizes of some data types are shown in Table 3.3. The sizes given

Data type	32-bit mode	64-bit mode
char	8	8
int	32	32
long	32	64
long long	64	64
float	32	32
double	64	64
*any	32	64

Table 3.3: Sizes (in bits) of some C/C++ data types in 32-bit and 64-bit modes. The size of a **long** depends on the mode. Pointers (addresses) are 32 bits in 32-bit mode and can be 32 or 64 bits in 64-bit mode.

in this table are taken from the System V Application Binary Interface specifications, reference [33] for 32-bit and reference [25] for 64-bit, and are used by the **gcc** compiler for the x86-64 architecture. Language specifications tend to be more permissive in order to accommodate other hardware architectures. For example, see reference [10] for the specifications for C.

A given “real world” value can usually be represented in more than one data type. For example, most people would think of “123” as representing “one hundred twenty-three.” This value could be stored in a computer in **int** format or as a text string. An **int** in our C/C++ environment is stored in 32 bits, and the bit pattern would be

**0x0000007b**

As a C-style text string, it would also require four bytes of memory, but their bit patterns would be

**0x31 0x32 0x33 0x00**

The **int** format is easier to use in arithmetic and logical expressions, but the interface with the outside world through the screen and the keyboard uses the **char** format. If a user entered 123 from the keyboard, the operating system would read the individual characters, each in **char** format. The text string must be converted to **int** format. After the numbers are manipulated, the result must be converted from the **int** format to **char** format for display on the screen.

C programmers use functions in the `stdio` library and C++ programmers use functions in the `iostream` library to do these conversions between the `int` and `char` formats. For example, the C code sequence

```
scanf("%i", &x);
x += 100;
printf("%i", x);
```

or the C++ code sequence

```
cin >> x;
x += 100;
cout << x;
```

- reads characters from the keyboard and converts the character sequence into the corresponding `int` format.
- adds 100 to the `int`.
- converts the resulting `int` into a character sequence and displays it on the screen.

The C or C++ I/O library functions in the code segments above do the necessary conversions between character sequences and the `int` storage format. However, once the conversion is performed, they ultimately call the `read` *system call* function to read bytes from the keyboard and the `write` *system call* function to write bytes to the screen. As shown in Figure 3.2, an application program can call the `read` and `write` functions directly to transfer bytes.

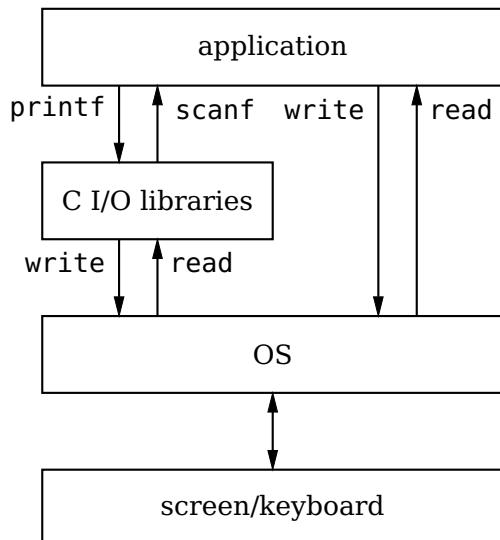


Figure 3.2: Relationship of I/O libraries to application and operating system. An application can use functions in the I/O libraries to convert between keyboard/screen chars and basic data types, or it can directly use the `read`/`write` system calls to transfer raw bytes.

When using the `read` and `write` system call functions for I/O, it is the programmer's responsibility to do the conversions between the `char` type used for I/O and the storage formats used within the program. We will soon be writing our own functions in assembly

language to convert between the character format used for screen display and keyboard input, and the internal storage format of integers in the binary number system. The purpose of writing our own functions is to gain a thorough understanding of how data is represented internally in the computer.

**Aside:** If the numerical data are used primarily for display, with few arithmetic operations, it makes more sense to store numerical data in character format. Indeed, this is done in many business data processing environments. But this makes arithmetic operation more complicated.

### 3.5.1 C/C++ Shift Operations

Since our primary goal here is to study storage formats, we will concentrate on bit patterns. We will develop a program in C that allows a user to enter bit patterns in hexadecimal. The program will read the characters from the keyboard in ASCII code and convert them into the corresponding `int` storage format as shown in Algorithm 3.8. This conversion algorithm involves manipulating data at the bit level.

---

**Algorithm 3.8:** Read hexadecimal value from keyboard.

---

- 1 `x = 0;`
- 2 Read character from keyboard;
- 3 **while** *more characters* **do**
- 4     `x = x shifted left four bit positions;`
- 5     `y = new character converted to an int;`
- 6     `x = x + y;`
- 7     Read character from keyboard;
- 8 Display the integer;

Let us examine this algorithm. Each character read from the keyboard represents a hexadecimal digit. That is, each character is one of '0', ..., '9', 'a', ..., 'f'. (We assume that the user does not make mistakes.) Since a hexadecimal digit represents four bits, we need to shift the accumulated integer four bits to the left in order to make room for the new four-bit value.

You should recognize that shifting an integer four bits to the left multiplies it by 16. As you will see in Sections 12.3 and 12.4 (pages 307 and 315), multiplication and division are complicated operations, and they can take a great deal of processor time. Using left/right shifts to effect multiplication/division by powers of two is very efficient. More importantly, the four-bit shift is more natural in this application.

The C/C++ operator for shifting bits to the left is `<<`.<sup>2</sup> For example, if `x` is an `int`, the statement

```
x = x << 4;
```

shifts the value in `x` four bits to the left, thus multiplying it by sixteen. Similarly, the C/C++ operator for shifting bits to the right is `>>`. For example, if `x` is an `int`, the statement

```
x = x >> 3;
```

shifts the value in `x` three bits to the right, thus dividing it by eight. Note that the three right-most bits are lost, so this is an integer `div` operation. The program in Listing 3.1 illustrates the use of the C shift operators to multiply and divide by powers of two.

---

<sup>2</sup>In C++ the `>>` and `<<` operators have been overloaded for use with the input and output streams.

---

```

1  /*
2   * mulDiv.c
3   * Asks user to enter an integer. Then prompts user to enter
4   * a power of two to multiply the integer, then another power
5   * of two to divide. Assumes that user does not request more
6   * than 30 as the power of 2.
7   * Bob Plantz - 4 June 2009
8   */
9
10 #include <stdio.h>
11
12 int main(void)
13 {
14     int x;
15     int leftShift, rightShift;
16
17     printf("Enter an integer: ");
18     scanf("%i", &x);
19
20     printf("Multiply by two raised to the power: ");
21     scanf("%i", &leftShift);
22     printf("%i x %i = %i\n", x, 1 << leftShift, x << leftShift);
23
24     printf("Divide by two raised to the power: ");
25     scanf("%i", &rightShift);
26     printf("%i / %i = %i\n", x, 1 << rightShift, x >> rightShift);
27
28     return 0;
29 }
```

---

Listing 3.1: Shifting to multiply and divide by powers of two.

### 3.5.2 C/C++ Bit Operations

We begin by reviewing the C/C++ bitwise logical operators,

and	&
or	
exclusive or	^
complement	~

It is easy to see what each of these operators does by using *truth tables*. To illustrate how truth tables work, consider the algorithm for binary addition. In Section 3.1 (page 30) we saw that the *i*th bit in the result is the sum of the *i*th bit of one number plus the *i*th bit of the other number plus the carry produced from adding the (*i*-1)*th* bits. This sum will produce a carry of zero or one. In other words, a bit adder has three inputs — the two corresponding bits from the two numbers being added and the carry from the previous bit addition — and two outputs — the result and the carry. In a truth table we have a column for each input and each output. Then we write down all possible input bit combinations and then show the output(s) in the corresponding row. A truth table for the bit addition operation is shown in Figure 3.3. We use the notation  $x[i]$  to represent the *i*th bit in the variable  $x$ ;  $x[i:j]$  would specify bits *i* through *j*.

$x[i]$	$y[i]$	$\text{carry}[(i-1)]$	$z[i]$	$\text{carry}[i]$
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Figure 3.3: Truth table for adding two bits with carry from a previous bit addition.  $x[i]$  is the  $i^{\text{th}}$  bit of  $x$ ;  $\text{carry}[(i-1)]$  is the carry from adding the  $(i-1)^{\text{th}}$  bits.

The bitwise logical operators act on the corresponding bits of two operands as shown in Figure 3.4.

and	$x[i]$	$y[i]$	$x[i] \& y[i]$
	0	0	0
	0	1	0
	1	0	0
	1	1	1
inclusive or	$x[i]$	$y[i]$	$x[i]   y[i]$
	0	0	0
	0	1	1
	1	0	1
	1	1	1
exclusive or	$x[i]$	$y[i]$	$x[i] ^ y[i]$
	0	0	0
	0	1	1
	1	0	1
	1	1	0
complement	$x[i]$	$\sim x[i]$	
	0	1	
	1	0	

Figure 3.4: Truth tables showing bitwise C/C++ operations.  $x[i]$  is the  $i^{\text{th}}$  bit in the variable  $x$ .

### Example 3-m

Let `int x = 0x1234abcd`. Compute the `and`, `or`, and `xor` with `0xdcba4321`.

*Solution:*

```
x & 0xdcba4321 = 0x10300301
x | 0xdcba4321 = 0xdebeebed
x ^ 0xdcba4321 = 0xce8ee8ec
```

Make sure that you distinguish these bitwise logical operators from the C/C++ logical operators, `&&`, `||`, and `!`. The logical operators work on groups of bits organized into integral data types rather than individual bits. For comparison, the truth tables for the C/C++ logical operators are shown in Figure 3.5

	x	y	x && y
and	0	0	0
	0	non-zero	0
	non-zero	0	0
	non-zero	non-zero	1

	x	y	x    y
or	0	0	0
	0	non-zero	1
	non-zero	0	1
	non-zero	non-zero	1

	x	!x
complement	0	1
	non-zero	0

Figure 3.5: Truth tables showing C/C++ logical operations. x and y are variables of integral data type.

### 3.5.3 C/C++ Data Type Conversions

Now we are prepared to see how we can convert from the ASCII character code to the `int` format. The `&` operator works very nicely for the conversion. If a numeric character is stored in the `char` variable `aChar`, from Table 3.4 we see that the required operation is

```
aChar = aChar & 0x0f;
```

Well, we still have an 8-bit value (with the four high-order bits zero), but we will work on this in a moment.

Next consider the alphabetic hexadecimal digits in Table 3.4. Notice that the low-order four bits are the same whether the character is upper case or lower case. We can use the same `&` operation to obtain these four bits, then add 9 to the result:

```
aChar = 0x09 + (aChar & 0x0f);
```

Conversion from the 8-bit `char` type to the 32-bit `int` type is accomplished by a type cast in C.

The resulting program is shown in Listing 3.2. Notice that we use the `printf` function to display the resulting stored value, both in hexadecimal and decimal. The conversion from stored `int` format to hexadecimal display is left as an exercise (Exercise 3-13).

Hex character	ASCII code	Corresponding int									
0	0011 0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
1	0011 0001	0000	0000	0000	0000	0000	0000	0000	0000	0000	0001
2	0011 0010	0000	0000	0000	0000	0000	0000	0000	0000	0000	0010
3	0011 0011	0000	0000	0000	0000	0000	0000	0000	0000	0000	0011
4	0011 0100	0000	0000	0000	0000	0000	0000	0000	0000	0000	0100
5	0011 0101	0000	0000	0000	0000	0000	0000	0000	0000	0000	0101
6	0011 0110	0000	0000	0000	0000	0000	0000	0000	0000	0000	0110
7	0011 0111	0000	0000	0000	0000	0000	0000	0000	0000	0000	0111
8	0011 1000	0000	0000	0000	0000	0000	0000	0000	0000	0000	1000
9	0011 1001	0000	0000	0000	0000	0000	0000	0000	0000	0000	1001
a	0110 0001	0000	0000	0000	0000	0000	0000	0000	0000	0000	1010
b	0110 0010	0000	0000	0000	0000	0000	0000	0000	0000	0000	1011
c	0110 0011	0000	0000	0000	0000	0000	0000	0000	0000	0000	1100
d	0110 0100	0000	0000	0000	0000	0000	0000	0000	0000	0000	1101
e	0110 0101	0000	0000	0000	0000	0000	0000	0000	0000	0000	1110
f	0110 0110	0000	0000	0000	0000	0000	0000	0000	0000	0000	1111

Table 3.4: Hexadecimal characters and corresponding int. Note the change in pattern from '9' to 'a'.

```

1  /*
2  * convertHex.c
3  * Asks user to enter a number in hexadecimal
4  * then echoes it in hexadecimal and in decimal.
5  * Assumes that user does not make mistakes.
6  * Bob Plantz - 4 June 2009
7  */
8
9 #include <stdio.h>
10 #include <unistd.h>
11
12 int main(void)
13 {
14     int x;
15     unsigned char aChar;
16
17     printf("Enter an integer in hexadecimal: ");
18     fflush(stdout);
19
20     x = 0;                                // initialize result
21     read(STDIN_FILENO, &aChar, 1);        // get first character
22     while (aChar != '\n')                  // look for return key
23     {
24         x = x << 4;                      // make room for next four bits
25         if (aChar <= '9')
26         {
27             x = x + (int)(aChar & 0x0f);
28         }
29         else
30         {

```

```

31     aChar = aChar & 0x0f;
32     aChar = aChar + 9;
33     x = x + (int)aChar;
34 }
35     read(STDIN_FILENO, &aChar, 1);
36 }
37
38 printf("You entered %#010x = %i (decimal)\n\n", x, x);
39
40 return 0;
41 }
```

Listing 3.2: Reading hexadecimal values from keyboard.

## 3.6 Other Codes

Thus far in this chapter we have used the binary number system to represent numerical values. It is an efficient code in the sense that each of the  $2^n$  bit patterns represents a value. On the other hand, there are some limitations in the code. We will explore some other codes in this section.

### 3.6.1 BCD Code

One limitation of using the binary number system is that a decimal number must be converted to binary before storing or performing arithmetic operations on it. And binary numbers must be converted to decimal for most real-world display purposes.

The Binary Coded Decimal (BCD) code is a code for individual decimal digits. Since there are ten decimal digits, the code must use four bits for each digit. The BCD code is shown in Table 3.5. For example, in a 16-bit storage location the decimal number 1234

Decimal digit	BCD code (four bits)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table 3.5: BCD code for the decimal digits.

would be stored in the BCD code as

0001 0010 0011 0100 # BCD

and in binary as

0000 0100 1101 0010 # binary

From Table 3.5 we can see that six bit patterns are “wasted.” The effect of this inefficiency is that a 16-bit storage location has a range of 0 – 9999 if we use BCD, but the range is 0 – 65535 if we use binary.

BCD is important in specialized systems that deal primarily with numerical data. There are I/O devices that deal directly with numbers in BCD without converting to/from a character code, for example, ASCII. The COBOL programming language supports a *packed BCD* format where two BCD characters are stored in each 8-bit byte. The last (4-bit) digit is used to store the sign of the number as shown in Table 3.6. The specific codes used depend upon the particular implementation.

<b>Sign</b>	<b>BCD code (four bits)</b>
+	1010
-	1011
+	1100
-	1101
+	1110
unsigned	1111

Table 3.6: Sign codes for packed BCD.

### 3.6.2 Gray Code

One of the problems with both the binary and BCD codes is that the difference between two adjacent values often requires that more than one bit be changed. For example, three bits must be changed when incrementing from 3 to 4 — 0011 to 0100. If the value is read during the time when the bits are being switched there may be an error. This is more apt to occur if the bits are implemented with, say, mechanical switches instead of electronic. The Gray code is one where there is only one bit that differs between any two adjacent values. As you will see in Section 4.3, this property also allows for a very useful visual tool for simplifying Boolean algebra expressions.

The Gray code is easily constructed. Start with one bit:

<b>decimal</b>	<b>Gray code</b>
0	0
1	1

To add a bit, first duplicate the existing pattern, but *reflected*:

<b>Gray code</b>
0
1
—
1
0

then add a zero to the beginning of each of the original bit patterns and a 1 to each of the reflected ones:

<b>decimal</b>	<b>Gray code</b>
0	00
1	01
—	—
2	11
3	10

Let us repeat these two steps to add another bit. Reflect the pattern:

<b>Gray code</b>
00
01
11
10
<hr/>
10
11
01
00

then add a zero to the beginning of each of the original bit patterns and a 1 to each of the reflected ones:

<b>decimal</b>	<b>Gray code</b>
0	000
1	001
2	011
3	010
<hr/>	<hr/>
4	110
5	111
6	101
7	100

The Gray code for four bits is shown in Table 3.7. Notice that the pattern of only changing one bit between adjacent values also holds when the bit pattern “wraps around.” That is, only one bit is changed when going from the highest value (15 for four bits) to the lowest (0).

<b>Decimal</b>	<b>Gray code</b>
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

Table 3.7: Gray code for 4 bits.

## 3.7 Exercises

- 3-1** (§3.1) How many bits are required to store a single decimal digit?
- 3-2** (§3.1) Using the answer from Exercise 1, invent a code for storing eight decimal digits in a thirty-two bit register. Using your new code, does binary addition produce the correct results?
- 3-3** (§3.3) Select several pairs of signed integers from Table 3.2, convert each to binary using the table, perform the binary addition, and check the results. Does this code always work?
- 3-4** (§3.3) If you did not select them in Exercise 3, add +4 and +5 using the four-bit, two's complement code (from Table 3.2). What answer do you get?
- 3-5** (§3.3) If you did not select them in Exercise 3, add -4 and -5 using the four-bit, two's complement code (from Table 3.2). What answer do you get?
- 3-6** (§3.3) Select any positive integer from Table 3.2. Add the binary representation for the positive value to the binary representation for the negative value. What is the four-bit result? What is the value of the CF? The OF? If you do the addition "on paper" (that is, you can use as many digits as you wish), how could you express, in English, the result of adding the positive representation of an integer to its negative representation in the two's complement notation? The negative representation to the positive representation? Which two integers do not have a representation of the opposite sign?
- 3-7** (§3.3) The following 8-bit hexadecimal values are stored in two's complement format. What are the equivalent signed decimal numbers?
- |       |       |
|-------|-------|
| a) 55 | e) 80 |
| b) aa | f) 63 |
| c) f0 | g) 7b |
| d) 0f |       |
- 3-8** (§3.3) The following 16-bit hexadecimal values are stored in two's complement format. What are the equivalent signed decimal numbers?
- |         |         |
|---------|---------|
| a) 1234 | e) 8000 |
| b) edcc | f) 0400 |
| c) fedc | g) ffff |
| d) 07d0 | h) 782f |
- 3-9** (§3.3) Show how each of the following signed, decimal integers would be stored in 8-bit two's complement format. Give your answer in hexadecimal.
- |        |         |
|--------|---------|
| a) 100 | e) 127  |
| b) -1  | f) -16  |
| c) -10 | g) -32  |
| d) 88  | h) -128 |

**3-10** (§3.3) Show how each of the following signed, decimal integers would be stored in 16-bit two's complement format. Give your answer in hexadecimal.

- |          |           |
|----------|-----------|
| a) 1024  | e) -256   |
| b) -1024 | f) -32768 |
| c) -1    | g) -32767 |
| d) 32767 | h) -128   |

**3-11** (§3.4) Perform binary addition of the following pairs of 8-bit numbers (shown in hexadecimal) and indicate whether your result is “right” or “wrong.” First treat them as unsigned values, then as signed values (stored in two’s complement format). Thus, you will have two “right/wrong” answers for each sum. Note that the computer performs only one addition, setting both the CF and OF according to the results of the addition. It is up to the program to test the appropriate flag depending on whether the numbers are being considered as unsigned or signed in the program.

- |            |            |
|------------|------------|
| a) 55 + aa | d) 63 + 7b |
| b) 55 + f0 | e) 0f + ff |
| c) 80 + 7b | f) 80 + 80 |

**3-12** (§3.4, 3.5) Perform binary addition of the following pairs of 16-bit numbers (shown in hexadecimal) and indicate whether your result is “right” or “wrong.” First treat them as unsigned values, then as signed values (stored in two’s complement format). Thus, you will have two “right/wrong” answers for each sum. Note that the computer performs only one addition, setting both the CF and OF according to the results of the addition. It is up to the program to test the appropriate flag depending on whether the numbers are being considered as unsigned or signed in the program.

- |                |                |
|----------------|----------------|
| a) 1234 + edcc | d) 0400 + ffff |
| b) 1234 + fedc | e) 07d0 + 782f |
| c) 8000 + 8000 | f) 8000 + ffff |

**3-13** (§3.5) Enter the program in Figure 3.1 and get it to work. Use the program to compute 1 (one) multiplied by 2 raised to the 31st power. What result do you get for 1 (one) multiplied by 2 raised to the 32nd power? Explain the results.

**3-14** (§3.5) Write a C program that prompts the user to enter a hexadecimal value, multiplies it by ten, then displays the result in hexadecimal. Your main function should

- a) declare a char array,
- b) call the readLn function to read from the keyboard,
- c) call a function to convert the input text string to an int,
- d) multiply the int by ten,
- e) call a function to convert the int to its corresponding hexadecimal text string,
- f) call writeStr to display the resulting hexadecimal text string.

Use the `readLn` and `writeStr` functions from Exercise 2 -32 to read from the keyboard and display on the screen. Place the functions to perform the conversions in separate files. Hint: review Figure 3.2.

**3-15** (§3.5) Write a C program that prompts the user to enter a binary value, multiplies it by ten, then displays the result in binary. (“Binary” here means that the user communicates with the program in ones and zeros.) Your `main` function should

- a) declare a `char` array,
- b) call the `readLn` function to read from the keyboard,
- c) call a function to convert the input text string to an `int`,
- d) multiply the `int` by ten,
- e) call a function to convert the `int` to its corresponding binary text string,
- f) call `writeStr` to display the resulting binary text string.

Use the `readLn` and `writeStr` functions from Exercise 2 -32 to read from the keyboard and display on the screen. Your functions to convert from a binary text string to an `int` and back should be placed in separate functions.

**3-16** (§3.5) Write a C program that prompts the user to enter unsigned decimal integer, multiplies it by ten, then displays the result in binary. (“Binary” here means that the user communicates with the program in ones and zeros.) Your `main` function should

- a) declare a `char` array,
- b) call the `readLn` function to read from the keyboard,
- c) call a function to convert the input text string to an `int`,
- d) multiply the `int` by ten,
- e) call a function to convert the `int` to its corresponding decimal text string,
- f) call `writeStr` to display the resulting decimal text string.

Use the `readLn` and `writeStr` functions from Exercise 2 -32 to read from the keyboard and display on the screen. Your function to convert from a decimal text string to an `int` should be placed in a separate function. Hint: this problem cannot be solved by simply shifting bit patterns. Think carefully about the mathematical equivalence of shifting bit patterns left or right.

**3-17** (§3.5) Modify the program in Exercise 3-16 so that it works with signed decimal integers.

# Chapter 4

# Logic Gates

This chapter provides an overview of the hardware components that are used to build a computer. We will limit the discussion to electronic computers, which use transistors to switch between two different voltages. One voltage represents 0, the other 1. The hardware devices that implement the logical operations are called *logic gates*.

## 4.1 Boolean Algebra

In order to understand how the components are combined to build a computer, you need to learn another algebra system — *Boolean algebra*. There are many approaches to learning about Boolean algebra. Some authors start with the postulates of Boolean algebra and develop the mathematical tools needed for working with switching circuits from them. We will take the more pragmatic approach of starting with the basic properties of Boolean algebra, then explore the properties of the algebra. For a more theoretical approach, including discussions of more general Boolean algebra concepts, search the internet, or take a look at books like [9], [20], [23], or [24].

There are only two values, 0 and 1, unlike elementary algebra that deals with an infinity of values, the real numbers. Since there are only two values, a *truth table* is a very useful tool for working with Boolean algebra. A truth table lists all possible combinations of the variables in the problem. The resulting value of the Boolean operation(s) for each variable combination is shown on the respective row.

Elementary algebra has four operations, addition, subtraction, multiplication, and division, but Boolean algebra has only three operations:

- AND — a binary operator; the result is 1 if and only if both operands are 1; otherwise the result is 0. We will use ‘‘ $\cdot$ ’’ to designate the AND operation. It is also common to use the ‘‘ $\wedge$ ’ symbol or simply “AND”. The hardware symbol for the AND gate is shown in Figure 4.1. The inputs are  $x$  and  $y$ . The resulting output,  $x \cdot y$ , is shown in the truth table in this figure.

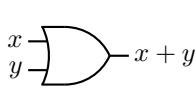


$x$	$y$	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

Figure 4.1: The AND gate acting on two variables,  $x$  and  $y$ .

We can see from the truth table that the AND operator follows similar rules as multiplication in elementary algebra.

- OR — a binary operator; the result is 1 if at least one of the two operands is 1; otherwise the result is 0. We will use ' $+$ ' to designate the OR operation. It is also common to use the ' $\vee$ ' symbol or simply "OR". The hardware symbol for the OR gate is shown in Figure 4.2. The inputs are  $x$  and  $y$ . The resulting output,  $x + y$ , is shown in the truth table in this figure. From the truth table we can see that the OR



$x$	$y$	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

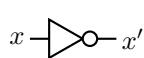
Figure 4.2: The OR gate acting on two variables,  $x$  and  $y$ .

operator follows the same rules as addition in elementary algebra except that

$$1 + 1 = 1$$

in Boolean algebra. Unlike elementary algebra, there is no carry from the OR operation. Since addition of integers can produce a carry, you will see in Section 5.1 that implementing addition requires more than a simple OR gate.

- NOT — a unary operator; the result is 1 if the operand is 0, or 0 if the operand is 1. Other names for the NOT operation are *complement* and *invert*. We will use  $x'$  to designate the NOT operation. It is also common to use  $\neg x$ , or  $\bar{x}$ . The hardware symbol for the NOT gate is shown in Figure 4.3. The input is  $x$ . The resulting output,  $x'$ , is shown in the truth table in this figure.



$x$	$x'$
0	1
1	0

Figure 4.3: The NOT gate acting on one variable,  $x$ .

The NOT operation has no analog in elementary algebra. Be careful to notice that inversion of a value in elementary algebra is a division operation, which does not exist in Boolean algebra.

Two-state variables can be combined into expressions with these three operators in the same way that you would use the C/C++ operators `&&`, `||`, and `!` to create logical expressions commonly used to control `if` and `while` statements. We now examine some Boolean algebra properties for manipulating such expressions. As you read through this material, keep in mind that the same techniques can be applied to logical expressions in programming languages.

These properties are commonly presented as theorems. They are easily proved from application of truth tables.

There is a duality between the AND and OR operators. In any equality you can interchange AND and OR along with the constants 0 and 1, and the equality still holds. Thus the properties will be presented in pairs that illustrate their duality. We first consider properties that are the *same* as in elementary algebra.

- AND and OR are *associative*:

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z \quad (4.1)$$

$$x + (y + z) = (x + y) + z \quad (4.2)$$

It is straightforward to prove these equations with truth tables. For example, for Equation 4.1:

$x$	$y$	$z$	$(y \cdot z)$	$(x \cdot y)$	$x \cdot (y \cdot z)$	$= (x \cdot y) \cdot z$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	1	0	0
1	1	1	1	1	1	1

And for Equation 4.2:

$x$	$y$	$z$	$(y + z)$	$(x + y)$	$x + (y + z)$	$= (x + y) + z$
0	0	0	0	0	0	0
0	0	1	1	0	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	0	1	1	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

- AND and OR have an *identity* value:

$$x \cdot 1 = x \quad (4.3)$$

$$x + 0 = x \quad (4.4)$$

Now we consider properties where Boolean algebra *differs* from elementary algebra.

- AND and OR are *commutative*:

$$x \cdot y = y \cdot x \quad (4.5)$$

$$x + y = y + x \quad (4.6)$$

This is easily proved by looking at the second and third lines of the respective truth tables. In elementary algebra, only the addition and multiplication operators are commutative.

- AND and OR have a *null* value:

$$x \cdot 0 = 0 \quad (4.7)$$

$$x + 1 = 1 \quad (4.8)$$

The null value for the AND is the same as multiplication in elementary algebra. But addition in elementary algebra does not have a null constant, while OR in Boolean algebra does.

- AND and OR have a *complement* value:

$$x \cdot x' = 0 \quad (4.9)$$

$$x + x' = 1 \quad (4.10)$$

Complement does not exist in elementary algebra.

- AND and OR are *idempotent*:

$$x \cdot x = x \quad (4.11)$$

$$x + x = x \quad (4.12)$$

That is, repeated application of either operator to the same value does not change it. This differs considerably from elementary algebra — repeated application of addition is equivalent to multiplication and repeated application of multiplication is the power operation.

- AND and OR are *distributive*:

$$x \cdot (y + z) = x \cdot y + x \cdot z \quad (4.13)$$

$$x + y \cdot z = (x + y) \cdot (x + z) \quad (4.14)$$

Going from right to left in Equation 4.13 is the very familiar *factoring* from addition and multiplication in elementary algebra. On the other hand, the operation in Equation 4.14 has no analog in elementary algebra. It follows from the idempotency property. The NOT operator has an obvious property:

- NOT shows *involution*:

$$(x')' = x \quad (4.15)$$

Again, since there is no complement in elementary algebra, there is no equivalent property.

- DeMorgan's Law is an important expression of the duality between the AND and OR operations.

$$(x \cdot y)' = x' + y' \quad (4.16)$$

$$(x + y)' = x' \cdot y' \quad (4.17)$$

The validity of DeMorgan's Law can be seen in the following truth tables. For Equation 4.16:

$x$	$y$	$(x \cdot y)$	$(x \cdot y)'$	$x'$	$y'$	$x' + y'$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

And for Equation 4.17:

$x$	$y$	$(x + y)$	$(x + y)'$	$x'$	$y'$	$x' \cdot y'$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

## 4.2 Canonical (Standard) Forms

Some terminology and definitions at this point will help our discussion. Consider two dictionary definitions of *literal*[26]:

- literal* 1b: adhering to fact or to the ordinary construction or primary meaning of a term or expression : ACTUAL.
- 2: of, relating to, or expressed in letters.

In programming we use the first definition of literal. For example, in the following code sequence

```
int xyz = 123;
char a = 'b';
char *greeting = "Hello";
```

the number “123”, the character ‘b’, and the string “Hello” are all literals. They are interpreted by the compiler exactly as written. On the other hand, “xyz”, “a”, and “greeting” are all names of variables.

In mathematics we use the second definition of literal. That is, in the algebraic expression

$$3x + 12y - z$$

the letters  $x$ ,  $y$ , and  $z$  are called literals. Furthermore, it is common to omit the “.” operator to designate multiplication. Similarly, it is often dropped in Boolean algebra expressions when the AND operation is implied.

The meaning of *literal* in Boolean algebra is slightly more specific.

**literal** A presence of a variable or its complement in an expression. For example, the expression

$$x \cdot y + x' \cdot z + x' \cdot y' \cdot z'$$

contains seven literals.

From the context of the discussion you should be able to tell which meaning of “literal” is intended and when the “.” operator is omitted.

A Boolean expression is created from the numbers 0 and 1, and literals. Literals can be combined using either the “.” or the “+” operators, which are multiplicative and additive operations, respectively. We will use the following terminology.

**product term:** A term in which the literals are connected with the AND operator. AND is multiplicative, hence the use of “product.”

**minterm or standard product:** A product term that contains each of the variables in the problem, either in its complemented or uncomplemented form. For example, if a problem involves three variables (say,  $x$ ,  $y$ , and  $z$ ),  $x \cdot y \cdot z$ ,  $x' \cdot y \cdot z'$ , and  $x' \cdot y' \cdot z'$  are all minterms, but  $x \cdot y$  is not.

**sum of products (SoP):** One or more product terms connected with OR operators. OR is additive, hence the use of “sum.”

**sum of minterms (SoM) or canonical sum:** An SoP in which each product term is a minterm. Since all the variables are present in each minterm, the canonical sum is unique for a given problem.

When first defining a problem, starting with the SoM ensures that the full effect of each variable has been taken into account. This often does not lead to the best implementation. In Section 4.3 we will see some tools to simplify the expression, and hence, the implementation.

It is common to index the minterms according to the values of the variables that would cause that minterm to evaluate to 1. For example,  $x' \cdot y' \cdot z' = 1$  when  $x = 0$ ,  $y = 0$ , and  $z = 0$ , so this would be  $m_0$ . The minterm  $x' \cdot y \cdot z'$  evaluates to 1 when  $x = 0$ ,  $y = 1$ , and  $z = 0$ , so is  $m_2$ . Table 4.1 lists all the minterms for a three-variable expression.

minterm	$x$	$y$	$z$
$m_0 = x' \cdot y' \cdot z'$	0	0	0
$m_1 = x' \cdot y' \cdot z$	0	0	1
$m_2 = x' \cdot y \cdot z'$	0	1	0
$m_3 = x' \cdot y \cdot z$	0	1	1
$m_4 = x \cdot y' \cdot z'$	1	0	0
$m_5 = x \cdot y' \cdot z$	1	0	1
$m_6 = x \cdot y \cdot z'$	1	1	0
$m_7 = x \cdot y \cdot z$	1	1	1

Table 4.1: Minterms for three variables.  $m_i$  is the  $i$ th minterm. The  $x$ ,  $y$ , and  $z$  values cause the corresponding minterm to evaluate to 1.

A convenient notation for expressing a sum of minterms is to use the  $\sum$  symbol with a numerical list of the minterm indexes. For example,

$$\begin{aligned} F(x, y, z) &= x' \cdot y' \cdot z' + x' \cdot y' \cdot z + x \cdot y' \cdot z + x \cdot y \cdot z' \\ &= m_0 + m_1 + m_5 + m_6 \\ &= \sum(0, 1, 5, 6) \end{aligned} \quad (4.18)$$

As you might expect, each of the terms defined above has a dual definition.

**sum term:** A term in which the literals are connected with the OR operator. OR is additive, hence the use of “sum.”

**maxterm or standard sum:** A sum term that contains each of the variables in the problem, either in its complemented or uncomplemented form. For example, if an expression involves three variables,  $x$ ,  $y$ , and  $z$ ,  $(x + y + z)$ ,  $(x' + y + z')$ , and  $(x' + y' + z')$  are all maxterms, but  $(x + y)$  is not.

**product of sums (PoS):** One or more sum terms connected with AND operators. AND is multiplicative, hence the use of “product.”

**product of maxterms (PoM) or canonical product:** A PoS in which each sum term is a maxterm. Since all the variables are present in each maxterm, the canonical product is unique for a given problem.

It also follows that any Boolean function can be uniquely expressed as a product of maxterms (PoM) that evaluate to 1. Starting with the product of maxterms ensures that the full effect of each variable has been taken into account. Again, this often does not lead to the best implementation, and in Section 4.3 we will see some tools to simplify PoMs.

It is common to index the maxterms according to the values of the variables that would cause that maxterm to evaluate to 0. For example,  $x + y + z = 0$  when  $x = 0$ ,  $y = 0$ , and  $z = 0$ , so this would be  $M_0$ . The maxterm  $x' + y + z'$  evaluates to 0 when  $x = 1$ ,  $y = 0$ , and  $z = 1$ , so is  $m_5$ . Table 4.2 lists all the maxterms for a three-variable expression.

Maxterm	x	y	z
$M_0 = x + y + z$	0	0	0
$M_1 = x + y + z'$	0	0	1
$M_2 = x + y' + z$	0	1	0
$M_3 = x + y' + z'$	0	1	1
$M_4 = x' + y + z$	1	0	0
$M_5 = x' + y + z'$	1	0	1
$M_6 = x' + y' + z$	1	1	0
$M_7 = x' + y' + z'$	1	1	1

Table 4.2: Maxterms for three variables.  $M_i$  is the  $i$ th maxterm. The  $x$ ,  $y$ , and  $z$  values cause the corresponding maxterm to evaluate to 0.

The similar notation for expressing a product of maxterms is to use the  $\prod$  symbol with a numerical list of the maxterm indexes. For example (and see Exercise 4-8),

$$\begin{aligned} F(x, y, z) &= (x + y' + z) \cdot (x + y' + z') \cdot (x' + y + z) \cdot (x' + y' + z') \\ &= M_2 \cdot M_3 \cdot M_4 \cdot M_7 \\ &= \prod(2, 3, 4, 7) \end{aligned} \quad (4.19)$$

The names “minterm” and “maxterm” may seem somewhat arbitrary. But consider the two functions,

$$F_1(x, y, z) = x \cdot y \cdot z \quad (4.20)$$

$$F_2(x, y, z) = x + y + z \quad (4.21)$$

There are eight ( $2^3$ ) permutations of the three variables,  $x$ ,  $y$ , and  $z$ .  $F_1$  has one minterm and evaluates to 1 for only one of the permutations,  $x = y = z = 1$ .  $F_2$  has one maxterm and evaluates to 1 for all permutations *except* when  $x = y = z = 0$ . This is shown in the following truth table:

$x$	$y$	$z$	<i>minterm</i>	<i>maxterm</i>
			$F_1 = (x \cdot y \cdot z)$	$F_2 = (x + y + z)$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

ORing more minterms to an SoP expression *expands* the number of cases where it evaluates to 1, and ANDing more maxterms to a PoS expression *reduces* the number of cases where it evaluates to 1.

## 4.3 Boolean Function Minimization

In this section we explore some important tools for manipulating Boolean expressions in order to simplify their hardware implementation. When implementing a Boolean function in hardware, each “.” operator represents an AND gate and each “+” operator

an OR gate. In general, the complexity of the hardware is related to the number of AND and OR gates. NOT gates are simple and tend not to contribute significantly to the complexity.

We begin with some definitions.

**minimal sum of products (mSoP):** A sum of products expression is minimal if all other mathematically equivalent SoPs

1. have at least as many product terms, and
2. those with the same number of product terms have at least as many literals.

**minimal product of sums (mPoS):** A product of sums expression is minimal if all other mathematically equivalent PoSs

1. have at least as many sum factors, and
2. those with the same number of sum factors have at least as many literals.

These definitions imply that there can be more than one minimal solution to a problem. Good hardware design practice involves finding all the minimal solutions, then assessing each one within the context of the available hardware. For example, judiciously placed NOT gates can actually reduce hardware complexity (Section 4.4.3, page 83).

### 4.3.1 Minimization Using Algebraic Manipulations

To illustrate the importance of reducing the complexity of a Boolean function, consider the following function:

$$F_1(x, y) = x \cdot y' + x' \cdot y + x \cdot y \quad (4.22)$$

The expression on the right-hand side is an SoM. The circuit to implement this function is shown in Figure 4.4. It requires three AND gates, one OR gate, and two NOT gates.

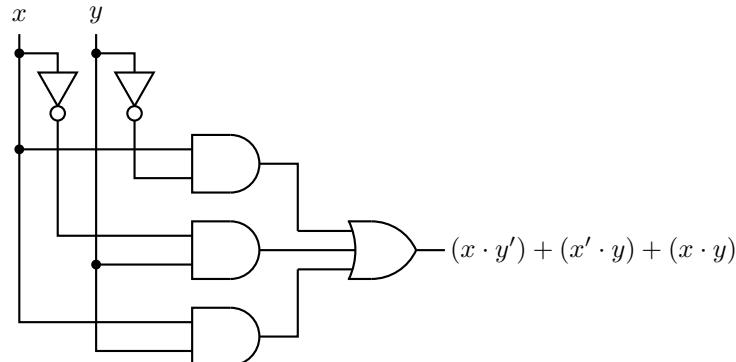


Figure 4.4: Hardware implementation of the function in Equation 4.22.

Now let us simplify the expression in Equation 4.22 to see if we can reduce the hardware requirements. This process will probably seem odd to a person who is not used to manipulating Boolean expressions, because there is not a single correct path to a solution. We present one way here. First we use the idempotency property (Equation 4.12) to duplicate the second term:

$$F_1(x, y) = x \cdot y' + x \cdot y + x' \cdot y + x \cdot y \quad (4.23)$$

Next we use the distributive property (Equation 4.13) to factor the expression:

$$F_1(x, y) = x \cdot (y' + y) + y \cdot (x' + x) \quad (4.24)$$

And from the complement property (Equation 4.10) we get:

$$\begin{aligned} F_1(x, y) &= x \cdot 1 + y \cdot 1 \\ &= x + y \end{aligned} \quad (4.25)$$

which you recognize as the simple OR operation. It is easy to see that this is a minimal sum of products for this function. We can implement Equation 4.22 with a single OR gate — see Figure 4.2 on page 62. This is clearly a less expensive, faster circuit than the one shown in Figure 4.4.

To illustrate how a product of sums expression can be minimized, consider the function:

$$F_2(x, y) = (x + y') \cdot (x' + y) \cdot (x' + y') \quad (4.26)$$

The expression on the right-hand side is a PoM. The circuit for this function is shown in Figure 4.5. It requires three OR gates, one AND gate, and two NOT gates.

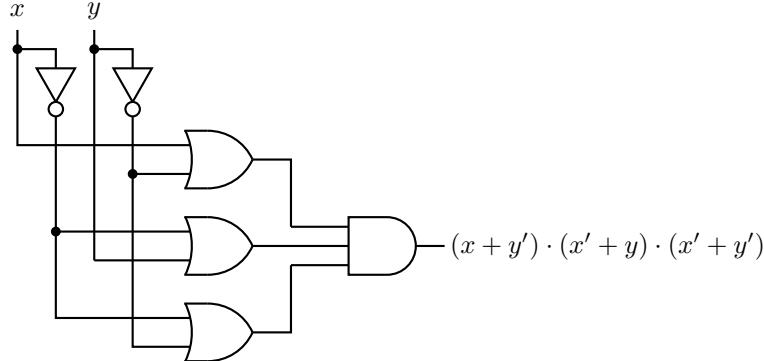


Figure 4.5: Hardware implementation of the function in Equation 4.28.

We will use the distributive property (Equation 4.14) on the right two factors and recognize the complement (Equation 4.9):

$$\begin{aligned} F_2(x, y, z) &= (x + y') \cdot (x' + y \cdot y') \\ &= (x + y') \cdot x' \end{aligned} \quad (4.27)$$

Now, use the distributive (Equation 4.13) and complement (Equation 4.9) properties to obtain:

$$\begin{aligned} F_2(x, y, z) &= x \cdot x' + x' \cdot y' \\ &= x' \cdot y' \end{aligned} \quad (4.28)$$

Thus, the function can be implemented with two NOT gates and a single AND gate, which is clearly a minimal product of sums. Again, with a little algebraic manipulation we have arrived at a much simpler solution.

**Example 4-a**

Design a function that will detect the even 4-bit integers.

*Solution:*

The even 4-bit integers are given by the function:

$$\begin{aligned} F(w, x, y, z) &= w' \cdot x' \cdot y' \cdot z' + w' \cdot x' \cdot y \cdot z' + w' \cdot x \cdot y' \cdot z' + w' \cdot x \cdot y \cdot z' \\ &\quad + w \cdot x' \cdot y' \cdot z' + w \cdot x' \cdot y \cdot z' + w \cdot x \cdot y' \cdot z' + w \cdot x \cdot y \cdot z' \end{aligned}$$

Using the distributive property repeatedly we get:

$$\begin{aligned} F(w, x, y, z) &= z' \cdot (w' \cdot x' \cdot y' + w' \cdot x' \cdot y + w' \cdot x \cdot y' + w' \cdot x \cdot y) \\ &\quad + w \cdot x' \cdot y' + w \cdot x' \cdot y + w \cdot x \cdot y' + w \cdot x \cdot y \\ &= z' \cdot (w' \cdot (x' \cdot y' + x' \cdot y + x \cdot y' + x \cdot y) + w \cdot (x' \cdot y' + x' \cdot y + x \cdot y' + x \cdot y)) \\ &= z' \cdot (w' + w) \cdot (x' \cdot y' + x' \cdot y + x \cdot y' + x \cdot y) \\ &= z' \cdot (w' + w) \cdot (x' \cdot (y' + y) + x \cdot (y' + y)) \\ &= z' \cdot (w' + w) \cdot (x' + x) \cdot (y' + y) \end{aligned}$$

And from the complement property we arrive at a minimal sum of products:

$$F(x, y, z) = z'$$

□

### 4.3.2 Minimization Using Graphic Tools

The Karnaugh map was invented in 1953 by Maurice Karnaugh while working as a telecommunications engineer at Bell Labs. Also known as a K-map, it provides a graphic view of all the possible minterms for a given number of variables. The format is a rectangular grid with a cell for each minterm. There are  $2^n$  cells for  $n$  variables.

Figure 4.6 shows how all four minterms for two variables are mapped onto a four-cell Karnaugh map. The vertical axis is used for plotting  $x$  and the horizontal for  $y$ . The

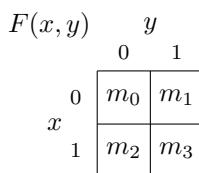


Figure 4.6: Mapping of two-variable minterms on a Karnaugh map.

value of  $x$  for each row is shown by the number (0 or 1) immediately to the left of the row, and the value of  $y$  for each column appears at the top of the column. Although it occurs automatically in a two-variable Karnaugh map, the cells must be arranged such that only one variable changes between two cells that share an edge. This is called the *adjacency property*.

The procedure for simplifying an SoP expression using a Karnaugh map is:

1. Place a 1 in each cell that corresponds to a minterm that evaluates to 1 in the expression.

2. Combine cells with 1s in them and that share edges into the largest possible groups. Larger groups result in simpler expressions. The number of cells in a group must be a power of 2. The edges of the Karnaugh map are considered to wrap around to the other side, both vertically and horizontally.
3. Groups may overlap. In fact, this is common. However, no group should be fully enclosed by another group.
4. The result is the sum of the product terms that represent each group.

The Karnaugh map provides a graphical means to find the same simplifications as algebraic manipulations, but some people find it easier to spot simplification patterns on a Karnaugh map. Probably the easiest way to understand how Karnaugh maps are used is through an example. We start with Equation 4.22 (repeated here):

$$F_1(x, y) = x \cdot y' + x' \cdot y + x \cdot y$$

and use a Karnaugh map to graphically find the same minimal sum of products that the algebraic steps in Equations 4.23 through 4.25 gave us.

We start by placing a 1 in each cell corresponding to a minterm that appears in the equation as shown in Figure 4.7. The two cells on the right side correspond to the

		$F_1(x, y)$	
		$y$	
$x$	0		1
	1	1	1

Figure 4.7: Karnaugh map for  $F_1(x, y) = x \cdot y' + x' \cdot y + x \cdot y$ .

minterms  $m_1$  and  $m_3$  and represent  $x' \cdot y + x \cdot y$ . Using the distributive (Equation 4.13) and complement (Equation 4.10) properties, we can see that

$$\begin{aligned} x' \cdot y + x \cdot y &= (x + x') \cdot y \\ &= y \end{aligned} \tag{4.29}$$

The simplification in Equation 4.29 is shown graphically by the grouping in Figure 4.8.

		$F_1(x, y)$	
		$y$	
$x$	0		1
	1	1	(1)

Figure 4.8: A Karnaugh map grouping showing that  $F_1(x, y) = x \cdot y' + x' \cdot y + x \cdot y = x \cdot y' + y$ .

Since groups can overlap, we create a second grouping as shown in Figure 4.9. This grouping shows the simplification,

$$\begin{aligned} x \cdot y' + x \cdot y &= x \cdot (y' + y) \\ &= x \end{aligned} \tag{4.30}$$

The group in the bottom row represents the product term  $x$ , and the one in the right-hand column represents  $y$ . So the simplification is:

$$F_1(x, y) = x + y \tag{4.31}$$

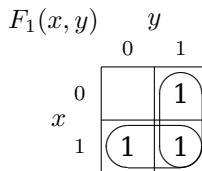


Figure 4.9: Two-variable Karnaugh map showing the groupings  $x$  and  $y$ .

Note that the overlapping cell,  $x \cdot y$ , is the term that we used the idempotent property to duplicate in Equation 4.23. Grouping overlapping cells is a graphical application of the idempotent property (Equation 4.12).

Next we consider a three-variable Karnaugh map. Table 4.1 (page 66) lists all the minterms for three variables,  $x$ ,  $y$ , and  $z$ , numbered from 0 – 7. A total of eight cells are needed, so we will draw it four cells wide and two high. Our Karnaugh map will be drawn with  $y$  and  $z$  on the horizontal axis, and  $x$  on the vertical. Figure 4.10 shows how the three-variable minterms map onto a Karnaugh map.

		$F(x, y, z)$			
		$yz$			
		00	01	11	10
$x$	0	$m_0$	$m_1$	$m_3$	$m_2$
	1	$m_4$	$m_5$	$m_7$	$m_6$

Figure 4.10: Mapping of three-variable minterms on a Karnaugh map.

Notice the order of the bit patterns along the top of the three-variable Karnaugh map, which is chosen such that only one variable changes value between any two adjacent cells (the adjacency property). It is the same as a two-variable Gray code (see Table 3.7, page 57). That is, the order of the columns is such that the  $yz$  values follow the Gray code.

A four-variable Karnaugh map is shown in Figure 4.11. The  $y$  and  $z$  variables are on the horizontal axis,  $w$  and  $x$  on the vertical. From this four-variable Karnaugh map we see that the order of the rows is such that the  $wx$  values also follow the Gray code, again to implement the adjacency property.

		$F(w, x, y, z)$			
		$yz$			
		00	01	11	10
$wx$	00	$m_0$	$m_1$	$m_3$	$m_2$
	01	$m_4$	$m_5$	$m_7$	$m_6$
	11	$m_{12}$	$m_{13}$	$m_{15}$	$m_{14}$
	10	$m_8$	$m_9$	$m_{11}$	$m_{10}$

Figure 4.11: Mapping of four-variable minterms on a Karnaugh map.

Other axis labeling schemes also work. The only requirement is that entries in adjacent cells differ by only one bit (which is a property of the Gray code). See Exercises 4-9 and 4-10.

**Example 4-b**

Find a minimal sum of products expression for the function

$$\begin{aligned} F(x, y, z) = & x' \cdot y' \cdot z' + x' \cdot y' \cdot z + x' \cdot y \cdot z' \\ & + x \cdot y' \cdot z' + x \cdot y \cdot z' + x \cdot y \cdot z \end{aligned}$$

*Solution:*

First we draw the Karnaugh map:

		yz				
		00	01	11	10	
x		0	1	1		1
		1	1		1	1

Several groupings are possible. Keep in mind that groupings can wrap around. We will work with

		yz				
		00	01	11	10	
x		0	(1)	(1)		(1)
		1	1		(1)	(1)

which yields a minimal sum of products:

$$F(x, y, z) = z' + x' \cdot y' + x \cdot y$$

□

We may wish to implement a function as a product of sums instead of a sum of products. From DeMorgan's Law, we know that the complement of an expression exchanges all ANDs and ORs, and complements each of the literals. The zeros in a Karnaugh map represent the complement of the expression. So if we

1. place a 0 in each cell of the Karnaugh map corresponding to a *missing* minterm in the expression,
2. find groupings of the cells with 0s in them,
3. write a sum of products expression represented by the grouping of 0s, and
4. complement this expression,

we will have the desired expression expressed as a product of sums. Let us use the previous example to illustrate.

**Example 4-c**

Find a minimal product of sums for the function (repeat of Example 4-b).

$$\begin{aligned} F(x, y, z) = & x' \cdot y' \cdot z' + x' \cdot y' \cdot z + x' \cdot y \cdot z' \\ & + x \cdot y' \cdot z' + x \cdot y \cdot z' + x \cdot y \cdot z \end{aligned}$$

*Solution:*

Using the Karnaugh map zeros,

		yz			
		00	01	11	10
		0		0	
x	0				
	1		0		

we obtain the complement of our desired function,

$$F'(x, y, z) = x' \cdot y \cdot z + x \cdot y' \cdot z$$

and from DeMorgan's Law:

$$F(x, y, z) = (x + y' + z') \cdot (x' + y + z')$$

□

We now work an example with four variables.

**Example 4-d**

Find a minimal sum of products expression for the function

$$\begin{aligned} F(x, y, z) = & w' \cdot x' \cdot y' \cdot z' + w' \cdot x' \cdot y \cdot z' + w' \cdot x \cdot y' \cdot z \\ & + w' \cdot x \cdot y \cdot z + w \cdot x \cdot y' \cdot z + w \cdot x \cdot y \cdot z \\ & + w \cdot x' \cdot y' \cdot z' + w \cdot x' \cdot y \cdot z' \end{aligned}$$

*Solution:*

Using the groupings on the Karnaugh map,

		yz				
		00	01	11	10	
		00	1			1
wx	01		1	1		
	11		1	1		
10	00	1				1
	01					

we obtain a minimal sum of products,

$$F(w, x, y, z) = x' \cdot z' + x \cdot z$$

Not only have we greatly reduced the number of AND and OR gates, we see that the two variables  $w$  and  $y$  are not needed. By the way, you have probably encountered a circuit that implements this function. A light controlled by two switches typically does this. □

As you probably expect by now a Karnaugh map also works when a function is specified as a product of sums. The differences are:

1. maxterms are numbered 0 for uncomplemented variables and 1 for complemented, and
2. a 0 is placed in each cell of the Karnaugh map that corresponds to a maxterm.

To see how this works let us first compare the Karnaugh maps for two functions,

$$F_1(x, y, z) = (x' \cdot y' \cdot z') \quad (4.32)$$

$$F_2(x, y, z) = (x + y + z) \quad (4.33)$$

$F_1$  is a sum of products with only one minterm, and  $F_2$  is a product of sums with only one maxterm. Figure 4.12(a) shows how the minterm appears on a Karnaugh map, and Figure 4.12(b) shows the maxterm.

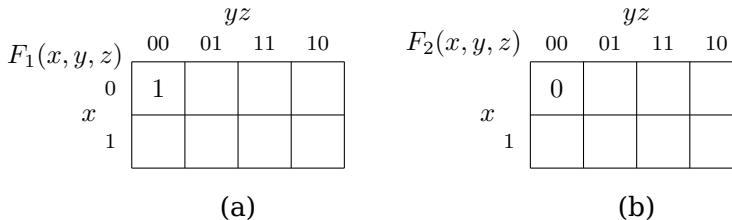


Figure 4.12: Comparison of one minterm (a) versus one maxterm (b) on a Karnaugh map.

Figure 4.13 shows how three-variable maxterms map onto a Karnaugh map. As with minterms,  $x$  is on the vertical axis,  $y$  and  $z$  on the horizontal. To use the Karnaugh map for maxterms, place a 0 in each cell corresponding to a maxterm.

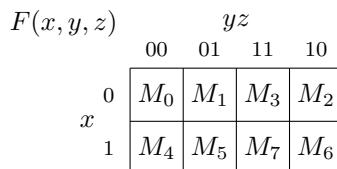


Figure 4.13: Mapping of three-variable maxterms on a Karnaugh map.

A four-variable Karnaugh map of maxterms is shown in Figure 4.14. The  $w$  and  $x$  variables are on the vertical axis,  $y$  and  $z$  on the horizontal.

		$yz$				
		00	01	11	10	
		00	$M_0$	$M_1$	$M_3$	$M_2$
$wx$	01	$M_4$	$M_5$	$M_7$	$M_6$	
	11	$M_{12}$	$M_{13}$	$M_{15}$	$M_{14}$	
		10	$M_8$	$M_9$	$M_{11}$	$M_{10}$

Figure 4.14: Mapping of four-variable minterms on a Karnaugh map.

### Example 4-e

Find a minimal product of sums for the function of Equation 4.26. That function is

$$\begin{aligned} F(x, y, z) = & (x + y + z) \cdot (x + y + z') \cdot (x + y' + z') \\ & \cdot (x' + y + z) \cdot (x' + y' + z') \end{aligned}$$

*Solution:*

So this expression includes maxterms 0, 1, 3, 4, and 7. These appear in a Karnaugh map:

		$yz$				
		00	01	11	10	
		0	0	0		
$x$	1	0		0		

Next we encircle the largest adjacent blocks, where the number of cells in each block is a power of two. Notice that maxterm  $M_0$  appears in two groups.

		$yz$				
		00	01	11	10	
		0	(0)	(0)	(0)	
$x$	1	(0)		(0)		

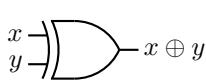
From this Karnaugh map it is very easy to write the function as a minimal product of sums:

$$F(x, y, z) = (x + y) \cdot (y + z) \cdot (y' + z')$$

which is the same as we found in Equation 4.28. □

There are situations where some minterms (or maxterms) are irrelevant in a function. This might occur, say, if certain input conditions are impossible in the design. As an example, assume that you have an application where the *exclusive or* (XOR) operation is required. The symbol for the operation and its truth table are shown in Figure 4.15. The minterms required to implement this operation are:

$$x \oplus y = x \cdot y' + x' \cdot y$$



$x$	$y$	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4.15: The XOR gate acting on two variables,  $x$  and  $y$ .

This is the simplest form of the XOR operation. It requires two AND gates, two NOT gates, and an OR gate for realization.

But let us say that we have the additional information that the two inputs,  $x$  and  $y$  can never be 1 at the same time. Then we can draw a Karnaugh map with an “ $\times$ ” for the minterm that cannot exist as shown in Figure 4.16. The “ $\times$ ” represents a “don’t care” cell — we don’t care whether this cell is grouped with other cells or not.

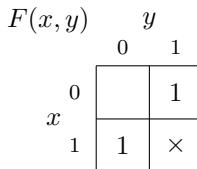


Figure 4.16: A “don’t care” cell on a Karnaugh map. Since  $x$  and  $y$  cannot both be 1 at the same time, we don’t care if the cell  $xy = 11$  is included in our groupings or not.

Since the cell that represents the minterm  $x \cdot y$  is a “don’t care”, we can include it in our minimization groupings, leading to the two groupings shown in Figure 4.17. We

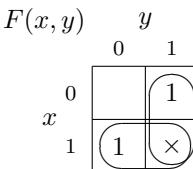


Figure 4.17: Karnaugh map for xor function if we know  $x = y = 1$  cannot occur.

easily recognize this Karnaugh map as being realizable with a single OR gate, which saves one OR gate and an AND gate.

## 4.4 Crash Course in Electronics

Although it is not necessary to be an electrical engineer in order to understand how logic gates work, some basic concepts will help. This section provides a very brief overview of the fundamental concepts of electronic circuits. We begin with two definitions.

**Current** is the movement of electrical charge. Electrical charge is measured in *coulombs*.

A flow of one coulomb per second is defined as one *ampere*, often abbreviated as one *amp*. Current only flows in a closed path through an electrical circuit.

**Voltage** is a difference in electrical potential between two points in an electrical circuit.

One *volt* is defined as the potential difference between two points on a conductor

when one ampere of current flowing through the conductor dissipates one watt of power.

The electronic circuits that make up a computer are constructed from:

- A power source that provides the electrical power.
- Passive elements that control current flow and voltage levels.
- Active elements that switch between various combinations of the power source, passive elements, and other active elements.

We will look at how each of these three categories of electronic components behaves.

#### 4.4.1 Power Supplies and Batteries

The electrical power is supplied to our homes, schools, and businesses in the form of *alternating current (AC)*. A plot of the magnitude of the voltage versus time shows a sinusoidal wave shape. Computer circuits use *direct current (DC)* power, which does not vary over time. A *power supply* is used to convert AC power to DC as shown in Figure 4.18. As you probably know, batteries also provide DC power.

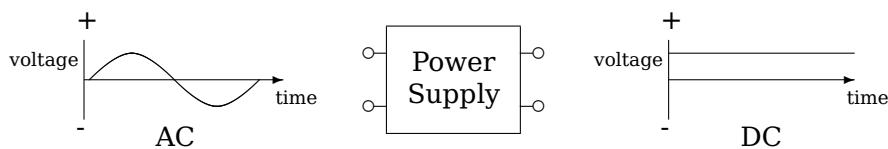


Figure 4.18: AC/DC power supply.

Computer circuits use DC power. They distinguish between two different voltage levels to provide logical 0 and 1. For example, logical 0 may be represented by 0.0 volts and logical 1 by +2.5 volts. Or the reverse may be used — +2.5 volts as logical 0 and 0.0 volts as logical 1. The only requirement is that the hardware design be consistent. Fortunately, programmers do not need to be concerned about the actual voltages used.

Electrical engineers typically think of the AC characteristics of a circuit in terms of an ongoing sinusoidal voltage. Although DC power is used, computer circuits are constantly switching between the two voltage levels. Computer hardware engineers need to consider circuit element time characteristics when the voltage is suddenly switched from one level to another. It is this *transient* behavior that will be described in the following sections.

#### 4.4.2 Resistors, Capacitors, and Inductors

All electrical circuits have resistance, capacitance, and inductance.

- *Resistance* dissipates power. The electric energy is transformed into heat.
- *Capacitance* stores energy in an electric field. Voltage across a capacitance cannot change instantaneously.
- *Inductance* stores energy in a magnetic field. Current through an inductance cannot change instantaneously.

All three of these electro-magnetic properties are distributed throughout any electronic circuit. In computer circuits they tend to limit the speed at which the circuit can operate and to consume power, collectively known as *impedance*. Analyzing their effects can be quite complicated and is beyond the scope of this book. Instead, to get a feel for the effects of each of these properties, we will consider the electronic devices that are used to add one of these properties to a specific location in a circuit; namely, resistors, capacitors, and inductors. Each of these circuit devices has a different relationship between the voltage difference across the device and the current flowing through it.

A *resistor* irreversibly transforms electrical energy into heat. It does not store energy. The relationship between voltage and current for a resistor is given by the equation

$$v = i R \quad (4.34)$$

where  $v$  is the voltage difference across the resistor at time  $t$ ,  $i$  is the current flowing through it at time  $t$ , and  $R$  is the value of the resistor. Resistor values are specified in *ohms*. The circuit shown in Figure 4.19 shows two resistors connected in series through a switch to a battery. The battery supplies 2.5 volts. The Greek letter  $\Omega$  is used to

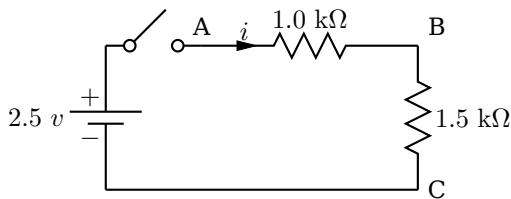


Figure 4.19: Two resistors in series.

indicate ohms, and  $k\Omega$  indicates  $10^3$  ohms. Since current can only flow in a closed path, none flows until the switch is closed.

Both resistors are in the same path, so when the switch is closed the same current flows through each of them. The resistors are said to be connected in *series*. The total resistance in the path is their sum:

$$\begin{aligned} R &= 1.0 \text{ k}\Omega + 1.5 \text{ k}\Omega \\ &= 2.5 \times 10^3 \text{ ohms} \end{aligned} \quad (4.35)$$

The amount of current can be determined from the application of Equation 4.34. Solving for  $i$ ,

$$\begin{aligned} i &= \frac{v}{R} \\ &= \frac{2.5 \text{ volts}}{2.5 \times 10^3 \text{ ohms}} \\ &= 1.0 \times 10^{-3} \text{ amps} \\ &= 1.0 \text{ ma} \end{aligned} \quad (4.36)$$

where "ma" means "milliamps."

We can now use Equation 4.34 to determine the voltage difference between points A and B.

$$\begin{aligned} v_{AB} &= i R \\ &= 1.0 \times 10^{-3} \text{ amps} \times 1.0 \times 10^3 \text{ ohms} \\ &= 1.0 \text{ volts} \end{aligned} \quad (4.37)$$

Similarly, the voltage difference between points B and C is

$$\begin{aligned} v_{BC} &= i R \\ &= 1.0 \times 10^{-3} \text{ amps} \times 1.5 \times 10^3 \text{ ohms} \\ &= 1.5 \text{ volts} \end{aligned} \quad (4.38)$$

Figure 4.20 shows the same two resistors connected in parallel. In this case, the

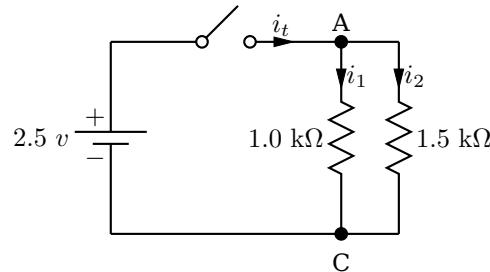


Figure 4.20: Two resistors in parallel.

voltage across the two resistors is the same: 2.5 volts when the switch is closed. The current in each one depends upon its resistance. Thus,

$$\begin{aligned} i_1 &= \frac{v}{R_1} \\ &= \frac{2.5 \text{ volts}}{1.0 \times 10^3 \text{ ohms}} \\ &= 2.5 \times 10^{-3} \text{ amps} \\ &= 2.5 \text{ ma} \end{aligned} \quad (4.39)$$

and

$$\begin{aligned} i_2 &= \frac{v}{R_2} \\ &= \frac{2.5 \text{ volts}}{1.5 \times 10^3 \text{ ohms}} \\ &= 1.67 \times 10^{-3} \text{ amps} \\ &= 1.67 \text{ ma} \end{aligned} \quad (4.40)$$

The total current,  $i_t$ , supplied by the battery when the switch is closed is divided at point A to supply both the resistors. It must equal the sum of the two currents through the resistors,

$$\begin{aligned} i_t &= i_1 + i_2 \\ &= 2.5 \text{ ma} + 1.67 \text{ ma} \\ &= 4.17 \text{ ma} \end{aligned} \quad (4.41)$$

A *capacitor* stores energy in the form of an electric field. It reacts slowly to voltage changes, requiring time for the electric field to build. The voltage across a capacitor changes with time according to the equation

$$v = \frac{1}{C} \int_0^t i \, dt \quad (4.42)$$

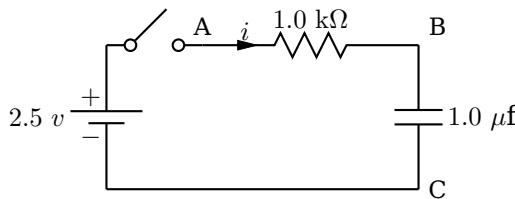


Figure 4.21: Capacitor in series with a resistor;  $v_{AB}$  is the voltage across the resistor and  $v_{BC}$  is the voltage across the capacitor.

where C is the value of the capacitor in farads.

Figure 4.21 shows a 1.0 microfarad capacitor being charged through a 1.0 kilohm resistor. This circuit is a rough approximation of the output of one transistor connected to the input of another. (See Section 4.4.3.) The output of the first transistor has resistance, and the input to the second transistor has capacitance. The switching behavior of the second transistor depends upon the voltage across the (equivalent) capacitor,  $v_{BC}$ .

Assuming the voltage across the capacitor,  $v_{BC}$ , is 0.0 volts when the switch is first closed, current flows through the resistor and capacitor. The voltage across the resistor plus the voltage across the capacitor must be equal to the voltage available from the battery. That is,

$$2.5 = i R + v_{BC} \quad (4.43)$$

If we assume that the voltage across the capacitor,  $v_{BC}$ , is 0.0 volts when the switch is first closed, the full voltage of the battery, 2.5 volts, will appear across the resistor. Thus, the initial current flow in the circuit will be

$$\begin{aligned} i_{\text{initial}} &= \frac{2.5 \text{ volts}}{1.0 \text{ k}\Omega} \\ &= 2.5 \text{ ma} \end{aligned} \quad (4.44)$$

As the voltage across the capacitor increases, according to Equation 4.42, the voltage across the resistor,  $v_{AB}$ , decreases. This results in an exponentially decreasing build up of voltage across the capacitor. When it finally equals the voltage of the battery, the voltage across the resistor is 0.0 volts and current flow in the circuit becomes zero. The rate of the exponential decrease is given by the product  $RC$ , called the *time constant*.

Using the values of R and C in Figure 4.21 we get

$$\begin{aligned} R C &= 1.0 \times 10^3 \text{ ohms} \times 1.0 \times 10^{-6} \text{ farads} \\ &= 1.0 \times 10^{-3} \text{ seconds} \\ &= 1.0 \text{ msec.} \end{aligned} \quad (4.45)$$

Thus, assuming the capacitor in Figure 4.21 has 0.0 volts across it when the switch is closed, the voltage that develops over time is given by

$$v_{BC} = 2.5 (1 - e^{-t/10^{-3}}) \quad (4.46)$$

This is shown in Figure 4.22. At the time  $t = 1.0$  millisecond (one time constant), the voltage across the capacitor is

$$\begin{aligned} v_{BC} &= 2.5 (1 - e^{-10^{-3}/10^{-3}}) \\ &= 2.5 (1 - e^{-1}) \\ &= 2.5 \times 0.63 \\ &= 1.58 \text{ volts} \end{aligned} \quad (4.47)$$

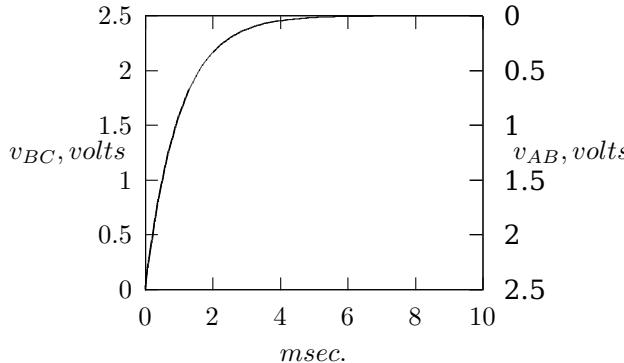


Figure 4.22: Capacitor charging over time in the circuit in Figure 4.21. The left-hand y-axis shows voltage across the capacitor, the right-hand voltage across the resistor.

After 6 time constants of time have passed, the voltage across the capacitor has reached

$$\begin{aligned}
 v_{BC} &= 2.5 (1 - e^{-6 \times 10^{-3} / 10^{-3}}) \\
 &= 2.5 (1 - e^{-6}) \\
 &= 2.5 \times 0.9975 \\
 &= 2.49 \text{ volts}
 \end{aligned} \tag{4.48}$$

At this time the voltage across the resistor is essentially 0.0 volts and current flow is very low.

*Inductors* are not used in logic circuits. In the typical PC, they are found as part of the CPU power supply circuitry. If you have access to the inside of a PC, you can probably see a small ( $\sim 1$  cm. in diameter) donut-shaped device with wire wrapped around it on the motherboard near the CPU. This is an inductor used to smooth the power supplied to the CPU.

An inductor stores energy in the form of a magnetic field. It reacts slowly to current changes, requiring time for the magnetic field to build. The relationship between voltage at time  $t$  across an inductor and current flow through it is given by the equation

$$v = L \frac{di}{dt} \tag{4.49}$$

where  $L$  is the value of the inductor in henrys.

Figure 4.23 shows an inductor connected in series with a resistor. When the switch

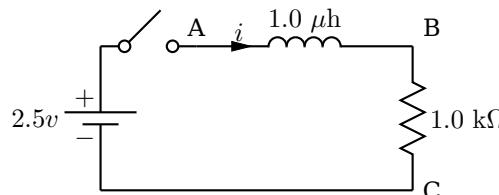


Figure 4.23: Inductor in series with a resistor.

is open no current flows through this circuit. Upon closing the switch, the inductor

initially impedes the flow of current, taking time for a magnetic field to be built up in the inductor.

At this initial point no current is flowing through the resistor, so the voltage across it,  $v_{BC}$ , is 0.0 volts. The full voltage of the battery, 2.5 volts, appears across the inductor,  $v_{AB}$ . As current begins to flow through the inductor the voltage across the resistor,  $v_{BC}$ , grows. This results in an exponentially decreasing voltage across the inductor. When it finally reaches 0.0 volts, the voltage across the resistor is 2.5 volts and current flow in the circuit is 2.5 mA.

The rate of the exponential voltage decrease is given by the time constant  $L/R$ . Using the values of R and L in Figure 4.23 we get

$$\begin{aligned}\frac{L}{R} &= \frac{1.0 \times 10^{-6} \text{ henrys}}{1.0 \times 10^3 \text{ ohms}} \\ &= 1.0 \times 10^{-9} \text{ seconds} \\ &= 1.0 \text{ nanoseconds}\end{aligned}\quad (4.50)$$

When the switch is closed, the voltage that develops across the inductor over time is given by

$$v_{AB} = 2.5 \times e^{-t/10^{-9}} \quad (4.51)$$

This is shown in Figure 4.24. Note that after about 6 nanoseconds (6 time constants) the

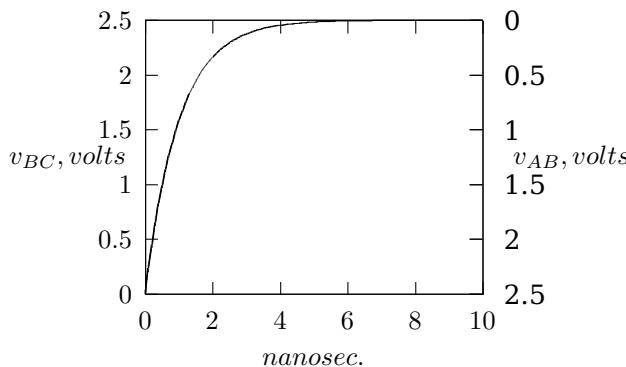


Figure 4.24: Inductor building a magnetic field over time in the circuit in Figure 4.23.

The left-hand y-axis shows voltage across the resistor, the right-hand voltage across the inductor.

voltage across the inductor is essentially equal to 0.0 volts. At this time the full voltage of the battery is across the resistor and a steady current of 2.5 mA flows.

This circuit in Figure 4.23 illustrates how inductors are used in a CPU power supply. The battery in this circuit represents the computer power supply, and the resistor represents the load provided by the CPU. The voltage produced by a power supply includes noise, which consists of small, high-frequency fluctuations added to the DC level. As can be seen in Figure 4.24, the voltage supplied to the CPU,  $v_{BC}$ , changes little over short periods of time.

### 4.4.3 CMOS Transistors

The general idea is to use two different voltages to represent 1 and 0. For example, we might use a high voltage, say +2.5 volts, to represent 1 and a low voltage, say 0.0 volts,

to represent 0. Logic circuits are constructed from components that can switch between these the high and low voltages.

The basic switching device in today's computer logic circuits is the metal-oxide-semiconductor field-effect transistor (MOSFET). Figure 4.25 shows a NOT gate implemented with a single MOSFET. The MOSFET in this circuit is an n-type. You can think

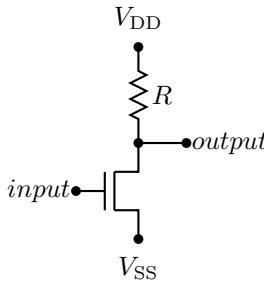


Figure 4.25: A single n-type MOSFET transistor switch.

of it as a three-terminal device. The input terminal is called the *gate*. The terminal connected to the output is the *drain*, and the terminal connected to  $V_{SS}$  is the *source*. In this circuit the drain is connected to positive (high) voltage of a DC power supply,  $V_{DD}$ , through a resistor,  $R$ . The source is connected to the zero voltage,  $V_{SS}$ .

When the input voltage to the transistor is high, the gate acquires an electrical charge, thus turning the transistor on. The path between the drain and the source of the transistor essentially become a closed switch. This causes the output to be at the low voltage. The transistor acts as a *pull down* device.

The resulting circuit is equivalent to Figure 4.26(a). In this circuit current flows from

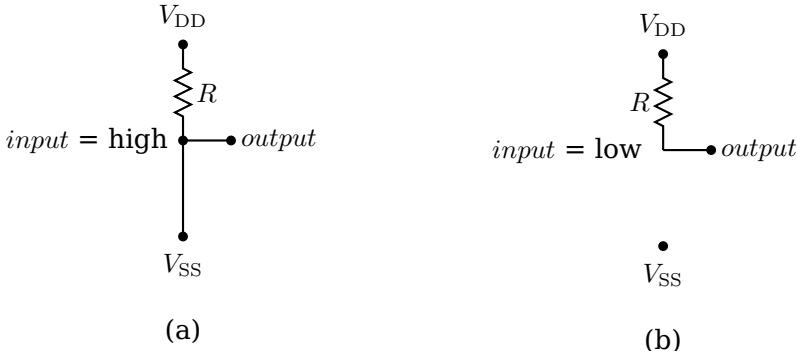


Figure 4.26: Single transistor switch equivalent circuit; (a) switch closed; (b) switch open.

$V_{DD}$  to  $V_{SS}$  through the resistor  $R$ . The output is connected to  $V_{SS}$ , that is, 0.0 volts. The current flow through the resistor and transistor is

$$i = \frac{V_{DD} - V_{SS}}{R} \quad (4.52)$$

The problem with this current flow is that it uses power just to keep the output low.

If the input is switched to the low voltage, the transistor turns off, resulting in the equivalent circuit shown in Figure 4.26(b). The output is typically connected to another transistor's input (its gate), which draws essentially no current, except during the time

it is switching from one state to the other. In the steady state condition the output connection does not draw current. Since no current flows through the resistor,  $R$ , there is no voltage change across it. So the output connection will be at  $V_{DD}$  volts, the high voltage. The resistor is acting as the *pull up* device.

These two states can be expressed in the truth table

<i>input</i>	<i>output</i>
<i>low</i>	<i>high</i>
<i>high</i>	<i>low</i>

which is the logic required of a NOT gate.

There is another problem with this hardware design. Although the gate of a MOSFET transistor draws essentially no current in order to remain in either an on or off state, current is required to cause it to change state. The gate of the transistor that is connected to the output must be charged. The gate behaves like a capacitor during the switching time. This charging requires a flow of current over a period of time. The problem here is that the resistor,  $R$ , reduces the amount of current that can flow, thus taking longer to charge the transistor gate. (See Section 4.4.2.)

From Equation 4.52, the larger the resistor, the lower the current flow. So we have a dilemma — the resistor should be large to reduce power consumption, but it should be small to increase switching speed.

This problem is solved with Complementary Metal Oxide Semiconductor (CMOS) technology. This technology packages a p-type MOSFET together with each n-type. The p-type works in the opposite way — a high value on the gate turns it off, and a low value turns it on. The circuit in Figure 4.27 shows a NOT gate using a p-type MOSFET as the pull up device.

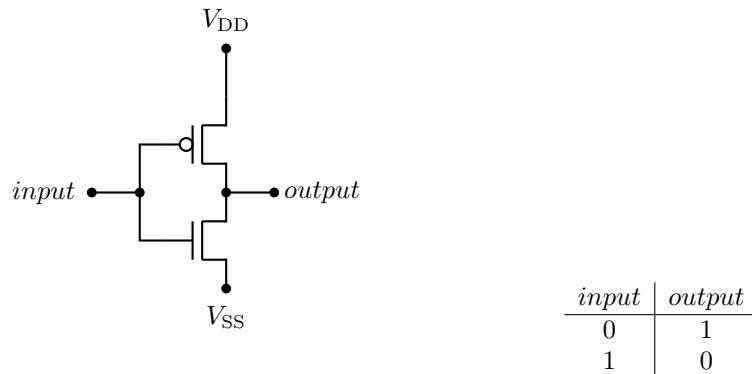


Figure 4.27: CMOS inverter (NOT) circuit.

Figure 4.28(a) shows the equivalent circuit with a high voltage input. The pull up transistor (a p-type) is off, and the pull down transistor (an n-type) is on. This results in the output being pulled down to the low voltage. In Figure 4.28(b) a low voltage input turns the pull up transistor on and the pull down transistor off. The result is the output is pulled up to the high voltage.

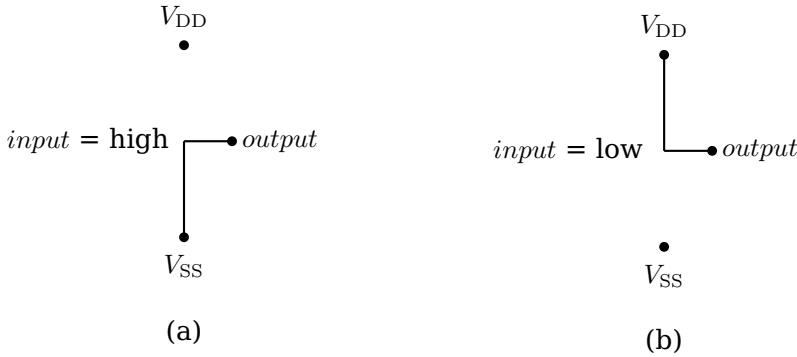


Figure 4.28: CMOS inverter equivalent circuit; (a) pull up open and pull down closed;  
 (b) pull up closed and pull down open.

Figure 4.29 shows an AND gate implemented with CMOS transistors. (See Exercise

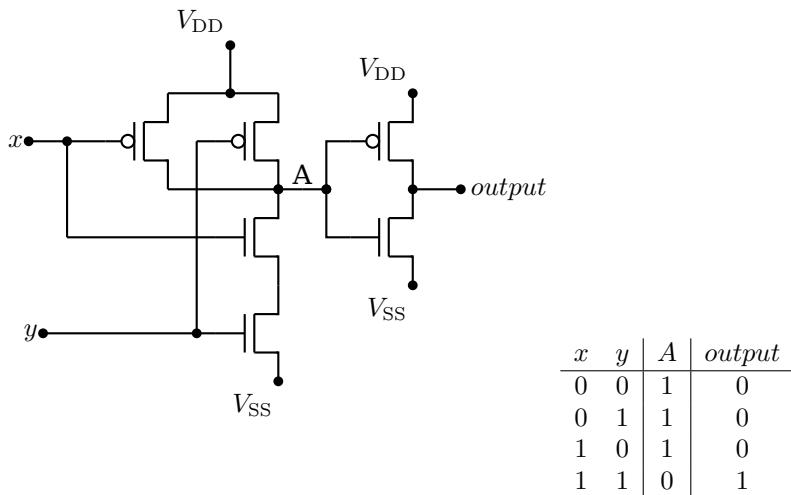


Figure 4.29: CMOS AND circuit.

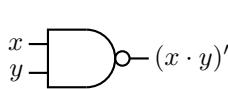
4-12.) Notice that the signal at point A is  $\text{NOT}(x \text{ AND } y)$ . The circuit from point A to the output is a NOT gate. It requires two fewer transistors than the AND operation. We will examine the implications of this result in Section 4.5.

## 4.5 NAND and NOR Gates

The discussion of transistor circuits in Section 4.4.3 illustrates a common characteristic. Because of the inherent way that transistors work, most circuits invert the signal. That is, a high voltage at the input produces a low voltage at the output and vice versa. As a result, an AND gate typically requires a NOT gate at the output in order to achieve a true AND operation.

We saw in that discussion that it takes fewer transistors to produce AND NOT than a pure AND. The combination is so common, it has been given the name NAND gate. And, of course, the same is true for OR gates, giving us a NOR gate.

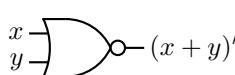
- NAND — a binary operator; the result is 0 if and only if both operands are 1; otherwise the result is 1. We will use  $(x \cdot y)'$  to designate the NAND operation. It is also common to use the ' $\uparrow$ ' symbol or simply "NAND". The hardware symbol for the NAND gate is shown in Figure 4.30. The inputs are  $x$  and  $y$ . The resulting output,  $(x \cdot y)'$ , is shown in the truth table in this figure.



$x$	$y$	$(x \cdot y)'$
0	0	1
0	1	1
1	0	1
1	1	0

Figure 4.30: The NAND gate acting on two variables,  $x$  and  $y$ .

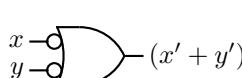
- NOR — a binary operator; the result is 0 if at least one of the two operands is 1; otherwise the result is 1. We will use  $(x + y)'$  to designate the NOR operation. It is also common to use the ' $\downarrow$ ' symbol or simply "NOR". The hardware symbol for the NOR gate is shown in Figure 4.31. The inputs are  $x$  and  $y$ . The resulting output,  $(x + y)'$ , is shown in the truth table in this figure.



$x$	$y$	$(x + y)'$
0	0	1
0	1	0
1	0	0
1	1	0

Figure 4.31: The NOR gate acting on two variables,  $x$  and  $y$ .

The small circle at the output of the NAND and NOR gates signifies "NOT", just as with the NOT gate (see Figure 4.3). Although we have explicitly shown NOT gates when inputs to gates are complemented, it is common to simply use these small circles at the input. For example, Figure 4.32 shows an OR gate with both inputs complemented. As



$x$	$y$	$(x' + y')$
0	0	1
0	1	1
1	0	1
1	1	0

Figure 4.32: An alternate way to draw a NAND gate.

the truth table in this figure shows, this is an alternate way to draw a NAND gate. See Exercise 4-14 for an alternate way to draw a NOR gate.

One of the interesting properties about NAND gates is that it is possible to build AND, OR, and NOT gates from them. That is, the NAND gate is sufficient to implement any Boolean function. In this sense, it can be thought of as a universal gate.

First, we construct a NOT gate. To do this, simply connect the signal to both inputs of a NAND gate, as shown in Figure 4.33.

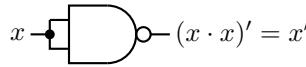


Figure 4.33: A NOT gate built from a NAND gate.

Next, we can use DeMorgan's Law to derive an AND gate.

$$\begin{aligned}
 (x \cdot y)' &= x' + y' \\
 (x' + y')' &= (x')' \cdot (y')' \\
 &= x \cdot y
 \end{aligned} \tag{4.53}$$

So we need two NAND gates connected as shown in Figure 4.34.

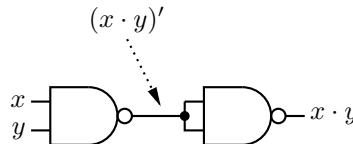


Figure 4.34: An AND gate built from two NAND gates.

Again, using DeMorgan's Law

$$\begin{aligned}
 (x' \cdot y')' &= (x')' + (y')' \\
 &= x + y
 \end{aligned} \tag{4.54}$$

we use three NAND gates connected as shown in Figure 4.35 to create an OR gate.

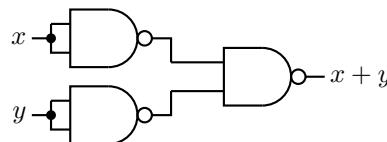


Figure 4.35: An OR gate built from three NAND gates.

It may seem like we are creating more complexity in order to build circuits from NAND gates. But consider the function

$$F(w, x, y, z) = (w \cdot x) + (y \cdot z) \tag{4.55}$$

Without knowing how logic gates are constructed, it would be reasonable to implement this function with the circuit shown in Figure 4.36. Using the involution property

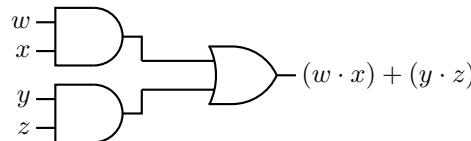


Figure 4.36: The function in Equation 4.55 using two AND gates and one OR gate.

(Equation 4.15) it is clear that the circuit in Figure 4.37 is equivalent to the one in Figure 4.36.

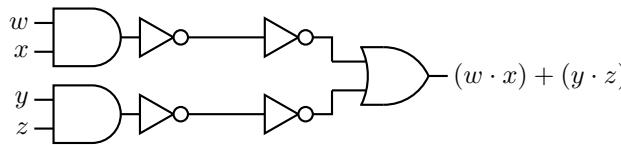


Figure 4.37: The function in Equation 4.55 using two AND gates, one OR gate and four NOT gates.

Next, comparing the AND-gate/NOT-gate combination with Figure 4.30, we see that each is simply a NAND gate. Similarly, comparing the NOT-gates/OR-gate combination with Figure 4.32, it is also a NAND gate. Thus we can also implement the function in Equation 4.55 with three NAND gates as shown in Figure 4.38.

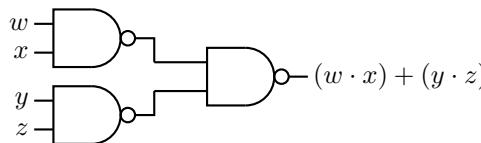


Figure 4.38: The function in Equation 4.55 using only three NAND gates.

From simply viewing the circuit diagrams, it may seem that we have not gained anything in this circuit transformation. But we saw in Section 4.4.3 that a NAND gate requires fewer transistors than an AND gate or OR gate due to the signal inversion properties of transistors. Thus, the NAND gate implementation is a less expensive and faster implementation.

The conversion from an AND/OR/NOT gate design to one that uses only NAND gates is straightforward:

1. Express the function as a minimal SoP.
2. Convert the products (AND terms) and the final sum (OR) to NANDs.
3. Add a NAND gate for any product with only a single literal.

As with software, hardware design is an iterative process. Since there usually is not a unique solution, you often need to develop several designs and analyze each one within the context of the available hardware. The example above shows that two solutions that look the same on paper may be dissimilar in hardware.

In Chapter 6 we will see how these concepts can be used to construct the heart of a computer — the CPU.

## 4.6 Exercises

- 4-1** (§4.1) Prove the *identity* property expressed by Equations 4.3 and 4.4.
- 4-2** (§4.1) Prove the *commutative* property expressed by Equations 4.5 and 4.6.
- 4-3** (§4.1) Prove the *null* property expressed by Equations 4.7 and 4.8.
- 4-4** (§4.1) Prove the *complement* property expressed by Equations 4.9 and 4.10.
- 4-5** (§4.1) Prove the *idempotent* property expressed by Equations 4.11 and 4.12.
- 4-6** (§4.1) Prove the *distributive* property expressed by Equations 4.13 and 4.14.

**4-7** (§4.1) Prove the *involution* property expressed by Equation 4.15.

**4-8** (§4.2) Show that Equations 4.18 and 4.19 represent the same function. This shows that the sum of minterms and product of maxterms are complementary.

**4-9** (§4.3.2) Show where each minterm is located with this Karnaugh map axis labeling using the notation of Figure 4.10.

		$xy$				
		00	01	11	10	
		0				
		1				

**4-10** (§4.3.2) Show where each minterm is located with this Karnaugh map axis labeling using the notation of Figure 4.10.

		$xz$				
		00	01	11	10	
		0				
		1				

**4-11** (§4.3.2) Design a logic function that detects the prime single-digit numbers. Assume that the numbers are coded in 4-bit BCD (see Section 3.6.1, page 55). The function is 1 for each prime number.

**4-12** (§4.4.3) Using drawings similar to those in Figure 4.28, verify that the logic circuit in Figure 4.29 is an AND gate.

**4-13** (§4.5) Show that the gate in Figure 4.32 is a NAND gate.

**4-14** (§4.5) Give an alternate way to draw a NOR gate, similar to the alternate NAND gate in Figure 4.32.

**4-15** (§4.5) Design a circuit using NAND gates that detects the “below” condition for two 2-bit values. That is, given two 2-bit variables  $x$  and  $y$ ,  $F(x, y) = 1$  when the unsigned integer value of  $x$  is less than the unsigned integer value of  $y$ .

- a) Give a truth table for the output of the circuit,  $F$ .
- b) Find a minimal sum of products for  $F$ .
- c) Implement  $F$  using NAND gates.

# Chapter 5

# Logic Circuits

In this chapter we examine how the concepts in Chapter 4 can be used to build some of the logic circuits that make up a CPU, Memory, and other devices. We will not describe an entire unit, only a few small parts. The goal is to provide an introductory overview of the concepts. There are many excellent books that cover the details. For example, see [20], [23], or [24] for circuit design details and [28], [31], [34] for CPU architecture design concepts.

Logic circuits can be classified as either

- **Combinational Logic Circuits** — the output(s) depend only on the input(s) at any specific time and not on any previous input(s).
- **Sequential Logic Circuits** — the output(s) depend both on previous and current input(s).

An example of the two concepts is a television remote control. You can enter a number and the output (a particular television channel) depends only on the number entered. It does not matter what channels been viewed previously. So the relationship between the input (a number) and the output is *combinational*.

The remote control also has inputs for stepping either up or down one channel. When using this input method, the channel selected depends on what channel has been previously selected and the sequence of up/down button pushes. The channel up/down buttons illustrate a *sequential* input/output relationship.

Although a more formal definition will be given in Section 5.3, this television example also illustrates the concept of *state*. My television remote control has a button I can push that will show the current channel setting. If I make a note of the beginning channel setting, and keep track of the sequence of channel up and down button pushes, I will know the ending channel setting. It does not matter how I originally got to the beginning channel setting. The channel setting is the state of the channel selection mechanism because it tells me everything I need to know in order to select a new channel by using a sequence of channel up and down button pushes.

## 5.1 Combinational Logic Circuits

*Combinational* logic circuits have no memory. The output at any given time depends completely upon the circuit configuration and the input(s).

### 5.1.1 Adder Circuits

One of the most fundamental operations the ALU must do is to add two bits. We begin with two definitions. (The reason for the names will become clear later in this section.)

**half adder:** A combinational logic device that has two 1-bit inputs,  $x_i$  and  $y_i$ , and two outputs that are related as shown by the truth table:

$x_i$	$y_i$	$Carry_{i+1}$	$Sum_i$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

where  $x_i$  is the  $i^{th}$  bit of the multiple bit value,  $x$ ;  $y_i$  is the  $i^{th}$  bit of the multiple bit value,  $y$ ;  $Sum_i$  is the  $i^{th}$  bit of the multiple bit value,  $Sum$ ;  $Carry_{i+1}$  is the carry from adding the next-lower significant bits,  $x_i$ ,  $y_i$ .

**full adder:** A combinational logic device that has three 1-bit inputs,  $Carry_i$ ,  $x_i$ , and  $y_i$ , and two outputs that are related by the truth table:

$Carry_i$	$x_i$	$y_i$	$Carry_{i+1}$	$Sum_i$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

where  $x_i$  is the  $i^{th}$  bit of the multiple bit value,  $x$ ;  $y_i$  is the  $i^{th}$  bit of the multiple bit value,  $y$ ;  $Sum_i$  is the  $i^{th}$  bit of the multiple bit value,  $Sum$ ;  $Carry_{i+1}$  is the carry from adding the next-lower significant bits,  $x_i$ ,  $y_i$ , and  $Carry_i$ .

First, let us look at the Karnaugh map for the sum:

		$x_i y_i$			
		00	01	11	10
$Carry_i$	0		1		1
	1	1		1	

There are no obvious groupings. We can write the function as a sum of product terms from the Karnaugh map.

$$\begin{aligned} Sum_i(Carry_i, x_i, y_i) = & Carry'_i \cdot x'_i \cdot y_i + Carry'_i \cdot x_i \cdot y'_i \\ & + Carry_i \cdot x'_i \cdot y'_i + Carry_i \cdot x_i \cdot y_i \end{aligned} \quad (5.1)$$

In the Karnaugh map for carry:

		$x_i y_i$			
		00	01	11	10
$Carry_i$	0			1	
	1		1	1	1

we can see three groupings:

		$x_i y_i$		
		00	01	11
		0		1
$Carry_i$	0			
	1		1	1

These groupings yield a three-term function that defines when  $Carry_{i+1} = 1$ :

$$Carry_{i+1} = x_i \cdot y_i + Carry_i \cdot x'_i \cdot y_i + Carry_i \cdot x_i \cdot y'_i \quad (5.2)$$

Equations 5.1 and 5.2 lead directly to the circuit for an adder in Figure 5.1.

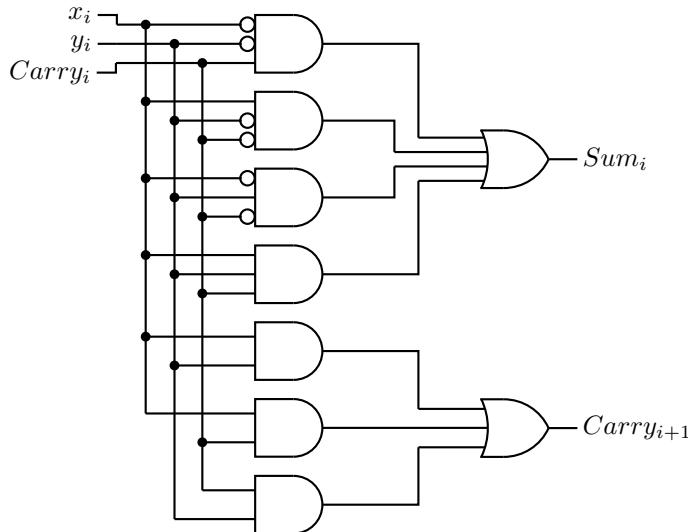


Figure 5.1: An adder circuit.

For a different approach, we look at the definition of half adder. The sum is simply the XOR of the two inputs, and the carry is the AND of the two inputs. This leads to the circuit in Figure 5.2.

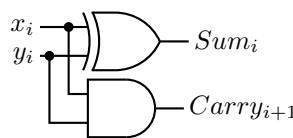


Figure 5.2: A half adder circuit.

Instead of using Karnaugh maps, we will perform some algebraic manipulations on Equation 5.1. Using the distribution rule, we can rearrange:

$$\begin{aligned} Sum_i(Carry_i, x_i, y_i) &= Carry'_i \cdot (x'_i \cdot y_i + x \cdot y'_i) + Carry_i \cdot (x'_i \cdot y'_i + x_i \cdot y_i) \\ &= Carry'_i \cdot (x_i \oplus y_i) + Carry_i \cdot (x'_i \cdot y'_i + x_i \cdot y_i) \end{aligned} \quad (5.3)$$

Let us manipulate the last product term in Equation 5.3.

$$\begin{aligned}x'_i \cdot y'_i + x_i \cdot y_i &= x_i \cdot x'_i + x_i \cdot y_i + x'_i \cdot y'_i + y_i \cdot y'_i \\&= x_i \cdot (x'_i + y_i) + y'_i \cdot (x'_i + y_i) \\&= (x_i + y'_i) \cdot (x'_i + y_i)' \\&= (x_i \oplus y_i)'\end{aligned}$$

Thus,

$$\begin{aligned}Sum_i(Carry_i, x_i, y_i) &= Carry'_i \cdot (x_i \oplus y_i) + Carry_i \cdot (x_i \oplus y_i)' \\&= Carry_i \oplus (x_i \oplus y_i)\end{aligned}\tag{5.4}$$

Similarly, we can rewrite Equation 5.2:

$$\begin{aligned}Carry_{i+1} &= x_i \cdot y_i + Carry_i \cdot x'_i \cdot y_i + Carry_i \cdot x_i \cdot y'_i \\&= x_i \cdot y_i + Carry_i \cdot (x'_i \cdot y_i + x_i \cdot y'_i) \\&= x_i \cdot y_i + Carry_i \cdot (x_i \oplus y_i)\end{aligned}\tag{5.5}$$

You should be able to see two other possible groupings on this Karnaugh map and may wonder why they are not circled here. The two ungrouped minterms,  $Carry_i \cdot x'_i \cdot y_i$  and  $Carry_i \cdot x_i \cdot y'_i$ , form a pattern that suggests an *exclusive or* operation.

Notice that the first product term in Equation 5.5,  $x_i \cdot y_i$ , is generated by the *Carry* portion of a half-adder, and that the *exclusive or* portion,  $x_i \oplus y_i$ , of the second product term is generated by the *Sum* portion. A logic gate implementation of a full adder is shown in Figure 5.3. You can see that it is implemented using two half adders and an

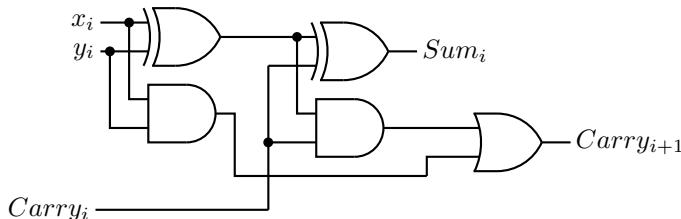


Figure 5.3: Full adder using two half adders.

OR gate. And now you understand the terminology “half adder” and “full adder.”

We cannot say which of the two adder circuits, Figure 5.1 or Figure 5.3, is better from just looking at the logic circuits. Good engineering design depends on many factors, such as how each logic gate is implemented, the cost of the logic gates and their availability, etc. The two designs are given here to show that different approaches can lead to different, but functionally equivalent, designs.

### 5.1.2 Ripple-Carry Addition/Subtraction Circuits

An  $n$ -bit adder can be implemented with  $n$  full adders. Figure 5.4 shows a 4-bit adder. Addition begins with the full adder on the right receiving the two lowest-order bits,  $x_0$  and  $y_0$ . Since this is the lowest-order bit there is no carry and  $c_0 = 0$ . The bit sum is  $s_0$ , and the carry from this addition,  $c_1$ , is connected to the carry input of the next full adder to the left, where it is added to  $x_1$  and  $y_1$ .

So the  $i^{th}$  full adder adds the two  $i^{th}$  bits of the operands, plus the carry (which is either 0 or 1) from the  $(i - 1)^{th}$  full adder. Thus, each full adder handles one bit (often

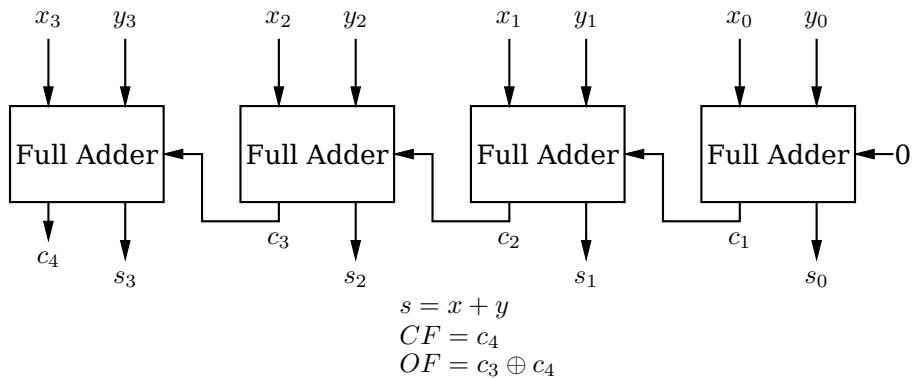


Figure 5.4: Four-bit adder.

referred to as a “slice”) of the total width of the values being added, and the carry “ripples” from the lowest-order place to the highest-order.

The final carry from the highest-order full adder,  $c_4$  in the 4-bit adder of Figure 5.4, is stored in the CF bit of the Flags register (see Section 6.2). And the exclusive or of the final carry and penultimate carry,  $c_4 \oplus c_3$  in the 4-bit adder of Figure 5.4, is stored in the OF bit.

Recall that in the 2’s complement code for storing integers a number is negated by taking its 2’s complement. So we can subtract  $y$  from  $x$  by doing:

$$\begin{aligned}
 x - y &= x + (2\text{'s complement of } y) \\
 &= x + [(y\text{'s bits flipped}) + 1]
 \end{aligned} \tag{5.6}$$

Thus, subtraction can be performed with our adder in Figure 5.4 if we complement each  $y_i$  and set the initial carry in to 1 instead of 0. Each  $y_i$  can be complemented by XOR-ing it with 1. This leads to the 4-bit circuit in Figure 5.5 that will add two 4-bit numbers when  $func = 0$  and subtract them when  $func = 1$ .

There is, of course, a time delay as the sum is computed from right to left. The computation time can be significantly reduced through more complex circuit designs that pre-compute the carry.

### 5.1.3 Decoders

Each instruction must be decoded by the CPU before the instruction can be carried out. In the x86-64 architecture the instruction for copying the 64 bits of one register to another register is

0100 0s0d 1000 1001 11ss sddd

where “ssss” specifies the source register and “dddd” specifies the destination register. (Yes, the bits that specify the registers are distributed through the instruction in this manner. You will learn more about this seemingly odd coding pattern in Chapter 9.) For example,

0100 0001 1000 1001 1100 0101

causes the ALU to copy the 64-bit value in register 0000 to register 1101. You will see in Chapter 9 that this instruction is written in assembly language as:

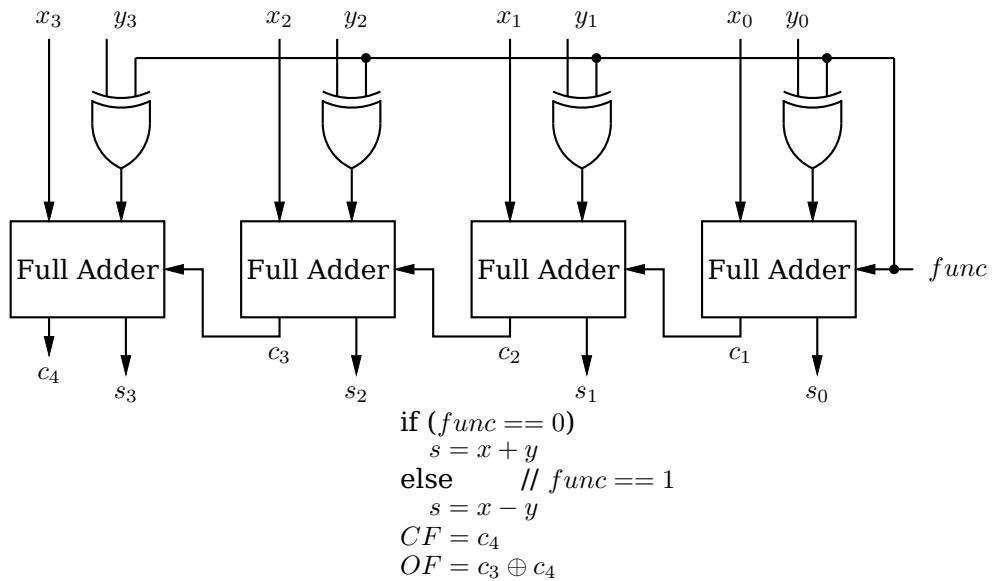


Figure 5.5: Four-bit adder/subtractor.

```
movq    %rax, %r13
```

The Control Unit must select the correct two registers based on these two 4-bit patterns in the instruction. It uses a *decoder* circuit to perform this selection.

**decoder:** A device with  $n$  binary inputs and  $2^n$  binary outputs. Each bit pattern at the input causes exactly one of the  $2^n$  to equal 1.

A decoder can be thought of as converting an  $n$ -bit input to a  $2^n$  output. But while the input can be an arbitrary bit pattern, each corresponding output value has only one bit set to 1.

In some applications not all the  $2^n$  outputs are used. For example, Table 5.1 is a truth table that shows how a decoder can be used to convert a BCD value to its corresponding decimal numeral display. A 1 in a “display” column means that is the numeral that is

input				display									
$x_3$	$x_2$	$x_1$	$x_0$	'9'	'8'	'7'	'6'	'5'	'4'	'3'	'2'	'1'	'0'
0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	1	0	0	0	0	0	1	0	0	0	0
0	1	1	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0

Table 5.1: BCD decoder. The 4-bit input causes the numeral with a 1 in its column to be displayed.

selected by the corresponding 4-bit input value. There are six other possible outputs

corresponding to the input values 1010 - 1111. But these input values are illegal in BCD, so these outputs are simply ignored.

It is common for decoders to have an additional input that is used to enable the output. The truth table in Table 5.2 shows a decoder with a 3-bit input, an enable line, and an 8-bit ( $2^3$ ) output. The output is 0 whenever  $enable = 0$ . When  $enable = 1$ , the  $i^{th}$

enable	$x_2$	$x_1$	$x_0$	$y_7$	$y_6$	$y_5$	$y_4$	$y_3$	$y_2$	$y_1$	$y_0$
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

Table 5.2: Truth table for a  $3 \times 8$  decoder with  $enable$ . If  $enable = 0$ ,  $y = 0$ . If  $enable = 1$ ,  $x = i \Rightarrow y_i = 1$  and  $y_j = 0$  for all  $j \neq i$ .

output bit is 1 if and only if the binary value of the input is equal to  $i$ . For example, when  $enable = 1$  and  $x = 011_2$ ,  $y = 00001000_2$ . That is,

$$\begin{aligned} y_3 &= x'_2 \cdot x_1 \cdot x_0 \\ &= m_3 \end{aligned} \quad (5.7)$$

This clearly generalizes such that we can give the following description of a decoder:

1. For  $n$  input bits (excluding an  $enable$  bit) there are  $2^n$  output bits.
2. The  $i^{th}$  output bit is equal to the  $i^{th}$  minterm for the  $n$  input bits.

The  $3 \times 8$  decoder specified in Table 5.2 can be implemented with 4-input AND gates as shown in Figure 5.6.

Decoders are more versatile than it might seem at first glance. Each possible input can be seen as a minterm. Since each output is one only when a particular minterm evaluates to one, a decoder can be viewed as a “minterm generator.” We know that any logical expression can be represented as the OR of minterms, so it follows that we can implement any logical expression by ORing the output(s) of a decoder.

For example, let us rewrite Equation 5.1 for the Sum expression of a full adder using minterm notation (see Section 4.3.2):

$$Sum_i(Carry_i, x_i, y_i) = m_1 + m_2 + m_4 + m_7 \quad (5.8)$$

And for the Carry expression:

$$Carry_{i+1}(Carry_i, x_i, y_i) = m_3 + m_5 + m_6 + m_7 \quad (5.9)$$

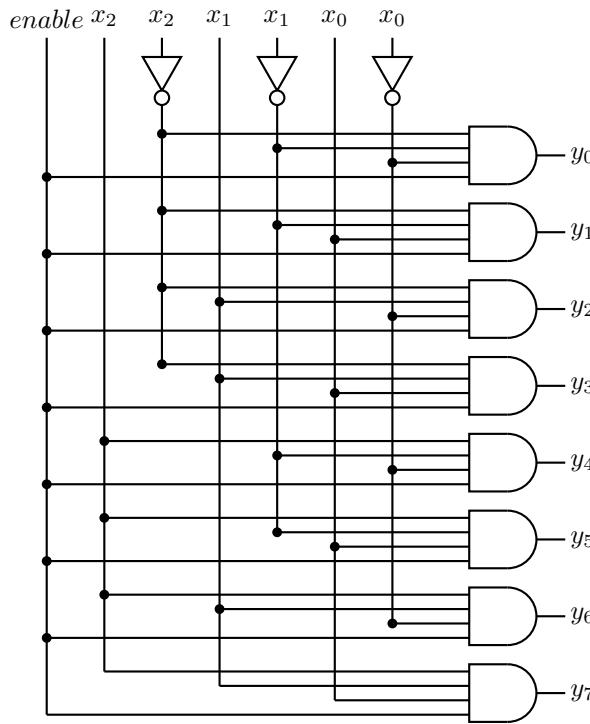


Figure 5.6: Circuit for a  $3 \times 8$  decoder with enable.

where the subscripts on  $x$ ,  $y$ , and *Carry* refer to the bit slice and the subscripts on  $m$  are part of the minterm notation. We can implement a full adder with a  $3 \times 8$  decoder and two 4-input OR gates, as shown in Figure 5.7.

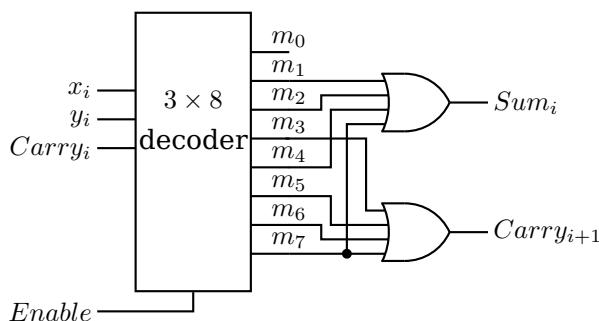


Figure 5.7: Full adder implemented with  $3 \times 8$  decoder. This is for one bit slice. An n-bit adder would require n of these circuits.

### 5.1.4 Multiplexers

There are many places in the CPU where one of several signals must be selected to pass onward. For example, as you will see in Chapter 9, a value to be added by the ALU may come from a CPU register, come from memory, or actually be stored as part of the instruction itself. The device that allows this selection is essentially a switch.

**multiplexer:** A device that selects one of multiple inputs to be passed on as the output based on one or more selection lines. Up to  $2^n$  inputs can be selected by  $n$  selection lines. Also called a *mux*.

Figure 5.8 shows a multiplexer that can switch between two different inputs,  $x$  and  $y$ . The select input,  $s$ , determines which of the sources, either  $x$  or  $y$ , is passed on to the output. The action of this 2-way multiplexer is most easily seen in a truth table:

$s$	Output
1	$x$
0	$y$

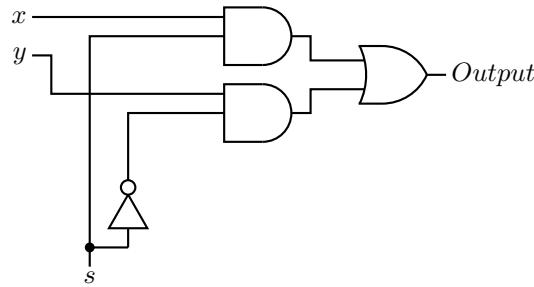


Figure 5.8: A 2-way multiplexer.

Here is a truth table for a multiplexer that can switch between four inputs,  $w$ ,  $x$ ,  $y$ , and  $z$ :

$s_1$	$s_0$	Output
0	0	$w$
0	1	$x$
1	0	$y$
1	1	$z$

That is,

$$\text{Output} = s'_0 \cdot s'_1 \cdot w + s'_0 \cdot s_1 \cdot x + s_0 \cdot S'_1 \cdot y + s_0 \cdot s_1 \cdot z \quad (5.10)$$

which is implemented as shown in Figure 5.9. The symbol for this multiplexer is shown in Figure 5.10. Notice that the selection input,  $s$ , must be 2 bits in order to select between four inputs. In general, a  $2^n$ -way multiplexer requires an  $n$ -bit selection input.

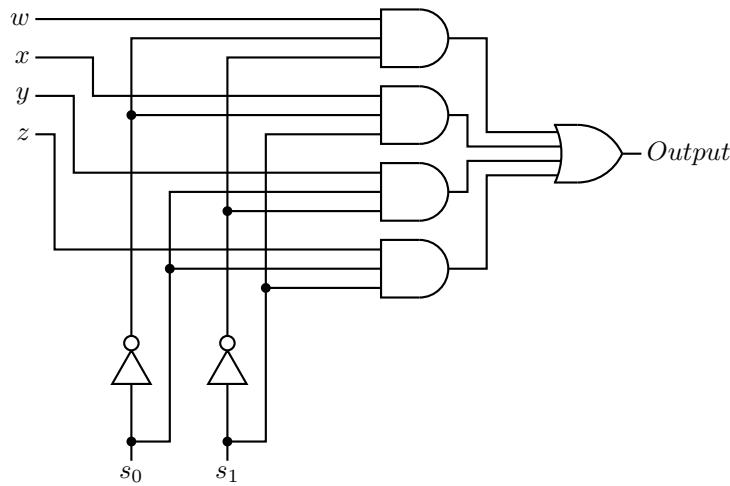


Figure 5.9: A 4-way multiplexer.

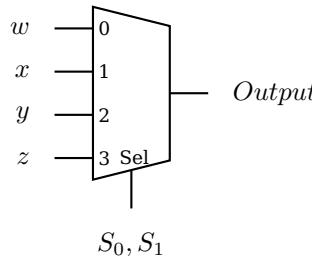


Figure 5.10: Symbol for a 4-way multiplexer.

## 5.2 Programmable Logic Devices

Combinational logic circuits can be constructed from *programmable logic devices (PLDs)*. The general idea is illustrated in Figure 5.11 for two input variables and two output functions of these variables. Each of the input variables, both in its uncomplemented and complemented form, are inputs to AND gates through fuses. (The "S" shaped lines in the circuit diagram represent fuses.) The fuses can be "blown" or left in place in order to program each AND gate to output a product. Since every input, plus its complement, is input to each AND gate, any of the AND gates can be programmed to output a minterm.

The products produced by the array of AND gates are all connected to OR gates, also through fuses. Thus, depending on which OR-gate fuses are left in place, the output of each OR gate is a sum of products. There may be additional logic circuitry to select between the different outputs. We have already seen that any Boolean function can be expressed as a sum of products, so this logic device can be programmed by "blowing" the fuses to implement any Boolean function.

PLDs come in many configurations. Some are pre-programmed at the time of manufacture. Others are programmed by the manufacturer. And there are types that can be programmed by a user. Some can even be erased and reprogrammed. Programming technologies range from specifying the manufacturing mask (for the pre-programmed devices) to inexpensive electronic programming systems. Some devices use "antifuses" instead of fuses. They are normally open. Programming such devices consists of completing the connection instead of removing it.

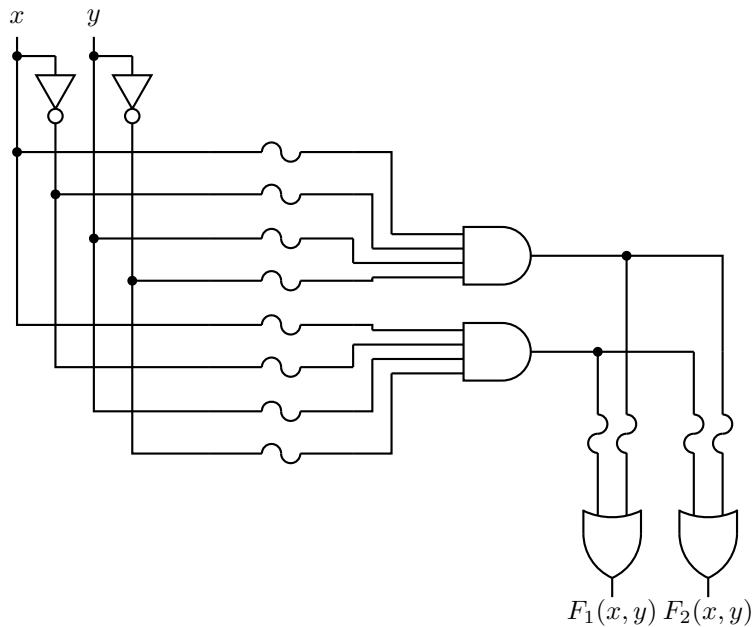


Figure 5.11: Simplified circuit for a programmable logic array. The “S” shaped line at the inputs to each gate represent fuses. The fuses are “blown” to remove that input.

There are three general categories of PLDs:

**Programmable Logic Array (PLA):** Both the AND gate plane and the OR gate plane are programmable.

**Read Only Memory (ROM):** Only the OR gate plane is programmable.

**Programmable Array Logic (PAL):** Only the AND gate plane is programmable.

We will now look at each category in more detail.

### 5.2.1 Programmable Logic Array (PLA)

Programmable logic arrays are typically larger than the one shown in Figure 5.11, which is already complicated to draw. Figure 5.12 shows how PLAs are typically diagrammed. This diagram deserves some explanation. Note in Figure 5.11 that each input variable and its complement is connected to the inputs of all the AND gates through a fuse. The AND gates have multiple inputs — one for each variable and its complement. Thus, the horizontal line leading to the inputs of the AND gates represent multiple wires. The diagram of Figure 5.12 has four input variables. So each AND gate has eight inputs, and the horizontal lines each represent the eight wires coming from the inputs and their complements.

The dots at the intersections of the vertical and horizontal line represent places where the fuses have been left intact. For example, the three dots on the topmost horizontal line indicate that there are three inputs to that AND gate. The output of the topmost AND gate is

$$w' \cdot y \cdot z$$

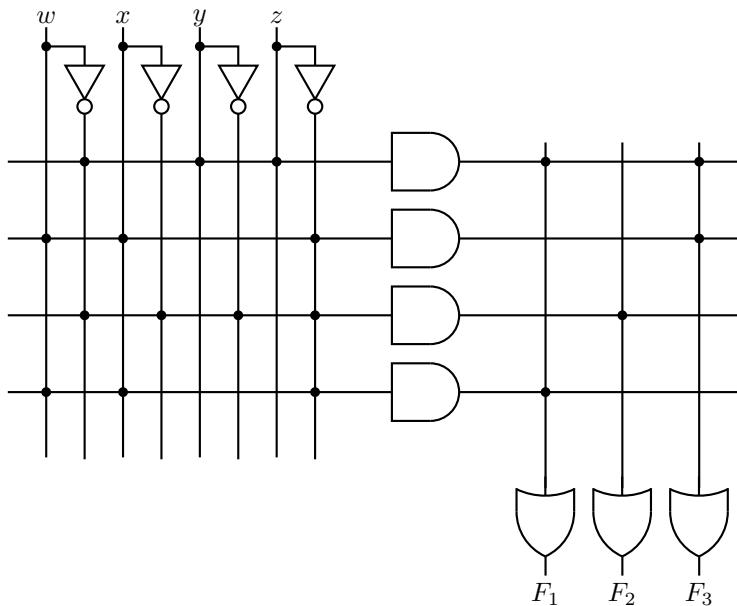


Figure 5.12: Programmable logic array schematic. The horizontal lines to the AND gate inputs represent multiple wires — one for each input variable and its complement. The vertical lines to the OR gate inputs also represent multiple wires — one for each AND gate output. The dots represent connections.

Referring again to Figure 5.11, we see that the output from each AND gate is connected to each of the OR gates. Therefore, the OR gates also have multiple inputs — one for each AND gate — and the vertical lines leading to the OR gate inputs represent multiple wires. The PLA in Figure 5.12 has been programmed to provide the three functions:

$$F_1(w, x, y, z) = w' \cdot y \cdot z + w \cdot x \cdot z' \quad (5.11)$$

$$F_2(w, x, y, z) = w' \cdot x' \cdot y' \cdot z' \quad (5.12)$$

$$F_3(w, x, y, z) = w' \cdot y \cdot z + w \cdot x \cdot z' \quad (5.13)$$

## 5.2.2 Read Only Memory (ROM)

Read only memory can be implemented as a programmable logic device where only the OR plane can be programmed. The AND gate plane is wired to provide all the minterms. Thus, the inputs to the ROM can be thought of as addresses. Then the OR gate plane is programmed to provide the bit pattern at each address.

For example, the ROM diagrammed in Figure 5.13 has two inputs,  $a_1$  and  $a_0$ . The AND gates are wired to give the minterms:

minterm	address
$a'_1 a'_0$	00
$a'_1 a_0$	01
$a_1 a'_0$	10
$a_1 a_0$	11

And the OR gate plane has been programmed to store the four characters (in ASCII code):

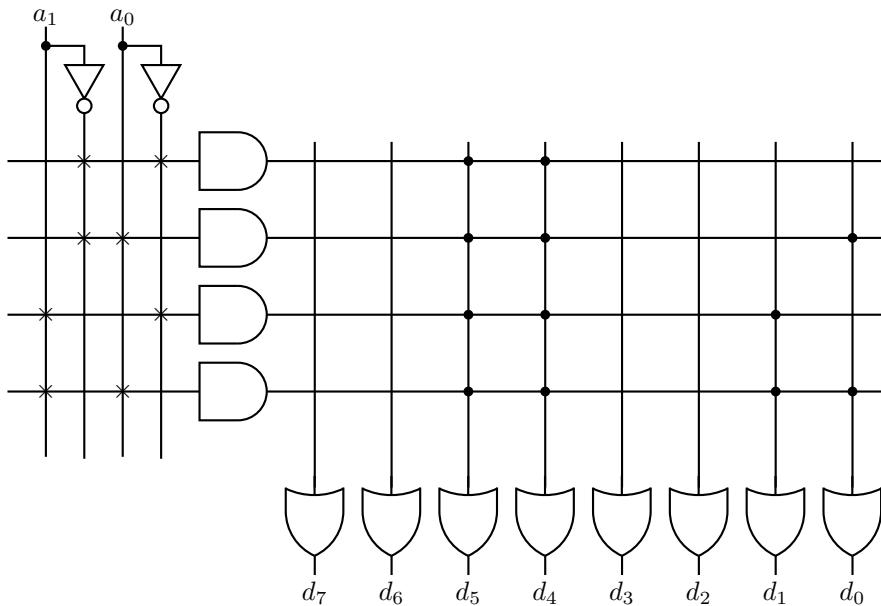


Figure 5.13: Eight-byte Read Only Memory (ROM). The “ $\times$ ” connections represent permanent connections. Each AND gate can be thought of as producing an address. The eight OR gates produce one byte. The connections (dots) in the OR plane represent the bit pattern stored at the address.

minterm	address	contents
$a'_1 a'_0$	00	'0'
$a'_1 a_0$	01	'1'
$a_1 a'_0$	10	'2'
$a_1 a_0$	11	'3'

You can see from this that the terminology “Read Only Memory” is perhaps a bit misleading. It is actually a *combinational* logic circuit. Strictly speaking, memory has a state that can be changed by inputs. (See Section 5.3.)

### 5.2.3 Programmable Array Logic (PAL)

In a Programmable Array Logic (PAL) device, each OR gate is permanently wired to a group of AND gates. Only the AND gate plane is programmable. The PAL diagrammed in Figure 5.14 has four inputs. It provides two outputs, each of which can be the sum of up to four products. The “ $\times$ ” connections in the OR gate plane show that the top four AND gates are summed to produce  $F_1$  and the lower four to produce  $F_2$ . The AND gate plane in this figure has been programmed to produce the two functions:

$$F_1(w, x, y, z) = w \cdot x' \cdot z + w' \cdot x + w \cdot x \cdot y' + w' \cdot x' \cdot y' \cdot z' \quad (5.14)$$

$$F_2(w, x, y, z) = w' \cdot y \cdot z + w \cdot x \cdot z' + w \cdot x \cdot y \cdot z + w \cdot x \cdot y' \cdot z' \quad (5.15)$$

## 5.3 Sequential Logic Circuits

Combinational circuits (Section 5.1) are instantaneous (except for the time required for the electronics to settle). Their output depends only on the input at the time the

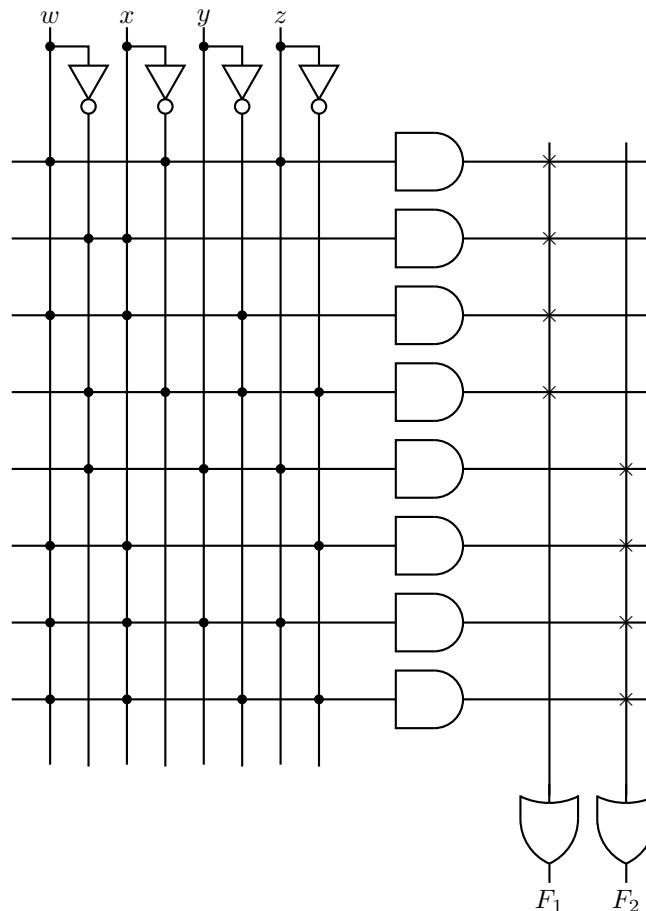


Figure 5.14: Two-function Programmable Array Logic (PAL). The “ $\times$ ” connections represent permanent connections. Each AND gate can be thought of as producing an address. The eight OR gates produce one byte. The connections (dots) in the OR plane represent the bit pattern stored at the address.

output is observed. Sequential logic circuits, on the other hand, have a time history. That history is summarized by the current *state* of the circuit.

**state:** The state of a system is the description of the system such that knowing

- (a) the state at time  $t_0$ , and
- (b) the input(s) from time  $t_0$  through time  $t_1$ ,

uniquely determines

- (c) the state at time  $t_1$ , and
- (d) the output(s) from time  $t_0$  through time  $t_1$ .

This definition means that knowing the state of a system at a given time tells you everything you need to know in order to specify its behavior from that time on. How it got into this state is irrelevant.

This definition implies that the system has memory in which the state is stored. Since there are a finite number of states, the term *finite state machine (FSM)* is commonly used. Inputs to the system can cause the state to change.

If the output(s) depend only on the state of the FSM, it is called a *Moore machine*. And if the output(s) depend on both the state and the current input(s), it is called a *Mealy machine*.

The most commonly used sequential circuits are *synchronous* — their action is controlled by a sequence of *clock pulses*. The clock pulses are created by a *clock generator* circuit. The clock pulses are applied to all the sequential elements, thus causing them to operate in synchrony.

*Asynchronous* sequential circuits are not based on a clock. They depend upon a timing delay built into the individual elements. Their behavior depends upon the order in which inputs are applied. Hence, they are difficult to analyze and will not be discussed in this book.

### 5.3.1 Clock Pulses

A clock signal is typically a square wave that alternates between the 0 and 1 levels as shown in Figure 5.15. The amount of time spent at each level may be unequal. Although not a requirement, the timing pattern is usually uniform.

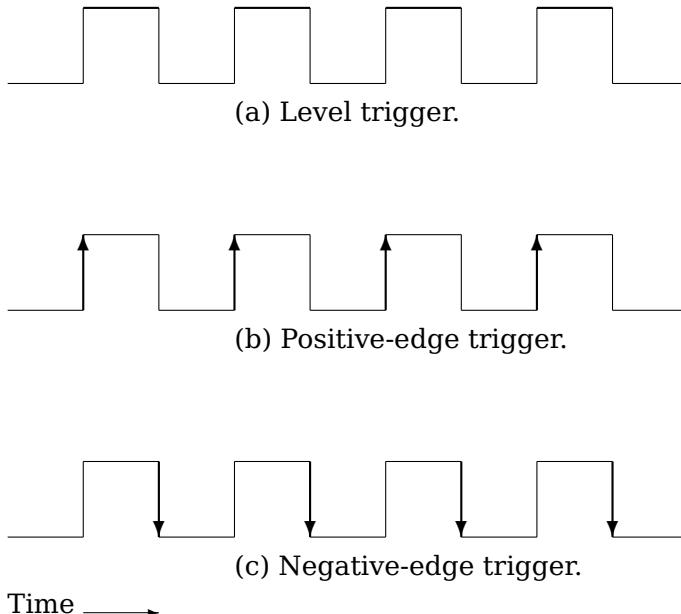


Figure 5.15: Clock signals. (a) For level-triggered circuits. (b) For positive-edge triggering. (c) For negative-edge triggering.

In Figure 5.15(a), the circuit operations take place during the entire time the clock is at the 1 level. As will be explained below, this can lead to unreliable circuit behavior. In order to achieve more reliable behavior, most circuits are designed such that a *transition* of the clock signal triggers the circuit elements to start their respective operations. Either a positive-going (Figure 5.15(b)) or negative-going (Figure 5.15(c)) transition may be used. The clock frequency must be slow enough such that all the circuit elements have time to complete their operations before the next clock transition (in the same direction) occurs.

### 5.3.2 Latches

A latch is a storage device that can be in one of two states. That is, it stores one bit. It can be constructed from two or more gates connected such that feedback maintains the state as long as power is applied. The most fundamental latch is the SR (Set-Reset).

A simple implementation using NOR gates is shown in Figure 5.16. When  $Q = 1$

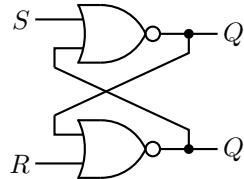


Figure 5.16: NOR gate implementation of an SR latch.

( $\Leftrightarrow Q' = 0$ ) it is in the *Set* state. When  $Q = 0$  ( $\Leftrightarrow Q' = 1$ ) it is in the *Reset* state.

There are four possible input combinations.

**$S = 0, R = 0$ : Keep current state.** If  $Q = 0$  and  $Q' = 1$ , the output of the upper NOR gate is  $(0 + 0)' = 1$ , and the output of the lower NOR gate is  $(1 + 0)' = 0$ .

If  $Q = 1$  and  $Q' = 0$ , the output of the upper NOR gate is  $(0 + 1)' = 0$ , and the output of the lower NOR gate is  $(0 + 0)' = 1$ .

Thus, the cross feedback between the two NOR gates maintains the state — *Set* or *Reset* — of the latch.

**$S = 1, R = 0$ : Set.** If  $Q = 1$  and  $Q' = 0$ , the output of the upper NOR gate is  $(1 + 1)' = 0$ , and the output of the lower NOR gate is  $(0 + 0)' = 1$ . The latch remains in the *Set* state.

If  $Q = 0$  and  $Q' = 1$ , the output of the upper NOR gate is  $(1 + 0)' = 0$ . This is fed back to the input of the lower NOR gate to give  $(0 + 0)' = 1$ . The feedback from the output of the lower NOR gate to the input of the upper keeps the output of the upper NOR gate at  $(1 + 1)' = 0$ . The latch has moved into the *Set* state.

**$S = 0, R = 1$ : Reset.** If  $Q = 1$  and  $Q' = 0$ , the output of the lower NOR gate is  $(0 + 1)' = 0$ . This causes the output of the upper NOR gate to become  $(0 + 0)' = 1$ . The feedback from the output of the upper NOR gate to the input of the lower keeps the output of the lower NOR gate at  $(1 + 1)' = 0$ . The latch has moved into the *Reset* state.

If  $Q = 0$  and  $Q' = 1$ , the output of the lower NOR gate is  $(1 + 1)' = 0$ , and the output of the upper NOR gate is  $(0 + 0)' = 1$ . The latch remains in the *Reset* state.

**$S = 1, R = 1$ : Not allowed.** If  $Q = 0$  and  $Q' = 1$ , the output of the upper NOR gate is  $(1 + 0)' = 0$ . This is fed back to the input of the lower NOR gate to give  $(0 + 1)' = 0$  as its output. The feedback from the output of the lower NOR gate to the input of the upper maintains its output as  $(1 + 0)' = 0$ . Thus,  $Q = Q' = 0$ , which is not allowed.

If  $Q = 1$  and  $Q' = 0$ , the output of the lower NOR gate is  $(0 + 1)' = 0$ . This is fed back to the input of the upper NOR gate to give  $(1 + 0)' = 0$  as its output. The feedback from the output of the upper NOR gate to the input of the lower maintains its output as  $(0 + 1)' = 0$ . Thus,  $Q = Q' = 0$ , which is not allowed.

The *state table* in Table 5.3 summarizes the behavior of a NOR-based SR latch. The inputs to a NOR-based SR latches are normally held at 0, which maintains the current state,  $Q$ . Its current state is available at the output. Momentarily changing  $S$  or  $R$  to 1 causes the state to change to *Set* or *Reset*, respectively, as shown in the  $Q_{next}$  column.

S	R	Current State	Next State
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

Table 5.3: SR latch state table. “X” indicates an indeterminate state. A circuit using this latch must be designed to prevent this input combination.

Notice that placing 1 on both the *Set* and *Reset* inputs at the same time causes a problem. Then the outputs of both NOR gates would become 0. In other words,  $Q = Q' = 0$ , which is logically impossible. The circuit design must be such to prevent this input combination.

The behavior of an SR latch can also be shown by the *state diagram* in Figure 5.17. A state diagram is a directed graph. The circles show the possible states. Lines with

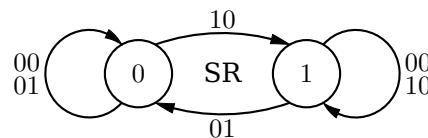


Figure 5.17: State diagram for an SR latch. There are two possible inputs, 00 or 01, that cause the latch to remain in state 0. Similarly, 00 or 10 cause it to remain in state 1. Since the output is simply the state, it is not shown in this state diagram. Notice that the input 11 is not allowed, so it is not shown on the diagram.

arrows show the possible transitions between the states and are labeled with the input that causes the transition.

The two circles in Figure 5.17 show the two possible states of the SR latch — 0 or 1. The labels on the lines show the two-bit inputs, *SR*, that cause each state transition. Notice that when the latch is in state 0 there are two possible inputs,  $SR = 00$  and  $SR = 01$ , that cause it to remain in that state. Similarly, when it is in state 1 either of the two inputs,  $SR = 00$  or  $SR = 10$ , cause it to remain in that state.

The output of the SR latch is simply the state so is not shown separately on this state diagram. In general, if the output of a circuit is dependent on the input, it is often shown on the directed lines of the state diagram in the format “*input/output*.” If the output is dependent on the state, it is more common to show it in the corresponding state circle in “*state/output*” format.

NAND gates are more commonly used than NOR gates, and it is possible to build an SR latch from NAND gates. Recalling that NAND and NOR have complementary properties, we will think ahead and use  $S'$  and  $R'$  as the inputs, as shown in Figure 5.18. Consider the four possible input combinations.

**$S' = 1, R' = 1$ : Keep current state.** If  $Q = 0$  and  $Q' = 1$ , the output of the upper NAND gate is  $(1 \cdot 1)' = 0$ , and the output of the lower NAND gate is  $(0 \cdot 1)' = 1$ .

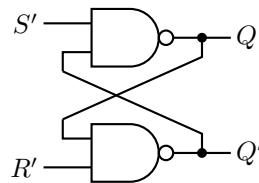


Figure 5.18: NAND gate implementation of an S'R' latch.

If  $Q = 1$  and  $Q' = 0$ , the output of the upper NAND gate is  $(1 \cdot 0)' = 1$ , and the output of the lower NAND gate is  $(1 \cdot 1)' = 0$ .

Thus, the cross feedback between the two NAND gates maintains the state — *Set* or *Reset* — of the latch.

**$S' = 0, R' = 1$ : Set.** If  $Q = 1$  and  $Q' = 0$ , the output of the upper NAND gate is  $(0 \cdot 0)' = 1$ , and the output of the lower NAND gate is  $(1 \cdot 1)' = 0$ . The latch remains in the *Set* state.

If  $Q = 0$  and  $Q' = 1$ , the output of the upper NAND gate is  $(0 \cdot 1)' = 1$ . This causes the output of the lower NAND gate to become  $(1 \cdot 1)' = 0$ . The feedback from the output of the lower NAND gate to the input of the upper keeps the output of the upper NAND gate at  $(0 \cdot 0)' = 1$ . The latch has moved into the *Set* state.

**$S' = 1, R' = 0$ : Reset.** If  $Q = 0$  and  $Q' = 1$ , the output of the lower NAND gate is  $(0 \cdot 0)' = 1$ , and the output of the upper NAND gate is  $(1 \cdot 1)' = 0$ . The latch remains in the *Reset* state.

If  $Q = 1$  and  $Q' = 0$ , the output of the lower NAND gate is  $(1 \cdot 0)' = 1$ . This is fed back to the input of the upper NAND gate to give  $(1 \cdot 1)' = 0$ . The feedback from the output of the upper NAND gate to the input of the lower keeps the output of the lower NAND gate at  $(0 \cdot 0)' = 1$ . The latch has moved into the *Reset* state.

**$S' = 0, R' = 0$ : Not allowed.** If  $Q = 0$  and  $Q' = 1$ , the output of the upper NAND gate is  $(0 \cdot 1)' = 1$ . This is fed back to the input of the lower NAND gate to give  $(1 \cdot 0)' = 1$  as its output. The feedback from the output of the lower NAND gate to the input of the upper maintains its output as  $(0 \cdot 0)' = 1$ . Thus,  $Q = Q' = 1$ , which is not allowed.

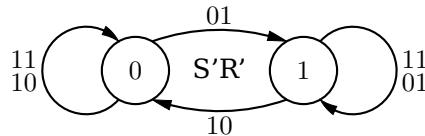
If  $Q = 1$  and  $Q' = 0$ , the output of the lower NAND gate is  $(1 \cdot 0)' = 1$ . This is fed back to the input of the upper NAND gate to give  $(0 \cdot 1)' = 1$  as its output. The feedback from the output of the upper NAND gate to the input of the lower maintains its output as  $(1 \cdot 1)' = 0$ . Thus,  $Q = Q' = 1$ , which is not allowed.

Figure 5.19 shows the behavior of a NAND-based S'R' latch. The inputs to a NAND-based S'R' latch are normally held at 1, which maintains the current state,  $Q$ . Its current state is available at the output. Momentarily changing  $S'$  or  $R'$  to 0 causes the state to change to *Set* or *Reset*, respectively, as shown in the “Next State” column.

Notice that placing 0 on both the *Set* and *Reset* inputs at the same time causes a problem. Then the outputs of both NOR gates would become 0. In other words,  $Q = Q' = 0$ , which is logically impossible. The circuit design must be such to prevent this input combination.

So the S'R' latch implemented with two NAND gates can be thought of as the complement of the NOR gate SR latch. The state is maintained by holding both  $S'$  and  $R'$  at 1.  $S' = 0$  causes the state to be 1 (Set), and  $R' = 0$  causes the state to be 0 (Reset). Using  $S'$  and  $R'$  as the activating signals are usually called *active-low* signals.

You have already seen that ones and zeros are represented by either a high or low voltage in electronic logic circuits. A given logic device may be activated by combinations



$S'$	$R'$	Current State	Next State
1	1	0	0
1	1	1	1
1	0	0	0
1	0	1	0
0	1	0	1
0	1	1	1
0	0	0	X
0	0	1	X

Figure 5.19: State table and state diagram for an  $S'R'$  latch. There are two possible inputs, 11 or 10, that cause the latch to remain in state 0. Similarly, 11 or 01 cause it to remain in state 1. Since the output is simply the state, it is not shown in this state diagram. Notice that the input 00 is not allowed, so it is not shown on the diagram.

of the two voltages. To show which is used to cause activation at any given input, the following definitions are used:

**active-high signal:** The *higher* voltage represents 1.

**active-low signal:** The *lower* voltage represents 1.

*Warning!* The definitions of active-high versus active-low signals vary in the literature. Make sure that you and the people you are working with have a clear agreement on the definitions you are using.

An active-high signal can be connected to an active-low input, but the hardware designer must take the difference into account. For example, say that the required logical input is 1 to an active-low input. Since it is active-low, that means the required voltage is the lower of the two. If the signal to be connected to this input is active-high, then a logical 1 is the higher of the two voltages. So this signal must first be complemented in order to be interpreted as a 1 at the active-low input.

We can get better control over the SR latch by adding two NAND gates to provide a *Control* input, as shown in Figure 5.20. In this circuit the outputs of both the control

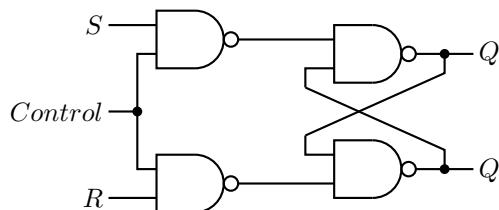


Figure 5.20: SR latch with *Control* input.

NAND gates remain at 1 as long as  $\text{Control} = 0$ . Table 5.4 shows the state behavior of the SR latch with control.

<i>Control</i>	<i>S</i>	<i>R</i>	Current State	Next State
0	—	—	0	0
0	—	—	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	X

Table 5.4: SR latch with Control state table. “—” indicates that the value does not matter. “X” indicates an indeterminate state. A circuit using this latch must be designed to prevent this input combination.

It is clearly better if we could find a design that eliminates the possibility of the “not allowed” inputs. Table 5.5 is a state table for a *D latch*. It has two inputs, one for control, the other for data, D. D = 1 sets the latch to 1, and D = 0 resets it to 0.

<i>Control</i>	<i>D</i>	Current State	Next State
0	—	0	0
0	—	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Table 5.5: D latch with Control state table. “—” indicates that the value does not matter.

The D latch can be implemented as shown in Figure 5.21. The one data input, D, is

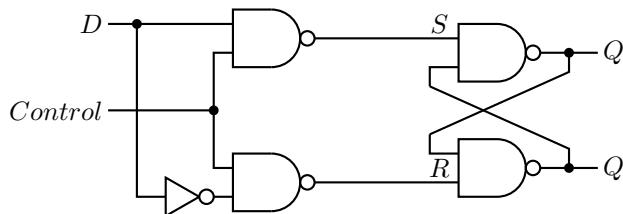


Figure 5.21: D latch constructed from an SR latch.

fed to the “S” side of the SR latch; the complement of the data value is fed to the “R” side.

Now we have a circuit that can store one bit of data, using the *D* input, and can be synchronized with a clock signal, using the *Control* input. Although this circuit is reliable by itself, the issue is whether it is reliable when connected with other circuit elements. The *D* signal almost certainly comes from an interconnection of combinational and sequential logic circuits. If it changes while the *Control* is still 1, the state of the latch will be changed.

Each electronic element in a circuit takes time to activate. It is a very short period of time, but it can vary slightly depending upon precisely how the other logic elements are interconnected and the state of each of them when they are activated. The problem here is that the *Control* input is being used to control the circuit based on the clock signal *level*. The clock level must be maintained for a time long enough to allow all the circuit elements to complete their activity, which can vary depending on what actions are being performed. In essence, the circuit timing is determined by the circuit elements and their actions instead of the clock. This makes it very difficult to achieve a reliable design.

It is much easier to design reliable circuits if the time when an activity can be triggered is made very short. The solution is to use edge-triggered logic elements. The inputs are applied and enough time is allowed for the electronics to settle. Then the next clock *transition* activates the circuit element. This scheme provides concise timing under control of the clock instead of timing determined more or less by the particular circuit design.

### 5.3.3 Flip-Flops

Although the terminology varies somewhat in the literature, it is generally agreed that (see Figure 5.15.):

- A latch uses a level based clock signal.
- A flip-flop is triggered by a clock signal edge.

At each “tick” of the clock, there are four possible actions that might be taken on a single bit — store 0, store 1, complement the bit (also called *toggle*), or leave it as is.

A *D flip-flop* is a common device for storing a single bit. We can turn the D latch into a D flip-flop by using two D latches connected in a *master/slave* configuration as shown in Figure 5.22. Let us walk through the operation of this circuit.

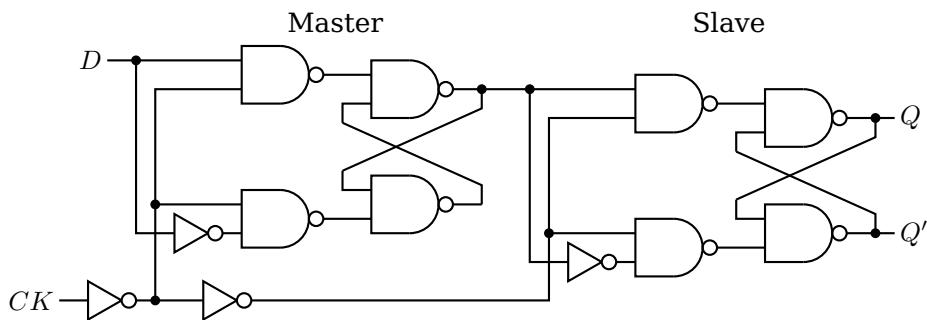


Figure 5.22: D flip-flop, positive-edge triggering.

The bit to be stored, 0 or 1, is applied to the *D* input of the Master D latch. The clock signal is applied to the *CK* input. It is normally 0. When the clock signal makes a transition from 0 to 1, the Master D latch will either Reset or Set, following the *D* input of 0 or 1, respectively.

While the *CK* input is at the 1 level, the control signal to the Slave D latch is 1, which deactivates this latch. Meanwhile, the output of this flip-flop, the output of the Slave D latch, is probably connected to the input of another circuit, which is activated by the same *CK*. Since the state of the Slave does not change during this clock half-cycle, the second circuit has enough time to read the current state of the flip-flop connected to its input. Also during this clock half-cycle, the state of the Master D latch has ample time to settle.

When the  $CK$  input transitions back to the 0 level, the control signal to the Master D latch becomes 1, deactivating it. At the same time, the control input to the Slave D latch goes to 0, thus activating the Slave D latch to store the appropriate value, 0 or 1. The new input will be applied to the Slave D latch during the second clock half-cycle, after the circuit connected to its output has had sufficient time to read its previous state. Thus, signals travel along a path of logic circuits in lock step with a clock signal.

There are applications where a flip-flop must be set to a known value before the clocking begins. Figure 5.23 shows a D flip-flop with an asynchronous preset input added to it. When a 1 is applied to the  $PR$  input,  $Q$  becomes 1 and  $Q'$  0, regardless of

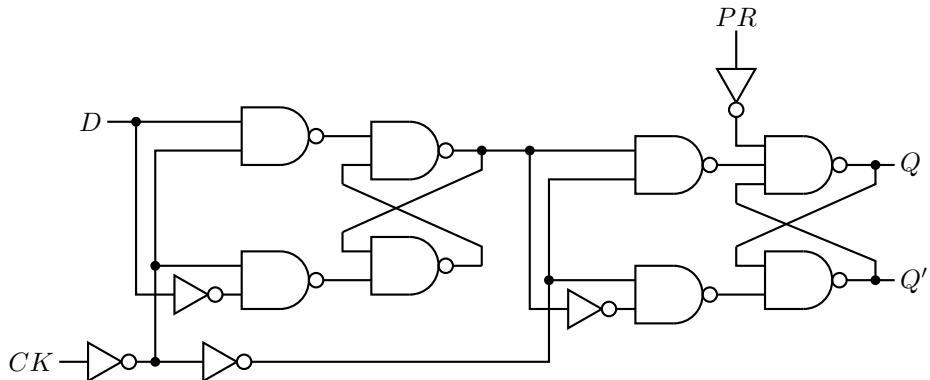


Figure 5.23: D flip-flop, positive-edge triggering with asynchronous preset.

what the other inputs are, even  $CLK$ . It is also common to have an asynchronous clear input that sets the state (and output) to 0.

There are more efficient circuits for implementing edge-triggered D flip-flops, but this discussion serves to show that they can be constructed from ordinary logic gates. They are economical and efficient, so are widely used in very large scale integration circuits. Rather than draw the details for each D flip-flop, circuit designers use the symbols shown in Figure 5.24. The various inputs and outputs are labeled in this figure.

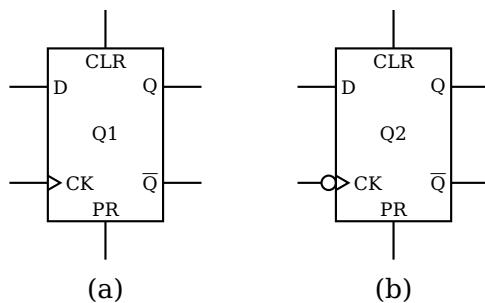
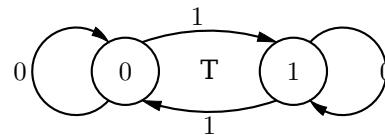


Figure 5.24: Symbols for D flip-flops. Includes asynchronous clear (CLR) and preset (PR). (a) Positive-edge triggering; (b) Negative-edge triggering.

Hardware designers typically use  $\bar{Q}$  instead of  $Q'$ . It is common to label the circuit as “ $Q_n$ ,” with  $n = 1, 2, \dots$  for identification. The small circle at the clock input in Figure 5.24(b) means that this D flip-flop is triggered by a negative-going clock transition. The D flip-flop circuit in Figure 5.22 can be changed to a negative-going trigger by simply removing the first NOT gate at the  $CK$  input.

The flip-flop that simply complements its state, a *T flip-flop*, is easily constructed from a D flip-flop. The state table and state diagram for a T flip-flop are shown in Figure

5.25.



$T$	Current State	Next State
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5.25: T flip-flop state table and state diagram. Each clock tick causes a state transition, with the next state depending on the current state and the value of the input,  $T$ .

To determine the value that must be presented to the D flip-flop in order to implement a T flip-flop, we add a column for D to the state table as shown in Table 5.6. By simply

$T$	Current State	Next State	D
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

Table 5.6: T flip-flop state table showing the D flip-flop input required to place the T flip-flop in the next state.

looking in the “Next State” column we can see what the input to the D flip-flop must be in order to obtain the correct state. These values are entered in the D column. (We will generalize this design procedure in Section 5.4.)

From Table 5.6 it is easy to write the equation for D:

$$\begin{aligned} D &= T' \cdot Q + T \cdot Q' \\ &= T \oplus Q \end{aligned} \tag{5.16}$$

The resulting design for the T flip-flop is shown in Figure 5.26.

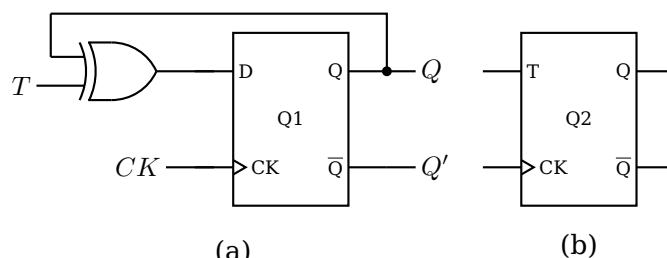
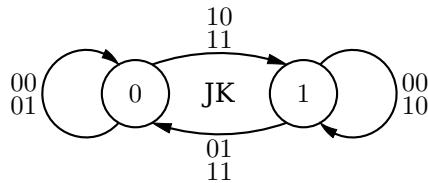


Figure 5.26: T flip-flop. (a) Circuit using a D flip-flop. (b) Symbol for a T flip-flop.

Implementing all four possible actions — set, reset, keep, toggle — requires two inputs,  $J$  and  $K$ , which leads us to the *JK flip-flop*. The state table and state diagram for a JK flip-flop are shown in Figure 5.27.



$J$	$K$	Current State	Next State
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Figure 5.27: JK flip-flop state table and state diagram.

In order to determine the value that must be presented to the D flip-flop we add a column for D to the state table as shown in Table 5.7. shows what values must be input

$J$	$K$	Current State	Next State	D
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

Table 5.7: JK flip-flop state table showing the D flip-flop input required to place the JK flip-flop in the next state.

to the D flip-flop. From this it is easy to write the equation for D:

$$\begin{aligned}
 D &= J' \cdot K' \cdot Q + J \cdot K' \cdot Q' + J \cdot K' \cdot Q + J \cdot K \cdot Q' \\
 &= J \cdot Q' \cdot (K' + K) + K' \cdot Q \cdot (J + J') \\
 &= J \cdot Q' + K' \cdot Q
 \end{aligned} \tag{5.17}$$

Thus, a JK flip-flop can be constructed from a D flip-flop as shown in Figure 5.28.

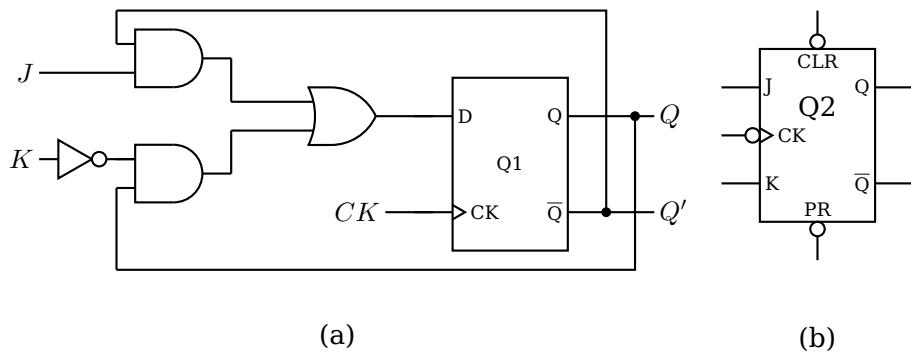


Figure 5.28: JK flip-flop. (a) Circuit using a D flip-flop. (b) Symbol for a JK flip-flop with asynchronous CLR and PR inputs.

## 5.4 Designing Sequential Circuits

We will now consider a more general set of steps for designing sequential circuits.<sup>1</sup> Design in any field is usually an iterative process, as you have no doubt learned from your programming experience. You start with a design, analyze it, and then refine the design to make it faster, less expensive, etc. After gaining some experience, the design process usually requires fewer iterations.

The following steps form a good method for a first working design:

1. From the word description of the problem, create a state table and/or state diagram showing what the circuit must do. These form the basic technical specifications for the circuit you will be designing.
2. Choose a binary code for the states, and create a binary-coded version of the state table and/or state diagram. For  $N$  states, the code will need  $\log_2 N$  bits. Any code will work, but some codes may lead to simpler combinational logic in the circuit.
3. Choose a particular type of flip-flop. This choice is often dictated by the components you have on hand.
4. Add columns to the state table that show the input required to each flip-flop in order to effect each transition that is required.
5. Simplify the input(s) to each flip-flop. Karnaugh maps or algebraic methods are good tools for the simplification process.
6. Draw the circuit.

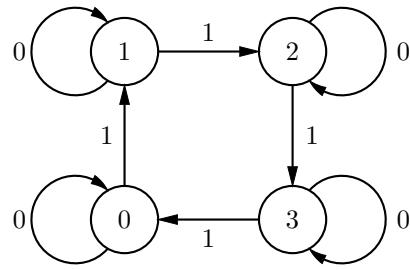
### Example 5-a

Design a counter that has an *Enable* input. When *Enable* = 1 it increments through the sequence 0, 1, 2, 3, 0, 1, ... with each clock tick. *Enable* = 0 causes the counter to remain in its current state.

*Solution:*

1. First we create a state table and state diagram:

<sup>1</sup>I wish to thank Dr. Lynn Stauffer for her valuable suggestions for this section.



Current $n$	$Enable = 0$	$Enable = 1$
	Next $n$	Next $n$
0	0	1
1	1	2
2	2	3
3	3	0

At each clock tick the counter increments by one if  $Enable = 1$ . If  $Enable = 0$  it remains in the current state. We have only shown the inputs because the output is equal to the state.

2. A reasonable choice is to use the binary numbering system for each state. With four states we need two bits. We will let  $n = n_1n_0$ , giving the state table:

Current	$Enable = 0$		$Enable = 1$	
	$n_1$	$n_0$	$n_1$	$n_0$
0 0	0	0	0	1
0 1	0	1	1	0
1 0	1	0	1	1
1 1	1	1	0	0

3. Since JK flip-flops are very general we will use those.
4. We need two flip-flops, one for each bit. So we add columns to the state table showing the input required to each JK flip-flop to cause the correct state transition. Referring to Figure 5.27 (page 114), we see that  $JK = 00$  keeps the current state,  $JK = 01$  resets it (to 0),  $JK = 10$  sets it (to 1), and  $JK = 11$  complements the state. We use  $X$  when the input can be either 0 or 1.

Current $n_1$ $n_0$	$Enable = 0$						$Enable = 1$					
	$n_1$	$n_0$	$J_1$	$K_1$	$J_0$	$K_0$	$n_1$	$n_0$	$J_1$	$K_1$	$J_0$	$K_0$
0 0	0	0	0	$X$	0	$X$	0	1	0	$X$	1	$X$
0 1	0	1	0	$X$	$X$	0	1	0	1	$X$	$X$	1
1 0	1	0	$X$	0	0	$X$	1	1	$X$	0	1	$X$
1 1	1	1	$X$	0	$X$	0	0	0	$X$	1	$X$	1

Notice the “don’t care” entries in the state table. Since the JK flip-flop is so versatile, including the “don’t cares” helps find simpler circuit realizations. (See Exercise 5-3.)

5. We use Karnaugh maps, using  $E$  for  $Enable$ .

$J_0(E, n_1, n_0)$		$n_1 n_0$				$K_0(E, n_1, n_0)$		$n_1 n_0$			
		00	01	11	10			00	01	11	10
$E$	0	X	X			$E$	0	X			X
	1	(1)	X	X	1		1	(X)	1	1	(X)

$J_1(E, n_1, n_0)$		$n_1 n_0$				$K_1(E, n_1, n_0)$		$n_1 n_0$			
		00	01	11	10			00	01	11	10
$E$	0		X	X		$E$	0	X	X		
	1		(1)	(X)	X		1	(X)	(1)		

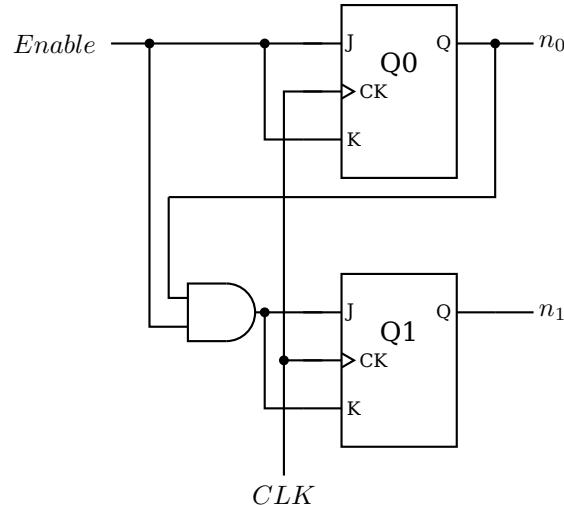
$$J_0(E, n_1, n_0) = E$$

$$K_0(E, n_1, n_0) = E$$

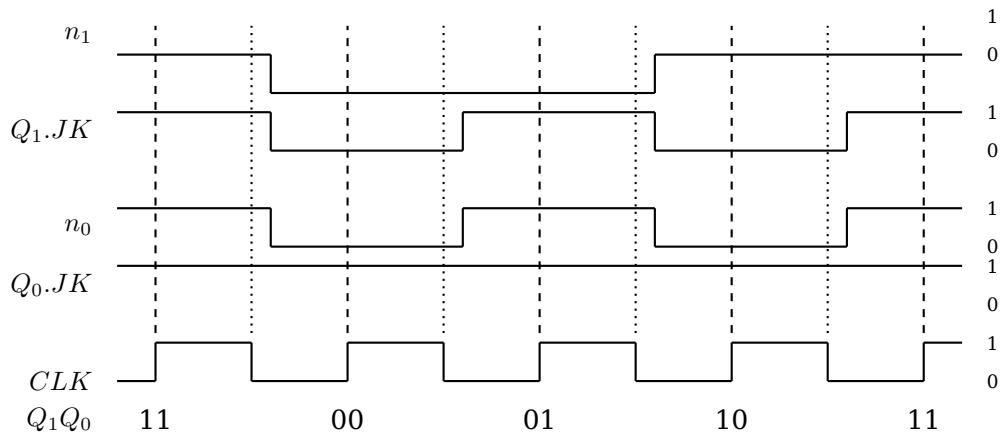
$$J_1(E, n_1, n_0) = E \cdot n_0$$

$$K_1(E, n_1, n_0) = E \cdot n_0$$

6. The circuit to implement this counter is:



The timing of the binary counter is shown here when counting through the sequence 3, 0, 1, 2, 3 (11, 00, 01, 10, 11).



$Q_{i.JK}$  is the input to the  $i^{th}$  JK flip-flop, and  $n_i$  is its output. (Recall that  $J = K$  in this design.) When the  $i^{th}$  input,  $Q_{i.JK}$ , is applied to its JK flip-flop, remember that the state of the flip-flop does not change until the second half of the clock cycle. This can be seen when comparing the trace for the corresponding output,  $n_i$ , in the figure.

Note the short delay after a clock transition before the value of each  $n_i$  actually changes. This represents the time required for the electronics to completely settle to the new values. □

Except for very inexpensive microcontrollers, most modern CPUs execute instructions in stages. An instruction passes through each stage in an assembly-line fashion, called a *pipeline*. The action of the first stage is to fetch the instruction from memory, as will be explained in Chapter 6.

After an instruction is fetched from memory, it passes onto the next stage. Simultaneously, the first stage of the CPU fetches the next instruction from memory. The result is that the CPU is working on several instructions at the same time. This provides some parallelism, thus improving execution speed.

Almost all programs contain conditional branch points — places where the next instruction to be fetched can be in one of two different memory locations. Unfortunately, the decision of which of the two instructions to fetch is not known until the decision-making instruction has moved several stages into the pipeline. In order to maintain execution speed, as soon as a conditional branch instruction has passed on from the fetch stage, the CPU needs to predict where to fetch the next instruction from.

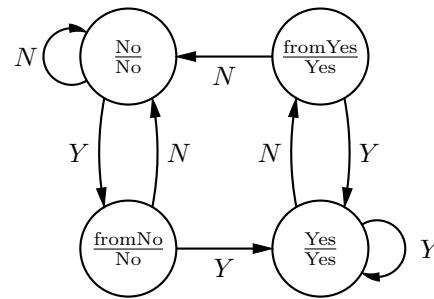
In this next example we will design a circuit to implement a prediction circuit.

### Example 5-b

Design a circuit that predicts whether a conditional branch is taken or not. The predictor continues to predict the same outcome, take the branch or do not take the branch, until it makes two mistakes in a row.

*Solution:*

1. We use “Yes” to indicate when the branch is taken and “No” to indicate when it is not. The state diagram shows four states:



Let us begin in the “No” state. The prediction is that the next branch will also not be taken. The notation in the state bubbles is  $\frac{\text{state}}{\text{output}}$ , showing that the output in this state is also “No.”

The input to the circuit is whether or not the branch was actually taken. The arc labeled “N” shows the transition when the branch was not taken. It loops back to the “No” state, with the prediction (the output) that the branch will not be taken the next time. If the branch is taken, the “Y” arc shows that the circuit moves into the “fromNo” state, but still predicting no branch the next time.

From the “fromNo” state, if the branch is not taken (the prediction is correct), the circuit returns to the “No” state. However, if the branch is taken, the “Y” shows that the circuit moves into the “Yes” state. This means that the circuit predicted incorrectly twice in a row, so the prediction is changed to “Yes.”

You should be able to follow this state diagram for the other cases and convince yourself that both the “fromNo” and “fromYes” states are required.

2. Next we look at the state table:

Current		$Taken = No$		$Taken = Yes$	
State	Prediction	State	Next Prediction	State	Next Prediction
No	No	No	No	fromNo	No
fromNo	No	No	No	Yes	Yes
fromYes	Yes	No	No	Yes	Yes
Yes	Yes	fromYes	Yes	Yes	Yes

Since there are four states, we need two bits. We will let 0 represent “No” and 1 represent “Yes.” The input is whether the branch is actually taken (1) or not (0). And the output is the prediction of whether it will be taken (1) or not (0).

We choose a binary code for the state,  $s_1 s_0$ , such that the high-order bit represents the prediction, and the low-order bit what the last input was. That is:

State	Prediction	$s_1$	$s_0$
No	No	0	0
fromNo	No	0	1
fromYes	Yes	1	0
Yes	Yes	1	1

This leads to the state table in binary:

		Input = 0		Input = 1	
Current		Next		Next	
$s_1$	$s_0$	$s_1$	$s_0$	$s_1$	$s_0$
0	0	0	0	0	1
0	1	0	0	1	1
1	0	0	0	1	1
1	1	1	0	1	1

3. We will use JK flip-flops for the circuit.

4. Next we add columns to the binary state table showing the JK inputs required in order to cause the correct state transitions.

		Input = 0					Input = 1								
Current		Next			$J_1$	$K_1$	$J_0$	$K_0$	Next			$J_1$	$K_1$	$J_0$	$K_0$
$s_1$	$s_0$	$s_1$	$s_0$	$J_1$	$K_1$	$J_0$	$K_0$	$s_1$	$s_0$	$J_1$	$K_1$	$J_0$	$K_0$		
0	0	0	0	0	$X$	0	$X$	0	1	0	$X$	1	$X$	0	
0	1	0	0	0	$X$	$X$	1	1	1	1	$X$	$X$	0	0	
1	0	0	0	$X$	1	0	$X$	1	1	1	$X$	0	1	$X$	
1	1	1	0	$X$	0	$X$	1	1	1	$X$	0	$X$	0	0	

5. We use Karnaugh maps to derive equations for the JK flip-flop inputs.

		$s_1 s_0$			
		00	01	11	10
$In$	0		$X$	$X$	
	1	(1)	$X$	$X$	(1)

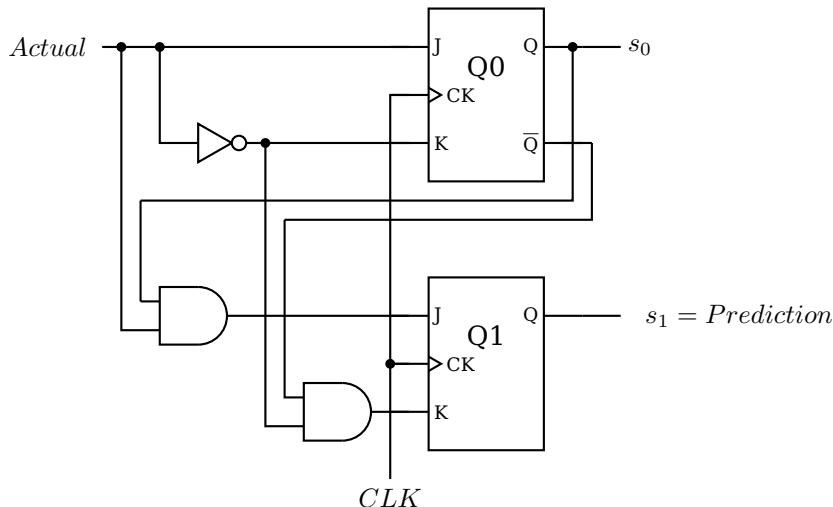
		$s_1 s_0$			
		00	01	11	10
$In$	0	( $X$ )	1	1	( $X$ )
	1	$X$			$X$

		$s_1 s_0$			
		00	01	11	10
$In$	0			$X$	$X$
	1	(1)	( $X$ )	$X$	(1)

		$s_1 s_0$			
		00	01	11	10
$In$	0	( $X$ )	$X$		(1)
	1	$X$	$X$		

$$\begin{aligned}
 J_0(In, s_1, s_0) &= In \\
 K_0(In, s_1, s_0) &= In' \\
 J_1(In, s_1, s_0) &= In \cdot s_0 \\
 K_1(In, s_1, s_0) &= In' \cdot s_0'
 \end{aligned}$$

6. The circuit to implement this predictor is:



## 5.5 Memory Organization

In this section we will discuss how registers, SRAM, and DRAM are organized and constructed. Keeping with the intent of this book, the discussion will be introductory only.

### 5.5.1 Registers

Registers are used in places where small amounts of very fast memory is required. Many are found in the CPU where they are used for numerical computations, temporary data storage, etc. They are also used in the hardware that serves to interface between the CPU and other devices in the computer system.

We begin with a simple 4-bit register, which allows us to store four bits. Figure 5.29 shows a design for implementing a 4-bit register using D flip-flops. As described above, each time the clock cycles the state of each of the D flip-flops is set according to the value of  $d = d_3d_2d_1d_0$ . The problem with this circuit is that any changes in any of the  $d_i$ 's will change the state of the corresponding bit in the next clock cycle, so the contents of the register are essentially valid for only one clock cycle.

One-cycle buffering of a bit pattern is sufficient for some applications, but there is also a need for registers that will store a value until it is explicitly changed, perhaps billions of clock cycles later. The circuit in Figure 5.30 uses adds a *load* signal and feedback from the output of each bit. When *load* = 1 each bit is set according to its corresponding input,  $d_i$ . When *load* = 0 the output of each bit,  $r_i$ , is used as the input, giving no change. So this register can be used to store a value for as many clock cycles as desired. The value will not be changed until *load* is set to 1.

Most computers need many general purpose registers. When two or more registers are grouped together, the unit is called a *register file*. A mechanism must be provided for addressing one of the registers in the register file.

Consider a register file composed of eight 4-bit registers,  $r0 - r7$ . We could build eight copies of the circuit shown in Figure 5.30. Let the 4-bit data input,  $d$ , be connected in parallel to all of the corresponding data pins,  $d_3d_2d_1d_0$ , of each of the eight registers.

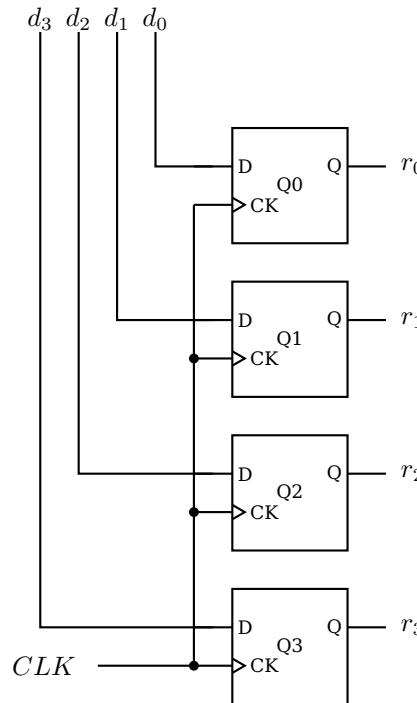


Figure 5.29: A 4-bit register. A D flip-flop is used to hold each bit. The state of the  $i^{th}$  bit is set by the value of  $d_i$  at each clock tick. The 4-bit value stored in the register is  $r = r_3r_2r_1r_0$ .

Three bits are required to address one of the registers ( $2^3 = 8$ ). If the 8-bit output from a  $3 \times 8$  decoder is connected to the eight *load* inputs of each of the registers,  $d$  will be loaded into one, and only one, of the registers during the next clock cycle. All the other registers will have *load* = 0, and they will simply maintain their current state. Selecting the output from one of the eight registers can be done with four 8-input multiplexers. One such multiplexer is shown in Figure 5.31. The inputs  $r_{0i} - r_{7i}$  are the  $i^{th}$  bits from each of eight registers,  $r_0 - r_7$ . One of the eight registers is selected for the 1-bit output,  $Reg\_Out_i$ , by the 3-bit input  $Reg\_Sel$ . Keep in mind that four of these output circuits would be required for 4-bit registers. The same  $Reg\_Sel$  would be applied to all four multiplexers simultaneously in order to output all four bits of the same register. Larger registers would, of course, require correspondingly more multiplexers.

There is another important feature of this design that follows from the master/slave property of the D flip-flops. The state of the slave portion does not change until the second half of the clock cycle. So the circuit connected to the output of this register can read the current state during the first half of the clock cycle, while the master portion is preparing to change the state to the new contents.

## 5.5.2 Shift Registers

There are many situations where it is desirable to shift a group of bits. A *shift register* is a common device for doing this. Common applications include:

- Inserting a time delay in a bit stream.
- Converting a serial bit stream to a parallel group of bits.

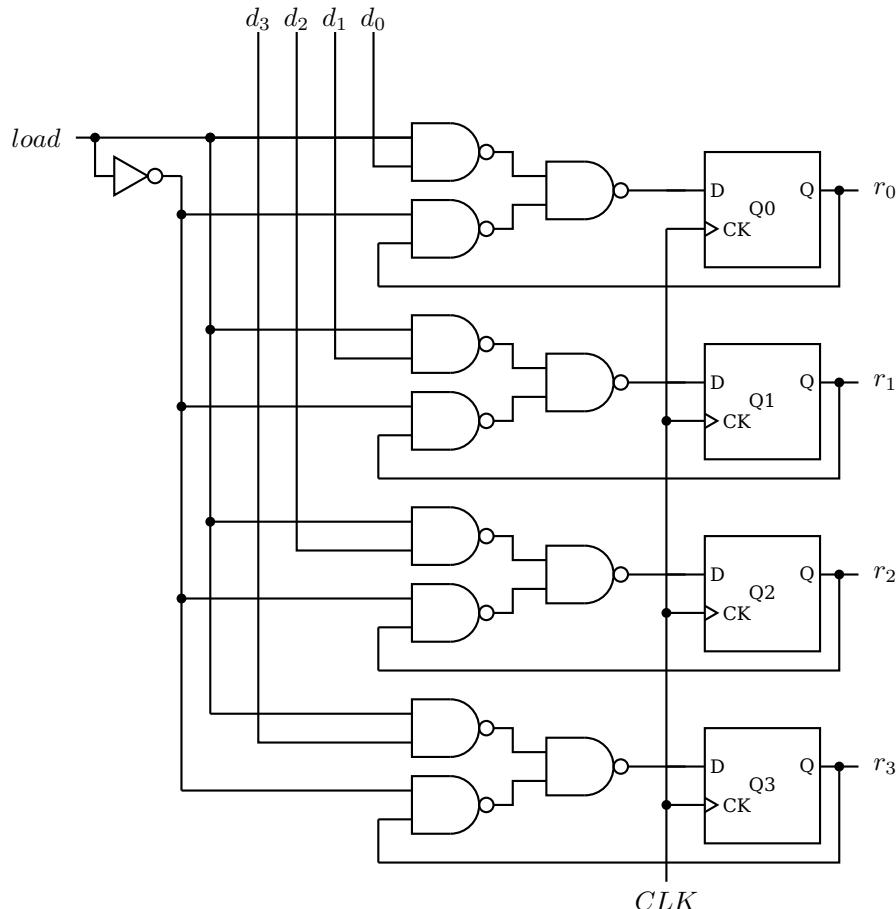


Figure 5.30: A 4-bit register with load. The storage portion is the same as in Figure 5.29. When  $load = 1$  each bit is set according to its corresponding input,  $d_i$ . When  $load = 0$  the output of each bit,  $r_i$ , is used as the input, giving no change.

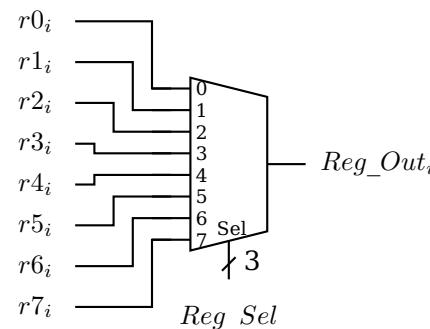


Figure 5.31: 8-way mux to select output of register file. This only shows the output of the  $i^{th}$  bit.  $n$  are required for  $n$ -bit registers.  $Reg\_Sel$  is a 3-bit signal that selects one of the eight inputs.

- Converting a parallel group of bits into a serial bit stream.

- Shifting a parallel group of bits left or right to perform multiplication or division by powers of 2.

Serial-to-parallel and parallel-to-serial conversion is required in I/O controllers because most I/O communication is serial bit streams, while data processing in the CPU is performed on groups of bits in parallel.

A simple 4-bit serial-to-parallel shift register is shown in Figure 5.32. A serial stream

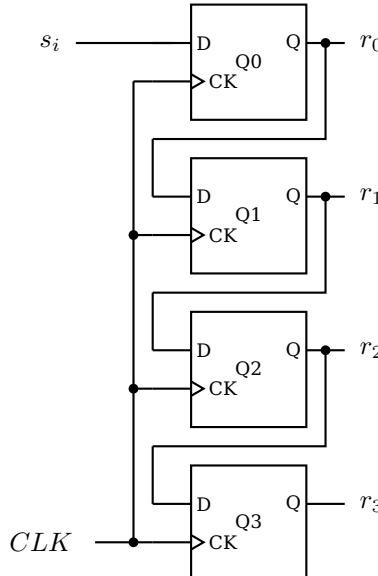


Figure 5.32: Four-bit serial-to-parallel shift register. A D flip-flop is used to hold each bit. Bits arrive at the input,  $s_i$ , one at a time. The last four input bits are available in parallel at  $r_3 - r_0$ .

of bits is input at  $s_i$ . At each clock tick, the output of  $Q_0$  is applied to the input of  $Q_1$ , thus copying the previous value of  $r_0$  to the new  $r_1$ . The state of  $Q_0$  changes to the value of the new  $s_i$ , thus copying this to be the new value of  $r_0$ . The serial stream of bits continues to ripple through the four bits of the shift register. At any time, the last four bits in the serial stream are available in parallel at the four outputs,  $r_3, \dots, r_0$ .

The same circuit could be used to provide a time delay of four clock ticks in a serial bit stream. Simply use  $r_3$  as the serial output.

### 5.5.3 Static Random Access Memory (SRAM)

There are several problems with trying to extend this design to large memory systems. First, although a multiplexer works for selecting the output from several registers, one that selects from a many million memory cells is simply too large. From Figure 5.9 (page 100), we see that such a multiplexer would need an AND gate for each memory cell, plus an OR gate with an input for each of these millions of AND gate outputs.

We need another logic element called a *tri-state buffer*. The tri-state buffer has three possible outputs — 0, 1, and “high Z.” “High Z” describes a very high impedance connection (see Section 4.4.2, page 78.) It can be thought of as essentially “no connection” or “open.”

It takes two inputs — data input and enable. The truth table describing a tri-state buffer is:

Enable	In	Out
0	0	highZ
0	1	highZ
1	0	0
1	1	1

and its circuit symbol is shown in Figure 5.33. When  $Enable = 1$  the output, which is

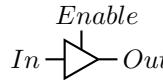


Figure 5.33: Tri-state buffer.

equal to the input, is connected to whatever circuit element follows the tri-state buffer. But when  $Enable = 0$ , the output is essentially disconnected. Be careful to realize that this is different from 0; being disconnected means it has no effect on the circuit element to which it is connected.

A 4-way multiplexer using a  $2 \times 4$  decoder and four tri-state buffers is illustrated in Figure 5.34. Compare this design with the 4-way multiplexer shown in Figure 5.9, page

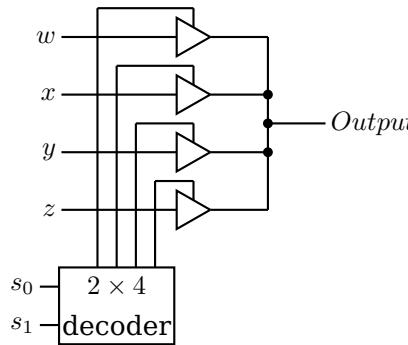


Figure 5.34: Four way multiplexer built from tri-state buffers.  $Output = w, x, y,$  or  $z$ , depending on which one is selected by  $s_1s_0$  fed into the decoder. Compare with Figure 5.9, page 100.

100. The tri-state buffer design may not be an advantage for small multiplexers. But an  $n$ -way multiplexer without tri-state buffers requires an  $n$ -input OR gate, which presents some technical electronic problems.

Figure 5.35 shows how tri-state buffers can be used to implement a single memory cell. This circuit shows only one 4-bit memory cell so you can compare it with the register design in Figure 5.29, but it scales to much larger memories.  $Write$  is asserted to store data in the D flip-flops.  $Read$  enables the output tri-state buffer in order to connect the single output line to  $Mem\_data\_out$ . The address decoder is also used to enable the tri-state buffers to connect a memory cell to the output,  $r_3r_2r_1r_0$ .

This type of memory is called *Static Random Access Memory (SRAM)*. “Static” because the memory retains its stored values as long as power to the circuit is maintained. “Random access” because it takes the same length of time to access the memory at any address.

A 1 MB memory requires a 20 bit address. This requires a  $20 \times 2^{20}$  address decoder as shown in Figure 5.36. Recall from Section 5.1.3 (page 95) that an  $n \times 2^n$  decoder requires  $2^n$  AND gates. We can simplify the circuitry by organizing memory into a grid of rows and columns as shown in Figure 5.37. Although two decoders are required, each

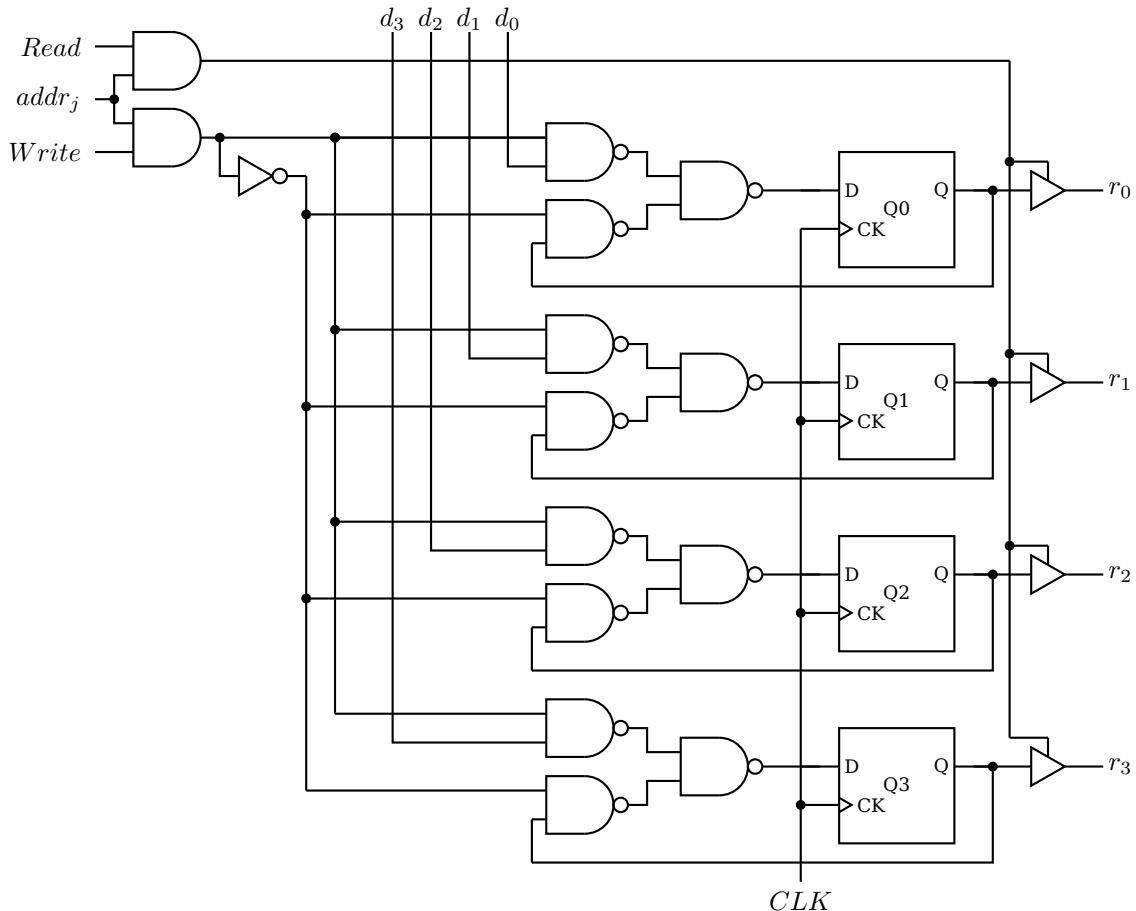


Figure 5.35: 4-bit memory cell. Each bit is output through a tri-state buffer.  $addr_j$  is one output from a decoder corresponding to an address.

requires  $2^{n/2}$  AND gates, for a total of  $2 \times 2^{n/2} = 2^{(n/2)+1}$  AND gates for the decoders. Of course, memory cell access is slightly more complex, and some complexity is added in order to split the 20-bit address into two 10-bit portions.

#### 5.5.4 Dynamic Random Access Memory (DRAM)

Each bit in SRAM requires about six transistors for its implementation. A less expensive solution is found in *Dynamic Random Access Memory (DRAM)*. In DRAM each bit value is stored by charging a capacitor to one of two voltages. The circuit requires only one transistor to charge the capacitor, as shown in Figure 5.38. This Figure shows only four bits in a single row.

When the “Row Address Select” line is asserted all the transistors in that row are turned on, thus connecting the respective capacitor to the Data Latch. The value stored in the capacitor, high voltage or low voltage, is stored in the Data Latch. There, it is available to be read from the memory. Since this action tends to discharge the capacitors, they must be refreshed from the values stored in the Data Latch.

When new data is to be stored in DRAM, the current values are first stored in the Data Latch, just as in a read operation. Then the appropriate changes are made in the Data Latch before the capacitors are refreshed.

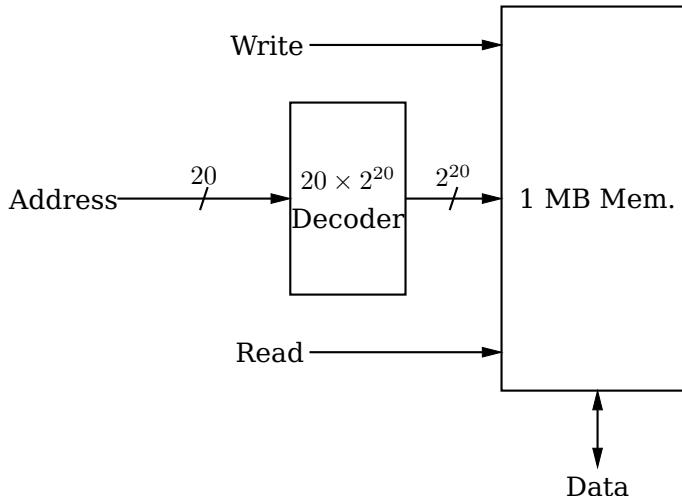


Figure 5.36: Addressing 1 MB of memory with one  $20 \times 2^{20}$  address decoder. The short line through the connector lines indicates the number of bits traveling in parallel in that connection.

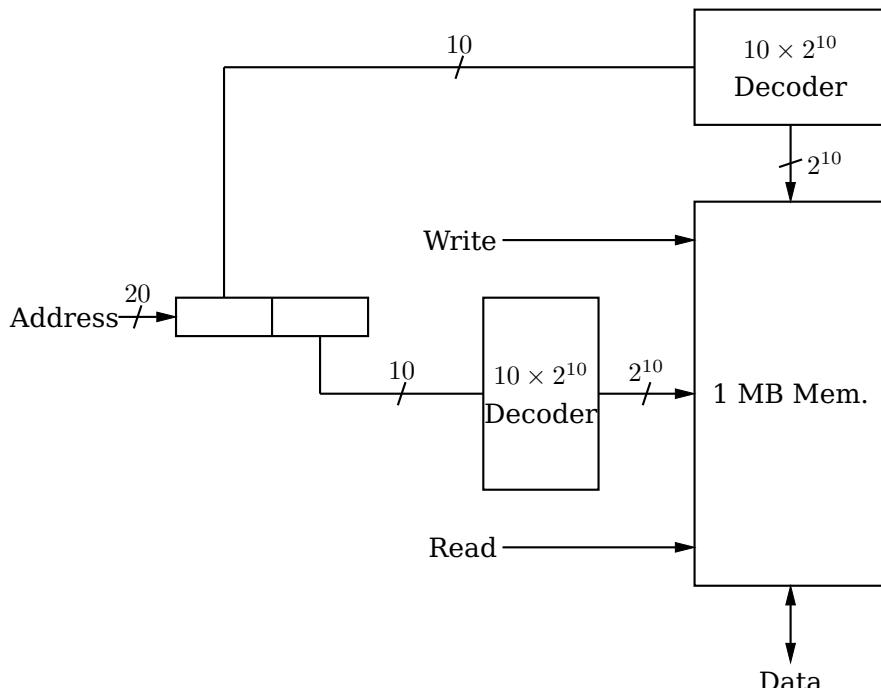


Figure 5.37: Addressing 1 MB of memory with two  $10 \times 2^{10}$  address decoders.

These operations take more time than simply switching flip-flops, so DRAM is appreciably slower than SRAM. In addition, capacitors lose their charge over time. So each row of capacitors must be read and refreshed in the order of every 60 msec. This requires additional circuitry and further slows memory access. But the much lower cost of DRAM compared to SRAM warrants the slower access time.

This has been only an introduction to how switching transistors can be connected

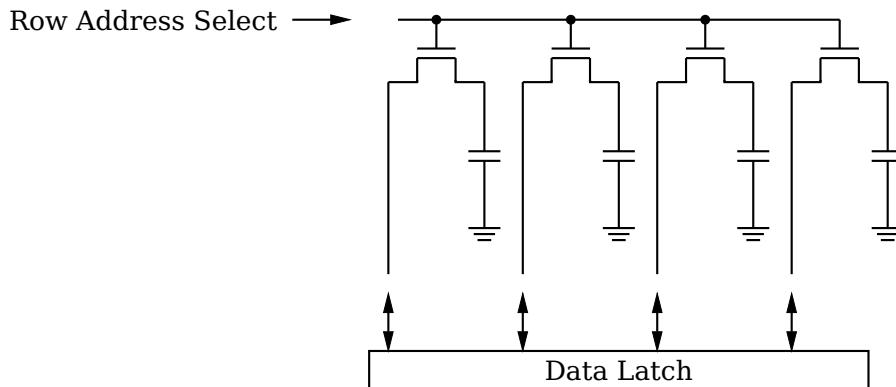


Figure 5.38: Bit storage in DRAM.

into circuits to create a CPU. We leave the details to more advanced books, e.g., [20], [23], [24], [28], [31], [34].

## 5.6 Exercises

The greatest benefit will be derived from these exercises if you either build the circuits with hardware or using a simulation program. Several free circuit simulation applications are available that run under GNU/Linux.

- 5-1** (§5.1) Build a four-bit adder.
- 5-2** (§5.1) Build a four-bit adder/subtractor.
- 5-3** (§5.4) Redesign the 2-bit counter of Example 5-a using only the “set” and “reset” inputs of the JK flip-flops. So your state table will not have any “don’t cares.”
- 5-4** (§5.4) Design a 4-bit up counter — 0, 1, 2,...,15, 0,...
- 5-5** (§5.4) Design a 4-bit down counter — 15, 14, 13,...,0, 15,...
- 5-6** (§5.4) Design a decimal counter — 0, 1, 2,...,9, 0,...
- 5-7** (§5.5) Build the register file described in Section 5.5.1. It has eight 4-bit registers. A  $3 \times 8$  decoder is used to select a register to be loaded. Four 8-way multiplexers are used to select the four bits from one register to be output.

# Chapter 6

# Central Processing Unit

In this chapter we move on to consider a programmer's view of the *Central Processing Unit (CPU)* and how it interacts with memory. X86-64 CPUs can be used with either a 32-bit or a 64-bit operating system. The CPU features available to the programmer depend on the operating mode of the CPU. The modes of interest to the applications programmer are summarized in Table 6.1. With a 32-bit operating system, the CPU behaves essentially the same as an x86-32 CPU.

Mode	Submode	Operating System	Default Address (bits)	Default int (bits)
IA-32e or Long	64-bit	64-bit	64	32
	Compatibility		32	
			16	16
Legacy	Protected	32-bit	32	32
	Virtual-8086		16	16
	Real	16-bit		

Table 6.1: X86-64 operating modes. Intel manuals use the term “IA-32e” and AMD manuals use “Long” when running a 64-bit operating system. Both manuals use the same terminology for the two sub-modes. Adapted from Table 1-1 in [2].

In this book we describe the view of the CPU when running a 64-bit operating system. Intel manuals call this the *IA-32e mode* and the AMD manuals call it the *long mode*. The CPU can run in one of two sub-modes under a 64-bit operating system. Both manuals use the same terminology for the two sub-modes.

- *Compatibility mode* – Most programs compiled for a 32-bit or 16-bit environment can be run without re-compiling.
- *64-bit mode* – The program must be compiled for 64-bit execution.

The two modes cannot be mixed in the same program.

The discussion in this chapter focuses on the 64-bit mode. We will also point out the differences of the compatibility mode, which we will refer to as the *32-bit mode*.

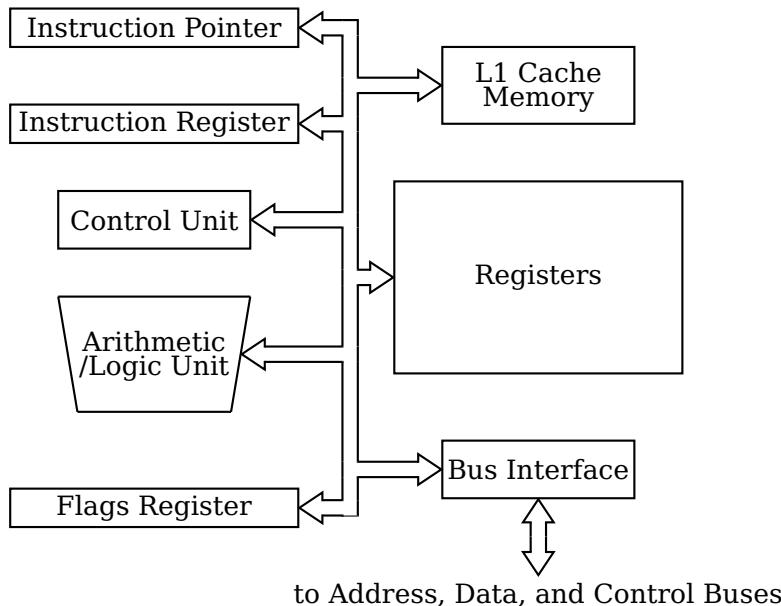


Figure 6.1: CPU block diagram. The CPU communicates with the Memory and I/O subsystems via the Address, Data, and Control buses. See Figure 1.1 (page 3).

## 6.1 CPU Overview

An overall block diagram of a typical CPU is shown in Figure 6.1. The subsystems are connected together through internal buses. Keep in mind that this is a highly simplified diagram. Actual CPUs are much more complicated, but the general concepts discussed in this chapter apply to all of them.

We will now describe briefly each of the subsystems in Figure 6.1. The descriptions provided here are generic and apply to most CPUs. Components that are of particular interest to a programmer are described within the context of the x86 ISA later in this chapter.

**Bus Interface:** This is the means for the CPU to communicate with the rest of the computer system — Memory and I/O Devices. It contains circuitry to place addresses on the address bus, read and write data on the data bus, and read and write signals on the control bus. The bus interface on many CPUs interfaces with external bus control units that in turn interface with memory and with different types of I/O buses, e.g., SATA, PCI-E, etc. The external control units are transparent to the programmer.

**L1 Cache Memory:** Although it could be argued that this is not a part of the CPU, most modern CPUs include very fast cache memory on the CPU chip. As you will see in Section 6.4, each instruction must be fetched from memory. The CPU can execute instructions much faster than they can be fetched. The interface with memory makes it more efficient to fetch several instructions at one time, storing them in L1 cache where the CPU has very fast access to them. Many modern CPUs use two L1 cache memories organized in a Harvard architecture — one for instructions, the other for data. (See Section 1.2, page 4.) Its use is generally transparent to an applications programmer.

**Registers:** A register is a group of bits that is intended to be used as a variable in a program. Compilers and assemblers have names for each register. Almost all arithmetic and logic operations and data movement operations involve at least one register. See Section 6.2 for more details.

**Instruction Pointer:** This is a 64-bit register that always contains the address of the next instruction to be executed. See Section 6.2 for more details.

**Instruction Register:** This register contains the instruction that is currently being executed. Its bit pattern determines what the Control Unit is causing the CPU to do. Once that action has been completed, the bit pattern in the instruction register can be changed, and the CPU will perform the operation specified by this next bit pattern.

Most modern CPUs use an instruction queue that is built into the chip. Several instructions are waiting in the queue, ready to be executed. Separate electronic circuitry keeps the instruction queue full while the regular control unit is executing the instructions. But this is simply an implementation detail that allows the control unit to run faster. The essence of how the control unit executes a program is represented by the single instruction register model.

**Control Unit:** The bits in the Instruction Register are decoded in the Control Unit. It generates the signals that control the other subsystems in the CPU to carry out the action(s) specified by the instruction. It is typically implemented as a finite-state machine and contains Decoders (Section 5.1.3), Multiplexers (Section 5.1.4), and other logic components.

**Arithmetic Logic Unit (ALU):** A device that performs arithmetic and logic operations on groups of bits. The logic circuitry to perform addition is discussed in Section 5.1.1.

**Flags Register:** Each operation performed by the ALU results in various conditions that must be recorded. For example, addition can produce a carry. One bit in the Flags Register will be set to either zero (no carry) or one (carry) after the ALU has completed any operation that may produce a carry.

We will now look at how the logic circuits discussed in Chapter 4 can be used to implement some of these subsystems.

## 6.2 CPU Registers

A portion of the memory in the CPU is organized into registers. Machine instructions access CPU registers by their addresses, just as memory contents are accessed. Of course, the register addresses are not placed on the address bus since the registers are in the CPU. The difference from a programmer's point of view is that the assembler has predefined names for the registers, whereas the programmer creates symbolic names for memory addresses. Thus in each program that you write in assembly language:

- CPU registers are accessed by using the names that are predefined in the assembler.
- Memory is accessed by the programmer providing a name for the memory location and using that name in the user program.

**Basic Programming Registers**

16	64-bit	General purpose ( <b>GPRs</b> )
1	64-bit	Flags
1	64-bit	Instruction pointer
6	16-bit	Segment

**Floating Point Registers**

8	80-bit	Floating point data
1	16-bit	Control
1	16-bit	Status
1	16-bit	Tag
1	11-bit	Opcode
1	64-bit	FPU Instruction Pointer
1	64-bit	FPU Data Pointer

**MMX Registers**

8	64-bit	MMX
---	--------	-----

**XMM Registers**

16	128-bit	XMM
1	32-bit	MXCSR

**Model-Specific Registers (MSRs)**

These vary depending on the specific hardware implementation. They are only accessible to the operating system.

Table 6.2: The x86-64 registers. Not all the registers shown here are discussed in this chapter. Some are discussed in subsequent chapters that deal with the related topic.

The x86-64 architecture registers are shown in Table 6.2. Each bit in each register is numbered from right to left, beginning with zero. So the right-most bit is number 0, the next one to the left number 1, etc. Since there are 64 bits in each register, the left-most bit is number 63.

The *general purpose registers* can be accessed in the following ways:

- Quadword — all 64 bits [63 – 0].
- Doubleword — the low-order 32 bits [31 – 0].
- Word — the low-order 16 bits [15 – 0].
- Byte — the low-order 8 bits [7 – 0] (and in four registers bits [15 – 8]).

The assembler uses a different name for each group of bits in a register. The assembler names for the groups of the bits are given in Table 6.3. In 64-bit mode, writing to an 8-bit or 16-bit portion of a register does not affect the other 56 or 48 bits in the register. However, when writing to the low-order 32 bits, the high-order 32 bits are set to zero.  
24mm

A pictorial representation of the naming of each portion of the general-purpose registers is shown in Figure 6.2.

<b>bits 63-0</b>	<b>bits 31-0</b>	<b>bits 15-0</b>	<b>bits 15-8</b>	<b>bits 7-0</b>
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	ch	cl	
rdx	edx	dx	dh	dl
rsi	esi	si		sil
rdi	edi	di		dil
rbp	ebp	bp		bpl
rsp	esp	sp		spl
r8	r8d	r8w		r8b
r9	r9d	r9w		r9b
r10	r10d	r10w		r10b
r11	r11d	r11w		r11b
r12	r12d	r12w		r12b
r13	r13d	r13w		r13b
r14	r14d	r14w		r14b
r15	r15d	r15w		r15b

Table 6.3: Assembly language names for portions of the general-purpose CPU registers. Programs running in 32-bit mode can only use the registers above the line in this table. 64-bit mode allows the use of all the registers. The ah, bh, ch, and dh registers cannot be used with any of the (8-bit) registers below the line.

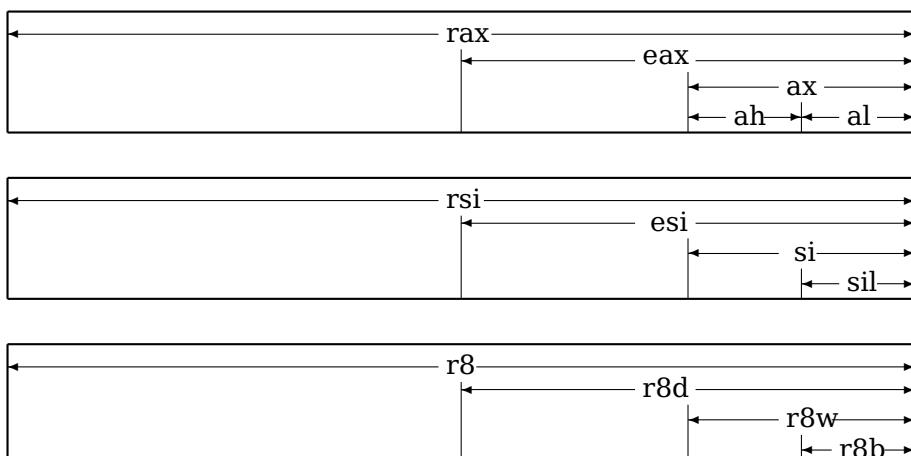


Figure 6.2: Graphical representation of general purpose registers. The three shown here are representative of the pattern of all the general purpose registers.

The 8-bit register portions ah, bh, ch, and dh are a holdover from the Intel® 8086/8088 architecture. It had four 16-bit registers, ax, bx, cx, and dx. The low-order bytes were named al, bl, cl, and dl and the high-order bytes named ah, bh, ch, and dh. Access to these registers has been maintained in 32-bit mode for backward compatibility but is limited in 64-bit mode. Access to the 8-bit low-order portions of the rsi, rdi, rsp, and rbp registers was added along with the move to 64 bits in the x86-64 architecture but cannot be used in the same instruction with the 8-bit register portions of the xh registers.

When using less than the entire 64 bits in a register, it is generally bad to write code that assumes the remaining portion is in any particular state. Such code is difficult to read and leads to errors during its maintenance phase.

Although these are called “general purpose,” the descriptions in Table 6.4 show that some of them have some special significance, depending upon how they are used. (Some of the descriptions may not make sense to you at this point.) In this book, we will use

<b>Register</b>	<b>Special usage</b>	<b>Called function preserves contents</b>
rax	1st function return value.	No
rbx	Optional base pointer.	Yes
rcx	Pass 4th argument to function.	No
rdx	Pass 3rd argument to function; 2nd function return value.	No
rsp	Stack pointer.	Yes
rbp	Optional frame pointer.	Yes
rdi	Pass 1st argument to function.	No
rsi	Pass 2nd argument to function.	No
r8	Pass 5th argument to function.	No
r9	Pass 6th argument to function.	No
r10	Pass function’s static chain pointer.	No
r11		No
r12		Yes
r13		Yes
r14		Yes
r15		Yes

Table 6.4: General purpose registers.

the rax, rdx, rdi, esi, and r8 – r15 registers for general-purpose storage. They will be used just like variables in a high-level language. Usage of the rsp and rbp registers follows a very strict discipline. You should not use either of them for your assembly language programs until you understand how to use them.

The *instruction pointer* register, rip<sup>1</sup>, always points to the next instruction to be executed. As explained in Section 6.4 (page 136), every time an instruction is fetched,

<sup>1</sup>In many other environments, the equivalent register is called the *program counter*.

the `rip` register is automatically incremented by the control unit to contain the address of the next instruction. Thus, the `rip` register is never directly accessed by the programmer. On the other hand, every instruction that is executed affects the contents of the `rip` register. Thus, the `rip` register is not a general-purpose register, but it guides the flow of the entire program.

Most arithmetic and logical operations affect the *condition codes* in the `rflags` register. The bits that are affected are shown in Figure 6.3.

11	10	9	8	7	6	5	4	3	2	1	0
OF				SF	ZF		AF		PF		CF

Figure 6.3: Condition codes portion of the `rflags` register. The high-order 32 bits (32 – 63) are reserved for other use and are not shown here. Neither are bits 12 – 31, which are for system flags (see [3]).

The names of the condition codes are:

OF	Overflow Flag
SF	Sign Flag
ZF	Zero Flag
AF	Auxiliary carry or Adjust Flag
PF	Parity Flag
CF	Carry Flag

The `OF`, `SF`, `ZF`, and `CF` are described at appropriate places in this book. See [3] and [14] for descriptions of the other flags.

Two other registers are very important in a program. The `rsp` register is used as a *stack pointer*, as will be discussed in Section 8.2 (page 176). The `rbp` register is typically used as a *base pointer*; it will be discussed in Section 8.4 (page 188).

The “e” prefix on the 32-bit portion of each register name comes from the history of the x86 architecture. The introduction of the 80386 in 1986 brought an increase of register size from 16 bits to 32 bits. There were no new registers. The old ones were simply “extended.”

## 6.3 CPU Interaction with Memory and I/O

The connections between the CPU and Memory are shown in Figure 6.4. This figure also includes the I/O (input and output) subsystem. The I/O system will be discussed in subsequent chapters. The control unit is connected to memory by three buses:

- address bus
- data bus
- control bus

**Bus:** a communication path between two or more devices.

Several devices can be connected to one bus, but only two devices can be communicating over the bus at one time.

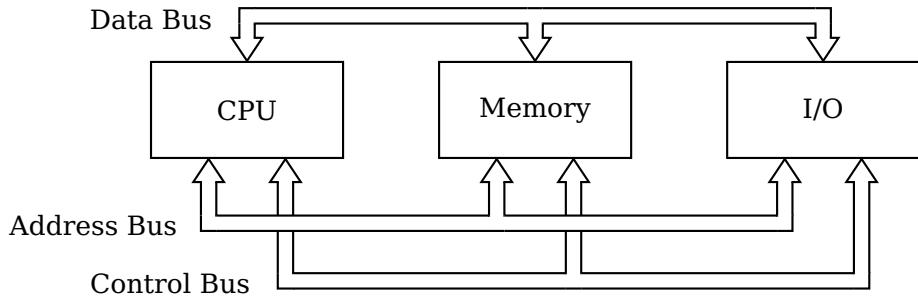


Figure 6.4: Subsystems of a computer. The CPU, Memory, and I/O subsystems communicate with one another via the three busses. (Repeat of Figure 1.1.)

As an example of how data can be stored in memory, let us imagine that we have some data in one of the CPU registers. Storing this data in memory is effected by setting the states of a group of bits in memory to match those in the CPU register. The control unit can be programmed to do this by

1. sending the memory address on the address bus,
2. sending a copy of the register bit states on the data bus, then
3. sending a “write” signal on the control bus.

For example, if the eight bits in memory at address `0x7ffffd9a43cef` are in the state:

`0x7ffffd9a43cef: b7`

the `al` register in the CPU is in the state:

`%al: e2`

and the control unit is programmed to store this value at location `0x7ffffd9a43cef`, the control unit then

1. places `0x7ffffd9a43cef` on the address bus,
2. places the bit pattern `e2` on the data bus, and
3. places a “write” signal on the control bus.

Then the bits at memory location `0x7ffffd9a43cef` will be changed to the state:

`0x7ffffd9a43cef: e2`

*Important.* When the state of any bit in memory or in a register is changed any previous states are lost forever. There is no way to “undo” this state change or to determine how the bit got in its current state.

## 6.4 Program Execution in the CPU

You may be wondering how the CPU is programmed. It contains a special register — the *instruction register* — whose bit pattern determines what the CPU will do. Once that

action has been completed, the bit pattern in the instruction register can be changed, and the CPU will perform the operation specified by this next bit pattern.

Most modern CPUs use an instruction queue. Several instructions are waiting in the queue, ready to be executed. Separate electronic circuitry keeps the instruction queue full while the regular control unit is executing the instructions. But this is simply an implementation detail that allows the control unit to run faster. The essence of how the control unit executes a program is represented by the single instruction register model.

Since instructions are simply bit patterns, they can be stored in memory. The instruction pointer register always has the memory address of (points to) the next instruction to be executed. In order for the control unit to execute this instruction, it is copied into the instruction register.

The situation is as follows:

1. A sequence of instructions is stored in memory.
2. The memory address where the first instruction is located is copied to the instruction pointer.
3. The CPU sends the address in the instruction pointer to memory on the address bus.
4. The CPU sends a “read” signal on the control bus.
5. Memory responds by sending a copy of the state of the bits at that memory location on the data bus, which the CPU then copies into its instruction register.
6. The instruction pointer is automatically incremented to contain the address of the next instruction in memory.
7. The CPU executes the instruction in the instruction register.
8. Go to step 3.

Steps 3, 4, and 5 are called an *instruction fetch*. Notice that steps 3 – 8 constitute a cycle, the *instruction execution cycle*. It is shown graphically in Figure 6.5.

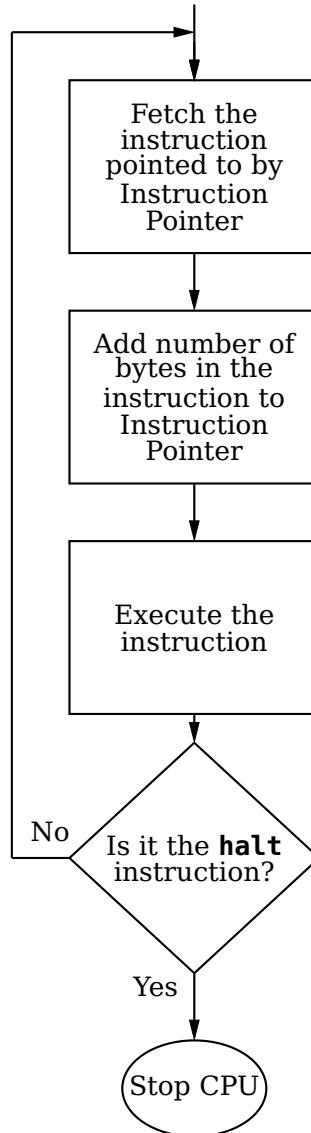


Figure 6.5: The instruction execution cycle.

This raises a couple of questions:

**How do we get the instructions into memory?** The instructions for a program are stored in a file on a storage device, usually a disk. The computer system is controlled by an operating system. When you indicate to the operating system that you wish to execute a program, e.g., by double-clicking on its icon, the operating system locates a region of memory large enough to hold the instructions in the program then copies them from the file to memory. The contents in the file remain unchanged.<sup>2</sup>

**How do we create a file on the disk that contains the instructions?** This is a multi-step process using several programs that are provided for you. The programs and the files that each create are:

- An *editor* is used to create source files.

The source file is written in a programming language, e.g., C++. This is very similar to creating a file with a word processor. The main differences are that an editor is much simpler than a word processor, and the contents of the source file are written in the programming language instead of, say, English.

- A *compiler/assembler* is used to create object files.

The compiler translates the programming language in a source file into the bit patterns that can be used by a CPU (machine language). The source file contents remains unchanged.

- A *linker* is used to create executable files.

Most programs are made up of several object files. For example, a GNU/Linux installation includes many object files that contain the machine instructions to perform common tasks. These are programs that have already been written and compiled. Related tasks are commonly grouped together into a single file called a library.

Whenever possible, you should use the short programs in these libraries to perform the computations your program needs rather than write it yourself. The linker program will merge the machine code from these several object files into one file.

You may have used an integrated development environment (IDE), e.g., Microsoft® Visual Studio®, Eclipse™, which combines all of these three programs into one package where each of the intermediate steps is performed automatically. You use the editor program to create the source file and then give the run command to the IDE. The IDE will compile the program in your source files, link the resulting object files with the necessary libraries, load the resulting executable file into memory, then start your program. In general, the intermediate object files resulting from the compilation of each source file are automatically deleted from the disk.

In this book we will explicitly perform each of these steps separately so we can learn the role of each program — editor, assembler, linker — used in preparing the application program.

## 6.5 Using gdb to View the CPU Registers

We will use the program in Listing 6.1 to illustrate the use of gdb to view the contents of the CPU registers. I have used the `register` storage class modifier to request that

<sup>2</sup>This is a highly simplified description. The details depend upon the overall system.

the compiler use a CPU register for the `int* ptr` variable. The `register` modifier is “advisory” only. See Exercise 6-3 for an example when the compiler may not be able to honor our request.

```
1 /*  
2  * gdbExample1.c  
3  * Subtracts one from user integer.  
4  * Demonstrate use of gdb to examine registers, etc.  
5  * Bob Plantz - 5 June 2009  
6 */  
7  
8 #include <stdio.h>  
9  
10 int main(void)  
11 {  
12     register int wye;  
13     int *ptr;  
14     int ex;  
15  
16     ptr = &ex;  
17     ex = 305441741;  
18     wye = -1;  
19     printf("Enter an integer: ");  
20     scanf("%i", ptr);  
21     wye += *ptr;  
22     printf("The result is %i\n", wye);  
23  
24     return 0;  
25 }
```

Listing 6.1: Simple program to illustrate the use of gdb to view CPU registers.

We introduced some `gdb` commands in Chapter 2. Here are some additional ones that will be used in this section:

- `n` — execute current source code statement of a program that has been running; if it’s a call to a function, the entire function is executed.
- `s` — execute current source code statement of a program that has been running; if it’s a call to a function, step into the function.
- `si` — execute current (machine) instruction of a program that has been running; if it’s a call to a function, step into the function.
- `i r` — `info registers` — displays the contents of the registers, except floating point and vector.

Here is a screen shot of how I compiled the program then used `gdb` to control the execution of the program and observe the register contents. My typing is **boldface** and the session is annotated in *italics*. Note that you will probably see different addresses if you replicate this example on your own (Exercise 6-1).

```
bob$ gcc -g -O0 -Wall -fno-asynchronous-unwind-tables \  
> -fno-stack-protector -o gdbExample1 gdbExample1.c
```

*The “-g” option is required. It tells the compiler to include debugger information in the executable program.*

```
bob$ gdb ./gdbExample1
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/bob/my_book_working/progs/chap06/gdbExample1...done
(gdb) li
7
8 #include <stdio.h>
9
10 int main(void)
11 {
12     register int wye;
13     int *ptr;
14     int ex;
15
16     ptr = &ex;
(gdb)
17     ex = 305441741;
18     wye = -1;
19     printf("Enter an integer: ");
20     scanf("%i", ptr);
21     wye += *ptr;
22     printf("The result is %i\n", wye);
23
24     return 0;
25 }
(gdb)
```

*The li command lists ten lines of source code. The display is centered around the current line. Since I have not started execution of this program, the display is centered around the beginning of main. The display ends with the (gdb) prompt. Pushing the return key repeats the previous command, and li is smart enough to display the next ten lines.*

```
(gdb) br 19
Breakpoint 1 at 0x4005a9: file gdbExample1.c, line 19.
(gdb) run
Starting program: /home/bob/my_book_64/progs/chap06/gdbExample1
```

```
Breakpoint 1, main () at gdbExample1.c:19
19         printf("Enter an integer: ");
```

*I set a breakpoint at line 19 then run the program. When line 19 is reached, the program is paused before the statement is executed, and control returns to gdb.*

```
(gdb) print ex
$1 = 305441741
(gdb) print &ex
$2 = (int *) 0x7fffffff044
```

*I use the print command to view the value assigned to the ex variable and learn its memory address.*

```
(gdb) help x
Examine memory: x/FMT ADDRESS.
```

ADDRESS is an expression for the memory address to examine.

FMT is a repeat count followed by a format letter and a size letter.

Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char) and s(string).

Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).

The specified number of objects of the specified size are printed according to the format.

Defaults for format and size letters are those previously used.

Default count is 1. Default address is following last thing printed with this command or "print".

*The help command will provide very brief instructions on using a command. We want to display values stored in specific memory locations in various formats, and the help command provides a reminder of how to use the command.*

```
(gdb) x/1dw 0x7fffffff044
0x7fffffff044: 305441741
```

*I verify that the value assigned to the ex variable is stored at location 0x7fffffff044.*

```
(gdb) x/1xw 0x7fffffff044
0x7fff504c473c: 0x1234abcd
```

*I examine the same integer in hexadecimal format.*

```
(gdb) x/4xb 0x7fffffff044
0x7fffffff044: 0xcd 0xab 0x34 0x12
```

*Next, I examine all four bytes of the word, one byte at a time. In this display,*

- *0xcd is stored in the byte at address 0x7fffffff044,*
- *0xab is stored in the byte at address 0x7fffffff045,*
- *0x34 is stored in the byte at address 0x7fffffff046, and*
- *0x12 is stored in the byte at address 0x7fffffff047.*

*In other words, the byte-wise display appears to be backwards. This is due to the values being stored in the little endian storage scheme as explained on page 20 in Chapter 2.*

```
(gdb) x/2xh 0x7fffffff044
0x7fffffff044: 0xabcd 0x1234
```

*I also examine all four bytes of the word, two bytes at a time. In this display,*

- *0xabcd is stored in the two bytes starting at address 0xfffffffffe044, and*
- *0x1234 is stored in the two bytes starting at address 0xfffffffffe046.*

*This shows how gdb displays these four bytes as though they represent two 16-bit ints stored in little endian format. (You can now see why I entered such a strange integer in this demonstration run.)*

```
(gdb) print ptr
$3 = (int *) 0x7fffffff044
(gdb) print &ptr
$4 = (int **) 0x7fffffff048
```

*Look carefully at the ptr variable. It is located at address 0xfffffffffe048 and it contains another address, 0x7fffffff044, that is, the address of the variable ex. It is important that you learn to distinguish between a memory address and the value that is stored there, which can be another memory address. Perhaps a good way to think about this is a group of numbered mailboxes, each containing a single piece of paper that you can write a single number on. You could write a number that represents a “data” value on the paper. Or you can write the address of a mailbox on the paper. One of the jobs of a programmer is to write the program such that it interprets the number appropriately — either a data value or an address.*

```
(gdb) print wye
$5 = -1
(gdb) print &wye
Address requested for identifier "wye" which is in register $rbx
```

*The compiler has honored our request and allocated a register for the wye variable. Registers are located in the CPU and do not have memory addresses, so gdb cannot print the address. We will need to use the i r command to view the register contents.*

```
(gdb) i r
rax            0x7fffffff044 140737488347204
rbx            0xffffffff 4294967295
rcx            0x4005e0 4195808
rdx            0x7fffffff158 140737488347480
rsi            0x7fffffff148 140737488347464
rdi            0x1 1
rbp            0x7fffffff060 0x7fffffff060
rsp            0x7fffffff040 0x7fffffff040
r8             0x400670 4195952
r9             0x7ffff7de9740 140737351948096
r10            0x7fffffffdec0 140737488346816
r11            0x7ffff7a3e680 140737348101760
r12            0x400480 4195456
r13            0x7fffffff140 140737488347456
r14            0x0 0
r15            0x0 0
rip            0x4005a9 0x4005a9 <main+29>
eflags          0x202 [ IF ]
```

cs	0x33 51
ss	0x2b 43
ds	0x0 0
es	0x0 0
fs	0x0 0
gs	0x0 0

The `i r` command displays the current contents of the CPU registers. The first column is the name of the register. The second shows the current bit pattern in the register, in hexadecimal. Notice that leading zeros are not displayed. The third column shows some the register contents in 64-bit signed decimal. The registers that always hold addresses are also shown in hexadecimal in the third column. The columns are often not aligned due to the tabbing of the display.

We see that the value in the `ebx` general purpose register is the same as that stored in the `wye` variable, `0xffffffff`.<sup>3</sup> (Recall that ints are 32 bits, even in 64-bit mode.) We conclude that the compiler chose to allocate `ebx` as the `wye` variable.

Notice the value in the `rip` register, `0x4005a9`. Refer back to where I set the breakpoint on source line 19. This shows that the program stopped at the correct memory location.

It is only coincidental that the address of the `ex` variable is currently stored in the `rax` register. If a general purpose register is not allocated as a variable within a function, it is often used to store results of intermediate computations. You will learn how to use registers this way in subsequent chapters of this book.

```
(gdb) br 21
Breakpoint 2 at 0x4005ce: file gdbExample1.c, line 21.
(gdb) br 22
Breakpoint 3 at 0x4005d6: file gdbExample1.c, line 22.
```

These two breakpoints will allow us to examine the value stored in the `wye` variable just before and after it is modified.

```
(gdb) cont
Continuing.
Enter an integer: 123

Breakpoint 2, main () at gdbExample1.c:21
21      wye += *ptr;
(gdb) print ex
$6 = 123
(gdb) print wye
$7 = -1
```

This verifies that the user's input value is stored correctly and that the `wye` variable has not yet been changed.

```
(gdb) cont
Continuing.
```

---

<sup>3</sup>If this is not clear, you need to review Section 3.3.

```
Breakpoint 3, main () at gdbExample1.c:22
22      printf("The result is %i\n", wye);
(gdb) print ex
$8 = 123
(gdb) print wye
$9 = 122
```

And this verifies that our (rather simple) algorithm works correctly.

```
(gdb) i r rbx rip
rbx          0x7a      122
rip          0x4005d6 0x4005d6 <main+74>
```

We can specify which registers to display with the `i r` command. This verifies that the `rbx` register is being used as the `wye` variable.

And we see that the `rip` has incremented from `0x4005a9` to `0x4005d6`. Don't forget that the `rip` register always points to the next instruction to be executed.

```
(gdb) cont
Continuing.
The result is 122
[Inferior 1 (process 4463) exited normally]
(gdb) q
bob$
```

Finally, I continue to the end of the program. Notice that `gdb` is still running and I have to quit the `gdb` program.

## 6.6 Exercises

**6-1** (§6.2, §6.5) Enter the program in Listing 6.1 and trace through the program one line at a time using `gdb`. Use the `n` command, not `s` or `si`. Keep a written record of the `rip` register at the beginning of each line. Hint: use the `i r` command. How many bytes of machine code are in each of the C statements in this program? Note that the addresses you see in the `rip` register may differ from the example given in this chapter.

**6-2** (§6.2, §6.4) As you trace through the program in Exercise 6-1 stop on line 22:

```
wye += *ptr;
```

We determined in the example above that the `%rbx` register is used for the variable `wye`. Inspect the registers.

- What is the address of the first instruction that will be executed when you enter the `n` command?
- How will `%rbx` change when this statement is executed?

**6-3** (§6.5) Modify the program in Listing 6.1 so that a register is also requested for the `ex` variable. Were you able to convince the compiler to do this for you? Did the compiler produce any error or warning messages? Why do you think the compiler would not use a register for this variable.

- 6-4** (§6.2, §6.5) Use the `gdb` debugger to observe the contents of memory in the program from Exercise 2-31. Verify that your algorithm creates a null-terminated string without the newline character.
- 6-5** (§6.2, §6.5) Write a program in C that allows you to determine the endianess of your computer. Hint: use `unsigned char* ptr`.
- 6-6** (§6.2, §6.5) Modify the program in Exercise 6-5 so that you can demonstrate, using `gdb`, that endianess is a property of the CPU. That is, even though a 32-bit `int` is stored little endian in memory, it will be read into a register in the “proper” order. Hint: declare a second `int` that is a register variable; examine memory one byte at a time.

# Chapter 7

# Programming in Assembly Language

*While reading this chapter, you should also consult the `info` resources available in most GNU/Linux installations for both the `make` and the `as` programs. Appendix B provides a general tutorial for writing `Makefiles`, but you need to get the details from `info`. `info` is especially important for learning about `as`'s assembler directives.*

*You should also reread the Development Environment section on page xvi.*

Creating a program in assembly language is essentially the same as creating one in a high-level compiled language like C, C++, Java, FORTRAN, etc. We will begin the chapter by looking in detail at the steps involved in creating a C program. Then we will look at which of these steps apply to assembly language programming.

## 7.1 Creating a New Program

You have probably learned how to program using an Integrated Development Environment (IDE) , which incorporates several programs within a single user interface:

1. A *text editor* is used to write the source code and save it in a file.
2. A *compiler* translates the source code into *machine language* that can be executed by the CPU.
3. A *linker* is used to integrate all the functions in your program, including externally accessed libraries of functions, and to determine where each component will be loaded into memory when the program is executed.
4. A *loader* is used to load the machine code version of the program into memory where the CPU can execute it.
5. A *debugger* is used to help the programmer locate errors that may have crept into the program. (Yes, none of us is perfect!)

You enter your source code in the text editor part, click on a “build” button to compile and link your program, then click on a “run” button to load and execute the program. There is typically a “debug” button that loads and executes the program under control of the debugger program if you need to debug it. The individual steps of program preparation

are obscured by the IDE user interface. In this book we use the GNU programming environment in which each step is performed explicitly.

Several excellent text editors exist for GNU/Linux, each with its own “personality.” My “favorite” changes from time to time. I recommend trying several that are available to you and deciding which one you prefer. You should avoid using a word processor to create source files because it will add formatting to the text (unless you explicitly specify text-only). Text editors I have used include:

- `gedit` is probably installed if you are using the `gnome` desktop.
- `kate` is probably installed if you are using the `kde` desktop.
- `vi` is supposed to be installed on all Linux (and Unix) systems. It provides a command line user interface that is mode oriented. Text is manipulated through keyboard commands. Several commands place `vi` in “text insert” mode. The ‘`esc`’ key is used to return to command mode. Most installations include `vim` (Vi IMproved) which has additional features helpful in editing program source code.
- `emacs` also has a command line user interface. Text is inserted directly. The ‘`ctrl`’ and “meta” keys are used to specify keyboard sequences for manipulating text.

GUI interfaces are available for both `vi` and `emacs`. Any of these, and many other, text editors would be an excellent choice for the programming covered in this book. Don’t spend too much time trying to pick the “best” one.

The GNU programming tools are executed from the command line instead of a graphical user interface (GUI). (IDEs for Linux and Unix are typically GUI frontends that execute GNU programming tools behind the scenes.) The GNU compiler, `gcc`, creates an executable program by performing several distinct steps [22]. The description here assumes a single C source file, `filename.c`.

1. *Preprocessing*. This resolves compiler directives such as `#include` (file inclusion), `#define` (macro definition), and `#if` (conditional compilation) by invoking the program `cpp`. Compilation can be stopped at the end of the preprocessing phase with the `-E` option, which writes the resulting C source code to standard out.
2. *Compilation itself*. The source code that results from preprocessing is translated into assembly language. Compilation can be stopped at the end of the compilation phase with the `-S` option, which writes the assembly language source code to `filename.s`.
3. *Assembly*. The assembly language source code that results from compilation is translated into machine code by invoking the `as` program. Compilation can be stopped at the end of the assembly phase with the `-c` option, which writes the machine code to `filename.o`.
4. *Linking*. The machine code that results from assembly is linked with other machine code from standard C libraries and other machine code modules, and addresses are resolved. This is accomplished by invoking the `ld` program. The default is to write the executable file, `a.out`. A different executable file name can be specified with the `-o` option.

## 7.2 Program Organization

Programs written in C are organized into functions. Each function has a name that is unique within the program. Program execution begins with the function named “`main`.”

I recommend that you create a separate directory for each program you write. Place all the source files, plus the `Makefile` (see Appendix B) for the program in this directory. This will help you keep your program files organized.

Let us consider the minimum C program, Listing 7.1.

```

1  /*
2   * doNothingProg1.c
3   * The minimum components of a C program.
4   * Bob Plantz - 6 June 2009
5   */
6
7 int main(void)
8 {
9     return 0;
10}
```

Listing 7.1: A “null” program (C).

The only thing this program does is return a zero.

Despite the fact that this program accomplishes very little, some instructions need to be executed just to return zero. In order to see what takes place, we first translate this program from C to assembly language with the GNU/Linux command:

```
gcc -S -O0 doNothingProg1.c
```

This creates the file `doNothingProg1.s` (see Listing 7.2), which contains the assembly language generated by the `gcc` compiler. The two compiler options used here have the following meanings:

**-S** Causes the compiler to create the `.s` file, which contains the assembly language equivalent of the source code. The machine code (`.o` file) is not created.

**-O0** Do not do any optimization. For instructional purposes, we want to see every step of the assembly language. (This is upper-case “oh” followed by the numeral zero.)

```

1      .file  "doNothingProg1.c"
2      .text
3      .globl  main
4      .type   main, @function
5 main:
6 .LFB0:
7      .cfi_startproc
8      pushq   %rbp
9      .cfi_def_cfa_offset 16
10     .cfi_offset 6, -16
11     movq   %rsp, %rbp
12     .cfi_def_cfa_register 6
13     movl   $0, %eax
14     popq   %rbp
15     .cfi_def_cfa 7, 8
16     ret
17     .cfi_endproc
18 .LFE0:
19     .size   main, .-main
```

```
20     .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
21     .section      .note.GNU-stack, "",@progbits
```

Listing 7.2: A “null” program (gcc assembly language). The lines that begin with `.cfi` are meant for error recovery and debugging and are not relevant to the concepts discussed in this book.

Unlike the relationship between assembly language and machine language, there is not a one-to-one relationship between higher-level languages and assembly language. The assembly language generated by a compiler may differ across different releases of the compiler, and different optimization levels will generally affect the code that is generated by the compiler. The code in Listing 7.2 was generated by release 4.7.0 of gcc and the optimization level was `-O0` (no optimization). If you attempt to replicate this example, your results may vary.

This is not easy to read, even for an experienced assembly language programmer. So we will start with the program in Listing 7.3, which was written in assembly language by a programmer (rather than by a compiler). Naturally, the programmer has added comments to improve readability.

```
1 # doNothingProg2.s
2 # Minimum components of a C program, in assembly language.
3 # Bob Plantz - 6 June 2009
4
5     .text
6     .globl main
7     .type main, @function
8 main: pushq %rbp      # save caller's frame pointer
9     movq %rsp, %rbp    # establish our frame pointer
10
11    movl $0, %eax    # return 0 to caller
12    movq %rbp, %rsp    # restore stack pointer
13    popq %rbp      # restore caller's frame pointer
14    ret             # back to caller
```

Listing 7.3: A “null” program (programmer assembly language).

After examining what the assembly language programmer did we will return to Listing 7.2 and look at the assembly language generated by the compiler.

*Assembly language* provides of a set of mnemonics that have a one-to-one correspondence to the machine language. A *mnemonic* is a short, English-like group of characters that suggests the action of the instruction. For example, “`mov`” is used to represent the instruction that copies (“moves”) a value from one place to another. Thus, the machine instruction

4889E5

copies the entire 64-bit value in the `rsp` register to the `rbp` register. Even if you have never seen assembly language before, the mnemonic representation of this instruction in Listing 7.2,

```
9     movq %rsp, %rbp    # establish our frame pointer
```

probably makes much more sense to you than the machine code. (The ‘`q`’ suffix on “`mov`” means a quadword (64 bits) is being moved.)

Strictly speaking, the mnemonics are completely arbitrary, as long as you have an assembler program that will translate them into the desired machine instructions. However, most assembler programs more or less use the mnemonics used in the manuals provided by CPU vendors.

The first thing to notice is that *assembly language is line-oriented*. That is, there is only one assembly language statement on each line, and none of the statements spans more than one line. A statement can continue onto subsequent lines, but this requires a special line-continuation character. This differs from the “free form” nature of C/C++ where the line structure is irrelevant. In fact, good C/C++ programmers take advantage of this to improve the readability of their code.

Next, notice that the pattern of each line falls into one of three categories:

- Lines 1 – 3 begin with the “#” character. The rest of the line is written in English and is easily read. The “#” character in the first column designates a *comment line*. Just as with a high-level language, comments are intended solely for the human reader and have no effect on the program.
- Lines 4 and 10 have been left blank in order to improve readability. (Well, they will improve readability once you learn how to read assembly language.)
- The remaining nine lines are organized into columns. They probably do not make much sense to you at this point because they are written in assembly language, but if you look carefully, each of the assembly language lines is organized into four possible fields:

*label : operation operand(s) #comment*

The assembler requires at least one space or tab character to separate the fields. When writing assembly language, your program will be much easier to read if you use the tab key to move from one field to the next.

Let us consider each field:

1. The *label field* allows us to give a symbolic name to any line in the program. Since each line corresponds to a memory location in the program, other parts of the program can then refer to the memory location by name.
  - (a) A label consists of an *identifier* immediately followed by the “:” character. You, as the programmer, must make up these identifiers. The rules for creating an identifier are given below.
  - (b) Notice that most lines are not labeled.
2. The *operation field* provides the basic purpose of the line. There are two types of operations:
  - (a) *assembly language mnemonic* — The assembler translates these into actual machine instructions, which are copied into memory when the program is to be executed. Each machine instruction will occupy from one to five bytes of memory.
  - (b) *assembler directive(pseudo op)* — Each of these operations begins with the period (“.”) character. They are used to direct the way in which the assembler translates the file. They do not translate directly into machine instructions, although some do cause memory to be allocated.
3. The *operand field* specifies the arguments to be used by the operation. The arguments are specified in several different ways:

- (a) an explicit — or literal — value, e.g., the integer 75.
- (b) a name that has meaning to the assembler, e.g., the name of a register.
- (c) a name that is made up by the programmer, e.g., the name of a variable or a constant.

Different operations require differing numbers of operands — zero, one, two, or three.

4. The *comment field* is just like a comment line, except it takes up only the remainder of the line. Since assembly language is not as easy to read as higher-level languages, good programmers will place a comment on almost every line.

The rules for creating an identifier are very similar to those for C/C++. Each identifier consists of a sequence of alphanumeric characters and may include other printable characters such as “.”, “\_”, and “\$”. The first character must not be a numeral. An identifier may be any length, and all characters are significant. Case is also significant. For example, “myLabel” and “MyLabel” are different. Compiler-generated labels begin with the “.” character, and many system related names begin with the “\_” character. It is a good idea to avoid beginning your own labels with the “.” or the “\_” character so that you do not inadvertently create one that is already in use by the system.

Integers can be used as labels, but they have a special meaning. They are used as *local labels*, which are sometimes useful in advanced assembly language programming techniques. They will not be used in this book.

The assembler program, as, will translate the file `doNothingProg2.s` (see Listing 7.3) into machine code and provide the memory allocation information for the operating system to use when the program is executed. We will first describe the contents of this file, then look at the GNU commands to convert it into an executable program.

Now we turn attention to the specific file in Listing 7.3, `doNothingProg2.s`. On line 5 you recognize

5        .text

as an assembler directive because it starts with a period character. It directs the assembler to place whatever follows in the *text section*.

What does “text section” mean? When a source code file is translated into machine code, an *object file* is produced. The object file organization follows the Executable and Linking Format (ELF). ELF files can be seen from two different points of view. Programs that store information in ELF files store it in *sections*. The ELF standard specifies many different types of sections, each depending on the type of information stored in it.

The `.text` directive specifies that when the following assembly language statements are translated into machine instructions, they should be stored in a text section in the object file. Text sections are used to store program instructions in machine code format.

GNU/Linux divides memory into different *segments* for specific purposes when a program is loaded from the disk. The four general categories are:

- *text* (also called *code*) is where program instructions and constant data are stored. It is read-only memory. The operating system prevents a program from changing anything stored in the text segment.
- *data* is where global variables and static local variables are stored. It is read-write memory and remains in place for the duration of the program.
- *stack* is where automatic local variables and the data that links functions are stored. It is read-write memory that is allocated and deallocated dynamically as the program executes.

- *heap* is the pool of memory available when a C program calls the `malloc` function (or C++ calls `new`). It is read-write memory that is allocated and deallocated by the program.

The operating system needs to view an ELF file as a set of segments. One of the functions of the `ld` program is to group sections together into segments so that they can be loaded into memory. Each segment contains one or more sections. This grouping is generally accomplished by arrays of pointers to the file, not necessarily by physically moving the sections. That is, there is still a section view of the ELF file remaining. So the information stored in an ELF file is grouped into sections, but it may or may not also be grouped into segments.

When the operating system loads the program into memory, it uses the segment view of the ELF file. Thus the contents of all the text sections will be loaded into the text segment of the program process.

This has been a very simplistic overview of ELF sections and segments. We will touch on the subject again briefly in Section 8.1. Further details can be found by reading the `man` page for `elf` and sources like [13] and [21]. The `readelf` program is also useful for learning about ELF files. It is included in the `binutils` collection of the GNU binary tools so is installed along with `as` and `ld`.

The assembler directive on line 6

```
6      .globl  main
```

has one operand, the identifier “`main`.” As you know, all C/C++ programs start with the function named “`main`.” In this book, we also start our assembly language programs with a `main` function and execute them within the C/C++ runtime environment. The `.globl` directive makes the name globally known, analogous to defining an identifier outside a function body in C/C++.<sup>1</sup> That is, code outside this file can refer to this name. When a program is executed, the operating system does some preliminary set up of system resources. It then starts program execution by calling a function named “`main`,” so the name must be global in scope.

One can write stand-alone assembly language programs. In GNU/Linux this is accomplished by using the `_start` label on the first instruction in the program. The object (`.o`) files are then linked using the `ld` command directly rather than use `gcc`. See Section 8.6.

The assembler directive on line 7

```
7      .type   main, @function
```

has two operands: a name and a type. The name is entered into the symbol table (see Section 7.3). In addition to the machine code, the object file contains the symbol table along with information about each symbol. The ELF format recognizes two types of symbols: data and function. The `.type` directive is used here to specify that the symbol `main` is the name of a function.

None of these three directives get translated into actual machine instructions, and none of them occupy any memory in the finished program. Rather, they are used to describe the characteristics of the statements that follow.

---

<sup>1</sup>Function names are defined outside the function body (outside the `{...}` block) in C/C++. Hence, the names are global, and a function can call functions defined in other files. Variables can also be declared outside functions. Functions in other files can reference such variables using the `extern` storage class specifier.

**IMPORTANT!** You need to distinguish

**assembler directives** — instructions to the assembler (the program that translates assembly language into machine code).

from

**assembly language instructions** — the code that gets translated into machine code.

What follows next in Listing 7.3 are the actual assembly language instructions. They will occupy memory when they are translated. The first instruction is on line 8:

```
8    main:  pushq  %rbp      # save caller's frame pointer
```

It illustrates the use of all four fields on a line of assembly language.

1. First, there is a label on this line, `main`. Since this name has been declared as a global name (with the assembler directive `.globl main`), functions defined in other files can call this function by name. In particular, after the operating system has loaded this function into memory, it can call `main`, and execution will start with this line.
2. The operation is a `pushq` instruction, which stands for “push quadword.” It “pushes” a value onto the call stack. This will be explained in Section 8.2 (page 176). For now, this is a technique for temporarily saving the value stored in the operand.

The “quadword” part of this instruction means that 64 bits are moved. As you will see in more detail later, as requires that a single letter be appended to most instructions:

“b”	⇒	“byte”	⇒	operand is 8 bits
“w”	⇒	“word”	⇒	operand is 16 bits
“l”	⇒	“long”	⇒	operand is 32 bits
“q”	⇒	“quadword”	⇒	operand is 64 bits

to specify the size of the operand(s).

3. There is one operand, `%rbp`. The GNU assembler requires the “%” prefix on the operand to indicate that this is the name of a register in the cpu. This instruction saves the 64-bit value in the `rbp` register on the call stack.

The value in the `rbp` register is an address. In 64-bit mode addresses can be 64 bits long, and we have to save the entire address.

4. Finally, we have added a comment to this line. The comment shows that the purpose of this instruction is to save the value that the calling function was using as a frame pointer. (The reasons for doing this will be explained in Chapter 8.)

The next line

```
9    movq  %rsp, %rbp  # establish our frame pointer
```

uses only three of the fields.

1. First, there is no label on this line. Notice that the label field is left blank by using the tab key to indent into the second field, the operation field. It is important for readability that you use the tab key to keep the beginning of each field lined up in a column.

2. The operation is a `movq` instruction, which stands for “move quadword.” It “moves” a bit pattern from one location to another. Actually, “copy” is probably a better term than “move,” because it does not change the bit pattern in the place copied from. But “move” has become the accepted terminology for this operation.
3. There are two operands, `%rsp` and `%rbp`. Again, the “%” prefix to each operand means that it is the name of a register in the cpu.
  - (a) The order of the operands in `as` is: *source, destination*.
  - (b) Thus this instruction copies the 64-bit value in the `rsp` register to the `rbp` register in the cpu.
4. Finally, I have added a comment to this line. The comment shows that the purpose of this instruction is to establish a new frame pointer in this function. (Again, the reasons for doing this will be explained in Chapter 8.)

As the name of this “program” implies, it does not do anything, but it still must return to the operating system. GNU/Linux expects the `main` function to return an integer to it, and the *return value is placed in the eax register*. Zero means that the program executed with no errors. This may not make a lot of sense to you at this point, but it should become clearer later in the book. Returning the integer zero to the operating system is accomplished on line 12:

11           `movl $0, %eax # return 0 to caller`

1. This line also has no label. After indenting, it begins with a `movl` instruction.
  2. The first operand is prefixed with a “\$” character, which indicates that the operand is to be taken as a literal value. That is, the source operand is the integer zero. You recognize that the second operand is the `eax` register in the cpu. This instruction places a copy of the 32-bit integer zero in the `eax` register.
- Even though the CPU is in 64-bit mode, 64-bit integers are seldom needed. So the default behavior of environment is to use 32 bits for `ints`. 64-bit `ints` can be specified in C/C++ with either the `long` or the `long long` modifier. In assembly language the programmer would use quadwords for integers. (As pointed out on page 132 this instruction also zeros the high-order 32 bits of the `rax` register. But you should not write code that depends upon this behavior.)
3. The comment on this line shows that the purpose of this instruction is to return a zero to the calling function (the operating system).

The first two instructions in this function,

8           `main: pushq %rbp # save caller's frame pointer`  
 9           `movq %rbp, %rsp # establish our frame pointer`

form a *prologue* to the actual processing that is performed by the function. They changed some values in registers and used the call stack. Before returning to the operating system, it is essential that an *epilogue* be executed to restore the values. The following two instructions accomplish this.

12           `movq %rbp, %rsp # restore stack pointer`  
 13           `popq %rbp # restore caller's frame pointer`

- No labels are used on these lines. The `movq` instruction ensures that the stack pointer is moved back to the location where the `rbp` register was saved. Since the stack pointer was not used in this function, this instruction is not necessary here. But your program will crash if the stack pointer is not in the correct location when the next instruction is executed, so it is a good idea to get into the habit of always using both these instructions at the end of a function.
- The `popq` instruction copies the 64-bit value on the top of the call stack into the operand and moves the stack pointer accordingly. (You will learn about using the stack pointer in Section 8.2.) The operand in this case is the `rbp` register.
- The comment states that the reason for the `popq` instruction is to restore the frame pointer value for the calling function (the operating system since this is `main`).

Finally, this function must return to the function that called it, which is back in the operating system.

14                    `ret`                    `# back to caller`

- This line has no label. And the instruction does not specify any operands. This is the instruction for returning program control back to the function that called this one. In this particular case, since this is the `main` function, control is passed back to the operating system.
- Here is an example of an instruction that changes the value in the instruction pointer register (`rip`) in order to alter the linear flow of the program. We will see later the mechanism that is used to implement this.
- The comment on this line briefly describes the reason for the instruction.

## 7.2.1 First instructions

As you can see from this example, even a function that does nothing requires several instructions. The most commonly used assembly language instruction is

`movs source, destination`

where *s* denotes the size of the operand:

<i>s</i>	meaning	number of bits
b	byte	8
w	word	16
l	longword	32
q	quadword	64

In the Intel syntax, the size of the data is determined by the operand, so the size character (b, w, l, or q) is not appended to the instruction, and the order of the operands is reversed:

Intel®  
Syntax

`mov destination, source`

The `mov` instruction copies the bit pattern from the source operand to the destination operand. The bit pattern of the source operand is not changed. If the destination operand is a register and its size is less than 64 bits, the effect on the other bits in the register is shown in Table 7.1.

The `mov` instruction does not affect the `rflags` register. In particular, neither the CF nor the OF flags are affected. No more than one of the two operands may be a memory

<b>size</b>	<b>destination bits</b>	<b>remaining bits</b>
8	7 - 0	63 - 8 are unchanged
8	15 - 8	63 - 16, 7 - 0 are unchanged
16	15 - 0	63 - 16 are unchanged
32	31 - 0	63 - 32 are set to 0

Table 7.1: Effect on other bits in a register when less than 64 bits are changed.

location. Thus, in order to move a value from one memory location to another, it must be moved from the first memory location into a register, then from that register into the second memory location. (Accessing data in memory will be covered in Sections 8.1 and 8.4.)

The other instructions used in this “do nothing” program — `pushq`, `popq`, and `ret` — use the call stack. The call stack will be discussed in Section 8.2, which will then allow us to discuss these instructions. For now, you should memorize how to use them as “boilerplate” for the prologue and epilogue of each function.

## 7.2.2 A Note About Syntax

If you have any experience with x86 assembly language, the syntax used by the GNU assembler, `as`, will look a little strange to you. In principle, the syntax is arbitrary. A programmer could invent any sort of assembly language and write a program that would translate it into the appropriate machine code. But most cpu manufacturers publish a manual with a suggested assembly language syntax for their cpu.

Most assemblers for the x86 cpus follow the syntax suggested by Intel®, but `as` uses the AT&T syntax. It is not radically different from Intel’s. Some of the more striking differences are:

	<b>AT&amp;T</b>	<b>Intel®</b>
operand order:	source, destination	destination, source
register names:	prefixed with the “%” character, e.g., %eax	just the name, e.g., eax
literal values:	prefixed with the “\$” character, e.g., \$123	just the value, e.g., 123
operand size:	use the b, w, l, or q suffix on opcode to denote byte, word, long, or quadruple word	determined by the register specification (more complicated if operand is stored in memory)

The GNU assembler, `as`, does not require the size suffix on instructions in all cases. From the `info` documentation for `as`:

### 9.13.4 Instruction Naming

---

Instruction mnemonics are suffixed with one character modifiers which specify the size of operands. The letters ‘b’, ‘w’, ‘l’ and ‘q’ specify byte, word, long and quadruple word operands. If no suffix is specified by an instruction then ‘as’ tries to fill in the missing suffix based on the destination register operand (the last one by convention). Thus, ‘`mov %ax, %bx`’ is equivalent to ‘`movw %ax, %bx`’; also, ‘`mov $1, %bx`’ is equivalent to ‘`movw $1, bx`’. Note that this is

incompatible with the AT&T Unix assembler which assumes that a missing mnemonic suffix implies long operand size. (This incompatibility does not affect compiler output since compilers always explicitly specify the mnemonic suffix.)

It is recommended that you get in the habit of using the size suffix letters when you begin writing your own assembly language. This will help you to avoid introducing obscure bugs in your code.

The assembler directives are typically not specified by the cpu manufacturer, so you will see a much wider variety of syntax, depending on the particular assembler program. We will not try to list any differences here.

The GNU assembler, `as`, also supports the Intel® syntax. The assembler directive `.intel_syntax` says that following assembly language is written in the Intel® syntax; `.att_syntax` says it is written in AT&T syntax. Using Intel® syntax, the assembly language code in Listing 7.3 would be written

Intel® Syntax	<pre> main:   push   rbp         mov    rbp,  rsp          mov    eax,  0         mov    rsp,  rbp         pop    rbp         ret       </pre>
------------------	--

Keep in mind that `gcc` produces assembly language in AT&T syntax, so you will undoubtedly find it easier to use that when you write your own code. The `.intel_syntax` directive might be useful if somebody gives you an entire function written in Intel® syntax assembly language.

The syntax rules for our particular assembler, `as`, are described in an on-line manual that is in the GNU `info` format. `as` supports some two dozen computer architectures, so it is a challenge to wade through the `info` manual to find what you need. On the other hand, it provides the most up to date information. And it is especially important for learning how to use assembler directives because they are specific to the assembler.

Now would be a good time to start learning how to use `info` for `as`. As you encounter new assembly language concepts in this book, also look them up in `info` for `as`. If you are unfamiliar with `info`, at the GNU/Linux prompt, simply type

```
$ info info
```

for a nice tutorial.

### 7.2.3 The Additional Assembly Language Generated by the Compiler

First, notice that the compiler-generated labels (`.LFB0`, `.LFE0`) each begin with a period character, just like assembler directives. You can tell that they are labels because of the `:` immediately following.

If you compare the assembly language program in Listing 7.3 with that generated by the compiler in Listing 7.2, you can see that the compiler includes much more information in the file. Most of this information will not be used elsewhere in this book. We explain it here for completeness.

The first line,

```
.file  "doNothingProg1.c"
```

identifies the name of the C source file. When you write in assembly language this information clearly does not apply.

There are six `.cfi_...` (Call Frame Information) directives in this code on lines 7, 9, 10, 12, 15, and 17. These are used to provide information that is helpful in error recovery and debugging the program when it is running. We will not use them in this book. The `gcc` option to generate assembly language code without them is `-fno-asynchronous-unwind-tables`. This will be discussed in more detail below.

The eight lines

```
5 main:
6 .LFB0:
7     .cfi_startproc
8     pushq %rbp
9     .cfi_def_cfa_offset 16
10    .cfi_offset 6, -16
11    movq %rsp, %rbp
12    .cfi_def_cfa_register 6
```

set up the call stack for this function. The use of the call stack will be explained in more detail in Section 8.2 on page 176 and in subsequent Sections.

The labels generated by the compiler, `.LFB0` and `.LFE0` are used to mark the beginning and end of the program to improve the linking efficiency. Our programs will be very simple, so we will not need such labels.

Notice that the lines after the labels `main`, `.LFB0`, and `.LFE0`, are blank. The assembler does not generate any machine code for either of these two lines, so they do not take up any memory. The next thing that comes in memory is the

```
7     pushq %rbp
```

instruction. Thus, both labels, `main` and `.LFB0`, apply to the address where this instruction is located.

We can use the `-fno-asynchronous-unwind-tables` option to turn off the `.cfi` directives feature, as shown in Listing 7.4. The GNU/Linux command is:

```
gcc -S -O0 -fno-asynchronous-unwind-tables doNothingProg1.c
```

which gives the compiler-generated assembly language in Listing 7.4.

---

```
1     .file "doNothingProg1.c"
2     .text
3     .globl main
4     .type main, @function
5 main:
6     pushq %rbp
7     movq %rsp, %rbp
8     movl $0, %eax
9     popq %rbp
10    ret
11    .size main, .-main
12    .ident "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
13    .section .note.GNU-stack, "",@progbits
```

**Listing 7.4:** A “null” program (gcc assembly language). We have used the `-fno-asynchronous-unwind-tables` compiler option to remove the `.cfi` directives.

Lines 19 – 21 in Listing 7.2 are the same as lines 11 – 13 in Listing 7.4. They also use directives that do not apply to the programs we will be writing in this book.

Finally, you may have noticed that the `main` label is on a line by itself in Listing 7.2 but not in Listing 7.3. When there is only a label on a line, no machine instructions are generated, and no memory is allocated. Thus, the label really applies to the next line. It is common to place labels on their own line so that longer, easier to read labels can be used while still keeping the operations visually lined up in a column. This technique is illustrated in Listing 7.5.

---

```

1 # doNothingProg3.s
2 # The minimum components of a C program, written in assembly
3 # language. Same as doNothingProg2.s, except with the main
4 # label on its own line.
5 # Bob Plantz - 7 June 2009
6
7     .text
8     .globl  main
9     .type   main, @function
10 main:
11     pushq   %rbp      # save caller's frame pointer
12     movq   %rsp, %rbp # establish our frame pointer
13
14     movl   $0, %eax    # return 0 to caller
15     movq   %rbp, %rsp  # restore stack pointer
16     popq   %rbp      # restore caller's frame pointer
17     ret          # back to caller

```

---

Listing 7.5: The “null” program rewritten to show a label placed on its own line.

## 7.2.4 Viewing Both the Assembly Language and C Source Code

The `gcc` compiler provides a set of options that will allow you to generate a listing that shows both the assembly language and the corresponding C statement(s). This will allow you to more easily see the assembly language that the compiler generates to implement a C statement in assembly language. Compiling the program in Listing 7.1 with the command:

```
$ gcc -O0 -g -Wa,-adhls doNothingProg1.c > doNothingProg1.lst
```

generates the assembly language code in Listing 7.6.

---

```

1 GAS LISTING /tmp/ccYnmi4d.s                               page 1
2
3
4     1                      .file   "doNothingProg1.c"
5
6     2                      .text
7     3     .Ltext0:           .cfi_sections .debug_frame
8     4
9     5     .globl  main
10    6     main:
11    7     .LFB0:
12    8
13    9     .file 1 "doNothingProg1.c"
14
15 1:doNothingProg1.c **** /*

```

---

```

13 2:doNothingProg1.c **** * doNothingProg1.c
14 3:doNothingProg1.c **** * The minimum components of a C program.
15 4:doNothingProg1.c **** * Bob Plantz - 6 June 2009
16 5:doNothingProg1.c **** */
17 6:doNothingProg1.c ****
18 7:doNothingProg1.c **** int main(void)
19 8:doNothingProg1.c **** {
20 10          .loc 1 8 0
21 11          .cfi_startproc
22 12 0000 55      pushq %rbp
23 13          .LCFI0:
24 14          .cfi_def_cfa_offset 16
25 15          .cfi_offset 6, -16
26 16 0001 4889E5    movq %rsp, %rbp
27 17          .LCFI1:
28 18          .cfi_def_cfa_register 6
29 19 9:doNothingProg1.c **** return 0;
30 20 0004 B8000000      .loc 1 9 0
31 21 20          movl $0, %eax
32 22 0009 5D      .loc 1 10 0
33 23 23          popq %rbp
34 24 24          .LCFI2:
35 25 25 000a C3      .cfi_def_cfa 7, 8
36 26 26          ret
37 27          .cfi_endproc
38 28          .LFE0:
39 29          .Letext0:

```

42 GAS LISTING /tmp/ccYnmi4d.s

[page 2](#)

43

44

#### 45 DEFINED SYMBOLS

46

47

/tmp/ccYnmi4d.s:7 \*ABS\*:0000000000000000 doNothingProg1.c

48

49

.text:0000000000000000 main

#### NO UNDEFINED SYMBOLS

Listing 7.6: Assembly language embedded in C source code listing. The line number in the C source file is also indicated with the .loc assembler directive.

The “-g” option tells the compiler to include symbols for debugging. “-Wa,” passes the immediately following options to the assembly phase of the compilation process. Thus, the options passed to the assembler are “-adhls”, which cause the assembler to generate a listing with the following characteristics:

- -ad: omit debugging directives
- -ah: include-high level source
- -al: include assembly
- -as: include symbols

As you can see above, the secondary letters can be combined with one “-a.” The listing is written to standard out, which can be redirected to a file. We gave this file the “.lst” file extension because it cannot be assembled.

*Aside:* The version of gcc used on Ubuntu 12.04 generates .cfi directives for the assembler. The -fno-dwarf2-cfi-asn option causes them not to be generated, but then unwind tables are generated. The -fno-asynchronous-unwind-tables option causes neither to be generated in the assembly language code. This may differ in other versions of gcc and other Linux distributions.

### 7.2.5 Minimum Program in 32-bit Mode

The x86-64 processors can also run in 32-bit mode. Most GNU/Linux distributions also provide a 32-bit version. Some distributions are only available in 32-bit.

The gcc option to compile a program for 32-bit mode is -m32. Listing 7.7 shows the assembly language generated by the GNU/Linux command:

```
gcc -S -O0 -m32 -fno-asynchronous-unwind-tables doNothingProg1.c
```

---

```

1      .file  "doNothingProg1.c"
2      .text
3      .globl main
4      .type  main, @function
5 main:
6      pushl %ebp
7      movl %esp, %ebp
8      movl $0, %eax
9      popl %ebp
10     ret
11     .size  main, .-main
12     .ident "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
13     .section .note.GNU-stack,"",@progbits

```

---

Listing 7.7: A “null” program (gcc assembly language in 32-bit mode).

The only differences between the 32-bit and 64-bit versions are that all the instructions use the “l” suffix to indicate “longword” because addresses are 32 bits, and only the 32-bit portion of the registers is used. That is, esp instead of rsp, etc.

An assembly language programmer would comment the code, as shown in Listing 7.8.

---

```

1 # doNothingProg4.s
2 # The minimum components of a C program, written in assembly
3 # language. A 32-bit version of doNothingProg1.s.
4 # Bob Plantz - 7 June 2009
5
6     .text
7     .globl main
8     .type  main, @function
9 main:
10    pushl %ebp      # save caller's frame pointer
11    movl %esp, %ebp # establish our frame pointer
12    movl $0, %eax   # return 0 to caller
13    movl %ebp, %esp # restore stack pointer

```

---

```

14      popl    %ebp      # restore caller's frame pointer
15      ret       # back to caller

```

Listing 7.8: A “null” program (programmer assembly language in 32-bit mode).

This assembly language version also includes the instruction to restore the stack pointer near the end of the function:

```

13      movl    %ebp, %esp # restore stack pointer

```

Since this function does not use the stack, this instruction is not required, but including it is a good habit to establish as you start to write your own functions in assembly language.

## 7.3 Assemblers and Linkers

We present a highly simplified view of how assemblers and linkers work here. The goal of this presentation is to introduce the concepts. Most assemblers and linkers have capabilities that go far beyond the concepts described here (e.g., macro expansion, dynamic load/link). We leave a more thorough discussion of assemblers and linkers to a book on systems programming.

### 7.3.1 Assemblers

An assembler must perform the following tasks:

- Translate assembly language mnemonics into machine language.
- Translate symbolic names for addresses into numeric addresses.

Since the numeric value of an address may be required before an instruction can be translated into machine language, there is a problem with forward references to memory locations. For example, a code sequence like:

```

1 # if (response == 'y')
2     cmpb    $'y', response # was it 'y'?
3     jne     noChange      # no, there is no change
4
5 # then print the "save" message
6     movq    $saveMsg, %rbx # point to first char
7 saveLoop:
8     cmpb    $0, (%rbx)   # at null character?
9     je      saveEnd      # yes, exit loop
10
11    movl    $1, %edx      # no, send one byte
12    movq    %rbx, %rsi    #      at this location
13    movl    $STDOUT, %edi  #      to screen.
14    call    write
15    incq    %rbx          # increment char pointer
16    jmp     saveLoop      # check at top of loop
17 saveEnd:
18    jmp     allDone        # skip over false block
19
20 # else print the "discard" message

```

```

21 noChange:
22     movq    $discardMsg, %rbx # point to first char
creates a problem for the assembler when it needs to translate the
3      jne    noChange # no, there is no change

```

instruction on line 3. (Don't forget that assembly language is line oriented; translation is done one line at a time.) When this code sequence is executed, the immediately previous instruction (`cmpb '$y', response`) compares the byte stored at location `response` with the character '`y`'. If they are not equal, i.e., a '`y`' is not stored at location `response`, the `jne` instruction causes program flow to jump to location `noChange`. In order to accomplish this action, the translation of this instruction (the machine code) must include a numerical value that specifies how far to jump. That is, it must include the distance, in number of bytes, between the `jne` instruction and the memory location labeled `noChange` on line 23. In order to compute this distance, the assembler must determine the address that corresponds to the label `noChange` when it translates this instruction, but the assembler has not even encountered the `noChange` label, much less determined its corresponding address.

The simplest solution is to use a two-pass assembler:

1. The first pass builds a symbol table, which provides an address for each memory label.
2. The second pass performs the actual translation into machine language, consulting the symbol table for numeric values of the symbols.

Algorithm 7.1 is a highly simplified description of how the first pass of an assembler works.

---

#### **Algorithm 7.1:** First pass of a two-pass assembler.

---

**Data:** `SymbolTable` with each entry a `Symbol/Number` pair

**Data:** `LocationCounter`

```

1 LocationCounter ← 0;
2 get first line of source code;
3 while more lines do
4   if line has label then
5     SymbolTable.Symbol ← label;
6     SymbolTable.Number ← LocationCounter;
7   determine number of bytes required by line when assembled;
8   LocationCounter ← LocationCounter + number of bytes;
9   get next line of source code;

```

---

The symbol table is carried from the first pass to the second. The second pass also consults a table of operation codes, which provides the machine code corresponding to each instruction mnemonic. A highly simplified description of the second pass is given in Algorithm 7.2.

---

**Algorithm 7.2:** Second pass of a two-pass assembler.

```

given: SymbolTable from Pass One
given: Op – CodeTable
Data: LocationCounter
1 LocationCounter  $\leftarrow$  0;
2 get first line of source code;
3 while more lines do
4   if line is instruction then
5     find machine code from Op-Code Table;
6     find symbol value from SymbolTable;
7     assemble instruction into machine code;
8   else
9     carry out directive;
10  write machine code to object file;
11  determine number of bytes used;
12  LocationCounter  $\leftarrow$  LocationCounter + number of bytes;
13  get next line of source code;

```

---

### 7.3.2 Linkers

Look again at the code sequence above. On line 14 there is the instruction:

call      write

This call to the `write` function is a reference to a memory label outside the file being assembled. Thus, the assembler has no way to determine the address of `write` for the symbol table during the first pass. The only thing the assembler can do during the second pass is to leave enough memory space for the address of `write` when it assembles this instruction. The actual address will have to be filled in later in order to create the entire program. Filling in these references to external memory locations is the job of the linker program.

The algorithm for linking functions together is very similar to that of the assembler. The same forward reference problem exists. Again, the simplest solution is to use a two-pass linker program.

The highly simplified algorithm in Algorithms 7.3 and 7.4 also provide for loading the entire program into memory. The functions are linked together as they are loaded. In practice, this is seldom done. For example, the GNU linker, `ld`, does not load the program into memory. Instead, it creates another machine language file — the executable program. The executable program file contains all the functions of the program with all the cross-function memory references resolved. Thus `ld` is a *link editor* program.

Getting even more realistic, many of the functions used by a program are not even included in the executable program file. They are loaded as required when the program is executing. The link editor program must provide dynamic links for the executable program file.

However, you can get the general idea of linking separately assembled (or compiled) functions together by studying the algorithms in Algorithms 7.3 and 7.4. In particular, notice that the assembler (or compiler) must include other information in addition to machine code in the object file. The additional information includes:

1. The name of the function.
2. The name of each external memory reference.

3. The location relative to the beginning of the function where the external memory reference is made.

**Algorithm 7.3:** First pass of a two-pass linker.

**Data:** *GlobalSymbolTable* with each entry a *Symbol/Number* pair  
**Data:** *LocationCounter*

```

1 LocationCounter  $\leftarrow$  0;
2 open first object file;
3 while more object files do
4   GlobalSymbolTable.Symbol  $\leftarrow$  function name;
5   GlobalSymbolTable.Number  $\leftarrow$  LocationCounter;
6   determine number of bytes required by the function;
7   LocationCounter  $\leftarrow$  LocationCounter + number of bytes;
8   open next object file;
```

**Algorithm 7.4:** Second pass of a two-pass linker.

**Data:** *MemoryPointer*  
**given:** *GlobalSymbolTable* from Pass One

```

1 MemoryPointer  $\leftarrow$  address from OS;
2 open first object file;
3 while more object files do
4   while more machine code do
5     CodeByte  $\leftarrow$  read byte of code from object file;
6     *MemoryPointer  $\leftarrow$  CodeByte;
7     MemoryPointer  $\leftarrow$  MemoryPointer + 1;
8   while more external memory references do
9     get value corresponding to reference from GlobalSymbolTable;
10    determine where value should be stored;
11    store value in code that was just loaded;
12   open next object file;
```

## 7.4 Creating a Program in Assembly Language

Since we are concerned with assembly language in this book, let us go through the steps of creating a program for the assembly language source code in Listing 7.5. Figure 7.1 is a screen shot of what I did with my typing in **boldface**. The notation I use here assumes that I am doing this for a class named CS 252, and my instructor has specified that each project should be submitted in a directory named *CS252lastNameNN*, where *lastName* is the student's surname and *NN* is the project number.<sup>2</sup> I have appended .0 to the project folder name for my own use. As I develop my project, subsequent versions will be numbered .1, .2, ....

<sup>2</sup>CS 252 is the number of the class in which this book is used at my university, Sonoma State University.

---

```
bob$ mkdir CS252plantz01.0
bob$ cd CS252plantz01.0/
bob$ ls
bob$ pwd /home/bob/CS252/CS252plantz01.0
bob$ gedit doNothingProg.s
```

*This is where I used gedit to enter the program from Listing 7.5, saved the program, and quit gedit.*

```
bob$ ls
doNothingProg.s
bob$ as -gstabs -o doNothingProg.o doNothingProg.s
bob$ ls
doNothingProg.o doNothingProg.s
bob$ gcc -o doNothingProg doNothingProg.o
bob$ ls
doNothingProg doNothingProg.o doNothingProg.s
bob$ ./doNothingProg
bob$
```

---

Figure 7.1: Screen shot of the creation of a program in assembly language.

Let us go through the steps in Figure 7.1 one line at a time, explaining each line.

```
bob$ mkdir CS252plantz01.0
```

*I create a directory named “CS252plantz01.0.” All the files that you create for each program should be kept in a separate directory only for that program.*

```
bob$ cd CS252plantz01.0/
```

*I make the newly created subdirectory the current working directory.*

```
bob$ ls
bob$ pwd /home/bob/CS252/CS252plantz01.0
```

*These two commands show that the new subdirectory is empty and where my current working directory is located within the file hierarchy.*

```
bob$ gedit doNothingProg.s
```

*This starts up the gedit program and creates a new file named “doNothingProg.s.” I typed my program using the text editor, saved the file, and quit the text editor. You may use any text editor. Avoid using a word processor because it will probably add formatting codes to the file.*

```
bob$ ls
doNothingProg.s
```

*This shows that I have created the file, doNothingProg.s.*

```
bob$ as -gstabs -o doNothingProg.o doNothingProg.s
bob$ ls
doNothingProg.o doNothingProg.s
```

*On the first line, I invoke the assembler, as. The -gstabs option directs the assembler to include debugging information with the output file. We will very definitely make use of the debugger! The -o option is followed by the name of the output (object) file. You should always use the same name as the source file, but with the .o extension. The second command simply shows the new file that has been created in my directory.*

```
bob$ gcc -o doNothingProg doNothingProg.o
bob$ ls
doNothingProg doNothingProg.o doNothingProg.s
```

*Next I link the object file. Even though there is only one object file, this step is required in order to bring in the GNU/Linux libraries needed to create an executable program. As in as, the -o option is used to specify the name of a file. In the linking case, this will be the name of the final product of our efforts.*

**Note:** The linker program is actually ld. The problem with using it directly, for example,  
`ld -o doNothingProg doNothingProg.o *** DOES NOT WORK ***`

*is that you must also explicitly specify all the libraries that are used. By using gcc for the linking, the appropriate libraries are automatically included in the linking.*

```
bob$ ./doNothingProg
bob$
```

*Finally, I execute the program (which does nothing).*

## 7.5 Instructions Introduced Thus Far

This summary shows the assembly language instructions introduced thus far in the book. It should be sufficient for doing the exercises in the current chapter. The page number where the instruction is explained in more detail, which may be in a subsequent chapter, is also given. The summary will be repeated and updated, as appropriate, at the end of each succeeding chapter in the book. This book provides only an introduction to the usage of each instruction. You need to consult the manuals ([2] – [6], [14] – [18]) in order to learn all the possible uses of the instructions.

### 7.5.1 Instructions

*data movement:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
movs	\$imm/%reg	%reg/mem	move	156
popw		%reg/mem	pop from stack	181
pushw	\$imm/%reg/mem		push onto stack	181

*s = b, w, l, q; w = l, q*

*arithmetic/logic:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
cmps	\$imm/%reg	%reg/mem	compare	237
incs	%reg/mem		increment	248

*s = b, w, l, q*

*program flow control:*

<b>opcode</b>	<b>location</b>	<b>action</b>	<b>see page:</b>
call	label	call function	173
je	label	jump equal	239
jmp	label	jump	241
jne	label	jump not equal	239
ret		return from function	192

## 7.6 Exercises

The functions you are asked to write in these exercises are not complete programs. You can check that you have written a valid function by writing a `main` function in C that calls the function you have written in assembly language. Compile the `main` function with the `-c` option so that you get the corresponding object (`.o`) file. Assemble your assembly language file. Make sure that you specify the debugging options when compiling/assembling. Use the linking phase of `gcc` to link the `.o` files together. Run your program under `gdb` and set a breakpoint in your assembly language function. (Hint: you can specify the source file name in `gdb` commands.) Now you can verify that your assembly language function is being called. If the function returns a value, you can print that value in the `main` function using `printf`.

**7-1** (§7.2) Write the C function:

```
/* f.c */
int f(void) {
    return 0;
}
```

in assembly language. Make sure that it assembles with no errors. Use the `-S` option to compile `f.c` and compare `gcc`'s assembly language with yours.

**7-2** (§7.2) Write the C function:

```
/* g.c */
void g(void) {
}
```

in assembly language. Make sure that it assembles with no errors. Use the `-S` option to compile `g.c` and compare `gcc`'s assembly language with yours.

**7-3** (§7.2) Write the C function:

```
/* h.c */
int h(void) {
    return 123;
}
```

in assembly language. Make sure that it assembles with no errors. Use the `-S` option to compile `h.c` and compare `gcc`'s assembly language with yours.

**7-4** (§7.2) Write three assembly language functions that do nothing but return an integer. They should each return different, non-zero, integers. Write a C `main` function to test your assembly language functions. The `main` function should capture each of the return values and display them using `printf`.

**7-5** (§7.2) Write three assembly language functions that do nothing but return a character. They should each return different characters. Write a C `main` function to test your assembly language functions. The `main` function should capture each of the return values and display them using `printf`.

**7-6** (§7.2, §6.5) Write an assembly language function that returns four characters. The return value is **always** in the `eax` register in our environment, so you can store four characters in it. The easiest way to do this is to determine the hexadecimal value for each character, then combine them so you can store one 32-bit hexadecimal value in `eax`.

Write a C `main` function to test your assembly language function. The `main` function should capture the return values and display them using the `write` system call.

Explain the order in which they are displayed.

# Chapter 8

# Program Data - Input, Store, Output

Most programs follow a similar pattern:

1. Read data from an input device, such as the keyboard, a disk file, the internet, etc., into main memory.
2. Load data from main memory into CPU registers.
3. Perform arithmetic/logic operations on the data.
4. Store the results in main memory.
5. Write the results to an output device, such as the screen, a disk file, audio speakers, etc.

In this chapter you will learn how to call functions that can read input from the keyboard, allocate memory for storing data, and write output to the screen.

## 8.1 Calling write in 64-bit Mode

We start with a program that has no input. It simply writes constant data to the screen — the “Hello World” program.

We will use the *C system call function* `write` to display the text on the screen and show how to call it in assembly language. As we saw in Section 2.8 (page 23) the `write` function requires three arguments. Reading the argument list from left to right in Listing 8.1:

1. `STDOUT_FILENO` is the file descriptor of standard out, normally the screen. This symbolic name is defined in the `unistd.h` header file.
2. Although the C syntax allows a programmer to place the text string here, only its address is passed to `write`, not the entire string.
3. The programmer has counted the number of characters in the text string to write to `STDOUT_FILENO`.

---

```

1  /*
2   * helloWorld2.c
3   *
4   * "hello world" program using the write() system call.
5   * Bob Plantz - 8 June 2009
6   */
7 #include <unistd.h>
8
9 int main(void)
10 {
11
12     write(STDOUT_FILENO, "Hello world.\n", 13);
13
14     return 0;
15 }
```

Listing 8.1: “Hello world” program using the `write` system call function (C).

This program uses only constant data — the text string “Hello world.” Constant data used by a program is part of the program itself and is not changed by the program.

Looking at the compiler-generated assembly language in Listing 8.2, the constant data appears on line 4, as indicated by the comment added on that line. Comments have also been added on lines 11 – 14 to explain the argument set up for the call to `write`.

---

```

1      .file    "helloWorld2.c"
2      .section    .rodata
3 .LC0:
4          .string   "Hello world.\n"
5      .text
6      .globl    main
7      .type     main, @function
8 main:
9      pushq    %rbp
10     movq    %rsp, %rbp
11     movl    $13, %edx      # third argument
12     movl    $.LC0, %esi      # second argument
13     movl    $1, %edi      # first argument
14     call     write
15     movl    $0, %eax
16     popq    %rbp
17     ret
18     .size    main, .-main
19     .ident   "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
20     .section    .note.GNU-stack, "",@progbits
```

Listing 8.2: “Hello world” program using the `write` system call function (gcc assembly language).

Data can only be located in one of two places in a computer:

- in memory, or
- in a CPU register.

(We are ignoring the case of reading from an input device or writing to an output device here.) Recall from the discussion of memory segments on page 152 that the Linux kernel uses different memory segments for the various parts of a program. The directive on line 2,

```
2      .section      .rodata
```

uses the `.section` assembler directive to direct the assembler to store the data that follows in a “read-only data” section in the object file. Even though it begins with a ‘.’ character `.rodata` is not an assembler directive but the name of a section in an ELF file.

Your first thought is probably that the `.rodata` section should be loaded into a data segment in memory, but recall that data memory segments are read/write. Thus `.rodata` sections are mapped into a text segment, which is a read-only memory segment.

The `.string` directive on line 4,

```
3 .LC0:  
4      .string "Hello world.\n" # constant data
```

allocates enough bytes in memory to hold each of the characters in the text string, plus one for the NUL character at the end. The first byte contains the ASCII code for the character ‘H’, the second the ASCII code for ‘e’, etc. Notice that the last character in this string is ‘\n’, the newline character; it occupies only one byte of memory. So fourteen bytes of memory are allocated in the `.rodata` section in this program, and each byte is set to the corresponding ASCII code for each character in the text string. The label on line 3 provides a symbolic name for the beginning address of the text string so that the program can refer to this memory location.

The most common directives for allocating memory for data are shown in Table 8.1. If these are used in the `.rodata` section, the values can only be used as constants in the

<code>[label]</code>	<code>.space</code>	<code>expression</code>	evaluates <code>expression</code> and allocates that many bytes; memory is not initialized
<code>[label]</code>	<code>.string</code>	<code>"text"</code>	initializes memory to null-terminated string
<code>[label]</code>	<code>.asciz</code>	<code>"text"</code>	same as <code>.string</code>
<code>[label]</code>	<code>.ascii</code>	<code>"text"</code>	initializes memory to the string without null
<code>[label]</code>	<code>.byte</code>	<code>expression</code>	allocates one byte and initializes it to the value of <code>expression</code>
<code>[label]</code>	<code>.word</code>	<code>expression</code>	allocates two bytes and initializes them to the value of <code>expression</code>
<code>[label]</code>	<code>.long</code>	<code>expression</code>	allocates four bytes and initializes them to the value of <code>expression</code>
<code>[label]</code>	<code>.quad</code>	<code>expression</code>	allocates eight bytes and initializes them to the value of <code>expression</code>

Table 8.1: Common assembler directives for allocating memory. The `label` is optional. program.

The assembly language instruction used to call a function is

```
call functionName
```

where `functionName` is the name of the function being called. The `call` instruction does two things:

1. The address in the `rip` register is pushed onto the call stack. (The call stack is described in Section 8.2.) Recall that the `rip` register is incremented immediately

after the instruction is fetched. Thus, when the `call` instruction is executed, the value that gets pushed onto the stack is the address of the instruction immediately following the `call` instruction. That is, the *return address* gets pushed onto the stack in this first step.

2. The address that *functionName* resolves to is placed in the `rip` register. This is the address of the function that is being called, so the next instruction to be fetched is the first instruction in the called function.

The call of the `write` function is made on line 14.

14      `call`    `write`

Before the call is made, any arguments to a function must be stored in their proper locations, as specified in the ABI [25]. Up to six arguments are passed in the general purpose registers. Reading the argument list from left to right in the C code, the order of using the registers is given in Table 8.2. If there are more than six arguments, the

Argument	Register
first	<code>rdi</code>
second	<code>rsi</code>
third	<code>rdx</code>
fourth	<code>rcx</code>
fifth	<code>r8</code>
sixth	<code>r9</code>

Table 8.2: Order of passing arguments in general purpose registers.

additional ones are pushed onto the call stack, but in right-to-left order. This will be described in Section 11.2.

Each of the three arguments to `write` in this program — the file descriptor, the address of the text string, and the number of bytes in the text string — is also a constant whose value is known when the program is first loaded into memory and is not changed by the program. The locations of these constants on lines 11 – 13,

11      `movl`    `$13, %edx`    *# third argument*  
 12      `movl`    `$.LC0, %esi`    *# second argument*  
 13      `movl`    `$1, %edi`    *# first argument*

are not as obvious. The location of the data that an instruction operates on must be specified in the instruction and its operands. The manner in which the instruction uses an operand to locate the data is called the *addressing mode*. Assembly language includes a syntax that the programmer uses to specify the addressing mode for each operand. When the assembler translates the assembly language into machine code it sets the bit pattern in the instruction to the corresponding addressing mode for each operand. Then when the CPU decodes the instruction during program execution it knows where to locate the data represented by that operand.

The simplest addressing mode is *register direct*. The syntax is to simply use the name of a register, and the data is located in the register itself.

**Register direct:** The data value is located in a CPU register.

*syntax:* the name of the register with a "%" prefix

*example:* `movl %eax, %ebx`

The instructions on lines 9 – 10,

```
9      pushq %rbp
10     movq %rsp, %rbp
```

use the register direct addressing mode for their operands. The `pushq` instruction has only one operand, and the `movq` has two.

Each of the instructions on lines 11 – 13 use the register direct addressing mode for the destination, but the source operand is the data itself. So all three instructions employ the *immediate data* addressing mode for the source.

**Immediate data:** The data value is located in memory immediately after the instruction.

This addressing mode can only be used for a source operand.

*syntax:* the data value with a “\$” prefix

*example:* `movq $0x123456789abcd, %rbx`

Although the register direct addressing mode can be used to specify either a source or destination operand, or both, the immediate data addressing mode is valid only for a source operand.

Let us consider the mechanism by which the control unit accesses the data in the immediate data addressing mode. First, we should say a few words about how a control unit executes an instruction. Although a programmer thinks of each instruction as being executed atomically, it is actually done in discrete steps by the control unit. In addition to the registers used by a programmer, the CPU contains many registers that cannot be used directly. The control unit uses these registers as “scratch paper” for temporary storage of intermediate values as it progresses through the steps of executing an instruction.

Now, recall that when the control unit fetches an instruction from memory, it automatically increments the instruction pointer (`rip`) to the next memory location immediately following the instruction it just fetched. Usually, the instruction pointer would now be pointing to the next instruction in the program. But in the case of the immediate data addressing mode, the “\$” symbol tells the assembler to store the operand at this location.

As the control unit decodes the just fetched instruction, it detects that the immediate data addressing mode has been used for the source operand. Since the instruction pointer is currently pointing to the data, it is a simple matter for the control unit to fetch it. Of course, when it does this fetch, the control unit increments the instruction pointer by the size of the data it just fetched.

Now the control unit has the source data, so it can continue executing the instruction. And when it has completed the current instruction, the instruction pointer is already pointing to the next instruction in the program.

The constants in the instructions on lines 11 and 13 are obvious. (The symbolic name “`STDOUT_FILENO`” is defined in `unistd.h` as 1.) The constant on line 12 is the label `.LC0`, which resolves to the address of this memory location. As explained above, this address will be in the `.rodata` section when the program is loaded into memory. The address is not known within the `.text` segment when the file is first compiled. The compiler leaves space for it immediately after the instruction (immediate addressing mode). Then when the address is determined during the linking phase, it is plugged in to the space left for it. The net result is that the address becomes immediate data when the program is executed.

So the following code sequence:

```

11      movl    $13, %edx      # third argument
12      movl    $.LC0, %esi      # second argument
13      movl    $1, %edi      # first argument
14      call    write

```

implements the C statement

```

13      write(STDOUT_FILENO, "Hello world.\n", 13);

```

in the original C program (Listing 8.1, page 172).

Some notes about the `write` function call:

- The characters written to the screen must be stored in memory.
- The number of bytes actually written to the screen is returned in the `eax` register. So if the current function is using `eax`, the value will be changed by the call to `write`.
- The `write` function is a C wrapper that sets up the registers for the `syscall` instruction. Unfortunately, there is no guarantee that it restores the values that were in the registers when it was called.

## 8.2 Introduction to the Call Stack

Most variables are stored on the *call stack*. Before describing how this is done, we need to understand what stacks are and how they are used.

A stack is an area of memory for storing data items together with a pointer to the “top” of the stack. Informally, you can think of a stack as being organized very much like a stack of dinner plates on a shelf. We can only access the one item at the top of the stack. There are only two fundamental operations on a stack:

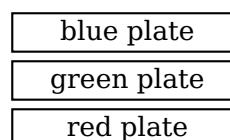
- **push *data-item*** causes the *data-item* to be placed on the top of the stack and moves the stack pointer to point to this latest item.
- **pop *location*** causes the data item on the top of the stack to be removed and placed at *location* and moves the stack pointer to point to the next item left on the stack.

Notice that a stack is a “last in, first out” (LIFO) data structure. That is, the last thing to be pushed onto the stack is the first thing to be popped off.

To illustrate the stack concept let us use our dinner plate example. Say we have three differently colored dinner plates, a red one on the dining table, a green one on the kitchen counter, and a blue one on the bedside table. Now we will stack them on the shelf in the following way:

1. push dining-table-plate
2. push kitchen-counter-plate
3. push bedside-table-plate

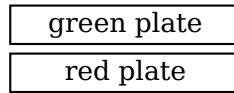
At this point, our stack looks like:



Now if we perform the operation:

1. pop kitchen-counter

we will have a blue dinner plate on our kitchen counter, and our stack will look like:



A stack must be used according to a very strict discipline:

1. Always push an item onto the stack before popping anything off.
2. Never pop more things off than you have pushed on.
3. Always pop everything off the stack.

If you have no use for the item(s) to be popped off, you may simply adjust the stack pointer. This is equivalent to discarding the items that are popped off. (Our dinner plate analogy breaks down here.)

A good way to maintain this discipline is to think of the use of parentheses in an algebraic expression. A push is analogous to a left parenthesis, and a pop is analogous to a right parenthesis. An attempt to push too many items onto a stack causes *stack overflow*. And an attempt to pop items off the stack beyond the “bottom” causes *stack underflow*.

Next we will explore how we might implement a stack in C. Our program will allocate space in memory for storing data elements and provide both a push operation and a pop operation. A simple program is shown in Listing 8.3.

```
1 /*  
2 * stack.c  
3 * implementation of push and pop stack operations in C  
4 * Bob Plantz - 7 June 2012  
5 *  
6 */  
7  
8 #include <stdio.h>  
9  
10 int theStack[500];  
11 int *stackPointer = &theStack[500];  
12  
13 /*  
14 * precondition:  
15 *     stackPointer points to data element at top of stack  
16 * postcondition:  
17 *     address in stackPointer is decremented by four  
18 *     dataValue is stored at top of stack  
19 */  
20 void push(int dataValue)  
21 {  
22     stackPointer--;  
23     *stackPointer = dataValue;  
24 }
```

```
25
26 /*  
27 * precondition:  
28 *     stackPointer points to data element at top of stack  
29 * postcondition:  
30 *     data element at top of stack is copied to *dataLocation  
31 *     address in stackPointer is incremented by four  
32 */  
33 void pop(int *dataLocation)  
34 {  
35     *dataLocation = *stackPointer;  
36     stackPointer++;  
37 }  
38  
39 int main(void)  
40 {  
41     int x = 12;  
42     int y = 34;  
43     int z = 56;  
44     printf("Start with the stack pointer at %p\n",  
45            (void *)stackPointer);  
46     printf("x = %i, y = %i, and z = %i\n", x, y, z);  
47  
48     push(x);  
49     push(y);  
50     push(z);  
51     x = 100;  
52     y = 200;  
53     z = 300;  
54     printf("push x\npush y\npush z\n");  
55     printf("Now the stack pointer is at %p\n",  
56            (void *)stackPointer);  
57     printf("Change x, y, and z:\n");  
58     printf("x = %i, y = %i, and z = %i\n", x, y, z);  
59     pop(&z);  
60     pop(&y);  
61     pop(&x);  
62  
63     printf("pop z\npop y\npop x\n");  
64     printf("And we end with the stack pointer at %p\n",  
65            (void *)stackPointer);  
66     printf("x = %i, y = %i, and z = %i\n", x, y, z);  
67  
68     return 0;  
69 }
```

---

Listing 8.3: A C implementation of a stack.

Read the code in Listing 8.3 and note the following:

- The program uses a pointer, `stackPointer`, to keep track of the data value that is currently at the top of the stack.
- The stack pointer is initialized to point to one beyond the highest array element

in the array that is allocated for the stack. Thus the stack must “grow” from high-numbered elements to low-numbered elements as items are pushed onto the stack.

- A push operation pre-decrements the stack pointer before storing an item on the stack.
- A pop operation post-increments the stack pointer after retrieving an item from the stack.

The states of the variables from the program in Listing 8.3 are shown just after the stack is initialized in Figure 8.1. Notice that the stack pointer is pointing beyond the end of the array as a result of the C statement,

```
int *stackPointer = &theStack[500];
```

The stack is “empty” at this point.

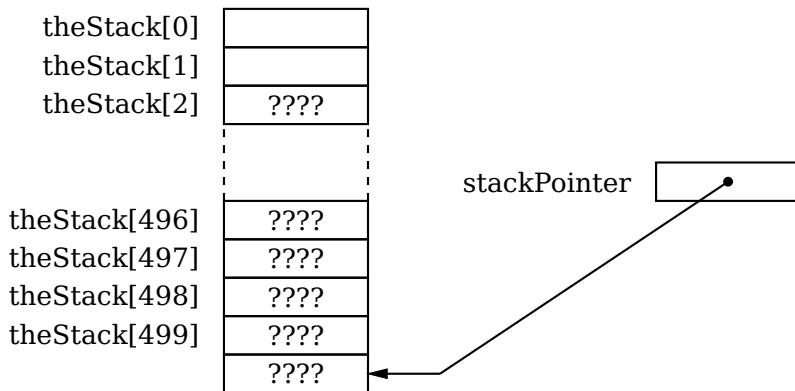


Figure 8.1: The stack in Listing 8.3 when it is first initialized. “????” means that the value in the array element is undefined.

After pushing one value onto the stack

```
push(x);
```

the stack appears as shown in Figure 8.2. Here you can see that since the push operation pre-decrements the stack pointer, the first data item to be placed on the stack is stored in a valid portion of the array.

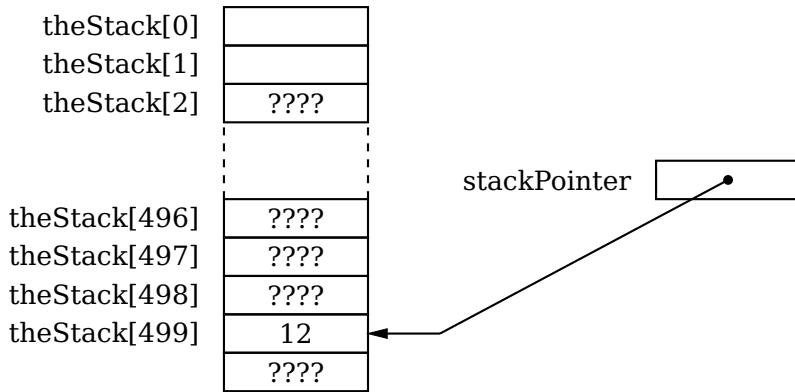


Figure 8.2: The stack with one data item on it.

After all three data items —  $x$ ,  $y$ , and  $z$  — are pushed onto the stack, it appears as shown in Figure 8.3. The stack pointer always points to the data item that is at the top of the stack. Notice that this stack is “growing” toward lower numbered elements in the array.

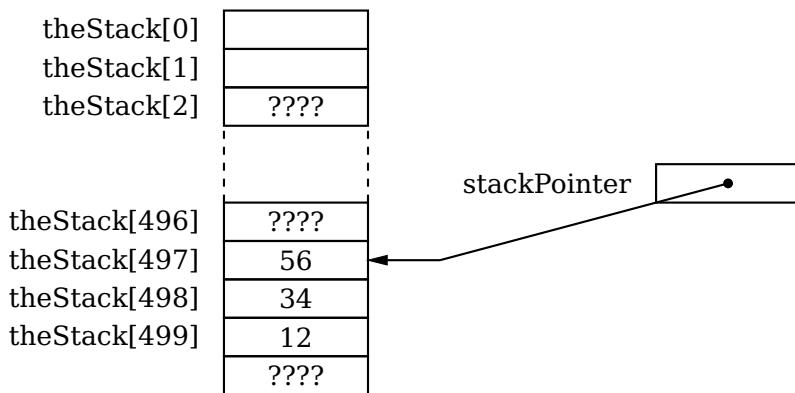


Figure 8.3: The stack with three data items on it.

After changing the values in the variables, the program in Listing 8.3 restores the original values by popping from the stack in reverse order. The state of the stack after all three pops are shown in Figure 8.4. Even though we know that the values are still stored in the array, the permissible stack operations — push and pop — will not allow us to access these values. Thus, from a programming point of view, the values are gone.

Our very simple stack in this program does not protect against stack overflow or stack underflow. Most software stack implementations also include operations to check for an empty stack and for a full stack. And many implementations include an operation for looking at, but not removing, the top element. But these are not the main features of a stack data structure, so we will not be concerned with them here.

In GNU/Linux, as with most operating systems, the call stack has already been set up for us. We do not need to worry about allocating the memory or initializing a stack pointer. When the operating system transfers control to our program, the stack is ready for us to use.

The x86-64 architecture uses the `rsp` register for the *call stack pointer*. Although you could create your own stack and stack pointer, several instructions use the `rsp` register implicitly. And all these instructions cause the stack to grow from high memory

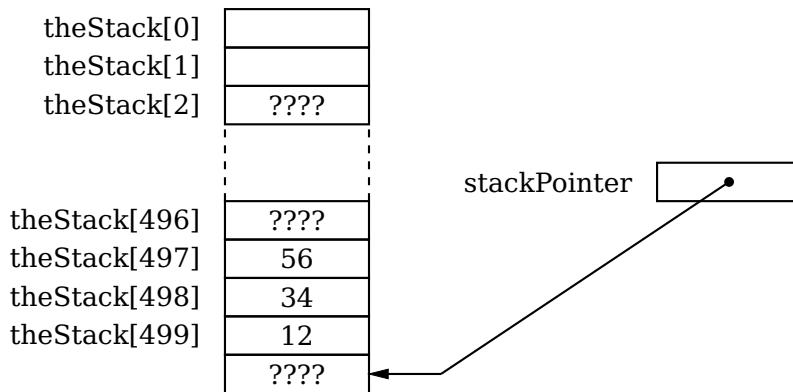


Figure 8.4: The stack after all three data items have been popped off. Even though the values are still stored in the array, it is considered a programming error to access them. The stack must be considered as “empty” when it is in this state.

addresses to low (see Exercise 8-2). Although this may seem a bit odd at first, there are some good reasons for doing it this way.

In particular, think about how you might organize things in memory. Recall that the instruction pointer (the `rip` register) is automatically incremented by the control unit as your program is executed. Programs come in vastly different sizes, so it makes sense to store the program instructions at low memory addresses. This allows maximum flexibility with respect to program size.

The stack is a dynamic structure. You do not know ahead of time how much stack space will be required by any given program as it executes. It is impossible to know how much space to allocate for the stack. So you would like to allocate as much space as possible, and to keep it as far away from the programs as possible. The solution is to start the stack at the highest address and have it grow toward lower addresses.

This is a highly simplified rationalization for implementing stacks such that they grow “downward” in memory. The organization of various program elements in memory is much more complex than the simple description given here. But this may help you to understand that there are some good reasons for what may seem to be a rather odd implementation.

The assembly language push instruction is:

```
pushq  source
```

The `pushq` instruction causes two actions:

1. The value in the `rsp` register is decremented by eight. That is, eight is subtracted from the stack pointer.
2. The eight bytes of the `source` operand are copied into memory at the new location pointed to by the (now decremented) stack pointer. The state of the operand is not changed.

The assembly language pop instruction is:

```
popq  destination
```

The `popq` instruction causes two actions:

1. The eight bytes in the memory location pointed to by the stack pointer are copied to the *destination* operand. The previous state of the operand is replaced by the value from memory.
2. The value in the `rsp` register is incremented by eight. That is, eight is added to the stack pointer.

In the Intel syntax the “q” is not appended to the instruction.

Intel®  
Syntax

<code>push source</code>	<code>pop destination</code>
--------------------------	------------------------------

The size of the operand, eight bytes, is determined by the operating system. When executing in 64-bit mode, all pushes and pops operate on 64-bit values. Unlike the `mov` instruction, you cannot push or pop 8-, 16-, or 32-bit values. This means that the address in the stack pointer (`rsp` register) will always be an integral multiple of eight.

A good example of using a stack is saving registers within a function. Recall that there is only one set of registers in the CPU. When one function calls another, the called function has no way of knowing which registers are being used by the calling function. The ABI [25] specifies that the values in registers `rbx`, `rbp`, `rsp`, and `r12 - r15` be preserved by the called function (see Table 6.4 on page 134).

The program in Listing 8.4 shows how to save and restore the values in these registers. Notice that since a stack is a LIFO structure, it is necessary to pop the values off the top of the stack in the *reverse order* from how they were pushed on.

---

```

1 # saveRegisters.s
2 # The rbx and r12 - r15 registers must be preserved by called function.
3 # Sets a bit pattern in these registers, but restores original values
4 # in the registers before returning to the OS.
5 # Bob Plantz - 8 June 2009

6
7     .text
8     .globl main
9     .type main, @function
10 main:
11     pushq %rbp      # save caller's frame pointer
12     movq %rsp, %rbp # establish our frame pointer
13
14     pushq %rbx      # "must-save" registers
15     pushq %r12
16     pushq %r13
17     pushq %r14
18     pushq %r15
19
20     movb $0x12, %bl # "use" the registers
21     movw $0xabcd, %r12w
22     movl $0x1234abcd, %r13d
23     movq $0xdcba, %r14
24     movq $0x9876, %r15
25
26     popq %r15      # restore registers
27     popq %r14

```

```

28      popq    %r13
29      popq    %r12
30      popq    %rbx
31
32      movl    $0, %eax      # return 0
33      popq    %rbp      # restore caller's frame pointer
34      ret      # back to caller

```

Listing 8.4: Save and restore the contents of the rbx and r12 – r15 registers. See Table 6.4, page 134, for the registers that should be saved/restored in a function if they are used in the function.

The problem with this technique is maintaining the address in the stack pointer at a 16-byte boundary. Another way to save/restore the registers will be given in Section 11.2.

## 8.3 Viewing the Call Stack

This seems like a good place to use `gdb` to see how a stack frame is created and used. We will use the program in Listing 8.4. I used an editor to enter the code then assembled and linked it.

When using `gdb` to examine programs written in assembly language, another variant of the `break` command may be helpful. The version of `gdb` I used for this book skips over the function prologue. To cause `gdb` to break at the first instruction of a function, the following form should be used:

- `break *label` — sets a breakpoint at the specified label in the current source file. Notice that you cannot specify a label in another source file. Control will return to `gdb` when the label is encountered.

My typing is **boldface**.

```
$ gdb ./saveRegisters

GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/bob/my_book_working/progs/chap09/assignment2...done
(gdb) li

1 # saveRegisters.s
2 # The rbx and r12 - r15 registers must be preserved by called function.
3 # Sets a bit pattern in these registers, but restores original values
4 # in the registers before returning to the OS.
5 # Bob Plantz - 8 June 2009
6
7      .text
```

```
8      .globl  main
9      .type   main, @function
10 main:
(gdb)
11     pushq   %rbp      # save caller's frame pointer
12     movq   %rsp, %rbp  # establish our frame pointer
13
14     pushq   %rbx      # "must-save" registers
15     pushq   %r12
16     pushq   %r13
17     pushq   %r14
18     pushq   %r15
19
20     movb   $0x12, %bl  # "use" the registers
```

*I use the li command to list enough of the program to see where I should set the first breakpoints.*

```
(gdb) br *main
```

```
Breakpoint 1 at 0x4004cc: file saveRegisters.s, line 11.
```

```
(gdb) br 20
```

```
Breakpoint 2 at 0x4004d9: file saveRegisters.s, line 20.
```

*I set the first breakpoint on the first instruction in the function. Notice that the label is on line 10, but it applies to the instruction on line 11. The second breakpoint is after all the registers have been saved on the call stack.*

```
(gdb) run
```

```
Starting program: /home/bob/my_book_working/progs/chap08/saveRegisters
```

```
Breakpoint 1, main () at saveRegisters.s:11
```

```
11     pushq   %rbp      # save caller's frame pointer
```

*I run the program, it breaks at the first breakpoint, and I can display the registers.*

```
(gdb) i r rsp rbp rbx r12 r13 r14 r15 rip
```

rsp	0x7fffffff068	0x7fffffff068
rbp	0x0	0x0
rbx	0x0	0
r12	0x4003c0	4195264
r13	0x7fffffff140	140737488347456
r14	0x0	0
r15	0x0	0
rip	0x4004cc	0x4004cc <main>

*I use the i r (info registers) command to display the contents of the registers that are used in this program. The numbers in the right-hand column show the decimal equivalent of the bit patterns for some of the registers. If you replicate this example (a good thing to do) you will probably get different values in your registers.*

Next I want to follow how the stack changes as the program executes. This is a little tricky. The stack grows toward lower addresses, but gdb displays memory from low to high addresses. So I need to display the area of memory that the stack will grow into in order to see how this program changes it.

```
(gdb) x/7xg 0x7fffffff068-6*8
```

```
0x7fffffff038: 0x00000000000400510 0x00000000000000000000  
0x7fffffff048: 0x000000000004003c0 0x00007fffffff140  
0x7fffffff058: 0x00000000000000000000000000000000  
0x7fffffff068: 0x00007ffff7a3e76d
```

Six 64-bit registers will be pushed onto the stack. Since there are eight bytes in each register, I start the display of the stack memory at the current stack pointer (in the `rsp` register) minus  $6*8$  bytes. (Use the help `x` command if you forget the syntax for the examine memory command.) By displaying seven 64-bit values, I can see the value that was pushed onto the stack just before this function was called.

```
(gdb) cont
```

Continuing.

```
Breakpoint 2, main () at saveRegisters.s:20  
20      movb    $0x12, %bl  # "use" the registers
```

```
(gdb) x/7xg 0x7fffffff068-6*8
```

```
0x7fffffff038: 0x0000000000000000 0x0000000000000000  
0x7fffffff048: 0x00007fffffff140 0x00000000004003c0  
0x7fffffff058: 0x0000000000000000 0x0000000000000000  
0x7fffffff068: 0x00007ffff7a3e76d
```

When I continue, the program stops at the next breakpoint. I examine the memory that the stack is growing into and can see that the register contents were saved on the stack.

```
(gdb) i r rsp rbp rbx r12 r13 r14 r15 rip
```

```
rsp      0x7fffffff038 0x7fffffff038  
rbp      0x7fffffff060 0x7fffffff060  
rbx      0x0 0  
r12      0x4003c0 4195264  
r13      0x7fffffff140 140737488347456  
r14      0x0 0  
r15      0x0 0  
rip      0x4004d9 0x4004d9 <main+13>
```

A display of the registers shows that their contents have not changed, except the stack pointer (`rsp`) and the instruction pointer (`rip`). Note that the stack pointer is now pointing to the top of the stack area used by this function.

```
(gdb) li
```

```
15      pushq  %r12
16      pushq  %r13
17      pushq  %r14
18      pushq  %r15
19
20      movb   $0x12, %bl  # "use" the registers
21      movw   $0xabcd, %r12w
22      movl   $0x1234abcd, %r13d
23      movq   $0xdcba, %r14
24      movq   $0x9876, %r15

(gdb)

25
26      popq  %r15          # restore registers
27      popq  %r14
28      popq  %r13
29      popq  %r12
30      popq  %rbx
31
32      movl   $0, %eax     # return 0
33      popq  %rbp          # restore caller's frame pointer
34      ret                 # back to caller
```

(gdb) **br 26**

Breakpoint 3 at 0x4004f4: file saveRegisters.s, line 26.

(gdb) **br 32**

Breakpoint 4 at 0x4004fd: file saveRegisters.s, line 32.

(gdb) **cont**

Continuing.

```
Breakpoint 3, main () at saveRegisters.s:26
26      popq  %r15          # restore registers
```

*The li command helps me decide where to set my next two breakpoints, and I continue to the next breakpoint.*

(gdb) **i r rsp rbp rbx r12 r13 r14 r15 rip**

```
rsp      0x7fffffff038 0x7fffffff038
rbp      0x7fffffff060 0x7fffffff060
rbx      0x12 18
r12      0x40abcd 4238285
r13      0x1234abcd 305441741
r14      0xdcba 56506
r15      0x9876 39030
rip      0x4004f4 0x4004f4 <main+40>
```

*Now we can see that the registers that were saved on the stack have been changed.*

(gdb) **cont**

Continuing.

```
Breakpoint 4, main () at saveRegisters.s:32
32          movl    $0, %eax      # return 0
(gdb) i r rsp rbp rbx r12 r13 r14 r15 rip
rsp          0x7fffffff060 0x7fffffff060
rbp          0x7fffffff060 0x7fffffff060
rbx          0x0 0
r12          0x4003c0 4195264
r13          0x7fffffff140 140737488347456
r14          0x0 0
r15          0x0 0
rip          0x4004fd 0x4004fd <main+49>
```

*Continuing on to the next breakpoint and displaying the registers shows that the general purpose registers that were used — rbx, r12, r13, r14, and r15 — have been restored.*

```
(gdb) si
33          popq    %rbp      # restore caller's frame pointer
(gdb)
main () at saveRegisters.s:34
34          ret      # back to caller
(gdb) i r rsp rbp rbx r12 r13 r14 r15 rip
rsp          0x7fffffff068 0x7fffffff068
rbp          0x0 0x0
rbx          0x0 0
r12          0x4003c0 4195264
r13          0x7fffffff140 140737488347456
r14          0x0 0
r15          0x0 0
rip          0x400503 0x400503 <main+55>
```

*Two single steps brings us to the last instruction, ret, which will return to the operating system. A display of the registers shows that they have been restored to the values they had when this function first started. Of course, the instruction pointer (rip) has changed.*

```
(gdb) cont
```

Continuing.

[Inferior 1 (process 3102) exited normally]

*Finally, I use the continue command (cont) to run the program out to its end. Note: If you use the si command to single step beyond the ret instruction at the end of the main function, gdb will dutifully take you through the system libraries. At best, this is a waste of time.*

```
(gdb) q
$
```

*And, of course, I have to tell gdb to quit.*

## 8.4 Local Variables on the Call Stack

Now we see that we can store values on the stack by pushing them, and that the push operation *decreases* the value in the stack pointer register, `rsp`. In other words, allocating variables on the call stack involves *subtracting* a value from the stack pointer. Similarly, *deallocating* variables from the call stack involves *adding* a value to the stack pointer.

From this it follows that we can create local variables on the call stack by simply *subtracting the number of bytes required by each variable* from the stack pointer. This does not store any data in the variables, it simply sets aside memory that we can use. (Perhaps you have experienced the error of forgetting to initialize a local variable in C!)

Next, we have to figure out a way to access this reserved data area on the call stack. Notice that there are no labels in this area of memory. So we cannot directly use a name like we did when accessing memory in the `.data` segment.

We could use the `popl` and `pushl` instructions to store data in this area. For example,

```
popl    %eax
movl    $0, %eax
pushl    %eax
```

could be used to store zero in a variable. But this technique would obviously be very tedious, and any changes made to your code would almost certainly lead to a great deal of debugging. For example, can you figure out the reason I had to do a pop before pushing the value onto the stack? (Recall that the four bytes have already been reserved on the stack.)

At first, it may seem tempting to use the stack pointer, `rsp`, as the reference pointer. But this creates complications if we wish to use the stack within the function.

A better technique would be to maintain another pointer to the local variable area on the stack. If we do not change this pointer throughout the function, we can always use the *base register plus offset* addressing mode to directly access any of the local variables. The syntax is:

*offset(register\_name)*

Intel® Syntax	[register_name + offset]
------------------	--------------------------

When it is zero, the offset is not required.

**base register plus offset:** The data value is located in memory. The address of the memory location is the sum of a value in a register plus an offset value, which can be an 8-, 16- or 32-bit signed integer.

*syntax:* place parentheses around the register name with the offset value immediately before the left parenthesis.

*examples:* -8(%rbp); (%rsi); 12(%rax)

Intel® Syntax	[rbp - 8]; [rsi]; [rax + 12]
------------------	------------------------------

The appropriate register for implementing this is the frame pointer, `rbp`.

When a function is called, the calling function begins the process of creating an area on the stack, called the *stack frame*. Any arguments that need to be passed on the call stack are first pushed onto it, as described in Section 11.2. Then the `call` instruction pushes the return address onto the call stack (page 173).

The first thing that the called function must do is to complete the creation of the stack frame. The function prologue, first introduced in Section 7.2 (page 148), performs the following actions at the very beginning of each function:

1. Save the caller's value in the frame pointer on the stack.
2. Copy the current value in the stack pointer to the frame pointer.
3. Subtract a value from the stack pointer to allow for the local variables.

Once the function prologue has completed the stack frame, we observe that:

- The local variables are located in an area of the call stack - between the addresses in the `rsp` and `rbp` registers.
- The `rbp` register is a pointer to the bottom (the numerically highest address) of the local variable area.
- The remaining area of the stack can be accessed using the stack pointer (`rsp`) as always.

Notice that each local variable is located at some fixed offset from the base register, `rbp`. In fact, it's a negative offset.

Listing 8.5 is the compiler-generated assembly language for the program in Listing 2.4 (page 24). Comments have been added to explain the parts of the code being discussed here.

---

```

1      .file   "echoChar1.c"
2      .section     .rodata
3 .LC0:    .string  "Enter one character: "
4 .LC1:    .string  "You entered: "
5 .text
6     .globl  main
7     .type   main, @function
8
9 main:
10    pushq   %rbp          # save caller's frame pointer
11    movq    %rsp, %rbp    # establish our frame pointer
12    subq    $16, %rsp     # space for local variable
13    movl    $21, %edx     # 21 characters
14    movl    $.LC0, %esi    # address of "Enter ... "
15    movl    $1, %edi      # STDOUT_FILENO
16    call    write
17    leaq    -1(%rbp), %rax # address of aLetter var.
18    movl    $1, %edx      # 1 character
19    movq    %rax, %rsi    # address in correct reg.
20    movl    $0, %edi      # STDIN_FILENO
21    call    read
22    movl    $13, %edx     # 13 characters
23    movl    $.LC1, %esi    # address of "You ... "
24    movl    $1, %edi      # STDOUT_FILENO
25    call    write
26    leaq    -1(%rbp), %rax # address of aLetter var
27    movl    $1, %edx      # 1 character
28    movq    %rax, %rsi    # address in correct reg.
29

```

```

30      movl    $1, %edi      # STDOUT_FILENO
31      call    write
32      movl    $0, %eax      # return 0;
33      leave
34      ret                 # back to caller
35      .size   main, .-main
36      .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
37      .section .note.GNU-stack,"",@progbits

```

Listing 8.5: Echoing characters entered from the keyboard (gcc assembly language).  
Comments added. Refer to Listing 2.4 for the original C version.

The function begins by pushing a copy of the caller's frame pointer (in the `rbp` register) onto the call stack, thus saving it. Next it sets the frame pointer for this register at the current top of the stack. These two actions establish a reference point to the stack frame for this function.

Next the program allocates sixteen bytes on the stack for the local variable, thus growing the stack frame by sixteen bytes. It may seem wasteful to set aside so much memory since the only variable in this program requires only one byte of memory, but the ABI [25] specifies that the stack pointer (`rsp`) should be on a sixteen-byte address boundary before calling another function. The easiest way to comply with this specification is to allocate memory for local variables in multiples of sixteen.

Figure 8.5 shows the state of the stack just after the prologue has been executed. The return address to the calling function is safely stored on the stack, followed by the

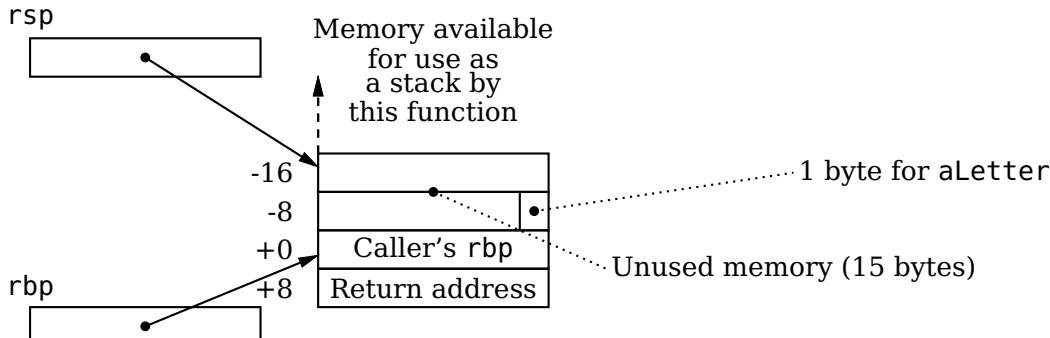


Figure 8.5: Local variables in the program from Listing 8.5 are allocated on the stack. Numbers on the left are offsets from the address in the frame pointer (`rbp` register).

caller's frame pointer value. The stack pointer (`rsp`) has been moved up the stack to allow memory for the local variable. If this function needs to push data onto the stack, such activity will not interfere with the local variable, the caller's frame pointer value, nor the return address. The frame pointer (`rbp`) provides a reference point for accessing the local variable.

**IMPORTANT:** The space for the local variables must be allocated *immediately after establishing the frame pointer*. Any other use of the stack within the function, e.g., saving registers, must be done *after* allocating space for local variables.

Most of the code in the body of the function is already familiar to you, but the instruction that loads the address of the local variable, `aString` into the `rax` register:

18        **leaq**    -1(%rbp), %rax # address of aLetter var.

is new. It uses the base register plus offset addressing mode for the source. We can see from the instruction on line 18 that the aString variable is located one byte negative from the address in the rbp register. Since the call stack grows toward negative addresses, this is the next available byte in this function's stack frame.

As with the write function, the second argument to the read function must be the address of a variable. However, the address of aString cannot be known when the program is compiled and linked because it is the address of a variable that exists in the stack frame. There is no way for the compiler or linker to know where this function's stack frame will be in memory when it is called. The address of the variable must be computed at run time.

Each instruction that accesses a stack frame variable must compute the variable's address, which is called the *effective address*. The instruction for computing addresses is *load effective address* — leal for 32-bit and lea for 64-bit addresses. The syntax of the lea instruction is

**leaw** source, %register

where w = l for 32-bit, q for 64-bit.

Intel® Syntax		<b>lea</b> register, source
------------------	--	-----------------------------

The source operand must be a memory location. The lea instruction computes the effective address of the source operand and stores that address in the destination register. So the instruction

**leaq** -1(%rbp), %rax

takes the value in rbp (the base address of this function's stack frame), adds -1 to it, and stores this sum in rax. Now rax contains the address of the variable aLetter. (The address still needs to be moved to rsi for the call to the read function.)

So the following code sequence:

```
18        leaq    -1(%rbp), %rax # address of aLetter var.
19        movl    $1, %edx        # 1 character
20        movq    %rax, %rsi      # address in correct reg.
21        movl    $0, %edi        # STDIN_FILENO
22        call    read
```

implements the C statement

```
14        read(STDIN_FILENO, &aLetter, 1);                                  // one character
```

in the original C program (Listing 2.4, page 24). (Yes, it would have been more efficient to use rsi as the destination for the leaq instruction. Recall that this program was compiled with the -O0 option, no optimization. You can also expect this to vary across different versions of the compiler.)

Some notes about the read function call:

- The characters read from the keyboard must be stored in memory. You cannot pass the name of a cpu register to the read function.
- The number of bytes actually read from the keyboard is returned in the eax register. So if the current function is using eax, the value will be changed by the call to read.
- The read function is a C wrapper that sets up the registers for the syscall instruction. Unfortunately, there is no guarantee that it restores the values that were in the registers when it was called.

**IMPORTANT:** Since neither the write nor the read system call functions are guaranteed to restore the values in the registers, your program must save any required register values before calling either of these functions.

There is also a new instruction on line 33:

33      **leave**                          **# undo stack frame**

Just before this function exits the portion of the stack frame allocated by this function must be released and the value in the rbp register restored. The leave instruction performs the actions:

```
movq    %rbp, %rsp
popq    %rbp
```

which effectively

1. deletes the local variables
2. restores the caller's frame pointer value

After the epilogue has been executed, the stack is in the state shown in Figure 8.6. The stack pointer (rsp) points to the address that will return program flow back to the

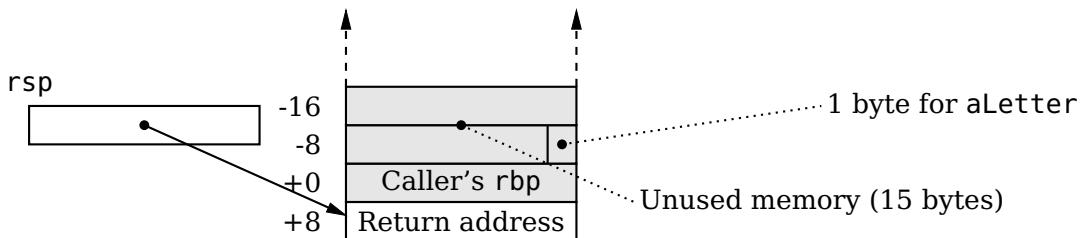


Figure 8.6: Local variable stack area in the program from Listing 8.5. Although the values in the gray area may remain they are invalid; using them at this point is a programming error.

instruction immediately after the call instruction that called this function. Although the data that was stored in the memory which is now above the stack pointer is still there, it is a violation of stack protocol to access it.

One more step remains in completing execution of this function — returning to the calling function. Since the return address is at the top of the call stack, this is a simple matter of popping the address from the top of the stack into the rip register. This requires a special instruction,

```
ret
```

which does not require any arguments.

Recall that there are two classes of local variables in C:

**Automatic** variables are created when the function is first entered. They are deleted upon exit from the function, so any value stored in them during execution of the function is lost.

**Static** variables are created when the program is first started. Any values stored in them persist throughout the lifetime of the program.

Most local variables in a function are automatic variables. General purpose registers are used for local variables whenever possible. Since there is only one set of general purpose registers, a function that is using one for a variable must be careful to save the value in the register before calling another function. Register usage is specified by the ABI [25] as shown in Table 6.4 on page 134. But you should not write code that depends upon everyone else following these recommendations, and there are only a small number of registers available for use as variables. In C/C++, most of the automatic variables are typically allocated on the call stack. As you have seen in the discussion above, they are created (automatically) in the prologue when the function first starts and are deleted in the epilogue just as it ends. Static variables must be stored in the data segment.

We are now in a position to write the echoChar program in assembly language. The program is shown in Listing 8.6.

```
1 # echoChar2.s
2 # Prompts user to enter a character, then echoes the response
3 # Bob Plantz - 8 June 2009
4
5 # Useful constants
6     .equ    STDIN,0
7     .equ    STDOUT,1
8 # Stack frame
9     .equ    aLetter,-1
10    .equ    localSize,-16
11 # Read only data
12    .section .rodata
13 prompt:
14     .string "Enter one character: "
15     .equ    promptSz,.-prompt-1
16 msg:
17     .string "You entered: "
18     .equ    msgSz,.-msg-1
19 # Code
20     .text          # switch to text section
21     .globl main
22     .type  main, @function
23 main:
24     pushq %rbp      # save caller's frame pointer
25     movq %rsp, %rbp  # establish our frame pointer
26     addq $localSize, %rsp # for local variable
27
28     movl $promptSz, %edx # prompt size
29     movl $prompt, %esi  # address of prompt text string
30     movl $STDOUT, %edi  # standard out
31     call write         # invoke write function
32
33     movl $2, %edx      # 1 character, plus newline
34     leaq aLetter(%rbp), %rsi # place to store character
35     movl $STDIN, %edi  # standard in
36     call read          # invoke read function
37
38     movl $msgSz, %edx  # message size
39     movl $msg, %esi   # address of message text string
40     movl $STDOUT, %edi # standard out
```

```

41      call    write          # invoke write function
42
43      movl    $2, %edx       # 1 character, plus newline
44      leaq    aLetter(%rbp), %rsi # place where character stored
45      movl    $STDOUT, %edi   # standard out
46      call    write          # invoke write function
47
48      movl    $0, %eax       # return 0
49      movq    %rbp, %rsp     # delete local variables
50      popq    %rbp          # restore caller's frame pointer
51      ret                 # back to calling function

```

Listing 8.6: Echoing characters entered from the keyboard (programmer assembly language).

This program introduces another assembler directive (lines 6,7,9,10,15,18):

`.equ name, expression`

The `.equ` directive evaluates the *expression* and sets the *name* equivalent to it. Note that the *expression* is evaluated during assembly, not during program execution. In essence, the *name* and its value are placed on the symbol table during the first pass of the assembler. During the second pass, wherever the programmer has used “*name*” the assembler substitutes the number that the *expression* evaluated to during the first pass.

You see an example on line 9 of Listing 8.6:

```
9      .equ    aLetter, -1
```

In this case the expression is simply -1. Then when the symbol is used on line 34:

```
34     leaq    aLetter(%rbp), %rsi # place to store character
```

the assembler substitutes -1 during the second pass, and it is exactly the same as if the programmer had written:

```
leaq    -1(%rbp), %rsi # place to store character
```

Of course, using `.equ` to provide a symbolic name makes the code much easier to read.

An example of a more complex expression is shown on lines 13 – 15:

```
13 prompt:
14     .string "Enter one character: "
15     .equ    promptSz, .-prompt-1
```

The “.” means “this address”. Recall that the `.string` directive allocates one byte for each character in the text string, plus one for the NUL character. So it has allocated 22 bytes here. The expression computes the difference between the beginning and the end of the memory allocated by `.string`, minus 1. Thus, `promptSz` is entered on the symbol table as being equivalent to 21. And on line 28 the programmer can use this symbolic name,

```
28     movl    $promptSz, %edx # prompt size
```

which is much easier than counting each of the characters by hand and writing:

```
movl    $21, %edx # prompt size
```

More importantly, the programmer can change the text string and the assembler will compute the new length and change the number in the instruction automatically. This is obviously much less prone to error.

Be careful not to mistake the .equ directive as creating a variable. It does not allocate any memory. It simply gives a symbolic name to a number you wish to use in your program, thus making your code easier to read.

Notice that the amount of memory allocated for local variables is a multiple of 16 in order to preserve optimal stack pointer alignment:

```
10      .equ    localSize, -16
```

### 8.4.1 Calling printf and scanf in 64-bit Mode

The printf function can be used to format data and write it to the screen, and the scanf function can be used to read formatted input from the keyboard. In order to see how to call these two functions in assembly language we begin with the C program in Listing 8.7.

```
1  /*
2  * echoInt1.c
3  * Reads an integer from the keyboard and echos it.
4  * Bob Plantz - 11 June 2009
5  */
6
7 #include <stdio.h>
8
9 int main(void)
10 {
11     int anInt;
12
13     printf("Enter an integer number: ");
14     scanf("%i", &anInt);
15     printf("You entered: %i\n", anInt);
16
17     return 0;
18 }
```

Listing 8.7: Calling printf and scanf to write and read formatted I/O (C).

The assembly language generated by the gcc compiler is shown in Listing 8.8. Comments have been added to explain the printf and scanf calls.

```
1      .file    "echoInt1.c"
2      .section .rodata
3 .LC0:
4      .string "Enter an integer number: "
5 .LC1:
6      .string "%i"
7 .LC2:
8      .string "You entered: %i\n"
9
10     .text
11     .globl  main
12     .type   main, @function
13
14     pushq  %rbp
15     movq  %rsp, %rbp
```

```

15      subq    $16, %rsp
16      movl    $.LC0, %edi      # address of message
17      movl    $0, %eax        # no floats
18      call    printf
19      leaq    -4(%rbp), %rax # address of anInt
20      movq    %rax, %rsi      # move to correct reg.
21      movl    $.LC1, %edi      # address of format string
22      movl    $0, %eax        # no floats
23      call    __isoc99_scanf
24      movl    -4(%rbp), %eax # copy of anInt value
25      movl    %eax, %esi      # move to correct register
26      movl    $.LC2, %edi      # address of format string
27      movl    $0, %eax        # no floats
28      call    printf
29      movl    $0, %eax
30      leave
31      ret
32      .size   main, .-main
33      .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
34      .section .note.GNU-stack,"",@progbits

```

Listing 8.8: Calling `printf` and `scanf` to write and read formatted I/O (gcc assembly language).

The first call to `printf` passes only one argument. However, on line 17 in Listing 8.8 0 is passed in `eax`:

```

16      movl    $.LC0, %edi      # address of message
17      movl    $0, %eax        # no floats
18      call    printf

```

The `eax` register is not listed as being used for passing arguments (see Section 8.1).

Both `printf` and `scanf` can take a variable number of arguments. The ABI [25] specifies that the total number of arguments passed in SSE registers must be passed in `rax`. As you will learn in Section 14.5, the SSE registers are used for passing floats in 64-bit mode. Since no float arguments are being passed in this call, `rax` must be set to 0. Recall that setting `eax` to 0 also sets the high-order bits of `rax` to 0 (Table 7.1, page 157).

The call to `scanf` on line 14 in the C version passes two arguments:

```
scanf("%i", &anInt);
```

That call is implemented in assembly language on lines 19 – 23 in Listing 8.8:

```

19      leaq    -4(%rbp), %rax # address of anInt
20      movq    %rax, %rsi      # move to correct reg.
21      movl    $.LC1, %edi      # address of format string
22      movl    $0, %eax        # no floats
23      call    __isoc99_scanf

```

Again, we see that the `eax` register must be set to 0 because there are no float arguments.

The program written in assembly language (Listing 8.9) is easier to read because the programmer has used symbolic names for the constants and the stack variable.

---

```

1 # echoInt2.s
2 # Prompts user to enter an integer, then echoes the response
3 # Bob Plantz -- 11 June 2009

```

```

4
5 # Stack frame
6     .equ    anInt,-4
7     .equ    localSize,-16
8 # Read only data
9     .section .rodata
10 prompt:
11     .string "Enter an integer number: "
12 scanFormat:
13     .string "%i"
14 printFormat:
15     .string "You entered: %i\n"
16 # Code
17     .text          # switch to text section
18     .globl main
19     .type main, @function
20 main:
21     pushq %rbp      # save caller's frame pointer
22     movq %rsp, %rbp # establish our frame pointer
23     addq $localSize, %rsp # for local variable
24
25     movl $prompt, %edi # address of prompt text string
26     movq $0, %rax      # no floating point args.
27     call printf        # invoke printf function
28
29     leaq anInt(%rbp), %rsi # place to store integer
30     movl $scanFormat, %edi # address of scanf format string
31     movq $0, %rax      # no floating point args.
32     call scanf          # invoke scanf function
33
34     movl anInt(%rbp), %esi # the integer
35     movl $printFormat, %edi # address of printf text string
36     movq $0, %rax      # no floating point args.
37     call printf        # invoke printf function
38
39     movl $0, %eax      # return 0
40     movq %rbp, %rsp    # delete local variables
41     popq %rbp          # restore caller's frame pointer
42     ret                # back to calling function

```

Listing 8.9: Calling `printf` and `scanf` to write and read formatted I/O (programmer assembly language).

## 8.5 Designing the Local Variable Portion of the Call Stack

When designing a function in assembly language, you need to determine where each local variable will be located in the memory that is allocated on the call stack. The ABI [25] specifies that:

1. Each variable should be aligned on an address that is a multiple of its size.

2. The address in the stack pointer (`rsp`) should be a multiple of 16 immediately before another function is called.

These rules are best illustrated by considering the program in Listing 8.10.

---

```

1  /*
2   * varAlign1.c
3   * Allocates some local variables to illustrate their
4   * alignment on the call stack.
5   * Bob Plantz - 11 June 2009
6  */
7
8 #include <stdio.h>
9
10 int main(void)
11 {
12     char alpha, beta, gamma;
13     char *letterPtr;
14     int number;
15     int *numPtr;
16
17     alpha = 'A';
18     beta = 'B';
19     gamma = 'C';
20     number = 123;
21     letterPtr = &alpha;
22     numPtr = &number;
23
24     printf("%c %c %c %i\n", *letterPtr,
25            beta, gamma, *numPtr);
26
27     return 0;
28 }
```

---

Listing 8.10: Some local variables (C).

The assembly language generated by the compiler is shown in Listing 8.11 with comments added for explanation.

---

```

1      .file  "varAlign1.c"
2      .section    .rodata
3 .LC0:
4      .string "%c %c %c %i\n"
5      .text
6      .globl  main
7      .type   main, @function
8 main:
9      pushq  %rbp
10     movq   %rsp, %rbp
11     subq   $32, %rsp      # 2 * 16
12     movb   $65, -23(%rbp) # alpha = 'A'
13     movb   $66, -22(%rbp) # beta = 'B';
14     movb   $67, -21(%rbp) # gamma = 'C';
15     movl   $123, -20(%rbp) # number = 123;
```

---

```

16    leaq    -23(%rbp), %rax
17    movq    %rax, -16(%rbp) # letterPtr = &alpha;
18    leaq    -20(%rbp), %rax
19    movq    %rax, -8(%rbp) # numPtr = &number;
20    movq    -8(%rbp), %rax
21    movl    (%rax), %esi
22    movsbl -21(%rbp), %ecx
23    movsbl -22(%rbp), %edx
24    movq    -16(%rbp), %rax
25    movzbl (%rax), %eax
26    movsbl %al, %eax
27    movl    %esi, %r8d
28    movl    %eax, %esi
29    movl    $.LC0, %edi
30    movl    $0, %eax
31    call    printf
32    movl    $0, %eax
33    leave
34    ret
35    .size   main, .-main
36    .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
37    .section .note.GNU-stack,"",@progbits

```

Listing 8.11: Some local variables (gcc assembly language).

Twenty-three bytes are required for storing these variables:

- three bytes for the three `char` variables.
- four bytes for the `int` variable.
- sixteen bytes for the two pointer variables.

However, the ABI states that the stack pointer must be on a 16-byte address boundary, so we need to allocate 32 bytes for the local variables:

```
11    subq    $32, %rsp      # 2 * 16
```

Listing 8.12 shows how an assembly language programmer uses symbolic names to write code that is easier to read. In this program, each variable is at the same relative location in the stack frame as in the gcc-generated version.

```

1 # varAlign2.s
2 # Allocates some local variables to illustrate their
3 # alignment on the call stack.
4 # Bob Plantz - 11 June 2009
5 # Stack frame
6     .equ    alpha,-23
7     .equ    beta,-22
8     .equ    gamma,-21
9     .equ    number,-20
10    .equ   letterPtr,-16
11    .equ   numPtr,-8
12    .equ   localSize,-32
13 # Read only data
14     .section .rodata

```

```

15 format:           .string "%c %c %c %i\n"
16 # Code
17 .text
18 .globl main
19 .type main, @function
20
21 main:
22     pushq %rbp          # save caller's frame pointer
23     movq %rsp, %rbp      # establish our frame pointer
24     addq $localSize, %rsp # for local vars
25
26     movb $'A', alpha(%rbp)    # initialize variables
27     movb $'B', beta(%rbp)
28     movb $'C', gamma(%rbp)
29     movl $123, number(%rbp)
30
31     leaq alpha(%rbp), %rax   # initialize pointers
32     movq %rax, letterPtr(%rbp)
33     leaq number(%rbp), %rax
34     movq %rax, numPtr(%rbp)
35
36     movq numPtr(%rbp), %rax  # load pointer
37     movl (%rax), %r8d        # for dereference
38     movb gamma(%rbp), %cl
39     movb beta(%rbp), %dl
40     movq letterPtr(%rbp), %rax
41     movb (%rax), %sil
42     movl $format, %edi
43     movq $0, %rax
44     call printf
45
46     movl $0, %eax          # return 0 to OS
47     movq %rbp, %rsp         # restore stack pointer
48     popq %rbp              # restore caller's frame pointer
49     ret

```

Listing 8.12: Some local variables (programmer assembly language).

Notice the assembly language syntax for single character constants on lines 26 – 28:

```

26     movb $'A', alpha(%rbp)    # initialize variables
27     movb $'B', beta(%rbp)
28     movb $'C', gamma(%rbp)

```

The GNU assembly language `info` documentation specifies that only the first single quote, '`A`', is required. But the C syntax, '`'A'`', also works, so we have used that because it is generally easier to read.<sup>1</sup>

We can summarize the proper sequence of instructions for establishing a local variable environment in a function:

1. Push the calling function's frame pointer onto the stack.

---

<sup>1</sup>Also, the `LATEX`macro used to pretty print listings in this book does not process the single-quote syntax correctly.

2. Copy the value in the stack pointer register (`rsp`) into the frame pointer register (`rbp`) to establish the frame pointer for the current function.
3. Allocate space for the local variables by moving the stack pointer to a lower address.

Just before ending this function, these three steps need to be undone. Since the frame pointer is pointing to where the top of the stack was before we allocated memory for local variables, the local variable memory can be deleted by simply copying the value in the frame pointer to the stack pointer. Now the calling function's frame pointer value is at the top of the stack. The ending sequence is:

1. Copy the value in the frame pointer register (`rbp`) to the stack pointer register (`rsp`).
2. Pop the value at the top of the stack into the frame pointer register (`rbp`).

Listing 8.13 shows the general format that must be followed when writing a function. If you follow this format and do everything in the order that is given for all your functions, you will have many fewer problems getting them to work properly. If you do not, I guarantee that you will have many problems.

```

1 # general.s
2     .text
3     .globl general
4     .type general, @function
5 general:
6         pushq %rbp      # save calling function's frame pointer
7         movq %rsp, %rbp # establish our frame pointer
8
9 # 1. Allocate memory for local variables and saving registers,
10 #    ensuring that the address in rsp is a multiple of 16.
11 # 2. Save the contents of general purpose registers that must be
12 #    preserved and are used in this function.
13
14 # 3. The code that implements the function goes here.
15
16 # 4. Restore the contents of the general purpose registers that
17 #    were saved in step 2.
18 # 5. Place the return value, if any, in the eax register.
19
20         movq %rbp, %rsp # delete local variables and reg. save area
21         popq %rbp        # restore calling function's frame
22                           # pointer
23         ret

```

Listing 8.13: General format of a function written in assembly language.

## 8.6 Using syscall to Perform I/O

The `printf` and `scanf` functions discussed in Section 2.5 (page 13) are C library functions that convert program data to and from text formats for interacting with users via the screen and keyboard. The `write` and `read` functions discussed in Section 2.8 (page 23) are C wrapper functions that only pass bytes to output and from input devices, relying

on the program to perform the conversions so that the bytes are meaningful to the I/O device. Ultimately, each of these functions call upon the services of the operating system to perform the actual byte transfers to and from I/O devices.

In assembly language, you do not need to use the C environment. The convention is to begin program execution at the `_start` label. (Note that there are two underscore characters.) The assembler is used as before, but instead of using `gcc` to link in the C libraries, use `ld` directly. You need to specify the entry point of your program. For example, the command for the program in Listing 8.14 is:

```
bob$ ld -e _start -o echoChar3 echoChar3.o
```

When performing I/O you invoke the Linux operations yourself. The technique involves moving the arguments to specific registers, placing a special code in the `eax` register, and then using the `syscall` instruction to call a function in the operating system. (The way this works is described in Section 15.6 on page 388.) The operating system will perform the action specified by the code in the `eax` register, using the arguments passed in the other registers. The values required for reading from and writing to files are given in Table 8.3.

system call	<code>eax</code>	<code>edi</code>	<code>rsi</code>	<code>edx</code>
read	0	file descriptor	pointer to place to store bytes	number of bytes to read
write	1	file descriptor	pointer to first byte to write	number of bytes to write
exit	60			

Table 8.3: Register set up for using `syscall` instruction to read, write, or exit.

In Listing 8.14 we have rewritten the program of Listing 8.6 without using the C environment.

```

1 # echoChar3.s
2 # Prompts user to enter a character, then echoes the response
3 # Does not use C libraries
4 # Bob Plantz -- 11 June 2009
5
6 # Useful constants
7     .equ    STDIN,0
8     .equ    STDOUT,1
9     .equ    READ,0
10    .equ    WRITE,1
11    .equ    EXIT,60
12 # Stack frame
13     .equ    aLetter,-16
14     .equ    localSize,-16
15 # Read only data
16     .section .rodata      # the read-only data section
17 prompt:
18     .string "Enter one character: "
19     .equ    promptSz,.-prompt-1
20 msg:
21     .string "You entered: "
22     .equ    msgSz,.-msg-1
23 # Code
```

```

24      .text          # switch to text section
25      .globl __start
26
27  __start:
28      pushq %rbp      # save caller's frame pointer
29      movq %rsp, %rbp  # establish our frame pointer
30      addq $localSize, %rsp # for local variable
31
32      movl $promptSz, %edx # prompt size
33      movl $prompt, %esi  # address of prompt text string
34      movl $STDOUT, %edi  # standard out
35      movl $WRITE, %eax   # request kernel service
36
37      movl $2, %edx      # 1 character, plus newline
38      leaq aLetter(%rbp), %rsi # place to store character
39      movl $STDIN, %edi    # standard in
40      movl $READ, %eax    # request kernel service
41
42      syscall
43
44      movl $msgSz, %edx  # message size
45      movl $msg, %esi    # address of message text string
46      movl $STDOUT, %edi  # standard out
47      movl $WRITE, %eax   # request kernel service
48
49      movl $2, %edx      # 1 character, plus newline
50      leaq aLetter(%rbp), %rsi # place where character stored
51      movl $STDOUT, %edi    # standard out
52      movl $WRITE, %eax    # request kernel service
53
54      syscall
55
56      movq %rbp, %rsp    # delete local variables
57      popq %rbp          # restore caller's frame pointer
58      movl $EXIT, %eax    # exit from this process
59

```

Listing 8.14: Echo character program using the `syscall` instruction.

Comparing this program with the one in Listing 8.6, the program arguments are the same and are passed in the same registers. The only difference with using the `syscall` function is that you have to provide a code for the operation to be performed in the `eax` register. The complete list of system operations that can be performed are in the system file `/usr/include/asm-x86_64/unistd.h`. (The path on your system may be different.)

To determine the arguments that must be passed to each system operation read section 2 of the `man` page for that operation. For example, the arguments for the `write` system call can be seen by using

```
bob$ man 2 write
```

Then follow the rules in Section 8.1 for placing the arguments in the proper registers.

## 8.7 Calling Functions, 32-Bit Mode

In 32-bit mode all the arguments are pushed onto the call stack in right-to-left order. Listing 8.15 shows how to call the `write()` system call function.

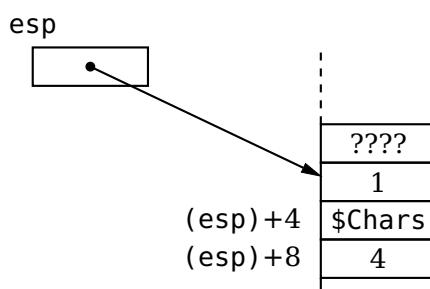
```

1 # fourChars_32.s
2 # displays four characters on the screen using the write() system call.
3 # (32-bit version.)
4 # Bob Plantz - 19 March 2008
5
6 # Read only data
7     .section  .rodata
8 Chars:
9     .byte    'A'
10    .byte    '-'
11    .byte    'Z'
12    .byte    '\n'
13 # Code
14     .text
15     .globl  main
16     .type   main, @function
17 main:
18     pushl  %ebp          # save frame pointer
19     movl  %esp, %ebp    # set new frame pointer
20
21     pushl  $4            # send four bytes
22     pushl  $Chars        #      at this location
23     pushl  $1            #      to screen.
24     call   write
25     addl  $12, %esp
26
27     movl  $0, %eax       # return 0;
28     movl  %ebp, %esp    # restore stack pointer
29     popl  %ebp          # restore frame pointer
30     ret

```

Listing 8.15: Displaying four characters on the screen using the `write` system call function in assembly language.

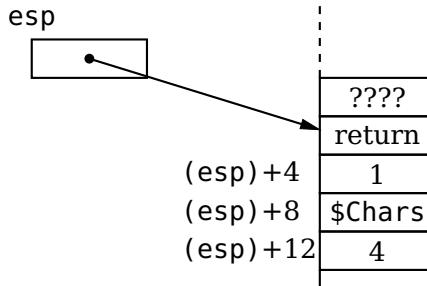
After all three arguments have been pushed onto the call stack, it looks like:



where the notation  $(\text{esp}) + n$  means “the address in the `esp` register plus  $n$ .” The stack pointer, the `esp` register, points to the last item pushed onto the call stack. The other two arguments are stored on the stack below the top item. Don’t forget that “below” on

the call stack is at numerically higher addresses because the stack grows toward lower addresses.

When the `call` instruction is executed, the return address is pushed onto the call stack as shown here:



where “return” is the address where the called function is supposed to return to at the end of its execution. So the arguments are readily available inside the called function; you will learn how to access them in Chapter 8. And as long as the called function does not change the return address, and restores the stack pointer to the position it was in when the function was called, it can easily return to the calling function.

Now, let’s look at what happens to the stack memory area in the assembly language program in Listing 8.15. Assume that the value in the `esp` register when the main function is called is `0xbfffffc5c` and that the value in the `ebp` register is `0xbfffffc6a`. Immediately after the `subl $8, %esp` instruction is executed, the stack looks like:

address	contents
bfffffc50:	????????
bfffffc54:	????????
bfffffc58:	bfffffc6a
bfffffc5c:	important information

the value in the `esp` register is `0xbfffffc50`, and the value in the `ebp` register is `0xbfffffc58`. The “?” indicates that the states of the bits in the indicated memory locations are irrelevant to us. That is, the memory between locations `0xbfffffc50` and `0xbfffffc57` is “garbage.”

We have to assume that the values in bytes number `0xbfffffc5c`, `5d`, `5e`, and `5f` were placed there by the function that called this function and have some meaning to that function. So we have to be careful to preserve the value there.

Since the `esp` register contains `0xbfffffc50`, we can continue using the stack — pushing and popping — without disturbing the eight bytes between locations `0xbfffffc50` and `0xbfffffc57`. These eight bytes are the ones we will use for storing the local variables. And if we take care not to change the value in the `ebp` register throughout the function, we can easily access the local variables.

## 8.8 Instructions Introduced Thus Far

This summary shows the assembly language instructions introduced thus far in the book. The page number where the instruction is explained in more detail, which may be in a subsequent chapter, is also given. This book provides only an introduction to the usage of each instruction. You need to consult the manuals ([2] – [6], [14] – [18]) in order to learn all the possible uses of the instructions.

## 8.8.1 Instructions

*data movement:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
movs	\$imm/%reg	%reg/mem	move	156
movsss	\$imm/%reg	%reg/mem	move, sign extend	245
movzss	\$imm/%reg	%reg/mem	move, zero extend	245
popw		%reg/mem	pop from stack	181
pushw	\$imm/%reg/mem		push onto stack	181

*s = b, w, l, q; w = l, q*

*arithmetic/logic:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
cmps	\$imm/%reg	%reg/mem	compare	237
incs	%reg/mem		increment	248
leaw	mem	%reg	load effective address	191
subs	\$imm/%reg	%reg/mem	subtract	215

*s = b, w, l, q; w = l, q*

*program flow control:*

<b>opcode</b>	<b>location</b>	<b>action</b>	<b>see page:</b>
call	label	call function	173
je	label	jump equal	239
jmp	label	jump	241
jne	label	jump not equal	239
leave		undo stack frame	192
ret		return from function	192
syscall		call kernel function	201

## 8.8.2 Addressing Modes

**register direct:** The data value is located in a CPU register.  
*syntax:* name of the register with a "%" prefix.  
*example:* movl %eax, %ebx

**immediate data:** The data value is located immediately after the instruction. Source operand only.  
*syntax:* data value with a "\$" prefix.  
*example:* movl \$0xabcd1234, %ebx

**base register plus offset:** The data value is located in memory. The address of the memory location is the sum of a value in a base register plus an offset value.  
*syntax:* use the name of the register with parentheses around the name and the offset value immediately before the left parenthesis.  
*example:* movl \$0xaabbccdd, 12(%eax)

## 8.9 Exercises

- 8-1** (§8.1) Enter the C program in Listing 8.1 and get it to work correctly. Run the program under gdb, setting a break point at the call to `write`. When the program breaks, use the `si` (Step one instruction exactly) command to execute the instructions that load registers with the arguments. As you do this, keep track of the contents in the appropriate argument registers and the `rip` register. What is the address where the text string is stored? If you single step into the `write` function, use the `cont` command to continue through it.
- 8-2** (§8.2) Modify the program in Listing 8.3 so that the stack grows from lower numbered array elements to higher numbered ones.
- 8-3** (§8.2) Enter the assembly language program in Listing 8.4 and show that the `rbp` and `rsp` registers are also saved and restored by this function.
- 8-4** (§8.4) Enter the C program in Listing 2.4 (page 24) and compile it with the debugging option, `-g`. Run the program under gdb, setting a break point at each of the calls to `write` and `read`. Each time the program breaks, use the `si` (Step one instruction exactly) command to execute the instructions that load registers with the arguments. As you do this, keep track of the contents in the appropriate argument registers and the `rip` register. What are the addresses where the text strings are stored? What is the address of the `aLetter` variable? If you single step into either the `write` or `read` functions, use the `cont` command to continue through it.
- 8-5** (§8.4) Modify the assembly language program in Listing 8.6 such that it also reads the newline character when the user enters a single character. Run the program with gdb. Set a breakpoint at the first instruction, then run the program. When it breaks, write down the values in the `rsp` and `rbp` registers. Write down the changes in these two registers as you single step (`si` command) through the first three instructions.
- Set breakpoints at the instruction that calls the `read` function and at the next instruction immediately after that one. Examine the values in the argument-passing registers.
- From the addresses you wrote down above, determine where the two characters (user's character plus newline) that are read from the keyboard will be stored, and examine that area of memory.
- Use the `cont` command to continue execution through the `read` function. Enter a character. When the program breaks back into gdb, examine the area of memory again to make sure the two characters got stored there.
- 8-6** (§8.4) Write a program in assembly language that prompts the user to enter an integer, then displays its hexadecimal equivalent.
- 8-7** (§8.4) Write a program in assembly language that “declares” four `char` variables and four `int` variables, and initializes all eight variables with appropriate values. Then call `printf` to display the values of all eight variables with only one call.

# Chapter 9

# Computer Operations

We are now ready to look more closely at the instructions that control the CPU. This will only be an introduction to the topic. We will examine the most common operations — assignment, addition, and subtraction. Additional operations will be described in subsequent chapters.

Each assembly language instruction must be translated into its corresponding machine code, including the locations of any data it manipulates. It is the bit pattern of the machine code that directs the activities of the control unit.

The goal here is to show you that a computer performs its operations based on bit patterns. As you read through this material, keep in mind that even though this material is quite tedious, the operations are very simple. Fortunately, instruction execution is very fast, so lots of meaningful work can be done by the computer.

## 9.1 The Assignment Operator

The C/C++ assignment operator, “=”, causes the expression on the right-hand side of the operator to be evaluated and the result to be associated with the variable that is named on the left-hand side. Subsequent uses of the variable name in the program will evaluate to this same value. For example,

```
int x;  
....  
x = 123;
```

will assign the integer 123 to the variable x. If x is later used in an expression, the value assigned to x will be used in evaluating the expression. For example, the expression

```
2 * x;
```

would evaluate to 246.

This assumes that the expression on the right-hand side evaluates to the same data type as the variable on the left-hand side. If not, some automatic type casting may occur, or the compiler may indicate an error. We ignore the issue of data type for now and will discuss it at several points when appropriate. For now, we are working with arbitrary bit patterns that have no meaning as “data.”

We now explore what assignment means at the assembly language level. The variable declaration,

```
int x;
```

causes memory to be allocated and the location of that memory to be given the name “x.” That is, other parts of the program can refer to the memory location where the value of x is stored by using the name “x.” The type name in the declaration, int, tells the compiler how many bytes to allocate and the code used to represent the data stored at this location. The int type uses the two’s complement code. The assignment statement,

```
x = 123;
```

sets the bit pattern in the location named x to 0x00000007b, the two’s complement code for the integer 123. The assignment statement

```
x = -123;
```

sets the bit pattern in the location named, x to 0xffffffff85, the two’s complement code for the integer -123.

Let us consider the simplest case where

- the allocated memory is within the CPU (i.e., a register).
- the bit pattern has no “real world” meaning.

That is, we will consider a program that simply sets a bit pattern in a CPU register. A C program to do this is shown in Listing 9.1.

---

```

1 /*  

2 * assignment1.c  

3 * Assign a 32-bit pattern to a register  

4 *  

5 * Bob Plantz - 11 June 2009  

6 */  

7  

8 #include <stdio.h>  

9  

10 int main(void)  

11 {  

12     register int x;  

13  

14     x = 0xabcd1234;  

15  

16     printf("x = %i\n", x);  

17  

18     return 0;  

19 }
```

---

Listing 9.1: Assignment to a register variable (C).

The register modifier “advises” the compiler to use a CPU register for the integer variable named “x.” And the notation 0xabcd1234 means that abcd1234 is written in hexadecimal. (Recall that hexadecimal is used as a compact notation for representing bit patterns.) When the C program in Listing 9.1 is compiled into its assembly language equivalent with no optimization:

```
bob$ gcc -S -O0 -fno-asynchronous-unwind-tables assignment1.c
```

the gcc compiler generates the assembly language program shown in Listing 9.2, with a comment added to show where the assignment operation takes place.

```

1      .file    "assignment1.c"
2      .section    .rodata
3 .LC0:
4      .string   "x = %i\n"
5      .text
6      .globl   main
7      .type    main, @function
8 main:
9      pushq   %rbp
10     movq    %rsp, %rbp
11     pushq   %rbx
12     subq    $8, %rsp
13     movl    $-1412623820, %ebx    # x = 0abcd1234;
14     movl    %ebx, %esi
15     movl    $.LC0, %edi
16     movl    $0, %eax
17     call    printf
18     movl    $0, %eax
19     addq    $8, %rsp
20     popq   %rbx
21     popq   %rbp
22     ret
23     .size   main, .-main
24     .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
25     .section    .note.GNU-stack,"",@progbits

```

Listing 9.2: Assignment to a register variable (gcc assembly language). Comment added to show the assignment operation.

The C assignment operation is implemented with the `mov` instruction. For example, in Listing 9.1,

```
14     x = 0abcd1234;
```

is implemented with

```
13     movl    $-1412623820, %ebx    # x = 0abcd1234;
```

on line 13 in Listing 9.2. We can see that the compiler chose to use the `ebx` register as the `x` variable.

The `mov` instruction has an “l” (“ell”, not “one”) appended to it to indicate that the operand size is 32 bits. This is redundant because the register named as an operand, `ebx`, is 32 bits, but it is the required syntax. The Intel syntax does not include this redundancy. If we consider the Intel syntax:

Intel® Syntax	mov esi, -1412623820
------------------	----------------------

we see the three other differences noted in Section 7.2.2 (page 157):

- the operand order is opposite,
- the AT&T syntax requires a “%” prefix to the name of a register, and
- the AT&T syntax requires a “\$” prefix to the immediate data.

These differences are specific to the assembler program being used and are not relevant to the behavior of the CPU. The assembler program will translate the assembly language instruction into the correct machine language code.

The instructions on lines 14 - 17 implement the call to the `printf` function. One reason for the call to the `printf` function is to prevent the compiler from eliminating the assignment statement during its optimization of this function. Yes, even with the `-O0` option the compiler does some optimization.

Compare this to Listing 7.4 on page 159. Notice that the prologue

```
8 main:
9     pushq %rbp
10    movq %rsp, %rbp
```

of this function is the same. However, the epilogue:

```
19    addq $8, %rsp
20    popq %rbx
21    popq %rbp
```

differs. Both are correct, just different ways to do the same thing.

You may wonder why the `gcc` compiler assigns the constant `-1412623820` to the variable, while the C version of the program assigns `0xabcd1234`. The answer is that they are the same values. The first is expressed in decimal and the second in hexadecimal. We discussed the equivalence of decimal and hexadecimal in Section 2.2 (page 8), and we discussed signed decimal integers in Section 3.3 (page 37).

In Listing 9.3 we show the essential assembly language required to implement the C program from Listing 9.1.

---

```
1 # assignment2.s
2 # Assigns a 32-bit pattern to the esi register.
3 # Bob Plantz - 11 June 2009
4
5     .text
6     .globl main
7     .type main, @function
8 main:
9         pushq %rbp      # save caller's base pointer
10        movq %rsp, %rbp # establish our base pointer
11        pushq %rbx      # save reg.
12
13        movl $0xabcd1234, %ebx # store a bit pattern in ebx
14
15        popq %rbx      # restore reg.
16        movl $0, %eax    # return 0 to caller
17        movq %rbp, %rsp # restore stack pointer
18        popq %rbp      # restore caller's base pointer
19        ret             # back to caller
```

Listing 9.3: Assignment to a register variable (programmer assembly language).

Compare Listing 9.3 to the general pattern in Listing 8.13 on page 201. Note that the single instruction,

```
13     movl $0xabcd1234, %ebx # store a bit pattern in ebx
```

is the only “data processing” performed by this function. From this comparison, you can see that this assembly language statement implements the two C statements:

```
register int x;
x = 0xabcd1234;
```

Like the compiler (Listing 9.2), we are using the ebx register as our variable. We can use the registers in Table 6.4 (page 134) as variables, except the stack pointer, %rsp, which has special uses. The "%" prefix tells the assembler that these are names of registers, hence in the CPU and not labels on memory locations.

Let us look more closely at the program in Listing 9.3. I used an editor to enter the code then assembled and linked it. Since it does not produce a display on the screen, I used gdb to observe the changes in the registers.

When using gdb to examine programs written in assembly language, another variant of the break command may be helpful. The version of gdb I used for this book skips over the function prologue. To cause gdb to break at the first instruction of a function, the following form should be used.

- `break *label` — sets a breakpoint at the specified label in the current source file. Notice that you cannot specify a label in another source file. Control will return to gdb when the label is encountered.

My typing is **boldface**.

```
$ gdb assignment2
```

```
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/bob/my_book_working/progs/chap09/assignment2...done
```

```
(gdb) li
```

```
1      # assignment2.s
2      # Assigns a 32-bit pattern to the esi register.
3      # Bob Plantz - 11 Jun 2009
4
5          .text
6          .globl  main
7          .type   main, @function
8      main:
9          pushq   %rbp          # save caller's frame pointer
10         movq    %rsp, %rbp    # establish our frame pointer
```

```
(gdb)
```

```
11         pushq   %rbx          # save reg.
12
13         movl    $0xabcd1234, %ebx # store a bit pattern in ebx
14
15         popq    %rbx          # restore reg.
16         movl    $0, %eax        # return 0 to caller
```

```
17      movq    %rbp, %rsp # restore stack pointer  
18      popq    %rbp       # restore caller's base pointer  
19      ret           # back to caller  
20
```

*I use the li command to list enough of the program to see where I should set the first breakpoint.*

```
(gdb) br 13
```

Breakpoint 1 at 0x4004d1: file assignment2.s, line 13.

*I set the breakpoint on the instruction that implements the assignment operation.*

```
(gdb) run
```

Starting program: /home/bob/my\_book\_working/progs/chap09/assignment2

```
Breakpoint 1, main () at assignment2.s:13
```

```
13      movl    $0xabcd1234, %ebx # store a bit pattern in ebx
```

*I run the program, it breaks at the first breakpoint, and I can display the registers that interest me.*

```
(gdb) i r rbx rip
```

```
rbx          0x0 0  
rip          0x4004d1 0x4004d1 <main+5>
```

*I use the i r (info registers) command to display the contents of the register that is used for the variable. The value in the rip register (the instruction pointer) is 0x4004d1.*

```
(gdb) si
```

```
15      popq    %rbx       # restore reg.
```

*Next I use the single instruction (si) command to execute one instruction.*

```
(gdb) i r rbx rip
```

```
rbx          0xabcd1234 2882343476  
rip          0x4004d6 0x4004d6 <main+10>
```

*Now I can see that the value, 0xabcd1234, has been assigned to the variable. Notice that the rip register has changed from 0x4004d1 to 0x4004d6. This tells us that the instruction that was just executed (movl \$0xabcd1234, %ebx) is 0x4004d6 - 0x4004d1 = 5 bytes long.*

```
(gdb) cont
```

Continuing.

[Inferior 1 (process 2622) exited normally]

*Finally, I use the continue command (cont) to run the program out to its end.*

```
(gdb) q  
$
```

*And, of course, I have to tell gdb to quit.*

## 9.2 Addition and Subtraction Operators

The assembly language instruction to perform binary addition is quite simple:

*adds source, destination*

where *s* denotes the size of the operand:

<i>s</i>	<b>meaning</b>	<b>number of bits</b>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

The add instruction adds the source operand to the destination operand using the rules of binary addition, leaving the result in the destination operand. As with the mov instruction, no more than one operand can be a memory location. The source operand is not changed. In C/C++ the operation could be expressed as:

*destination += source*

For example, the instruction

**addq %rax, %rdx**

adds the 64-bit value in the rax register to the 64-bit value in the rdx register, leaving the rax register intact. The instruction

**addw %dx, %r10w**

adds the 32-bit value in the dx register to the 32-bit value in the r10w register.

In the Intel syntax, the size of the data is determined by the operand, so the size character (b, w, l, or q) is not appended to the instruction. (And the order of the operands is reversed.)

Intel®  
Syntax

*add destination, source*

We saw in Chapter 3 that addition may cause carry or overflow. Carry and overflow are recorded in the 64-bit rflags register. The CF is bit number zero, and the OF is bit number eleven (numbering from right to left). Whenever an add instruction is executed both bits are set as shown in Algorithm 9.1.

---

**Algorithm 9.1:** Carry Flag and Overflow Flag after add.

```

1 if there is no carry then
2   CF ⇐ 0;
3 else
4   CF ⇐ 1;
5 if there is no overflow then
6   OF ⇐ 0;
7 else
8   OF ⇐ 1;
```

---

If the values being added represent unsigned ints, CF indicates whether the result fits within the operand size or not. If the values represent signed ints, OF indicates whether the result fits within the operand size or not. If the size of the operands is less than 64 bits and the operation produces a carry and/or an overflow, this is *not*

propagated up through the next bits in the destination operand. The carry and overflow conditions are simply recorded in the corresponding bits in the `rflags` register.

For example, if we consider the initial conditions

register	contents
<code>rax:</code>	<code>ffff eeee dddd cccc</code>
<code>r8:</code>	<code>2222 4444 6666 8888</code>
<code>CF:</code>	<code>?</code>
<code>OF:</code>	<code>?</code>

the instruction

`addl %eax, %r8w`

would produce

register	contents
<code>rax:</code>	<code>ffff eeee dddd cccc</code>
<code>r8:</code>	<code>2222 4444 4444 5554</code>
<code>CF:</code>	<code>1</code>
<code>OF:</code>	<code>0</code>

Whereas (starting from the same initial conditions) the instruction

`addb %al, %r8b`

would produce

register	contents
<code>rax:</code>	<code>ffff eeee dddd cccc</code>
<code>r8:</code>	<code>2222 4444 6666 8854</code>
<code>CF:</code>	<code>1</code>
<code>OF:</code>	<code>1</code>

The assembly language instruction to perform binary subtraction is

`subs source, destination`

where  $s$  denotes the size of the operand:

<u><math>s</math></u>	<u>meaning</u>	<u>number of bits</u>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

The `sub` instruction subtracts the source operand from the destination operand using the rules of binary subtraction, leaving the result in the destination operand. As with the `mov` instruction, no more than one operand can be a memory location. The source operand is not changed. In C/C++ the operation could be expressed as:

`destination -= source`

For example, the instruction

`subl %eax, %edx`

subtracts the 32-bit value in the `eax` register from the 32-bit value in the `edx` register. The instruction

`subb %dh, %ah`

subtracts the 8-bit value in the dh register from the 8-bit value in the ah register.

In the Intel syntax, the size of the data is determined by the operand, so the size character (b, w, or l) is not appended to the instruction. (And the order of the operands is reversed.)

Intel®  
Syntax

**sub** *destination, source*

Subtraction also affects the CF and the OF. Whenever a **sub** instruction is executed both bits are set as shown in Algorithm 9.2.

---

**Algorithm 9.2:** Carry Flag and Overflow Flag after subtraction.

---

```

1 if there is no borrow then
2   └ CF ⇐ 0;
3 else
4   └ CF ⇐ 1;
5 if there is no overflow then
6   └ OF ⇐ 0;
7 else
8   └ OF ⇐ 1;
```

---

Just as with addition, if the values being subtracted represent unsigned ints, CF indicates whether there was a borrow from beyond the operand size or not. If the values represent signed ints, OF indicates whether the result fits within the operand size or not. If the size of the operands is less than 64 bits and the operation produces a carry and/or an overflow, this is *not* propagated up through the next bits in the destination operand. The carry and overflow conditions are simply recorded in the corresponding bits in the rflags register.

For example, if we consider the initial conditions

register	contents
rax:	ffff eeee dddd cccc
r8:	2222 4444 6666 8888
CF:	?
OF:	?

the instruction

**subl**    %eax, %r8w

would produce

register contents
rax:      ffff eeee dddd cccc
r8:      2222 4444 8888 bbbc
CF:      1
OF:      1

Whereas (starting from the same initial conditions) the instruction

**subb**    %al, %r8b

would produce

```

register contents
    rax:      ffff eeee dddd cccc
    r8:       2222 4444 6666 88bc
    CF:        1
    OF:        0

```

A simple program given in Listing 9.4 illustrates both addition and subtraction in C.

---

```

1  /*
2  * addAndSubtract1.c
3  * Reads two integers from user, then
4  * performs addition and subtraction
5  * Bob Plantz - 11 June 2009
6  */
7
8 #include <stdio.h>
9
10 int main(void)
11 {
12     int w, x, y, z;
13
14     printf("Enter two integers: ");
15     scanf("%i %i", &w, &x);
16     y = w + x;
17     z = w - x;
18     printf("sum = %i, difference = %i\n", y, z);
19
20     return 0;
21 }

```

---

Listing 9.4: Addition and subtraction (C).

Unfortunately, this program can give incorrect results:

```

$ ./addAndSubtract1
Enter two integers: 1000000000 2000000000
sum = -1294967296, difference = -1000000000
$ ./addAndSubtract1
Enter two integers: -1000000000 2000000000
sum = 1000000000, difference = 1294967296

```

Worse, there is no message even warning that these are incorrect results. You know (see Section 3.4, page 42) that the results have overflowed. C does not check for overflow, so you would have to write code that explicitly checks for it.

The assembly language generated by gcc is shown in Listing 9.5 with comments added.

---

```

1      .file    "addAndSubtract1.c"
2      .section .rodata
3 .LC0:
4      .string "Enter two integers: "
5 .LC1:
6      .string "%i %i"
7 .LC2:
8      .string "sum = %i, difference = %i\n"

```

---

```

9      .text
10     .globl main
11     .type main, @function
12 main:
13     pushq %rbp
14     movq %rsp, %rbp
15     subq $16, %rsp
16     movl $.LC0, %edi
17     movl $0, %eax
18     call printf
19     leaq -12(%rbp), %rdx    # rdx <- address of x
20     leaq -16(%rbp), %rax
21     movq %rax, %rsi
22     movl $.LC1, %edi
23     movl $0, %eax
24     call __isoc99_scanf
25     movl -16(%rbp), %edx
26     movl -12(%rbp), %eax
27     addl %edx, %eax
28     movl %eax, -8(%rbp)
29     movl -16(%rbp), %edx
30     movl -12(%rbp), %eax
31     movl %edx, %ecx
32     subl %eax, %ecx
33     movl %ecx, %eax        # x = w-x;
34     movl %eax, -4(%rbp)    # z = w-x;
35     movl -4(%rbp), %edx
36     movl -8(%rbp), %eax
37     movl %eax, %esi
38     movl $.LC2, %edi
39     movl $0, %eax
40     call printf
41     movl $0, %eax
42     leave
43     ret
44     .size main, .-main
45     .ident "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
46     .section .note.GNU-stack,"",@progbits

```

Listing 9.5: Addition and subtraction (gcc assembly language).

We see that a rather simple C statement:

```
16     y = w + x;
```

must be broken down into distinct steps at the assembly language level:

```

25     movl -16(%rbp), %edx    # load w
26     movl -12(%rbp), %eax    # load x
27     addl %edx, %eax        # x = w + x;
28     movl %eax, -8(%rbp)    # y = x;

```

Similarly, the C statement:

```
17     z = w - x;
```

is broken down into the distinct steps:

```

29      movl    -16(%rbp), %edx  # load w
30      movl    -12(%rbp), %eax  # load x
31      movl    %edx, %ecx
32      subl    %eax, %ecx
33      movl    %ecx, %eax      # x = w-x;
34      movl    %eax, -4(%rbp)   # z = w-x;

```

It is easy to see that the compiler did not generate the most efficient code. (This was compiled with no optimization.)

An important lesson here is that writing complex statements in high-level programming languages does not improve efficiency. The statements are ultimately broken down into simple steps. See Exercise 9.3 for a comparison.

We have seen that the computations performed by both these C statements can produce overflow. Table 9.1 shows how the variables (and CF and OF) change as we walk through the code in the program of Listing 9.4. There are two runs of the program using the input values above.

statement	w	x	y	z	CF	OF
scanf();	0x3b9aca00	0x77359400	?????????	?????????	?	?
y = w + x;	0x3b9aca00	0x77359400	0xb2d05e00	?????????	0	0
z = w - x;	0x3b9aca00	0x77359400	0xb2d05e00	0xc4653600	1	0
scanf();	0xc4653600	0x77359400	?????????	?????????	?	?
y = w + x;	0xc4653600	0x77359400	0x3b9aca00	?????????	0	0
z = w - x;	0xc4653600	0x77359400	0x3b9aca00	0x4d2fa200	0	1

Table 9.1: Walking through the code in Listing 9.4. There are two runs of the program here.

Listing 9.6 shows an assembly language program that performs the same operations as the C program in Listing 9.4 but uses the `jno` (jump if no overflow) instruction to check for overflow. These checks are easy in assembly language. They add very little to the execution time of the program, because most of the time only the conditional jumps are executed, and the jumps do not take place.

```

1 # addAndSubtract2.s
2 # Gets two integers from user, then
3 # performs addition and subtraction
4 # Bob Plantz - 11 June 2009
5 # Stack frame
6     .equ    w,-8
7     .equ    x,-4
8     .equ    localSize,-16
9 # Read only data
10    .section .rodata
11 prompt:
12    .string "Enter two integers: "
13 getData:
14    .string "%i %i"
15 display:
16    .string "sum = %i, difference = %i\n"
17 warning:

```

```

18         .string "Overflow has occurred.\n"
19 # Code
20         .text
21         .globl main
22         .type main, @function
23 main:
24     pushq    %rbp      # save caller's base pointer
25     movq    %rsp, %rbp # establish our base pointer
26     addq    $localSize, %rsp # for local vars
27
28     movl    $prompt, %edi # prompt user
29     movl    $0, %eax      # no floats
30     call    printf
31
32     leaq    x(%rbp), %rdx # &x
33     leaq    w(%rbp), %rsi # &w
34     movl    $getData, %edi # get user data
35     movl    $0, %eax      # no floats
36     call    scanf
37
38     movl    w(%rbp), %esi # y = w
39     addl    x(%rbp), %esi # y += x
40     jno    nOver1        # skip warning if no OF
41     movl    $warning, %edi
42     movl    $0, %eax
43     call    printf
44 nOver1:
45     movl    w(%rbp), %edx # z = w
46     subl    x(%rbp), %edx # z -= x
47     jno    nOver2        # skip warning if no OF
48     movl    $warning, %edi
49     movl    $0, %eax
50     call    printf
51 nOver2:
52     movl    $display, %edi # display results
53     movl    $0, %eax      # no floats
54     call    printf
55
56     movl    $0, %eax      # return 0 to OS
57     movq    %rbp, %rsp    # restore stack pointer
58     popq    %rbp          # restore caller's base pointer
59     ret

```

Listing 9.6: Addition and subtraction (programmer assembly language).

## 9.3 Introduction to Machine Code

This section provides only a very brief glimpse of the machine code for the x86 architecture. The goal here is to provide you with a taste of what machine code looks like and thus emphasize that the computer is really controlled by groups of bit settings. The vast majority of computer professionals never need to know the machine code for the

computer they are working with. For a complete description you will need to consult the manufacturer's documentation.

Let us consider for a moment how we might design a set of machine instructions for a simple four-function computer. Our proposed computer can add, subtract, multiply, and divide. And we will suppose that it has 1 MB of memory. Each instruction must encode the following information for the control unit:

1. the operation to be performed, and
2. the location of the operand(s), if any, to operate on.

We will ignore the problem of getting data into the computer for this example, but we will certainly want to be able to move data from location to location in our computer. So we will have five operations:

```
move  
add  
subtract  
multiply  
divide
```

Our design will need to allow three bits for encoding each of these operations. For example, we could use the following code:

move	000
add	001
subtract	010
multiply	100
divide	111

Recall that  $N$  bits can be used to encode  $2^N$  different values. We want 1 MB of memory. From  $2^{10} = 1024 = 1K$ , and  $1M = 1K \times 1K = 2^{10} \times 2^{10} = 2^{20}$ , we see that we need to allow 20 bits for memory addressing.

Thus, if we want our computer to be able to add a value stored in one memory location to the value at another we need  $3 + 20 + 20 = 43$  bits to encode the instruction. Question: how many bits would be required if we wanted a design that would allow us to add two values stored in memory and store the sum at a third location?

Our silly design falls far short of practicality. The instructions themselves take too much memory, and we have allowed for only a very limited number of operations on the data. This was a more serious problem in the early days of computer design because memory was very expensive. The result was that computer designers came up with some clever ways to encode the necessary information into very few bits.

The design of the x86 processors is a very good example of this cleverness. Intel has paid particular attention to backwards compatibility as their designs have evolved. Thus, we see the remnants of the earlier designs — when memory was very expensive — in the latest Intel processors. The more common instructions generally take fewer bytes of memory. As newer, more complex features have been added, they generally take more bytes.

Computer design took a different turn in the 1980s. Memory had become much cheaper and CPUs had become much faster. This led to designs where all the instructions are the same size — 32 bits being very common these days.

We now turn our attention to the machine code that is produced by the assembler. Recall that it is the machine code that is actually executed by the control unit in the CPU. That is, the computer is controlled by bit patterns that are loaded into the instruction register in the CPU.

Programmers seldom need to know what the machine code is for any given assembly language instruction. The actual instruction depends upon the operation to be performed, the location(s) of the data to operate on, and the size of the data. Even when writing in assembly language, the programmer uses mnemonic names to specify each of these, and the assembler program translates them into the proper machine code instruction. So you do not need to memorize machine code. However, learning how assembly language instructions translate to machine code is important for learning how a computer actually works. And knowing how to “hand assemble” an instruction using a manual can help you find obscure bugs.

### 9.3.1 Assembler Listings

Most assemblers can provide the programmer with a listing file, which shows the machine code for each instruction. The assembly listing option for the gnu assembler is `-al`. For example, the “program” in Listing 9.7 contains some instructions that we will assemble and study to illustrate how to read machine language from a listing file.

```

1 # someMachineCode.s
2 # Some instructions to illustrate machine code.
3 # Bob Plantz - 11 June 2009
4
5     .text
6     .globl main
7     .type main, @function
8 main:
9     pushq %rbp      # save caller's base pointer
10    movq %rsp, %rbp # establish our base pointer
11
12    movq $0x1234567890abcdef, %r10 # 64-bit immediate
13    movl $0x12345678, %r11d # 32-bit immediate
14    movw $0x1234, %r12w # 16-bit immediate
15    movb $0x12, %r13b # 8-bit immediate
16
17    movq %rax, %r10      # 64-bit operands
18    movl %ecx, %r11d      # 32-bit operands
19    movw %dx, %r12w      # 16-bit operands
20    movb %bl, %r13b      # 8-bit operands
21
22    addq %r10, %rax      # add 64-bit operands
23
24    movb %al, (%rdi)      # register indirect
25    movq %r12, 24(%rsi)  # register indirect with offset
26
27    movl $0, %eax      # return 0 to caller
28    movq %rbp, %rsp      # restore stack pointer
29    popq %rbp          # restore caller's base pointer
30    ret                 # back to caller

```

Listing 9.7: Some instructions for us to assemble. (This is not a program, just some instructions.)

The command to assemble the source file in Listing 9.7 and create a listing file is

```
as --gstabs -al -o someMachineCode.o someMachineCode.s
```

The `-al` option sends the listing file to the standard output file, which defaults to the screen. You can capture this output by redirecting the standard output to a disk file. A good extension for the file name is `".lst."` The complete command is<sup>1</sup>

```
as --gstabs -al -o someMachineCode.o someMachineCode.s \
> someMachineCode.lst
```

which produces the file shown in Figure 9.1.

GAS LISTING someMachineCode.s page 1

```

1      # someMachineCode.s
2      # Some instructions to illustrate machine code.
3      # Bob Plantz - 11 June 2009
4
5          .text
6          .globl main
7          .type main, @function
8          main:
9 0000 55      pushq %rbp      # save caller's base pointe
10 0001 4889E5     movq %rsp, %rbp # establish our base pointe
11
12 0004 49BAEFC0      movq $0x1234567890abcdef, %r10 # 64-bit immediate
12 AB907856
12 3412
13 000e 41BB7856      movl $0x12345678, %r11d # 32-bit immediate
13 3412
14 0014 6641BC34      movw $0x1234, %r12w # 16-bit immediate
14 12
15 0019 41B512      movb $0x12, %r13b # 8-bit immediate
16
17 001c 4989C2      movq %rax, %r10 # 64-bit operands
18 001f 4189CB      movl %ecx, %r11d # 32-bit operands
19 0022 664189D4      movw %dx, %r12w # 16-bit operands
20 0026 4188DD      movb %bl, %r13b # 8-bit operands
21
22 0029 4C01D0      addq %r10, %rax # add 64-bit operands
23
24 002c 8807      movb %al, (%rdi) # register indirect
25 002e 4C896618      movq %r12, 24(%rsi) # register indirect with displacement
26
27 0032 B8000000      movl $0, %eax # return 0 to caller
27 00
28 0037 4889EC      movq %rbp, %rsp # restore stack pointe
29 003a 5D      popq %rbp # restore caller's base pointe
30 003b C3      ret # back to caller

```

Figure 9.1: Assembler listing file for the function shown in Listing 9.7.

The first column is the line number of the original source. You should recognize the right-hand two-thirds of the listing as the assembly language source. We will focus our

---

<sup>1</sup>Line continuation is not required but is used here to fit the command on this page.

attention on the second and third columns on the left-hand side.

The values in the first column are displayed in decimal, while the values in the second and third columns are in hexadecimal.

The function itself starts on line 8 with the label “main.” Since there is nothing else on this line in the source file, it does not occupy any memory in the program.

The first entry in the second column — `0000` — occurs at line 9. It shows the memory location relative to the beginning of the function. Since the source code on line 8 has only a label, the instruction on line 9 is the first one in this function. Furthermore, the label on line 8 applies to (relative) memory location `0000`. The label allows other parts of the program to refer to this memory location by name. In particular, since the label, `main`, is declared as a `.globl`, functions in other files linked to this one can refer to this memory location. It effectively names this function as the main function.

The entry in the third column on line 9 is `55`. It is the machine code at relative location `0000`. That is, byte number `0000` in this function is set to the bit pattern  $55_{16}$ . Following the line across, we can see that this is the machine code corresponding to the instruction

`pushq %rbp`

Since the first instruction occupies one byte of memory, the second instruction will start in byte number `0001` (the second byte from the beginning). From the assembly listing file (Figure 9.1) we see that the machine code for

`movq %rsp, %rbp`

is the bit pattern

$$4889e5_{16} = 0100\ 1000\ 1000\ 1001\ 1110\ 0101_2$$

This instruction occupies three bytes. Thus, the third instruction in this function begins at the fifth byte — relative location `0004`. Continuing to line 30, the last instruction in the program

`ret`

is a one-byte instruction. It is the sixtieth byte in the function and is located at relative location `003b` with the bit pattern,

$$c3_{16} = 1100\ 0011_2$$

So you can use the `-al` option for the `as` assembler to produce an assembler listing, which will show you exactly what the bit patterns are for each instruction and which bytes, relative to the beginning of the function, are set to these patterns.

### 9.3.2 General Format of Instructions

Instructions in the X86-64 architecture can be from one to fifteen bytes in length. Each byte falls into one of several categories:

- **Opcode** — This is the first byte in the instruction and specifies the basic operation performed by executing the instruction. It can also include operand location.
- **ModRM** — The mode/register/memory byte specifies operand locations and how they are accessed.
- **SIB** — The scale/index/base byte specifies operand locations and how they are accessed.

- Data — These bytes are used to encode constants, either those that are part of the program, or those that are relative address offsets to operand locations in memory.
- Prefix — If placed in before the opcode, these modify the behavior of the instruction, typically the size of the operands.

The general placement of these bytes is shown in Figure 9.2.

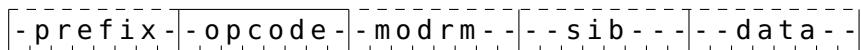


Figure 9.2: General format of instructions. There can be more than one prefix byte. The number of data bytes depends on the size of the data.

### 9.3.3 REX Prefix Byte

In order for an instruction to use the 64-bit features the x86-64 architecture uses a prefix byte, a *REX prefix*, placed immediately before the primary instruction. The assembler recognizes when a REX prefix is required and inserts it automatically; the programmer does not need to explicitly specify it. However, the assembler may give an error message that implies it is the responsibility of the programmer to insert a REX prefix. For example, when attempting to use

```
subb    %ah, %dil      # subtract bytes
```

the assembler gave the error message:

```
addAndSubtract2.s:23: Error: can't encode register '%ah' in an
instruction requiring REX prefix.
```

The reason for this error is explained in Section 6.2 (page 131). Accessing the %dil register requires that the assembler insert a REX prefix, but the %ah register cannot be accessed by an instruction that has a REX prefix.

REX prefixes are a byproduct of maintaining backward compatibility. The x86-32 architecture has only 8 general purpose registers, so it is sufficient to have only three bits in an instruction to specify any register. There are 16 general purpose registers in the x86-64 architecture, so four bits are required to specify a register. Some instructions involve up to three registers, thus there must be a place for three more bits to specify all the registers. Rather than change the register-specifying patterns in the Opcode, ModRM, and SIB bytes, the CPU designers decided to use the REX.R, REX.X, and REX.B bits in the REX prefix byte as the high-order bits for specifying registers. This provides the necessary three bits for register specification. A fourth bit in the REX prefix, the REX.W bit, is set to 1 when the operand is 64 bits. For all other operand sizes — 8, 16, or 32 bits — REX.W is set to 0. The format of the REX prefix byte is shown in Figure 9.3.



Figure 9.3: REX prefix byte. The four lettered bits are named REX.W, REX.R, REX.X, and REX.B.

### 9.3.4 ModRM Byte

The format of a ModRM byte is shown in Figure 9.4. When one operand uses the base

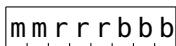


Figure 9.4: ModRM byte. The mode is specified by the **mm** bits, register by the **rrr** bits, and address base register by the **bbb** bits.

register plus offset addressing mode, that register is specified by the 3-bit **bbb** register field, and the other register is specified by the **rrr** register field. Table 9.2 shows the meaning of the 2-bit **mm** field. If **mm** = 11 both operands are register direct and are

<b>mm</b>	<b>meaning</b>
00	memory operand; address in register specified by bbb
01	memory operand; address in register specified by bbb plus 8-bit offset
10	memory operand; address in register specified by bbb plus 16-bit offset
11	register operand; register specified by bbb

Table 9.2: The **mm** field in the ModRM byte. Shows how to interpret the **bbb** register field.

specified by the two register fields, **bbb** and **rrr**. If **mm** = 00 the **bbb** register contains the memory address of one of the operands. The **bbb** register contains a base address for the other two values of **mm**. 01 means that an 8-bit offset, and 10 a 16-bit offset, is added to the base address to obtain the memory address. The offset is stored as part of the instruction.

The meaning of the register fields is shown in Table 9.3. For 64-bit mode, the REX bit column is explained in Section 9.3.3.

### 9.3.5 SIB Byte

The format of an SIB byte is shown in Figure 9.5. An SIB byte is required to implement

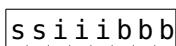


Figure 9.5: SIB byte. The **ss** bits specify a scale factor, the **iiii** bits the index register, and the **bbb** bits the address base register.

the indexed addressing mode (see Section 13.1, page 327). The memory address is given by multiplying the value in the index register by the scale factor and adding this to the address in the base register. There can also be a offset, which is added to this sum.

REX bit	register field	register names
0	0 0 0	rax, eax, ax, al
0	0 0 1	rcx, ecx, cx, cl
0	0 1 0	rdx, edx, dx, dl
0	0 1 1	rbx, ebx, bx, bl
0	1 0 0	rsp, esp, sp, spl, ah
0	1 0 1	rbp, ebp, bp, bpl, ch
0	1 1 0	rsi, esi, si, sil, dh
0	1 1 1	rdi, edi, di, dil, bh
1	0 0 0	r8, r8d, r8w, r8b
1	0 0 1	r9, r9d, r9w, r9b
1	0 1 0	r10, r10d, r10w, r10b
1	0 1 1	r11, r11d, r11w, r11b
1	1 0 0	r12, r12d, r12w, r12b
1	1 0 1	r13, r13d, r13w, r13b
1	1 1 0	r14, r14d, r14w, r14b
1	1 1 1	r15, r15d, r15w, r15b

#### Notes:

1. A 3-bit register field can be in an opcode, ModRM, or SIB byte, depending upon the instruction.
2. The REX bit is the REX.R, REX.X, or REX.B bit in the REX prefix (Section 9.3.3), depending on the location of the register field.
3. If a REX prefix is required, the REX.W bit is set to 1 for 64-bit operands.
4. The ah, bh, ch, and dh registers cannot be used in an instruction that requires a REX prefix; the spl, bpl, sil, and dil registers require a REX prefix.

Table 9.3: Machine code of general purpose registers. The register name specified by the programmer determines other bit patterns in the instruction in addition to those shown here.

### 9.3.6 The mov Instruction

We next consider the instruction on line 10 of Figure 9.1:

10 0001 4889E5                        movq      %rsp, %rbp # establish our base poi

This instruction copies all eight bytes from the rsp register to the rbp register. It starts with a REX Prefix, followed by two bytes for the instruction itself. The general format of the instruction for moving data from one register to another is shown in Figure 9.6. The

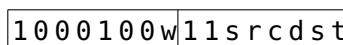


Figure 9.6: Machine code for the `mov` from a register to a register instruction. The source register is coded in the src bits and the destination in the dst bits. See Table 9.3 for the bit patterns in each of these fields.

REX Prefix is followed by the opcode, then an ModRM byte.

The opcode includes a “w” bit. This bit is 0 for 8-bit moves and 1 for all other sizes. The instruction operates on a 64-bit value, so w = 1 in the opcode ( $89_{16}$ ).

The  $11_2$  in the mod field of the ModRM byte shows that both the source and destination register numbers are encoded in this byte. The src field shows the source and the dst field shows the destination.

From Table 9.3 we see that the source register is either `rsp`, `esp`, or `sp`, and the destination register is either `rbp`, `ebp`, or `bp`. ( $w = 1$  rules out the 8-bit registers.) Since the REX.W bit in the REX Prefix is 1, the operand size is 64 bits. Thus, the instruction makes a copy of all 64 bits in the `rsp` register into the `ebp` register.

The second `mov` format covered here is moving immediate data to a register. Examples are given on lines 11 – 14 of Figure 9.1. The first operand (the source) is a literal — the value itself is stated. This value will be stored immediately after the instruction. Of course, the instruction must encode the fact that this operand is located at the address immediately following the instruction — the immediate data addressing mode. The destination operand is a register — the register direct addressing mode. The general format for the move immediate data to a register instruction is shown in Figure 9.7 in binary.

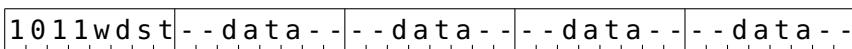


Figure 9.7: Machine code for the `mov` immediate data to a register instruction. The number of data bytes depends on the size of the data.

Consider the

11 0004 49BAEFCD	movq	\$0x1234567890abcdef, %r10
11 AB907856		
11 3412		

instruction, the assembler determines that this is a `mov` instruction and the source operand is immediate data (due to the “\$” character), so the first four bits of the opcode are `1011` (see Figure 9.7). Since the operand is not 8 bits, the “`w`” bit is 1. Next, the assembler figures out that the destination register is the `r10` register. Looking this up on Table 9.3 (which is built into the assembler) shows that the remaining three bits are `010`. Thus, the assembler generates the first byte of the instruction:

$$1011 \ 1010_2 = ba_{16}$$

Since the operand size is 64 bits, the data value, `0x1234567890abcdef`, is stored immediately (immediate addressing mode) after the instruction. Notice that the bytes seem to be stored backwards. That is, it looks like the assembler stored the 64-bit value `0xefcdab9078563412!` Recall that the x86-64 architecture uses the little endian order for storing data in memory, so when the `movl` instruction copies four bytes from memory into a register, the byte at the lowest memory address is loaded into the least significant byte of the register, the byte at the next memory address is loaded into the next higher order byte of the register, etc. The assembler takes this into account for us and stores the immediate data in memory in little endian format.

The endian issue is irrelevant if you are always consistent with the size of the data item. However, if your algorithm changes data size, you need to be very aware of the endianness of the processor. For example, if you use a `movl` to store four bytes in memory, then four `movbs` to read them back into registers, you need to be aware of how they are physically stored in memory.

Finally, since this instruction operates on a 64-bit value, the instruction requires a REX Prefix. Referring to Figure 9.3 we see that the REX.W bit is 1, indicating the 64-bit

size of the operands. And the REX.B bit is 1, which is used with the dst field to give the 4-bit number of the r10 register,  $1010_2$ .

**BE CAREFUL!** Notice that the instruction is ten bytes long (Figure 9.1), but the operand size is four bytes. Do not confuse the size of the instruction with the size of the operand(s).

### 9.3.7 The add Instruction

The add instruction has three different general formats. We present only a partial description here.

The format for adding an immediate value to a value in the `rax`, `eax`, `ax`, or `al` register is shown in Figure 9.8. The `w` bit is 0 for `al` and 1 for all others. The immediate data value must be the same size as the register to which it is added, except when adding to the `rax` register. Then the immediate data is 32 bits and is sign-extended to 64 bits before adding it to the value in the `rax` register. Note that this instruction is *not* used for the `ah` portion of the `a` register. For adding an immediate value to a value to the `ah` register or any of the other registers, the assembler program must use the instruction shown in Figure 9.9.

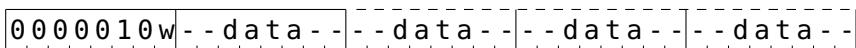


Figure 9.8: Machine code for the add immediate data to the A register (except ah) instruction. The number of data bytes depends on the size of the data.

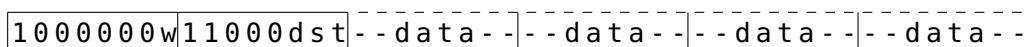


Figure 9.9: Machine code for the add immediate data to register (not `al`, `ax`, nor `eax` registers) instruction. The number of data bytes depends on the size of the data.

Notice that the instruction for adding to the `a` register (except the `ah` portion) is one byte shorter than when adding to the other registers (compare Figures 9.8 and 9.9). There is an historical reason for this. Early CPU designs had only one general purpose register. It was used as the “accumulator” for performing arithmetic. (Perhaps naming it the “`a`” register makes a little more sense.) As more general purpose registers were added to the designs, assembly language programmers tended to continue using the “accumulator” register more frequently than the others. And compiler writers continued this same pattern of register usage. Hence, the “`a`” register is used much more for addition in a program than the other registers, and making it a shorter instruction reduces memory usage and increases execution speed. The differences are generally irrelevant these days, but the x86 architecture has evolved in such a way to maintain backward compatibility.

The add instruction shown in Figure 9.10 is used when the data value is small enough to fit into one byte, but it is being added to a two-, four-, or eight-byte register. The value is sign-extended to a full 16-bit, 32-bit, or 64-bit value, respectively, inside the CPU before it is added to the register. Sign-extension consists of copying the high-order bit into each bit to the left until the full width is reached. For example, sign-extending `0x7f` to 32 bits would give `0x0000007f`; sign-extending `0x80` to 32 bits would give `0xffffffff80`.

Notice that sign-extension preserves the signed decimal value of the bit pattern. (Review Section 3.3.)

1	0	0	0	0	0	1	1	1	0	0	0	d	s	t	-	-	d	a	-	-
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Figure 9.10: Machine code for the add immediate data to a register instruction. Used when the data will fit into one byte, but the register is two, four, or eight bytes. Value is sign-extended.

An example of this is the instruction

addl \$5, %ecx

Even though the value can be coded in only eight bits, the full 32 bits of the register may be affected by the addition. That is, the machine code is 83c105 (the data is coded in only one byte), but the CPU adds 0x00000005 to the rcx register. (Recall that this may produce different results than simply adding 0x05 to the cl portion of the ecx register.)

The format for adding a value in a register to a value in a register is shown in Figure 9.11. Again, the registers and size of data are specified by the bits w, src, and dst are given in Table 9.3, and “src” means “source” and “dst” means “destination.”

0	0	0	0	0	0	0	w	1	1	s	r	c	d	s	t
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Figure 9.11: Machine code for the add register to register instruction.

Let us look at the add instruction on line 17 in Figure 9.1:

addl %ecx, %edx

This instruction adds the 32 bits from the ecx register to the 32 bits in the edx register, leaving the result in the edx register. From Table 9.3, w = 1, src = 001, and dst = 010. Thus the instruction is

$$00000001\ 11001010_2 = 01ca8_{16}$$

## 9.4 Instructions Introduced Thus Far

This summary shows the assembly language instructions introduced thus far in the book. The page number where the instruction is explained in more detail, which may be in a subsequent chapter, is also given. This book provides only an introduction to the usage of each instruction. You need to consult the manuals ([2] - [6], [14] - [18]) in order to learn all the possible uses of the instructions.

### 9.4.1 Instructions

*data movement:*

opcode	source	destination	action	see page:
movs	\$imm/%reg	%reg/mem	move	156
movsss	\$imm/%reg	%reg/mem	move, sign extend	245
movzss	\$imm/%reg	%reg/mem	move, zero extend	245
popw		%reg/mem	pop from stack	181
pushw	\$imm/%reg/mem		push onto stack	181

*s = b, w, l, q; w = l, q*

*arithmetic/logic:*

opcode	source	destination	action	see page:
adds	\$imm/%reg	%reg/mem	add	214
adds	mem	%reg	add	214
cmps	\$imm/%reg	%reg/mem	compare	237
incs	%reg/mem		increment	248
leaw	mem	%reg	load effective address	191
subs	\$imm/%reg	%reg/mem	subtract	215
subs	mem	%reg	subtract	215

*s = b, w, l, q; w = l, q*

*program flow control:*

opcode	location	action	see page:
call	label	call function	173
je	label	jump equal	239
jmp	label	jump	241
jne	label	jump not equal	239
jno	label	jump no overflow	239
leave		undo stack frame	192
ret		return from function	192
syscall		call kernel function	201

### 9.4.2 Addressing Modes

**register direct:** The data value is located in a CPU register.  
*syntax:* name of the register with a "%" prefix.  
*example:* movl %eax, %ebx

**immediate data:** The data value is located immediately after the instruction. Source operand only.  
*syntax:* data value with a "\$" prefix.  
*example:* movl \$0xabcd1234, %ebx

**base register plus offset:** The data value is located in memory. The address of the memory location is the sum of a value in a base register plus an offset value.  
*syntax:* use the name of the register with parentheses around the name and the offset value immediately before the left parenthesis.  
*example:* movl \$0xaabbccdd, 12(%eax)

## 9.5 Exercises

- 9-1** (§9.1) Enter the assembly language program in Listing 9.3. Use `gdb` to single step through the program as shown in the book. Before executing each instruction, predict how the `rax`, `rbp`, and `rsp` registers will change. Also record the values in the `rip` and `eflags` registers as you single step through the program. How many bytes are there in each instruction?
- 9-2** (§9.2) Enter the C program in Listing 9.4. Using `gdb`, verify that the program works correctly, as shown in Table 9.1.
- 9-3** (§9.2) Modify the C program in Listing 9.4 such that the arithmetic operations are done in single steps:

```
y = w;
y += x;
z = w;
z -= x;
```

Use the `-S gcc` option to generate the assembly language and compare it with the version in Listing 9.5.

- 9-4** (§9.2) Enter the assembly language program in Listing 9.6 and run it. Notice that it gives different results than the C version if there is overflow. Why is this? Modify the program so that it gives the same results as the C version but still gives an overflow warning.
- 9-5** (§9.3) Assemble each of the `mov` instructions in Listings 9.7 by hand. Check your answers with the assembly listing.
- 9-6** (§9.3) Assemble each of the `add` instructions in Listing 9.7 by hand. Check your answers with the assembly listing.
- 9-7** (§9.3) Assemble each of the following instructions by hand (on paper).

- |   |   |
|---|---|
| a) <code>movl \$0x89abcdef, %ecx</code> | e) <code>movq %r8, %r15</code>              |
| b) <code>movw \$0xabcd, %ax</code>      | f) <code>movb %r9b, %r10b</code>            |
| c) <code>movb \$0x30, %al</code>        | g) <code>movl %r11d, %r12d</code>           |
| d) <code>movb \$0x31, %ah</code>        | h) <code>movq \$0x7fffec9b2cf4, %rsi</code> |

Check your work by entering the code into a source file of the form

```
.text
.globl main
.type main, @function
main:
    pushq %rbp
    movq %rsp, %rbp
    # Your code sequence goes here.
    movl $0, %eax
    popq %rbp
    ret
```

and creating a listing file.

**9-8** (§9.3) Assemble each of the following instructions by hand (on paper).

- |                            |                     |
|----------------------------|---------------------|
| a) addl \$0x89abcdef, %ecx | e) addq %r12, %r15  |
| b) addw \$0xabcd, %ax      | f) addw %r8w, %r10w |
| c) addb \$0x30, %al        | g) addb %r9b, %sil  |
| d) addb \$0x31, %ah        | h) addl %esi, %edi  |

Check your work by entering the code into a source file of the form

```
.text
.globl main
.type main, @function
main:
    pushq %rbp
    movq %rsp, %rbp
    # Your code sequence goes here.
    movl $0, %eax
    popq %rbp
    ret
```

and creating a listing file.

**9-9** (§9.3) Design an experiment that will allow you to determine what the machine code is for the

```
pushq 64-bit_register
```

instruction, where “64-bit\_register” is any of the general purpose registers. What is the general format of the instruction? Show your answer as a drawing similar to Figure 9.7. Which ones use a REX prefix? Hint: assemble with the -al option.

**9-10** (§9.3) Design an experiment that will allow you to determine what the machine code is for the

```
popq 64-bit_register
```

instruction, where “64-bit\_register” is any of the general purpose registers. What is the general format of the instruction? Show your answer as a drawing similar to Figure 9.7. Which ones use a REX prefix? Hint: assemble with the -al option.

**9-11** (§9.3) Disassemble each of the machine instruction sequences by hand (on paper). (Find the corresponding assembly language instruction for each machine code instruction.) Notice that this is a much more difficult problem, because it is difficult to tell where one instruction ends and the next one begins. I have placed one machine instruction on each line to help you. Enter each of your assembly language programs into a source file and use the assembler to check your work.

- |    |        |    |        |
|----|--------|----|--------|
| a) | b0ab   | b) | 40b723 |
|    | b4cd   |    | 40b634 |
|    | 41b0ef |    | b256   |
|    | 41b701 |    | b678   |

- |    |  |    |  |
|----|--|----|--|
| c) | b83412cdab<br>bbabcd1234<br>41b900000000<br>41be7b000000 | i) | 053412cdab<br>81c3abcd1234<br>81c1d4c3b2a1<br>81c2a1b2c3d4 |
| d) | 66b8cdab<br>66bbbacd<br>66b93412<br>66ba2143             | j) | 5ab000000000<br>83c301<br>83c100<br>81c2ff000000           |
| e) | 88c4<br>8808<br>88480a<br>8a08<br>8a480a                 | k) | 6605cdab<br>6681c3bace<br>6681c13412<br>6681c22143         |
| f) | 89c3<br>6689d8<br>4889ca<br>4589c6                       | l) | 6605ab00<br>6683c301<br>6683c100<br>6681c2ff00             |
| g) | 04ab<br>80c4cd<br>80c3ef<br>80c701                       | m) | 00c4<br>4100c2<br>00ca<br>4500c1                           |
| h) | 80c123<br>80c534<br>80c256<br>80c678                     | n) | 01c3<br>6600d8<br>4801ca<br>4501c6                         |

# Chapter 10

# Program Flow Constructs

The assembly language we have studied thus far is executed in sequence. In this chapter we will learn how to organize assembly language instructions to implement the other two required program flow constructs — repetition and binary decision.

Text string manipulations provide many examples of using program flow constructs, so we will use them to illustrate many of the concepts. Almost any program displays many text string messages on the screen, which are simply arrays of characters.

## 10.1 Repetition

The algorithms we choose when programming interact closely with the data storage structure. As you probably know, a string of characters is stored in an array. Each element of the array is of type `char`, and in C the end of the data is signified with a sentinel value, the NUL character (see Table 2.3 on page 22).

The other technique for specifying the length of the string is to store the number of characters in the string together with the string. This is implemented in Pascal by storing the number of characters in the first byte of the array, and the actual characters are stored immediately following.

Array processing is usually a repetitive task. The processing of a character string is a good example of repetition. Consider the C program in Listing 10.1.

```
1 /*  
2 * helloWorld1.c  
3 * "hello world" program using the write() system call  
4 * one character at a time.  
5 * Bob Plantz - 12 June 2009  
6 */  
7 #include <unistd.h>  
8  
9 int main(void)  
10 {  
11     char *aString = "Hello World.\n";  
12  
13     while (*aString != '\0')  
14     {  
15         write(STDOUT_FILENO, aString, 1);  
16     }  
17 }
```

```
16     aString++;
17 }
18
19 return 0;
20 }
```

Listing 10.1: Displaying a string one character at a time (C).

The while statement on lines 13 – 17,

```
while (*aString != '\0')
{
    ...
}
```

controls the execution of the statements within the {...} block.

1. It evaluates the boolean expression \*aString != '\0'.
2. If the boolean expression evaluates to false, program flow jumps to the statement immediately following the {...} block.
3. If the boolean expression evaluates to true, program flow enters the {...} block and executes the statements there in sequence.
4. At the end of the {...} block program flow jumps back up to the evaluation of the boolean expression.

The pointer variable is incremented with the

```
aString++;
```

statement. Notice that this variable must be changed inside the {...} block. Otherwise, the boolean expression will always evaluate to true, giving an “infinite” loop.

It is important that you identify the variable that the while construct uses to control program flow — the Loop Control Variable (LCV). Make sure that the value of the LCV is changed within the {...} block. Note that there may be more than one LCV.

The way that the while construct controls program flow can be seen in the flow chart in Figure 10.1. This flow chart shows that we need the following assembly language tools to construct a while loop:

- Instruction(s) to evaluate boolean expressions.
- An instruction that conditionally transfers control (jumps) to another location in the program. This is represented by the large diamond, which shows two possible paths.
- An instruction that unconditionally transfers control to another location in the program. This is represented by the line that leads from “Execute body of while loop” back to the top.

We will explore instructions that provide these tools in the next three subsections.

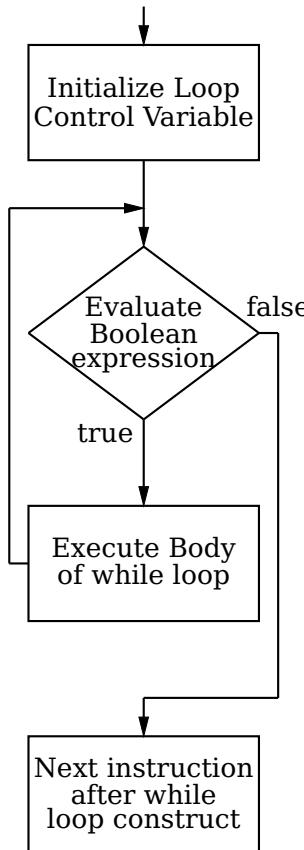


Figure 10.1: Flow chart of a `while` loop. The large diamond represents a binary decision that leads to two possible paths, “true” or “false.” Notice the path that leads back to the top of the `while` loop after the body has been executed.

### 10.1.1 Comparison Instructions

Most arithmetic and logic instructions affect the condition code bits in the `rflags` register. (See page 135.) In this section we will look at two instructions that are used to set the condition codes to show the relationship between two values without changing either of them.

One is `cmp` (compare). The syntax is

`cmps source, destination`

where *s* denotes the size of the operand:

<i>s</i>	meaning	number of bits
b	byte	8
w	word	16
l	longword	32
q	quadword	64

The `cmp` operation consists of subtracting the source operand from the destination

operand and setting the condition code bits in the rflags register accordingly. Neither of the operand values is changed. The subtraction is done internally simply to get the result and set the OF, SF, ZF, AF, PF, CF condition codes according to the result.

The other instruction is test. The syntax is

`tests source, destination`

<u>s</u>	<u>meaning</u>	<u>number of bits</u>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

where *s* denotes the size of the operand:

Intel®  
Syntax

`test destination, source`

The test operation consists of performing a bit-wise and between the two operands and setting the condition codes in the rflags register accordingly. Neither of the operand values is changed. The and operation is done internally simply to get the result and set the SF, ZF, and PF condition codes according to the result. The OF and CF are set to 0, and the AF value is undefined.

### 10.1.2 Conditional Jumps

These instructions are used to alter the flow of the program depending on the settings of the condition code bits in the rflags register. The general format is

`jcc label`

where *cc* is a 1 – 4 letter sequence specifying the condition codes, and *label* is a memory location. Program flow is transferred to *label* if *cc* is true. Otherwise, the instruction immediately following the conditional jump is executed. The conditional jump instructions are listed in Table 10.1.

A good way to appreciate the meaning of the *cc* sequences in this table is to consider a very common application of a conditional jump:

```
cmpb %al, %bl
jae somePlace
movb $0x123, %ah
```

If the value in the bl register is numerically above the value in the al register, or if they are equal, then program control transfers to the address labeled “somePlace.” Otherwise, program control continues with the movb instruction.

The differences between “greater” versus “above”, and “less” versus “below”, are a little subtle. “Above” and “below” refer to a sequence of unsigned numbers. For example, characters would probably be considered to be unsigned in most applications. “Greater” and “less” refer to signed values. Integers are commonly considered to be signed.

<b>instruction</b>	<b>action</b>	<b>condition codes</b>
ja	jump if above	$(CF = 0) \cdot (ZF = 0)$
jae	jump if above or equal	$CF = 0$
jb	jump if below	$CF = 1$
jbe	jump if below or equal	$(CF = 1) + (ZF = 1)$
jc	jump if carry	$CF = 1$
jcxz	jump if cx register zero	
jecxz	jump if ecx register zero	
jrcxz	jump if rcx register zero	
je	jump if equal	$ZF = 1$
jk	jump if greater	$(ZF = 0) \cdot (SF = OF)$
jge	jump if greater or equal	$SF = OF$
jl	jump if less	$SF \neq OF$
jle	jump if less or equal	$(ZF = 1) + (SF \neq OF)$
jna	jump if not above	$(CF = 1) + (ZF = 1)$
jnae	jump if not above or equal	$CF = 1$
jnb	jump if not below	$CF = 0$
jnbe	jump if not below or equal	$(CF = 0) \cdot (ZF = 0)$
jnc	jump if not carry	$CF = 0$
jne	jump if not equal	$ZF = 0$
jng	jump if not greater	$(ZF = 1) + (SF \neq OF)$
jnge	jump if not greater or equal	$SF \neq OF$
jnl	jump if not less	$SF = OF$
jnle	jump if not less or equal	$(ZF = 0) \cdot (SF = OF)$
jno	jump if not overflow	$OF = 0$
jnp	jump if not parity or equal	$PF = 0$
jns	jump if not sign	$SF = 0$
jnz	jump if not zero	$ZF = 0$
jo	jump if overflow	$OF = 1$
jp	jump if parity	$PF = 1$
jpe	jump if parity even	$PF = 1$
jpo	jump if parity odd	$PF = 0$
js	jump if sign	$SF = 1$
z	jump if zero	$ZF = 1$

Table 10.1: Conditional jump instructions.

Table 10.2 lists four conditional jumps that are commonly used when processing unsigned values, and Table 10.3 lists four commonly used with signed values.

<b>instruction</b>	<b>meaning</b>	<b>immediately after a cmp ...</b>
ja	jump above	jump if destination is above source in sequence
jae	jump above or equal	jump if destination is above or in same place as source in sequence
jb	jump below	jump if destination is below source in sequence
jbe	jump below or equal	jump if destination is below or in same place as source in sequence

Table 10.2: Conditional jump instructions for unsigned values.

<b>instruction</b>	<b>meaning</b>	<b>immediately after a cmp ...</b>
jg	jump greater	jump if destination is greater than source
jge	jump greater or equal	jump if destination is greater than or equal to source
jl	jump less	jump if destination is less than source
jle	jump less or equal	jump if destination is less than or equal to source

Table 10.3: Conditional jump instructions for signed values.

Since most instructions affect the settings of the condition codes in the `rflags` register, each must be used immediately after the instruction that determines the conditions that the programmer intends to cause the jump.

HINT: It is easy to forget how the order of the source and destination controls the conditional jump in this construct. Here is a place where the debugger can save you time. Simply put a breakpoint at the conditional jump instruction. When the program stops there, look at the values in the source and destination. Then use the `si` debugger command to execute one instruction and see where it goes.

The jump instructions bring up another addressing mode — *rip-relative*.<sup>1</sup>

**rip-relative:** The target is a memory address determined by adding an offset to the current address in the `rip` register.

*syntax:* a programmer-defined label

*example:* `je somePlace`

The offset, which can be positive or negative, is stored immediately following the opcode for the instruction in two's complement format. Thus, the offset becomes a part of the instruction, similar to the immediate data addressing mode. Just like the immediate addressing mode, the offset is stored in little endian order in memory.

The following steps occur during program execution of a `jcc` instruction (recall Figure 6.5):

1. The jump instruction, including the offset value, is fetched.
2. As always, the `rip` register is incremented by the number of bytes in the jump instruction, including the offset value that is stored as part of the jump instruction.
3. If the conditions to cause a jump are true, the offset is added to the `rip` register.
4. If they are not true, the instruction has no effect.

When a conditional jump instruction is assembled, the assembler computes the number of bytes from the jump instruction to the specified label. The assembler then subtracts the number of bytes in the jump instruction from the distance to the label to yield the offset. This computed offset is stored as part of the jump instruction. Each jump instruction has several forms, depending on the number of bytes that must be

<sup>1</sup>In an environment where the instruction pointer is called the “program counter” this would be called “pc-relative.”

used to store the offset. Note that the offset is stored in two's complement format to allow for negative jumps.

For example, if the offset will fit into eight bits the opcode for the `je` instruction is  $74_{16}$ , and it is  $0f84_{16}$  if more than eight bits are required to store the offset (in which case the offset is stored in as a thirty-two bit value). The machine code is shown in Table 10.4 for four different target address offsets. Notice that the 32-bit offsets are stored in little endian order in memory.

distance to target address (bytes, decimal)	machine code (hexadecimal)
+100	7462
-100	749a
+300	0f8426010000
-300	0f84cefeffff

Table 10.4: Machine code for the `je` instruction. Four different distances to the jump target address. Notice that the 32-bit offsets are stored in little endian order.

### 10.1.3 Unconditional Jump

We also need an instruction that unconditionally transfers control to another location in the program. The instruction has three forms:

```
jmp  label
jmp  *register
jmp  *memory
```

Program flow is transferred to the location specified by the operand.

The first form is limited to those situations where the distance, in number of bytes, to the target location will fit within a 32-bit signed integer. The addressing mode is rip-relative. That is, the 32-bit signed integer is added to the current value in the `rip` register. This is sufficient for most cases.

In the other two forms, the target address is stored in the specified register or memory location, and the operand is accessed indirectly. The address is an unsigned 64-bit value. The `jmp` instruction moves this stored address directly into the `rip` register, replacing the address that was in there. The “`*`” character is used to indicate “indirection.”

**BE CAREFUL:** The unconditional jump uses “`*`” for indirection, while all other instructions use “`(register)`.” It might be tempting to use something like “`*(%rax)`.” Although the `(...)` are not an error here, they are superfluous. They have essentially the same effect as something like `(x)` in an algebraic expression.

The three ways to use an unconditional jump are shown in Listing 10.2.

```
1 # jumps.s
2 # demonstrates unconditional jumps
3 # Bob Plantz - 12 June 2009
4 # global variable
5     .data
6 pointer:
7     .quad    0
8 format:
9     .string "The jump pattern is %x.\n"
```

```

10 # code
11     .text
12     .globl main
13     .type main, @function
14 main:
15     pushq %rbp      # save frame pointer
16     movq %rsp, %rbp # set new frame pointer
17
18     movl $7, %esi    # assume all three jumps
19     jmp here1
20     andl $0xfffffffffe, %esi # no jump, turn off bit 0
21 here1:
22     leaq here2, %rax
23     jmp *%rax
24     andl $0xfffffffffd, %esi # no jump, turn off bit 1
25 here2:
26     leaq here3, %rax
27     movq %rax, pointer
28     jmp *pointer
29     andl $0xfffffffffb, %esi # no jump, turn off bit 2
30 here3:
31     movl $format, %edi
32     movl $0, %eax    # no floats
33     call printf      # show pattern
34
35     movl $0, %eax    # return 0;
36     movq %rbp, %rsp # restore stack pointer
37     popq %rbp       # restore frame pointer
38     ret

```

Listing 10.2: Unconditional jumps.

The most commonly used form is rip-relative as shown on line 19:

```
19     jmp here1
```

On lines 22 – 23 an address is loaded into a register, then the jump is made indirectly via the register to that address.

```
22     leaq here2, %rax
23     jmp *%rax
```

Lines 26 – 28 show how an address can be stored in memory, then the memory used indirectly for the jump.

```
26     leaq here3, %rax
27     movq %rax, pointer
28     jmp *pointer
```

Of course, the indirect techniques are not required in this simple example, but they might be needed for some programs.

#### 10.1.4 while Loop

We are now prepared to look at how a `while` loop is constructed at the assembly language level. As usual, we begin with the assembly language generated by the `gcc` compiler for the program in Listing 10.1, which is shown in Listing 10.3 with comments added.

---

```

1      .file    "helloWorld1.c"
2      .section .rodata
3 .LC0:
4      .string "Hello World.\n"
5      .text
6      .globl  main
7      .type   main, @function
8 main:
9      pushq  %rbp
10     movq   %rsp, %rbp
11     subq   $16, %rsp
12     movq   $.LC0, -8(%rbp) # pointer to string
13     jmp    .L2             # go to bottom of loop
14 .L3:
15     movq   -8(%rbp), %rax # load pointer to string
16     movl   $1, %edx        # 3rd arg. - 1 character
17     movq   %rax, %rsi       # 2nd arg. - pointer
18     movl   $1, %edi        # 1st arg. - standard out
19     call   write
20     addq   $1, -8(%rbp)    # aString++;
21 .L2:
22     movq   -8(%rbp), %rax # load pointer
23     movzbl (%rax), %eax  # get current character
24     testb  %al, %al        # is it NUL?
25     jne    .L3             # no, go to top of loop
26     movl   $0, %eax
27     leave
28     ret
29     .size  main, .-main
30     .ident "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
31     .section .note.GNU-stack,"",@progbits

```

---

Listing 10.3: Displaying a string one character at a time (gcc assembly language). Comments added.

Let us consider the loop:

```

12     movq   $.LC0, -8(%rbp) # pointer to string
13     jmp    .L2             # go to bottom of loop
14 .L3:
15     movq   -8(%rbp), %rax # load pointer to string
16     movl   $1, %edx        # 3rd arg. - 1 character
17     movq   %rax, %rsi       # 2nd arg. - pointer
18     movl   $1, %edi        # 1st arg. - standard out
19     call   write
20     addq   $1, -8(%rbp)    # aString++;
21 .L2:
22     movq   -8(%rbp), %rax # load pointer
23     movzbl (%rax), %eax  # get current character
24     testb  %al, %al        # is it NUL?
25     jne    .L3             # no, go to top of loop

```

Notice that after initializing the loop control variable it jumps to the condition test,

```

12     movq    $.LC0, -8(%rbp) # pointer to string
13     jmp     .L2             # go to bottom of loop

```

which is at the *bottom* of the loop:

```

21 .L2:
22 .L2:
23     movq    -8(%rbp), %rax # load pointer
24     movzbl (%rax), %eax   # get current character
25     testb  %al, %al       # is it NUL?
26     jne     .L3            # no, go to top of loop

```

Let us rearrange the instructions so that this is a true while loop — the condition test is at the *top* of the loop. The exit condition has been changed from `jne` to `je` for correctness. The original is on the left, the rearranged on the right:

### Compiler's version

```

12     movq    $.LC0, -8(%rbp)
13     jmp     .L2
14 .L3:
15     movq    -8(%rbp), %rax
16     movl    $1, %edx
17     movq    %rax, %rsi
18     movl    $1, %edi
19     call    write
20     addq    $1, -8(%rbp)
21 .L2:
22     movq    -8(%rbp), %rax
23     movzbl (%rax), %eax
24     testb  %al, %al
25     jne     .L3

```

### Test at top of loop

```

12     movq    $.LC0, -8(%rbp)
13 .L2:
14     movq    -8(%rbp), %rax
15     movzbl (%rax), %eax
16     testb  %al, %al
17     je      .L3
18     movq    -8(%rbp), %rax
19     movl    $1, %edx
20     movq    %rax, %rsi
21     movl    $1, %edi
22     call    write
23     addq    $1, -8(%rbp)
24     jmp     .L2
25 .L3:

```

Both versions have exactly the same number of instructions. However, the unconditional jump instruction, `jmp`, is executed every time through the loop when testing at the top but is executed only once in the compiler's version. Thus, the compiler's version is more efficient. The savings is probably insignificant in the vast majority of applications. However, if a loop is nested within another loop or two, the difference could be important.

We also see another version of the `mov` instruction on line 22 of the compiler's version:

```

22     movzbl (%rax), %eax

```

This instruction converts the data size from 8-bit to 32-bit, placing zeros in the high-order 24 bits, as it copies the byte from memory to the `eax` register. The memory address of the copied byte is in the `rax` register. (Yes, this instruction writes over the address in the register as it executes.)

The x86-64 architecture includes instructions for extending the size of a value by adding more bits to the left. There are two ways to do this:

- **Sign extend** — copy the sign bit to each of the new high-order bits. For example, when sign extending an 8-bit value to 16 bits, 85 would become `ff85`, but 75 would become `0075`.
- **Zero extend** — make each of the new high-order bits zero. When zero extending 85 to sixteen bits, it becomes `0085`.

Sign extension can be accomplished with the `movs` instruction:

`movssd source, destination`

where *s* denotes the size of the source operand and *d* the size of the destination operand.

<u>s</u>	<u>meaning</u>	<u>number of bits</u>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

(Use the *s* column for *d*.) It can be used to move an 8-bit value from memory or a register into a 16-, 32-, or 64-bit register; move a 16-bit value from memory or a register into a 32-bit register; or move a 32-bit value from memory or a register into a 64-bit register. The “*s*” causes the rest of the high-order bits in the destination register to be a copy of the sign bit in the source value. It does not affect the condition codes in the `rflags` register.

In the Intel syntax the instruction is `movsx`. The size of the data is determined by the operands, so the size characters (b, w, l, or q) are not appended to the instruction, and the order of the operands is reversed.

Intel®  
Syntax

`movsx destination, source`

In some cases the Intel syntax is ambiguous. Intel-syntax assemblers use keywords to specify the data size in such cases. For example, the `nasm` assembler uses

`movsx destination, BYTE [source]`  
to move one byte and zero extend, and uses  
`movsx destination, WORD [source]`  
to move two bytes and sign extend.

Zero extension can be accomplished with the `movz` instruction:

`movzsd source, destination`

where *s* denotes the size of the source operand and *d* the size of the destination operand.

<u>s</u>	<u>meaning</u>	<u>number of bits</u>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

(Use the *s* column for *d*.) It can be used to move an 8-bit value from memory or a register into a 16-, 32-, or 64-bit register; or move a 16-bit value from memory or a register into a 32-bit register. The “*z*” causes the rest of the high-order bits in the destination register to be set to zero. It does not affect the condition codes in the `rflags` register. Recall that moving a 32-bit value from memory or a register into a 64-bit register sets the high-order 32 bits to zero, so there is no `movzlq` instruction.

In the Intel syntax the instruction is `movzx`. The size of the data is determined by the operands, so the size characters (b, w, l, or q) are not appended to the instruction, and the order of the operands is reversed.

Intel®  
Syntax

`movzx destination, source`

There is also a set of instructions that double the size of data in portions of the rax register, sign extending as they do so. The instructions are:

AT&T syntax	Intel® syntax	start	result
cbtw	cbw	byte in al	word in ax
cwtl	cwde	word in ax	long in eax
cwtd	cwd	word in ax	long in dx:ax
cltd	cdq	long in eax	quad in edx:eax
cltq	cdqe	long in eax	quad in rax
cqto	cqo	quad in rax	octuple in rdx:rax

where the notation “long in dx:ax” means a 32-bit value with the high-order 16 bits in dx and the low-order 16 bits in ax. Notice that these instructions do not explicitly specify any operands, but they change the rax and possibly the rdx registers. They do not affect the condition codes in the rflags register.

Returning to while loops, the general structure of a count-controlled while loop is shown in Listing 10.4.

```

1 # generalWhile.s
2 # general structure of a while loop (not a program)
3 #
4 #     count = 10;
5 #     while (count > 0)
6 #     {
7 #         // loop body
8 #         count--;
9 #     }
10 #
11 # Bob Plantz - 10 June 2009
12
13     movl    $10, count(%rbp) # initialize loop control variable
14 whileLoop:
15     cmpb    $0, count(%rbp) # check continuation conditions
16     jle     whileDone        # if false, leave loop
17 #
18 # -----#
19 # loop body processing
20 # -----#
21     subl    $1, count(%rbp) # change loop control variable
22     jmp     whileLoop       # back to top
23 whileDone:
24     # next programming construct

```

Listing 10.4: General structure of a count-controlled while loop.

This is not a complete program or even a function. It simply shows the key elements of a while loop.

Loops, of course, take the most execution time in a program. However, in almost all cases code readability is more important than efficiency. You should determine that a loop is an efficiency bottleneck before sacrificing its structure for efficiency. And then you should generously comment what you have done.

Our assembly language version of a “Hello world” program in Listing 10.5 uses a sentinel-controlled while loop.

```

1 # helloWorld3.s
2 # "hello world" program using the write() system call

```

```

3 # one character at a time.
4 # Bob Plantz - 12 June 2009
5
6 # Useful constants
7     .equ    STDOUT,1
8 # Stack frame
9     .equ    aString,-8
10    .equ    localSize,-16
11 # Read only data
12    .section .rodata
13 theString:
14    .string "Hello world.\n"
15 # Code
16    .text
17    .globl main
18    .type main, @function
19 main:
20    pushq %rbp      # save base pointer
21    movq %rsp, %rbp # set new base pointer
22    addq $localSize, %rsp # for local var.
23
24    movl $theString, %esi
25    movl %esi, aString(%rbp) # *aString = "Hello World.\n";
26 whileLoop:
27    movl aString(%rbp), %esi # current char in string
28    cmpb $0, (%esi) # null character?
29    je allDone # yes, all done
30
31    movl $1, %edx # one character
32    movl $STDOUT, %edi # standard out
33    call write # invoke write function
34
35    incl aString(%rbp) # aString++;
36    jmp whileLoop # back to top
37 allDone:
38    movl $0, %eax # return 0;
39    movq %rbp, %rsp # restore stack pointer
40    popq %rbp # restore base pointer
41    ret

```

Listing 10.5: Displaying a string one character at a time (programmer assembly language).

Consider the sequence on lines 26 – 28:

```

26 whileLoop:
27     movl aString(%rbp), %esi # current char in string
28     cmpb $0, (%esi) # null character?

```

We had to move the pointer value into a register in order to dereference the pointer. These two instruction implement the C expression:

```
(*aString != '\0')
```

In particular, you have to move the address into a register, then dereference it with the "(register)" syntax.

Be careful not to confuse this with the *indirection* operator, "\*", used with the `jmp` instruction that you saw in Section 10.1.3, especially since the assembly language indirection operator is the same as the dereference operator in C/C++.

There are two common errors when using the assembly language syntax.

- The assembly language dereference operator does not work on variable names. For example, you cannot use

```
cmpb    $0, (ptr(%rbp)) # *** DOES NOT WORK ***
```

to dereference the variable, `ptr`.

Neither do

```
cmpb $0, (theString) # *** DOES NOT WORK ***
```

nor

```
cmpb $0, (\$theString) # *** DOES NOT WORK ***
```

work to dereference the `theString` location. Unfortunately, the assembler may not consider any of these to be syntax errors, just an unnecessary set of parentheses. Therefore, you probably will not get an assembler error message, just incorrect program behavior.

- Another common error is to forget to dereference the register once you get the address stored in it:

```
cmpb    $0, %esi # *** DOES NOT WORK ***
```

This would compare a byte in the `eax` register itself with the value zero. Since there are four bytes in the `eax` register, this code will generate an assembler warning message because it does not specify which byte.

**BE CAREFUL:** The C/C++ syntax for the NUL character, '\0', is not recognized by the gnu assembler, `as`. From Table 2.3 we see that the bit pattern for the NUL character is `0x00`, and this value must be used in the gnu assembly language.

We also need to add one to the pointer variable so as to move it to the next character in the string. Adding one is a common operation, so there is an operator that simply adds one,

```
incs source
```

where  $s$  denotes the size of the operand:

<u>s</u>	<u>meaning</u>	<u>number of bits</u>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

The `inc` instruction adds one to the source operand. The operand can be a register or a memory location.

On line 34 of the program in Listing 10.5, `incl` is used to add one to the address stored in memory minus four bytes relative to the frame pointer:

```
incl    aString(%rbp) # aString++;
```

**BE CAREFUL:** It is easy to think that the instruction ought to be `incb` since each character is only one byte. The address in this program is 32 bits, so we have to use `incl`. And, of course, when we use a 64-bit address, we need to use `incq`. Don't forget that the value we are adding one to is an *address*, not the value stored at that address.

Subtracting one from a counter is also a common operation. The `dec` instruction subtracts one from an operand and sets the `rflags` register accordingly. The operand can be a register or a memory location.

```
decs  source
```

s	meaning	number of bits
b	byte	8
w	word	16
l	longword	32
q	quadword	64

where s denotes the size of the operand:

A `decl` instruction is used on line 27 in Listing 10.6 to both subtract one from the counter variable and to set the condition codes in the `rflags` register for the `jg` instruction.

```

1 # printStars.s
2 # prints 10 * characters on a line
3 # Bob Plantz - 12 June 2009
4
5 # Useful constants
6     .equ    STDOUT,1
7 # Stack frame
8     .equ    theChar,-1
9     .equ    counter,-16
10    .equ    localSize,-16
11 # Code
12    .text
13    .globl   main
14    .type    main, @function
15 main:
16    pushq   %rbp      # save base pointer
17    movq    %rsp, %rbp # set new base pointer
18    addq    $localSize, %rsp # for local var.
19
20    movb    $'*', theChar(%rbp) # character to print
21    movl    $10, counter(%rbp) # ten times
22 doWhileLoop:
23    leaq    theChar(%rbp), %rsi # address of char
24    movl    $1, %edx    # one character
25    movl    $STDOUT, %edi # standard out
26    call    write     # invoke write function
27    decl    counter(%rbp) # counter--;
28    jg     doWhileLoop # repeat if > 0
29
30    movl    $0, %eax    # return 0;

```

```

31     movq    %rbp, %rsp  # restore stack pointer
32     popq    %rbp      # restore base pointer
33     ret

```

Listing 10.6: A do-while loop to print 10 characters.

This is clearly better than using

```

...
subl    $1, counter(%rbp)  # counter--;
cmpl    $0, counter(%rbp)
jg      dowhileLoop        # repeat if > 0
...

```

This program also demonstrates how to implement a do-while loop.

## 10.2 Binary Decisions

We now know how to implement two of the primary program flow constructs — sequence and repetition. We continue on with the third — binary decision. You know this construct from C/C++ as the `if-else`.

We start the discussion with a common example — a simple program that asks the user whether changes should be saved or not (Listing 10.7). This example program does not do anything, so there really is nothing to change, but you have certainly seen this construct. (As usual, this program is meant to illustrate concepts, not good C/C++ programming practices.)

```

1 /*
2  * yesNo1.c
3  * Prompts user to enter a y/n response.
4  *
5  * Bob Plantz - 12 June 2009
6 */
7
8 #include <unistd.h>
9
10 int main(void)
11 {
12     char *ptr;
13     char response;
14
15     ptr = "Save changes? ";
16
17     while (*ptr != '\0')
18     {
19         write(STDOUT_FILENO, ptr, 1);
20         ptr++;
21     }
22
23     read (STDIN_FILENO, &response, 1);
24
25     if (response == 'y')
26     {
27         ptr = "Changes saved.\n";

```

```

28     while (*ptr != '\0')
29     {
30         write(STDOUT_FILENO, ptr, 1);
31         ptr++;
32     }
33 }
34 else
35 {
36     ptr = "Changes discarded.\n";
37     while (*ptr != '\0')
38     {
39         write(STDOUT_FILENO, ptr, 1);
40         ptr++;
41     }
42 }
43 return 0;
44 }
```

Listing 10.7: Get yes/no response from user (C).

Let's look at the flow of the program that the `if-else` controls.

1. The boolean expression (`response == 'y'`) is evaluated.
2. If the evaluation is true, the first block, the one that displays “Changes saved.”, is executed.
3. If the evaluation is false, the second block, the one that displays “Changes discarded.”, is executed.
4. In both cases the next statement to be executed is the `return 0;`

The program control flow of the `if-else` construct is illustrated in Figure 10.2.

We already know all the assembly language instructions needed to implement the `if-else` in Listing 10.7. The important thing to note is that there must be an unconditional jump at the end of the “then” block to transfer program flow around the “else” block. The assembly language generated for this program is shown in Listing 10.8.

```

1 .file  "yesNo1.c"
2 .section      .rodata
3 .LC0:
4     .string "Save changes? "
5 .LC1:
6     .string "Changes saved.\n"
7 .LC2:
8     .string "Changes discarded.\n"
9
10    .text
11    .globl  main
12    .type   main, @function
13 main:
14     pushq  %rbp
15     movq  %rsp, %rbp
16     subq  $16, %rsp
17     movq  $.LC0, -8(%rbp)
18     jmp   .L2
```

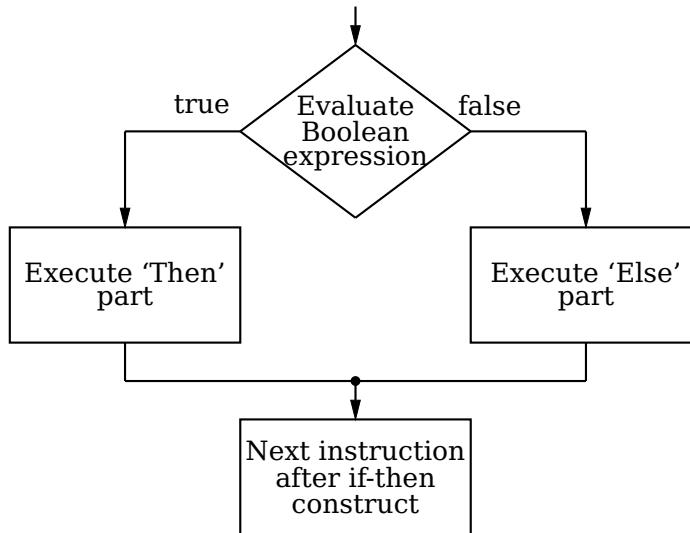


Figure 10.2: Flow chart of if-else construct. The large diamond represents a binary decision that leads to two possible paths, “true” or “false.” Notice that either the “then” block or the “else” block is executed, but not both. Each leads to the end of the if-else construct.

```

18 .L3:
19     movq    -8(%rbp), %rax
20     movl    $1, %edx
21     movq    %rax, %rsi
22     movl    $1, %edi
23     call    write
24     addq    $1, -8(%rbp)

25 .L2:
26     movq    -8(%rbp), %rax
27     movzbl (%rax), %eax
28     testb   %al, %al
29     jne     .L3
30     leaq    -9(%rbp), %rax      # place to store user response
31     movl    $1, %edx
32     movq    %rax, %rsi
33     movl    $0, %edi
34     call    read
35     movzbl -9(%rbp), %eax      # get user response
36     cmpb    $121, %al          # response == 'y' ?
37     jne     .L4                # no, go to else part
38     movq    $.LC1, -8(%rbp)    # yes, write "Changes saved.\n"
39     jmp     .L5

40 .L6:
41     movq    -8(%rbp), %rax
42     movl    $1, %edx
43     movq    %rax, %rsi
44     movl    $1, %edi
45     call    write
  
```

```

46      addq    $1, -8(%rbp)
47 .L5:
48      movq    -8(%rbp), %rax
49      movzbl (%rax), %eax
50      testb   %al, %al
51      jne     .L6
52      jmp     .L7          # jump around else part
53 .L4:
54      movq    $.LC2, -8(%rbp) #   write "Changes discarded.\n"
55      jmp     .L8
56 .L9:
57      movq    -8(%rbp), %rax
58      movl    $1, %edx
59      movq    %rax, %rsi
60      movl    $1, %edi
61      call    write
62      addq    $1, -8(%rbp)
63 .L8:
64      movq    -8(%rbp), %rax
65      movzbl (%rax), %eax
66      testb   %al, %al
67      jne     .L9          # after if-else statement
68 .L7:
69      movl    $0, %eax
70      leave
71      ret
72      .size   main, .-main
73      .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
74      .section .note.GNU-stack,"",@progbits

```

Listing 10.8: Get yes/no response from user (gcc assembly language).

The general structure of an if-else construct is shown in Listing 10.9.

```

1 # generalIf-else.s
2 # general structure of an if-else (not a program)
3 #
4 # if (response == 'y')
5 # {
6 #     then part
7 # }
8 # else
9 # {
10 #     else part
11 # }
12 #
13 # Bob Plantz - 10 June 2009
14
15      cmpb    $'y', response(%rbp) # check conditions
16      jne     noChange          # false, go to else part
17      # -----
18      # "then" part processing
19      # -----
20      jmp     allDone           # go to end of if-else

```

```

21 noChange:
22     # -----
23     # "else" part processing
24     # -----
25 allDone:
26     # next programming construct

```

Listing 10.9: General structure of an if-else construct. Don't forget the "jmp" at the end of the "then" block (line 20).

This is not a complete program or even a function. It simply shows the key elements of an if-else construct.

Our assembly language version of the yes/no program in Listing 10.10 follows this general pattern. It, of course, uses more meaningful labels than what the compiler generated.

```

1 # yesNo2.s
2 # Prompts user to enter a y/n response.
3 # Bob Plantz - 12 June 2009
4
5 # Useful constants
6     .equ    STDIN,0
7     .equ    STDOUT,1
8 # Stack frame
9     .equ    response,-1
10    .equ   ptr,-16
11    .equ   localSize,-16
12 # Read only data
13     .section .rodata
14 queryMsg:
15     .string "Save changes? "
16 saveMsg:
17     .string "Changes saved.\n"
18 discardMsg:
19     .string "Changes discarded.\n"
20 # Code
21     .text
22     .globl main
23     .type main, @function
24 main:
25     pushq %rbp          # save base pointer
26     movq %rsp, %rbp      # establish our base pointer
27     addq $localSize, %rsp # for local vars.
28     pushq %rbx          # save for caller
29
30     movl $queryMsg, %esi
31     movl %esi, ptr(%rbp) # point to query message
32 queryLoop:
33     movl ptr(%rbp), %esi # current char in string
34     cmpb $0, (%esi) # null character?
35     je getResp # yes, get user response
36
37     movl $1, %edx      # one character
38     movl $STDOUT, %edi  # standard out

```

```
39      call    write      # invoke write function
40
41      incl    ptr(%rbp)  # ptr++;
42      jmp     queryLoop # back to top
43
44 getResp:
45      movl    $1, %edx      # read one byte
46      leaq    response(%rbp), %rsi # into this location
47      movl    $STDIN, %edi      # from keyboard
48      call    read
49 # if (response == 'y')
50      cmpb    '$y', response(%rbp) # was it 'y'?
51      jne     noChange      # no, there is no change
52
53 # then print the "save" message
54      movl    $saveMsg, %esi
55      movl    %esi, ptr(%rbp) # point to message
56 saveLoop:
57      movl    ptr(%rbp), %esi # current char in string
58      cmpb    $0, (%esi) # null character?
59      je      saveEnd      # yes, leave while loop
60
61      movl    $1, %edx      # one character
62      movl    $STDOUT, %edi  # standard out
63      call    write        # invoke write function
64
65      incl    ptr(%rbp)  # ptr++;
66      jmp     saveLoop    # back to top
67
68 saveEnd:
69      jmp     allDone      # go to end of if-else
70
71 # else print the "discard" message
72 noChange:
73      movl    $discardMsg, %esi
74      movl    %esi, ptr(%rbp) # point to message
75 discardLoop:
76      movl    ptr(%rbp), %esi # current char in string
77      cmpb    $0, (%esi) # null character?
78      je      allDone      # yes, leave while loop
79
80      movl    $1, %edx      # one character
81      movl    $STDOUT, %edi  # standard out
82      call    write        # invoke write function
83
84      incl    ptr(%rbp)  # ptr++;
85      jmp     discardLoop # back to top
86
87 allDone:
88      movl    $0, %eax      # return 0;
89      popq    %rbx        # restore reg.
90      movq    %rbp, %rsp    # restore stack pointer
```

```

91      popq    %rbp          # restore for caller
92      ret

```

Listing 10.10: Get yes/no response from user (programmer assembly language).

The exit from the while loop on line 59

```

59      je      saveEnd      # yes, leave while loop

```

jumps to the end of the “then” block of the if-else statement, which then jumps to the end of the entire if-else statement:

```

68 saveEnd:
69      jmp    allDone       # go to end of if-else

```

In this particular program we could gain some efficiency by using

```

je      allDone      # yes, program done

```

on line 59. But this very slight efficiency gain comes at the expense of good software engineering. In general, there could be more processing to do after the while loop in the “then” block of the if-else statement. The real danger here is that additional processing will be added during the program’s maintenance phase and the programmer will forget to change the structure. Good, easy to read structure is almost always better than execution efficiency.

Another common programming problem is to check to see if a variable is within a certain range. This requires a compound boolean expression, as shown in the C program in Listing 10.11.

```

1 /*
2 * range.c
3 * Checks to see if a character entered by user is a numeral.
4 * Bob Plantz - 12 June 2009
5 */
6
7 #include <unistd.h>
8
9 int main()
10 {
11     char response; // For user's response
12     char* ptr;      // For text messages
13
14     ptr = "Enter single character: ";
15     while (*ptr != '\0')
16     {
17         write(STDOUT_FILENO, ptr, 1);
18         ptr++;
19     }
20
21     read(STDIN_FILENO, &response, 1);
22
23     if ((response <= '9') && (response >= '0'))
24     {
25         ptr = "You entered a numeral.\n";
26         while (*ptr != '\0')
27         {
28             write(STDOUT_FILENO, ptr, 1);

```

```

29         ptr++;
30     }
31 }
32 else
33 {
34     ptr = "You entered some other character.\n";
35     while (*ptr != '\0')
36     {
37         write(STDOUT_FILENO, ptr, 1);
38         ptr++;
39     }
40 }
41 return 0;
42 }
```

Listing 10.11: Compound boolean expression in an if-else construct (C).

Each condition of the boolean expression generally requires a separate comparison/conditional jump pair. The best way to see this is to study the compiler-generated assembly language code of the numeral checking program in Listing 10.12.

```

1      .file  "range.c"
2      .section    .rodata
3 .LC0:
4      .string "Enter single character:  "
5 .LC1:
6      .string "You entered a numeral.\n"
7      .align 8
8 .LC2:
9      .string "You entered some other character.\n"
10     .text
11     .globl main
12     .type  main, @function
13 main:
14     pushq %rbp
15     movq %rsp, %rbp
16     subq $16, %rsp
17     movq $.LC0, -8(%rbp)
18     jmp .L2
19 .L3:
20     movq -8(%rbp), %rax
21     movl $1, %edx
22     movq %rax, %rsi
23     movl $1, %edi
24     call write
25     addq $1, -8(%rbp)
26 .L2:
27     movq -8(%rbp), %rax
28     movzbl (%rax), %eax
29     testb %al, %al
30     jne .L3
31     leaq -9(%rbp), %rax
32     movl $1, %edx
33     movq %rax, %rsi
```

```

34      movl    $0, %edi
35      call    read
36      movzbl -9(%rbp), %eax      # load numeral character
37      cmpb    $57, %al          # is numeral > '9'?
38      jg     .L4              # yes, go to else part
39      movzbl -9(%rbp), %eax      # load numeral character
40      cmpb    $47, %al          # is numeral <= '/'?
41      jle    .L4              # yes, go to else part
42      movq    $.LC1, -8(%rbp)    # "then" part
43      jmp    .L5
44 .L6:
45      movq    -8(%rbp), %rax
46      movl    $1, %edx
47      movq    %rax, %rsi
48      movl    $1, %edi
49      call    write
50      addq    $1, -8(%rbp)
51 .L5:
52      movq    -8(%rbp), %rax
53      movzbl (%rax), %eax
54      testb   %al, %al
55      jne    .L6
56      jmp    .L7              # skip over "else" part
57 .L4:
58      movq    $.LC2, -8(%rbp)
59      jmp    .L8
60 .L9:
61      movq    -8(%rbp), %rax
62      movl    $1, %edx
63      movq    %rax, %rsi
64      movl    $1, %edi
65      call    write
66      addq    $1, -8(%rbp)
67 .L8:
68      movq    -8(%rbp), %rax
69      movzbl (%rax), %eax
70      testb   %al, %al
71      jne    .L9              # end of if-else construct
72 .L7:
73      movl    $0, %eax
74      leave
75      ret
76      .size   main, .-main
77      .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
78      .section .note.GNU-stack,"",@progbits

```

Listing 10.12: Compound boolean expression in an if-else construct (gcc assembly language).

In particular, notice that the decision regarding whether the character entered by the user is a numeral or not is made on the lines:

```

36      movzbl -9(%rbp), %eax      # load numeral character
37      cmpb    $57, %al          # is numeral > '9'?

```

```

38      jg      .L4          # yes, go to else part
39      movzbl -9(%rbp), %eax # load numeral character
40      cmpb    $47, %al       # is numeral <= '/'?
41      jle     .L4          # yes, go to else part
42      movq    $.LC1, -8(%rbp) # "then" part

```

Consulting Table 2.3 on page 22 we see that the program first compares the character entered by the user with the ascii code for the numeral “9” ( $57_{10} = 39_{16}$ ). If the character is numerically greater, the program jumps to .L5, which is the beginning of the “else” part. Then the character is compared to the ASCII code for the character “/”, which is numerically one less than the ascii code for the numeral “0” ( $48_{10} = 30_{16}$ ). If the character is numerically equal to or less than, the program also jumps to .L5.

If neither of these conditions causes a jump to the “else” part, the program simply continues on to execute the “then” part. At the end of the “then” part, the program skips over the “else” part to the end of the program:

```

56      jmp     .L7          # skip over "else" part
57 .L4:                                # "else" part

```

## 10.2.1 Short-Circuit Evaluation

Consider the boolean expression use for the if-else conditional:

```
22      if ((response <= '9') && (response >= '0')) {
```

On lines 35 and 36 in the assembly language,

```

35      cmpb    $57, %al       # is numeral > '9'?
36      jg      .L5          # yes, go to else part

```

we see that the test for '0' is never made if `(response <= '9')` is false.

This is called *short-circuit evaluation* in C/C++. When connecting boolean tests with the `&&` and `||` operators, each the boolean tests is each performed. If the overall result of the expression — true or false — is known before all the tests are made, the remaining tests are not executed. This is one of the most important reasons for not writing boolean expressions that include side effects; the operation that produces a needed side effect may never get executed.

## 10.2.2 Conditional Move

Many binary decisions are very simple. For example, the decision in Listing 10.7 could be written:

```

ptr = "Changes discarded.\n";
if (response == 'y')
{
    ptr = "Changes saved.\n";
}
while (*ptr != '\0')
{
    write(STDOUT_FILENO, ptr, 1);
    ptr++;
}

```

This code segment assigns an address to the `ptr` variable. If the condition, `response == 'y'`, is true, then the address in the `ptr` variable is written over with another address. This could be written in assembly language (see Listing 10.10) as:

```

        movl    $discardMsg, %esi
# if (response == 'y')
        cmpb    $'y', response(%rbp) # was it 'y'?
        jne     noChange      # no, there is no change
        movl    $saveMsg, %esi  # yes, get other message
noChange:
        movl    %esi, ptr(%rbp) # point to message
msgLoop:
        movl    ptr(%rbp), %esi # current char in string
        cmpb    $0, (%esi)   # null character?
        je      allDone       # yes, leave while loop

        movl    $1, %edx      # one character
        movl    $STDOUT, %edi  # standard out
        call    write         # invoke write function

        incl    ptr(%rbp)    # ptr++;
        jmp     msgLoop      # back to top

```

The x86-64 architecture provides a *conditional move* instruction, `cmovecc`, for simple if constructs like this. The general format is

`cmovecc source, destination`

where `cc` is a 1 - 4 letter sequence specifying the settings of the condition codes. Similar to the conditional jump instructions, the conditional data move takes place if the status flag settings are true, and does not if they are false.

Possible letter sequences are the same as for the conditional jump instructions listed in Table 10.1 on page 239. The source operand can be either a register or a memory location, and the destination must be a register. Unlike other data movement instructions, the `cmovecc` instruction does not use the operand size suffix; the size is implicitly specified by the size of the destination register.

The conditional move instruction would allow the above assembly language to be written with a `cmove` instruction, where the “e” means “equal” (see Table 10.1).

```

        movl    $discardMsg, %esi # load addresses of
        movl    $saveMsg, %edi   # both messages
# if (response == 'y')
        cmpb    $'y', response(%rbp) # was it 'y'?
        cmove  %edi, %esi      # yes, "save" message
        movl    %esi, ptr(%rbp) # point to message
msgLoop:
        movl    ptr(%rbp), %esi # current char in string
        cmpb    $0, (%esi)   # null character?
        je      allDone       # yes, leave while loop

        movl    $1, %edx      # one character
        movl    $STDOUT, %edi  # standard out
        call    write         # invoke write function

```

```
incl    ptr(%rbp)    # ptr++;
jmp     msgLoop      # back to top
```

Although this actually increases the average number of instructions executed, it allows the CPU to make more efficient use of the pipeline. So a conditional move may provide faster program execution by eliminating possible pipeline inefficiencies caused by a conditional jump. See for example [28], [31], and [34].

## 10.3 Instructions Introduced Thus Far

This summary shows the assembly language instructions introduced thus far in the book. The page number where the instruction is explained in more detail, which may be in a subsequent chapter, is also given. This book provides only an introduction to the usage of each instruction. You need to consult the manuals ([2] - [6], [14] - [18]) in order to learn all the possible uses of the instructions.

### 10.3.1 Instructions

*data movement:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
cmovecc	%reg/mem	%reg	conditional move	260
movs	\$imm/%reg	%reg/mem	move	156
movss	\$imm/%reg	%reg/mem	move, sign extend	245
movzs	\$imm/%reg	%reg/mem	move, zero extend	245
popw		%reg/mem	pop from stack	181
pushw	\$imm/%reg/mem		push onto stack	181

*s = b, w, l, q; w = l, q; cc = condition codes*

*arithmetic/logic:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
adds	\$imm/%reg	%reg/mem	add	214
adds	mem	%reg	add	214
cmps	\$imm/%reg	%reg/mem	compare	237
cmps	mem	%reg	compare	237
decs	%reg/mem		decrement	249
incs	%reg/mem		increment	248
leaw	mem	%reg	load effective address	191
subs	\$imm/%reg	%reg/mem	subtract	215
subs	mem	%reg	subtract	215
tests	\$imm/%reg	%reg/mem	test bits	238
tests	mem	%reg	test bits	238

*s = b, w, l, q; w = l, q*

*program flow control:*

<b>opcode</b>	<b>location</b>	<b>action</b>	<b>see page:</b>
call	label	call function	173
ja	label	jump above (unsigned)	239
jae	label	jump above/equal (unsigned)	239
jb	label	jump below (unsigned)	239
jbe	label	jump below/equal (unsigned)	239
je	label	jump equal	239
jg	label	jump greater than (signed)	240
jge	label	jump greater than/equal (signed)	240
jl	label	jump less than (signed)	240
jle	label	jump less than/equal (signed)	240
jmp	label	jump	241
jne	label	jump not equal	239
jno	label	jump no overflow	239
jcc	label	jump on condition codes	239
leave		undo stack frame	192
ret		return from function	192
syscall		call kernel function	201

*cc = condition codes*

### 10.3.2 Addressing Modes

**register direct:** The data value is located in a CPU register.  
*syntax:* name of the register with a "%" prefix.  
*example:* `movl %eax, %ebx`

**immediate data:** The data value is located immediately after the instruction. Source operand only.  
*syntax:* data value with a "\$" prefix.  
*example:* `movl $0xabcd1234, %ebx`

**base register plus offset:** The data value is located in memory. The address of the memory location is the sum of a value in a base register plus an offset value.  
*syntax:* use the name of the register with parentheses around the name and the offset value immediately before the left parenthesis.  
*example:* `movl $0xaabbccdd, 12(%eax)`

**rip-relative:** The target is a memory address determined by adding an offset to the current address in the rip register.  
*syntax:* a programmer-defined label  
*example:* `je somePlace`

## 10.4 Exercises

**10-1** (§10.1) Verify on paper that the machine instructions in Table 10.4 actually cause a jump of the number of bytes shown (in decimal) when the jump is taken.

**10-2** (§10.1) Enter the program in Listing 10.2 and verify that the jump to `here1` uses the rip-relative addressing mode, and the other two jumps use the direct address.

Hint: Produce a listing file for the program and use `gdb` to examine register and memory contents.

**10-3** (§10.1) Enter the program in Listing 10.5, changing the while loop to use `eax` as a pointer:

```

        movl    $theString, %eax
whileLoop:
        cmpb    $0, (%eax)  # null character?
        je      allDone     # yes, all done

        movl    $1, %edx    # one character
        movl    %eax, %esi   # current pointer
        movl    $STDOUT, %edi  # standard out
        call    write       # invoke write function

        incl    %eax        # aString++;
        jmp     whileLoop   # back to top

```

This would seem to be more efficient than reading the pointer from memory each time through the loop. Use `gdb` to debug the program. Set a break point at the `call` instruction and another break point at the `incl` instruction. Inspect the registers each time the program breaks into `gdb`. What is happening to the value in `eax`? Hint: Read what the “`man 2 write`” shell command has to say about the `write` system call function. This exercise points out the necessity of understanding what happens to registers when calling another function. In general, it is safer to use local variables in the stack frame.

**10-4** (§10.1) Assume that you do not know how many numerals there are, only that the first one is ‘0’ and the last one is ‘9’ (the character “0” and character “9”). Write a program in assembly language that displays all the numerals, 0 – 9, on the screen, one character at a time. Use only one byte in the `.data` segment for storing a character; do not allocate a separate byte for each numeral.

**10-5** (§10.1) Assume that you do not know how many upper case letters there are, only that the first one is ‘A’ and the last one is ‘Z’. Write a program in assembly language that displays all the upper case letters, A – Z, on the screen, one character at a time. Use only one byte in the `.data` segment for storing a character; do not allocate a separate byte for each numeral.

**10-6** (§10.1) Assume that you do not know how many lower case letters there are, only that the first one is ‘a’ and the last one is ‘z’. Write a program in assembly language that displays all the lower case letters, a – z, on the screen, one character at a time. Use only one byte in the `.data` segment for storing a character; do not allocate a separate byte for each numeral.

**10-7** (§10.1) Enter the following C program and use the “`-S`” option to generate the assembly language:

---

```

1  /*
2  * forLoop.c
3  * For loop multiplication.
4  *
5  * Bob Plantz - 21 June 2009
6  */

```

```
7  
8 #include<stdio.h>  
9  
10 int main ()  
11 {  
12     int x, y, z;  
13     int i;  
14  
15     printf("Enter two integers: ");  
16     scanf("%i %i", &x, &y);  
17     z = x;  
18     for (i = 1; i < y; i++)  
19         z += x;  
20  
21     printf("%i * %i = %i\n", x, y, z);  
22     return 0;  
23 }
```

---

Listing 10.13: Simple for loop to perform multiplication.

Identify the loop that performs the actual multiplication. Write an equivalent C program that uses a `while` loop instead of the `for` loop, and also generate the assembly language for it. Do the loops differ? If so, how?

**10-8** (§10.2) Enter the C program in Listing 10.7 and get it to work. Do you see any odd behavior when the program terminates? Can you fix it? Hint: When the program prompts the user, how many keys did you press? What was the second key press?

**10-9** (§10.2) Enter the program in Listing 10.10 and get it to work.

**10-10** (§10.2) Write a program in assembly language that displays all the printable characters that are neither numerals nor letters on the screen, one character at a time. Don't forget that the space character, ' ', is printable. Do not display the DEL character. Use only one byte for storing a character; do not allocate a separate byte for each character.

Use only one `while` loop in this program. You will need an `if-else` construct with a compound boolean conditional statement.

**10-11** (§10.2) Write a program in assembly language that

- a) prompts the user to enter a text string,
- b) reads the user's input into a `char` array,
- c) echoes the user's input string,
- d) increments each character in the string to the next character in the ASCII sequence, with the last printable character "wrapping around" to the first printable character, and
- e) displays the modified string.

**10-12** (§10.2) Write a program in assembly language that

- a) prompts the user to enter a text string,
- b) reads the user's input into a `char` array,
- c) echoes the user's input string,

- d) decrements each character in the string to the previous character in the ASCII sequence, with the first printable character “wrapping around” to the last printable character, and
- e) displays the modified string.

**10-13** (§10.2) Write a program in assembly language that

- a) instructs the user,
- b) prompts the user to enter a character,
- c) reads the user’s input into a char variable,
- d) if the user enters a ‘q’, the program terminates,
- e) if the user enters a numeral, the program echoes the numeral the number of times represented by the numeral plus one, and
- f) any other printable character is echoed just once.

The program continues to run until the user enters a ‘q’.

For example, a run of the program might look like (user input is **boldface**):

A single numeral, N, is echoed N+1 times, other characters are echoed once. ‘q’ ends program.

Enter a single character: **a**

You entered: a

Enter a single character: **z**

You entered: z

Enter a single character: **5**

You entered: 5

Enter a single character: **%**

You entered: %

Enter a single character: **q**

End of program.

# Chapter 11

# Writing Your Own Functions

Good software engineering practice generally includes breaking problems down into functionally distinct subproblems. This leads to software solutions with many functions, each of which solves a subproblem. This “divide and conquer” approach has some distinct advantages:

- It is easier to solve a small subproblem.
- Previous solutions to subproblems are often reusable.
- Several people can be working on different parts of the overall problems simultaneously.

The main disadvantage of breaking a problem down like this is coordinating the many subsolutions so that they work together correctly to provide a correct overall solution. In software, this translates to making sure that the interface between a calling function and a called function works correctly. In order to ensure correct operation of the interface, it must be specified in a very explicit way.

In Chapter 8 you learned how to pass arguments into a function and call it. In this chapter you will learn how to use these arguments inside the called function.

## 11.1 Overview of Passing Arguments

Be careful to distinguish data input/output to/from a called function from user input/output. User input typically comes from an input device (keyboard, mouse, etc.) and user output is typically sent to an output device (screen, printer, speaker, etc.).

Functions can interact with the data in other parts of the program in three ways:

1. **Input.** The data comes from another part of the program and is used by the function, but is not modified by it.
2. **Output.** The function provides new data to another part of the program.
3. **Update.** The function modifies a data item that is held by another part of the program. The new value is based on the value before the function was called.

All three interactions can be performed if the called function also knows the location of the data item. This can be done by the calling function passing the address to the called function or by making the address globally known to both functions. Updates require that the address be known by the called function.

Outputs can also be implemented by placing the new data item in a location that is accessible to both the called and the calling function. In C/C++ this is done by placing the return value from a function in the `eax` register. And inputs can be implemented by passing a copy of the data item to the called function. In both of these cases the called function does not know the location of the original data item, and thus does not have access to it.

In addition to global data, C syntax allows three ways for functions to exchange data:

- **Pass by value** — an input value is passed by making a copy of it available to the function.
- **Return value** — an output value can be returned to the calling function.
- **Pass by pointer** — an output value can be stored for the calling function by passing the address where the output value should be stored to the called function. This can also be used to update a data item.

The last method, pass by pointer, can also be used to pass large inputs, or to pass inputs that should be changed — also called updates. It is also the method by which C++ implements pass by reference.

When one function calls another, the information that is required to provide the interface between the two is called an *activation record*. Since both the registers and the call stack are common to all the functions within a program, both the calling function and the called function have access to them. So arguments can be passed either in registers or on the call stack. Of course, the called function must know *exactly* where each of the arguments is located when program flow transfers to it.

In principle, the locations of arguments need only be consistent within a program. As long as all the programmers working on the program observe the same rules, everything should work. However, designing a good set of rules for any real-world project is a very time-consuming process. Fortunately, the ABI [25] for the x86-64 architecture specifies a good set of rules. These rules are very tedious because they are meant to cover all possible situations. In this book we will consider only the simpler rules in order to get an overall picture of how this works.

In 64-bit mode six of the general purpose registers and a portion of the call stack are used for the activation record. The area of the stack used for the activation record is called a *stack frame*. Within any function, the stack frame contains the following information:

- Arguments (in excess of six) passed from the calling function.
- The return address back to the calling function.
- The calling function's frame pointer.
- Local variables for the current function.

and often includes:

- Copies of arguments passed in registers.
- Copies of values in the registers that must be preserved by a function — `rbx`, `r12` – `r15`.

Some general memory usage rules (64-bit mode) are:

- Each argument is passed within an 8-byte unit. For example, passing three char values requires three registers. This 8-byte rule also applies to arguments passed on the stack.

- Local variables can be allocated to take up only the amount of memory they require. For example, three `char` values can be accommodated in a three-byte memory area.
- The address in the frame pointer (`rbp` register) must always be a multiple of sixteen. It should never be changed within a function, except during the prologue and epilogue.
- The address in the stack pointer (`rsp` register) must always be a multiple of sixteen before transferring program flow to another function.

We can see how this works by studying the program in Listing 11.1.

---

```
1 /*  
2  * addProg.c  
3  * Adds two integers  
4  * Bob Plantz - 13 June 2009  
5  */  
6  
7 #include <stdio.h>  
8 #include "sumInts1.h"  
9  
10 int main(void)  
11 {  
12     int x, y, z;  
13     int overflow;  
14  
15     printf("Enter two integers: ");  
16     scanf("%i %i", &x, &y);  
17     overflow = sumInts(x, y, &z);  
18     printf("%i + %i = %i\n", x, y, z);  
19     if (overflow)  
20         printf("!!! Overflow occurred !!!\n");  
21  
22     return 0;  
23 }  
  
1 /*  
2  * sumInts1.h  
3  * Returns N + (N-1) + ... + 1  
4  * Bob Plantz - 4 June 2008  
5  */  
6  
7 #ifndef SUMINTS1_H  
8 #define SUMINTS1_H  
9 int sumInts(int, int, int *);  
10 #endif  
  
1 /*  
2  * sumInts1.c  
3  * Adds two integers and outputs their sum.  
4  * Returns 0 if no overflow, else returns 1.  
5  * Bob Plantz - 13 June 2009  
6  */  
7
```

---

```

8 #include "sumInts1.h"
9
10 int sumInts(int a, int b, int *sum)
11 {
12     int overflow = 0;    // assume no overflow
13
14     *sum = a + b;
15
16     if (((a > 0) && (b > 0) && (*sum < 0)) ||
17         ((a < 0) && (b < 0) && (*sum > 0)))
18     {
19         overflow = 1;
20     }
21     return overflow;
22 }
```

Listing 11.1: Passing arguments to a function (C). (There are three files here.)

The compiler-generated assembly language for the `sumInts` function is shown in Listing 11.2 with comments added.

```

1      .file  "sumInts1.c"
2      .text
3      .globl sumInts
4      .type sumInts, @function
5 sumInts:
6      pushq %rbp
7      movq %rsp, %rbp
8      movl %edi, -20(%rbp)    # save a
9      movl %esi, -24(%rbp)    # save b
10     movq %rdx, -32(%rbp)   # save pointer to sum
11     movl $0, -4(%rbp)       # overflow = 0;
12     movl -24(%rbp), %eax   # load b
13     movl -20(%rbp), %edx   # load a
14     addl %eax, %edx        # add them
15     movq -32(%rbp), %rax   # load address of sum
16     movl %edx, (%rax)       # *sum = a + b;
17     cmpl $0, -20(%rbp)
18     jle .L2
19     cmpl $0, -24(%rbp)
20     jle .L2
21     movq -32(%rbp), %rax
22     movl (%rax), %eax
23     testl %eax, %eax
24     js .L3
25 .L2:
26     cmpl $0, -20(%rbp)
27     jns .L4
28     cmpl $0, -24(%rbp)
29     jns .L4
30     movq -32(%rbp), %rax
31     movl (%rax), %eax
32     testl %eax, %eax
33     jle .L4
```

```

34 .L3:
35     movl    $1, -4(%rbp)
36 .L4:
37     movl    -4(%rbp), %eax    # return overflow;
38     popq    %rbp
39     ret
40     .size   sumInts, .-sumInts
41     .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
42     .section .note.GNU-stack, "",@progbits

```

Listing 11.2: Accessing arguments in the `sumInts` function from Listing 11.1 (gcc assembly language).

As we go through this description, it is very easy to confuse the *frame pointer* (`rbp` register) and the *stack pointer* (`rsp` register). They each are used to access different areas of the stack.

- The *frame pointer* (`rbp` register) remains unchanged. It is used to access the area of the stack that *belongs to the current function*, including local variables and arguments passed into the current function.
- The *stack pointer* (`rsp` register) can be changed. It is used to *create a new stack frame* for a function about to be called, including storing the return address and passing arguments beyond the first six.

After saving the caller's frame pointer and establishing its own frame pointer, this function stores the argument values in the local variable area:

```

5 sumInts:
6     pushq  %rbp
7     movq    %rsp, %rbp
8     movl    %edi, -20(%rbp)  # save a
9     movl    %esi, -24(%rbp)  # save b
10    movq   %rdx, -32(%rbp)  # save pointer to sum
11    movl    $0, -4(%rbp)     # overflow = 0;

```

The arguments are in the following registers (see Table 8.2, page 174):

- `a` is in `edi`.
- `b` is in `esi`.
- The pointer to `sum` is in `rdx`.

Storing them in the local variable area frees up the registers so they can be used in this function. Although this is not very efficient, the compiler does not need to work very hard to optimize register usage within the function. The only local variable, `overflow`, is initialized on line 11.

The observant reader will note that no memory has been allocated on the stack for local variables or saving the arguments. The ABI [25] defines the 128 bytes beyond the stack pointer — that is, the 128 bytes at addresses lower than the one in the `rsp` register — as a *red zone*. The operating system is not allowed to use this area, so the function can use it for temporary storage of values that do not need to be saved when another function is called. In particular, *leaf functions* can store local variables in this area without moving the stack pointer because they do not call other functions.

Notice that both the argument save area and the local variable area are aligned on 16-byte address boundaries. Figure 11.1 provides a pictorial view of where the three arguments and the local variable are in the red zone.

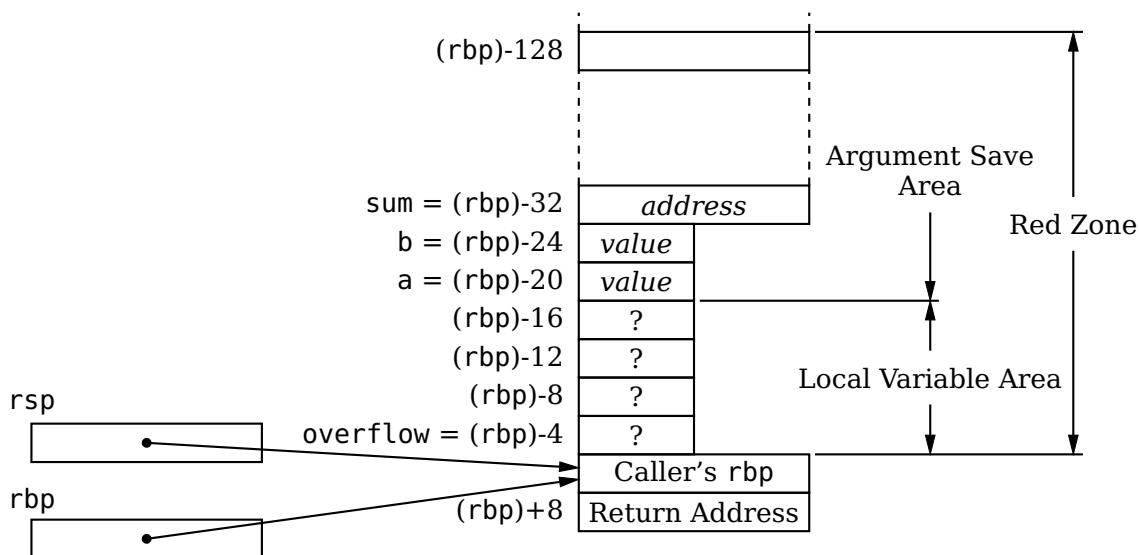


Figure 11.1: Arguments and local variables in the stack frame, `sumInts` function. The two input values and the address for the output are passed in registers, then stored in the Argument Save Area by the called function. Since this is a leaf function, the Red Zone is used for this function’s stack frame.

As you know, some functions take a variable number of arguments. In these functions, the ABI [25] specifies the relative offsets of the register save area. The offsets are shown in Table 11.1.

Register	Offset
rdi	0
rsi	8
rdx	16
rcx	24
r8	32
r9	40
xmm0	48
xmm1	64
...	...
xmm15	288

Table 11.1: Argument register save area in stack frame. These relative offsets should be used in functions with a variable number of arguments.

One of the problems with the C version of `sumInts` is that it requires a separate check for overflow:

```

16 sumInts:
17     if (((a > 0) && (b > 0) && (*sum < 0)) ||
18         ((a < 0) && (b < 0) && (*sum > 0)))
19 {

```

```
20     overflow = 1;
21 }
```

Writing the function in assembly language allows us to directly check the overflow flag, as shown in Listing 11.3.

---

```
1 # sumInts.s
2 # Adds two 32-bit integers. Returns 0 if no overflow
3 # else returns 1
4 # Bob Plantz - 13 June 2009
5 # Calling sequence:
6 #     rdx <- address of output
7 #     esi <- 1st int to be added
8 #     edi <- 2nd int to be added
9 #     call    sumInts
10 #    returns 0 if no overflow, else returns 1
11 # Read only data
12 .section .rodata
13 overflow:
14     .word 1
15 # Code
16 .text
17 .globl sumInts
18 .type sumInts, @function
19 sumInts:
20     pushq %rbp          # save caller's frame pointer
21     movq %rsp, %rbp      # establish our frame pointer
22
23     movl $0, %eax        # assume no overflow
24     addl %edi, %esi      # add values
25     cmovo overflow, %eax # overflow occurred
26     movl %esi, (%rdx)    # output sum
27
28     movq %rbp, %rsp      # restore stack pointer
29     popq %rbp            # restore caller's frame pointer
30     ret
```

Listing 11.3: Accessing arguments in the `sumInts` function from Listing 11.1 (programmer assembly language)

The code to perform the addition and overflow check is much simpler.

```
23     movl $0, %eax        # assume no overflow
24     addl %edi, %esi      # add values
25     cmovo overflow, %eax # overflow occurred
26     movl %esi, (%rdx)    # output sum
```

The body of the function begins by assuming there will not be overflow, so 0 is stored in `eax`, ready to be the return value. The value of the first argument is added to the second, because the programmer realizes that the values in the argument registers do not need to be saved. If this addition produces overflow, the `cmovo` instruction changes the return value to 1. Finally, in either case the sum is stored at the memory location whose address was passed to the function as the third argument.

## 11.2 More Than Six Arguments, 64-Bit Mode

When a calling function needs to pass more than six arguments to another function, the additional arguments beyond the first six are passed on the call stack. They are effectively pushed onto the stack in eight-byte chunks before the call. The order of pushing is from right to left in the C argument list. (As you will see shortly the compiler actually uses a more efficient method than pushes.) Since these arguments are on the call stack, they are within the called function's stack frame, so the called function can access them.

Consider the program in Listing 11.4.

---

```
1 /*  
2  * nineInts1.c  
3  * Declares and adds nine integers.  
4  * Bob Plantz - 13 June 2009  
5 */  
6 #include <stdio.h>  
7 #include "sumNine1.h"  
8  
9 int main(void)  
10 {  
11     int total;  
12     int a = 1;  
13     int b = 2;  
14     int c = 3;  
15     int d = 4;  
16     int e = 5;  
17     int f = 6;  
18     int g = 7;  
19     int h = 8;  
20     int i = 9;  
21  
22     total = sumNine(a, b, c, d, e, f, g, h, i);  
23     printf("The sum is %i\n", total);  
24     return 0;  
25 }  
  
1 /*  
2  * sumNine1.h  
3  * Computes sum of nine integers.  
4  * Bob Plantz - 13 June 2009  
5 */  
6 #ifndef SUMNINE_H  
7 #define SUMNINE_H  
8 int sumNine(int one, int two, int three, int four, int five,  
9             int six, int seven, int eight, int nine);  
10 #endif  
  
1 /*  
2  * sumNine1.c  
3  * Computes sum of nine integers.  
4  * Bob Plantz - 13 June 2009  
5 */
```

---

```

6 #include <stdio.h>
7 #include "sumNine1.h"
8
9 int sumNine(int one, int two, int three, int four, int five,
10             int six, int seven, int eight, int nine)
11 {
12     int x;
13
14     x = one + two + three + four + five + six
15         + seven + eight + nine;
16     printf("sumNine done.\n");
17     return x;
18 }
```

Listing 11.4: Passing more than six arguments to a function (C). (There are three files here.)

The assembly language generated by `gcc` from the program in Listing 11.4 is shown in Listing 11.5, with comments added to explain parts of the code.

```

1      .file   "nineInts1.c"
2      .section .rodata
3 .LC0:
4          .string "The sum is %i\n"
5
6 .text
7 .globl main
8 .type main, @function
main:
9     pushq %rbp
10    movq %rsp, %rbp
11    subq $80, %rsp
12    movl $1, -40(%rbp)
13    movl $2, -36(%rbp)
14    movl $3, -32(%rbp)
15    movl $4, -28(%rbp)
16    movl $5, -24(%rbp)
17    movl $6, -20(%rbp)
18    movl $7, -16(%rbp)
19    movl $8, -12(%rbp)
20    movl $9, -8(%rbp)
21    movl -20(%rbp), %r9d      # f is 6th argument
22    movl -24(%rbp), %r8d      # e is 5th argument
23    movl -28(%rbp), %ecx      # d is 4th argument
24    movl -32(%rbp), %edx      # c is 3rd argument
25    movl -36(%rbp), %esi      # b is 2nd argument
26    movl -40(%rbp), %eax      # load a
27    movl -8(%rbp), %edi       # load i
28    movl %edi, 16(%rsp)        # insert on stack
29    movl -12(%rbp), %edi       # load h
30    movl %edi, 8(%rsp)        # insert on stack
31    movl -16(%rbp), %edi       # load g
32    movl %edi, (%rsp)          # insert on stack
33    movl %eax, %edi           # a is 1st argument
34    call sumNine
```

```

35      movl    %eax, -4(%rbp)
36      movl    -4(%rbp), %eax
37      movl    %eax, %esi
38      movl    $.LC0, %edi
39      movl    $0, %eax
40      call    printf
41      movl    $0, %eax
42      leave
43      ret
44      .size   main, .-main
45      .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
46      .section .note.GNU-stack,"",@progbits

```

```

1      .file   "sumNine1.c"
2      .section .rodata
3      .LC0:
4          .string "sumNine done."
5          .text
6          .globl  sumNine
7          .type   sumNine, @function
8      sumNine:
9          pushq  %rbp
10         movq   %rsp, %rbp
11         subq   $48, %rsp
12         movl   %edi, -20(%rbp) # save one
13         movl   %esi, -24(%rbp) # save two
14         movl   %edx, -28(%rbp) # save three
15         movl   %ecx, -32(%rbp) # save four
16         movl   %r8d, -36(%rbp) # save five
17         movl   %r9d, -40(%rbp) # save six
18         movl   -24(%rbp), %eax # load two
19         movl   -20(%rbp), %edx # load one, subtotal
20         addl   %eax, %edx # add two to subtotal
21         movl   -28(%rbp), %eax # load three
22         addl   %eax, %edx # add to subtotal
23         movl   -32(%rbp), %eax # load four
24         addl   %eax, %edx # add to subtotal
25         movl   -36(%rbp), %eax # load five
26         addl   %eax, %edx # add to subtotal
27         movl   -40(%rbp), %eax # load six
28         addl   %eax, %edx # add to subtotal
29         movl   16(%rbp), %eax # load seven
30         addl   %eax, %edx # add to subtotal
31         movl   24(%rbp), %eax # load eight
32         addl   %eax, %edx # add to subtotal
33         movl   32(%rbp), %eax # load nine
34         addl   %edx, %eax # add to subtotal
35         movl   %eax, -4(%rbp) # x <- total
36         movl   $.LC0, %edi
37         call    puts
38         movl   -4(%rbp), %eax
39         leave

```

```

40      ret
41      .size    sumNine, .-sumNine
42      .ident   "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
43      .section .note.GNU-stack,"",@progbits

```

Listing 11.5: Passing more than six arguments to a function (gcc assembly language).  
(There are two files here.)

Before main calls sumNine the values of the second through sixth arguments, b - f, are moved to the appropriate registers, and the first argument, a is loaded into a temporary register:

```

21      movl    -20(%rbp), %r9d    # f is 6th argument
22      movl    -24(%rbp), %r8d    # e is 5th argument
23      movl    -28(%rbp), %ecx    # d is 4th argument
24      movl    -32(%rbp), %edx    # c is 3rd argument
25      movl    -36(%rbp), %esi    # b is 2nd argument
26      movl    -40(%rbp), %eax    # load a

```

The the values of the seventh, eighth, and ninth arguments, g - i, are moved to their appropriate locations on the call stack. Enough space was allocated at the beginning of the function to allow for these arguments. They are moved into their correct locations on lines 27 - 32:

```

27      movl    -8(%rbp), %edi    # load i
28      movl    %edi, 16(%rsp)    # insert on stack
29      movl    -12(%rbp), %edi    # load h
30      movl    %edi, 8(%rsp)    # insert on stack
31      movl    -16(%rbp), %edi    # load g
32      movl    %edi, (%rsp)    # insert on stack

```

The stack pointer, `rsp`, is used as the reference point for storing the arguments on the stack here because the main function is starting a new stack frame for the function it is about to call, `sumNine`.

Then the first argument, a, is moved to the appropriate register:

```
33      movl    %eax, %edi        # a is 1st argument
```

When program control is transferred to the `sumNine` function, the partial stack frame appears as shown in Figure 11.2. Even though each argument is only four bytes (`int`), each is passed in an 8-byte portion of stack memory. Compare this with passing arguments in registers; only one data item is passed per register even if the data item does not take up the entire eight bytes in the register. The return address is at the top

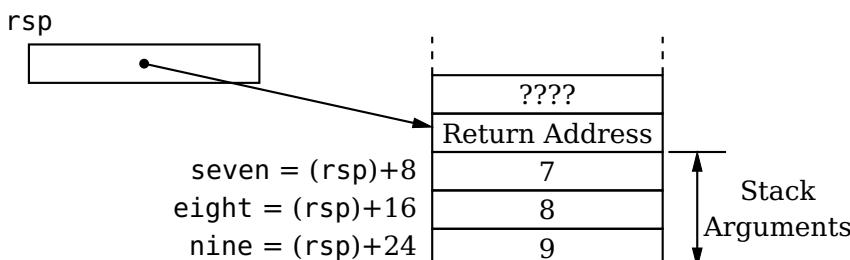


Figure 11.2: Arguments 7 - 9 are passed on the stack to the `sumNine` function. State of the stack when control is first transferred to this function.

of the stack, immediately followed by the three arguments (beyond the six passed in registers). Notice that each argument is in the same position on the stack as it would have been if it had been pushed onto the stack just before the call instruction. Since the address in the stack pointer (`rsp`) was 16-byte aligned before the call to this function, and the `call` instruction pushed the 8-byte return address onto the stack, the address in `rsp` is now 8-byte aligned.

The prologue of `sumNine` completes the stack frame. Then the function saves the register arguments in the register save area of the stack frame:

```

9      pushq  %rbp
10     movq   %rsp, %rbp
11     subq   $48, %rsp
12     movl   %edi, -20(%rbp)  # save one
13     movl   %esi, -24(%rbp)  # save two
14     movl   %edx, -28(%rbp)  # save three
15     movl   %ecx, -32(%rbp)  # save four
16     movl   %r8d, -36(%rbp)  # save five
17     movl   %r9d, -40(%rbp)  # save six

```

The state of the stack frame at this point is shown in Figure 11.3.

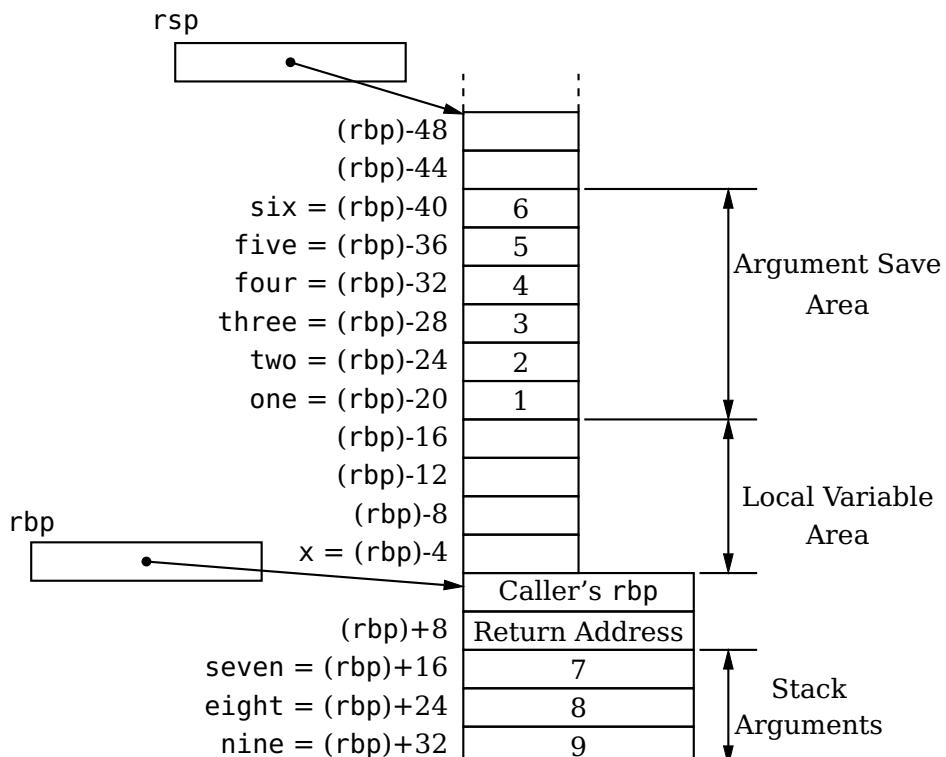


Figure 11.3: Arguments and local variables in the stack frame, `sumNine` function. The first six arguments are passed in registers but saved in the stack frame. Arguments beyond six are passed in the portion of the stack frame that is created by the calling function.

You may question why the compiler did not simply use the red zone. The `sumNine` function is not a leaf function. It calls another function, which may require use of the

call stack. So space must be explicitly allocated on the call stack for local variables and the register argument save areas.

By the way, the compiler has replaced this function call, a call to `printf`, with a call to `puts`:

```
36      movl    $.LC0, %edi
37      call    puts
```

Since the only thing to be written to the screen is a text string, the `puts` function is equivalent.

After the register arguments are safely stored in the argument save area, they can be easily summed and the total saved in the local variable:

```
18     movl    -24(%rbp), %eax # load two
19     movl    -20(%rbp), %edx # load one, subtotal
20     addl    %eax, %edx    # add two to subtotal
21     movl    -28(%rbp), %eax # load three
22     addl    %eax, %edx    # add to subtotal
23     movl    -32(%rbp), %eax # load four
24     addl    %eax, %edx    # add to subtotal
25     movl    -36(%rbp), %eax # load five
26     addl    %eax, %edx    # add to subtotal
27     movl    -40(%rbp), %eax # load six
28     addl    %eax, %edx    # add to subtotal
29     movl    16(%rbp), %eax # load seven
30     addl    %eax, %edx    # add to subtotal
31     movl    24(%rbp), %eax # load eight
32     addl    %eax, %edx    # add to subtotal
33     movl    32(%rbp), %eax # load nine
34     addl    %edx, %eax    # add to subtotal
35     movl    %eax, -4(%rbp) # x <- total
```

Notice that the seventh, eighth, and ninth arguments are accessed by *positive* offsets from the frame pointer, `rbp`. They were stored in the stack frame by the calling function. The called function “owns” the entire stack frame so it does not need to make additional copies of these arguments.

It is important to realize that once the stack frame has been completed within a function, that area of the call stack cannot be treated as a stack. That is, it cannot be accessed through pushes and pops. It must be treated as a record. (You will learn more about records in Section 13.2, page 333.)

If we were to recompile these functions with higher levels of optimization, many of these assembly language operations would be removed (see Exercise 11-2). But the point here is to examine the mechanisms that can be used to work with arguments and to write easily read code, so we study the unoptimized code.

A version of this program written in assembly language is shown in Listing 11.6.

---

```
1 # nineInts2.s
2 # Demonstrate how integral arguments are passed in 64-bit mode.
3 # Bob Plantz - 13 June 2009
4
5 # Stack frame
6 #   passing arguments on stack (rsp)
7 #   need 3x8 = 24 -> 32 bytes
8     .equ    seventh,0
9     .equ    eighth,8
```

```

10      .equ    ninth,16
11 # local vars (rbp)
12 # need 10x4 = 40 -> 48 bytes
13     .equ    i,-4
14     .equ    h,-8
15     .equ    g,-12
16     .equ    f,-16
17     .equ    e,-20
18     .equ    d,-24
19     .equ    c,-28
20     .equ    b,-32
21     .equ    a,-36
22     .equ    total,-40
23     .equ    localSize,-80
24 # Read only data
25     .section .rodata
26 format:
27     .string "The sum is %i\n"
28 # Code
29     .text
30     .globl main
31     .type main, @function
main:
32     pushq %rbp          # save caller's base pointer
33     movq %rsp, %rbp       # establish ours
34     addq $localSize, %rsp # space for local variables
35                               # + argument passing
36     movl $1, a(%rbp)      # initialize local variables
37     movl $2, b(%rbp)
38     movl $3, c(%rbp)
39     movl $4, d(%rbp)
40     movl $5, e(%rbp)
41     movl $6, f(%rbp)
42     movl $7, g(%rbp)
43     movl $8, h(%rbp)
44     movl $9, i(%rbp)
45
46     movl f(%rbp), %edx    # load f
47     movl e(%rbp), %ecx    # ....
48     movl d(%rbp), %esi
49     movl c(%rbp), %edi
50     movl b(%rbp), %r10d
51     movl a(%rbp), %r11d    # load a
52
53     movl i(%rbp), %eax    # load i
54     movl %eax, ninth(%rsp) # 9th argument
55     movl h(%rbp), %eax    # load h
56     movl %eax, eighth(%rsp) # 8th argument
57     movl g(%rbp), %eax    # load g
58     movl %eax, seventh(%rsp) # 7th argument
59     movl %edx, %r9d        # f is 6th
60     movl %ecx, %r8d        # e is 5th
61

```

```

62      movl    %esi, %ecx          # d is 4th
63      movl    %edi, %edx          # c is 3rd
64      movl    %r10d, %esi          # b is 2nd
65      movl    %r11d, %edi          # a is 1st
66      call    sumNine
67      movl    %eax, total(%rbp)    # total = nineInts(...)
68
69      movl    total(%rbp), %esi
70      movl    $format, %edi
71      movl    $0, %eax
72      call    printf
73
74      movl    $0, %eax          # return 0;
75      movq    %rbp, %rsp          # delete locals
76      popq    %rbp              # restore caller's base pointer
77      ret

```

```

1 # sumNine2.s
2 # Sums nine integer arguments and returns the total.
3 # Bob Plantz - 13 June 2009
4
5 # Stack frame
6 #   arguments already in stack frame
7     .equ    seven,16
8     .equ    eight,24
9     .equ    nine,32
10 #  local variables
11     .equ    total,-4
12     .equ    localSize,-16
13 # Read only data
14     .section .rodata
15 doneMsg:
16     .string "sumNine done"
17 # Code
18     .text
19     .globl  sumNine
20     .type   sumNine, @function
21 sumNine:
22     pushq  %rbp              # save caller's base pointer
23     movq    %rsp, %rbp          # set our base pointer
24     addq    $localSize, %rsp    # for local variables
25
26     addl    %esi, %edi          # add two to one
27     addl    %ecx, %edi          # plus three
28     addl    %edx, %edi          # plus four
29     addl    %r8d, %edi          # plus five
30     addl    %r9d, %edi          # plus six
31     addl    seven(%rbp), %edi  # plus seven
32     addl    eight(%rbp), %edi  # plus eight
33     addl    nine(%rbp), %edi   # plus nine
34     movl    %edi, total(%rbp)  # save total
35

```

```

36      movl    $doneMsg, %edi
37      call    puts
38
39      movl    total(%rbp), %eax # return total;
40      movq    %rbp, %rsp      # delete local vars.
41      popq    %rbp          # restore caller's base pointer
42      ret

```

Listing 11.6: Passing more than six arguments to a function (programmer assembly language). (There are two files here.)

The assembly language programmer realizes that all nine integers can be summed in the `sumNine` function before it calls another function. In addition, none of the values will be needed after this summation. So there is no reason to store the register arguments locally:

```

26      addl    %esi, %edi      # add two to one
27      addl    %ecx, %edi      # plus three
28      addl    %edx, %edi      # plus four
29      addl    %r8d, %edi      # plus five
30      addl    %r9d, %edi      # plus six
31      addl    seven(%rbp), %edi # plus seven
32      addl    eight(%rbp), %edi # plus eight
33      addl    nine(%rbp), %edi # plus nine

```

However, the `edi` register will be needed for passing an argument to `puts`, so the total is saved in a local variable in the stack frame:

```
34      movl    %edi, total(%rbp) # save total
```

Then it is loaded into `eax` for return to the calling function:

```
39      movl    total(%rbp), %eax # return total;
```

The overall pattern of a stack frame is shown in Figure 11.4. The `rbp` register serves as the frame pointer to the stack frame. Once the frame pointer address has been established in a function, its value must never be changed. The return address is always located +8 bytes offset from the frame pointer. Arguments to the function are positive offsets from the frame pointer, and local variables are negative offsets from the frame pointer.

It is essential that you follow the register usage and argument passing disciplines precisely. Any deviation can cause errors that are very difficult to debug.

1. In the *calling function*:

- (a) Assume that the values in the `rax`, `rcx`, `rdx`, `rsi`, `rdi` and `r8 - r11` registers will be changed by the called function.
- (b) The first six arguments are passed in the `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9` registers in left-to-right order.
- (c) Arguments beyond six are stored on the stack as though they had been pushed onto the stack in right-to-left order.
- (d) Use the `call` instruction to invoke the function you wish to call.

2. Upon *entering the called function*:

- (a) Save the caller's frame pointer by pushing `rbp` onto the stack.

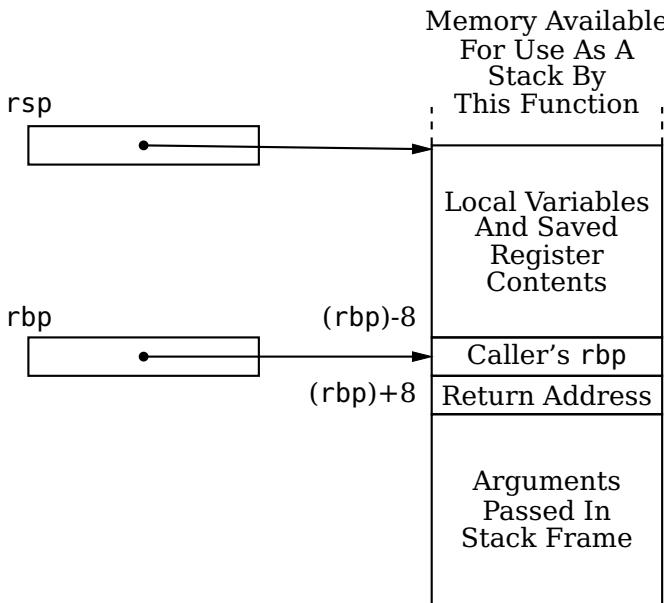


Figure 11.4: Overall layout of the stack frame.

- Establish a new frame pointer at the current top of stack by copying `rsp` to `rbp`.
- Allocate space on the stack for all the local variables, plus any required register save space, by subtracting the number of bytes required from `rsp`; this value must be a multiple of sixteen.
- If a called function changes any of the values in the `rbx`, `rbp`, `rsp`, or `r12 - r15` registers, they must be saved in the register save area, then restored before returning to the calling function.
- If the function calls another function, save the arguments passed in registers on the stack.

### 3. Within the *called* function:

- `rsp` is pointing to the current bottom of the stack that is accessible to this function. Observe the usual stack discipline (see §8.2). In particular, DO NOT use the stack pointer to access arguments or local variables.
- Arguments passed in registers to the function and saved on the stack are accessed by negative offsets from the frame pointer, `rbp`.
- Arguments passed on the stack to the function are accessed by positive offsets from the frame pointer, `rbp`.
- Local variables are accessed by negative offsets from the frame pointer, `rbp`.

### 4. When leaving the *called* function:

- Place the return value, if any, in `eax`.
- Restore the values in the `rbx`, `rbp`, `rsp`, and `r12 - r15` registers from the register save area in the stack frame.
- Delete the local variable space and register save area by copying `rbp` to `rsp`.

- (d) Restore the caller's frame pointer by popping rbp off the stack save area.
- (e) Return to calling function with ret.

The best way to design a stack frame for a function is to make a drawing on paper following the pattern in Figure 11.3. Show all the local variables and arguments to the function. To be safe, assume that all the register-passed arguments will be saved in the function. Compute and write down all the offset values on your drawing. When writing the source code for your function, use the .equ directive to give meaningful names to each of the numerical offsets. If you do this planning *before* writing the executable code, you can simply use the `name(%rbp)` syntax to access the value stored at `name`.

## 11.3 Interface Between Functions, 32-Bit Mode

In 32-bit mode, all arguments are passed on the call stack. The 32-bit assembly language generated by gcc is shown in Listing 11.7.

---

```

1      .file   "nineInts1.c"
2      .section     .rodata
3 .LC0:
4          .string  "The sum is %i\n"
5          .text
6          .globl   main
7          .type    main, @function
8 main:
9          pushl   %ebp
10         movl    %esp, %ebp
11         andl    $-16, %esp
12         subl    $96, %esp
13         movl    $1, 56(%esp)
14         movl    $2, 60(%esp)
15         movl    $3, 64(%esp)
16         movl    $4, 68(%esp)
17         movl    $5, 72(%esp)
18         movl    $6, 76(%esp)
19         movl    $7, 80(%esp)
20         movl    $8, 84(%esp)
21         movl    $9, 88(%esp)
22         movl    88(%esp), %eax      # load i
23         movl    %eax, 32(%esp)    # store in stack frame
24         movl    84(%esp), %eax      # load h
25         movl    %eax, 28(%esp)    # store in stack frame
26         movl    80(%esp), %eax      # load g
27         movl    %eax, 24(%esp)    # etc....
28         movl    76(%esp), %eax      # load f
29         movl    %eax, 20(%esp)
30         movl    72(%esp), %eax      # load e
31         movl    %eax, 16(%esp)
32         movl    68(%esp), %eax      # load d
33         movl    %eax, 12(%esp)
34         movl    64(%esp), %eax      # load c
35         movl    %eax, 8(%esp)
36         movl    60(%esp), %eax      # load b

```

```
37      movl    %eax, 4(%esp)
38      movl    56(%esp), %eax      # load a
39      movl    %eax, (%esp)       # store in stack frame
40      call    sumNine
41      movl    %eax, 92(%esp)     # total <- sum
42      movl    92(%esp), %eax
43      movl    %eax, 4(%esp)
44      movl    $.LC0, (%esp)
45      call    printf
46      movl    $0, %eax
47      leave
48      ret
49      .size   main, .-main
50      .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
51      .section .note.GNU-stack,"",@progbits
```

---

```
1      .file   "sumNine1.c"
2      .section .rodata
3      .LC0:
4          .string "sumNine done."
5          .text
6          .globl  sumNine
7          .type   sumNine, @function
8      sumNine:
9          pushl  %ebp
10         movl   %esp, %ebp
11         subl   $40, %esp
12         movl   12(%ebp), %eax      # load two
13         movl   8(%ebp), %edx      # load one, subtotal
14         addl   %eax, %edx        # add two
15         movl   16(%ebp), %eax      # load three
16         addl   %eax, %edx        # add to subtotal
17         movl   20(%ebp), %eax      # load four
18         addl   %eax, %edx        # etc...
19         movl   24(%ebp), %eax      # load five
20         addl   %eax, %edx
21         movl   28(%ebp), %eax      # load six
22         addl   %eax, %edx
23         movl   32(%ebp), %eax      # load seven
24         addl   %eax, %edx
25         movl   36(%ebp), %eax      # load eight
26         addl   %eax, %edx
27         movl   40(%ebp), %eax      # load nine
28         addl   %edx, %eax        # total
29         movl   %eax, -12(%ebp)     # x <- total
30         movl   $.LC0, (%esp)
31         call    puts
32         movl   -12(%ebp), %eax    # return x;
33         leave
34         ret
35         .size   sumNine, .-sumNine
36         .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
```

```
.section .note.GNU-stack, "",@progbits
```

Listing 11.7: Passing more than six arguments to a function (gcc assembly language, 32-bit). (There are two files here.)

The argument passing sequence can be seen on lines 22 – 39 in the `main` function. Rather than pushing each argument onto the stack, the compiler has used the technique of allocating space on the stack for the arguments, then storing each argument directly in the appropriate location. The result is the same as if they had been pushed onto the stack, but the direct storage technique is more efficient.

I find it odd that the compiler writer has chosen to set up a base pointer in `ebp` but not used it in this function. This is NOT a recommended technique when writing in assembly language.

The state of the call stack just before calling the `nineInts` function is shown in Figure 11.5. Comparing this with the 64-bit version in Figure 11.3, we see that the local variables are treated in essentially the same way. But the 32-bit version differs in the way it passes arguments:

- All the arguments are passed on the call stack, none in registers.
- Arguments are passed in 4-byte blocks.

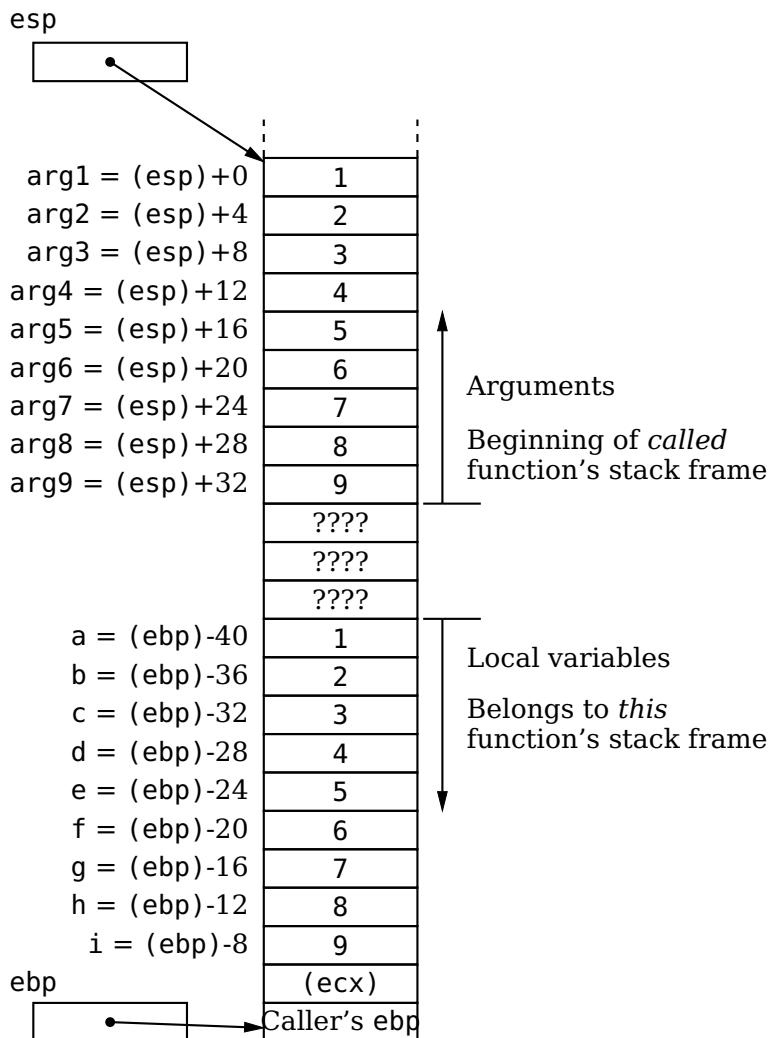


Figure 11.5: Calling function's stack frame, 32-bit mode. Local variables are accessed relative to the frame pointer (ebp register). In this example, they are all 4-byte values. Arguments are accessed relative to the stack pointer (esp register). Arguments are passed in 4-byte blocks.

## 11.4 Instructions Introduced Thus Far

This summary shows the assembly language instructions introduced thus far in the book. The page number where the instruction is explained in more detail, which may be in a subsequent chapter, is also given. This book provides only an introduction to the usage of each instruction. You need to consult the manuals ([2] - [6], [14] - [18]) in order to learn all the possible uses of the instructions.

## 11.4.1 Instructions

*data movement:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
cmovecc	%reg/mem	%reg	conditional move	260
movs	\$imm/%reg	%reg/mem	move	156
movss	\$imm/%reg	%reg/mem	move, sign extend	245
movzs	\$imm/%reg	%reg/mem	move, zero extend	245
popw		%reg/mem	pop from stack	181
pushw	\$imm/%reg/mem		push onto stack	181

*s = b, w, l, q; w = l, q; cc = condition codes*

*arithmetic/logic:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
adds	\$imm/%reg	%reg/mem	add	214
adds	mem	%reg	add	214
cmps	\$imm/%reg	%reg/mem	compare	237
cmps	mem	%reg	compare	237
decs	%reg/mem		decrement	249
incs	%reg/mem		increment	248
leaw	mem	%reg	load effective address	191
subs	\$imm/%reg	%reg/mem	subtract	215
subs	mem	%reg	subtract	215
tests	\$imm/%reg	%reg/mem	test bits	238
tests	mem	%reg	test bits	238

*s = b, w, l, q; w = l, q*

*program flow control:*

<b>opcode</b>	<b>location</b>	<b>action</b>	<b>see page:</b>
call	label	call function	173
ja	label	jump above (unsigned)	239
jae	label	jump above/equal (unsigned)	239
jb	label	jump below (unsigned)	239
jbe	label	jump below/equal (unsigned)	239
je	label	jump equal	239
jg	label	jump greater than (signed)	240
jge	label	jump greater than/equal (signed)	240
jl	label	jump less than (signed)	240
jle	label	jump less than/equal (signed)	240
jmp	label	jump	241
jne	label	jump not equal	239
jno	label	jump no overflow	239
jcc	label	jump on condition codes	239
leave		undo stack frame	192
ret		return from function	192
syscall		call kernel function	201

*cc = condition codes*

## 11.4.2 Addressing Modes

<b>register direct:</b>	The data value is located in a CPU register. <i>syntax:</i> name of the register with a "%" prefix. <i>example:</i> movl %eax, %ebx
<b>immediate data:</b>	The data value is located immediately after the instruction. Source operand only. <i>syntax:</i> data value with a "\$" prefix. <i>example:</i> movl \$0xabcd1234, %ebx
<b>base register plus offset:</b>	The data value is located in memory. The address of the memory location is the sum of a value in a base register plus an offset value. <i>syntax:</i> use the name of the register with parentheses around the name and the offset value immediately before the left parenthesis. <i>example:</i> movl \$0xaabbccdd, 12(%eax)
<b>rip-relative:</b>	The target is a memory address determined by adding an offset to the current address in the rip register. <i>syntax:</i> a programmer-defined label <i>example:</i> je somePlace

## 11.5 Exercises

**11-1** (§11.2) Enter the program in Listing 11.6. Single-step through the program with `gdb` and record the changes in the `rsp` and `rip` registers and the changes in the stack on paper. Use drawings similar to Figure 11.3.

Note: Each of the two functions should be in its own source file. You can single-step into the subfunction with `gdb` at the `call` instruction in `main`, then single-step back into `main` at the `ret` instruction in `addConst`.

**11-2** (§11.2) Enter the C program in Listing 11.4. Using the “-S” compiler option, compile it with differing levels of optimization, i.e., “-O1, -O2, -O3,” and discuss the assembly language that is generated. Is the optimized code easier or more difficult to read?

**11-3** (§11.2, §10.1) Write the function, `writeStr`, in assembly language. The function takes one argument, a `char *`, which is a pointer to a C-style text string. It displays the text string on the screen. It returns the number of characters displayed.

Demonstrate that your function works correctly by writing a `main` function that calls `writeStr` to display “Hello world” on the screen.

Note that the `main` function will not do anything with the character count that is returned by `writeStr`.

**11-4** (§11.2, §10.1) Write the function, `readLn`, in assembly language. The function takes one argument, a `char *`, which is a pointer to a `char` array for storing a text string. It reads characters from the keyboard and stores them in the array as a C-style text string. It does not store the ‘\n’ character. It returns the number of characters, excluding the NUL character, that were stored in the array.

Demonstrate that your function works correctly by writing a `main` function that prompts the user to enter a text string, then echoes the user’s input.

When testing your program, be careful not to enter more characters than the allocated space. Explain what would occur if you did enter too many characters.

Note that the `main` function will not do anything with the character count that is returned by `readLn`.

**11-5** (§11.2, §10.1) Write a program in assembly language that

- a) prompts the user to enter any text string,
- b) reads the entered text string, and
- c) echoes the user's input.

Use the `writeStr` function from Exercise 11-3 and the `readLn` function from Exercise 11-4 to implement the user interface in this program.

**11-6** (§11.2, §10.1) Modify the `readLn` function in Exercise 11-4 so that it takes a second argument, the maximum length of the text string, including the NULL character. Excess characters entered by the user are discarded.

# Chapter 12

# Bit Operations; Multiplication and Division

We saw in Section 3.5 (page 48) that input read from the keyboard and output written on the screen is in the ASCII code and that integers are stored in the binary number system. So if a program reads user input as, say,  $123_{10}$ , that input is read as the characters '1', '2', and '3', but the value used in the program is represented by the bit pattern  $00000007b_{16}$ .<sup>1</sup> In this chapter, we return to the conversion algorithms between these two storage codes and look at the assembly language that is involved.

## 12.1 Logical Operators

Two numeric operators, addition and subtraction, were introduced in Section 9.2 (page 214). Many data items are better thought of as bit patterns rather than numerical entities. For example, study Table 2.3 on page 22 and see if you can determine which bit determines the case (upper/lower) of the alphabetic characters.

In order to manipulate individual character codes in a text string, we introduce the bit-wise logical operators in this section. The logical operations are shown in the truth tables in Figure 3.4 (page 52). The instructions available to us to perform these three operations are:

ands <i>source, destination</i>	ors <i>source, destination</i>	xors <i>source, destination</i>
---------------------------------	--------------------------------	---------------------------------

where *s* denotes the size of the operand:

<b><i>s</i></b>	<b>meaning</b>	<b>number of bits</b>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

Intel® Syntax	and <i>destination, source</i>	or <i>destination, source</i>	xor <i>destination, source</i>
---------------	--------------------------------	-------------------------------	--------------------------------

<sup>1</sup>Some programs, notably those that do not perform many arithmetic operations, maintain the numbers in the character code. This requires more complex algorithms for performing arithmetic.

For example, the instruction

```
andl    %eax, %edx
```

performs an and operation between each of the respective 32 bits in the eax register with the 32 bits in the edx register, leaving the result in the edx register. The instruction

```
andb    %dh, %ah
```

performs an and operation between each of the respective 8 bits in the dh register with the 8 bits in the ah register, leaving the result in the ah register.

The addressing modes available for the arithmetic operators, add and sub, are also available for the logical operators. For example, if eax contains the bit pattern 0x89abcdef, the instruction

```
andb    $0xee, %ah
```

would change eax to contain 0x89abcdee. If we follow this with the instruction

```
orl    $0x11111111, %eax
```

the bit pattern in eax becomes 0x99bbddff. Finally, if we then use

```
xorw    $0x1111, %ax
```

we end up with 0x89bbccee in eax.

The program in Listing 12.1 shows the use of the C bit-wise logical operators “&” and “|” to change the case of alphabetic characters.

```
1 /*  
2  * upperLower.c  
3  * Converts alphabetic characters to all upper case  
4  * and all lower case.  
5  * Bob Plantz - 14 June 2009  
6 */  
7  
8 #include "writeStr.h"  
9 #include "readLn.h"  
10 #include "toUpper.h"  
11 #include "toLower.h"  
12 #define MAX 50  
13 int main()  
14 {  
15     char stringOrig[MAX];  
16     char stringWork[MAX];  
17  
18     writeStr("Enter some alphabetic characters: ");  
19     readLn(stringOrig, MAX);  
20  
21     writeStr("All upper: ");  
22     toUpper(stringOrig, stringWork);  
23     writeStr(stringWork);  
24     writeStr("\n");  
25  
26     writeStr("All lower: ");  
27     toLower(stringOrig, stringWork);  
28     writeStr(stringWork);  
29     writeStr("\n");
```

```
30     writeStr("Original: ");
31     writeStr(stringOrig);
32     writeStr("\n");
33
34     return 0;
35 }
36 
```

---

```
1 /*
2 * toUpper.h
3 * Converts letters in a C string to upper case.
4 * Bob Plantz - 14 June 2009
5 */
6
7 #ifndef TOUPPER_H
8 #define TOUPPER_H
9 int toUpper(char *, char *);
10#endif
```

---

```
1 /*
2 * toUpper.c
3 * Converts alphabetic letters in a C string to upper case.
4 * Bob Plantz - 14 June 2009
5 */
6
7 #include "toUpper.h"
8 #define UPMASK 0xdf
9
10int toUpper(char *srcPtr, char *destPtr)
11{
12    int count = 0;
13    while (*srcPtr != '\0')
14    {
15        *destPtr = *srcPtr & UPMASK;
16        srcPtr++;
17        destPtr++;
18        count++;
19    }
20    *destPtr = '\0'; // terminate string
21    return count;
22}
```

---

```
1 /*
2 * toLower.h
3 * Converts letters in a C string to lower case.
4 * Bob Plantz - 14 June 2009
5 */
6
7 #ifndef TOLOWER_H
8 #define TOLOWER_H
9 int toLower(char *, char *);
10#endif
```

---

```

1  /*
2   * toLower.c
3   * Converts letters in a C string to lower case.
4   * Bob Plantz - 14 June 2009
5   */
6
7 #include "toLower.h"
8 #define LOWMASK 0x20
9
10 int toLower(char *srcPtr, char *destPtr)
11 {
12     int count = 0;
13     while (*srcPtr != '\0')
14     {
15         *destPtr = *srcPtr | LOWMASK;
16         srcPtr++;
17         destPtr++;
18         count++;
19     }
20     *destPtr = '\0'; // terminate string
21     return count;
22 }
```

**Listing 12.1:** Convert letters to upper/lower case (C). The functions `writeStr` and `readLn` are not shown here; see Exercises 11-3 and 11-4 for the assembly language versions. (There are three files here.)

The program assumes that the user enters all alphabetic characters without making mistakes. Of course, the conversions could be accomplished with addition and subtraction, but in this application the bit-wise logical operators are more natural.

In Listing 12.2 we show only the gcc-generated assembly language for the `main` and `toUpperCase` functions.

---

```

1      .file  "upperLower.c"
2      .section        .rodata
3      .align 8
4 .LC0:    .string "Enter some alphabetic characters: "
5 .LC1:    .string "All upper: "
6 .LC2:    .string "\n"
7 .LC3:    .string "All lower: "
8 .LC4:    .string "Original: "
9
10 .text
11 .globl main
12 .type  main, @function
13
14 main:
15     pushq %rbp
16     movq %rsp, %rbp
17     addq $-128, %rsp
```

```
21     movq    %fs:40, %rax      # load guard value
22     movq    %rax, -8(%rbp)   # store at end of stack
23     xorl    %eax, %eax      # clear rax
24     movl    $.LC0, %edi
25     call    writeStr
26     leaq    -128(%rbp), %rax
27     movl    $50, %esi
28     movq    %rax, %rdi
29     call    readLn
30     movl    $.LC1, %edi
31     call    writeStr
32     leaq    -64(%rbp), %rdx
33     leaq    -128(%rbp), %rax
34     movq    %rdx, %rsi
35     movq    %rax, %rdi
36     call    toUpper
37     leaq    -64(%rbp), %rax
38     movq    %rax, %rdi
39     call    writeStr
40     movl    $.LC2, %edi
41     call    writeStr
42     movl    $.LC3, %edi
43     call    writeStr
44     leaq    -64(%rbp), %rdx
45     leaq    -128(%rbp), %rax
46     movq    %rdx, %rsi
47     movq    %rax, %rdi
48     call    toLower
49     leaq    -64(%rbp), %rax
50     movq    %rax, %rdi
51     call    writeStr
52     movl    $.LC2, %edi
53     call    writeStr
54     movl    $.LC4, %edi
55     call    writeStr
56     leaq    -128(%rbp), %rax
57     movq    %rax, %rdi
58     call    writeStr
59     movl    $.LC2, %edi
60     call    writeStr
61     movl    $0, %eax
62     movq    -8(%rbp), %rdx
63     xorq    %fs:40, %rdx
64     je     .L3
65     call    __stack_chk_fail # check for stack overflow
66 .L3:
67     leave
68     ret
69     .size   main, .-main
70     .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
71     .section .note.GNU-stack, "",@progbits
```

```

1      .file  "toUpper.c"
2      .text
3      .globl toUpper
4      .type toUpper, @function
5 toUpper:
6      pushq %rbp
7      movq %rsp, %rbp
8      movq %rdi, -24(%rbp)    # save srcPtr
9      movq %rsi, -32(%rbp)    # save destPtr
10     movl $0, -4(%rbp)
11     jmp .L2
12 .L3:
13     movq -24(%rbp), %rax    # srcPtr
14     movzbl (%rax), %eax    # load char there
15     movl %eax, %edx
16     andl $-33, %edx        # make upper case
17     movq -32(%rbp), %rax    # destPtr
18     movb %dl, (%rax)        # store char there
19     addq $1, -24(%rbp)       # srcPtr++;
20     addq $1, -32(%rbp)       # destPtr++;
21     addl $1, -4(%rbp)        # count++;
22 .L2:
23     movq -24(%rbp), %rax
24     movzbl (%rax), %eax
25     testb %al, %al          # NUL character?
26     jne .L3
27     movq -32(%rbp), %rax
28     movb $0, (%rax)          # yes, load destPtr
29     movl -4(%rbp), %eax      # and store NUL there
30     popq %rbp
31     ret
32     .size toUpper, ..toUpper
33     .ident "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
34     .section .note.GNU-stack,"",@progbits

```

Listing 12.2: Convert letters to upper/lower case (gcc assembly language). Only two of the functions in Listing 12.1 are shown. (There are two files here.)

The `toLowerCase` function is similar to the `toUpperCase`, and the `writeStr` and `readLn` functions were covered in the exercises in Chapter 11.

Most of the code in Listing 12.2 should be familiar from previous chapters. Note that the C code specifies `char` arrays in the `main` function that are 50 elements long (lines 13 and 14). But the compiler generates assembly language that allocates 64 bytes for each array:

```

17 main:
18     pushq %rbp
19     movq %rsp, %rbp
20     addq $-128, %rsp

```

and:

```

32     leaq -64(%rbp), %rdx
33     leaq -128(%rbp), %rax
34     movq %rdx, %rsi

```

```
35    movq    %rax, %rdi
36    call    toUpper
```

Recall that this 16-byte address alignment is specified by the ABI [25].

The code sequence on lines 21 - 23 in main:

```
21    movq    %fs:40, %rax      # load guard value
22    movq    %rax, -8(%rbp)   # store at end of stack
23    xorl    %eax, %eax       # clear rax
```

is new to you. This code sequence stores a value supplied by the operating system near the end of the stack. The purpose is described in the `gcc` man page entry for the `-fstack-protector` option:

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call `alloca`, and functions with buffers larger than 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits.

The value stored there is checked at the end of the function, on lines 54 - 58:

```
62    movq    -8(%rbp), %rdx
63    xorq    %fs:40, %rdx
64    je     .L3
65    call    __stack_chk_fail # check for stack overflow
66 .L3:
```

If the value has been overwritten, the `__stack_chk_fail` function is called, which notifies the user about the problem.

Your version of `gcc` may be compiled without this option as the default. It can be turned off with the `-fno-stack-protector` option. Since the assembly language we are writing in this book is not “industrial strength,” we will not include this stack protection code.

In the `toUpperCase` function, the compiler-generated assembly language first loads the address stored in the `srcPtr` variable into a register so it can dereference the pointer.

```
13    movq    -24(%rbp), %rax      # srcPtr
```

It then moves the byte at that address into the register, using the `movzbl` instruction to zero out the remaining 24 bits of the register. (Recall that changing the low-order 32 bits of a register also zeros out the high-order 32 bits.)

```
14    movzbl (%rax), %eax      # load char there
```

Next it moves the byte into another working register and then performs the bit-wise and operation with the bit pattern `ffffffd` ( $= -33_{10}$ ), leaving the result in the `edx` register. This and operation leaves all the bits in the `edx` register as they were, except the sixth bit is set to zero. The sixth bit in the ASCII code determines whether a letter is upper or lower case.

```
15    movl    %eax, %edx
16    andl    $-33, %edx        # make upper case
```

Regardless of whether the letter was upper or lower case, it is now upper case. The letter is stored in the low-order eight bits of the `edx` register, the `dl` register. So the program loads the address stored in the `destPtr` variable into a register so it can dereference it and store the character there.

```

17      movq    -32(%rbp), %rax      # destPtr
18      movb    %dl, (%rax)        # store char there

```

We will now consider the version of this program written in assembly language (Listing 12.3).

---

```

1 # upperLower.s
2 # Converts alphabetic characters to all upper case
3 # and all lower case.
4 # Bob Plantz - 14 June 2009
5
6 # Constant
7     .equ    MAX,50
8 # Local variable names
9     .equ    stringOrig,-64          # original char array
10    .equ   stringWork,-128        # working char array
11    .equ   localSize,-128
12 # Read only data
13     .section .rodata
14 prompt: .string "Enter some alphabetic characters: "
15 upMsg:  .string "All upper: "
16 lowMsg: .string "All lower: "
17 origMsg: .string "Original: "
18 endl:   .string "\n"
19 # Code
20     .text
21     .globl main
22     .type  main, @function
23 main:  pushq  %rbp            # save base pointer
24     movq  %rsp, %rbp          # base pointer = current top of stack
25     addq  $localSize, %rsp   # allocate local var. space
26
27     movq  $prompt, %rdi # point to prompt message
28     call   writeStr
29
30     leaq   stringOrig(%rbp), %rdi # place to store string
31     movl   $MAX, %esi           # max number of char
32     call   readLn
33
34     leaq   stringOrig(%rbp), %rdi # original string stored here
35     leaq   stringWork(%rbp), %rsi # modified string goes here
36     call   toLower
37     movq  $lowMsg, %rdi
38     call   writeStr
39     leaq   stringWork(%rbp), %rdi # show modified string
40     call   writeStr
41     movq  $endl, %rdi
42
43
44
45
46
47

```

```

48      call    writeStr
49
50      leaq    stringWork(%rbp), %rdi # original string stored here
51      leaq    stringWork(%rbp), %rsi # modified string goes here
52      call    toUpper
53      movq    $upMsg, %rdi
54      call    writeStr
55      leaq    stringWork(%rbp), %rdi # show modified string
56      call    writeStr
57      movq    $endl, %rdi
58      call    writeStr
59
60      movq    $origMsg, %rdi
61      call    writeStr
62      leaq    stringOrig(%rbp), %rdi # original string stored here
63      call    writeStr
64      movq    $endl, %rdi
65      call    writeStr
66 done:
67      movl    $0, %eax      # return 0;
68      movq    %rbp, %rsp    # remove local variables
69      popq    %rbp          # restore caller base pointer
70      ret                 # back to OS

```

---

```

1 # toUpper.s
2 # Converts alpha characters to upper case.
3 # Bob Plantz - 14 June 2009
4
5 # Calling sequence:
6 #     rdi <- address of string to be converted
7 #     rsi <- address to store result string
8 #     call    toUpper
9 # returns number of characters written
10 # If rdi and rsi have the same address, original string
11 # is overwritten.
12
13 # Useful constant
14 .equ    UPMASK,0xdff
15 # Stack frame, showing local variables and arguments
16 .equ    destPtr,-32
17 .equ    srcPtr,-24
18 .equ    count,-4
19 .equ    localSize,-32
20 # Code
21 .text
22 .globl  toUpper
23 .type   toUpper, @function
24 toUpper:
25     pushq  %rbp          # save frame pointer
26     movq    %rsp, %rbp    # new frame pointer
27     addq    $localSize, %rsp # local vars. and arg.
28

```

```

29      movq    %rdi, srcPtr(%rbp)    # source pointer
30      movq    %rsi, destPtr(%rbp)   # destination pointer
31      movl    $0, count(%rbp)      # count = 0;
32 upLoop:
33      movq    srcPtr(%rbp), %rax # source pointer
34      movb    (%rax), %al       # get current char
35      cmpb    $0, %al           # at end yet?
36      je     done             # yes, all done
37
38      andb    $UPMASK, %al      # in range, convert
39      movq    destPtr(%rbp), %r8 # destination pointer
40      movb    %al, (%r8)        # store character
41
42      incl    count(%rbp)      # count++;
43      incl    srcPtr(%rbp)      # srcPtr++;
44      incl    destPtr(%rbp)      # destPtr++;
45      jmp    upLoop          # and check for end
46 done:
47      movq    destPtr(%rbp), %r8 # destination pointer
48      movb    $0, (%r8)         # store NUL
49      movl    count(%rbp), %eax # return count
50      movq    %rbp, %rsp        # restore stack pointer
51      popq    %rbp            # restore frame pointer
52      ret                 # back to caller

```

---

```

1 # toLower.s
2 # Converts alpha characters to lower case.
3 # Bob Plantz - 14 June 2009
4
5 # Calling sequence:
6 #      rdi <- address of string to be converted
7 #      rsi <- address to store result string
8 #      call  toLower
9 # returns number of characters written
10 # If rdi and rsi have the same address, original string
11 # is overwritten.
12
13 # Useful constant
14     .equ    LOWMASK,0x20
15 # Stack frame, showing local variables and arguments
16     .equ    destPtr,-32
17     .equ    srcPtr,-24
18     .equ    count,-4
19     .equ    localSize,-32
20 # Code
21     .text
22     .globl  toLower
23     .type   toLower, @function
24 toLower:
25     pushq   %rbp            # save frame pointer
26     movq    %rsp, %rbp       # new frame pointer
27     addq    $localSize, %rsp # local vars. and arg.

```

```

28      movq    %rdi, srcPtr(%rbp)    # source pointer
29      movq    %rsi, destPtr(%rbp)   # destination pointer
30      movl    $0, count(%rbp)     # count = 0;
31
32 lowLoop:
33      movq    srcPtr(%rbp), %rax # source pointer
34      movb    (%rax), %al       # get current char
35      cmpb    $0, %al          # at end yet?
36      je     done             # yes, all done
37
38      orb     $LOWMASK, %al     # in range, convert
39      movq    destPtr(%rbp), %r8 # destination pointer
40      movb    %al, (%r8)        # store character
41
42      incl    count(%rbp)      # count++;
43      incl    srcPtr(%rbp)      # srcPtr++;
44      incl    destPtr(%rbp)     # destPtr++;
45      jmp    lowLoop          # and check for end
46 done:
47      movq    destPtr(%rbp), %r8 # destination pointer
48      movb    $0, (%r8)         # store NUL
49      movl    count(%rbp), %eax # return count
50      movq    %rbp, %rsp        # restore stack pointer
51      popq    %rbp             # restore frame pointer
52      ret                 # back to caller

```

Listing 12.3: Convert letters to upper/lower case (programmer assembly language). See Exercises 11.3 and 11.4 for the functions `writeStr` and `readLn`. (There are three files here.)

Again, we will describe on the `toUpperCase` function. Writing directly in assembly language, we also need to get the address in `srcPtr` so we can dereference it. But in copying the character stored there, we simply ignore the remaining 56 bits of the `rax` register. Notice that the `movb` instruction first uses the full 64-bit address in the `rax` register to *fetch* the byte stored there, and it then can write over the low-order 8 bits of the same register. (This, of course, “destroys” the address.)

```

33      movq    srcPtr(%rbp), %rax # source pointer
34      movb    (%rax), %al       # get current char

```

Since we are ignoring the high-order 56 bits of the `rax` register, we must be consistent when operating on the data in the low-order 8 bits. So we use the `andb` instruction to operate only on the `al` portion of the `rax` register.

```
38      andb    $UPMASK, %al      # in range, convert
```

Storing the final result is the same, except we are using different registers.

```

39      movq    destPtr(%rbp), %r8 # destination pointer
40      movb    %al, (%r8)        # store character

```

Both ways of implementing this algorithm are correct, and there is probably no significant efficiency difference. However, comparing the two shows the importance of maintaining consistency in data sizes. You do not need to zero out unused portions of registers, but you should also never assume that they are zero.

## 12.2 Shifting Bits

It is sometimes useful to be able to shift all the bits to the left or right. Since the relative position of a bit in an integer has significance, shifting all the bits to the left one position effectively multiplies the value by two. And shifting them one position to the right effectively divides the value by two. As you will see in Sections 12.3 and 12.4, the multiplication and division instructions are complicated. They also take a great deal of processor time. Using left/right shifts to effect multiplication/division by powers of two is very efficient.

There are two instructions for shifting bits to the right — *shift right* and *shift arithmetic right*:

`shrs source, destination`  
`sars source, destination`

<u>s</u>	<u>meaning</u>	<u>number of bits</u>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

where s denotes the size of the operand:

`shr destination, source`  
`sar destination, source`

Intel®  
Syntax

The *source* operand can be either an immediate value, or the value can be located in the cl register. If it is an immediate value, it can be up to  $63_{10}$ . The *destination* operand can be either a memory location or a register. Any of the addressing modes that we have covered can be used to specify a memory location.

The action of the `shr` instruction is to shift all the bits in the destination operand to the right by the number of bit positions specified by the source operand. The “vacated” bit positions at the high-order end of the destination operand are filled with zeros. The last bit to be shifted out of the low-order bit position is copied into the carry flag (CF). For example, if the eax register contained the bit pattern aabb 2233, then the instruction

`shrw $1, %ax`

would produce

`eax: aabb 1119`

and the CF would be one. With the same initial conditions, the instruction

`shrl $4, %eax`

would produce

`eax: 0aab b223`

and the CF would be zero.

The action of the `sar` instruction is to shift all the bits in the destination operand to the right by the number of bit positions specified by the source operand. The “vacated” bit positions at the high-order end of the destination operand are filled with the same value that was originally in the highest-order bit. The last bit to be shifted out of the low order bit position is copied into the carry flag (CF). For example, if the eax register contained the bit pattern aabb 2233, then the instruction

`sarw $1, %ax`

would produce

`eax: aabb 1119`

and the CF would be one. With the same initial conditions, the instruction

`sarl $4, %eax`

would produce

`eax: faab b223`

and the CF would be zero.

Thus the difference between “shift right” and “shift arithmetic right” is that the arithmetic shift preserves the sign of the value – as though it represents an integer stored in two’s complement code.

There are two instructions for shifting bits to the left — *shift left* and *shift arithmetic left*:

`shls source, destination`  
`sals source, destination`

where *s* denotes the size of the operand:

<i>s</i>	meaning	number of bits
b	byte	8
w	word	16
l	longword	32
q	quadword	64

`shl destination, source`  
`sal destination, source`

The *source* operand can be either an immediate value, or the value can be located in the *cl* register. If it is an immediate value, it can be up to  $31_{10}$ . The *destination* operand can be either a memory location or a register. Any of the addressing modes that we have covered can be used to specify a memory location.

The action of both the *shl* and *sal* instructions is to shift all the bits in the destination operand to the left by the number of bit positions specified by the source operand. In fact, these are really two different assembly language mnemonics for the same machine code. The “vacated” bit positions at the low-order end of the destination operand are filled with zeros. The last bit to be shifted out of the highest-order bit position is copied into the carry flag (CF). For example, if the *eax* register contained the bit pattern *bbaa 2233*, then the instruction

`shlw $1, %ax`

would produce

`eax: bbaa 4466`

and the CF would be zero. With the same initial conditions, the instruction

`shll $4, %eax`

would produce

`eax: baa2 2330`

and the CF would be one.

We see how shifts can be used in the hexToInt function shown in Listing 12.4.

---

```
1 /*  
2  * readHex.c  
3  * Gets hex number from user and stores it as int.  
4  * Bob Plantz - 14 June 2009  
5 */  
6 #include <stdio.h>  
7 #include "writeStr.h"  
8 #include "readLn.h"  
9 #include "toLower.h"  
10 #include "hexToInt.h"  
11 #define MAX 20  
12 int main()  
13 {  
14     char theString[MAX];  
15     long int theInt;  
16  
17     writeStr("Enter up to 16 hex characters: ");  
18     readLn(theString, MAX);  
19  
20     toLower(theString, theString);  
21     theInt = hexToInt(theString);  
22     printf("%lx = %li\n", theInt, theInt);  
23     return 0;  
24 }  
  
1 /*  
2  * hexToInt.h  
3  * Converts hex character string to int.  
4  * Bob Plantz - 8 April 2008  
5 */  
6  
7 #ifndef HEXToint_H  
8 #define HEXToint_H  
9 long int hexToInt(char *);  
10 #endif  
  
1 /*  
2  * hexToInt.c  
3  * Converts hex character string to int.  
4  * Assumes A - F in upper case.  
5  * Bob Plantz - 14 June 2009  
6 */  
7  
8 #include "hexToInt.h"  
9 #define NUMERAL 0x30  
10 #define ALPHA 0x57  
11  
12 long int hexToInt(char *stringPtr)  
13 {  
14     long int accumulator = 0;
```

```

15     char current;
16
17     current = *stringPtr;
18     while (current != '\0')
19     {
20         accumulator = accumulator << 4;
21         if (current <= '9') // only works for 0-9,A-F
22             current -= NUMERAL;
23         else
24             current -= ALPHA;
25         accumulator += (long int)current;
26         stringPtr++;
27         current = *stringPtr;
28     }
29     return accumulator;
30 }
```

Listing 12.4: Shifting bits (C). (There are three files here.)

Notice that “`<<`” (on line 20 in the `hexToInt` function) is the left shift operator and “`>>`” is the right shift operator in C/C++. In C++ these operators are overloaded to provide file output and input.

The code in the main function is familiar. The compiler-generated assembly language for `hexToInt` is shown in Listing 12.5 with comments added.

```

1     .file  "hexToInt.c"
2     .text
3     .globl hexToInt
4     .type  hexToInt, @function
5 hexToInt:
6     pushq %rbp
7     movq %rsp, %rbp
8     movq %rdi, -24(%rbp)
9     movq $0, -8(%rbp)
10    movq -24(%rbp), %rax
11    movzbl (%rax), %eax
12    movb %al, -9(%rbp)
13    jmp .L2          # jump to bottom of loop
14 .L5:
15    salq $4, -8(%rbp)      # accumulator = accumulator << 4;
16    cmpb $57, -9(%rbp)      # if (current > '9')
17    jg .L3                #   jump to "else" part
18    movzbl -9(%rbp), %eax  #   load current character
19    subl $48, %eax          #   convert numeral char to int
20    movb %al, -9(%rbp)      #   and update current
21    jmp .L4                #   jump around the "else" part
22 .L3:
23    movzbl -9(%rbp), %eax  #   load current character
24    subl $87, %eax          #   convert (hex) letter char to int
25    movb %al, -9(%rbp)      #   and update current
26 .L4:
27    movsbq -9(%rbp), %rax  # type-cast char to a long int
28    addq %rax, -8(%rbp)      # add it to accumulator
29    addq $1, -24(%rbp)      # stringPtr++;
```

```

30      movq    -24(%rbp), %rax
31      movzbl (%rax), %eax
32      movb    %al, -9(%rbp)   # current = *stringPtr;
33 .L2:
34      cmpb    $0, -9(%rbp)   # while (current != '\0')
35      jne     .L5          # go to top of loop
36      movq    -8(%rbp), %rax # 64-bit return value
37      popq    %rbp
38      ret
39      .size   hexToInt, .-hexToInt
40      .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
41      .section .note.GNU-stack,"",@progbits

```

Listing 12.5: Shifting bits (gcc assembly language).

As usual, gcc has converted the while loop in `hexToInt` to a do-while loop, which is entered at the bottom. Most of this code has been covered previously. The instruction

```
15      salq    $4, -16(%rbp)   # accumulator = accumulator << 4;
```

shifts the 64 bits that make up the value of the variable `accumulator` four bits to the left. Make sure that you understand the four high-order bits in this group of 64 are lost. That is, the shift does not carry on to other memory areas beyond these 64 bits. As stated above, the last bit to get shifted out of these 64 bits is copied to the CF.

The conversions from characters to integers

```
18      movzbl -9(%rbp), %eax # load current character
19      subl    $48, %eax      # convert numeral char to int
20      movb    %al, -9(%rbp)  # and update current
```

and

```
23      movzbl -9(%rbp), %eax # load current character
24      subl    $87, %eax      # convert (hex) letter char to int
25      movb    %al, -9(%rbp)  # and update current
```

start by moving a one-byte character into an `int`-sized register with the high-order 24 bits zeroed. The actual conversion consists of subtracting off the “character part” as an integer arithmetic operation. Then the result, which is guaranteed to fit within a byte, is stored back in the single byte allocated for the original character.

Actually, we can easily see that the result of this conversion operation is a four-bit value in the range  $0000_2 - 1111_2$ . The four-bit left shift of the variable `accumulator` has left space for inserting these four bits. The bit insertion operation consists of first type casting the four-bit integer to a 64-bit integer as we load it from the variable:

```
27      movsbq -9(%rbp),%rax   # type-cast char to a long int
```

then adding this 64-bit integer to the variable `accumulator`:

```
28      addq    %rax, -8(%rbp) # add it to accumulator
```

We also note that although the standard return value is 32-bits in the `eax` register, declaring a `long int` (64-bit) return value causes the compiler to use the entire `rax` register:

```
36      movq    -8(%rbp), %rax # 64-bit return value
```

Listing 12.6 shows a version of the `hexToInt` function written in assembly language.

```

1 # hexToInt_a.s
2 # Converts hex characters to a 64-bit int.
3 # Bob Plantz - 14 June 2009
4
5 # Calling sequence:
6 #     rdi <- address of hex string to be converted
7 #     call    string2Hex
8 # returns 64-bit int represented by the hex string
9
10 # Useful constants
11     .equ    NUMERAL, 0x30
12     .equ    ALPHA, 0x57
13     .equ    HEXBITS, 4
14     .equ    TYPEMASK, 0xf
15 # Stack frame, showing local variables and arguments
16     .equ    accumulator, -16
17     .equ    current, -1
18     .equ    localSize, -32
19 # Code
20     .text
21     .globl hexToInt
22     .type hexToInt, @function
23 hexToInt:
24     pushq %rbp          # save frame pointer
25     movq %rsp, %rbp      # new frame pointer
26     addq $localSize, %rsp # local vars. and arg.
27
28     movq $0, accumulator(%rbp)    # accumulator = 0;
29 loop:
30     movb (%rdi), %al        # load character
31     cmpb $0, %al           # at end yet?
32     je done                # yes, all done
33
34     salq $HEXBITS, accumulator(%rbp) # accumulator = accumulator << 4
35
36     cmpb $'9', %al          # is it numeral?
37     ja isAlpha              # no, so it's alpha
38     subb $NUMERAL, %al      # convert to int
39     jmp addIn               # and add to accumulator
40 isAlpha:
41     subb $ALPHA, %al        # convert to int
42 addIn:
43     andq $TYPEMASK, %rax   # 4 bits -> 64 bits
44     addq %rax, accumulator(%rbp) # insert the 4 bits
45     incq %rdi               # stringPtr++;
46     jmp loop                # and check for end
47 done:
48     movq accumulator(%rbp), %rax # return accumulator;
49     movq %rbp, %rsp           # restore stack pointer
50     popq %rbp                # restore frame pointer
51     ret                      # back to caller

```

---

Listing 12.6: Shifting bits (programmer assembly language).

It differs from the C version in several ways. First, since this is a leaf function, we do not save the argument in the stack frame. Instead, we simply use the register as the `stringPtr` variable:

```
29 loop:
30     movb    (%rdi), %al      # load character
31     cmpb    $0, %al        # at end yet?
32     je      done          # yes, all done
```

We do, however, explicitly allocate stack space for the local variable:

```
23 hexToInt:
24     pushq   %rbp            # save frame pointer
25     movq    %rsp, %rbp      # new frame pointer
26     addq    $localSize, %rsp # local vars. and arg.
27
28     movq    $0, accumulator(%rbp) # accumulator = 0;
```

Although this is not required because this is a leaf function, it is somewhat better software engineering. If this function is ever modified such that it does call another function, the programmer may forget to allocate the stack space, which would then be required. There is less chance that saving the contents of the `rdi` register would be overlooked since it is where the first argument is passed. Both of these issues are arguable design decisions.

Next, for the conversion from 4-bit integer values to 64-bit, we define a *bit mask*:

```
14 .equ    TYPEMASK, 0xf
```

Performing an `and` operation with this bit mask leaves the four low-order bits as they were and sets the 60 high-order bits all to zero. Then we simply add this to the 64-bit `accumulator` (which has already been shifted four bits to the left) it effectively insert the four bits into the correct location:

```
43     andq    $TYPEMASK, %rax # 4 bits -> 64 bits
44     addq    %rax, accumulator(%rbp) # insert the 4 bits
```

## 12.3 Multiplication

The `hexToInt` function discussed in Section 12.2 shows how to convert a string of hexadecimal characters into the integer they represent. That function uses the fact that each hexadecimal character represents four bits. So as the characters are read from left to right, the bits in the `accumulator` are shifted four places to the left in order to make space for the next four-bit value. The character is converted to the four bits it represents and added to the `accumulator`.

Although the four-bit left shift seems natural for hexadecimal, it is equivalent to multiplying the value in the `accumulator` by sixteen. This follows from the positional notation used to write numbers. Add another hexadecimal digit to the right of an existing number effectively multiplies that existing number by sixteen. A little thought shows

that this algorithm, shown in Algorithm 12.1, works in any number base.

---

**Algorithm 12.1:** Character to integer conversion.
 

---

```

1 accumulator ← 0;
2 while more characters do
3   accumulator ← base × accumulator;
4   templnt ← integer equivalent of the character;
5   accumulator ← accumulator + templnt;
```

---

Of course, you probably want to write programs that allow users to work with decimal numbers. So we need to know how to convert a string of decimal characters to the integer they represent. The characters that represent decimal numbers are in the range  $30_{16} - 39_{16}$ . Table 12.1 shows the 32-bit int that corresponds to each numeric character. For a string of characters that represents a decimal integer, Algorithm 12.1 can be

Numeral (ASCII code)	int (Binary number system)									
	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0011 0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0011 0001	0000	0000	0000	0000	0000	0000	0000	0000	0000	0001
0011 0010	0000	0000	0000	0000	0000	0000	0000	0000	0000	0010
0011 0011	0000	0000	0000	0000	0000	0000	0000	0000	0000	0011
0011 0100	0000	0000	0000	0000	0000	0000	0000	0000	0000	0100
0011 0101	0000	0000	0000	0000	0000	0000	0000	0000	0000	0101
0011 0110	0000	0000	0000	0000	0000	0000	0000	0000	0000	0110
0011 0111	0000	0000	0000	0000	0000	0000	0000	0000	0000	0111
0011 1000	0000	0000	0000	0000	0000	0000	0000	0000	0000	1000
0011 1001	0000	0000	0000	0000	0000	0000	0000	0000	0000	1001

Table 12.1: Bit patterns (in binary) of the ASCII numerals and the corresponding 32-bit ints.

specialized to give Algorithm 12.2. (Recall that “.” is the bit-wise and operator.)

---

**Algorithm 12.2:** Decimal character to integer conversion.
 

---

```

1 accumulator ← 0;
2 while more characters do
3   accumulator ← 10 × accumulator;
4   templnt ← 0xf · character;
5   accumulator ← accumulator + templnt;
```

---

Shifting N bits to the left multiplies a number by  $2^N$ , so it can only be used to multiply by powers of two. Algorithm 12.2 multiplies the accumulator by 10, which cannot be accomplished with only shifts. Thus, we need to use the multiplication instruction for decimal conversions.

The multiplication instruction is somewhat more complicated than addition. The main problem is that the product can, in general, occupy the number of digits in the multiplier plus the number of digits in the multiplicand. This is easily seen by computing

$99 \times 99 = 9801$  (in decimal). Thus in general,

$$\begin{aligned}8\text{-bit} \times 8\text{-bit} &\Rightarrow 16\text{-bit} \\16\text{-bit} \times 16\text{-bit} &\Rightarrow 32\text{-bit} \\32\text{-bit} \times 32\text{-bit} &\Rightarrow 64\text{-bit} \\64\text{-bit} \times 64\text{-bit} &\Rightarrow 128\text{-bit}\end{aligned}$$

The unsigned multiplication instruction is:

`muls source`

<b>s</b>	<b>meaning</b>	<b>number of bits</b>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

where s denotes the size of the operands:

Intel® Syntax | `mul source`

In the x86-64 architecture, the destination operand contains the multiplicand and must be in the `al`, `ax`, `eax`, or `rax` register, depending on the size of the operand, for the unsigned multiplication instruction, `mul`. This register is not specified as an operand. The instruction specifies the source operand, which contains the multiplier and must be the same size. It can be located in another general-purpose register or in memory. If the numbers are eight bits (hence, one number is in `al`), the high-order portion of the result will be in the `ah` register, and the low-order portion of the result will be in the `al` register. For sixteen and thirty-two bit numbers, the low-order portion of the product will be stored in a portion of the `rax` register and the high-order will be stored in a portion of the `rdx` register as shown in Table 12.2.

<b>Operand Size</b>	<b>Portion of A Reg.</b>	<b>High-Order Result</b>	<b>Low-Order Result</b>
8 bits	<code>al</code>	<code>ah</code>	<code>al</code>
16 bits	<code>ax</code>	<code>dx</code>	<code>ax</code>
32 bits	<code>eax</code>	<code>edx</code>	<code>eax</code>
64 bits	<code>rax</code>	<code>rdx</code>	<code>rax</code>

Table 12.2: Register usage for the `mul` instruction.

For example, let's see how the computation  $7 \times 24 = 168$  looks in 8-bit, 16-bit, and 32-bit values. First, note that:

$$\begin{aligned}7_{10} &= 0000\ 0111_2 \\&= 07_{16}\end{aligned}\tag{12.1}$$

$$\begin{aligned}24_{10} &= 0001\ 1000_2 \\&= 18_{16}\end{aligned}\tag{12.2}$$

and

$$\begin{aligned}168_{10} &= 1010\ 1000_2 \\&= a8_{16}\end{aligned}\tag{12.3}$$

Now, if we declare the constants:

```
byteDay:
    .byte 24
wordDay:
    .word 24
longDay:
    .long 24
```

These declarations cause the assembler to do the following:

- `byteDay` → allocate one byte of memory and set the bit pattern of the byte to `0x18`.
- `wordDay` → allocate two bytes of memory and set the bit pattern of those two bytes to `0x0018`.
- `longDay` → allocate four bytes of memory and set the bit pattern of those four bytes to `0x00000018`.

First, consider 8-bit multiplication. If `eax` contains the bit pattern `0x?????07`, then

```
mulb byteDay
```

changes `eax` such that it contains `0x????00a8`. Notice that only the `al` portion of the `A` register can be used for the operand, but the result will occupy the entire `ax` portion of the register even though the result would fit into only the `al` portion. That is, the instruction *will* produce a 16-bit result, and anything stored in the `ah` portion will be lost.

Next, consider 16-bit multiplication. If `eax` contains `0x????0007`, then

```
mulw wordDay
```

changes `eax` to contain `0x????00a8` and `edx` to contain `0x????0000`. Two points are important in this example:

- the `ah` portion of the `A` register must be set to zero before executing the `mulw` instruction so that the `ax` portion contains the proper value, and
- the `dx` portion of the `D` register is used, even though the result is fits within the 16 bits of the `ax` register.

Finally, 32-bit multiplication. If `eax` contains `0x00000007`, then

```
mull longDay
```

changes `eax` to contain `0x000000a8` and `edx` to contain `0x00000000`. This example shows the entire `eax` register must be used for the operand before `mull` is executed, and the entire `edx` register is used for the high-order portion of the result, even though it is not needed. That is, the instruction *will* produce a 64-bit result, and anything stored in the `edx` register will be lost.

These examples show that the `rax` and `rdx` registers are used without ever explicitly appearing in the instruction. You must be very careful not to write over a required value that is stored in one of these registers. Using the multiplication instruction requires some careful planning.

There is also a signed multiply instruction, which has three forms:

```
imuls source
imuls source, destination
imuls immediate, source, destination
```

where  $s$  denotes the size of the operand:

<u><math>s</math></u>	<u>meaning</u>	<u>number of bits</u>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

Intel®  
Syntax

<code>imul source</code>
<code>imul destination, source</code>
<code>imul destination, source, immediate</code>

In the one-operand format the signed multiply instruction uses the `rdx:rax` register combination in the same way as the `mul` instruction.

In its two-operand format the destination must be a register. The source can be a register, an immediate value, or a memory location. The source and destination are multiplied, and the result is stored in the destination register. Unfortunately, if the result is too large to fit into the destination register, it is simply truncated. In this case, both the CF and OF flags are set to 1. If the result was able to fit into the destination register, both flags are set to 0.

In its three-operand format the destination must be a register. The source can be a register or a memory location. The source is multiplied by the immediate value and the result is stored in the destination register. As in the two-operand form, if the result is too large to fit into the destination register, it is simply truncated. In this case, both the CF and OF flags are set to 1. If the result was able to fit into the destination register, both flags are set to 0.

The difference between signed and unsigned multiplication can be illustrated with the following multiplication of two 16-bit values. Given the declaration:

```
.data
mOne: .word -1
```

and the initial conditions in the `rdx` and `rax` registers:

<code>rdx</code>	0x7fffec889ec8	140737161764552
<code>rax</code>	0x2ba1be79ffff	47973685395455

we will multiply the two 16-bit values in the memory location `mOne` and the register `ax`. Notice that if we consider them to be signed integers, both values represent -1, and we would expect the result to be +1 ( $= 00000001_{16}$ ). However, if we consider them to be unsigned integers, they both represent  $65535_{10}$ , and we would expect the result to be  $4294836225_{10}$  ( $= \text{ffff}0001_{16}$ ).

Indeed, starting with the initial conditions above, the instruction:

```
mulw mOne
```

yields:

<code>rdx</code>	0x7fffec88ffff	140737161789438
<code>rax</code>	0x2ba1be790001	47973685329921

We see that the register combination `dx:ax = fffe:0001`. And with the same initial conditions, the instruction

```
imulw mOne
```

yields:

<code>rdx</code>	0x7fffec880000	140737161723904
<code>rax</code>	0x2ba1be790001	47973685329921

With signed multiplication we get  $dx:ax = 0000:0001$ . Both of these operations multiplied 16-bit values to provide a 32-bit result. They each used the sixteen low-order bits of the rax and rdx registers for the result. Notice that the upper 48 bits of these registers were not changed and that neither “ax” nor “dx” appeared in either instruction.

Multiplication is used on line 19 in the decToInt function shown in Listing 12.7.

```

1  /*
2   * decToInt.c
3   * Converts decimal character string to int.
4   * Bob Plantz - 15 June 2009
5   */
6
7 #include "decToInt.h"
8 #define NUMERALMASK 0xf
9
10 unsigned int decToInt(char *stringPtr)
11 {
12     unsigned int accumulator = 0;
13     unsigned int base = 10;
14     unsigned char current;
15
16     current = *stringPtr;
17     while (current != '\0')
18     {
19         accumulator = accumulator * base;
20         current = current & NUMERALMASK;
21         accumulator += (int)current;
22         stringPtr++;
23         current = *stringPtr;
24     }
25     return accumulator;
26 }
```

Listing 12.7: Convert decimal text string to int (C).

As we can see on line 17 in Listing 12.8 the compiler has chosen to use the imull instruction for multiplication.

```

1      .file  "decToInt.c"
2      .text
3      .globl decToInt
4      .type  decToInt, @function
5 decToInt:
6      pushq %rbp
7      movq %rsp, %rbp
8      movq %rdi, -24(%rbp)
9      movl $0, -4(%rbp)
10     movl $10, -12(%rbp)
11     movq -24(%rbp), %rax
12     movzbl (%rax), %eax
13     movb %al, -5(%rbp)
14     jmp .L2
15 .L3:    movl -4(%rbp), %eax    # destination must
```

```

17    imull    -12(%rbp), %eax  #  be in a
18    movl    %eax, -4(%rbp)   #  register
19    andb    $15, -5(%rbp)
20    movzbl  -5(%rbp), %eax
21    addl    %eax, -4(%rbp)
22    addq    $1, -24(%rbp)
23    movq    -24(%rbp), %rax
24    movzbl  (%rax), %eax
25    movb    %al, -5(%rbp)

26 .L2:
27    cmpb    $0, -5(%rbp)
28    jne     .L3
29    movl    -4(%rbp), %eax
30    popq    %rbp
31    ret
32    .size   decToInt, .-decToInt
33    .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
34    .section .note.GNU-stack, "",@progbits

```

Listing 12.8: Convert decimal text string to int (gcc assembly language).

Recall that the destination must be in a register. The compiler has chosen eax in this case. The value to be multiplied must be loaded from its memory location into the register, multiplied, then stored back into memory:

```

16    movl    -4(%rbp), %eax  # destination must
17    imull    -12(%rbp), %eax  #  be in eax
18    movl    %eax, -4(%rbp)   #  register

```

It may appear that the compiler has made an error here. Since both the multiplier and the multiplicand are 32-bit values, the product can be 64 bits wide. However, the compiler has chosen code that assumes the product will be no wider than 32 bits. This can lead to arithmetic errors when multiplying large integers, but according to the C programming language standard [10] this is acceptable:

A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

In Listing 12.9 the programmer has chosen unsigned multiplication, mull.

```

1 # decToInt.s
2 # Converts decimal character string to unsigned int.
3 # Bob Plantz - 15 June 2009
4
5 # Calling sequence:
6 #     rdi <- address of decimal string to be converted
7 #     call    decToInt
8 # returns 64-bit int represented by the decimal string
9
10 # Useful constants
11    .equ    NUMERALMASK, 0xf
12    .equ    DECIMAL, 10
13 # Stack frame, showing local variables and arguments

```

```

14      .equ    accumulator, -8
15      .equ    current, -1
16      .equ    localSize, -16
17
18      .text
19      .globl  decToInt
20      .type   decToInt, @function
21 decToInt:
22      pushq  %rbp          # save base pointer
23      movq   %rbp, %rsp     # new base pointer
24      addq   $localSize, %rsp # local vars. and arg.
25
26      movl   $0, accumulator(%rbp)    # accumulator = 0;
27 loop:
28      movb   (%rdi), %sil      # load character
29      cmpb   $0, %sil        # at end yet?
30      je     done            # yes, all done
31
32      movl   $DECIMAL, %eax    # multiplier
33      mull   accumulator(%rbp) # accumulator * 10
34      movl   %eax, accumulator(%rbp) # update accumulator
35
36      andl   $NUMERALMASK, %esi  # char -> int
37      addl   %esi, accumulator(%rbp) # add the digit
38      incq   %rdi             # stringPtr++;
39      jmp    loop              # and check for end
40 done:
41      movl   accumulator(%rbp), %eax # return accumulator;
42      movq   %rbp, %rsp        # restore stack pointer
43      popq   %rbp             # restore base pointer
44      ret                 # back to caller

```

Listing 12.9: Convert decimal text string to int (programmer assembly language).

And since this is a leaf function, the register used to pass the address of the text string (`rdi`) is simply used as the pointer variable rather than allocate a register save area in the stack frame:

```

27 loop:
28      movb   (%rdi), %sil      # load character
29      cmpb   $0, %sil        # at end yet?
30      je     done            # yes, all done
31
32      movl   $DECIMAL, %eax    # multiplier
33      mull   accumulator(%rbp) # accumulator * 10
34      movl   %eax, accumulator(%rbp) # update accumulator
35
36      andl   $NUMERALMASK, %esi  # char -> int
37      addl   %esi, accumulator(%rbp) # add the digit
38      incq   %rdi             # stringPtr++;
39      jmp    loop              # and check for end

```

This is safe because no other functions are called within this loop. Of course, the programmer must be careful that the pointer variable (`rdi`) is not changed unintentionally.

The discussion of multiplication here highlights a problem with C — if an arithmetic operation produces a result that is too large to fit into the allocated data type, the most significant portion of the result is lost. It is important to realize that this loss of information can occur in intermediate results when executing even simple arithmetic expressions. It is therefore *VERY IMPORTANT* that you analyze each step of arithmetic operations with the range of values that is possible at that step in order to ensure that the result does not overflow the allocated data type. This is one more place where your knowledge of assembly language can help you to write better programs in C/C++.

## 12.4 Division

Division poses a different problem. In general, the quotient will not be larger than the dividend (except when attempting to divide by zero). Division is also complicated by the existence of a remainder. The divide instruction starts with a dividend that is twice as wide as the divisor. Both the quotient and remainder are the same width as the divisor. The unsigned division instruction is:

`divs source`

where *s* denotes the size of the operands:

<b><i>s</i></b>	<b>meaning</b>	<b>number of bits</b>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

Intel®  
Syntax

`div source`

The *source* operand specifies the divisor. It can be either a register or a memory location. Table 12.3 shows how to set up the registers with the dividend and where the quotient and remainder will be located after the unsigned division instruction, `div`, is executed. Notice that the quotient is the C '/' operation, and the remainder is the '%' operation.

<b>Operand Size</b>	<b>High-Order Dividend</b>	<b>Low-Order Dividend</b>	<b>Quotient</b>	<b>Remainder</b>
8 bits	ah	al	al	ah
16 bits	dx	ax	ax	dx
32 bits	edx	eax	eax	edx
64 bits	rdx	rax	rax	rdx

Table 12.3: Register usage for the `div` instruction.

For example, let's see how the computation  $93 \div 19 = 4$  with remainder 17 looks in 8-bit, 16-bit, and 32-bit values. First, note that:

$$\begin{aligned} 19_{10} &= 0001\ 0011_2 \\ &= 13_{16} \end{aligned} \tag{12.4}$$

and

$$\begin{aligned} 93_{10} &= 0101\ 1101_2 \\ &= 5d_{16} \end{aligned} \tag{12.5}$$

Now if we declare the constants:

```
byteDivisor:
    .byte 19
wordDivisor:
    .word 19
longDivisor:
    .long 19
```

These declarations cause the assembler to do the following:

- `byteDivisor` → allocate one byte of memory and set the bit pattern of the byte to `0x13`.
- `wordDivisor` → allocate two bytes of memory and set the bit pattern of those two bytes to `0x0013`.
- `longDivisor` → allocate four bytes of memory and set the bit pattern of those four bytes to `0x00000013`.

First, consider 8-bit division. If `eax` contains the bit pattern `0x????005d`, then

`divb byteDivisor`

changes `eax` such that it contains `0x????1104`. Notice that `ah` had to be set to 0 *before* executing `divb` even though the dividend fits into one byte. That's because the `divb` instruction starts with the `ah:al` pair as a 16-bit number. We also see that after executing the instruction, `ax` contains what appears to be a much larger number as a result of the division. Of course, we no longer consider `ax`, but `al` (the quotient) and `ah` (the remainder) as two separate numbers.

Next, consider 16-bit division. If `eax` contains `0x????005d` and `edx` `0x????0000`, then

`divw wordDivisor`

changes `eax` to contain `0x????0004` and `edx` to contain `0x????0011`. You may wonder why the `divw` instruction does not start with the 32-bit dividend in `eax`. This is for backward compatibility — Intel processors prior to the 80386 had only 16-bit registers.

Finally, 32-bit division. If `eax` contains `0x00000005d` and `edx` `0x00000000`, then

`divl longDivisor`

changes `eax` to contain `0x000000004` and `edx` to contain `0x000000011`. Again, we see that the entire `edx` register must be filled with zeros before executing the `divl` instruction, even though the dividend fits within two bytes.

One of the more common errors with division occurs when performing repeated division of a number. Since the first division places the remainder in the area occupied

by the high-order portion of the dividend, you must remember to set that area to zero before dividing again.

The signed division instruction is:

`idivs source`

where  $s$  denotes the size of the operand:

<b><i>s</i></b>	<b>meaning</b>	<b>number of bits</b>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

Intel®  
Syntax

`| idiv source`

Unlike the signed multiply instruction, signed divide only has one form, which is the same as unsigned divide. That is, the divisor is in the source operand, and the dividend is set up in the `rax` and `rdx` registers as shown in Table 12.3.

We can see the difference between signed and unsigned division by dividing a 32-bit value by a 16-bit value. Given the declaration:

```
.data
mOne: .word -1
```

we load the 32-bit dividend, +1, into the `dx:ax` register pair

```
movw $0, %dx
movw $1, %ax
```

This give the conditions

<code>rdx</code>	<code>0x7ffffb6d00000</code>	<code>140736260472832</code>
<code>rax</code>	<code>0x2ae8f4310001</code>	<code>47180017631233</code>

Now, the unsigned value in the `mOne` variable is  $0\text{fff}_{16} = 255_{10}$ . When we divide 1 by 255 we expect to get 0 with a remainder of 1. Indeed, unsigned division:

`divw mOne`

yields:

<code>rdx</code>	<code>0x7ffffb6d00001</code>	<code>140736260472833</code>
<code>rax</code>	<code>0x2ae8f4310000</code>	<code>47180017631232</code>

The quotient is in `ax` and is 0, while the remainder (in `dx`) is 1.

With the same initial conditions, when we use the signed divide instruction, we are dividing +1 by -1. Then we expect to get -1 with a remainder of 0. Indeed, signed division:

`idivw mOne`

yields:

<code>rdx</code>	<code>0x7ffffb6d00000</code>	<code>140736260472832</code>
<code>rax</code>	<code>0x2ae8f431ffff</code>	<code>47180017696767</code>

The quotient is in `ax` and is  $\text{ffff}_{16} (= -1)$ , while the remainder (in `dx`) is 0.

The “/” operation is used on line 34 in the `intToUDec` function shown in Listing 12.7, and the “%” operation is used on line 35.

```
1  /*
2   * intToUDec.c
3   *
4   * Converts an int to corresponding unsigned text
5   * string representation.
6   *
7   * input:
8   *     32-bit int
9   *     pointer to at least 10-char array
10  * output:
11  *     null-terminated string in array
12  *
13  * Bob Plantz - 15 June 2009
14  */
15
16 #include "intToUDec.h"
17 #define TOASCII 0x30
18
19 void intToUDec(char *decString, int theInt)
20 {
21     unsigned int x = theInt;
22     int base = 10;
23     char reverseArray[10];
24     char digit;
25     char *ptr = reverseArray;
26
27     *ptr = '\0';      // start with NUL
28     ptr++;
29     if (x == 0)      // zero case
30     {
31         *ptr = '0';
32         ptr++;
33     }
34     while (x > 0)    // create the string
35     {
36         digit = x % base;
37         x = x / base;
38         digit = TOASCII | digit;
39         *ptr = digit;
40         ptr++;
41     }
42     do                  // reverse the string
43     {
44         ptr--;
45         *decString = *ptr;
46         decString++;
47     } while (*ptr != '\0');
48 }
```

Listing 12.10: Convert unsigned int to decimal text string (C).

The assembly language generated by `gcc` is shown in Listing 12.11 (with comments

added).

```

1      .file  "intToUDec.c"
2      .text
3      .globl intToUDec
4      .type  intToUDec, @function
5 intToUDec:
6      pushq %rbp
7      movq %rsp, %rbp
8      movq %rdi, -40(%rbp)
9      movl %esi, -44(%rbp)
10     movl -44(%rbp), %eax
11     movl %eax, -4(%rbp)
12     movl $10, -20(%rbp)
13     leaq -32(%rbp), %rax
14     movq %rax, -16(%rbp)
15     movq -16(%rbp), %rax
16     movb $0, (%rax)
17     addq $1, -16(%rbp)
18     cmpl $0, -4(%rbp)
19     jne .L3
20     movq -16(%rbp), %rax
21     movb $48, (%rax)
22     addq $1, -16(%rbp)
23     jmp .L3
24 .L4:
25     movl -20(%rbp), %ecx
26     movl -4(%rbp), %eax
27     movl $0, %edx
28     divl %ecx
29     movl %edx, %eax
30     movb %al, -21(%rbp)
31     movl -20(%rbp), %edx
32     movl %edx, -48(%rbp)
33     movl -4(%rbp), %eax
34     movl $0, %edx
35     divl -48(%rbp)
36     movl %eax, -4(%rbp)
37     orb $48, -21(%rbp)
38     movq -16(%rbp), %rax
39     movzbl -21(%rbp), %edx
40     movb %dl, (%rax)
41     addq $1, -16(%rbp)
42 .L3:
43     cmpl $0, -4(%rbp)
44     jne .L4
45 .L5:
46     subq $1, -16(%rbp)
47     movq -16(%rbp), %rax
48     movzbl (%rax), %edx
49     movq -40(%rbp), %rax
50     movb %dl, (%rax)
51     addq $1, -40(%rbp)

```

# load base value (10)  
# load the integer  
# edx forms part of dividend  
# this is for % operation  
# get remainder  
# type cast to char and store  
# load base value (10)  
# and store it  
# load the integer  
# edx forms part of dividend  
# this is for / operation  
# x = x / base;  
# digit = TOASCII | digit;  
# load ptr for dereference  
# load digit  
# \*ptr = digit;  
# ptr++;

```

52     movq    -16(%rbp), %rax
53     movzbl (%rax), %eax
54     testb   %al, %al
55     jne     .L5
56     popq    %rbp
57     ret
58     .size   intToUDec, .-intToUDec
59     .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
60     .section .note.GNU-stack,"",@progbits

```

Listing 12.11: Convert unsigned int to decimal text string (gcc assembly language).

Compare the code sequence from lines 25 – 30:

```

25    movl    -20(%rbp), %ecx      # load base value (10)
26    movl    -4(%rbp), %eax       # load the integer
27    movl    $0, %edx            # edx forms part of dividend
28    divl    %ecx                # this is for % operation
29    movl    %edx, %eax          # get remainder
30    movb    %al, -21(%rbp)      # type cast to char and store
31    movb    %al, -9(%rbp)        # type cast to char and store

```

with that from lines 31 – 36:

```

31    movl    -20(%rbp), %edx      # load base value (10)
32    movl    %edx, -48(%rbp)      # and store it
33    movl    -4(%rbp), %eax       # load the integer
34    movl    $0, %edx            # edx forms part of dividend
35    divl    -48(%rbp)           # this is for / operation
36    movl    %eax, -4(%rbp)        # x = x / base;

```

Even though the divl instruction produces both the quotient (“/” operation) and remainder (“%” operation), the compiler uses almost the same code sequence twice, once for each operation.

In Listing 12.12 the programmer has chosen to retrieve both the quotient and the remainder from one execution of the divl instruction.

```

1 # intToUDec.s
2 # Converts unsigned int to corresponding unsigned decimal string
3 # Bob Plantz - 15 June 2009
4
5 # Calling sequence
6 #     esi <- value of the int
7 #     rdi <- address of place to store string
8 #     call intToUDec
9 # Useful constant
10    .equ    asciiNumeral,0x30
11 # Stack frame
12    .equ    reverseArray,-12
13    .equ    localSize,-16
14 # Read only data
15    .section .rodata
16 ten:   .long   10
17 # Code
18    .text
19    .globl  intToUDec

```

```

20      .type    intToUDec, @function
21 intToUDec:
22     pushq   %rbp          # save caller's base ptr
23     movq    %rsp, %rbp    # our stack frame
24     addq    $localSize, %rsp  # local char array
25
26     movl    %esi, %eax    # eax used for division
27     leaq    reverseArray(%rbp), %r8  # ptr to local array
28     movb    $0, (%r8)    # store NUL
29     incq    %r8          # increment pointer
30
31     cmpl    $0, %eax    # integer == 0?
32     jne     stringUp    # no, start on the string
33     movb    $'0', (%r8)  # yes, this is the string
34     incq    %r8          # for reverse copy
35 stringUp:
36     cmpl    $0, %eax    # integer > 0?
37     jbe     copyUp      # no, do reverse copy
38     movl    $0, %edx    # yes, high-order = 0
39     divl    ten         # divide by ten
40     orb     $asciiNumeral, %dl # convert to ascii
41     movb    %dl, (%r8)  # store character
42     incq    %r8          # increment pointer
43     jmp     stringUp    # check at top
44
45 copyUp:
46     decq    %r8          # decrement pointer
47     movb    (%r8), %dl    # get char
48     movb    %dl, (%rdi)  # store it
49     incq    %rdi         # increment storage pointer
50     cmpb    $0, %dl    # NUL character?
51     jne     copyUp      # no, keep copying
52
53     movq    %rbp, %rsp    # delete local vars.
54     popq    %rbp          # restore caller base ptr
55     ret

```

Listing 12.12: Convert unsigned int to decimal text string (programmer assembly language).

On line 38 the high-order 32 bits of the quotient (edx register) are set to 0.

```

38     movl    $0, %edx    # yes, high-order = 0
39     divl    ten         # divide by ten
40     orb     $asciiNumeral, %dl # convert to ascii
41     movb    %dl, (%r8)  # store character

```

The division on line 39 leaves “ $x / \text{base}$ ” in the eax register for the next execution of the loop body. It also places “ $x \% \text{base}$ ” in the edx register. We know that this value is in the range 0 – 9 and thus fits entirely within the dl portion of the register. Lines 40 and 41 show how the value is converted to its ASCII equivalent and stored in the local char array.

As in the decToInt function (Listing 12.9), since this is a leaf function, the register used to pass the address of the text string (rdi) is simply used as the pointer variable

rather than allocate a register save area in the stack frame. Similarly, the eax register is used as the local “x” variable.

## 12.5 Negating Signed ints

For dealing with signed numbers, the x86-64 architecture provides an instruction that will perform the two’s complement operation. That is, this instruction will negate an integer that is stored in the two’s complement notation. The mnemonic for the instruction is neg.

`negs source`

<b>s</b>	<b>meaning</b>	<b>number of bits</b>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

where s denotes the size of the operand:

Intel®  
Syntax

`neg source`

neg performs a two’s complement operation on the value in the operand, which can be either a memory location or a register. Any of the addressing modes that we have covered can be used to specify a memory location.

## 12.6 Instructions Introduced Thus Far

This summary shows the assembly language instructions introduced thus far in the book. The page number where the instruction is explained in more detail, which may be in a subsequent chapter, is also given. This book provides only an introduction to the usage of each instruction. You need to consult the manuals ([2] - [6], [14] - [18]) in order to learn all the possible uses of the instructions.

### 12.6.1 Instructions

*data movement:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
cmovecc	%reg/mem	%reg	conditional move	260
movs	\$imm/%reg	%reg/mem	move	156
movss	\$imm/%reg	%reg/mem	move, sign extend	245
movzs	\$imm/%reg	%reg/mem	move, zero extend	245
popw		%reg/mem	pop from stack	181
pushw	\$imm/%reg/mem		push onto stack	181

*s = b, w, l, q; w = l, q; cc = condition codes*

*arithmetic/logic:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
adds	\$imm/%reg	%reg/mem	add	214
adds	mem	%reg	add	214
ands	\$imm/%reg	%reg/mem	bit-wise and	290
ands	mem	%reg	bit-wise and	290
cmps	\$imm/%reg	%reg/mem	compare	237
cmps	mem	%reg	compare	237
decs	%reg/mem		decrement	249
divs	%reg/mem		unsigned divide	315
idivs	%reg/mem		signed divide	317
imuls	%reg/mem		signed multiply	310
incs	%reg/mem		increment	248
leaw	mem	%reg	load effective address	191
subs	\$imm/%reg	%reg/mem	subtract	215
muls	%reg/mem		unsigned multiply	309
negs	%reg/mem		negate	322
ors	\$imm/%reg	%reg/mem	bit-wise inclusive or	290
ors	mem	%reg	bit-wise inclusive or	290
sals	\$imm/%cl	%reg/mem	shift arithmetic left	302
sars	\$imm/%cl	%reg/mem	shift arithmetic right	301
shls	\$imm/%cl	%reg/mem	shift left	302
shrs	\$imm/%cl	%reg/mem	shift right	301
subs	mem	%reg	subtract	215
tests	\$imm/%reg	%reg/mem	test bits	238
tests	mem	%reg	test bits	238
xors	\$imm/%reg	%reg/mem	bit-wise exclusive or	290
xors	mem	%reg	bit-wise exclusive or	290

*s = b, w, l, q; w = l, q**program flow control:*

<b>opcode</b>	<b>location</b>	<b>action</b>	<b>see page:</b>
call	label	call function	173
ja	label	jump above (unsigned)	239
jae	label	jump above/equal (unsigned)	239
j b	label	jump below (unsigned)	239
j be	label	jump below/equal (unsigned)	239
je	label	jump equal	239
jg	label	jump greater than (signed)	240
j ge	label	jump greater than/equal (signed)	240
jl	label	jump less than (signed)	240
j le	label	jump less than/equal (signed)	240
jmp	label	jump	241
jne	label	jump not equal	239
jno	label	jump no overflow	239
jcc	label	jump on condition codes	239
leave		undo stack frame	192
ret		return from function	192
syscall		call kernel function	201

*cc = condition codes*

## 12.6.2 Addressing Modes

<b>register direct:</b>	The data value is located in a CPU register. <i>syntax:</i> name of the register with a "%" prefix. <i>example:</i> movl %eax, %ebx
<b>immediate data:</b>	The data value is located immediately after the instruction. Source operand only. <i>syntax:</i> data value with a "\$" prefix. <i>example:</i> movl \$0xabcd1234, %ebx
<b>base register plus offset:</b>	The data value is located in memory. The address of the memory location is the sum of a value in a base register plus an offset value. <i>syntax:</i> use the name of the register with parentheses around the name and the offset value immediately before the left parenthesis. <i>example:</i> movl \$0xaabbccdd, 12(%eax)
<b>rip-relative:</b>	The target is a memory address determined by adding an offset to the current address in the rip register. <i>syntax:</i> a programmer-defined label <i>example:</i> je somePlace

## 12.7 Exercises

**12-1** (§12.2) Write a program in assembly language that

- prompts the user to enter a number in binary,
- reads the user input into a char array, and
- converts the string of characters in the char array into a decimal integer stored in a local int variable.
- calls printf to display the int.

Your program should use the writeStr function from Exercise 11-3 to display the user prompt. And it should use the readStr function from Exercise 11-4 or 11-6 to read the user's input.

Your program should read the user's input into the local char array, then perform the conversion using the stored characters. Do not do the conversion as the characters are entered by the user.

Your program does not need to check for user errors. You can assume that the user will enter only ones and zeros. And you can assume that the user will not enter more than 32 bits. (Be careful when you test your program.)

**12-2** (§12.2) Write a program in assembly language that allows the user to enter a decimal integer then displays it in binary.

Your program should convert the decimal integer into the corresponding C-style text string of ones and zeros, then use the writeStr function from Exercise 11-3 to display the text string.

This program will require some careful planning in order to get the bits to print in the correct order.

**12-3** (§12.3) Write a function, `mul16`, in assembly language that takes two 16-bit integers as arguments and returns the 32-bit product of the argument. Write a main driver function to test `mul16`. You may use `printf` and `scanf` in the main function for the user interface.

Hint: Notice that most of the numbers in this problem are 16-bit unsigned integers. Read the man pages for `printf` and `scanf`. In particular, the "u" flag character is used to indicate a short (16-bit) int.

**12-4** (§12.4) Write a function, `div32`, in assembly language that implements the C / operation. The function takes two 32-bit integers as arguments and returns the 32-bit quotient of the first argument divided by the second. Write a main driver function to test `div32`. You may use `printf` and `scanf` in the main function for the user interface.

**12-5** (§12.4) Write a function, `mod32`, in assembly language that implements the C % operation. The function takes two 32-bit integers as arguments and returns the 32-bit quotient of the first argument divided by the second. Write a main driver function to test `mod32`. You may use `printf` and `scanf` in the main function for the user interface.

**12-6** (§12.4) Write a function in assembly language, `decimal2uint`, that takes two arguments: a pointer to a `char`, and a pointer to an `unsigned int`.

```
int decimal2uint(char *, unsigned int *);
```

The function assumes that the first argument points to a C-style text string that contains only numeric characters representing an unsigned decimal integer. It computes the binary value of the integer and stores the result at the location where the second argument points. It returns zero.

Write a program that demonstrates the correctness of `decimal2uint`. Your program will allocate a `char` array, call `readStr` (from Exercise 11-4 or 11-5) to get a decimal integer from the user, and call `decimal2uint` to convert the text string to binary format. Then it adds an integer to the user's input integer and uses `printf` to display the result.

Hint: Start with the program from Exercise 12-1. Rewrite it so that the conversion from the text string to the binary number is performed by a function. Then modify the function so that it performs a decimal conversion instead of binary.

**12-7** (§12.4) Write a function in assembly language, `uint2dec`, that takes two arguments: a pointer to a `char`, and an `unsigned int`.

```
int uint2dec(char *, unsigned int);
```

The function assumes that the first argument points to a `char` array that is large enough to hold a text string that represents the largest possible 32-bit unsigned integer in decimal. It computes the characters that represent the integer (the second argument) and stores this representation as a C-style text string where the first argument points. It returns zero.

Write an assembly language program that demonstrates the correctness of `uint2dec`. Your program will allocate one `char` array, call `readStr` (from Exercise 11-4) or 11-6 to get a decimal integer from the user, and call `decimal2uint` (from Exercise 12-6) to convert the text string to binary format. It should add a constant integer to

this converted value. Then it calls `uint2dec` to convert the sum to its corresponding text string, storing the string in the `char` array.

Hint: Start with the program from Exercise 10-6. Rewrite it so that the conversion from the binary number to the text string is performed by a function. Then modify the function so that it performs a decimal conversion instead of binary.

**12-8** (§12.3) Modify the program in Exercise 12-7 so that it deals with signed `ints`.

Hint: Write the function `decimal2sint`, which will call `decimal2uint`, and write the function `sint2dec`, which will call `uint2dec`.

# Chapter 13

## Data Structures

An essential part of programming is determining how to organize the data. Homogeneous data is often grouped in an array, and heterogeneous data in a `struct`. In this chapter, we study how both these data structures are implemented.

### 13.1 Arrays

An array in C/C++ consists of one or more elements, all of the same type, arranged contiguously in memory. To access an element in an array you need to specify two address-related items:

- the beginning of the array, and
- the number of the element to access. (Element numbering begins at zero.)

For example, given the declaration:

```
int array[50];
```

you can store an integer, say 123, in the *i*th element with the statement

```
array[i] = 123;
```

In this example the beginning of the array is specified by using the name, and the number of the element is specified by the [...] syntax, as illustrated by the program in Listing 13.1.

---

```
1 /*  
2 * arrayElement.c  
3 * Stores a value in one element of an array.  
4 * Bob Plantz - 15 June 2009  
5 */  
6  
7 #include <stdio.h>  
8  
9 int main(void)  
10 {  
11     int myArray[50];  
12     int i = 25;  
13  
14     myArray[i] = 123;
```

```

15     printf("The value is %i\n", myArray[i]);
16
17     return 0;
18 }
```

Listing 13.1: Storing a value in one element of an array (C).

We would expect this program to allocate  $4 \times 50 = 200$  bytes for `myArray`, plus 4 bytes for `i` in the local variable area. Indeed, the gcc-generated assembly language in Listing 13.2 shows that this total (204) has been increased to the next multiple of sixteen, and 208 bytes have been allocated in the stack frame.

```

1      .file    "arrayElement.c"
2      .section .rodata
3 .LC0:
4      .string "The value is %i\n"
5
6      .text
7      .globl  main
8      .type   main, @function
9 main:
10     pushq   %rbp
11     movq    %rsp, %rbp
12     subq    $224, %rsp          # myArray and i
13     movl    $25, -212(%rbp)    # i = 25;
14     movl    -212(%rbp), %eax   # load i
15     cltq                # convert to 64-bit
16     movl    $123, -208(%rbp,%rax,4) # myArray[i] = 123;
17     movl    -212(%rbp), %eax   # load i
18     cltq                # convert to 64-bit
19     movl    -208(%rbp,%rax,4), %eax # esi <- myArray[i]
20     movl    %eax, %esi
21     movl    $.LC0, %edi
22     movl    $0, %eax
23     call    printf
24     movl    $0, %eax
25     leave
26     ret
27     .size   main, .-main
28     .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
29     .section .note.GNU-stack,"",@progbits
```

Listing 13.2: Storing a value in one element of an array (gcc assembly language).

Next, we see that the number of the element that is being accessed, 25, is stored in the variable `i`. Then it is loaded into `eax` and converted from 32-bit to 64-bit.

```

12     movl    $25, -212(%rbp)    # i = 25;
13     movl    -212(%rbp), %eax   # load i
14     cltq                # convert to 64-bit
```

The next instruction

```
15     movl    $123, -208(%rbp,%rax,4) # myArray[i] = 123;
```

uses an addressing mode for the destination that is new to you, *indexed*. The syntax in the GNU assembly language is

*offset(base\_register\_name, index\_register\_name, scale\_factor)*

Intel®  
Syntax | [base\_register\_name + index\_register\_name \* scale\_factor + offset]

When it is zero, the offset is not required.

**indexed:** The data value is located in memory. The address of the memory location is the sum of the value in the base register plus the scale factor times the value in the index register, plus the offset.

*syntax:* place parentheses around the comma separated list — *base\_register, index\_register, scale* — and preface it with the *offset*.

*example:* -16(%rdx, %rax, 4)

Intel®  
Syntax | [rdx + rax\*4 - 16]

The indexed addressing mode allows us to specify

- the beginning address of the array,
- the index, and
- the number of bytes in each element

in one instruction. The number of bytes in each element can be 1, 2, 4, or 8. Both registers, the beginning address register and the index register, must be the same size. (Hence the cltq instruction on line 14 of Listing 13.2 to convert the index value from a 32 to 64 bits in the rax register.)

So from the destination operand of the instruction on line 15, we can see that

- the first byte of myArray is -208 bytes from the stack frame pointer, rbp,
- the index into the array, i, is in rax, and
- each array element is four bytes.

Thus, the address of the element in the array is given by

$$\text{effective address} = 4 \times \text{index in rax} + \text{address in rbp} - 208$$

Now that we know how a single array element is accessed, let us see how an entire array is processed.

---

```

1 /* 
2  * clearArray1.c
3  * Allocates an int array, stores zero in each element,
4  * and prints results.
5  * Bob Plantz - 16 June 2009
6 */
7 #include <stdio.h>
8
9 int main(void)
10 {
11     int intArray[10];
12     int index;
```

```

13
14     index = 0;
15     while (index < 10)
16     {
17         intArray[index] = 0;
18         index++;
19     }
20     index = 0;
21     while (index < 10)
22     {
23         printf("intArray[%i] = %i\n", index, intArray[index]);
24         index++;
25     }
26     return 0;
27 }
```

Listing 13.3: Clear an array (C).

The gcc compiler generated the assembly language shown in Listing 13.4 for this array clearing program.

```

1     .file  "clearArray1.c"
2     .section    .rodata
3 .LC0:
4     .string "intArray[%i] = %i\n"
5     .text
6     .globl  main
7     .type   main, @function
8 main:
9     pushq  %rbp
10    movq   %rsp, %rbp
11    subq   $64, %rsp
12    movl   $0, -52(%rbp)          # index = 0;
13    jmp    .L2
14 .L3:
15    movl   -52(%rbp), %eax      # load current index value
16    cltq
17    movl   $0, -48(%rbp,%rax,4)  # convert to 64 bits
18    addl   $1, -52(%rbp)          # intArray[index] = 0;
19 .L2:
20    cmpl   $9, -52(%rbp)
21    jle    .L3
22    movl   $0, -52(%rbp)          # index = 0;
23    jmp    .L4
24 .L5:
25    movl   -52(%rbp), %eax      # load current index value
26    cltq
27    movl   -48(%rbp,%rax,4), %edx # convert to 64 bits
28    movl   -52(%rbp), %eax      # load array element
29    movl   %eax, %esi
30    movl   $.LC0, %edi
31    movl   $0, %eax              # no floats
32    call   printf
33    addl   $1, -52(%rbp)          # index++;
```

```

34 .L4:
35     cmpl    $9, -52(%rbp)
36     jle     .L5
37     movl    $0, %eax
38     leave
39     ret
40     .size   main, .-main
41     .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
42     .section .note.GNU-stack,"",@progbits

```

Listing 13.4: Clear an array (gcc assembly language).

We can see from line 17

```
17     movl    $0, -48(%rbp,%rax,4)    # intArray[index] = 0;
```

that the address of the first element of this array is  $0 * 4 - 48 = -48$  from the address in rbp, and the address of the last element is  $9 * 4 - 48 = -12$  from the address in rbp. Since this function does not call any others, the array is stored in the red zone.

Indexing through the array is accomplished by loading the current value of the index variable into the rax register. Although this example simply increments the index through the array, you can see that the value used to index the array element is independent of maintaining the beginning address of the array.

The third value in the parentheses, 4, allows you to use the actual element number as the array index. This is clearly more convenient — hence, less error prone — than having to compute the number of bytes from the beginning of the array using the index value.

If we did not have this addressing mode, we would have to do something like:

```
# clear the array
.L3:
    movl    -4(%rbp), %eax
    cltq
    salq    $2, %rax          # multiply index by 4
    leaq    -48(%rbp), %rsi  # address of array start
    addq    %rax, %rsi        # address of current element
    movl    $0, (%rsi)        # store zero there
    addl    $1, -4(%rbp)      # index ++
.L2:
    cmpl    $9, -4(%rbp)
    jle     .L3
```

Although this is logically correct, it requires two more instructions and uses more registers.

RISC architectures (e.g., PowerPC, MIPS, Itanium) typically do not have the indexed addressing mode, hence this algorithm must be used.

Listing 13.5 shows the equivalent program written in assembly language.

```

1 # clearArray2.s
2 # Allocates an int array, stores zero in each element,
3 # and prints results.
4 # Bob Plantz - 16 June 2009
5
6 # Stack frame
```

```

7      .equ    intArray,-40      # space for 10 ints in the array
8      .equ    rbxSave,-48       # preserve registers
9      .equ    r12Save,-56
10     .equ    localSize,-64

11 # Constant data
12     .section .rodata
13     .string "intArray[%i] = %i\n"
14 format: .string "intArray[%i] = %i\n"

15 # The program
16     .text
17     .globl main
18     .type main, @function

main:
19     pushq %rbp
20     movq %rsp, %rbp
21     addq $localSize, %rsp
22     movq %rbx, rbxSave(%rbp)
23     movq %r12, r12Save(%rbp)

24     # save caller base pointer
25     # set our base pointer
26     # local variables
27     # save regs.

# clear the array
28     movq $0, %rax          # index = 0
29     leaq intArray(%rbp), %rbx # beginning of array
30 clearLup:
31     movl $0, (%rbx,%rax,4) # store zero
32     incq %rax              # index++
33     cmpq $9, %rax           # all filled?
34     jle clearLup            # do rest of elements

# print the array
35
36     movq $0, %r12          # index = 0
37     leaq intArray(%rbp), %rbx # beginning of array
38 printLup:
39     movl (%rbx,%r12,4), %edx # load element value
40     movl %r12d, %esi         # get index value
41     movq $format, %rdi        # format string
42     movl $0, %eax             # no floats
43     call printf
44     incq %r12                # index++
45     cmpq $9, %r12             # all filled?
46     jle printLup              # do rest of elements

47     movq rbxSave(%rbp), %rbx # restore regs.
48     movq r12Save(%rbp), %r12
49     movl $0, %eax             # return 0;
50     movq %rbp, %rsp           # remove local vars
51     popq %rbp                 # restore caller base ptr
52                                         # back to OS
53
54     ret

```

Listing 13.5: Clear an array (programmer assembly language).

This version uses a do-while loop instead of a while loop entered at the bottom. The index and the address of the beginning of the array are maintained in registers.

Using a register for the index value presents a potential problem. Recall that some registers are guaranteed to be preserved by a function (Table 6.4, page 134). We have used r12 for the print do-while loop in this program because it calls another function — `printf`. The current value of `index` must be copied to the correct argument register for the call to `printf`:

```
41      movl    %r12d, %esi           # get index value
```

Although the operating system probably does not depend on registers being saved, we have done so in this program:

```
24      movq    %rbx, rbxSave(%rbp)  # save regs.  
25      movq    %r12, r12Save(%rbp)
```

and:

```
49      movq    rbxSave(%rbp), %rbx  # restore regs.  
50      movq    r12Save(%rbp), %r12
```

just to be safe.

## 13.2 structs (Records)

An array is useful for grouping homogeneous data items that are of the same data type. A record (struct in C/C++) is used for grouping heterogeneous data items, which may be of the same or different data types. For example, an array is probably better for storing a list of test scores in a program that works with the  $i^{th}$  test score, but a struct might be better for storing the coordinates of a point on an  $x - y$  graph.

The data elements in a struct are usually called *fields*. Accessing a field in a struct also requires two address-related items:

- the name of the struct, and
- the name of the field.

Consider the C program in Listing 13.6

---

```
1 /*  
2  * structField1.c  
3  * Allocates two structs and assigns a value to each field  
4  * in each struct.  
5  * Bob Plantz - 16 June 2009  
6 */  
7  
8 #include <stdio.h>  
9  
10 struct theTag  
11 {  
12     char aByte;  
13     int anInt;  
14     char anotherByte;  
15 };  
16  
17 int main(void)  
18 {  
19     struct theTag x;
```

```

20 struct theTag y;
21
22     x.aByte = 'a';
23     x.anInt = 123;
24     x.anotherByte = 'b';
25     y.aByte = '1';
26     y.anInt = 456;
27     y.anotherByte = '2';
28
29     printf("x: %c, %i, %c and y: %c, %i, %c\n",
30             x.aByte, x.anInt, x.anotherByte,
31             y.aByte, y.anInt, y.anotherByte);
32     return 0;
33 }
```

Listing 13.6: Two struct variables (C).

Assignment to each of the three fields in the “x” struct is:

```

21 x.aByte = 'a';
22 x.anInt = 123;
23 x.anotherByte = 'b';
```

The name of the struct variable is specified first, followed by a dot (.), followed by the field name. The field names and their individual data types are declared between the {} pair of the struct declaration.

The amount of memory required by a struct variable is equal to the sum of the amount of memory required by each of its fields. Thus in the above program, the amount of memory required is:

aByte:	1 byte
anInt:	4 bytes
anotherByte:	1 byte
<i>total</i> =	6 bytes

If we were to allocate these six bytes of memory without some thought, the first char variable could occupy the first byte, the int variable the next four bytes, and the second char variable the following byte. That is, relative to the address of the beginning of the struct,

- aByte would be stored in byte number 0,
- anInt would be stored in bytes number 1 - 4, and
- anotherByte would be stored in byte number 5.

However, the ABI [25] specifies that the alignment of each element should be the same as that of the “most strictly aligned component.” In this example the int element should be aligned on a 4-byte boundary. So even though the char elements only require one byte, they should also be aligned on 4-byte boundaries. Also, as explained in Section 8.5 (page 197), we should allocate memory in multiples of sixteen for local variables (see Exercise 13-4). These requirements suggest that each struct variable will be allocated on the stack as shown in Figure 13.1. Thus we see that each of the struct variables in Listing 13.6 requires that we allocate sixteen bytes in the stack frame.

The next issue is access each of the fields in these two structs. You learned in Section 9.1 (page 208) that assignment in C is implemented with the mov instruction. So in this program assignment at the assembly language level is implemented:

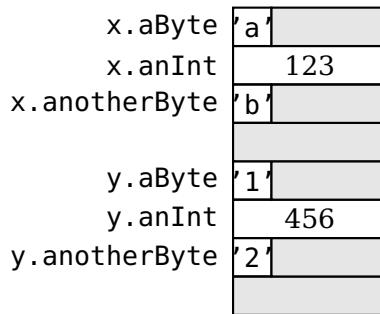


Figure 13.1: Memory allocation for the variables `x` and `y` from the C program in Listing 13.6. Shaded areas are padding bytes used to properly align the address of each variable; no data is stored in them.

```

movb    $'a', address_of_aByte_field_in_x
movl    $123, address_of_anInt_field_in_x
movb    $'b', address_of_anotherByte_field_in_x

```

The *base register plus offset* addressing mode (page 188) provides a convenient way to access each field in a struct. Simply load the address of the struct variable into a register, then use the offset of the field. We can see how the compiler has implemented this in Listing 13.7.

```

1      .file  "structField1.c"
2      .section .rodata
3      .align 8
4 .LC0:
5      .string "x: %c, %i, %c and y: %c, %i, %c\n"
6      .text
7      .globl main
8      .type main, @function
9 main:
10     pushq %rbp
11     movq %rsp, %rbp
12     subq $48, %rsp
13     movb $97, -32(%rbp)      # x.aByte = 'a';
14     movl $123, -28(%rbp)     # x.anInt = 123;
15     movb $98, -24(%rbp)      # x.anotherByte = 'b';
16     movb $49, -16(%rbp)      # y.aByte = '1';
17     movl $456, -12(%rbp)     # y.anInt = 456;
18     movb $50, -8(%rbp)       # y.anotherByte = '2';
19     movzbl -8(%rbp), %eax   # load y.anotherByte
20     movsbl %al, %esi
21     movl -12(%rbp), %r8d     # load y.anInt
22     movzbl -16(%rbp), %eax   # load y.aByte
23     movsbl %al, %edi
24     movzbl -24(%rbp), %eax   # load x.anotherByte
25     movsbl %al, %ecx
26     movl -28(%rbp), %edx     # load x.anInt
27     movzbl -32(%rbp), %eax   # load x.aByte
28     movsbl %al, %eax
29     movl %esi, (%rsp)

```

```

30      movl    %r8d, %r9d
31      movl    %edi, %r8d
32      movl    %eax, %esi
33      movl    $.LC0, %edi
34      movl    $0, %eax
35      call    printf
36      movl    $0, %eax
37      leave
38      ret
39      .size   main, .-main
40      .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
41      .section .note.GNU-stack,"",@progbits

```

Listing 13.7: Two struct variables (gcc assembly language).

The compiler allocated 48 bytes in the stack frame. Thirty-two are for the two struct variables. The additional sixteen are needed for passing the seventh argument to the `printf` function (line 29), while maintaining 16-byte addressing of the stack pointer. Rather than load the address of each struct into a register, the compiler has computed the total offset to each of the fields in each of the structs.

As usual, we equate symbolic names to these numbers when writing in assembly language. We have tried to make our assembly language version (Listing 13.8) a little more readable than the version generated by gcc.

```

1 # structField2.s
2 # Allocates two structs and assigns a value to each field
3 # in each struct.
4 # Bob Plantz - 18 June 2009
5
6 # struct field offsets from start of struct
7     .equ    aByte,0
8     .equ    anInt,4
9     .equ    anotherByte,8
10    .equ    structSize,16
11 # Stack frame
12    .equ    y,x-structSize
13    .equ    x,-structSize
14    .equ    localSize,y-8 # include space for 7th arg
15 # Read only data
16    .section .rodata
17 formatString:
18     .string "x: %c, %i, %c and y: %c, %i, %c\n"
19 # Code
20     .text
21     .globl  main
22     .type   main, @function
23 main:
24     pushq   %rbp          # save frame pointer
25     movq    %rsp, %rbp      # our frame pointer
26     addq    $localSize, %rsp # local variables
27     andq    $-16, %rsp       # align stack pointer
28
29 # fill the x struct
30     leaq    x(%rbp), %rax    # the x struct

```

```

31     movb    $'a', aByte(%rax) # x.aByte = 'a'
32     movl    $123, anInt(%rax) # x.anInt = 123
33     movb    $'b', anotherByte(%rax) # x.anotherByte = 'b'
34
35 # fill the y struct
36     leaq    y(%rbp), %rax      # the y struct
37     movb    $'1', aByte(%rax) # y.aByte = '1'
38     movl    $456, anInt(%rax) # y.anInt = 456
39     movb    $'2', anotherByte(%rax) # y.anotherByte = '2'
40
41 # print values
42     movq    $formatString, %rdi
43     leaq    x(%rbp), %rax      # the x struct
44     movb    aByte(%rax), %sil
45     movl    anInt(%rax), %edx
46     movb    anotherByte(%rax), %cl
47     leaq    y(%rbp), %rax      # the y struct
48     movb    aByte(%rax), %r8b
49     movl    anInt(%rax), %r9d
50     movb    anotherByte(%rax), %al
51     movb    %al, (%rsp)        # pass on stack
52     movl    $0, %eax          # no floating point
53     call    printf
54
55     movl    $0, %eax          # return 0;
56     movq    %rbp, %rsp         # remove local vars
57     popq    %rbp              # restore caller's frame ptr
58     ret                  # back to OS

```

Listing 13.8: Two struct variables (programmer assembly language).

The version written in assembly language loads the address of a `struct` variable into a register before accessing the fields. This allows the use of the symbolic names for each field:

```

30     leaq    x(%rbp), %rax      # the x struct
31     movb    $'a', aByte(%rax) # x.aByte = 'a'
32     movl    $123, anInt(%rax) # x.anInt = 123
33     movb    $'b', anotherByte(%rax) # x.anotherByte = 'b'

```

and:

```

36     leaq    y(%rbp), %rax      # the y struct
37     movb    $'1', aByte(%rax) # y.aByte = '1'
38     movl    $456, anInt(%rax) # y.anInt = 456
39     movb    $'2', anotherByte(%rax) # y.anotherByte = '2'

```

This technique would be necessary with, say, an array of `structs` or a function that takes an address of a `struct` as an argument.

## 13.3 structs as Function Arguments

The general rules for passing arguments to functions are:

- An input is passed by value.

- An output is passed by reference.

While C++ supports pass by reference for output parameters, C does not. In C, a pass by reference is simulated by passing a pointer to the variable to the function. Then the function can change the variable, thus affecting an output. At the assembly language level, pass by reference is implemented in C++ by passing a pointer, so these two rules can be restated:

- An input is a copy of the original value.
- An output provides the address of the original value.

Some languages, e.g., ADA, also support passing an “update.” In this case the function replaces the original value with a new value that depends upon the original value. Passing an argument for update is also implemented by passing its address.

There is an *important exception* to the rule of passing a copy for inputs. When the amount of data is large, making a copy of it is inefficient. So we organize it into a single entity and pass the address of that entity.

The most common example of this is an array. In fact, it is so common that in C arrays are automatically passed by address. Thus, in C/C++

```
void f(int a, int b[]);

-----
int x;
int y[100];
-----
f(x, y);
-----
```

means that `x` is passed by value and `y` is passed by address.

Another common example of passing a possibly large amount of data as input to a function is a struct. Of course, not all structs are large. And it is possible to pass the value in a single struct field, but the main reason for organizing data into a struct format is usually to treat several pieces of data as a more or less single unit.

Since structs are not automatically passed by address in C, we must use the address-of operator (`&`) on the name of the struct variable if we wish to avoid making a copy of the entire variable on the stack. The technique is exactly the same as passing an address of a simple variable.

Let us rewrite the C program from Listing 13.6 such that the `main` function calls another function to fill the two structs. In this example, the structs must be passed by address because the function, `loadStruct`, outputs values to them. This new version is shown in Listing 13.9.

---

```
1 /* 
2  * structPass1.c
3  * Demonstrates passing structs as arguments in c
4  * Bob Plantz - 16 June 2009
5 */
6
7 #include <stdio.h>
8 #include "loadStruct1.h" // includes struct theTag def.
9
10 int main(void)
11 {
```

```
12 struct theTag x;
13 struct theTag y;
14
15 loadStruct(&x, 'a', 123, 'b');
16 loadStruct(&y, '1', 456, '2');
17
18 printf("x: %c, %i, %c and y: %c, %i, %c\n",
19     x.aByte, x.anInt, x.anotherByte,
20     y.aByte, y.anInt, y.anotherByte);
21
22 return 0;
23 }
```

---

```
1 /*
2 * structPass1.h
3 * Assigns values to the fields of a struct.
4 *
5 * precondition
6 *     aStruct is the address of a theTag struct
7 * postcondition
8 *     firstChar is stored in the aByte field of aStruct
9 *     aNumber is stored in the anInt field of aStruct
10 *     secondChar is stored in the anotherByte field of aStruct
11 * Bob Plantz - 16 June 2009
12 */
13
14 #ifndef LOADSTRUCT_H
15 #define LOADSTRUCT_H
16
17 struct theTag {
18     char aByte;
19     int anInt;
20     char anotherByte;
21 };
22
23 void loadStruct(struct theTag* aStruct, char firstChar,
24                 int aNumber, char secondChar);
25 #endif
```

---

```
1 /*
2 * loadStruct1.c
3 * Assigns values to the fields of a struct.
4 *
5 * precondition
6 *     aStruct is the address of a theTag struct
7 * postcondition
8 *     firstChar is stored in the aByte field of aStruct
9 *     aNumber is stored in the anInt field of aStruct
10 *     secondChar is stored in the anotherByte field of aStruct
11 * Bob Plantz - 16 June 2009
12 */
13
```

```

14 #include "loadStruct1.h" // includes struct theTag def.
15
16 void loadStruct(struct theTag* aStruct, char firstChar,
17                 int aNumber, char secondChar)
18 {
19     aStruct->aByte = firstChar;
20     aStruct->anInt = aNumber;
21     aStruct->anotherByte = secondChar;
22 }
```

Listing 13.9: Passing struct variables (C). (There are three files here.)

In Listing 13.10 we examine the compiler-generated assembly language for the `loadStruct` function.

```

1      .file  "loadStruct1.c"
2
3      .text
4      .globl loadStruct
5      .type  loadStruct, @function
loadStruct:
6          pushq %rbp
7          movq %rsp, %rbp
8          movq %rdi, -8(%rbp)    # save address of struct
9          movl %edx, -16(%rbp)   # save aNumber
10         movl %ecx, %eax       # move secondChar
11         movb %sil, -12(%rbp)   # save firstChar
12         movb %al, -20(%rbp)    # save secondChar
13         movq -8(%rbp), %rax   # load struct addresss
14         movzbl -12(%rbp), %edx # load firstChar
15         movb %dl, (%rax)       # aStruct->aByte = firstChar;
16         movq -8(%rbp), %rax   # load struct addresss
17         movl -16(%rbp), %edx   # load aNumber
18         movl %edx, 4(%rax)      # aStruct->anInt = aNumber;
19         movq -8(%rbp), %rax   # load struct addresss
20         movzbl -20(%rbp), %edx # load secondChar
21         movb %dl, 8(%rax)      # aStruct->anotherByte = secondChar;
22         popq %rbp
23         ret
24         .size  loadStruct, .-loadStruct
25         .ident "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
26         .section .note.GNU-stack,"",@progbits
```

Listing 13.10: Passing struct variables (gcc assembly language). Only the `loadStruct` function is shown.

The type declaration in the function signature, `struct theTag* aStruct`, together with the `struct` definition in the header file, `loadStruct1.h`, tell the compiler what offsets to use for the `struct` fields on lines 15, 18, and 21.

We have already covered all the assembly language instructions and addressing modes needed to express the program in Listing 13.9 in assembly language. However, the `.include` assembler directive will make things much easier. The syntax is

```
.include "filename"
```

which causes the assembler to insert everything in the file named “filename” into the source at the point of the `.include` directive. This assembler directive is essentially

the same as the `#include` directive in C/C++. The assembly language version of our `structPass` program is shown in Listing 13.11.

Pay particular attention to the header file, `loadStruct.h`, which defines the offset to each field within the struct and provides overall size of the struct. This header file must be `.included` in any file that uses the struct.

Note that specifying the overall size of the struct makes it easier to allocate space for it. For example, we use

```
8     .equ    y,x-structSize  # Space for y struct
9     .equ    x,-structSize   # Space for x struct
```

in Listing 13.11 to compute the offsets to the `x` and `y` variables in the stack frame.

---

```
1
2 # loadStruct2.h
3 # The struct definition
4 # Bob Plantz - 16 June 2009
5
6 # struct definition
7     .equ    aByte,0
8     .equ    anInt,4
9     .equ    anotherByte,8
10    .equ    structSize,16

1 # structPass2.s
2 # Demonstrates passing structs as arguments in assembly language
3 # Bob Plantz - 16 June 2009
4
5     .include "loadStruct2.h"
6
7 # Stack frame
8     .equ    y,x-structSize  # Space for y struct
9     .equ    x,-structSize   # Space for x struct
10    .equ    round16,-16      # 0xfffffffffffff0
11    .equ    passArgs,-16      # Space for passing args
12 # Read only data
13     .data
14 formatString:
15     .string "x: %c, %i, %c and y: %c, %i, %c\n"
16 # Code
17     .text
18     .globl main
19     .type main, @function
20 main:
21     pushq %rbp          # save frame pointer
22     movq %rsp, %rbp       # our frame pointer
23     addq $y, %rsp         # local variables
24     andq $round16, %rsp    # round down to 16-byte boundary
25     addq $passArgs, %rsp    # for passing 7th arg
26
27     leaq x(%rbp), %rdi    # get address of x struct
28     movb $'a', %sil        # 1st char
29     movl $123, %edx        # the int
30     movb $'b', %cl         # 2nd char
```

```

31      call    loadStruct
32
33      leaq    y(%rbp), %rdi      # get address of y struct
34      movb    $'1', %sil        # 1st char
35      movl    $456, %edx        # the int
36      movb    $'2', %cl         # 2nd char
37      call    loadStruct
38
39 # print values
40      movq    $formatString, %rdi
41      leaq    x(%rbp), %rax      # the x struct
42      movb    aByte(%rax), %sil
43      movl    anInt(%rax), %edx
44      movb    anotherByte(%rax), %cl
45      leaq    y(%rbp), %rax      # the y struct
46      movb    aByte(%rax), %r8b
47      movl    anInt(%rax), %r9d
48      movb    anotherByte(%rax), %al
49      movb    %al, (%rsp)        # pass on stack
50      movl    $0, %eax          # no floating point
51      call    printf
52
53      movl    $0, %eax          # return 0;
54      movq    %rbp, %rsp        # delete local vars.
55      popq    %rbp             # restore frame pointer for OS
56      ret                  # back to caller (OS)

```

---

```

1 # loadStruct2.s
2 # Stores values in struct fields
3 # Calling sequence:
4 #     rdi <- address of the struct
5 #     sil <- first character to be stored
6 #     edx <- integer to be stored
7 #     cl <- second character to be stored
8 #     call loadStruct
9 # Bob Plantz - 16 June 2009
10
11     .include "loadStruct2.h"
12
13     .text
14     .globl loadStruct
15     .type loadStruct, @function
16 loadStruct:
17     pushq  %rbp            # save caller's frame pointer
18     movq    %rsp, %rbp       # our frame pointer
19
20     movb    %sil, aByte(%rdi)      # 1st character
21     movl    %edx, anInt(%rdi)      # the int
22     movb    %cl, anotherByte(%rdi) # 2nd character
23
24     movq    %rbp, %rsp        # delete local vars.
25     popq    %rbp             # restore frame pointer

```

```
ret          # back to caller
```

Listing 13.11: Passing struct variables (programmer assembly language). (There are three files here.)

The number in main,

```
.equ round16, -16      # 0xffffffffffff0000
```

and its use,

```
andq    $round16, %rsp # round down to 16-byte boundary
```

deserve some discussion here. After allocating space for the two structs on the stack, there is no way to know if the stack pointer is aligned on a 16-byte boundary. If it is not, the lowest-order four bits will be non-zero. The `andq` instruction sets these bits to zero, thus rounding the address down to the next lower 16-byte address boundary. Notice that this works because the stack grows toward numerically lower addresses.

The prologue and epilogue in `loadStruct` are not really needed in this simple function. But it is good to get in the habit of coding them into all your functions. It certainly has a negligible effect on execution time, and they help establish a structure to the function if it is ever changed.

## 13.4 Structs as C++ Objects

In C++ the data that defines an instance of an *object* is organized as a **struct**. The name of the object is essentially the name of a **struct** variable. The class member functions have direct access to the **struct**'s fields, even if these fields are private data members. This direct access is implemented by passing the address of the object (the **struct** variable) as an implicit argument to the member function. In our environment, it is passed as the first (the left-most) argument, but it does not appear in the argument list.

Let's look at a simple fraction class (Listing 13.12) as an example. The programs in this section assume the existence of the functions:

- `writeStr` — displays a text string on the screen
  - `getUInt` — reads an unsigned integer from the keyboard and returns it
  - `putUInt` — displays an unsigned integer on the screen

```
1 /*  
2 * incFraction.cc  
3 * Gets a fraction from user and increments by one  
4 * Bob Plantz - 18 June 2009  
5 */  
6  
7 #include "fraction.h"  
8 #include "writeStr.h"  
9  
10 int main(void)  
11 {  
12     // char array is used because writeStr takes  
13     //      a pointer to a C-style string.  
14     char newline[] = "\n";
```

```
15     fraction x;
16
17     x.get();
18     x.add(1);
19     x.display();
20     writeStr(newline);
21     return 0;
22 }
```

---

```
1 /*
2  * fraction.h
3  * simple fraction class
4  * Bob Plantz - 18 June 2009
5  */
6
7 #ifndef FRACTION_H
8 #define FRACTION_H
9
10 class fraction
11 {
12     public:
13         fraction();      // default constructor
14         void get();      // gets user's values
15         void display(); // displays fraction
16         void add(int);  // adds integer
17     private:
18         int num;        // numerator
19         int den;        // denominator
20 };
21
22 #endif
```

---

```
1 /*
2  * fraction.cc
3  * simple fraction class
4  * Bob Plantz - 18 June 2009
5  */
6
7 #include "writeStr.h"
8 #include "getUint.h"
9 #include "putUint.h"
10 #include "fraction.h"
11
12 fraction::fraction()
13 {
14     num = 0;
15     den = 1;
16 }
17
18 void fraction::get()
19 {
20     // char arrays are used because writeStr takes
```

```

21 //      a pointer to a C-style string.
22 char numMsg[] = "Enter numerator: ";
23 char denMsg[] = "Enter denominator: ";
24
25 writeStr(numMsg);
26 num = getUInt();
27
28 writeStr(denMsg);
29 den = getUInt();
30 }
31
32 void fraction::display()
33 {
34     // char array is used because writeStr takes
35     //      a pointer to a C-style string.
36     char over[] = "/";
37
38     putUInt(num);
39     writeStr(over);
40     putUInt(den);
41 }
42
43 void fraction::add(int theValue)
44 {
45     num += theValue * den;
46 }
```

Listing 13.12: Add 1 to user's fraction (C++). The C functions `getUInt`, `putUInt`, and `writeStr` are not shown here. (There are three files here.)

Let us consider the `main` function and see how arguments are passed on the stack. Recall that declaring an object in C++

```
fraction x;
```

calls the constructor function. As we said above, the address of the object (actually a `struct`) is passed to the constructor function, even though it is not explicitly stated in the object declaration statement. When program flow is passed to the constructor, the address of the `x` object is placed in the `rdi` register. The same thing occurs when the other member functions are called. The `add` member function takes an explicit argument, which is actually the second argument to the function, so is passed in the `rsi` register.

Before showing how the program of Listing 13.12 could be implemented in assembly language, we look at a C implementation, since we already know a lot about the transition from C to assembly language.

In C, the member data would be explicitly implemented as a `struct`. Implementing the member functions in C may seem very straightforward, but there is an important issue to consider — how do the member functions gain access to the data members? Since the data members are organized as a `struct`, passing its address as an argument to the member functions will allow each of them access to the data members. This is effectively what C++ does. The address of the “object” (actually, a `struct`) is passed as an implicit argument to each member function.

Another issue arises if you think about the possible names of member functions. Different C++ classes can have the same member function names, but functions in C do not belong to any class, so each must have a unique name. (Actually, “free” functions in

C++ must also have unique names.) The C++ compiler takes care of this by adding the class name to the member function name. This is called *name mangling*. (There is no standard for how this is actually done, so each compiler may do it differently.) We do our own “name mangling” for the C version of the program as shown in Listing 13.13.

```
1 /*  
2  * createFraction.c  
3  * creates a fraction and gets user's values  
4  * Bob Plantz - 16 June 2009  
5 */  
6  
7 #include "fraction.h"  
8 #include "fractionGet.h"  
9 #include "fractionAdd.h"  
10 #include "fractionDisplay.h"  
11 #include "writeStr.h"  
12  
13 int main(void)  
14 {  
15     struct fraction x;  
16  
17     fraction(&x);          // "constructor"  
18     fractionGet(&x);  
19     fractionAdd(&x, 1);  
20     fractionDisplay(&x);  
21     writeStr("\n");  
22     return 0;  
23 }
```

```
1 /*  
2  * fraction.h  
3  * A fraction "constructor" in C  
4  * Bob Plantz - 16 June 2009  
5 */  
6  
7 #ifndef FRACTION_H  
8 #define FRACTION_H  
9  
10 struct fraction  
11 {  
12     int num;  
13     int den;  
14 };  
15  
16 void fraction(struct fraction* this);  
17  
18 #endif
```

```
1 /*  
2  * fraction.c  
3  * A fraction "constructor" in C  
4  * Bob Plantz - 16 June 2009  
5 */
```

```
6
7 #include "fraction.h"
8
9 void fraction(struct fraction* this)
10 {
11     this->num = 0;
12     this->den = 1;
13 }



---


1 /*
2 * fractionGet.h
3 * Gets numerator and denominator from user.
4 * Bob Plantz - 16 June 2009
5 */
6
7 #ifndef FRACTION_ADD_H
8 #define FRACTION_ADD_H
9
10 #include "fraction.h"
11
12 void fractionAdd(struct fraction* this, int theValue);
13
14 #endif



---


1 /*
2 * fractionGet.c
3 * Gets user values for a fraction
4 * Bob Plantz - 16 June 2009
5 */
6
7 #include "writeStr.h"
8 #include "getUInt.h"
9 #include "fractionGet.h"
10
11 void fractionGet(struct fraction* this)
12 {
13     writeStr("Enter numerator: ");
14     this->num = getUInt();
15
16     writeStr("Enter denominator: ");
17     this->den = getUInt();
18 }



---


1 /*
2 * fractionAdd.h
3 * adds an integer to the fraction
4 * Bob Plantz - 16 June 2009
5 */
6
7 #ifndef FRACTION_ADD_H
8 #define FRACTION_ADD_H
9 #include "fraction.h"
```

```
10 void fractionAdd(struct fraction* this, int theValue);
11
12 #endif


---


1 /* 
2  * fractionAdd.c
3  * adds an integer to the fraction
4  * Bob Plantz - 16 June 2009
5 */
6
7
8 #include "fractionAdd.h"
9
10 void fractionAdd(struct fraction* this, int theValue)
11 {
12     this->num += theValue * this->den;
13 }


---


1 /* 
2  * fractionDisplay.h
3  * Displays a fraction in num/den format
4  * Bob Plantz - 16 June 2009
5 */
6
7 #ifndef FRACTION_DISPLAY_H
8 #define FRACTION_DISPLAY_H
9
10 #include "fraction.h"
11
12 void fractionDisplay(struct fraction* this);
13
14 #endif


---


1 /*
2  * fractionDisplay.c
3  * Displays a fraction in num/den format
4  * Bob Plantz - 16 June 2009
5  * precondition
6  *      this points to fraction, both num and den within 0 - 9
7  * postcondition
8  *      num/den displayed on the screen
9 */
10
11 #include "writeStr.h"
12 #include "putUInt.h"
13 #include "fractionDisplay.h"
14
15 void fractionDisplay(struct fraction* this)
16 {
17     putUInt(this->num);
18     writeStr("/");

```

```

19     putUInt(this->den);
20 }

```

Listing 13.13: Add 1 to user's fraction (C). (There are nine files here.)

Notice the use of the `this` pointer in the C equivalents of the “member” functions. Its place in the parameter list coincides with the “implicit” argument to C++ member functions — that is, the address of the object. The `this` pointer is implicitly available for use within C++ member functions. Its use depends upon the specific algorithm. Listing 13.13 should give you a good idea of how C++ implements objects.

From the C version in Listing 13.13 it is straightforward to move to the assembly language version in Listing 13.14.

```

1 # incFraction.s
2 # adds one to a fraction
3 # Bob Plantz - 18 June 2009
4
5 # Include object data definition
6     .include "fraction.h"
7 # Stack frame
8     .equ    x,-fractionSize # Space for a fraction object
9     .equ    localSize,x
10 # Read only data
11     .section .rodata
12 endl: .string "\n"
13 # Code
14     .text
15     .globl main
16     .type main, @function
17 main:
18     pushq %rbp          # save frame pointer
19     movq %rsp, %rbp      # our frame pointer
20     addq $localSize, %rsp # local variables
21     andq $-16, %rsp      # ensure 16-byte boundary
22
23     leaq x(%rbp), %rdi   # get address of object
24     call fraction         # construct it
25
26     leaq x(%rbp), %rdi   # get address of object
27     call fractionGet      # get user's values
28
29     movl $1, %esi          # increment fraction by 1
30     leaq x(%rbp), %rdi   # get address of object
31     call fractionAdd      # add the value
32
33     leaq x(%rbp), %rdi   # get address of object
34     call fractionDisplay   # display result
35
36     movq $endl, %rdi        # do next line
37     call writeStr
38
39     movl $0, %eax          # return 0;
40     movq %rbp, %rsp        # delete local vars.
41     popq %rbp              # restore frame pointer

```

```
42         ret                      # back to caller (OS)



---



```
1 # fraction.h
2 # simple fraction class
3 # Bob Plantz - 18 June 2009
4
5 # struct definition
6     .equ    num,0             # numerator
7     .equ    den,4             # denominator
8     .equ    fractionSize,8   # total size needed for struct



---



```
1 # fraction.s
2 # constructs a fraction to be 0/1
3 # Bob Plantz - 18 June 2009
4 # Calling sequence:
5 #     rdi <- address of object
6 #     call  dectoUInt
7 # Include object data definition
8     .include "fraction.h"
9 # Read only data
10    .section .rodata
11 zero:  .long   0
12 one:   .long   1
13 # Code
14    .text
15    .globl  fraction
16    .type   fraction, @function
17 fraction:
18     pushq  %rbp            # save frame pointer
19     movq   %rsp, %rbp      # our frame pointer
20
21     movl   zero, %eax
22     movl   %eax, num(%rdi) # numerator = 0
23     movl   one, %eax
24     movl   %eax, den(%rdi) # denominator = 1
25
26     movq   %rbp, %rsp      # delete local vars.
27     popq   %rbp            # restore frame pointer
28     ret                  # back to caller



---



```
1 # fractionGet.s
2 # gets user values for a fraction
3 # Bob Plantz - 18 June 2009
4 # Calling sequence:
5 #     rdi <- address of object
6 #     call  fractionGet
7 # Include object data definition
8     .include "fraction.h"
9 # local register save area
10    .equ    this,-8        # pointer to object
11    .equ    localSize,-16
12 # Read only data
```


```


```


```

```

13          .section  .rodata
14 numPrompt:
15     .string "Enter numerator: "
16 denPrompt:
17     .string "Enter denominator: "
18 # Code
19     .text
20     .globl fractionGet
21     .type fractionGet, @function
22 fractionGet:
23     pushq %rbp           # save frame pointer
24     movq %rsp, %rbp       # our frame pointer
25     addq $localSize, %rsp
26     movq %rdi, this(%rbp) # save this pointer
27
28     movq $numPrompt, %rdi # prompt for numerator
29     call writeStr
30     call getUInt          # get numerator
31     movq this(%rbp), %rdi # this pointer
32     movl %eax, num(%rdi)  # store in object
33
34     movq $denPrompt, %rdi   # prompt for denominator
35     call writeStr
36     call getUInt          # get denominator
37     movq this(%rbp), %rdi # this pointer
38     movl %eax, den(%rdi)  # store in object
39
40     movq %rbp, %rsp        # delete local vars.
41     popq %rbp             # restore frame pointer
42     ret                   # back to caller
43
44 # fractionDisplay.s
45 # Displays a fraction in num/den format
46 # Bob Plantz - 18 June 2009
47 # Calling sequence:
48 #     rdi <- address of object
49 #     call    fractionDisplay
50 # Include object data definition
51     .include "fraction.h"
52 # local register save area
53     .equ    this,-8        # pointer to object
54     .equ    localSize,-16
55 # Read only data
56     .section  .rodata
57 slash:
58     .string " / "
59 # Code
60     .text
61     .globl fractionDisplay
62     .type fractionDisplay, @function
63 fractionDisplay:
64     pushq %rbp            # save frame pointer

```

```

22      movq    %rsp, %rbp      # our frame pointer
23      addq    $localSize, %rsp # local vars
24      movq    %rdi, this(%rbp) # save this pointer
25
26      movl    num(%rdi), %edi # numerator
27      call    putUInt          # display it
28
29      movq    $slash, %rdi     # "over"
30      call    writeStr
31
32      movq    this(%rbp), %rdi # get this pointer
33      movl    den(%rdi), %edi # denominator
34      call    putUInt          # display it
35
36      movq    %rbp, %rsp      # delete local vars.
37      popq    %rbp            # restore frame pointer
38      ret

```

---

```

1 # fractionAdd.s
2 # adds input value to a fraction
3 # Bob Plantz - 18 June 2009
4 # Calling sequence:
5 #      esi <- int to be added
6 #      rdi <- address of object
7 #      call   fractionAdd
8 # Include object data definition
9      .include "fraction.h"
10 # local register save area
11     .equ    this,-8        # pointer to object
12 # Code
13     .text
14     .globl  fractionAdd
15     .type   fractionAdd, @function
16 fractionAdd:
17     pushq   %rbp            # save frame pointer
18     movq    %rsp, %rbp      # our frame pointer
19     movq    %rdi, this(%rsp) # save this pointer in
20                  # red zone
21     movl    %esi, %eax      # int to be added
22     mull    den(%rdi)       # times denominator
23     movq    this(%rsp), %rdi # restore this pointer
24     addl    %eax, num(%rdi) # add to numerator
25
26     movq    %rbp, %rsp      # delete local vars.
27     popq    %rbp            # restore frame pointer
28     ret

```

**Listing 13.14:** Add 1 to user's fraction (programmer assembly language). (There are six files here; note that the assembly language header file, `fraction.h`, is different from the C++ version.)

The “header” file, `fraction.h`, contains offsets for the fields of the struct that defines the state variables for a fraction object. Notice on line 8 that it includes a symbolic

name for the size of the object.

```
8     .equ    fractionSize,8  # total size needed for struct
```

This makes it easier to define offsets in the stack frame in `main`:

```
7 # Stack frame
8     .equ    x,-fractionSize # Space for a fraction object
9     .equ    localSize,x
```

We then do:

```
20 # Stack frame
21     addq    $localSize, %rsp # local variables
22     andq    $-16, %rsp      # ensure 16-byte boundary
```

technique to allocate space on the stack and make sure the stack pointer is on a 16-byte boundary.

In the `fraction` constructor function we see the use of the field names that are defined in the header file to access the state variables of the object:

```
21     movl    zero, %eax
22     movl    %eax, num(%rdi) # numerator = 0
23     movl    one, %eax
24     movl    %eax, den(%rdi) # denominator = 1
```

The `fraction_add` function is a leaf function. So we use the red zone for saving the `this` pointer:

```
16 fraction_add:
17     pushq   %rbp          # save base pointer
18     movq    %rsp, %rbp    # our frame pointer
19     movq    %rdi, this(%rsp) # save this pointer in
20                           #       red zone
```

Be careful not to use the red zone in non-leaf functions.

## 13.5 Instructions Introduced Thus Far

This summary shows the assembly language instructions introduced thus far in the book. The page number where the instruction is explained in more detail, which may be in a subsequent chapter, is also given. This book provides only an introduction to the usage of each instruction. You need to consult the manuals ([2] - [6], [14] - [18]) in order to learn all the possible uses of the instructions.

### 13.5.1 Instructions

*data movement:*

opcode	source	destination	action	see page:
cmovecc	%reg/mem	%reg	conditional move	260
movs	\$imm/%reg	%reg/mem	move	156
movss	\$imm/%reg	%reg/mem	move, sign extend	245
movzs	\$imm/%reg	%reg/mem	move, zero extend	245
popw		%reg/mem	pop from stack	181
pushw	\$imm/%reg/mem		push onto stack	181

*s = b, w, l, q; w = l, q; cc = condition codes*

*arithmetic/logic:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
adds	\$imm/%reg	%reg/mem	add	214
adds	mem	%reg	add	214
ands	\$imm/%reg	%reg/mem	bit-wise and	290
ands	mem	%reg	bit-wise and	290
cmps	\$imm/%reg	%reg/mem	compare	237
cmps	mem	%reg	compare	237
decs	%reg/mem		decrement	249
divs	%reg/mem		unsigned divide	315
idivs	%reg/mem		signed divide	317
imuls	%reg/mem		signed multiply	310
incs	%reg/mem		increment	248
leaw	mem	%reg	load effective address	191
subs	\$imm/%reg	%reg/mem	subtract	215
muls	%reg/mem		unsigned multiply	309
negs	%reg/mem		negate	322
ors	\$imm/%reg	%reg/mem	bit-wise inclusive or	290
ors	mem	%reg	bit-wise inclusive or	290
sals	\$imm/%cl	%reg/mem	shift arithmetic left	302
sars	\$imm/%cl	%reg/mem	shift arithmetic right	301
shls	\$imm/%cl	%reg/mem	shift left	302
shrs	\$imm/%cl	%reg/mem	shift right	301
subs	mem	%reg	subtract	215
tests	\$imm/%reg	%reg/mem	test bits	238
tests	mem	%reg	test bits	238
xors	\$imm/%reg	%reg/mem	bit-wise exclusive or	290
xors	mem	%reg	bit-wise exclusive or	290

*s = b, w, l, q; w = l, q**program flow control:*

<b>opcode</b>	<b>location</b>	<b>action</b>	<b>see page:</b>
call	label	call function	173
ja	label	jump above (unsigned)	239
jae	label	jump above/equal (unsigned)	239
j b	label	jump below (unsigned)	239
j be	label	jump below/equal (unsigned)	239
je	label	jump equal	239
jg	label	jump greater than (signed)	240
j ge	label	jump greater than/equal (signed)	240
jl	label	jump less than (signed)	240
j le	label	jump less than/equal (signed)	240
jmp	label	jump	241
jne	label	jump not equal	239
jno	label	jump no overflow	239
jcc	label	jump on condition codes	239
leave		undo stack frame	192
ret		return from function	192
syscall		call kernel function	201

*cc = condition codes*

### 13.5.2 Addressing Modes

<b>register direct:</b>	The data value is located in a CPU register. <i>syntax:</i> name of the register with a "%" prefix. <i>example:</i> movl %eax, %ebx
<b>immediate data:</b>	The data value is located immediately after the instruction. Source operand only. <i>syntax:</i> data value with a "\$" prefix. <i>example:</i> movl \$0xabcd1234, %ebx
<b>base register plus offset:</b>	The data value is located in memory. The address of the memory location is the sum of a value in a base register plus an offset value. <i>syntax:</i> use the name of the register with parentheses around the name and the offset value immediately before the left parenthesis. <i>example:</i> movl \$0xaabbccdd, 12(%eax)
<b>rip-relative:</b>	The target is a memory address determined by adding an offset to the current address in the rip register. <i>syntax:</i> a programmer-defined label <i>example:</i> je somePlace
<b>indexed:</b>	The data value is located in memory. The address of the memory location is the sum of the value in the base_register plus scale times the value in the index_register, plus the offset. <i>syntax:</i> place parentheses around the comma separated list (base_register, index_register, scale) and preface it with the offset. <i>example:</i> movl \$0x6789cdef, -16(%edx, %eax, 4)

## 13.6 Exercises

**13-1** (§13.1) Write a program in assembly language that allocates a twenty-five element integer array and stores the index value in each element. That is, the first element will be assigned zero, the second element one, etc. Use four-byte integers.

After the array has been completely initialized, display the contents of the array in a column.

**13-2** (§13.1) Write a program in assembly language that allocates a ten element integer array and prompts the user to enter an integer to be stored in each element of the array. Use four-byte integers.

After the user's values have been stored in the array, compute the sum of the integers. (Do not accumulate the sum as the numbers are entered.) Display the sum.

**13-3** (§13.1) Write a program in assembly language that allocates a ten element integer array and prompts the user to enter an integer to be stored in each element of the array. Use four-byte integers.

After the user's values have been stored in the array, compute the average of the integers. (Do not accumulate the sum as the numbers are entered.) Display the average.

**13-4** (§13.2) Modify the program in Listing 13.6 so that it displays the total number of bytes allocated for the struct. Hint: use the C `sizeof` operator.

**13-5** (§13.2) Modify the program of Exercise 13-4 such that it also displays the offset of each field within a struct. Hint: use the C `&` operator to get addresses.

**13-6** (§13.2) Enter the program in Listing 13.8 and make sure that you understand how it works.

**13-7** (§13.3) Enter the three files from Listing 13.11 and get the program to work.

- a) Create a `makefile` to assemble and link the files into a program.
- b) Using the debugger, `gdb`, set breakpoints in the `main` function at each call to `loadStruct` and at the instruction immediately following each call.
- c) Use the debugger to observe the values that are stored in the fields of the `aByte`, `anInt`, and `anotherByte` fields each time `loadStruct` is called. Hint: Note the address in `rdi` just before executing each function call.

**13-8** (§13.3) Modify the program in Listing 13.8 such that it has separate functions for:

- getting the data from the user for a `struct`, and
- displaying the data in a `struct`.

Add two more `struct` variables to the program. Your program will then call the first function three times, once for each variable. Then it calls the second function three times, also once for each variable.

**13-9** (§13.3) Write a program in assembly language that allocates three variables of the type:

```
struct item {  
    char name[50];  
    int cost;  
};
```

That is, each variable will have two fields, one for the name of the item, and one for its cost.

The user will be prompted to enter the name and cost of each item, and the user's input will be stored in the respective variables.

After the data for all three items is entered, the program will list the name and cost of each of the three items and then display the total cost for all three items.

**13-10** (§13.4) Implement the program in Listing 13.14 such that it allows the user to enter an integer value to be added to the fraction.

**13-11** (§13.4) Modify the program in Exercise 13-10 such that it allows the user to enter a fractional value, then adds the two fractions.

**13-12** (§13.4) Write a program that allows the user to maintain an address book. Each entry into the address book should allow the user to enter

- 48 characters for the name
- 80 characters for the street address
- 24 characters for the city
- 2 characters for the state (abbreviation)

- 5 characters for the zip code

The user should be able to display the entries.

**13-13** (§13.4) Modify the program in Exercise 13-12 so that the user can sort the address book entries on any of the five fields.

**13-14** (§13.4) Write a program that allows the user to set up and maintain two bank accounts. Each account should have a unique account name. The user should be able to credit or debit the account.

**13-15** (§13.4) Modify the program in Exercise 13-14 so that it requires a pin number in order to access each of the bank accounts.

# Chapter 14

# Fractional Numbers

So far in this book we have used only integers for numerical values. In this chapter you will see two methods for storing fractional values — fixed point and floating point. Storing numbers in fixed point format requires that the programmer keep track of the location of the binary point<sup>1</sup> within the bits allocated for storage. In the floating point format, the number is essentially written in scientific format and both the significand<sup>2</sup> and exponent are stored.

## 14.1 Fractions in Binary

Before discussing the storage formats, we need to think about how *fractional values* are stored in binary. The concept is quite simple. We can extend Equation 2.6

$$N = d_{n-1} \times 2^{n-1} + d_{n-2} \times 2^{n-2} + \dots + d_1 \times 2^1 + d_0 \times 2^0 + d_{-1} \times 2^{-1} + d_{-2} \times 2^{-2} + \dots \quad (14.1)$$

For example,

$$123.6875_{10} = 11110111.1011_2$$

because

$$\begin{aligned}d_{-1} \times 2^{-1} &= 1 \times 0.5 \\d_{-2} \times 2^{-2} &= 0 \times 0.25 \\d_{-3} \times 2^{-3} &= 1 \times 0.125 \\d_{-4} \times 2^{-4} &= 1 \times 0.0625\end{aligned}$$

and thus

$$\begin{aligned}0.1011_2 &= 0.5_{10} + 0.125_{10} + 0.0625_{10} \\&= 0.6875_{10}\end{aligned} \quad (14.2)$$

See Exercise 14-1 for an algorithm to convert decimal fractions to binary. We assume that you can convert the integral part and that Equation 14.1 is sufficient for converting from binary to decimal.

---

<sup>1</sup>The binary point is equivalent to the decimal point when a number is stored in binary. In particular, it separates the integral and fractional parts.

<sup>2</sup>This is often called the “mantissa,” which means the fractional part of a logarithm.

Although any integer can be represented as a sum of powers of two, an exact representation of fractional values in binary is limited to sums of *inverse* powers of two. For example, consider an 8-bit representation of the fractional value 0.9. From

$$\begin{aligned}0.11100110_2 &= 0.89843750_{10} \\0.11100111_2 &= 0.90234375_{10}\end{aligned}$$

we can see that

$$0.11100110_2 < 0.9_{10} < 0.11100111_2 \quad (14.3)$$

In fact,

$$0.9_{10} = 0.\overline{11100110}_2 \quad (14.4)$$

where  $\overline{1100}$  means that this bit pattern repeats forever.

Rounding off fractional values in binary is very simple. If the next bit to the right is one, add one to the bit position where rounding off. In the above example, we are rounding off to eight bits. The ninth bit to the right of the binary point is zero, so we do not add one in the eighth bit position. Thus, we use

$$0.9_{10} = 0.1110\ 0110_2 \quad (14.5)$$

which gives a round off error of

$$\begin{aligned}0.9_{10} - 0.11100110_2 &= 0.9_{10} - 0.8984375_{10} \\&= 0.0015625_{10}\end{aligned} \quad (14.6)$$

We note here that two's complement also works correctly for storing negative fractional values. You are asked to show this in Exercise 14-2.

## 14.2 Fixed Point ints

In a *fixed point* format, the storage area is divided into the integral part and the fractional part. The programmer must keep track of where the binary point is located. For example, we may decide to divide a 32-bit int in half and use the high-order 16 bits for the integral part and the low-order 16 bits for the fractional part.

My bank provides me with printed deposit slips that use this method. There are seven boxes for numerals. There is also a decimal point printed just to the left of the rightmost two boxes. Note that the decimal point does not occupy a box. That is, there are no digits allocated for the decimal point. So the bank assumes up to five decimal digits for the "dollars" part (rather optimistic), and the rightmost two decimal digits represent the "cents" part. The bank's printing tells me how they have allocated the digits, but it is my responsibility to keep track of the location of the decimal point when filling in the digits.

One advantage of a fixed point format is that integer instructions can be used for arithmetic computations. Of course, the programmer must be very careful to keep track of which bits are allocated for the integral part and which for the fractional part. And the range of possible values is restricted by the number of bits.

An example of using ints for fixed point addition is shown in Listing 14.1.

---

```

1 /* 
2 * rulerAdd.c
3 * Adds two ruler measurements, to nearest 1/16th inch.
4 * Bob Plantz - 18 June 2009
5 */
6 #include <stdio.h>

```

```

7
8 int main(void)
9 {
10    int x, y, fraction_part, sum;
11
12    printf("Enter first measurement, inches: ");
13    scanf("%d", &x);
14    x = x << 4;           /* shift to integral part of variable */
15    printf("                         sixteenths: ");
16    scanf("%d", &fraction_part);
17    x = x | (0xf & fraction_part); /* add in fractional part */
18
19    printf("Enter second measurement, inches: ");
20    scanf("%d", &y);
21    y = y << 4;           /* shift to integral part of variable */
22    printf("                         sixteenths: ");
23    scanf("%d", &fraction_part);
24    y = y | (0xf & fraction_part); /* add in fractional part */
25
26    sum = x + y;
27    printf("Their sum is %d and %d/16 inches\n",
28           (sum >> 4), (sum & 0xf));
29
30    return 0;
31 }
```

Listing 14.1: Fixed point addition. The high-order 28 bits are used for the integral part, the low-order 4 for the fractional part.)

The numbers are input to the nearest 1/16th inch, so the programmer has allocated four bits for the fractional part. This leaves 28 bits for the integral part. After the integral part is read, the stored number must be shifted four bit positions to the left to put it in the high-order 28 bits. Then the fractional part (in number of sixteenths) is added into the low-order four bits with a simple bit-wise or operation. Printing the answer also requires some bit shifting and some masking to filter out the fractional part.

This is clearly a contrived example. A program using floats would work just as well and be somewhat easier to write. However, the program in Listing 14.1 uses integer instructions, which execute faster than floating point. The hardware issues have become less significant in recent times. Modern CPUs use various parallelization schemes such that a mix of floating point and integer instructions may actually execute faster than only integer instructions. Fixed point arithmetic is often used in embedded applications where the CPU is small and may not have floating point capabilities.

## 14.3 Floating Point Format

The most important concept in this section is that *floating point* numbers are not *real numbers*.<sup>3</sup> Real numbers include the continuum of all numbers from  $-\infty$  to  $+\infty$ . You already understand that computers are finite, so there is certainly a limit on the largest values that can be represented. But the problem is much worse than simply a size limit.

---

<sup>3</sup>It is unfortunate that Pascal uses the keyword “real” for the floating point type.

As you will see in this section, floating point numbers comprise a very small subset of real numbers. There are significant gaps between adjacent floating point numbers. These gaps can produce the following types of errors:

- Rounding — the number cannot be exactly represented in floating point.
- Absorption — a very small number gets lost when adding it to a large one.
- Cancellation — a very small number gets lost when subtracting it from a large one.

To make matters worse, these errors can occur in intermediate results, where they are very difficult to debug.

The idea behind floating point formats is to think of numbers written in scientific format. This notation requires two numbers to completely specify a value — a significand and an exponent. To review, a decimal number is written in scientific notation as a significand times ten raised to an exponent. For example,

$$1,024 = 1.024 \times 10^3 \quad (14.7)$$

$$-0.000089372 = -8.9372 \times 10^{-5} \quad (14.8)$$

Notice that the number is *normalized* such that only one digit appears to the left of the decimal point. The exponent of 10 is adjusted accordingly.

If we agree that each number is normalized and that we are working in base 10, then each floating point number is completely specified by three items:

1. The significand.
2. The exponent.
3. The sign.

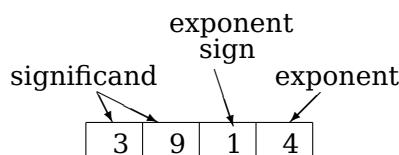
That is, in the above examples

- 1024, 3, and + represent  $1.024 \times 10^3$  (The “+” is understood.)
- 89372, -5, and - represent  $8.9372 \times 10^{-5}$

The advantage of using a floating point format is that, for a given number of digits, we can represent a larger range of values. To illustrate this, consider a four-digit, unsigned decimal system. The range of integers that could be represented is

$$0 \leq N \leq 9999$$

Now, let's allocate two digits for the significand and two for the exponent. We will restrict our scheme to unsigned numbers, but we will allow negative exponents. So we will need to use one of the exponent digits to store the sign of the exponent. We will use 0 for positive and 1 for negative. For example,  $3.9 \times 10^{-4}$  would be stored



where each box holds one decimal digit. Some other examples are:

$$\begin{aligned} 1000 &\Leftrightarrow 1.0 \times 10^0 \\ 3702 &\Leftrightarrow 3.7 \times 10^2 \\ 9316 &\Leftrightarrow 9.3 \times 10^{-6} \end{aligned}$$

Our normalization scheme requires that there be a single non-zero digit to the left of the decimal point. We should also allow the special case of 0.0:

$$0000 \Leftrightarrow 0.0 \times 10^0$$

A little thought shows that this scheme allows numbers in the range

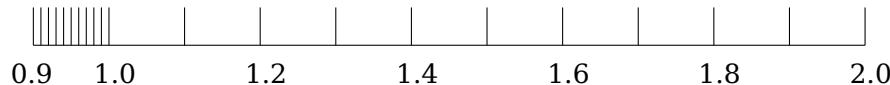
$$1.0 \times 10^{-9} \leq N \leq 9.9 \times 10^9$$

That is, we have increased the range of possible values by a factor of  $10^{14}$ ! However, it is important to realize that in both storage schemes, the integer and the floating point, we have exactly the same number of possible values —  $10^4$ .

Although floating point formats can provide a much greater range of numbers, the distance between any two adjacent numbers depends upon the value of the exponent. Thus, floating point is generally *less* accurate than an integer representation, especially for large numbers.

To see how this works, let's look at a plot of numbers (using our current scheme) in the range

$$9.0 \times 10^{-1} \leq N \leq 2.0 \times 10^0$$



Notice that the larger numbers are further apart than the smaller ones. (See Exercise 14-7 after you read Section 14.4.)

Let us pick some numbers from this range and perform some addition.

$$9111 \Leftrightarrow 9.1 \times 10^{-1}$$

$$9311 \Leftrightarrow 9.3 \times 10^{-1}$$

If we add these values, we get  $0.91 + 0.93 = 1.84$ . Now we need to round off our “paper” addition in order to fit this result into our current floating point scheme:

$$1800 \Leftrightarrow 1.8 \times 10^0$$

On the other hand,

$$9411 \Leftrightarrow 9.4 \times 10^{-1}$$

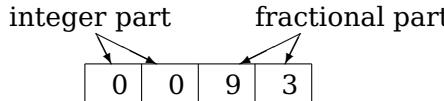
$$9311 \Leftrightarrow 9.3 \times 10^{-1}$$

and adding these values, we get  $0.94 + 0.93 = 1.87$ . Rounding off, we get:

$$1900 \Leftrightarrow 1.9 \times 10^0$$

So we see that starting with two values expressed to the nearest 1/100th, their sum is accurate only to the nearest 1/10.

To compare this with fixed point arithmetic, we could use the same four digits to store 0.93 this way



It is clear that this storage scheme allows us to perform both additions ( $0.91 + 0.93$  and  $0.94 + 0.93$ ) and store the results exactly.

These *round off errors* must be taken into account when performing floating point arithmetic. In particular, the errors can occur in intermediate results when doing even moderately complex computations, where they are very difficult to detect. 4

## 14.4 IEEE 754

Specific floating point formats involve trade-offs between resolution, round off errors, size, and range. The most commonly used formats are the *IEEE 754*.<sup>4</sup> They range in size from four to sixteen bytes. The most common sizes used in C/C++ are floats (4 bytes) and doubles (8 bytes). The x86 processor performs floating point computations using an extended 10-byte format. The results are rounded to 4-byte mode if the programmer uses the float data type or 8-byte mode for the double data type.

In the IEEE 754 4-byte format, one bit is used for the sign, eight for the exponent, and twenty-three for the significand. The IEEE 754 8-byte format specifies one bit for the sign, eleven for the exponent, and fifty-two for the significand.

In this section we describe the 4-byte format in order to save ourselves (hand) computation effort. The goal is to get a feel for the *limitations* of floating point formats. The normalized form of the number in binary is given by Equation 14.9.

$$N = (-1)^s \times 1.f \times 2^e \quad (14.9)$$

where:  $s$  is the sign bit

$f$  is the 23-bit fractional part

$e$  is the 8-bit exponent

The bit patterns for floats and doubles are arranged as shown in Figure 14.1.

(a)	<table border="1"> <tr> <td><math>s</math></td><td><math>e+127</math></td><td><math>f</math></td><td></td></tr> </table>	$s$	$e+127$	$f$	
$s$	$e+127$	$f$			
	31 30    23 22		0		
(b)	<table border="1"> <tr> <td><math>s</math></td><td><math>e+1023</math></td><td><math>f</math></td><td></td></tr> </table>	$s$	$e+1023$	$f$	
$s$	$e+1023$	$f$			
	63 62    52 51		0		

Figure 14.1: IEEE 754 bit patterns. (a) Float. (b) Double.

As in decimal, the exponent is adjusted such that there is only one non-zero digit to the left of the binary point. In binary, though, this digit is always one. Since it is always one, it need not be stored. Only the fractional part of the normalized value needs to be stored as the significand. This adds one bit to the significance of the fractional part. The integer part (one) that is not stored is sometimes called the *hidden bit*.

The sign bit,  $s$ , refers to the number. Another mechanism is used to represent the sign of the exponent,  $e$ .<sup>4</sup> Your first thought is probably to use two's complement. However, the IEEE format was developed in the 1970s, when floating point computations took a

<sup>4</sup>There is a slightly newer standard, IEEE 854. It is a generalization of IEEE 754.

lot of CPU time. Many algorithms depend upon only the comparison of two numbers, and the computer scientists of the day realized that a format that allowed integer comparison instructions would result in faster execution times. So they decided to store a *biased exponent* as an *unsigned int*. The amount of the bias is one-half the range allocated for the exponent. In the case of an 8-bit exponent, the bias amount is 127.

### Example 14-c

---

Show how 97.8125 is stored in 32-bit IEEE 754 binary format.

*Solution:*

First, convert the number to binary.

$$\begin{aligned} 97.8125_{10} &= 1100001.1101_2 \\ &= (-1)^0 \times 1100001.1101 \times 2^0 \end{aligned}$$

Adjust the exponent to obtain the normalized form.

$$(-1)^0 \times 1100001.1101 \times 2^0 = (-1)^0 \times 1.1000011101 \times 2^6$$

Compute  $s$ ,  $e+127$ , and  $f$ .

$$\begin{aligned} s &= 0 \\ e + 127 &= 6 + 127 \\ &= 133 \\ &= 10000101_2 \\ f &= 100001110100000000000000 \end{aligned}$$

Finally, use Figure 14.1 to place the bit patterns. (Remember that the *hidden* bit is not stored; it is understood to be there.)

$$\begin{aligned} 97.8125 &= 0\ 10000101\ 1000011101000000000000_2 \\ &= 42c3a000_{16} \end{aligned}$$


---



### Example 14-d

---

Using IEEE 754 32-bit format, what decimal number does the bit pattern  $3e400000_{16}$  represent?

*Solution:*

First, convert the hexadecimal to binary, using spaces suggested by Figure 14.1.

$$3e400000_{16} = 0\ 01111100\ 100000000000000000000000_2$$

Now compute the values of  $s$ ,  $e$ , and  $f$ .

$$\begin{aligned} s &= 0 \\ e + 127 &= 01111100_2 \\ &= 124_{10} \\ e &= -3_{10} \\ f &= 10000000000000000000000000000000 \end{aligned}$$

Finally, plug these values into Equation 14.9. (Remember to add the *hidden bit*.)

$$\begin{aligned} (-1)^0 \times 1.100\ldots00 \times 2^{-3} &= (-1)^0 \times 0.0011 \times 2^0 \\ &= 0.1875_{10} \end{aligned}$$


---



### Example 14-e

Using IEEE 754 32-bit format, what decimal number would the bit pattern  $00000000_{16}$  represent? (The specification states that it is an exception to the format and is defined to represent 0.0. This example provides some motivation for this exception.)

*Solution:*

The conversion to binary is trivial. Computing the values of  $s$ ,  $e$ , and  $f$ .

$$\begin{aligned} s &= 0 \\ e + 127 &= 00000000_2 \\ e &= -127_{10} \\ f &= 00000000000000000000000000000000 \end{aligned}$$

Finally, plug these values into Equation 14.9. (Remember to add the *hidden bit*.)

$$(-1)^0 \times 1.00\ldots00 \times 2^{-127} = \text{very small number}$$


---



This last example illustrates a problem with the hidden bit — there is no way to represent zero. To address this issue, the IEEE 754 standard has several special cases.

- **Zero** — all the bits in the exponent and significand are zero. Notice that this allows for  $-0.0$  and  $+0.0$ , although  $(-0.0) == +0.0$ ) computes to `true`.
- **Denormalized** — all the bits in the exponent are zero. In this case there is no hidden bit. Zero can be thought of as a special case of denormalized.
- **Infinity** — all the bits in the exponent are one, and all the bits in the significand are zero. The sign bit allows for  $-\infty$  and  $+\infty$ .
- **NaN** — all the bits in the exponent are one, and the significand is non-zero. This is used when the results of an operation are undefined. For example,  $\pm\text{nonzero} \div 0.0$  yields infinity, but  $\pm 0.0 \div \pm 0.0$  yields NaN.

## 14.5 Floating Point Hardware

Until the introduction of the Intel 486DX in April 1989, the *x87 Floating Point Unit* was on a separate chip. It is now included on the CPU chip although it uses a somewhat different execution architecture than the Integer Unit in the CPU.

In 1997 Intel added *MMX™* (Multimedia Extensions) to their processors, which includes instructions that process multiple data values with a single instruction instruction (SIMD). Operations on single data items are called *scalar* operations, while those on multiple data items in parallel are called *vector* operations. Vector operations are useful

for many multi-media and scientific applications. In this book we will discuss only scalar floating point operations. Originally, MMX only performed integer computations. But in 1998 AMD added the *3DNow!*™ extension to MMX, which includes floating point instructions. Intel soon followed suit.

Intel then introduced *SSE* (Streaming SIMD Extension) on the Pentium III in 1999. Several versions have evolved over the years — SSE, SSE2, SSE3, and SSE4A — as of this writing. There are instructions for performing both integer and floating point operations on both scalar and vector values.

The x86-64 architecture includes three sets of instructions for working with floating point values:

- SSE2 instructions operate on 32-bit or 64-bit values.<sup>5</sup> Four 32-bit values or two 64-bit values can be processed simultaneously.
- **x87 Floating Point Unit** instructions operate on 80-bit values.
- **3DNow!** instructions operate on two 32-bit values.

All three floating point instruction sets include a wide variety of instructions to perform the following operations:

- Move data from memory to a register, from a register to memory, and from a register to another register.
- Convert data from integer to floating point, and from floating point to integer formats.
- Perform the usual add, subtract, multiply, and divide arithmetic operations. They also provide square root instructions.
- Compare two values.
- Perform the usual and, or, and xor logical operations.

In addition, the x87 includes instructions for transcendental functions — sine, cosine, tangent, and arc tangent, and logarithm functions.

We will not cover all the instructions in this book. The following subsections provide an introduction to how each of the three sets of instructions is used. See the manuals [2] - [6] and [14] - [18] for details.

### 14.5.1 SSE2 Floating Point

Most of the SSE2 instructions operate on multiple data items simultaneously — *single instruction, multiple data* (SIMD). There are SSE2 instructions for both integer and floating point operations. Integer instructions operate on up to sixteen 8-bit, eight 16-bit, four 32-bit, two 64-bit, or one 128-bit integers at a time. Vector floating point instructions operate on all four 32-bit or both 64-bit floats in a register simultaneously. Each data item is treated independently. These instructions are useful for algorithms that do things like process arrays. One SSE2 instruction can operate on several array elements in parallel, resulting in considerable speed gains. Such algorithms are common in multi-media and scientific applications.

In this book we will only consider some of the scalar floating-point instructions, which operate on only single data items. The SSE2 instructions are the preferred floating-point

<sup>5</sup>Newer x86-64 processors have later versions of SSE, but SSE2 is part of the definition of x86-64, so it is the only version that can be assumed to be available.

implementation in 64-bit mode. These instructions operate on either 32-bit (single-precision) or 64-bit (double-precision) values. The scalar instructions operate on only the low-order portion of the 128-bit `xmm` registers, with the high-order 64 or 96 bits remaining unchanged.

SSE includes a 32-bit MXCSR register that has flags for handling floating-point arithmetic errors. These flags are shown in Table 14.1. SSE instructions that perform

<b>bits</b>	<b>mnemonic</b>	<b>meaning</b>	<b>default</b>
31 - 18	-	reserved	
17	MM	Misaligned Exception Mask	0
16	-	reserved	
15	FZ	Flush-toZero for Masked Underflow	0
14 - 13	RC	Floating-Point Rounding Control	00
12	PM	Precision Exception Mask	1
11	UM	Underflow Exception Mask	1
10	OM	Overflow Exception Mask	1
9	ZM	Zero-Divide Exception Mask	1
8	DM	Denormalized-Operand Exception Mask	1
7	IM	Invalid-Operation Exception Mask	1
6	DAZ	Denormals Are Zero	0
5	PE	Precision Exception	0
4	UE	Underflow Exception 4	0
3	OE	Overflow Exception	0
2	ZE	Zero-Divide Exception	1
1	DE	Denormalized-Operand Exception	0
0	IE	Invalid-Operation Exception	0

Table 14.1: MXCSR status register.

arithmetic operations and the SSE compare instructions also affect the status flags in the `rflags` register. Thus the regular conditional jump instructions (Section 10.1.2, page 238) are used to control program flow based on floating-point computations.

The instruction mnemonics used by the gnu assembler are mostly the same as given in the manuals, [2] - [6] and [14] - [18]. Since they are quite descriptive with respect to operand sizes, a size letter is not appended to the mnemonic, except when one of the operands is in memory and the size is ambiguous. Of course, the operand order used by the gnu assembler is still reversed compared to the manufacturers' manuals, and the register names are prefixed with the "%" character.

A very important set of instructions provided for working with floating point values are those to convert between integer and floating point formats. The scalar conversion SSE2 instructions are shown in Table 14.2.

Data movement and arithmetic instructions must distinguish between scalar and vector operations on values in the `xmm` registers. The low-order portion of the register is used for scalar operations. Vector operations are performed on multiple data values packed into a single register. See Table 14.3 for a sampling of SSE2 data movement and arithmetic instructions. Notice that the code for the basic operation is followed by a "p" or "s" for "packed" or "scalar." This character is then followed by a "d" or "s" for "double" (64-bit) or "single" (32-bit) data item.

<b>mnemonic</b>	<b>source</b>	<b>destination</b>	<b>meaning</b>
cvtsd2si	xmm reg. or mem.	32-bit integer reg.	convert scalar 64-bit float to 32-bit integer
cvtsd2ss	xmm reg.	xmm reg. or mem.	convert scalar 64-bit float to 32-bit float
cvtssi2sd	integer reg. or mem.	xmm reg.	convert 32-bit integer to scalar 64-bit float
cvtssi2sdq	integer reg. or mem.	xmm reg.	convert 64-bit integer to scalar 64-bit float
cvtssi2ss	integer reg. or mem.	xmm reg.	convert 32-bit integer to scalar 32-bit float
cvtssi2ssq	integer reg. or mem.	xmm reg.	convert 64-bit integer to scalar 32-bit float
cvtss2sd	xmm reg.	another xmm reg. or mem.	convert scalar 32-bit float to 64-bit float
cvtss2si	xmm reg. or mem.	32-bit integer reg.	convert scalar 32-bit float to 32-bit integer
cvtss2siq	xmm reg. or mem.	64-bit integer reg.	convert scalar 32-bit float to 64-bit integer

Table 14.2: SSE scalar floating point conversion instructions. Source and destination `xmm` registers must be different. The low-order portion of the `xmm` register is used. When reading from or writing to memory, the “q” suffix is used to designate a 64-bit value.

<b>mnemonic</b>	<b>source</b>	<b>destination</b>	<b>meaning</b>
addps	xmm reg. or mem.	xmm reg.	add packed 32-bit floats
addpd	xmm reg. or mem.	xmm reg.	add packed 64-bit floats
addss	xmm reg. or mem.	xmm reg.	add scalar 32-bit floats
addsd	xmm reg. or mem.	xmm reg.	add scalar 64-bit floats
divps	xmm reg. or mem.	xmm reg.	divide packed 32-bit floats
divpd	xmm reg. or mem.	xmm reg.	divide packed 64-bit floats
divss	xmm reg. or mem.	xmm reg.	divide scalar 32-bit floats
divsd	xmm reg. or mem.	xmm reg.	divide scalar 64-bit floats
movss	xmm reg. or mem.	xmm reg.	move scalar 32-bit float
movss	xmm reg.	xmm reg. or mem.	move scalar 32-bit float
movsd	xmm reg. or mem.	xmm reg.	move scalar 64-bit float
movsd	xmm reg.	xmm reg. or mem.	move scalar 64-bit float
mulps	xmm reg. or mem.	xmm reg.	multiply packed 32-bit floats
mulpd	xmm reg. or mem.	xmm reg.	multiply packed 64-bit floats
mulss	xmm reg. or mem.	xmm reg.	multiply scalar 32-bit floats
mulsd	xmm reg. or mem.	xmm reg.	multiply scalar 64-bit floats
subps	xmm reg. or mem.	xmm reg.	subtract packed 32-bit floats
subpd	xmm reg. or mem.	xmm reg.	subtract packed 64-bit floats
subss	xmm reg. or mem.	xmm reg.	subtract scalar 32-bit floats
subsd	xmm reg. or mem.	xmm reg.	subtract scalar 64-bit floats

Table 14.3: Some SSE floating point arithmetic and data movement instructions. Source and destination `xmm` registers must be different. Scalar instructions use the low-order portion of the `xmm` registers.

We will use the program in Listing 14.2 to illustrate a few floating point operations.

---

```

1  /*
2   * frac2float.c
3   * Converts fraction to floating point.
4   * Bob Plantz - 18 June 2009
5   */
6
7 #include <stdio.h>
8
9 int main(void)
10 {
11     int x, y;
12     double z;
13
14     printf("Enter two integers: ");
15     scanf("%i %i", &x, &y);
16     z = (double)x / y;
17     printf("%i / %i = %lf\n", x, y, z);
18     return 0;
19 }
```

---

Listing 14.2: Converting a fraction to a float.

Compiling this program in 64-bit mode produced the assembly language in Listing 14.3.

---

```

1      .file    "frac2float.c"
2      .section .rodata
3 .LC0:   .string "Enter two integers: "
4 .LC1:   .string "%i %i"
5 .LC2:   .string "%i / %i = %lf\n"
6
7 .text
8     .globl  main
9     .type   main, @function
10
11 main:
12     pushq  %rbp
13     movq  %rsp, %rbp
14     subq  $32, %rsp
15     movl  $.LC0, %edi
16     movl  $0, %eax
17     call   printf
18     leaq   -12(%rbp), %rdx      # address of y
19     leaq   -16(%rbp), %rax      # address of x
20     movq  %rax, %rsi
21     movl  $.LC1, %edi
22     movl  $0, %eax             # no xmm arguments
23     call   __isoc99_scanf
24     movl  -16(%rbp), %eax      # load x
25     cvtsi2sd    %eax, %xmm0 # convert to double
26     movl  -12(%rbp), %eax      # load y
27
```

---

```

28      cvtsi2sd    %eax, %xmm1 # convert to double
29      divsd     %xmm1, %xmm0      # (double)x / (double)y
30      movsd     %xmm0, -8(%rbp)   # store z
31      movl      -12(%rbp), %edx  # load y
32      movl      -16(%rbp), %ecx  # load x
33      movq      -8(%rbp), %rax
34      movq      %rax, -24(%rbp)
35      movsd     -24(%rbp), %xmm0 # load z
36      movl      %ecx, %esi       # x to correct register
37      movl      $.LC2, %edi
38      movl      $1, %eax        # one xmm argument (in xmm0)
39      call      printf
40      movl      $0, %eax
41      leave
42      ret
43      .size    main, .-main
44      .ident   "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
45      .section .note.GNU-stack, "",@progbits

```

Listing 14.3: Converting a fraction to a float (gcc assembly language, 64-bit).

Before the division is performed, both integers must be converted to floating point. This takes place on lines 25 – 28:

```

25      movl      -16(%rbp), %eax  # load x
26      cvtsi2sd    %eax, %xmm0 # convert to double
27      movl      -12(%rbp), %eax  # load y
28      cvtsi2sd    %eax, %xmm1 # convert to double

```

The `cvtsi2sd` instruction on lines 26 and 28 converts a signed integer to a scalar double-precision floating point value. The signed integer can be either 32 or 64 bits and can be located in a general purpose register or in memory. The double-precision float will be stored in the low-order 64 bits of the specified `xmmn` register. The high-order 64 bits to the `xmmn` register are not changed.

The division on line 29 leaves the result in the low-order 64 bits of `xmm0`, which is then stored in `z`:

```

29      divsd     %xmm1, %xmm0      # (double)x / (double)y
30      movsd     %xmm0, -8(%rbp)   # store z

```

The `movapd` instruction moves the entire 128 bits, and the `movsd` instruction moves only the low-order 64 bits.

The floating point arguments are passed in the registers `xmm0`, `xmm1`, ..., `xmm15` in left-to-right order. So the value of `z` is loaded into the (`xmm0`) register for passing to the `printf` function, and the number of floating point values passed to it must be stored in `eax`:

```

35      movsd     -24(%rbp), %xmm0 # load z
36      movl      %ecx, %esi       # x to correct register
37      movl      $.LC2, %edi
38      movl      $1, %eax        # one xmm argument (in xmm0)
39      call      printf

```

### 14.5.2 x87 Floating Point Unit

The x87 FPU has eight 80-bit data registers, its own status register, and its own stack pointer. Floating point values are stored in the floating point data registers in an *extended* format.:

- bit 79 is the sign bit: 0 for positive, 1 for negative.
- bits 78 – 64 are for the exponent: 2's complement, biased by 16383.
- bit 63 is the integer: 1 for normalized.
- bits 62 – 0 are for the fraction.

So there are 64 bits (63 – 0) for the significand. Since there is no hidden bit in the extended format, one of these bits, bit 63, is required for the integer part.

#### Example 14-f

---

Show how 97.8125 is stored in 80-bit extended IEEE 754 binary format.

*Solution:*

First, convert the number to binary.

$$97.8125_{10} = 1100001.1101_2 \\ (-1)^0 \times 1100001.1101 \times 2^0$$

Adjust the exponent to obtain the normalized form.

$$(-1)^0 \times 1100001.1101 \times 2^0 = (-1)^0 \times 1.1000011101 \times 2^6$$

Compute  $s, e+16383$ .

$$\begin{aligned} s &= 0 \\ e + 16383 &= 6 + 16383 \\ &= 16389 \\ &= 100000000000101_2 \end{aligned}$$

Filling in the bit patterns as specified above:

$$\begin{aligned} 97.8125 &= 0\ 100000000000101\ 1\ 1000011101000000000000\dots0_2 \\ &= 4005c3a0000000000000000_{16} \end{aligned}$$

Compare this with the 32-bit format in Example 14-a above.

The 16-bit Floating Point Unit Status Word register shows the results of floating point operations. The meaning of each bit is shown in Table 14.4.

Figure 14.2 shows a pictorial representation of the x87 floating point registers. The absolute locations are named  $fpr0, fpr1, \dots, fpr7$  in this figure. The floating point registers are accessed by program instructions as a stack with  $st(0)$  being the register at the top of the stack. It “grows” from higher number registers to lower. The **TOP** field (bits 13 – 11) in the FPU Status Word holds the (absolute) register number that is

bit number	mnemonic	meaning
0	IE	invalid operation
1	DE	denormalized operation
2	ZE	zero divide
3	OE	overflow
4	UE	underflow
5	PE	precision
6	SF	stack fault
7	ES	error summary status
8	C0	condition code 0
9	C1	condition code 1
10	C2	condition code 2
11 - 13	TOP	top of stack
14	C3	condition code 3
15	B	FPU busy

Table 14.4: x87 Status Word.

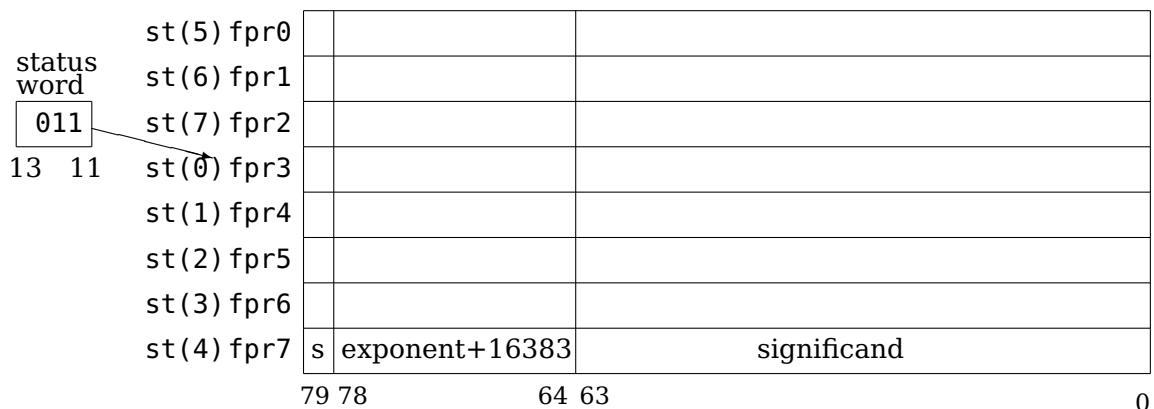


Figure 14.2: x87 floating point register stack. The  $fpr_i$  represent the absolute locations. The  $st(j)$  are the stack names, which are used by the instructions. In this example the top of the stack is at  $fpr3$ , as shown in bits 13 - 11 of the x87 status register.

currently the top of the stack. If the stack is full, i.e.,  $fpr0$  is the top of the stack, a push causes the  $TOP$  field to roll over, and the next item goes into register  $fpr7$ . (The value that was in  $fpr7$  is lost.)

The instructions that read data from memory automatically push the value onto the top of the register stack. Arithmetic instructions are provided that operate on the value(s) on the top of the stack. For example, the `faddp` instruction adds the two values on the top of the stack and leaves their sum on the top. The stack has one less value on it. The original two values are gone.

Many floating point instructions allow the programmer to access any of the floating point registers,  $\%st(i)$ , where  $i = 0 \dots 7$ , relative to the top of the stack. Fortunately, the programmer does not need to keep track of where the top is. When using this format,  $\%st(i)$  refers to the  $i$ th register from the top of the stack. For example, if  $fpr3$  is the current top of the stack, the instruction

`fadd %st(2), %st(0)`

will add the value in the `fpr5` register to the value in the `fpr3` register, leaving the result in the `fpr3` register.

Table 14.5 provides some examples of the floating point instruction set. Notice that the instructions that deal only with the floating point register stack do not use the size suffix letter, *s*. To avoid ambiguity the gnu assembler requires a single letter suffix on

<b>mnemonic</b>	<b>source</b>	<b>destination</b>	<b>meaning</b>
<code>fadds</code>	<i>memfloat</i>		add <i>memfloat</i> to <code>st(0)</code>
<code>faddp</code>			add <code>st(0)</code> to <code>st(1)</code> and pop register stack
<code>fchs</code>			change sign of <code>st(0)</code>
<code>fcoms</code>	<i>memfloat</i>		compare <code>st(0)</code> with <i>memfloat</i>
<code>fcomp</code>			compare <code>st(0)</code> with <code>st(1)</code> and pop register stack
<code>fcos</code>			replace <code>st(0)</code> with its cosine
<code>fdivs</code>	<i>memfloat</i>		divide <code>st(0)</code> by <i>memfloat</i>
<code>fdivp</code>			divide <code>st(0)</code> by <code>st(1)</code> , store result in <code>st(1)</code> , and pop register stack
<code>flds</code>	<i>memint</i>		convert integer at <i>memint</i> to 80-bit float and push onto register stack
<code>fists</code>		<i>memint</i>	convert 80-bit float at <code>st(0)</code> to int and store at <i>memint</i>
<code>flds</code>	<i>memint</i>		convert float at <i>memint</i> to 80-bit float and push onto register stack
<code>fmuls</code>	<i>memfloat</i>		multiply <code>st(0)</code> by <i>memfloat</i>
<code>fmulp</code>			multiply <code>st(0)</code> by <code>st(1)</code> , store result in <code>st(1)</code> , and pop register stack
<code>fsin</code>			replace <code>st(0)</code> with its sine
<code>fsqrt</code>			replace <code>st(0)</code> with its square root
<code>fsts</code>		<i>memint</i>	convert 80-bit float at <code>st(0)</code> to <i>s</i> size float and store at <i>memint</i>
<code>fsubs</code>	<i>memfloat</i>		subtract <i>memfloat</i> from <code>st(0)</code>
<code>fsubp</code>			subtract <code>st(0)</code> from <code>st(1)</code> and pop register stack

*s* = s, l, t

Table 14.5: A sampling of x87 floating point instructions. Size characters are: *s* = 32-bit, *l* = 64-bit, *t* = 80-bit.

the floating point instructions that access memory. The suffixes are:

- 's' for single precision - 32-bit
- 'l' for long (or double) precision - 64-bit
- 't' for ten-byte - 80-bit

Most of the floating point instructions have several variants. See [2] – [6] and [14] – [18] for details. In general,

- Data cannot be moved directly between the integer and floating point registers. Only data stored in memory or another floating point register can be pushed onto the floating point register stack.
- `st(0)` is always involved when performing floating point arithmetic.
- Many floating point instructions have a pop variant. The mnemonic includes a 'p' after the basic mnemonic, immediately before the size character. For example,

```
fistl    someplace(%ebp)
```

converts the 80-bit floating point number in `st(0)` to a 32-bit integer and stores it at the specified memory location. Using the `pop` variant,

```
fistpl   someplace(%ebp)
```

does the same thing but also pops one from the floating point register stack.

Compiling the fraction conversion program of Listing 14.2 in 32-bit mode shows (Listing 14.4) that the compiler uses the x87 floating-point instructions. This ensures backward compatibility since the x86-32 architecture does not need to include SSE instructions.

---

```

1      .file  "frac2float.c"
2      .section        .rodata
3 .LC0:
4      .string "Enter two integers: "
5 .LC1:
6      .string "%i %i"
7 .LC2:
8      .string "%i / %i = %lf\n"
9      .text
10     .globl  main
11     .type   main, @function
12 main:
13     pushl  %ebp
14     movl  %esp, %ebp
15     andl $-16, %esp
16     subl $64, %esp
17     movl  $.LC0, (%esp)
18     call  printf
19     leal  52(%esp), %eax      # address of x
20     movl  %eax, 8(%esp)
21     leal  48(%esp), %eax      # address of y
22     movl  %eax, 4(%esp)
23     movl  $.LC1, (%esp)
24     call  __isoc99_scanf
25     movl  48(%esp), %eax      # load y
26     movl  %eax, 44(%esp)
27     fildl 44(%esp)           # needs to be in memory to
28     movl  52(%esp), %eax      # convert to 80-bit float and push
29     movl  %eax, 44(%esp)      # load x
30     fildl 44(%esp)
31     fdivrp %st, %st(1)        # convert to 80-bit float and push
32     fstpl 56(%esp)           # (long double)x / (long double) y
33     movl  52(%esp), %edx
34     movl  48(%esp), %eax
35     fldl  56(%esp)           # store z
36     fstpl 12(%esp)           # load x
37     movl  %edx, 8(%esp)
38     movl  %eax, 4(%esp)
39     movl  $.LC2, (%esp)        # load y
40     call  printf

```

```

41      movl    $0, %eax
42      leave
43      ret
44      .size   main, .-main
45      .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
46      .section .note.GNU-stack,"",@progbits

```

Listing 14.4: Converting a fraction to a float (gcc assembly language, 32-bit).

We add comments to lines 19 and 21 to show where the *x* and *y* variables are located in the stack frame.

```

19     leal    52(%esp), %eax    # address of x
20     movl    %eax, 8(%esp)
21     leal    48(%esp), %eax    # address of y
22     movl    %eax, 4(%esp)
23     movl    $.LC1, (%esp)
24     call    __isoc99_scanf

```

Rather than actually push the arguments onto the stack, enough space was allocated on the stack (line 15) to directly store the values in the location where they would be if they had been pushed there. This is more efficient than pushing each argument.

Casting an *int* to a *float* requires a conversion in the storage format. This conversion is done by the x87 FPU as an integer is pushed onto the floating point register stack using the *fildl* instruction. This conversion can only be done to an integer that is stored in memory. The compiler uses a location on the call stack to temporarily store each integer so it can be converted:

```

25     movl    48(%esp), %eax    # load y
26     movl    %eax, 44(%esp)    # needs to be in memory to
27     fildl  44(%esp)          # convert to 80-bit float and push
28     movl    52(%esp), %eax    # load x
29     movl    %eax, 44(%esp)    # needs to be in memory to
30     fildl  44(%esp)          # convert to 80-bit float and push

```

Each of the *fildl* instructions on lines 27 and 30 converts a 32-bit integer to an 80-bit float and pushes the float onto the x87 register stack. At this point the floating-point equivalent of *x* is at the top of the stack, and the floating-point equivalent of *y* is immediately below it. Then the floating-point division instruction:

```

31     fdivrp %st, %st(1)      # (long double)x / (long double) y

```

divides the number at *st(0)* (the *(0)* can be omitted) by the number at *st(1)* and pops the x87 register stack so that the result is now at the top of the stack.

Finally, the *fstpl* instruction is used to pop the value off the top of the x87 register stack and store it in memory — at its proper location on the call stack. The “*l*” suffix indicates that 64 bits of memory should be used for storing the floating-point value. So the 80-bit value on the top of the x87 register stack is rounded to 64 bits as it is stored in memory. The other three arguments are also stored on the call stack.

```

32     fstpl  56(%esp)        # store z
33     movl    52(%esp), %edx  # load x
34     movl    48(%esp), %eax  # load y
35     fildl  56(%esp)        # load z
36     fstpl  12(%esp)        # put z on call stack
37     movl    %edx, 8(%esp)    # put x on call stack
38     movl    %eax, 4(%esp)    # put y on call stack

```

Note that the 32-bit version of `printf` does not receive arguments in registers, so `eax` is not used.

### 14.5.3 3DNow! Floating Point

The 3DNow! instructions use the low-order 64 bits in the same physical registers as the x87. These 64-bit portions are named `mmx0`, `mmx1`, ..., `mmx7`. They are used as fixed register, not in a stack configuration. Execution of a 3DNow! instruction changes the `TOP` field (bits 13 – 11) in the MXCSR status register, so the top of stack is lost for any subsequent x87 instructions. The bottom line is that x87 and 3DNow! instructions cannot be used simultaneously.

Another limitation of the 3DNow! instructions set is that it only handles 32-bit floating point.

These limitations of the 3DNow! instruction set make it essentially obsolete in the x86-64 architecture, so it will not be discussed further in this book.

## 14.6 Comments About Numerical Accuracy

Beginning programmers often see floating point arithmetic as more accurate than integer. It is true that even adding two very large integers can cause overflow. Multiplication makes it even more likely that the result will be very large and, thus, overflow. And when used with two integers, the `/` operator in C/C++ causes the fractional part to be lost. However, as you have seen in this chapter, floating point representations have their own set of inaccuracies.

Arithmetically accurate results require a thorough analysis of your algorithm. Some points to consider:

- Try to scale the data such that integer arithmetic can be used.
- All floating point computations are performed in 80-bit extended format. So there is no processing speed improvement from using `floats` instead of `doubles`.
- Try to arrange the order of computations so that similarly sized numbers are added or subtracted.
- Avoid complex arithmetic statements, which may obscure incorrect intermediate results.
- Choose test data that “stresses” your algorithm. For example, 0.00390625 can be stored exactly in eight bits, but 0.1 has no exact binary equivalent.

## 14.7 Instructions Introduced Thus Far

This summary shows the assembly language instructions introduced thus far in the book. The page number where the instruction is explained in more detail, which may be in a subsequent chapter, is also given. This book provides only an introduction to the usage of each instruction. You need to consult the manuals ([2] - [6], [14] - [18]) in order to learn all the possible uses of the instructions.

### 14.7.1 Instructions

*data movement:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
cmovecc	%reg/mem	%reg	conditional move	260
movs	\$imm/%reg	%reg/mem	move	156
movss	\$imm/%reg	%reg/mem	move, sign extend	245
movzs	\$imm/%reg	%reg/mem	move, zero extend	245
popw		%reg/mem	pop from stack	181
pushw	\$imm/%reg/mem		push onto stack	181

*s = b, w, l, q; w = l, q; cc = condition codes*

*arithmetic/logic:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
adds	\$imm/%reg	%reg/mem	add	214
adds	mem	%reg	add	214
ands	\$imm/%reg	%reg/mem	bit-wise and	290
ands	mem	%reg	bit-wise and	290
cmps	\$imm/%reg	%reg/mem	compare	237
cmps	mem	%reg	compare	237
decs	%reg/mem		decrement	249
divs	%reg/mem		unsigned divide	315
idivs	%reg/mem		signed divide	317
imuls	%reg/mem		signed multiply	310
incs	%reg/mem		increment	248
leaw	mem	%reg	load effective address	191
muls	%reg/mem		unsigned multiply	309
negs	%reg/mem		negate	322
ors	\$imm/%reg	%reg/mem	bit-wise inclusive or	290
ors	mem	%reg	bit-wise inclusive or	290
sals	\$imm/%cl	%reg/mem	shift arithmetic left	302
sars	\$imm/%cl	%reg/mem	shift arithmetic right	301
shls	\$imm/%cl	%reg/mem	shift left	302
shrs	\$imm/%cl	%reg/mem	shift right	301
subs	\$imm/%reg	%reg/mem	subtract	215
subs	mem	%reg	subtract	215
tests	\$imm/%reg	%reg/mem	test bits	238
tests	mem	%reg	test bits	238
xors	\$imm/%reg	%reg/mem	bit-wise exclusive or	290
xors	mem	%reg	bit-wise exclusive or	290

*s = b, w, l, q; w = l, q*

*program flow control:*

<i>opcode</i>	<i>location</i>	<i>action</i>	<i>see page:</i>
call	<i>label</i>	call function	173
ja	<i>label</i>	jump above (unsigned)	239
jae	<i>label</i>	jump above/equal (unsigned)	239
jb	<i>label</i>	jump below (unsigned)	239
jbe	<i>label</i>	jump below/equal (unsigned)	239
je	<i>label</i>	jump equal	239
jg	<i>label</i>	jump greater than (signed)	240
jge	<i>label</i>	jump greater than/equal (signed)	240
jl	<i>label</i>	jump less than (signed)	240
jle	<i>label</i>	jump less than/equal (signed)	240
jmp	<i>label</i>	jump	241
jne	<i>label</i>	jump not equal	239
jno	<i>label</i>	jump no overflow	239
jcc	<i>label</i>	jump on condition codes	239
leave		undo stack frame	192
ret		return from function	192
syscall		call kernel function	201

*cc = condition codes*

*x87 floating point:*

<i>opcode</i>	<i>source</i>	<i>destination</i>	<i>action</i>	<i>see page:</i>
fadds	<i>memfloat</i>		add	373
faddp			add/pop	373
fchs			change sign	373
fcoms	<i>memfloat</i>		compare	373
fcomp			compare/pop	373
fcos			cosine	373
fdivs	<i>memfloat</i>		divide	373
fdivp		divide/pop	373	
flds	<i>memint</i>		load integer	373
fists		<i>memint</i>	store integer	373
flds	<i>memint</i>		load floating point	373
fmuls	<i>memfloat</i>		multiply	373
fmulp			multiply/pop	373
fsin			sine	373
fsqrt			square root	373
fstps		<i>memint</i>	floating point store	373
fsubs	<i>memfloat</i>		subtract	373
fsubp			subtract/pop	373

*s = b, w, l, q; w = l, q*

*SSE floating point conversion:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page</b>
cvtsd2si	%xmmreg/mem	%reg	scalar double to signed integer	368
cvtsd2ss	%xmmreg	%xmmreg/%reg	scalar double to single float	368
cvtsi2sd	%reg	%xmmreg/mem	signed integer to scalar double	368
cvtsi2sdq	%reg	%xmmreg/mem	signed integer to scalar double	368
cvtsi2ss	%reg	%xmmreg/mem	signed integer to scalar single	368
cvtsi2ssq	%reg	%xmmreg/mem	signed integer to scalar single	368
cvtss2sd	%xmmreg	%xmmreg/mem	scalar single to scalar double	368
cvtss2si	%xmmreg/mem	%reg	scalar single to signed integer	368
cvtss2siq	%xmmreg/mem	%reg	scalar single to signed integer	368

**14.7.2 Addressing Modes**

<b>register direct:</b>	The data value is located in a CPU register. <i>syntax:</i> name of the register with a "%" prefix. <i>example:</i> movl %eax, %ebx
<b>immediate data:</b>	The data value is located immediately after the instruction. Source operand only. <i>syntax:</i> data value with a "\$" prefix. <i>example:</i> movl \$0xabcd1234, %ebx
<b>base register plus offset:</b>	The data value is located in memory. The address of the memory location is the sum of a value in a base register plus an offset value. <i>syntax:</i> use the name of the register with parentheses around the name and the offset value immediately before the left parenthesis. <i>example:</i> movl \$0xaabbccdd, 12(%eax)
<b>rip-relative:</b>	The target is a memory address determined by adding an offset to the current address in the rip register. <i>syntax:</i> a programmer-defined label <i>example:</i> je somePlace
<b>indexed:</b>	The data value is located in memory. The address of the memory location is the sum of the value in the base_register plus scale times the value in the index_register, plus the offset. <i>syntax:</i> place parentheses around the comma separated list (base_register, index_register, scale) and preface it with the offset. <i>example:</i> movl \$0x6789cdef, -16(%edx, %eax, 4)

**14.8 Exercises**

**14-1** (§14.1) Develop an algorithm for converting decimal fractions to binary. Hint: Multiply both sides of Equation 14.1 by two.

**14-2** (§14.1) Show that two's complement works correctly for fractional values. What is the decimal range of 8-bit, two's complement fractional values? Hint: +0.5 does not exist, but -0.5 does.

**14-3** (§14.3) Copy the following program and run it:

```
1 /*  
2  * exer14_3.c  
3  * Use float for Loop Control Variable?  
4  * Bob Plantz - 18 June 2009  
5 */  
6  
7 #include <stdio.h>  
8  
9 int main()  
10 {  
11     float number;  
12     int counter = 20;  
13  
14     number = 0.5;  
15     while ((number != 0.0) && (counter > 0))  
16     {  
17         printf("number = %.10f and counter = %i\n", number, counter);  
18  
19         number -= 0.1;  
20         counter -= 1;  
21     }  
22  
23     return 0;  
24 }
```

Listing 14.5: Use float for Loop Control Variable?

Explain the behavior. What happens if you change the decrement of `number` from 0.1 to 0.0625? Explain.

**14-4** (§14.3 §14.4) Copy the following program and run it:

```
1 /*  
2  * exer14_3.c  
3  * Are floats accurate?  
4  * Bob Plantz - 18 June 2009  
5 */  
6  
7 #include <stdio.h>  
8  
9 int main()  
10 {  
11     float fNumber = 2147483646.0;  
12     int iNumber = 2147483646;  
13  
14     printf(" Before adding the float is %f\n", fNumber);  
15     printf("           and the integer is %i\n\n", iNumber);  
16     fNumber += 1.0;  
17     iNumber += 1;  
18     printf("After adding 1 the float is %f\n", fNumber);  
19     printf("           and the integer is %i\n", iNumber);  
20 }
```

```

21     return 0;
22 }
```

Listing 14.6: Are floats accurate?

Explain the behavior. What is the maximum value of fNumber such that adding 1.0 to it works?

**14-5** (§14.4) Convert the following decimal numbers to 32-bit IEEE 754 format by hand:

- |               |               |
|---------------|---------------|
| a) 1.0        | e) -3125.3125 |
| b) -0.1       | f) 0.33       |
| c) 2005.0     | g) -0.67      |
| d) 0.00390625 | h) 3.14       |

**14-6** (§14.4) Convert the following 32-bit IEEE 754 bit patterns to decimal.

- |              |              |
|--------------|--------------|
| a) 4000 0000 | e) 42c8 1000 |
| b) bf80 0000 | f) 3f99 999a |
| c) 3d80 0000 | g) 42f6 e666 |
| d) c180 4000 | h) c259 48b4 |

**14-7** (§14.4) Show that half the floats (in 32-bit IEEE 754 format) are between -2.0 and +2.0.

**14-8** (§14.5) The following C program

---

```

1 /* 
2  * casting.c
3  * Casts two integers to floats and adds them.
4  * Bob Plantz - 18 June 2009
5 */
6
7 #include <stdio.h>
8
9 int main()
10 {
11     int x;
12     double y, z;
13
14     printf("Enter an integer: ");
15     scanf("%i", &x);
16     y = 1.23;
17     z = (double)x + y;
18     printf("%i + %lf = %lf\n", x, y, z);
19
20     return 0;
21 }
```

---

Listing 14.7: Casting integer to float in C.

was compiled with the -S option to produce

---

```

1      .file    "casting.c"
2      .section     .rodata
3 .LC0:   .string  "Enter an integer: "
4 .LC1:   .string  "%i"
5 .LC3:   .string  "%i + %lf = %lf\n"
6
7
8      .text
9
10     .globl   main
11     .type    main, @function
12 main:
13     pushq   %rbp
14     movq    %rsp, %rbp
15     subq    $48, %rsp
16     movl    $.LC0, %edi
17     movl    $0, %eax
18     call    printf
19     leaq    -20(%rbp), %rax
20     movq    %rax, %rsi
21     movl    $.LC1, %edi
22     movl    $0, %eax
23     call    __isoc99_scanf
24     movabsq $4608218246714312622, %rax
25     movq    %rax, -16(%rbp)
26     movl    -20(%rbp), %eax
27     cvtsi2sd    %eax, %xmm0
28     addsd   -16(%rbp), %xmm0
29     movsd   %xmm0, -8(%rbp)
30     movl    -20(%rbp), %ecx
31     movq    -8(%rbp), %rdx
32     movq    -16(%rbp), %rax
33     movq    %rdx, -40(%rbp)
34     movsd   -40(%rbp), %xmm1
35     movq    %rax, -40(%rbp)
36     movsd   -40(%rbp), %xmm0
37     movl    %ecx, %esi
38     movl    $.LC3, %edi
39     movl    $2, %eax
40     call    printf
41     movl    $0, %eax
42     leave
43     ret
44     .size   main, .-main
45     .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
46     .section     .note.GNU-stack,"",@progbits

```

---

Listing 14.8: Casting integer to float in assembly language.

Identify the assembly language sequence that performs the C sequence

```

15     y = 1.23;
16     z = (double)x + y;

```

and describe what occurs.

# Chapter 15

## Interrupts and Exceptions

Thus far in this book, all programs have been executed under the Linux operating system. An operating system (OS) can be viewed as a set of programs that provide services to application programs. These services allow the application programs to use the hardware, but only under the auspices of the OS.

Linux allows multiple programs to be executing concurrently, and each of the programs is accessing the hardware resources of the computer. One of the jobs of the OS is to manage the hardware resources in such a way that the programs do not interfere with one another. In this chapter we introduce the CPU features that enable Linux to carry out this management task.

The `read` system call is a good example of a program using the services of the OS. It requests input from the keyboard. The OS handles all input from the keyboard, so the `read` function must first request keyboard input from the OS. One of the reasons this request must be funneled through the OS is that other programs may also be requesting input from the keyboard, and the OS needs to ensure that each program gets the keyboard input intended for it.

Once the request for input has been made, it would be very inefficient for the OS to wait until a user strikes a key. So the OS allows another program to use the CPU, and the keyboard notifies the OS when a key has been struck. To avoid losing a character, this notification interrupts the CPU so that the OS can read the character from the keyboard.

Another example comes from something you probably did not intend to do. Unless you are a perfect programmer, you have probably seen a “segmentation fault.” This can occur when your program attempts to access memory that has not been allocated for your program. I have gotten this error (yes, I still make programming mistakes!) when I have made a mistake using the stack, or when I dereference a register that contains a bad address.

We can summarize these three types of events:

- a *software interrupt* can be used to request a service from the OS.
- most I/O devices can generate a *hardware interrupt* when they are ready to transfer data.
- certain conditions within the CPU (typically caused by our programming errors) generate *exceptions*.

In response to any of these events, the CPU performs an operation that is very similar to the `call` instruction. The value in the `rip` register is pushed onto the stack, and another address is placed in the `rip` register. The net effect is that a function is called, just as in the `call` instruction, but the address of the called function is specified in a

different way, and additional information is pushed onto the stack. Before describing the differences, we discuss what ought to occur in order for the OS to deal with each of these events.

## 15.1 Hardware Interrupts

Keyboard input is a good place to start the discussion. It is impossible to know exactly when someone will strike a key on the keyboard, nor how soon the next key will be struck. For example, if a key is struck in the middle of executing the first of the following two instructions

```
cmpb    $0, (%ebx)
je      allDone
```

in order to avoid losing the keystroke, we would like to read the character immediately after the `cmpb` instruction is executed but before the CPU starts working on the `je` instruction.

The function that reads the character from the keyboard is called an *interrupt handler* or simply *handler*. Handlers are part of the OS. In Linux they can either be built into the kernel or loaded as separate modules as needed.

The timing — between the two instructions — means that the CPU will acknowledge an interrupt only between instruction execution cycles. Just before executing the `je` instruction the `rip` register has the address of the instruction, and it is that address that gets pushed onto the stack. That is, since calling a handler occurs automatically and does not involve fetching an instruction, the current value of the `rip` pushed onto the stack is the correct return address from the handler.

There is another important issue. It is almost certain that the `rflags` register will be changed by the handler that gets called. When program control returns to the `je` instruction (which is supposed to depend on the state of the `rflags` register as a result of executing the `cmpb` instruction), there is little chance that the program will do what the programmer intended. Thus we conclude that in addition to saving the `rip` register,

- an interrupt causes the CPU to save the `rflags` register on the stack.

The next issue is the question of how the CPU knows the address of the appropriate handler to call. In the `call` instruction, the address of the function to call is specified as an operand to the instruction. For example,

```
call    toUpperCase
```

Since the keyboard has no knowledge of the software, there must be some other mechanism for specifying the address of the handler to call. The answer to this problem is that addresses of interrupt handlers are stored in an Interrupt Descriptor Table (IDT). Each possible interrupt in the system is associated with a unique entry in the IDT.

The IDT table entries are data structures (128 bits in 64-bit mode, 64 bits in 32-bit mode) called *gate descriptors*. In addition to the handler address, they contain information that the CPU uses to help protect the integrity of the OS.

After it has completed execution of the current instruction, the following actions must occur when there is an interrupt from a device external to the CPU:

- A copy of the `rflags` register must be saved.
- The address in the `rip` register must be saved so that the CPU can return to the current program after it has handled the interrupting device.
- The address of the handler associated with this interrupt must be placed in the `rip` register.

## 15.2 Exceptions

We next consider exceptions. These are typically the result of a number that the CPU cannot deal with. Examples are

- division by zero
- an invalid instruction
- an invalid address

In a perfect world, the application software would include all the checks that would prevent the occurrence of many of these errors. The reality is that no program is perfect, so some of these errors will occur.

When they do occur, it is the responsibility of the OS to take an appropriate action. The currently executing instruction may have caused the exception to occur. So the CPU often reacts to an exception in the midst of a normal instruction execution cycle. The actions that the CPU must take in response to an exception are essentially the same as those for an interrupt:

- A copy of the `rflags` register must be saved.
- The address in the `rip` register must be saved. Depending on the nature of the exception, the handler may or may not return to the current program after it has handled the exception.
- The address of the handler associated with this exception must be placed in the `rip` register.

Not all exceptions are due to actual program errors. For example, when a program references an address in another part of the program that has not yet been loaded into memory, it causes a page fault exception. The OS must provide a handler that loads the appropriate part of the program from the disk into memory, then continues with normal program execution.

## 15.3 Software Interrupts

The usefulness of the interrupt/exception handling mechanism for requesting OS services is not apparent until we discuss privilege levels. As mentioned above, one of the jobs of the OS is to keep concurrently executing programs from interfering with one another. It uses the privilege level mechanism in the CPU to do this.

At any given time, the CPU is running in one of four possible *privilege levels*. The levels, from most privileged to least, are:

- 0 Provides direct access to all hardware resources. Restricted to the lowest-level operating system functions, e.g., BIOS, memory management.
- 1 Somewhat restricted access to hardware resources. Might be used by library routines and software that controls I/O devices.
- 2 More restricted access to hardware resources. Might be used by library routines and software that controls I/O devices.
- 3 No direct access to hardware resources. Applications programs run at this level.

The OS needs to have direct access to all the hardware, so it executes at privilege level 0. Application programs should be limited, so they execute at privilege level 3. The CPU includes a mechanism for recognizing the privilege level of the memory associated

with each program. A program can access memory at a lower privilege level, but not at a higher level. Thus, an application program (running at level 3) cannot access memory that belongs to the OS.

Gate descriptors include privilege level information in addition to the handler address. The CPU's interrupt/exception mechanism automatically switches to this privilege level when it calls the handler function. Thus, for example, the keyboard might interrupt during the execution of an application program running at privilege level 3, but its handler function would execute at privilege level 0.

The software interrupt allows an application program to use OS services while still allowing the OS to control this access. The instruction is

```
int $n
```

where n specifies the  $n^{th}$  entry in the IDT table.

Older versions of the Linux kernel used

```
int $0x80 # Should be avoided in 64-bit mode.
```

to make system calls. The code corresponding to the desired action is loaded into eax and the arguments are loaded into the proper registers before the system call is executed. The recommended technique for making system calls is discussed in Section 15.6 on page 388.

## 15.4 CPU Response to an Interrupt or Exception

Each entry in the IDT is called a *vector*. The CPU is hardwired to associate vectors 0 - 31 with specific exceptions. For example, vector number 0 represents a divide-by-zero exception. Vector number 14 is a page fault exception.

Vectors 32 - 255 can be assigned to interrupts, both external and the int \$n instruction. These assignments are determined by the OS programmers.

During OS initialization, the address of a handler function is stored in the gate descriptor corresponding to the vector number it is designed to handle. Other information in the gate descriptor causes the CPU to switch to a higher (numerically lower) privilege level, so the handler function has appropriate access to the hardware.

Whenever an interrupt or exception occurs the CPU executes an *exception processing cycle*, which consists of the following actions:

1. Push the rflags register onto the stack.
2. Push the rip register onto the stack.
3. Determine the address of the corresponding gate descriptor in the IDT table.
4. Load the handler address from the gate descriptor into the rip register.

The CPU continues with a normal instruction processing cycle — fetch the instruction at the address in rip, etc. Thus, control will transfer to the handler function.

Depending upon the nature of an exception and what actually caused it, CPU execution may or may not be returned to the program that was executing when the exception occurred.

## 15.5 Return from Interrupt/Exception

There is one more part of this puzzle. Since the `ret` instruction simply pops the value at the top of the stack into the `rip` register, it will not work for the OS's handler function. The CPU has another instruction

```
iret
```

that correctly pops everything off the stack into the `rip` and `rflags` registers and restores the privilege level to where it was before the handler function was invoked. (The privilege level information was also stored on the stack.)

## 15.6 The `syscall` and `sysret` Instructions

Using a software interrupt to invoke one of the services provided by the OS is somewhat of an overkill. The x86-64 architecture includes another instruction that causes the CPU to change priority levels but does not use the stack nor goes through the IDT table, thus saving execution time. The instruction is

```
syscall
```

We first introduced it in Section 8.6 (page 201) to perform I/O.

The `syscall` instruction causes the CPU to

1. Move the low-order 32 bits of the `rflags` register to the `r11` register.
2. Move the address in the `rip` register to the `rcx` register.
3. Load the address from the `LSTAR` register into the `rip` register. The `LSTAR` register is a Model-Specific Register; see Table 6.2 on page 132.
4. Change the privilege level to 0.

Now the CPU has been switched to privilege level 0, and the OS has control and can enforce orderly use of the hardware.

The program in Listing 15.1 illustrates the use of `syscall` to do system calls without using the C libraries. See Exercise 15-1 for using `syscall` within the C runtime environment.

---

```

1 # myCat.s
2 # Writes a file to standard out
3 # Does not use C libraries
4 # Bob Plantz -- 18 June 2009
5
6 # Useful constants
7     .equ    STDIN,0
8     .equ    STDOUT,1
9 # from asm/unistd_64.h
10    .equ    READ,0
11    .equ    WRITE,1
12    .equ    OPEN,2
13    .equ    CLOSE,3
14    .equ    EXIT,60
15 # from bits/fcntl.h

```

```

16      .equ    _0_RDONLY, 0
17      .equ    _0_WRONLY, 1
18      .equ    _0_RDWR, 3
19 # Stack frame
20      .equ    aLetter, -16
21      .equ    fd, -8
22      .equ    localSize, -16
23      .equ    fileName, 24
24 # Code
25          .text           # switch to text segment
26          .globl __start
27          .type __start, @function
28 __start:
29      pushq %rbp           # save caller's frame pointer
30      movq %rsp, %rbp       # establish our frame pointer
31      addq $localSize, %rsp # for local variable
32
33      movl $OPEN, %eax       # open the file
34      movq fileName(%rbp), %rdi # the filename
35      movl $0_RDONLY, %esi   # read only
36      syscall
37      movl %eax, fd(%rbp)    # save file descriptor
38
39      movl $READ, %eax
40      movl $1, %edx          # 1 character
41      leaq aLetter(%rbp), %rsi # place to store character
42      movl fd(%rbp), %edi   # standard in
43      syscall               # request kernel service
44
45 writeLoop:
46      cmpl $0, %eax          # any chars?
47      je allDone             # no, must be end of file
48      movl $1, %edx            # yes, 1 character
49      leaq aLetter(%rbp), %rsi # place to store character
50      movl $STDOUT, %edi      # standard out
51      movl $WRITE, %eax
52      syscall                # request kernel service
53
54      movl $READ, %eax
55      movl $1, %edx
56      leaq aLetter(%rbp), %rsi
57      movl fd(%rbp), %edi
58      syscall
59      jmp writeLoop          # check the char
60 allDone:
61      movl $CLOSE, %eax        # close the file
62      movl fd(%rbp), %edi      # file descriptor
63      syscall                # request kernel service
64      movq %rbp, %rsp          # delete local variables
65
66      popq %rbp
67      movl $EXIT, %eax         # restore caller's frame pointer
                                # end this process

```

**syscall**

Listing 15.1: Using `syscall` to cat a file. Use “`ld -e __start -o myCat myCat.o`” after assembling this file.

In Section 8.1 (page 171) we saw how to call the `write` system call function to write characters to standard out (the screen). `write` and the other system call functions are simply C wrappers that load the proper code in `eax` and the arguments into the appropriate registers.

Several system call codes are shown in Table 15.1. For additional system call codes

<b>function</b>	<b>eax</b>	<b>rdi</b>	<b>rsi</b>	<b>rdx</b>	<b>returns</b>
read	0	file descriptor	pointer to storage area	number of bytes to read	number of bytes read
	0				
write	1	file descriptor	pointer to first byte	number of bytes to write	number of bytes written
	1				
open	2	pointer to filename	flags	mode	file descriptor
open	2				
close	3	file descriptor			
exit	60				

Table 15.1: Some system call codes for the `syscall` instruction.

see the `unistd_64.h` file on your system. The arguments for each system call are given in the `man` page for the corresponding C version. For example,

```
bob@bob-desktop:~$ man 2 write
```

describes the `write` system call.

There is a complementary instruction, `sysret`, which the OS executes in order to return from a system call:

**sysret**

The `sysret` instruction causes the CPU to

1. Move the low-order 32 bits of the `rl1` register to the `rflags` register.
2. Move the value in the `rcx` register to the `rip` register.
3. Change the privilege level to 3. (We omit the details of how this is done.)

## 15.7 Summary

We summarize the differences between a `call` instruction and an interrupt/exception. The similarities are

- the address in the `rip` is pushed onto the stack, thus providing a way for the CPU to return to the normal flow of the application program (if appropriate), and
- the address of the function to be called is placed in the `rip`.

The additional features of the interrupt/exception are

- the value in the `rflags` register is also pushed onto the stack,
- the address of the called function is stored in the IDT table instead of being specified by the programmer, and
- the privilege level of the called function can be changed (and it usually is).

## 15.8 Instructions Introduced Thus Far

This summary shows the assembly language instructions introduced thus far in the book. The page number where the instruction is explained in more detail, which may be in a subsequent chapter, is also given. This book provides only an introduction to the usage of each instruction. You need to consult the manuals ([2] - [6], [14] - [18]) in order to learn all the possible uses of the instructions.

### 15.8.1 Instructions

*data movement:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
<code>cmovecc</code>	<code>%reg/mem</code>	<code>%reg</code>	conditional move	260
<code>movs</code>	<code>\$imm/%reg</code>	<code>%reg/mem</code>	move	156
<code>movss</code>	<code>\$imm/%reg</code>	<code>%reg/mem</code>	move, sign extend	245
<code>movzs</code>	<code>\$imm/%reg</code>	<code>%reg/mem</code>	move, zero extend	245
<code>popw</code>		<code>%reg/mem</code>	pop from stack	181
<code>pushw</code>	<code>\$imm/%reg/mem</code>		push onto stack	181

$s = b, w, l, q; w = l, q; cc = \text{condition codes}$

*arithmetic/logic:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
adds	\$imm/%reg	%reg/mem	add	214
adds	mem	%reg	add	214
ands	\$imm/%reg	%reg/mem	bit-wise and	290
ands	mem	%reg	bit-wise and	290
cmps	\$imm/%reg	%reg/mem	compare	237
cmps	mem	%reg	compare	237
decs	%reg/mem		decrement	249
divs	%reg/mem		unsigned divide	315
idivs	%reg/mem		signed divide	317
imuls	%reg/mem		signed multiply	310
incs	%reg/mem		increment	248
leaw	mem	%reg	load effective address	191
muls	%reg/mem		unsigned multiply	309
negs	%reg/mem		negate	322
ors	\$imm/%reg	%reg/mem	bit-wise inclusive or	290
ors	mem	%reg	bit-wise inclusive or	290
sals	\$imm/%cl	%reg/mem	shift arithmetic left	302
sars	\$imm/%cl	%reg/mem	shift arithmetic right	301
shls	\$imm/%cl	%reg/mem	shift left	302
shrs	\$imm/%cl	%reg/mem	shift right	301
subs	\$imm/%reg	%reg/mem	subtract	215
subs	mem	%reg	subtract	215
tests	\$imm/%reg	%reg/mem	test bits	238
tests	mem	%reg	test bits	238
xors	\$imm/%reg	%reg/mem	bit-wise exclusive or	290
xors	mem	%reg	bit-wise exclusive or	290

*s = b, w, l, q; w = l, q*

*program flow control:*

<i>opcode</i>	<i>location</i>	<i>action</i>	<i>see page:</i>
call	<i>label</i>	call function	173
iret		return from kernel function	388
ja	<i>label</i>	jump above (unsigned)	239
jae	<i>label</i>	jump above/equal (unsigned)	239
jb	<i>label</i>	jump below (unsigned)	239
jbe	<i>label</i>	jump below/equal (unsigned)	239
je	<i>label</i>	jump equal	239
jg	<i>label</i>	jump greater than (signed)	240
jge	<i>label</i>	jump greater than/equal (signed)	240
jl	<i>label</i>	jump less than (signed)	240
jle	<i>label</i>	jump less than/equal (signed)	240
jmp	<i>label</i>	jump	241
jne	<i>label</i>	jump not equal	239
jno	<i>label</i>	jump no overflow	239
jcc	<i>label</i>	jump on condition codes	239
leave		undo stack frame	192
ret		return from function	192
syscall		call kernel function	201
sysret		return from kernel function	390

*cc = condition codes*

*x87 floating point:*

<i>opcode</i>	<i>source</i>	<i>destination</i>	<i>action</i>	<i>see page:</i>
fadds	<i>memfloat</i>		add	373
faddp			add/pop	373
fchs			change sign	373
fcoms	<i>memfloat</i>		compare	373
fcomp			compare/pop	373
fcos			cosine	373
fdivs	<i>memfloat</i>		divide	373
fdivp		divide/pop	373	
flds	<i>memint</i>		load integer	373
fists		<i>memint</i>	store integer	373
flds	<i>memint</i>		load floating point	373
fmuls	<i>memfloat</i>		multiply	373
fmulp			multiply/pop	373
fsin			sine	373
fsqrt			square root	373
fstps		<i>memint</i>	floating point store	373
fsubs	<i>memfloat</i>		subtract	373
fsubp			subtract/pop	373

*s = b, w, l, q; w = l, q*

*SSE floating point conversion:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page</b>
cvtsd2si	%xmmreg/mem	%reg	scalar double to signed integer	368
cvtsd2ss	%xmmreg	%xmmreg/%reg	scalar double to single float	368
cvtsi2sd	%reg	%xmmreg/mem	signed integer to scalar double	368
cvtsi2sdq	%reg	%xmmreg/mem	signed integer to scalar double	368
cvtsi2ss	%reg	%xmmreg/mem	signed integer to scalar single	368
cvtsi2ssq	%reg	%xmmreg/mem	signed integer to scalar single	368
cvtss2sd	%xmmreg	%xmmreg/mem	scalar single to scalar double	368
cvtss2si	%xmmreg/mem	%reg	scalar single to signed integer	368
cvtss2siq	%xmmreg/mem	%reg	scalar single to signed integer	368

**15.8.2 Addressing Modes**

<b>register direct:</b>	The data value is located in a CPU register. <i>syntax:</i> name of the register with a "%" prefix. <i>example:</i> movl %eax, %ebx
<b>immediate data:</b>	The data value is located immediately after the instruction. Source operand only. <i>syntax:</i> data value with a "\$" prefix. <i>example:</i> movl \$0abcd1234, %ebx
<b>base register plus offset:</b>	The data value is located in memory. The address of the memory location is the sum of a value in a base register plus an offset value. <i>syntax:</i> use the name of the register with parentheses around the name and the offset value immediately before the left parenthesis. <i>example:</i> movl \$0xaabbccdd, 12(%eax)
<b>rip-relative:</b>	The target is a memory address determined by adding an offset to the current address in the rip register. <i>syntax:</i> a programmer-defined label <i>example:</i> je somePlace
<b>indexed:</b>	The data value is located in memory. The address of the memory location is the sum of the value in the base_register plus scale times the value in the index_register, plus the offset. <i>syntax:</i> place parentheses around the comma separated list (base_register, index_register, scale) and preface it with the offset. <i>example:</i> movl \$0x6789cdef, -16(%edx, %eax, 4)

**15.9 Exercises**

**15-1** (§15.6) Modify the program in Figure 15.1 so that it uses the C environment. That is, turn it into a `main` function using the prototype `int main(int argc, char **argv);`. `argc` is the number of space-delimited strings on the command line, including the command to execute the program. `argv` is a pointer to an array of pointers to each of the command line strings.

# Chapter 16

## Input/Output

In this chapter we discuss the I/O subsystem. The I/O subsystem is the means by which the CPU communicates with the outside world. By “outside world” we mean devices other than the CPU and memory.

As you have learned, the CPU executes instructions, and memory provides a place to store data and instructions. Most programs read data from one or more input devices, process the data, then write the results to one or more output devices.

Typical input devices are keyboards and mice. Common output devices are display screens and printers. Although most people do not think of them as such, magnetic disks, CD drives, etc. are considered as I/O devices. It may be a little more obvious that a connection with the internet is also seen as I/O. The reasons will become clearer in this chapter, where we discuss how I/O devices are programmed.

### 16.1 Memory Timing

Since the CPU accesses I/O devices via the same buses as memory (see Figure 1.1, page 3), it might seem that the CPU could access the I/O devices in the same way as memory. That is, it might seem that I/O could be performed by using the `movb` instruction to transfer bytes of data between the CPU and the specific I/O device. This can be done with many devices, but there are other issues that must be taken into account in order to make it work correctly. One of the main issues lies in the timing differences between memory and I/O. Before tackling the I/O timing issues, let us consider memory timing characteristics.

**Aside:** As pointed out in Section 1.2 (page 4), the three-bus description given here shows the logical interaction between the CPU and I/O. Most modern general purpose computers employ several types of buses. The way in which the CPU connects to the various buses is handled by hardware controllers. A programmer generally deals only with the logical view.

Two types of RAM are commonly used in PCs.

- SRAM holds its values as long as power is on. Access times are very fast. It requires more components to do this, so it is more expensive and larger.
- DRAM uses passive components that hold data values for only a few fractions of a second. Thus DRAM includes circuitry that automatically refreshes the data values before the values are completely lost. It is much less expensive than SRAM, but also much slower.

Most of the memory in a PC is DRAM because it is much less expensive and smaller than SRAM. Of course, each instruction must be fetched from memory, so slow memory access would limit CPU speed. This problem is solved by using cache systems made from SRAM.

A cache is a small amount of fast memory placed between the CPU and main memory. When the CPU needs to access a byte in main memory, that byte, together with several surrounding bytes, are copied into the cache memory. There is a high probability that the surrounding bytes will be accessed soon, and the CPU can work with the values in the much faster cache. This is handled by the system hardware. See [28] and [31] for more details.

Modern CPUs include cache memory on the same chip, which can be accessed at CPU speeds. Even small cache systems are very effective in speeding up memory access. For example, the CPU in my desktop system (built in 2005) has 64 KB of Level 1 instruction cache, 64 KB of Level 1 data cache, and 512 KB of Level 2 cache (both instructions and data). In contrast, most of the memory in the system consists of 1 GB of DDR 400 memory.

The important point here is that memory is matched to the CPU by the hardware. Very seldom is memory access speed a programming issue.

**Aside:** There are some cases where knowing how to manipulate memory caches can speed up execution time. The x86 has instructions for working directly with cache. Optimizing cache usage is an advanced topic beyond the scope of this book.

## 16.2 I/O Device Timing

I/O devices are much slower than memory. Consider a common input device, the keyboard. Typing at 120 words per minute is equivalent to 10 characters per second, or 100 milliseconds. between each character. A CPU running at 2 GHz can execute approximately 200 million instructions during that time. And the time intervals between keystrokes are very inconsistent. Many will be much longer than this.

Even a magnetic disk is very slow compared to memory. What if the byte that needs to be read has just passed under the read/write head on a disk that is rotating at 7200 RPM? The system must wait for a full revolution of the disk, which takes 8.33 milliseconds. Again, there is a great deal of variability in the rotational delay between reads from the disk.

In addition to being much slower, I/O devices exhibit much more variance in their timing. Some people type very fast on a keyboard, some very slow. The required byte on a magnetic disk might be just coming up to the read/right head, or it may have just passed. We need a mechanism to determine whether an input device has a byte ready for our program to read, and whether an output device is ready to accept a byte that is sent to it.

## 16.3 Bus Timing

Thus far in this book buses have been shown simply as wires connecting the subsystems. Since more than one device is connected to the same wires, the devices must follow a protocol for deciding which two devices can use the bus at any given time. There are many protocols in use, which fall into one of two types:

**Synchronous** — data transfer is controlled by a clock signal. Typically, a centralized bus controller generates the clock signal, which is sent on a separate control line in the bus.

**Asynchronous** — data transfer is controlled by a “handshaking” exchange between the two devices. Many asynchronous protocols are handled by the devices themselves over the data and address lines in the bus.

Modern computer systems employ both types of buses. A typical PC arrangement is shown in Figure 16.1.

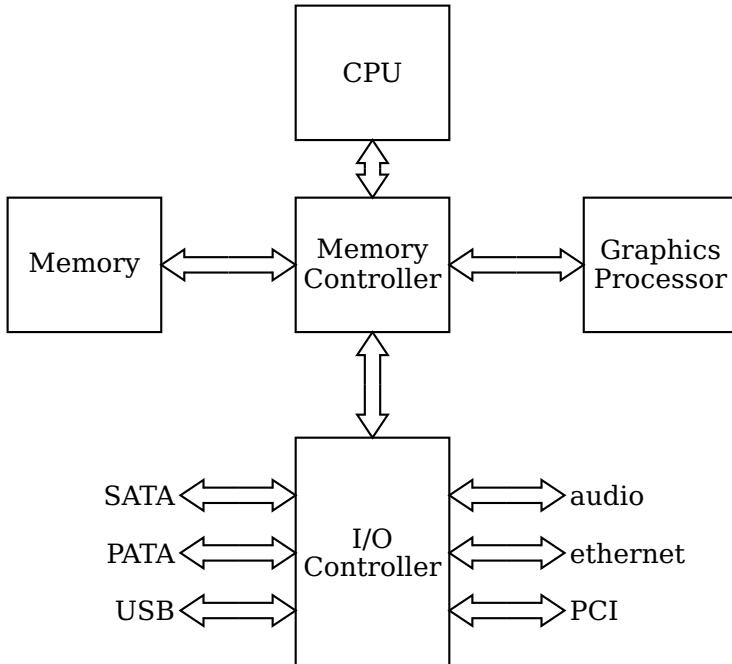


Figure 16.1: Typical bus controllers in a modern PC. The Memory Controller is often called the North Bridge; it provides synchronous communication with main memory and the graphics interface. The I/O Controller is often called the South Bridge; it provides asynchronous communication with the several types of buses that connect to I/O devices.

## 16.4 I/O Interfacing

In addition to a very wide range in their timing, there is an enormous range of I/O devices that are commonly attached to computers, which differ greatly in how they handle data. A mouse provides position information. A monitor displays graphic information. Most computers have speakers connected to them. Ultimately, the CPU must be able to communicate with I/O devices in bit patterns at the speed of the device.

The hardware between the CPU and the actual I/O device consists of two subsystems — the *controller* and the *interface*. The controller is the portion that works directly with the device. For example, a keyboard controller detects which keys are pressed and converts this to a code. It also detects whether a key is pressed or not. A disk controller moves the read/write head to the requested track. It then detects the sector number

and waits until the requested sector comes into position. Some very simple devices do not need a controller.

The interface subsystem provides *registers* that the CPU can read from or write to. An I/O device is programmed through the interface registers. In general, the following types of registers are provided:

- **Transmit** — Allows data to be written to an output device.
- **Receive** — Allows data to be read from an input device.
- **Status** — Provides information about the current state of the device, including the controller.
- **Control** — Allows a program to send commands to the controller and to change its settings.

It is common for one register to provide multiple functionality. For example, there may be one register for transmitting and receiving, its functionality depending on whether the CPU writes to or reads from the register. And it is common for an interface to have more than one register of the same type, especially control registers.

## 16.5 I/O Ports

The CPU communicates with an I/O device through I/O ports. The specific port is specified by a value on the address bus. There are two ways to distinguish an I/O port address from a physical memory address:

- **Isolated I/O**
- **Memory-Mapped I/O**

With isolated I/O, the I/O ports can be numbered from 0x0000 to 0xffff. This address space is separate from the physical memory address space. Instructions are provided for accessing the I/O address space. The distinction between the two addressing spaces is made in the control bus.

One instruction to perform input is:

*ins source, destination*

where *s* denotes the size of the operand:

<i>s</i>	<u>meaning</u>	<u>number of bits</u>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

Intel®  
Syntax

|      *in destination, source*

The *in* instruction moves data from the I/O port specified by the source into the register specified by the destination. The source operand can be either an immediate value, or a value in the dx register. The destination must be al, ax, or eax, consistent with the operand size. For example, the instruction

*inb \$4, %al*

reads I/O port number 4, placing the value in the `al` register.

An instruction to perform output is:

`outs source, destination`

where `s` denotes the size of the operand:

<b><i>s</i></b>	<b>meaning</b>	<b>number of bits</b>
b	byte	8
w	word	16
l	longword	32
q	quadword	64

Intel®  
Syntax

`out destination, source`

The `out` instruction moves data to the I/O port specified by the destination from the register specified by the source. The destination operand can be either an immediate value, or a value in the `dx` register. The source must be `al`, `ax`, or `eax`, consistent with the operand size. For example, the instruction

`outb %al, $6`

writes the value in the `al` register to I/O port number 6.

## 16.6 Programming Issues

One of the primary jobs of an operating system is to handle I/O. The software that does this is called a *device handler*. The operating system coordinates the activities of all the device handlers so that the hardware is utilized in an efficient manner. In Linux, a device handler may either be compiled into the kernel or in a separate module that is loaded into memory only if needed.

Thus, programming I/O devices generally means changing the operating system kernel. This can be done, but it requires considerably more knowledge than is provided in this book. It is possible to give user applications permission to directly access specific I/O devices, but this can produce disastrous results, especially in a multi-user environment.

We will not do any direct I/O programming in this book, but we will look at the general concepts. Listing 16.1 sketches the general algorithms in C. The code was abstracted from some I/O routines that work with a Dual Asynchronous Universal Receiver/Transmitter (DUART) on a single board computer. It is incomplete code and does not run on any known computer, but it illustrates the basic concepts.

This example uses memory-mapped I/O. The program calls three functions:

- `init_io` — Initialize the I/O interface. This includes placing the hardware in an “all clear” state and setting parameters such as speed, etc.
- `charin` — Read one character from the input.
- `charout` — Write one character to the output.

We will examine what each does.

---

```

1 /*  

2 * io_sketch_mm.c  

3 * This code sketches the algorithms to initialize  

4 * a DUART, read one character and echo it using

```

```
5 * isolated I/O.
6 * WARNING: This code does not run on any known
7 * device. It is meant to sketch some
8 * general I/O concepts only.
9 * Bob Plantz - 18 June 2009
10 */
11
12 /* register offsets */
13 #define MR 0x01 /* mode register */
14 #define SR 0x03 /* status register */
15 #define CSR 0x03 /* clock select register */
16 #define CR 0x05 /* command register */
17 #define RR 0x07 /* receiver register */
18 #define TR 0x07 /* transmitter register */
19 #define ACR 0x09 /* auxiliary control register */
20 #define IMR 0x0B /* interrupt mask register */
21
22 /* status bits */
23 #define RxRDY 1 /* receiver ready */
24 #define TxRDY 4 /* transmitter ready */
25
26 /* commands */
27 #define RESETRECEIVER 0x20
28 #define RESETTRANSMIT 0x30
29 #define RESETERROR 0x40
30 #define RESETMODE 0x10
31 #define TIMER 0xF0
32 #define NOPARITY8BITS 0x13
33 #define STOPBIT2 0x0F
34 #define BAUD19200 0xC
35 #define BAUDRATE BAUD19200+(BAUD19200<<4)
36 #define ENABLE 0x05
37 #define NOINTERRUPT 0x00
38
39 void init_io();
40 unsigned char charin();
41 void charout( unsigned char c );
42
43 int main() {
44     unsigned char aCharacter;
45
46     init_io();
47     aCharacter = charin();
48     charout(aCharacter);
49
50     return 0;
51 }
52
53 void init_io() {
54     unsigned char* port = (unsigned char*) 0xff000;
55
56     *(port+CR) = RESETRECEIVER; /* reset receiver */
```

```

57 *(port+CR) = RESETTRANSMIT; /* reset transmitter */
58 *(port+CR) = RESETERROR; /* clear any errors */
59 *(port+CR) = RESETMODE; /* make sure we're using MR1 */
60
61 *(port+ACR) = TIMER; /* baud set 2, crystal divide by 16 */
62 *(port+MR) = NOPARITY8BITS; /* no parity, 8 bits */
63 *(port+MR) = STOPBIT2; /* stop bit length 2.000 */
64 *(port+CSR) = BAUDRATE; /* set baud */
65 *(port+IMR) = 0; /* turn off interrupts */
66 *(port+CR) = ENABLE; /* enable receiver and transmitter */
67 }
68
69 unsigned char charin() {
70     unsigned char* port = (unsigned char*) 0xff000;
71     unsigned char character, status;
72
73     do
74     {
75         status = *(port+SR);
76     } while ((status & RxRDY) != 0);
77     character = *(port+RR);
78     return character;
79 }
80
81 void charout( unsigned char c )
82 {
83     unsigned char* port = (unsigned char*) 0xff000;
84     unsigned char status;
85     do
86     {
87         status = *(port+SR);
88     } while ((status & TxRDY) != 0);
89     *(port+TR) = c;
90 }
```

Listing 16.1: Sketch of basic I/O functions using memory-mapped I/O — C version.

Lines 12 – 37 define symbolic names for values that are used to program the device. Notice that some names have the same value. For example, on lines 17 and 18 the receiver register (RR) and transmitter register (TR) are actually the same register. The CPU receives when it reads from this register and transmits when it writes to it. A similar situation is seen on lines 14 and 15. Reading from register 0x03 provides status information, and the clock selection commands are written to the same register. This illustrates an important point — *I/O interface registers are not simply data storage places like CPU registers*. It would probably be more accurate to call them “interface ports,” but “registers” is the commonly used terminology.

This example uses memory-mapped I/O, so simple assignment statements are used to access the I/O interface registers. The memory addresses 0xff0000 – 0xff020 are associated with I/O registers for this device instead of physical memory. The base address of the device is assigned to a pointer variable on line 54 in the `init_io` function. Then the commands to initialize the device are written to the appropriate registers on lines 56 – 66. It is not important that you completely understand what this function is doing, but the comments should give you a rough idea.

Lines 56 – 59 assign four different values to the same location:

```
*(port+CR) = RESETRECEIVER; /* reset receiver */
*(port+CR) = RESETTRANSMIT; /* reset transmitter */
*(port+CR) = RESETERROR; /* clear any errors */
*(port+CR) = RESETMODE; /* make sure we're using MR1 */
```

If these were assignment to an actual memory location or to a CPU register, only the final statement would be required. But the Command Register is an I/O interface register. And as described above, it really is not a storage register, even on the I/O interface. In fact, these are four different commands that are sent to the Command Register “port” on the I/O interface.

The order in which commands are sent to the I/O interface may also be important. For example, on this particular device, the sequence on lines 62 – 63

```
*(port+MR) = NOPARITY8BITS; /* no parity, 8 bits */
*(port+MR) = STOPBIT2; /* stop bit length 2.000 */
```

must be performed in this order. There are actually two Mode Registers, which are both accessed through the same I/O interface register. The first time the register is accessed, it is connected to Mode Register 1. This access causes the hardware to automatically switch to Mode Register 2 for all subsequent accesses. Now you can understand the reason for sending the “RESETMODE” command to the Command Register on line 59. It’s important to ensure that the first access will be to Mode Register 1.

When compiling I/O functions, it is very important not to use optimization. If you do, the compiler may try coalesce command values into one value. (See Exercise 1.)

The next function is `charin()`. Its job is to read a character from the DUART. In the lab where this code was used, the DUART receiver was connected to a keyboard. The DUART must wait until somebody presses a key on the keyboard, then convert the code for that key to an eight-bit ASCII code representing the character. When the DUART has a character ready to be read from its receiver register, it sets the “receiver ready” bit in its status register to one. The do-while loop on lines 73 – 76 in `charin` show how the code must wait for this event.

When the status indicates that a character is ready, line 77 shows how it is read from the receiver register.

The `charout()` function writes a character to the transmitter. As you might expect, the transmitter was connected to a computer monitor. Although it is clear that keyboard input is very slow, writing on a monitor screen is also slow compared to CPU processing. Thus, we need a similar do-while loop (lines 83 – 88) to wait until the monitor is ready to accept a new character. Once the value provided by the status register shows it is ready, line 89 shows how the character is written to the DUART’s transmitter register.

Listing 16.2 shows the assembly language generated by the `gcc` compiler for the C program in Listing 16.1. Some comments have been added to explain the general concepts.

---

```

1      .file  "io_sketch_mm.c"
2      .text
3      .globl main
4      .type main, @function
5 main:
6      pushq %rbp
7      movq %rsp, %rbp
8      subq $16, %rsp
9      movl $0, %eax
10     call init_io

```

```
11      movl    $0, %eax
12      call    charin
13      movb    %al, -1(%rbp)
14      movzbl -1(%rbp), %eax
15      movl    %eax, %edi
16      call    charout
17      movl    $0, %eax
18      leave
19      ret
20      .size   main, .-main
21      .globl  init_io
22      .type   init_io, @function
23 init_io:
24      pushq   %rbp
25      movq    %rsp, %rbp
26      movq    $1044480, -8(%rbp)
27      movq    -8(%rbp), %rax
28      addq    $5, %rax
29      movb    $32, (%rax)
30      movq    -8(%rbp), %rax
31      addq    $5, %rax
32      movb    $48, (%rax)
33      movq    -8(%rbp), %rax
34      addq    $5, %rax
35      movb    $64, (%rax)
36      movq    -8(%rbp), %rax
37      addq    $5, %rax
38      movb    $16, (%rax)
39      movq    -8(%rbp), %rax
40      addq    $9, %rax
41      movb    $-16, (%rax)
42      movq    -8(%rbp), %rax
43      addq    $1, %rax
44      movb    $19, (%rax)
45      movq    -8(%rbp), %rax
46      addq    $1, %rax
47      movb    $15, (%rax)
48      movq    -8(%rbp), %rax
49      addq    $3, %rax
50      movb    $-52, (%rax)
51      movq    -8(%rbp), %rax
52      addq    $11, %rax
53      movb    $0, (%rax)
54      movq    -8(%rbp), %rax
55      addq    $5, %rax
56      movb    $5, (%rax)
57      popq    %rbp
58      ret
59      .size   init_io, .-init_io
60      .globl  charin
61      .type   charin, @function
62 charin:
```

```

63      pushq  %rbp
64      movq  %rsp, %rbp
65      movq  $1044480, -8(%rbp)
66 .L5:
67      movq  -8(%rbp), %rax
68      movzbl 3(%rax), %eax
69      movb  %al, -10(%rbp)
70      movzbl  -10(%rbp), %eax
71      andl  $1, %eax
72      testl  %eax, %eax
73      jne   .L5
74      movq  -8(%rbp), %rax
75      movzbl 7(%rax), %eax
76      movb  %al, -9(%rbp)
77      movzbl  -9(%rbp), %eax
78      popq  %rbp
79      ret
80      .size  charin, .-charin
81      .globl  charout
82      .type  charout, @function
83 charout:
84      pushq  %rbp
85      movq  %rsp, %rbp
86      movl  %edi, %eax
87      movb  %al, -20(%rbp)
88      movq  $1044480, -8(%rbp)
89 .L8:
90      movq  -8(%rbp), %rax
91      movzbl 3(%rax), %eax
92      movb  %al, -9(%rbp)
93      movzbl  -9(%rbp), %eax
94      andl  $4, %eax
95      testl  %eax, %eax
96      jne   .L8
97      movq  -8(%rbp), %rax
98      leaq  7(%rax), %rdx
99      movzbl  -20(%rbp), %eax
100     movb  %al, (%rdx)
101     popq  %rbp
102     ret
103     .size  charout, .-charout
104     .ident "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
105     .section .note.GNU-stack,"",@progbits

```

Listing 16.2: Memory-mapped I/O in assembly language. Comments have been added to explain the code.

The comments on line 25 – 40 in the `init_io` function describe how values are written to the appropriate memory addresses, which are mapped to I/O registers.

Lines 65 – 73 in the `charin` function make up a loop that waits until the receiver has a character ready to be read. The readiness of the receiver is indicated by bit 2 in the status register. The address of the receiver register is computed on lines 74 – 75, then the character is read from that register on line 75. A similar loop is used on lines 89 –

97 in the `charout` function to wait until the status register shows that the transmitter is ready for another character. When it is ready, the address of the transmitter register is computed on lines 98 – 99, the byte to be sent is loaded into the `eax` register on line 100, and it is written to the transmitter register on line 101.

As we saw in Section 16.5, special instructions are required to access isolated I/O. The Linux kernel source includes macros to use these instructions. The macros are defined in the file `io.h`. Listing 16.3 illustrates the use of these macros to write the same program as in Listing 16.1 if the DUART interface were connected to the isolated I/O system.

```
1  /*
2  * io_sketch_iso.c
3  * This code sketches the algorithms to initialize
4  * a DUART, read one character and echo it using
5  * isolated I/O.
6  * WARNING: This code does not run on any known
7  * device. It is meant to sketch some
8  * general I/O concepts only.
9  * Bob Plantz - 18 June 2009
10 */
11 #include <sys/io.h>
12
13 /* register offsets */
14 #define MR 0x01 /* mode register */
15 #define SR 0x03 /* status register */
16 #define CSR 0x03 /* clock select register */
17 #define CR 0x05 /* command register */
18 #define RR 0x07 /* receiver register */
19 #define TR 0x07 /* transmitter register */
20 #define ACR 0x09 /* auxiliary control register */
21 #define IMR 0x0B /* interrupt mask register */
22
23 /* status bits */
24 #define RxRDY 1 /* receiver ready */
25 #define TxRDY 4 /* transmitter ready */
26
27 /* commands */
28 #define RESETRECEIVER 0x20
29 #define RESETTRANSMIT 0x30
30 #define RESETERROR 0x40
31 #define RESETMODE 0x10
32 #define TIMER 0xF0
33 #define NOPARITY8BITS 0x13
34 #define STOPBIT2 0x0F
35 #define BAUD19200 0xC
36 #define BAUDRATE BAUD19200+(BAUD19200<<4)
37 #define ENABLE 0x05
38 #define NOINTERRUPT 0x00
39 #define NOINTERRUPT 0x00
40
41 void init_io();
42 unsigned char charin();
43 void charout( unsigned char c );
```

```

44
45 int main() {
46     unsigned char aCharacter;
47
48     init_io();
49     aCharacter = charin();
50     charout(aCharacter);
51
52     return 0;
53 }
54
55 void init_io() {
56     outb(CR, RESETRECEIVER);
57     outb(CR, RESETTRANSMIT);
58     outb(CR, RESETERROR);
59     outb(CR, RESETMODE);
60     outb(ACR, TIMER);
61     outb(MR, NOPARITY8BITS);
62     outb(MR, STOPBIT2);
63     outb(CSR, BAUDRATE);
64     outb(IMR, NOINTERRUPT);
65     outb(CR, ENABLE);
66 }
67
68 unsigned char charin() {
69     unsigned char character, status;
70
71     do
72     {
73         status = inb(SR);
74     } while ((status & RxRDY) != 0);
75     character = inb(RR);
76     return character;
77 }
78
79 void charout( unsigned char c )
80 {
81     unsigned char status;
82     do
83     {
84         status = inb(SR);
85     } while ((status & TxRDY) != 0);
86     outb(TR, c);
87 }

```

Listing 16.3: Sketch of basic I/O functions, isolated I/O — C version.

On line 11 we need to include the file containing the macros:

```
#include <sys/io.h>
```

The use of the `outb()` macro can be seen in lines 55 – 64. And on line 72 we see the `inb()` macro being used to read the status register.

The gcc compiler generates assembly language as shown in Listing 16.4

```
1      .file  "io_sketch_iso.c"
2      .text
3      .type  inb, @function
4 inb:
5      pushq  %rbp
6      movq  %rsp, %rbp
7      pushq  %rbx
8      movl  %edi, %eax
9      movw  %ax, -28(%rbp)
10     movzwl -28(%rbp), %edx
11     movw  %dx, -30(%rbp)
12     movzwl -30(%rbp), %edx
13 #APP
14 # 48 "/usr/include/x86_64-linux-gnu/sys/io.h" 1
15     inb %dx,%al
16 # 0 "" 2
17 #NO_APP
18     movl  %eax, %ebx
19     movb  %bl, -9(%rbp)
20     movzbl -9(%rbp), %eax
21     popq  %rbx
22     popq  %rbp
23     ret
24     .size  inb, .-inb
25     .type  outb, @function
26 outb:
27     pushq  %rbp
28     movq  %rsp, %rbp
29     movl  %edi, %edx
30     movl  %esi, %eax
31     movb  %dl, -4(%rbp)
32     movw  %ax, -8(%rbp)
33     movzbl -4(%rbp), %eax
34     movzwl -8(%rbp), %edx
35 #APP
36 # 99 "/usr/include/x86_64-linux-gnu/sys/io.h" 1
37     outb %al,%dx
38 # 0 "" 2
39 #NO_APP
40     popq  %rbp
41     ret
42     .size  outb, .-outb
43     .globl main
44     .type  main, @function
45 main:
46     pushq  %rbp
47     movq  %rsp, %rbp
48     subq  $16, %rsp
49     movl  $0, %eax
50     call  init_io
51     movl  $0, %eax
```

```
52      call    charin
53      movb   %al, -1(%rbp)
54      movzbl -1(%rbp), %eax
55      movl   %eax, %edi
56      call    charout
57      movl   $0, %eax
58      leave
59      ret
60      .size  main, .-main
61      .globl init_io
62      .type  init_io, @function
63 init_io:
64      pushq  %rbp
65      movq   %rsp, %rbp
66      movl   $32, %esi
67      movl   $5, %edi
68      call    outb
69      movl   $48, %esi
70      movl   $5, %edi
71      call    outb
72      movl   $64, %esi
73      movl   $5, %edi
74      call    outb
75      movl   $16, %esi
76      movl   $5, %edi
77      call    outb
78      movl   $240, %esi
79      movl   $9, %edi
80      call    outb
81      movl   $19, %esi
82      movl   $1, %edi
83      call    outb
84      movl   $15, %esi
85      movl   $1, %edi
86      call    outb
87      movl   $204, %esi
88      movl   $3, %edi
89      call    outb
90      movl   $0, %esi
91      movl   $11, %edi
92      call    outb
93      movl   $5, %esi
94      movl   $5, %edi
95      call    outb
96      popq   %rbp
97      ret
98      .size  init_io, .-init_io
99      .globl charin
100     .type  charin, @function
101 charin:
102     pushq  %rbp
103     movq   %rsp, %rbp
```

```

104      subq    $16, %rsp
105 .L8:
106      movl    $3, %edi
107      call    inb
108      movb    %al, -2(%rbp)
109      movzbl -2(%rbp), %eax
110      andl    $1, %eax
111      testl   %eax, %eax
112      jne     .L8
113      movl    $7, %edi
114      call    inb
115      movb    %al, -1(%rbp)
116      movzbl -1(%rbp), %eax
117      leave
118      ret
119      .size   charin, .-charin
120      .globl  charout
121      .type   charout, @function
122 charout:
123      pushq   %rbp
124      movq    %rsp, %rbp
125      subq    $24, %rsp
126      movl    %edi, %eax
127      movb    %al, -20(%rbp)
128 .L11:
129      movl    $3, %edi
130      call    inb
131      movb    %al, -1(%rbp)
132      movzbl -1(%rbp), %eax
133      andl    $4, %eax
134      testl   %eax, %eax
135      jne     .L11
136      movzbl -20(%rbp), %eax
137      movl    %eax, %esi
138      movl    $7, %edi
139      call    outb
140      leave
141      ret
142      .size   charout, .-charout
143      .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
144      .section .note.GNU-stack, "",@progbits

```

Listing 16.4: Isolated I/O in assembly language. Comments have been added to explain the code.

Looking at lines 58 – 73 and lines 95 – 110, we see that the `outb()` and `inb()` macros generate functions. The actual `outb` instruction is used on line 68 and `inb` is used on line 103.

At the points where the macros are called in the C source code, the compiler generates calls to the appropriate function. For example, the C sequence

```

55      outb(CR, RESETRECEIVER);
56      outb(CR, RESETTRANSMIT);

```

generates the assembly language

```

25      movl    $32, %esi
26      movl    $5, %edi
27      call    outb
28      movl    $48, %esi
29      movl    $5, %edi
30      call    outb

```

## 16.7 Interrupt-Driven I/O

Reading the code in Section 16.6, you probably realize that the CPU can waste a lot of time simply waiting for I/O devices. Most I/O interfaces include hardware that can send an interrupt signal to the CPU when they have data ready for input or are able to accept output (see Section 15.1, page 385). While waiting for an I/O device, the operating system will suspend the requesting process and allow another process, perhaps being run by another user, to use the CPU.

The device handler for each I/O device that can interrupt includes a special *interrupt handler* function. The address of each interrupt handler is stored in a table in the operating system. When the requested I/O device is ready for I/O, it sends an interrupt signal to the CPU on the control bus. The device identifies itself to the CPU, and the CPU consults the table to obtain the address of the corresponding interrupt handler. CPU execution control then transfers to the interrupt handler function, which contains code to read from or write to the device as needed. When the interrupt handler function completes its servicing of the I/O device, the last instruction in the function is an `iret` (see Section 15.5 on page 388). This causes CPU execution control to return to the control flow where it was interrupted.

This is a highly simplified description. The operating system must perform a great deal of “bookkeeping” in this transfer of control. For example, before allowing the interrupt handler function to execute, at least any registers that will be used in the function must be saved. And more than one process may be waiting for I/O to complete. The operating system must keep track of which process is waiting for which I/O device and make sure that the process gets or sends the correct input or output.

Many other issues face the device handler programmer. For example, I/O devices are left to run on their own time, so one device may attempt to interrupt while another device’s interrupt handling function is being executed. The programmer must decide whether the interrupt should be allowed or not. In general, it cannot be ignored because this would cause the loss of I/O data. On the other hand, spending too much time handling the second interrupt may cause the first device to lose data.

## 16.8 I/O Instructions

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>see page:</b>
<code>ins</code>	<code>\$imm/%reg</code>	<code>%reg/mem</code>	398
<code>outs</code>	<code>\$imm/%reg</code>	<code>%reg/mem</code>	399

`s = b, w, l, q`

## 16.9 Exercises

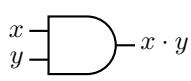
- 16-1** (§16.6) Enter the C program in Listing 16.1. Compile it to the assembly language stage (use the `-S` option) with different levels of optimization. For example, `-O1`, `-O2`. Compare the results with the non-optimized version in Listing 16.2.
- 16-2** (§16.6) Enter the C program in Listing 16.3. Compile it to the assembly language stage (use the `-S` option) with different levels of optimization. For example, `-O1`, `-O2`. Compare the results with the non-optimized version in Listing 16.4.

# Appendix A

## Reference Material

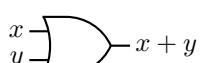
### A.1 Basic Logic Gates

1. AND gate (page 61)



$x$	$y$	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate (page 62)



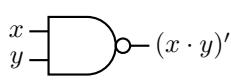
$x$	$y$	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

3. NOT gate (page 62)



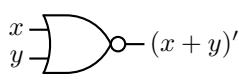
$x$	$x'$
0	0
0	1

4. NAND gate (page 87)



$x$	$y$	$(x \cdot y)'$
0	0	1
0	1	1
1	0	1
1	1	0

5. NOR gate (page 87)



$x$	$y$	$(x + y)'$
0	0	1
0	1	0
1	0	0
1	1	0

## A.2 Register Names

(page 129)

<b>bits 63-</b>	<b>bits 31-</b>	<b>bits 15-</b>	<b>bits 15-</b>	<b>bits 7-</b>
<b>0</b>	<b>0</b>	<b>0</b>	<b>8</b>	<b>0</b>
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	ch	cl	
rdx	edx	dx	dh	dl
rsi	esi	si		sil
rdi	edi	di		dil
rbp	ebp	bp		bpl
rsp	esp	sp		spl
r8	r8d	r8w		r8b
r9	r9d	r9w		r9b
r10	r10d	r10w		r10b
r11	r11d	r11w		r11b
r12	r12d	r12w		r12b
r13	r13d	r13w		r13b
r14	r14d	r14w		r14b
r15	r15d	r15w		r15b

## A.3 Argument Order in Registers

(page 174)

<b>Argument</b>	<b>Register</b>
first	rdi
second	rsi
third	rdx
fourth	rcx
fifth	r8
sixth	r9

## A.4 Register Usage

(page 134)

Register	Special usage	Called function preserves contents
rax	1st function return value.	No
rbx	Optional base pointer.	Yes
rcx	Pass 4th argument to function.	No
rdx	Pass 3rd argument to function; 2nd function return value.	No
rsp	Stack pointer.	Yes
rbp	Optional frame pointer.	Yes
rdi	Pass 1st argument to function.	No
rsi	Pass 2nd argument to function.	No
r8	Pass 5th argument to function.	No
r9	Pass 6th argument to function.	No
r10	Pass function's static chain pointer.	No
r11		No
r12		Yes
r13		Yes
r14		Yes
r15		Yes

## A.5 Assembly Language Instructions Used in This Book

This summary shows the assembly language instructions used in this book. The page number where the instruction is explained in more detail, is also given. This book provides only an introduction to the usage of each instruction. You need to consult the manuals ([2] - [6], [14] - [18]) in order to learn all the possible uses of the instructions.

*data movement:*

opcode	source	destination	action	see page:
cmovecc	%reg/mem	%reg	conditional move	260
movs	\$imm/%reg	%reg/mem	move	156
movsss	\$imm/%reg	%reg/mem	move, sign extend	245
movzss	\$imm/%reg	%reg/mem	move, zero extend	245
popw		%reg/mem	pop from stack	181
pushw	\$imm/%reg/mem		push onto stack	181

s = b, w, l, q; w = l, q; cc = condition codes

*arithmetic/logic:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
adds	\$imm/%reg	%reg/mem	add	214
adds	mem	%reg	add	214
ands	\$imm/%reg	%reg/mem	bit-wise and	290
ands	mem	%reg	bit-wise and	290
cmps	\$imm/%reg	%reg/mem	compare	237
cmps	mem	%reg	compare	237
decs	%reg/mem		decrement	249
divs	%reg/mem		unsigned divide	315
idivs	%reg/mem		signed divide	317
imuls	%reg/mem		signed multiply	310
incs	%reg/mem		increment	248
leaw	mem	%reg	load effective address	191
muls	%reg/mem		unsigned multiply	309
negs	%reg/mem		negate	322
ors	\$imm/%reg	%reg/mem	bit-wise inclusive or	290
ors	mem	%reg	bit-wise inclusive or	290
sals	\$imm/%cl	%reg/mem	shift arithmetic left	302
sars	\$imm/%cl	%reg/mem	shift arithmetic right	301
shls	\$imm/%cl	%reg/mem	shift left	302
shrs	\$imm/%cl	%reg/mem	shift right	301
subs	\$imm/%reg	%reg/mem	subtract	215
subs	mem	%reg	subtract	215
tests	\$imm/%reg	%reg/mem	test bits	238
tests	mem	%reg	test bits	238
xors	\$imm/%reg	%reg/mem	bit-wise exclusive or	290
xors	mem	%reg	bit-wise exclusive or	290

*s = b, w, l, q; w = l, q*

*program flow control:*

<b>opcode</b>	<b>location</b>	<b>action</b>	<b>see page:</b>
call	label	call function	173
iret		return from kernel function	388
ja	label	jump above (unsigned)	239
jae	label	jump above/equal (unsigned)	239
jb	label	jump below (unsigned)	239
jbe	label	jump below/equal (unsigned)	239
je	label	jump equal	239
jg	label	jump greater than (signed)	240
jge	label	jump greater than/equal (signed)	240
jl	label	jump less than (signed)	240
jle	label	jump less than/equal (signed)	240
jmp	label	jump	241
jne	label	jump not equal	239
jno	label	jump no overflow	239
jcc	label	jump on condition codes	239
leave		undo stack frame	192
ret		return from function	192
syscall		call kernel function	201
sysret		return from kernel function	390

*cc = condition codes**SSE floating point conversion:*

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page</b>
cvtsd2si	%xmmreg/mem	%reg	scalar double to signed integer	368
cvtsd2ss	%xmmreg	%xmmreg/%reg	scalar double to single float	368
cvtsi2sd	%reg	%xmmreg/mem	signed integer to scalar double	368
cvtsi2sdq	%reg	%xmmreg/mem	signed integer to scalar double	368
cvtsi2ss	%reg	%xmmreg/mem	signed integer to scalar single	368
cvtsi2ssq	%reg	%xmmreg/mem	signed integer to scalar single	368
cvtss2sd	%xmmreg	%xmmreg/mem	scalar single to scalar double	368
cvtss2si	%xmmreg/mem	%reg	scalar single to signed integer	368
cvtss2siq	%xmmreg/mem	%reg	scalar single to signed integer	368

x87 floating point:

<b>opcode</b>	<b>source</b>	<b>destination</b>	<b>action</b>	<b>see page:</b>
fadds	memfloat		add	373
faddp			add/pop	373
fchs			change sign	373
fcoms	memfloat		compare	373
fcomp			compare/pop	373
fcos			cosine	373
fdivs	memfloat		divide	373
fdivp			divide/pop	373
filds	memint		load integer	373
fists		memint	store integer	373
flds	memint		load floating point	373
fmuls	memfloat		multiply	373
fmulp			multiply/pop	373
fsin			sine	373
fsqrt			square root	373
fstps		memint	floating point store	373
fsubs	memfloat		subtract	373
fsubp			subtract/pop	373

*s = b, w, l, q; w = l, q*

## A.6 Addressing Modes

<b>register direct:</b>	The data value is located in a CPU register. <i>syntax:</i> name of the register with a "%" prefix. <i>example:</i> movl %eax, %ebx
<b>immediate data:</b>	The data value is located immediately after the instruction. Source operand only. <i>syntax:</i> data value with a "\$" prefix. <i>example:</i> movl \$0xabcd1234, %ebx
<b>base register plus offset:</b>	The data value is located in memory. The address of the memory location is the sum of a value in a base register plus an offset value. <i>syntax:</i> use the name of the register with parentheses around the name and the offset value immediately before the left parenthesis. <i>example:</i> movl \$0xaabbccdd, 12(%eax)
<b>rip-relative:</b>	The target is a memory address determined by adding an offset to the current address in the rip register. <i>syntax:</i> a programmer-defined label <i>example:</i> je somePlace
<b>indexed:</b>	The data value is located in memory. The address of the memory location is the sum of the value in the base_register plus scale times the value in the index_register, plus the offset. <i>syntax:</i> place parentheses around the comma separated list (base_register, index_register, scale) and preface it with the offset. <i>example:</i> movl \$0x6789cdef, -16(%edx, %eax, 4)

## Appendix B

# Using GNU make to Build Programs

This discussion covers the fundamental concepts employed in a Makefile. My intent is to show you how to write Makefiles that help you debug your programs. The problem with many discussions of `make` is that they show how to use many of the “features” of the `make` program. Many problems students have when debugging their programs are actually caused by errors in the Makefile that cause `make` to use its default behavior.

For example, if you try to compile the program `myProg` with the command:

```
make myProg
```

`make` will look for its instructions in a file in the current directory named `Makefile` (or `makefile`). Assuming `Makefile` exists, `make` searches the file for a *target* (defined later in this chapter) named `myProg`. If there is no `Makefile`, `make` searches for a file named `myProg.s`, `myProg.c` or `myProg.cc`. If either of the `.s` or `.c` source files exists, `make` issues the command

```
cc myProg.c -o myProg
```

and if the `.cc` source file exists,

```
g++ myProg.cc -o myProg
```

Notice that the compiler is invoked with only the `-o` option. For example, you cannot use `gdb` to debug the program because the `-g` option does not get used. This means that if one of the entries in your `Makefile` is incorrect, the default behavior may cause `make` to compile a source file without the debugging option. It is much easier to avoid these problems if you:

- keep your `Makefiles` very simple, and
- read what `make` writes on the screen very carefully when it executes a `Makefile`.

A `Makefile` consists of a series of entries. Each *entry* in a `Makefile` consists of:

1. One dependency line. The format of a *dependency line* is:

```
target: [prerequisite1] [prerequisite2] ...
```

Prerequisites are names of files or other targets in this `Makefile`. Use spaces as separators between the prerequisites, not tabs.

2. Zero or more Unix command lines. The format of a *command line* is:

- (a) Each line must begin with a tab character (not a group of spaces). Be careful if you write a Makefile with an editor on another platform; some editors automatically replace tabs with spaces.
- (b) The remainder of the line is a Unix command.

There can also be (should be!) *comment lines* that begin with a #.

If you invoke make with no argument:

```
make
```

make starts with the first entry in the Makefile.

Starting with the prerequisites of the first target (the first entry or the argument used in the command), make follows the hierarchical tree of target / prerequisite entries until it gets to the lowest level — either a file or no prerequisites. It works its way back up through this tree. If any entry along the tree has a target that is not at least as recent as all of its prerequisites, the commands for that entry are executed.

A common organization is to have an entry for the program name, with each of the object files that make up the program as prerequisites. The command(s) in this entry link the object files together.

Then there is an entry for each object file, with its source file and any required local header files as prerequisites. The command(s) in each of these entries compile/assemble the source file and produce the corresponding object file.

It is also common to have various utility targets in a Makefile.

These concepts are illustrated in Listing B.1. Notice that there are no prerequisites for the clean target. Thus the command

```
make clean
```

causes make to start with the clean entry. Since there are no prerequisites, the tree ends here, and the target is always “out of date.” Hence, the command for this entry will always be executed, but none of the other entries is executed.

**IMPORTANT:** When using the make program, it will echo each command as it is executed and provide diagnostic error messages on the screen. You must read this display very carefully in order to determine what has taken place.

```

1 # makefile to create myProg
2
3 # link object file to system libraries to create executable
4 myProg: myProg.o
5     gcc -o myProg myProg.o
6
7 # assemble source file to create object file
8 myProg.o: myProg.s
9     as --gstabs myProg.s -o myProg.o
10
11 # remove object files and backup files
12 clean:
13     rm -i *.o *~

```

Listing B.1: An example of a Makefile for an assembly language program with one source file.

The functions in real programs are distributed amongst many files. When changes are made, it is clearly a waste of time to recompile all the functions. With a properly designed Makefile, make only recompiles files that have been changed. (This is a motivation for placing each function in its own file.) Listing B.2 illustrates a Makefile for a program where the main function and one subfunction are written in C and one subfunction is written in assembly language. Notice that header files have been created for both subfunctions to provide prototype statements for the main function, which is written in C. The assembly language source file does not #include its own header file because prototype statements do not apply to assembly language.

---

```

1 # makefile to create biggerProg
2
3 # link object files and system libraries to create executable
4 biggerProg: biggerProg.o sub1.o sub2.o
5     gcc -o biggerProg biggerProg.o sub1.o sub2.o
6
7 # compile/assemble source files to create object files
8 biggerProg.o: biggerProg.c sub1.h sub2.h
9     gcc -g -c biggerProg.c
10
11 sub1.o: sub1.c sub1.h
12     gcc -g -c sub1.c
13
14 sub2.o: sub2.s
15     as --gstabs sub2.s -o sub2.o
16
17 # remove object files and backup files
18 clean:
19     rm -i *.* ~

```

---

Listing B.2: An example of a Makefile for a program with both C and assembly language source files.

As you can see in Listing B.2, there is quite a bit of repetition in a Makefile. Variables provide a good way to reduce the chance of typing errors. Listing B.3 illustrates the use of variables to simplify the Makefile from Listing B.2.

---

```

1 # Makefile for biggerProg
2 # Bob Plantz - 19 June 2009
3
4 # Specify the compiler and assembler options.
5 compflags = -g -c -O1 -Wall
6 asmflags = --gstabs
7
8 # The object files are specific to this program.
9 objects = biggerProg.o sub1.o sub2.o
10
11 biggerProg: $(objects)
12     gcc -o biggerProg $(objects)
13
14 biggerProg.o: biggerProg.c sub1.h sub2.h
15     gcc $(compflags) -o biggerProg.o biggerProg.c
16
17 sub1.o: sub1.c sub1.h

```

---

```

18     gcc $(compflags) -o sub1.o sub1.c
19
20 sub2.o: sub2.s
21     as $(asmflags) -o sub2.o sub2.s
22
23 clean:
24     rm $(objects) biggerProg *~

```

Listing B.3: Makefile variables. Another version of Figure B.2

Executing `make` with the Makefile in Listing B.3 shows that the two C source files are compiled, and the assembly source file is assembled with the proper flags:

```

bob$ make
gcc -g -c -O1 -Wall -o biggerProg.o biggerProg.c
gcc -g -c -O1 -Wall -o sub1.o sub1.c
as --gstabs -o sub2.o sub2.s
gcc -o biggerProg biggerProg.o sub1.o sub2.o
bob$

```

`make` shows each command in your terminal window as it executes it. Reading them is the best way to ensure that your Makefile is written correctly. This display also shows how `make` starts with the commands at the leaves of the tree and works its way back up to the top of the tree.

We can use `gdb` to follow execution of the program:

```

bob$ gdb biggerProg
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/bob/biggerProg/biggerProg...done.
(gdb) li
2  * Demonstrate simple Makefile design
3  * Bob Plantz - 19 June 2009
4  */
5
6 #include <stdio.h>
7 #include "sub1.h"
8 #include "sub2.h"
9
10 int main()
11
(gdb)
12     printf("Starting in main, about to call sub1...\n");
13     sub1();
14     printf("Back in main, about to call sub2...\n");
15     sub2();
16     printf("Back in main.\nProgram ending.\n");

```

```

17
18     return 0;
19
(gdb) run
Starting program: /home/bob/biggerProg/biggerProg
Starting in main, about to call sub1...
In sub1
Back in main, about to call sub2...
In sub2
Back in main.
Program ending.
[Inferior 1 (process 4489) exited normally]
(gdb) q
bob$
```

However, if we do not write a complete Makefile (see Listing B.4)

---

```

1 # Makefile for biggerProg
2 # Bob Plantz - 19 June 2009
3 # WARNING! THIS IS A BAD MAKEFILE!!
4
5 # Specify the compiler and assembler options.
6 compflags = -g -c
7 asmflags = --gstabs
8
9 # The object files are specific to this program.
10 objects = biggerProg.o sub1.o sub2.o
11
12 biggerProg: $(objects)
13         gcc -o biggerProg $(objects)
14
15 clean:
16         rm $(objects) biggerProg *~
```

---

**Listing B.4:** Incomplete Makefile. Several entries are missing, so make invokes its default behavior.

executing make shows that the two C source files are compiled, and the assembly source file is assembled using the make program's own default behavior to give:

```

bob$ make
cc    -c -o biggerProg.o biggerProg.c
cc    -c -o sub1.o sub1.c
as    -o sub2.o sub2.s
gcc -o biggerProg biggerProg.o sub1.o sub2.o
bob$
```

Note that make does not give any error messages even though our Makefile is incomplete. It appears to have created the program correctly. However, when we try to use gdb we see:

```

bob$ gdb biggerProg
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.htm
```

```
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.  
This GDB was configured as "x86_64-linux-gnu".  
For bug reporting instructions, please see:  
<http://bugs.launchpad.net/gdb-linaro/>...  
Reading symbols from /home/bob/biggerProg/biggerProg... (no debugging symbols  
(gdb) li  
No symbol table is loaded. Use the "file" command.  
(gdb) run  
Starting program: /home/bob/biggerProg/biggerProg  
Starting in main, about to call sub1...  
In sub1  
Back in main, about to call sub2...  
In sub2  
Back in main.  
Program ending.  
[Inferior 1 (process 4541) exited normally]  
(gdb) q  
bob$
```

Reading the make messages on the screen shows that it created the program without using all our flags. The important lesson to note here is that an error-free execution of make is not sufficient to guarantee your program was built as you intended. You need to read the screen messages written on the screen when using make.

To learn more about using make see [30].

## Appendix C

# Using the gdb Debugger for Assembly Language

The program in Listing 10.5 uses a while loop to write “Hello World” on the screen one character at a time. A common programming error is to create an “infinite” loop. It would be nice to have a tool that allows us to stop such a program in the middle of the loop so we can observe the state of registers and memory locations. That can help us to determine such things as whether the loop control variable is being changed as we planned.

Fortunately, the gnu program development environment includes a debugger, gdb (see [29]), that allows us to do just that. The gdb debugger allows you to load another program into memory and use gdb commands to control the execution of the other program — the *target program* — and to observe the states of its variables.

There is another, very important, reason for learning how to use gdb. This book describes how registers and memory are controlled by computer instructions. The gdb program is a very valuable learning tool, since it allows you to observe the behavior of each instruction, one step at a time.

gdb has a large number of commands, but the following are the most common ones that will be used in this book:

- `li lineNumber` — lists ten lines of the source code, centered at the specified line number.
- `break sourceFilename:lineNumber` — sets a breakpoint at the specified line in the source file. Control will return to gdb when the line number is encountered.
- `clear lineNumber` — removes the breakpoint at the specified line number.
- `run` — begins execution of a program that has been loaded under control of gdb.
- `cont` — continues execution of a program that has been running.
- `n` — execute current source code statement of a program that has been running; if it’s a call to a function, the entire function is executed.
- `s` — execute current source code statement of a program that has been running; if it’s a call to a function, step into the function.
- `si` — execute current (machine) instruction of a program that has been running; if it’s a call to a function, step into the function.

- `print expression` — evaluate expression and display the value.
- `i r` — info registers — displays the contents of the registers, except floating point and vector.
- `x/nfs memoryAddress` — displays (examine)  $n$  values in memory in format  $f$  of size  $s$ .

Here is a screen shot of how I assembled, linked, and then used gdb to control the execution of the program and observe its behavior. User input is **`boldface`** and the session is annotated in *italics*.

```
bob@ubuntu:~$ as --gstabs -o helloWorld3.o helloWorld3.s
bob@ubuntu:~$ gcc -o helloworld3 helloWorld3.o
bob@ubuntu:~$ gdb helloworld3
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/bob/progs/chap10/helloWorld3...done.
```

*After assembling and linking the program, we start gdb program and load the helloworld program — the one we want to observe — into memory. This leaves me in gdb. The target program is not running.*

```
(gdb) li
1 # helloWorld3.s
2 # "hello world" program using the write() system call
3 # one character at a time.
4 # Bob Plantz - 12 June 2009
5
6 # Useful constants
7     .equ    STDOUT,1
8 # Stack frame
9     .equ    aString,-8
10    .equ    localSize,-16
```

*The li command lists ten lines of the source code.*

```
(gdb) li 29
24      movl    $theString, %esi
25      movl    %esi, aString(%rbp) # *aString = "Hello World.\n";
26 whileLoop:
27      movl    aString(%rbp), %esi # current char in string
28      cmpb    $0, (%esi) # null character?
29      je     allDone      # yes, all done
30
31      movl    $1, %edx      # one character
32      movl    $STDOUT, %edi # standard out
```

```

33      call    write      # invoke write function
(gdb)
34
35      incl    aString(%rbp)  # aString++;
36      jmp     whileLoop   # back to top
37 allDone:
38      movl    $0, %eax    # return 0;
39      movq    %rbp, %rsp  # restore stack pointer
40      popq    %rbp       # restore base pointer
41      ret

```

We are trying to observe the while loop. Providing an argument to the `li` command causes it to list ten lines centered around the value of the argument. We still do not see the entire loop. Pressing the Enter key tells `gdb` to repeat the immediately previous command. The `li` command is smart enough to list the next ten lines (only eight in this example since that takes us to the end of the source code in this file).

```

(gdb) br 29
Breakpoint 1 at 0x40050b: file helloWorld3.s, line 29.
(gdb) br 37
Breakpoint 2 at 0x400521: file helloWorld3.s, line 37.

```

From the listed source code, we can see that the decision to exit the loop is made on line 29 in the source code. The jump to the `allDone` label will occur if the `cmpb` instruction on line 28 shows that the `rsi` register is pointing to a byte that contains zero — the ASCII NUL character. I set a breakpoint at line 29 so we can see what `esi` is pointing to.

I also set a breakpoint at line 37, the target of the jump. This second breakpoint serves as a sort of “safety net” in case I did not read the code correctly. If the program does not reach the breakpoint within the loop, perhaps I can work backwards and figure out my error from examining the registers and memory at this point.

```

(gdb) run
Starting program: /home/bob/progs/chap10//helloWorld3

Breakpoint 1, whileLoop () at helloWorld3.s:29
29      je     allDone    # yes, all done

```

The `run` command causes the target program, `helloworld`, to execute until it reaches a breakpoint. Control then returns to the `gdb` program.

**IMPORTANT:** The instruction at the breakpoint is not executed when the break occurs. It will be the first instruction to be executed when we command `gdb` to resume execution of the target program.

```

(gdb) i r
rax          0x7ffff7dd6568 140737351869800
rbx          0x0 0
rcx          0x400530 4195632
rdx          0x7fffffff128 140737488347432
rsi          0x40061c 4195868
rdi          0x1 1

```

```

rbp      0x7fffffff030 0x7fffffff030
rsp      0x7fffffff020 0x7fffffff020
r8       0x4005c0 4195776
r9       0x7fffff7de9740 140737351948096
r10      0x7fffffffde90 140737488346768
r11      0x7fffff7a3e680 140737348101760
r12      0x400410 4195344
r13      0x7fffffff110 140737488347408
r14      0x0 0
r15      0x0 0
rip      0x40050b 0x40050b <whileLoop+7>
eflags   0x206 [ PF IF ]
cs       0x33 51
ss       0x2b 43
ds       0x0 0
es       0x0 0
fs       0x0 0
gs       0x0 0
(gdb) i r rsi
rsi      0x40061c 4195868

```

The *i r* command (notice the space between “*i*” and “*r*”) is used to display all the registers. The left-hand column shows the contents of the register in hexadecimal, and the right-hand column is in decimal. Addresses are usually stated in hexadecimal, so the contents of registers that are supposed to hold only addresses are not converted to decimal.

Since our primary interest is the *rsi* register, we can simplify the display by explicitly specifying which register(s) to display.

```

(gdb) help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
t(binary), f(float), a(address), i(instruction), c(char) and s(string).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format.

```

Defaults for format and size letters are those previously used.  
Default count is 1. Default address is following last thing printed  
with this command or “print”.

```
(gdb) x/10cb 0x40061c
0x40061c <theString>: 72 'H'101 'e'108 'l'108 'l'111 'o'32 ' '119 'w'111 'o'
0x400624 <theString+8>: 114 'r'108 'l'
```

We should examine the byte that *rsi* is pointing to because that determines whether this jump instruction transfers control or not. The *help x* command provides a very brief reminder of the codes to use. The character display (*c*) shows two values for each byte — first in decimal, then the equivalent ASCII letter. We can see that *rsi* is pointing to the beginning of the text string. I chose to display ten characters to confirm that this is the correct text string.

```
(gdb) si
31      movl    $1, %edx    # one character
(gdb)
32      movl    $STDOUT, %edi  # standard out
(gdb)
33      call    write       # invoke write function
(gdb)
0x000000000004003f0 in write@plt ()
```

We use the `si` command to single-step through a portion of the program. Recall that simply pushing the Enter key repeats the immediately previous `gdb` command.

The last step in this sequence gave an odd result. It caused the program to execute the `call` instruction, which took us into the `write` function. Since `write` is a library function, `gdb` does not have access to its source code. Hence, it cannot display the source code for us.

```
(gdb) cont
Continuing.
H
Breakpoint 1, whileLoop () at helloWorld3.s:29
29      je      allDone    # yes, all done
(gdb) i r rsi
rsi          0x40061d 4195869
(gdb) x/10cb 0x40061d
0x40061d <theString+1>: 101 'e'108 'l'108 'l'111 'o'32 ' '119 'w'111 'o'114
0x400625 <theString+9>: 108 'l'100 'd'
```

Not wanting to single-step through the `write` function, I use the `cont` command. The program displays the first letter of the string, "H", on the screen, then loops back and breaks again at line 20. I display register `rsi` and examine the memory it is pointing to. We can see that the pointer variable, `aString`, is marching through the text string one character at a time.

```
(gdb) cont
Continuing.
e
Breakpoint 1, whileLoop () at helloWorld3.s:29
29      je      allDone    # yes, all done
(gdb) clear 29
Deleted breakpoint 1
```

Continuing the program shows that it will break back into `gdb` each time through the loop. We are reasonably confident that the loop is executing properly, so we remove the breakpoint in the loop.

```
(gdb) cont
Continuing.
llo world.

Breakpoint 2, allDone () at helloWorld3.s:38
38      movl    $0, %eax    # return 0;
(gdb) i r rsi
```

```
rsi          0x400629 4195881
```

```
(gdb) x/10cb 0x400649
```

```
0x400629 <theString+13>: 0 '\ 000'0 '\ 000'0 '\ 000'1 '\ 001'27 '\ 033'3 '\  
0x400631: 0 '\ 000'0 '\ 000'
```

*With the breakpoint inside the loop removed, continuing the program displays the remainder of the text. Then it breaks at the breakpoint we set outside the loop. Recall that I set the breakpoint at line 37, but the program breaks at line 32. The reason is that there is no instruction on line 37, just a label. The first instruction following the label is on line 38.*

*I then look at the address in rsi. By examining two bytes previous to where it is currently pointing, we can easily see the last two characters that the program displayed before reaching the NUL character. And it is the NUL character that caused the loop to terminate.*

```
(gdb) cont
```

```
Continuing.
```

```
[Inferior 1 (process 2612) exited normally]
```

```
(gdb) q
```

```
bob@ubuntu:~$
```

*Continuing the program, it completes normally. Notice that even though our target program has completed, we are still in gdb. We need to use the q command to exit from gdb.*

## Appendix D

# Embedding Assembly Code in a C Function

The gcc C compiler has an extension to standard C that allows a programmer to write assembly language instructions within a C function. Of course, you need to be very careful when doing this because you do not know how the compiler has allocated memory and/or registers for the variables. Yes, you can use the “-S” option to see what the compiler did, but if anybody make one change to the function, even compiling it with a different version of gcc, things almost certainly will have changed.

The way to do this is covered in the *info* pages for gcc. In my version (4.1.2) I found it by going to “C Extensions,” then “Extended Asm.” (No, it’s not obvious to me, either.) The presentation here is a very brief introduction.

The overall format is a C statement of the form:

```
asm("assembly_language_instruction" : output(s) : inputs(s));
```

The *output* operands are destinations for the *assembly\_language\_instruction*, and the *input* operands are sources. Each operand is of the form

```
"operand_constraint" : C_expression
```

where the *operand\_constraint* describes what type of register, memory location, etc. should be used for the operand, and *C\_expression* is a C expression, often just a variable name. If there is more than one operand, they are separated by commas.

The *assembly\_language\_instruction* can refer to each operand numerically with the “%n” syntax, starting with  $n = 0$  for the first operand, 1 for the second, etc.

For example, let us consider a case where we wish to add two 32-bit integers. (Yes, there is a C operation to do this, but it is generally better to start with simple examples.) The program is shown in Listing D.1.

---

```
1 /*  
2 * embedAsm1.c  
3 * Very simple example of how to embed assembly language  
4 * in a C function.  
5 * Bob Plantz - 18 June 2009  
6 */  
7  
8 #include <stdio.h>  
9  
10 int main()
```

```

11 {
12     int x, y;
13
14     printf("Enter an integer: ");
15     scanf("%i", &x);
16     printf("Enter another integer: ");
17     scanf("%i", &y);
18     asm("addl %1, %0" : "=m" (x) : "r" (y));
19     printf("The sum is %i\n", x);
20
21     return 0;
22 }
```

Listing D.1: Embedding an assembly language instruction in a C function (C).

There is only one output (destination), and its operand constraint is "`=m`". The '`=`' sign is required to show that it is an output. The '`m`' character shows that this operand is located in memory.

Now, recall that the `addl` instruction requires that at least one of its operands be a register. So we specify the input operand as a register with the "`r`" operand constraint. We have to do this for the assembly language instruction even though the C code does not specify whether the variable, `y`, is in memory or in a register.

The operand constraints are described in the info pages for `gcc`. In my version (4.7.0) I found it by going to "C Extensions," then "Constraints." The documentation covers all the architectures supported by `gcc`, so it is difficult to wade through.

Listing D.2 shows the assembly language actually generated by the compiler.

```

1      .file    "embedAsm1.c"
2      .section .rodata
3 .LC0:   .string "Enter an integer: "
4 .LC1:   .string "%i"
5 .LC2:   .string "Enter another integer: "
6 .LC3:   .string "The sum is %i\n"
7
8 .text
9 .globl main
10 .type  main, @function
11
12 main:
13     pushq %rbp
14     movq %rsp, %rbp
15     subq $16, %rsp
16     movl $.LC0, %edi
17     movl $0, %eax
18     call printf
19     leaq -8(%rbp), %rax
20     movq %rax, %rsi
21     movl $.LC1, %edi
22     movl $0, %eax
23     call __isoc99_scanf
24     movl $.LC2, %edi
25     movl $0, %eax
26
27 
```

```

28     call    printf
29     leaq    -4(%rbp), %rax
30     movq    %rax, %rsi
31     movl    $.LC1, %edi
32     movl    $0, %eax
33     call    __isoc99_scanf
34     movl    -4(%rbp), %eax
35 #APP
36 # 18 "embedAsm1.c" 1
37         addl %eax, -8(%rbp)
38 # 0 "" 2
39 #NO_APP
40     movl    -8(%rbp), %eax
41     movl    %eax, %esi
42     movl    $.LC3, %edi
43     movl    $0, %eax
44     call    printf
45     movl    $0, %eax
46     leave
47     ret
48     .size   main, .-main
49     .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
50     .section .note.GNU-stack,"",@progbits

```

Listing D.2: Embedding an assembly language instruction in a C function gcc assembly language.

In fact, the compiler did allocate *y* in memory, at -4(%rbp). It had to do that because *scanf* needs an address when reading a value from the keyboard.

The embedded assembly language is between the **#APP** and **#NO\_APP** comments on lines 35 and 39, respectively.

```

34     movl    -4(%rbp), %eax
35 #APP
36 # 18 "embedAsm1.c" 1
37         addl %eax, -8(%rbp)
38 # 0 "" 2
39 #NO_APP

```

The *movl* instruction on line 34 loads *y* into a register so that the *addl* instruction on line 37 can add the value to a memory location (*x*). Of course, it would have had to do that even if we had used a C statement for the addition instead of embedding an assembly language instruction.

There may be situations where you need to use a specific register for a variable. Listing D.3 shows how to do this.

```

1 /*
2 * embed_asm2.c
3 * Shows two assembly language instructions embedded
4 * in a C function.
5 * Bob Plantz - 18 June 2009
6 */
7
8 #include <stdio.h>
9

```

```

10 int main()
11 {
12     int x, y;
13     register int z asm("edx");
14
15     printf("Enter an integer: ");
16     scanf("%i", &x);
17     printf("Enter another integer: ");
18     scanf("%i", &y);
19     asm("movl %1, %0\n\taddl %2, %0\n\tsall $4, %0" : "=r" (z) : "m" (x), "m" (y));
20     printf("Sixteen times there sum is %i\n", z);
21
22     return 0;
23 }

```

Listing D.3: Embedding more than one assembly language instruction in a C function and specifying a register (C).

The declaration on line 13,

```
13     register int z asm("edx");
```

shows how to request that the compiler use the edx register for the variable z.

We have decided to embed three assembly language instructions. Recall that each assembly language statement is on a separate line. And on the next line, we tab to the place where the operation code begins. In C, the newline character is '\n' and the tab character is 't'. So if you read line 18 carefully, you will see that there are three lines of assembly language. The first one is terminated by a '\n'. The second instruction begins with a '\t' and is terminated by a '\n'. And the third begins with a '\t'.

The assembly language results are shown in Listing D.4.

```

1     .file    "embedAsm2.c"
2     .section .rodata
3 .LC0:
4     .string "Enter an integer: "
5 .LC1:
6     .string "%i"
7 .LC2:
8     .string "Enter another integer: "
9     .align 8
10 .LC3:
11     .string "Sixteen times there sum is %i\n"
12
13     .globl  main
14     .type   main, @function
15 main:
16     pushq  %rbp
17     movq  %rsp, %rbp
18     subq  $16, %rsp
19     movl  $.LC0, %edi
20     movl  $0, %eax
21     call   printf
22     leaq   -8(%rbp), %rax
23     movq  %rax, %rsi
24     movl  $.LC1, %edi

```

```

25      movl    $0, %eax
26      call    __isoc99_scanf
27      movl    $.LC2, %edi
28      movl    $0, %eax
29      call    printf
30      leaq    -4(%rbp), %rax
31      movq    %rax, %rsi
32      movl    $.LC1, %edi
33      movl    $0, %eax
34      call    __isoc99_scanf
35 #APP
36 # 19 "embedAsm2.c" 1
37      movl    -8(%rbp), %edx
38      addl    -4(%rbp), %edx
39      sall    $4, %edx
40 # 0 "" 2
41 #NO_APP
42      movl    %edx, %eax
43      movl    %eax, %esi
44      movl    $.LC3, %edi
45      movl    $0, %eax
46      call    printf
47      movl    $0, %eax
48      leave
49      ret
50      .size   main, .-main
51      .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
52      .section .note.GNU-stack,"",@progbits

```

Listing D.4: Embedding more than one assembly language instruction in a C function and specifying a register (gcc assembly language).

Indeed, we can see our embedded assembly language on lines 37 - 39:

```

35 #APP
36 # 19 "embedAsm2.c" 1
37      movl    -8(%rbp), %edx
38      addl    -4(%rbp), %edx
39      sall    $4, %edx
40 # 0 "" 2
41 #NO_APP

```

This has been a much abbreviated introduction to embedding assembly language in C. Each situation will be unique, and you will need to study the `info` pages for `gcc` in order to determine what needs to be done. You can also expect the rules to change — hopefully become easier to use — as `gcc` evolves.

# Appendix E

# Exercise Solutions

The solutions to most of the exercises in the book are in this Appendix. You should attempt to work the exercise *before* looking at the solution. But don't allow yourself to get bogged down. If the solution does not come to you within a reasonable amount of time, peek at the solution for a hint.

A word of warning: I have proofread these solutions many times. Each time has turned up several errors. I am amazed at how difficult it is to make everything perfect. If you find an error, please email me and I will try to correct the next printing.

When reading my programming solutions, be aware that my goal is to present simple, easy-to-read code that illustrates the point. I have *not* tried to optimize, neither for size nor performance.

I am also aware that each of us has our own programming style. Yours probably differs from mine. If you are working with an instructor, I encourage you to discuss programming style with him or her. I probably will not change my style, but I support other people's desire to use their own style.

## E.2 Data Storage Formats

- |             |                        |                        |
|-------------|------------------------|------------------------|
| <b>2 -1</b> | a) 4567                | c) fedc                |
|             | b) 89ab                | d) 0250                |
| <b>2 -2</b> | a) 1000 0011 1010 1111 | c) 1010 1010 1010 1010 |
|             | b) 1001 0000 0000 0001 | d) 0101 0101 0101 0101 |
| <b>2 -3</b> | a) 32                  | d) 16                  |
|             | b) 48                  | e) 8                   |
|             | c) 4                   | f) 32                  |
| <b>2 -4</b> | a) 2                   | d) 3                   |
|             | b) 8                   | e) 5                   |
|             | c) 16                  | f) 2                   |

- 2 -5**  $r = 10, n = 8, d_7 = 2, d_6 = 9, d_5 = 4, d_4 = 5, d_3 = 8, d_2 = 2, d_1 = 5, d_0 = 4.$   
 $r = 16, n = 8, d_7 = 2, d_6 = 9, d_5 = 4, d_4 = 5, d_3 = 8, d_2 = 2, d_1 = 5, d_0 = 4.$

- 2 -6** a) 170 e) 128  
b) 85 f) 99  
c) 240 g) 123  
d) 15 h) 255

- 2 -7** a) 43981 e) 32768  
b) 4660 f) 1024  
c) 65244 g) 65535  
d) 2000 h) 12345

- 2 -8** 1. compute the value of each power of 16 in decimal.  
2. multiply each power of 16 by its corresponding  $d_i$ .  
3. sum the terms.

- a) 160 e) 100  
b) 80 f) 12  
c) 255 g) 17  
d) 137 h) 200

- 2 -9** 1. compute the value of each power of 16 in decimal.  
2. multiply each power of 16 by its corresponding  $d_i$ .  
3. sum the terms.

- a) 40960 e) 34952  
b) 65535 f) 400  
c) 1024 g) 43981  
d) 4369 h) 21845

- 2 -10** a) 64 e) ff  
b) 7b f) 10  
c) 0a g) 20  
d) 58 h) 80

- 2 -11** a) 0400 e) 0100  
b) 03e8 f) ffff  
c) 8000 g) 07d5  
d) 7fff h) abcd

**2 -12** Since there are 12 values, we need 4 bits. Any 4-bit code would work. For example,

**code grade**

0000	A
0001	A-
0010	B+
0011	B
0100	B-
0101	C+
0110	C
0111	C-
1000	D+
1001	D
1010	D-
1011	F

**2 -13** The addressing in Figure 2.1 uses only four bits. This limits us to a 16-byte addressing space. In order to increase our space to 17 bytes, we need another bit for the address. The 17th byte would be number 10000.

**2 -14**

address	contents	address	contents
00000000:	106	00000008:	240
00000001:	240	00000009:	2
00000002:	94	0000000a:	51
00000003:	0	0000000b:	60
00000004:	255	0000000c:	195
00000005:	81	0000000d:	60
00000006:	207	0000000e:	85
00000007:	24	0000000f:	170

**2 -15**

address	contents	address	contents
00000000:	0000 0000	00000008:	0000 1000
00000001:	0000 0001	00000009:	0000 1001
00000002:	0000 0010	0000000a:	0000 1010
00000003:	0000 0011	0000000b:	0000 1011
00000004:	0000 0100	0000000c:	0000 1100
00000005:	0000 0101	0000000d:	0000 1101
00000006:	0000 0110	0000000e:	0000 1110
00000007:	0000 0111	0000000f:	0000 1111

**2 -16**

address	contents	address	contents
00000000:	00	00000008:	08
00000001:	01	00000009:	09
00000002:	02	0000000a:	0a
00000003:	03	0000000b:	0b
00000004:	04	0000000c:	0c
00000005:	05	0000000d:	0d
00000006:	06	0000000e:	0e
00000007:	08	0000000f:	0f

**2 -17** The range of 32-bit unsigned ints is 0 – 4,294,967,295, so four bytes will be required. If the storage area begins at byte number 0x2ffffeb96, the number will also occupy bytes number 0x2ffffeb97, 0x2ffffeb98, 0x2ffffeb99.

<b>2 -18</b>	address	contents	address	contents
	00001000:	00	0000100f:	0f
	00001001:	01	00001010:	10
	00001002:	02	00001011:	11
	00001003:	03	00001012:	12
	00001004:	04	00001013:	13
	00001005:	05	00001014:	14
	00001006:	06	00001015:	15
	00001007:	07	00001016:	16
	00001008:	08	00001017:	17
	00001009:	09	00001018:	18
	0000100a:	0a	00001019:	19
	0000100b:	0b	0000101a:	1a
	0000100c:	0c	0000101b:	1b
	0000100d:	0d	0000101c:	1c
	0000100e:	0e	0000101d:	1d
<b>2 -19</b>	number	letter	grade	
	0		A	
	1		B	
	2		C	
	3		D	
	4		F	
<b>2 -26</b>				
1	/*			
2	* echoDecHexAddr.c			
3	* Asks user to enter a number in decimal or hexadecimal			
4	* then echoes it in both bases, also showing where values			
5	* are stored.			
6	*			
7	* Bob Plantz - 19 June 2009			
8	*/			
9				
10	#include <stdio.h>			
11				
12	int main(void)			
13	{			
14	int x;			
15	unsigned int y;			
16				
17	while(1)			
18	{			
19	printf("Enter a decimal integer: ");			
20	scanf("%i", &x);			
21	if (x == 0) break;			
22				
23	printf("Enter a bit pattern in hexadecimal: ");			
24	scanf("%x", &y);			
25	if (y == 0) break;			
26				
27	printf("%i is stored as %#010x at %p, and\n", x, x, &x);			
28	printf("%#010x represents the decimal integer %d stored at %p\n");			

```

29         y, y, &y);
30     }
31     printf("End of program.\n");
32
33     return 0;
34 }
```

---

**2 -28**

```

1 /*
2  * stringInHex.c
3  * displays "Hello world" in hex.
4  *
5  * Bob Plantz - 19 June 2009
6 */
7
8 #include <stdio.h>
9
10 int main(void)
11 {
12     char *stringPtr = "Hello world.\n";
13
14     while (*stringPtr != '\0')
15     {
16         printf("%p: ", stringPtr);
17         printf("0x%02x\n", *stringPtr);
18         stringPtr++;
19     }
20     printf("%p: ", stringPtr);
21     printf("0x%02x\n", *stringPtr);
22
23     return 0;
24 }
```

---

**2 -29** Keyboard input is line buffered by the operating system and is not available to the application program until the user presses the enter key. This action places two characters in the keyboard buffer – the character key pressed and the end of line character. (The “end of line” character differs in different operating systems.)

The call to the `read` function gets one character from the keyboard buffer – the one corresponding to the key the user pressed. Since there is a breakpoint at the instruction following the call to `read`, control returns to the debugger, `gdb`. But the end of line character is still in the keyboard buffer, and the operating system dutifully provides it to `gdb`.

The net result is the same as if you had pushed the enter key immediately in response to `gdb`’s prompt. This causes `gdb` to execute the previous command, which was the `continue` command. So the program immediately loops back to its prompt.

Experiment with this. Try to enter more than one character before pressing the enter key. It is all very consistent. You just have to think through exactly which keys you are pressing when using the debugger to determine what your call to `read` are doing.

**2 -30**

```
1 /*
2 * echoString1.c
3 * Echoes a string entered by user.
4 *
5 * Bob Plantz - 19 June 2009
6 */
7
8 #include <unistd.h>
9 #include <string.h>
10
11 int main(void)
12 {
13     char aString[200];
14     char *stringPtr = aString;
15
16     write(STDOUT_FILENO, "Enter a text string: ",
17           strlen("Enter a text string: ")); // prompt user
18
19     read(STDIN_FILENO, stringPtr, 1); // get first character
20     while (*stringPtr != '\n') // look for end of line
21     {
22         stringPtr++; // move to next location
23         read(STDIN_FILENO, stringPtr, 1); // get next character
24     }
25
26     // now echo for user
27     write(STDOUT_FILENO, "You entered:\n",
28           strlen("You entered:\n"));
29     stringPtr = aString;
30     do
31     {
32         write(STDOUT_FILENO, stringPtr, 1);
33         stringPtr++;
34     } while (*stringPtr != '\n');
35     write(STDOUT_FILENO, stringPtr, 1);
36
37     return 0;
38 }
```

**2 -31**

```
1 /*
2 * echoString2.c
3 * Echoes a string entered by user. Converts input
4 * to C-style string.
5 * Bob Plantz - 19 June 2009
6 */
7
8 #include <stdio.h>
9 #include <unistd.h>
10 #include <string.h>
11
12 int main(void)
```

```
13 {
14     char aString[200];
15     char *stringPtr = aString;
16
17     write(STDOUT_FILENO, "Enter a text string: ",
18           strlen("Enter a text string: ")); // prompt user
19
20     read(STDIN_FILENO, stringPtr, 1);      // get first character
21     while (*stringPtr != '\n')            // look for end of line
22     {
23         stringPtr++;                    // move to next location
24         read(STDIN_FILENO, stringPtr, 1); // get next character
25     }
26     *stringPtr = '\0';                  // make into C string
27
28     // now echo for user
29     printf("You entered:\n%s\n", aString);
30
31     return 0;
32 }
```

---

**2 -32**

```
1 /*
2  * echoString3.c
3  * Echoes a string entered by user.
4  *
5  * Bob Plantz - 19 June 2009
6  */
7
8 #include "readLn.h"
9 #include "writeStr.h"
10
11 int main(void)
12 {
13     char aString[STRLEN]; // limited to 5 for testing readStr
14                         // change to 200 for use
15     writeStr("Enter a text string: ");
16     readLn(aString, STRLEN);
17     writeStr("You entered:\n");
18     writeStr(aString);
19     writeStr("\n");
20
21     return 0;
22 }
```

---

```
1 /*
2  * writeStr.h
3  * Writes a line to standard out.
4  *
5  * input:
6  *   pointer to C-style text string
7  * output:
```

```
8 *      to screen
9 *      returns number of chars written
10 *
11 * Bob Plantz - 19 June 2009
12 */
13
14 #ifndef WRITESTR_H
15 #define WRITESTR_H
16 int writeStr(char *);
17 #endif



---


1 /*
2 * writeStr.c
3 * Writes a line to standard out.
4 *
5 * input:
6 *     pointer to C-style text string
7 * output:
8 *     to screen
9 *     returns number of chars written
10 *
11 * Bob Plantz - 19 June 2009
12 */
13
14 #include <unistd.h>
15 #include "writeStr.h"
16
17 int writeStr(char *stringAddr)
18 {
19     int count = 0;
20
21     while (*stringAddr != '\0')
22     {
23         write(STDOUT_FILENO, stringAddr, 1);
24         count++;
25         stringAddr++;
26     }
27
28     return count;
29 }



---


1 /*
2 * readLn.h
3 * Reads a line from standard in.
4 * Drops newline character. Eliminates
5 * excess characters from input buffer.
6 *
7 * input:
8 *     from keyboard
9 * output:
10 *     null-terminated text string
11 *     returns number of chars in text string
```

```
12  *
13  * Bob Plantz - 19 June 2009
14 */
15
16 #ifndef READLN_H
17 #define READLN_H
18 int readLn(char *, int);
19#endif
20
21 /*
22  * readLn.c
23  * Reads a line from standard in.
24  * Drops newline character. Eliminates
25  * excess characters from input buffer.
26  *
27  * input:
28  *   from keyboard
29  * output:
30  *   null-terminated text string
31  *   returns number of chars in text string
32  *
33  * Bob Plantz - 19 June 2009
34 */
35
36 #include <unistd.h>
37 #include "readLn.h"
38
39 int readLn(char *stringAddr, int maxLength)
40 {
41     int count = 0;
42     maxLength--;           // allow space for NUL
43     read(STDIN_FILENO, stringAddr, 1);
44     while (*stringAddr != '\n')
45     {
46         if (count < maxLength)
47         {
48             count++;
49             stringAddr++;
50         }
51         read(STDIN_FILENO, stringAddr, 1);
52     }
53     *stringAddr = '\0';    // terminate C string
54
55     return count;
56 }
```

## E.3 Computer Arithmetic

**3 -2** Store a digit in every four bits. Thus, the lowest-order digit would be stored in bits 7 – 0, the next lowest-order in 15 – 8, etc., with the highest-order digit in bits 31 – 24.

No, binary addition does not work. For example, let's consider  $48 + 27$ :

<i>number</i>	<i>32bits(hex)</i>
48	00000048
+27	00000027
75	0000007f

### 3 -3 See next answer.

**3 -4** No, it doesn't work. The problem is that the range of 4-bit signed numbers in two's complement format is  $-8 \leq x \leq +7$ , and  $(+4) + (+5)$  exceeds this range.

<i>number</i>		<i>4bits</i>
(+4)	→	0100
+ (5)	→	0101
(-7)	←	1001

**3 -5** No, it doesn't work. The problem is that the range of 4-bit signed numbers in two's complement format is  $-8 \leq x \leq +7$ , and  $(-4) + (-5)$  exceeds this range.

<i>number</i>		<i>4bits</i>
(-4)	→	1100
+ (-5)	→	1011
(+7)	←	0111

**3 -6** Adding any number to its negative will set the CF to one and the OF to zero. The sum is  $2^n$ , where n is the number of bits used for representing the signed integer. That is, the sum is one followed by n zeroes. The one gets recorded in the CF. Since the CF is irrelevant in two's complement arithmetic, the result — n zeroes — is correct.

In two's complement, zero does not have a representation of opposite sign. (-0.0 does exist in IEEE 754 floating point.) Also,  $-2^{n-1}$  does not have a representation of opposite sign.

- 3 -7** a) +85 e) -128  
b) -86 f) +99  
c) -16 g) +123  
d) +15

**3 -8** a) +4660 e) -32768  
b) -4660 f) +1024  
c) -292 g) -1  
d) +2000 h) +30767

**3 -9** a) 64 e) 7f  
b) ff f) f0  
c) f6 g) e0  
d) 58 h) 80

- 3 -10** a) 0400  
 b) fc00  
 c) ffff  
 d) 7fff  
 e) ff00  
 f) 8000  
 g) 8001  
 h) ff80
- 3 -11** a) ff  
 $CF = 0 \Rightarrow$  unsigned right  
 $OF = 0 \Rightarrow$  signed right  
 b) 45  
 $CF = 1 \Rightarrow$  unsigned wrong  
 $OF = 0 \Rightarrow$  signed right  
 c) fb  
 $CF = 0 \Rightarrow$  unsigned right  
 $OF = 0 \Rightarrow$  signed right  
 d) de  
 $CF = 0 \Rightarrow$  unsigned right  
 $OF = 1 \Rightarrow$  signed wrong  
 e) 0e  
 $CF = 1 \Rightarrow$  unsigned wrong  
 $OF = 0 \Rightarrow$  signed right  
 f) 00  
 $CF = 1 \Rightarrow$  unsigned wrong  
 $OF = 1 \Rightarrow$  signed wrong
- 3 -12** a) 0000  
 $CF = 1 \Rightarrow$  unsigned wrong  
 $OF = 0 \Rightarrow$  signed right  
 b) 1110  
 $CF = 1 \Rightarrow$  unsigned wrong  
 $OF = 0 \Rightarrow$  signed right  
 c) 0000  
 $CF = 1 \Rightarrow$  unsigned wrong  
 $OF = 1 \Rightarrow$  signed wrong  
 d) 03ff  
 $CF = 1 \Rightarrow$  unsigned wrong  
 $OF = 0 \Rightarrow$  signed right  
 e) 7fff  
 $CF = 0 \Rightarrow$  unsigned right  
 $OF = 0 \Rightarrow$  signed right  
 f) 7fff  
 $CF = 1 \Rightarrow$  unsigned wrong  
 $OF = 1 \Rightarrow$  signed wrong

---

**3 -14**

```

1 /* 
2 * hexTimesTen.c
3 * Multiplies a hex number by 10.
4 * Bob Plantz - 19 June 2009
5 */
6
7 #include "readLn.h"
8 #include "writeStr.h"
9 #include "hex2int.h"
10 #include "int2hex.h"
11
12 int main(void)
13 {
14     char aString[9];
15     unsigned int x;
16
17     writeStr("Enter a hex number: ");
18     readLn(aString, 9);
19     x = hex2int(aString);
20     x *= 10;
21     int2hex(aString, x);
22     writeStr("Multiplying by ten gives: ");
23     writeStr(aString);
24     writeStr("\n");
25

```

```
26     return 0;
27 }



---



```
1  /*
2  *  hex2int.h
3  *
4  *  Converts a hexadecimal text string to corresponding
5  *  unsigned int format.
6  *  Assumes text string is valid hex chars.
7  *
8  *  input:
9  *      pointer to null-terminated text string
10 *  output:
11 *      returns the unsigned int.
12 *
13 *  Bob Plantz - 19 June 2009
14 */

15 #ifndef HEX2INT_H
16 #define HEX2INT_H
17
18
19 unsigned int hex2int(char *hexString);
20
21 #endif



---



```
1  /*
2  *  hex2int.c
3  *
4  *  Converts a hexadecimal text string to corresponding
5  *  unsigned int format.
6  *  Assumes text string is valid hex chars.
7  *
8  *  input:
9  *      pointer to null-terminated text string
10 *  output:
11 *      returns the unsigned int.
12 *
13 *  Bob Plantz - 19 June 2009
14 */

15
16 #include "hex2int.h"
17
18 unsigned int hex2int(char *hexString)
19 {
20     unsigned int x;
21     unsigned char aChar;
22
23     x = 0;                      // initialize result
24     while (*hexString != '\0')    // end of string?
25     {
26         x = x << 4;           // make room for next four bits
27         aChar = *hexString;
```


```


```

```
28         if (aChar <= '9')
29             x = x + (aChar & 0x0f);
30         else
31         {
32             aChar = aChar & 0x0f;
33             aChar = aChar + 9;
34             x = x + aChar;
35         }
36         hexString++;
37     }
38
39     return x;
40 }
```

---

```
1 /*
2  * int2hex.h
3  *
4  * Converts an unsigned int to corresponding
5  * hex text string format.
6  * Assumes char array is big enough.
7  *
8  * input:
9  *     unsigned int
10 * output:
11 *     null-terminated text string
12 *
13 * Bob Plantz - 19 June 2009
14 */
15
16 #ifndef INT2HEX_H
17 #define INT2HEX_H
18
19 void int2hex(char *hexString, unsigned int number);
20
21 #endif
```

---

```
1 /*
2  * int2hex.c
3  *
4  * Converts an unsigned int to corresponding
5  * hex text string format.
6  * Assumes char array is big enough.
7  *
8  * input:
9  *     unsigned int
10 * output:
11 *     null-terminated text string
12 *
13 * Bob Plantz - 19 June 2009
14 */
15
16 #include "int2hex.h"
```

```
17
18 void int2hex(char *hexString, unsigned int number)
19 {
20     unsigned char aChar;
21     int i;
22
23     hexString[8] = '\0';           // install string terminator
24     for (i = 7; i >= 0; i--)
25     {
26         aChar = number & 0x0f;   // get four bits
27         if (aChar <= 9)
28             aChar += '0';
29         else
30             aChar = aChar - 10 + 'a';
31         hexString[i] = aChar;
32         number = number >> 4;
33     }
34 }
```

See Section E.2 for writeStr and readLn.

---

**3 -15**

```
1 /*
2  * binTimesTen.c
3  * Multiplies a hex number by 10.
4  *
5  * Bob Plantz - 19 June 2009
6  */
7
8 #include "readLn.h"
9 #include "writeStr.h"
10 #include "bin2int.h"
11 #include "int2bin.h"
12
13 int main(void)
14 {
15     char aString[33];
16     unsigned int x;
17
18     writeStr("Enter a binary number: ");
19     readLn(aString, 33);
20     x = bin2int(aString);
21     x *= 10;
22     int2bin(aString, x);
23     writeStr("Multiplying by ten gives: ");
24     writeStr(aString);
25     writeStr("\n");
26
27     return 0;
28 }
```

---

```
1 /*
2  * bin2int.h
```

```
3  *
4  * bin2int.c
5  * Converts a binary text string to corresponding
6  * unsigned int format.
7  * Assumes text string contains valid binary chars.
8  *
9  * input:
10 *      pointer to null-terminated text string
11 * output:
12 *      returns the unsigned int.
13 *
14 * Bob Plantz - 19 June 2009
15 */
16
17 #ifndef BIN2INT_H
18 #define BIN2INT_H
19
20 unsigned int bin2int(char *binString);
21
22 #endif
```

---

```
1 /*
2  * bin2int.c
3  * Converts a binary text string to corresponding
4  * unsigned int format.
5  * Assumes text string contains valid binary chars.
6  *
7  * input:
8  *      pointer to null-terminated text string
9  * output:
10 *      returns the unsigned int.
11 *
12 * Bob Plantz - 19 June 2009
13 */
14
15 #include "bin2int.h"
16
17 unsigned int bin2int(char *binString)
18 {
19     unsigned int x;
20     unsigned char aChar;
21
22     x = 0;           // initialize result
23     while (*binString != '\0') // end of string?
24     {
25         x = x << 1;        // make room for next bit
26         aChar = *binString;
27         x |= (0x1 & aChar); // sift out the bit
28         binString++;
29     }
30
31     return x;
```

```
32 }



---



```
1 /*
2  * int2bin.h
3  *
4  * Converts an unsigned int to corresponding
5  * binary text string format.
6  * Assumes char array is big enough.
7  *
8  * input:
9  *     unsigned int
10 * output:
11 *     null-terminated text string
12 *
13 * Bob Plantz - 19 June 2009
14 */
15
16 #ifndef INT2BIN_H
17 #define INT2BIN_H
18
19 void int2bin(char *binString, unsigned int number);
20
21 #endif


---



```
1 /*
2  * int2bin.c
3  *
4  * Converts an unsigned int to corresponding
5  * binary text string format.
6  * Assumes char array is big enough.
7  *
8  * input:
9  *     unsigned int
10 * output:
11 *     null-terminated text string
12 *
13 * Bob Plantz - 19 June 2009
14 */
15
16 #include "int2bin.h"
17
18 void int2bin(char *binString, unsigned int number)
19 {
20     int i;
21
22     binString[32] = '\0';      // install string terminator
23     for (i = 31; i >= 0; i--)
24     {
25         if (number & 0x01)
26             binString[i] = '1';
27         else
28             binString[i] = '0';

```


```


```

```
29         number = number >> 1;
30     }
31 }
```

See Section E.2 for `writeStr` and `readLn`.

---

**3 -16**

```
1 /*
2  * uDecTimesTen.c
3  * Multiplies a decimal number by 10.
4  * Bob Plantz - 20 June 1009
5  */
6
7 #include "readLn.h"
8 #include "writeStr.h"
9 #include "udec2int.h"
10 #include "int2bin.h"
11
12 int main(void)
13 {
14     char aString[33];
15     unsigned int x;
16
17     writeStr("Enter a decimal number: ");
18     readLn(aString, 33);
19     x = udec2int(aString);
20     x *= 10;
21     int2bin(aString, x);
22     writeStr("Multiplying by ten gives (in binary): ");
23     writeStr(aString);
24     writeStr("\n");
25
26     return 0;
27 }
```

---

```
1 /*
2  * uDec2int.h
3  *
4  * Converts a decimal text string to corresponding
5  * unsigned int format.
6  * Assumes text string is valid decimal chars.
7  *
8  * input:
9  *     pointer to null-terminated text string
10 * output:
11 *     returns the unsigned int.
12 *
13 * Bob Plantz - 19 June 2009
14 */
15
16 #ifndef UDEC2INT_H
17 #define UDEC2INT_H
18
```

```
19 unsigned int uDec2int(char *decString);  
20  
21 #endif

---

  
1 /*  
2  * uDec2int.c  
3  *  
4  * Converts a decimal text string to corresponding  
5  * unsigned int format.  
6  * Assumes text string is valid decimal chars.  
7  *  
8  * input:  
9  *      pointer to null-terminated text string  
10 * output:  
11 *      returns the unsigned int.  
12 *  
13 * Bob Plantz - 19 June 2009  
14 */  
15  
16 #include "uDec2int.h"  
17  
18 unsigned int uDec2int(char *decString)  
19 {  
20     unsigned int x;  
21     unsigned char aChar;  
22  
23     x = 0;                                // initialize result  
24     while (*decString != '\0')    // end of string?  
25     {  
26         x *= 10;  
27         aChar = *decString;  
28         x += (0xf & aChar);  
29         decString++;  
30     }  
31  
32     return x;  
33 }
```

---

See above for int2bin. See Section E.2 for writeStr and readLn.

**3 -17**

```
1 /*  
2  * sDecTimesTen.c  
3  * Multiplies a signed decimal number by 10  
4  * and shows result in binary.  
5  * Bob Plantz - 21 June 2009  
6  */  
7  
8 #include "readLn.h"  
9 #include "writeStr.h"  
10 #include "sDec2int.h"  
11 #include "int2bin.h"  
12
```

```
13 int main(void)
14 {
15     char aString[33];
16     int x;
17
18     writeStr("Enter a signed decimal number: ");
19     readLn(aString, 33);
20     x = sDec2int(aString);
21     x *= 10;
22     int2bin(aString, x);
23     writeStr("Multiplying by ten gives (in binary): ");
24     writeStr(aString);
25     writeStr("\n");
26
27     return 0;
28 }
```

---

```
1 /*
2  * sDec2int.h
3  *
4  * Converts a decimal text string to corresponding
5  * signed int format.
6  * Assumes text string is valid decimal chars.
7  *
8  * input:
9  *     pointer to null-terminated text string
10 * output:
11 *     returns the signed int.
12 *
13 * Bob Plantz - 19 June 2009
14 */
15
16 #ifndef SDEC2INT_H
17 #define SDEC2INT_H
18
19 int sDec2int(char *decString);
20
21 #endif
```

---

```
1 /*
2  * sDec2int.c
3  *
4  * Converts a decimal text string to corresponding
5  * signed int format.
6  * Assumes text string is valid decimal chars.
7  *
8  * input:
9  *     pointer to null-terminated text string
10 * output:
11 *     returns the signed int.
12 *
13 * Bob Plantz - 19 June 2009
```

```

14  */
15
16 #include "uDec2int.h"
17 #include "sDec2int.h"
18
19 int sDec2int(char *decString)
20 {
21     int x;
22     int negative = 0;
23
24     if (*decString == '-')
25     {
26         negative = 1;
27         decString++;
28     }
29     else
30     {
31         if (*decString == '+')
32             decString++;
33     }
34
35     x = uDec2int(decString);
36
37     if (negative)
38         x *= -1;
39
40     return x;
41 }
```

See above for `int2bin` and `uDec2int`. See Section E.2 for `writeStr` and `readLn`.

## E.4 Logic Gates

**4 -1** Using truth tables:

$x$	$x \cdot 1$
0	0
1	1

$x$	$x + 0$
0	0
1	1

**4 -2** Using truth tables:

$x$	$y$	$x \cdot y$	$y \cdot x$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

$x$	$y$	$x + y$	$y + x$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

**4 -3** Using truth tables:

$x$	$y$	$x \cdot 0$
0	0	0
1	0	0

$x$	$y$	$x + 1$
0	1	1
1	1	1

**4 -4** Using truth tables:

$x$	$x'$	$x \cdot 0$
0	1	0
1	0	0

$x$	$x'$	$x + 1$
0	1	1
1	0	1

**4 -5** Using truth tables:

$x$	$x$	$x \cdot 0$
0	0	0
1	1	1

$x$	$x$	$x + 1$
0	0	0
1	1	1

**4 -6** Using truth tables:

$x$	$y$	$z$	$x \cdot (y + z)$	$x \cdot y + x \cdot z$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

$x$	$y$	$z$	$x + y \cdot z$	$(x + y) \cdot (x + z)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

**4 -7** Using a truth table and letting  $y = x'$ :

$x$	$y = x'$	$y'$
0	1	0
1	0	1

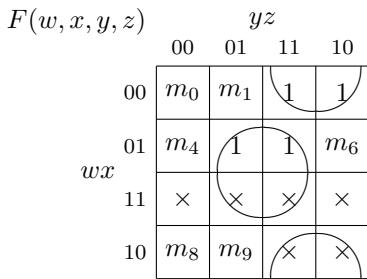
**4 -9** Minterms:

$F(x, y, z)$		$xy$			
		00	01	11	10
$z$	0	$m_0$	$m_2$	$m_6$	$m_4$
	1	$m_1$	$m_3$	$m_7$	$m_5$

**4 -10** Minterms:

$F(x, y, z)$		$xz$			
		00	01	11	10
$y$	0	$m_0$	$m_1$	$m_5$	$m_4$
	1	$m_2$	$m_3$	$m_7$	$m_6$

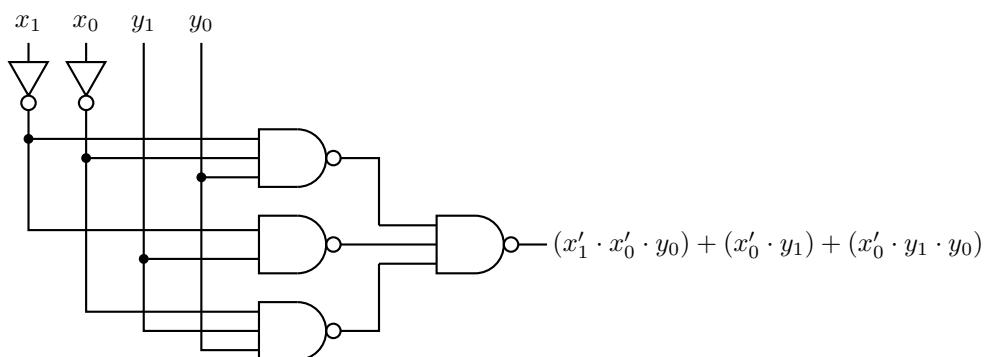
**4 -11** The prime numbers correspond to the minterms  $m_2, m_3, m_5$ , and  $m_7$ . The minterms  $m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, m_{15}$  cannot occur so are marked “don’t care” on the Karnaugh map.



$$F(w, x, y, z) = x \cdot z + x' \cdot y$$

**4 -15** 2-bit “below” circuit.

$x_1$	$x_0$	$y_1$	$y_0$	$F$
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



## E.5 Logic Circuits

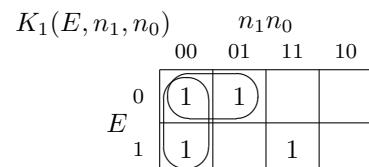
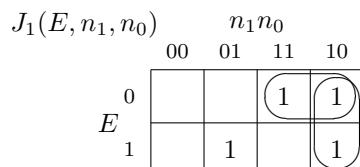
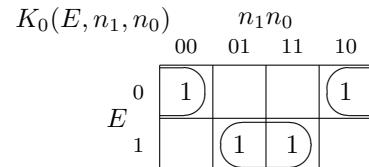
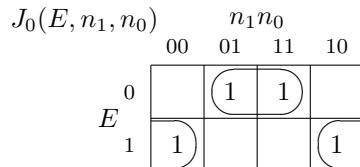
**5 -3** Referring to Figure 5.27 (page 114), we see that  $JK = 10$  is the set (state = 1) input and  $JK = 01$  is the reset (state = 0).

Current		Enable = 0						Enable = 1					
		Next		$J_1$	$K_1$	$J_0$	$K_0$	Next		$J_1$	$K_1$	$J_0$	$K_0$
$n_1$	$n_0$	$n_1$	$n_0$					$n_1$	$n_0$				
0	0	0	0	1	0	1	1	0	1	0	1	1	0
0	1	0	1	0	1	1	0	1	0	1	0	0	1
1	0	1	0	1	0	0	1	1	1	1	0	1	0
1	1	1	1	1	0	1	0	0	0	0	1	0	1

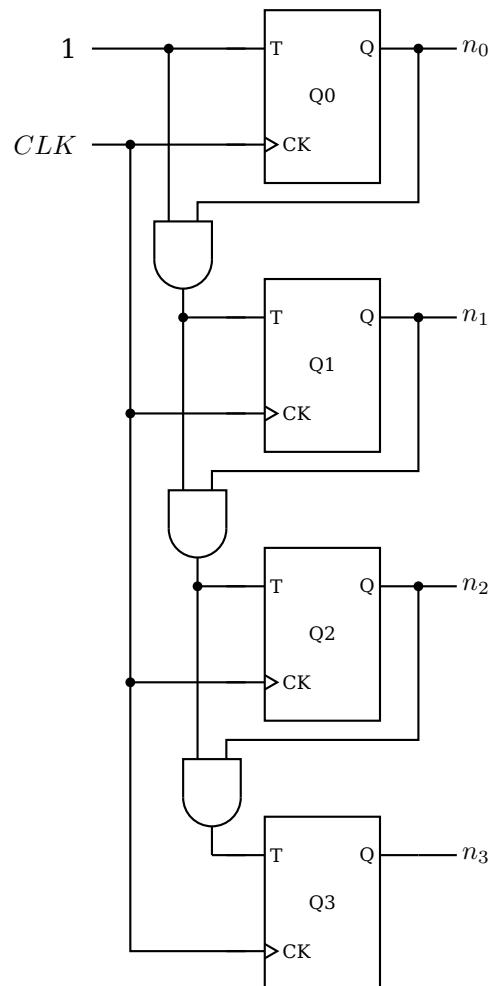
This leads to the following equations for the inputs to the JK flip-flops (using “ $E$ ” for “ $Enable$ ”):

$$\begin{aligned} J_0 &= E' \cdot n'_1 \cdot n_0 + E' \cdot n_1 \cdot n_0 + E \cdot n'_1 \cdot n'_0 + E \cdot n_1 \cdot n'_0 \\ K_0 &= E' \cdot n'_1 \cdot n'_0 + E' \cdot n_1 \cdot n'_0 + E \cdot n'_1 \cdot n_0 + E \cdot n_1 \cdot n_0 \\ J_1 &= E' \cdot n_1 \cdot n'_0 + E' \cdot n_1 \cdot n_0 + E \cdot n'_1 \cdot n_0 + E \cdot n_1 \cdot n'_0 \\ K_1 &= E' \cdot n'_1 \cdot n'_0 + E' \cdot n'_1 \cdot n_0 + E \cdot n'_1 \cdot n'_0 + E \cdot n_1 \cdot n_0 \end{aligned}$$

Simplify these equations using Karnaugh maps.



$$\begin{aligned} J_0 &= E' \cdot n_0 + E \cdot n'_0 \\ K_0 &= E' \cdot n'_0 + E \cdot n_1 \\ J_1 &= E' \cdot n_1 + n_1 \cdot n'_0 + E \cdot n'_1 \cdot n_0 \\ K_1 &= E' \cdot n'_1 + n'_1 \cdot n'_0 + E \cdot n_1 \cdot n_0 \end{aligned}$$

**5 -4** Four-bit up counter.**E.6 Central Processing Unit**

**6 -3** The compiler will not use a register for the `theInteger` variable because the algorithm requires the address of this variable, and registers have no memory address.

**6 -5**

```

1 /*
2 * endian.c
3 * Determines endianess. If endianess cannot be determined
4 * from input value, defaults to "big endian"
5 * Bob Plantz - 22 June 2009
6 */
7
8 #include <stdio.h>
9
10 int main(void)
11 {

```

```

12     unsigned char *ptr;
13     int x, i, bigEndian;
14
15     ptr = (unsigned char *)&x;
16
17     printf("Enter a non-zero integer: ");
18     scanf("%i", &x);
19
20     printf("You entered %#010x and it is stored\n", x);
21     for (i = 0; i < 4; i++)
22         printf("    %p: %02x\n", ptr + i, *(ptr + i));
23
24     bigEndian = (*ptr == (unsigned char)(0xff & (x >> 24))) &&
25             (*ptr + 1) == (unsigned char)(0xff & (x >> 16))) &&
26             (*ptr + 2) == (unsigned char)(0xff & (x >> 8))) &&
27             (*ptr + 3) == (unsigned char)(0xff & x));
28     if (bigEndian)
29         printf("which is big endian.\n");
30     else
31         printf("which is little endian.\n");
32
33     return 0;
34 }
```

**6 -6**

```

1 /* 
2  * endianReg.c
3  * Stores user int in memory then copies to register var.
4  * Use gdb to observe endianess.
5  * Bob Plantz - 22 June 2009
6 */
7
8 #include <stdio.h>
9
10 int main(void)
11 {
12     int x;
13     register int y;
14
15     printf("Enter an integer: ");
16     scanf("%i", &x);
17
18     y = x;
19     printf("You entered %i\n", y);
20
21     return 0;
22 }
```

When I ran this program with the input -1985229329, I got the results:

```
(gdb) print &x
$5 = (int *) 0x7fffff74f473c
(gdb) x/4xb 0x7fffff74f473c
```

```

0x7fffff74f473c: 0xef 0xcd 0xab 0x89
(gdb) i r rcx
rcx          0xffffffff89abcdef -1985229329
(gdb) print x
$6 = -1985229329
(gdb)

```

which shows the value stored in `rcx` (used as the `y` variable) is in regular order, and the value store in memory (the `x` variable) is in little endian.

## E.7 Programming in Assembly Language

---

7 -1

```

1 # f.s
2 # Does nothing but return zero to caller.
3 # Bob Plantz - 22 June 2009
4
5     .text
6     .globl  f
7     .type   f, @function
8 f:
9     pushq  %rbp          # save caller's frame pointer
10    movq    %rsp, %rbp    # establish ours
11
12    movl    $0, %eax      # return 0;
13
14    movq    %rbp, %rsp    # delete local vars.
15    popq    %rbp          # restore caller's frame pointer
16    ret                 # return to caller

```

---

7 -2

```

1 # g.s
2 # Does nothing but return to caller.
3 # Bob Plantz - 22 June 2009
4
5     .text
6     .globl  g
7     .type   g, @function
8 g:
9     pushq  %rbp          # save caller's frame pointer
10    movq    %rsp, %rbp    # establish ours
11
12 # A function that returns void has "garbage" in eax.
13
14    movq    %rbp, %rsp    # delete local vars.
15    popq    %rbp          # restore caller's frame pointer
16    ret                 # return to caller

```

---

7 -3

```

1 # h.s
2 # Does nothing but return 123 to caller.
3 # Bob Plantz - 22 June 2009

```

```

4
5      .text
6      .globl h
7      .type h, @function
8 h:
9      pushq %rbp          # save caller's frame pointer
10     movq %rsp, %rbp    # establish ours
11
12     movl $123, %eax    # return 123;
13
14     movq %rbp, %rsp    # delete local vars.
15     popq %rbp          # restore caller's frame pointer
16     ret                 # return to caller

```

---

## 7 -4

```

1 /*
2 * checkRetNos.c
3 * calls three assembly language functions and
4 * prints their return numbers.
5 *
6 * Bob Plantz - 22 June 2009
7 */
8
9 #include <stdio.h>
10 int one();
11 int two();
12 int three();
13
14 int main()
15 {
16     int x;
17
18     x = one();
19     printf("one returns %i, ", x);
20
21     x = two();
22     printf("two returns %i, and ", x);
23
24     x = three();
25     printf("three returns %i.\n", x);
26
27     return 0;
28 }

```

---

```

1 # one.s
2 # returns 1 to calling function.
3 # Bob Plantz - 22 June 2009
4
5      .text
6      .globl one
7      .type one, @function
8 one:

```

---

```

9      pushq  %rbp          # save caller's base pointer
10     movq   %rsp, %rbp    # establish ours
11
12     movl   $1, %eax      # return 1;
13
14     movq   %rbp, %rsp    # delete local vars.
15     popq   %rbp          # restore caller's base pointer
16     ret                # return to caller

```

---

```

1 # two.s
2 # returns 2 to calling function.
3 # Bob Plantz - 22 June 2009
4
5      .text
6      .globl two
7      .type two, @function
8 two:
9      pushq  %rbp          # save caller's base pointer
10     movq   %rsp, %rbp    # establish ours
11
12     movl   $2, %eax      # return 1;
13
14     movq   %rbp, %rsp    # delete local vars.
15     popq   %rbp          # restore caller's base pointer
16     ret                # return to caller

```

---

```

1 # three.s
2 # returns 3 to calling function.
3 # Bob Plantz - 22 June 2009
4
5      .text
6      .globl three
7      .type three, @function
8 three:
9      pushq  %rbp          # save caller's base pointer
10     movq   %rsp, %rbp    # establish ours
11
12     movl   $3, %eax      # return 3;
13
14     movq   %rbp, %rsp    # delete local vars.
15     popq   %rbp          # restore caller's base pointer
16     ret                # return to caller

```

---

7 -5

```

1 /*
2 * checkRetLtrs.c
3 * calls three assembly language functions and
4 * prints their return characters.
5 *
6 * Bob Plantz - 22 June 2009
7 */
8

```

```
9 #include <stdio.h>
10 char el();
11 char em();
12 char en();
13
14 int main()
15 {
16     char letter;
17
18     letter = el();
19     printf("el returns %c, ", letter);
20
21     letter = em();
22     printf("en returns %c, and ", letter);
23
24     letter = en();
25     printf("em returns %c.\n", letter);
26
27     return 0;
28 }
```

---

```
1 # el.s
2 # returns L to calling function.
3 # Bob Plantz - 22 June 2009
4         .text
5         .globl el
6         .type el, @function
7 el:
8         pushq %rbp          # save caller's base pointer
9         movq %rsp, %rbp      # establish ours
10
11        movl $'L', %eax      # return 'L';
12
13        movq %rbp, %rsp      # delete local vars.
14        popq %rbp            # restore caller's base pointer
15        ret                  # return to caller
```

---

```
1 # em.s
2 # returns M to calling function.
3 # Bob Plantz - 22 June 2009
4         .text
5         .globl em
6         .type em, @function
7 em:
8         pushq %rbp          # save caller's base pointer
9         movq %rsp, %rbp      # establish ours
10
11        movl $'M', %eax      # return 'M';
12
13        movq %rbp, %rsp      # delete local vars.
14        popq %rbp            # restore caller's base pointer
15        ret                  # return to caller
```

---

---

```

1 # en.s
2 # returns N to calling function.
3 # Bob Plantz - 22 June 2009
4     .text
5     .globl en
6     .type en, @function
7 en:
8     pushq %rbp          # save caller's base pointer
9     movq %rsp, %rbp      # establish ours
10    movl $'N', %eax      # return 'N';
11    movq %rbp, %rsp      # delete local vars.
12    popq %rbp            # restore caller's base pointer
13    ret                  # return to caller

```

---

**7 -6** The four characters are returned as a 4-byte word and then stored in memory by main. They are then written to standard out one character at a time. Storage order in memory is little endian, so the characters are displayed “backwards.”

---

```

1 /*
2  * fourLetterWord.c
3  * calls a function to get a four letter word, then
4  * prints it.
5  *
6  * Bob Plantz - 22 June 2009
7 */
8
9 #include <unistd.h>
10 #include "retWord.h"
11
12 int main()
13 {
14     int x;
15     char endl = '\n';
16
17     x = retWord();
18     write(STDOUT_FILENO, &x, 4);
19
20     write(STDOUT_FILENO, &endl, 1);
21
22     return 0;
23 }

```

---



---

```

1 # retWord.s
2 # returns 4-letter word to calling function.
3 # Bob Plantz - 22 June 2009
4     .text
5     .globl retWord
6     .type retWord, @function
7 retWord:

```

```
8      pushq  %rbp          # save caller's base pointer
9      movq    %rsp, %rbp    # establish ours
10
11     movl    $0x61426339, %eax   # return "aBc9";
12
13     movq    %rbp, %rsp    # delete local vars.
14     popq    %rbp          # restore caller's base pointer
15     ret                 # return to caller
```

## E.8 Program Data - Input, Store, Output

8 -2

```
1  /*
2  * stackPositive.c
3  * implementation of push and pop stack operations in C
4  *
5  * Bob Plantz - 22 June 2009
6  */
7
8 #include <stdio.h>
9
10 int theStack[500];
11 int *stackPointer = &theStack[0];
12
13 /*
14 * precondition:
15 *     stackPointer points to data element at top of stack
16 * postcondtion:
17 *     address in stackPointer is incremented by four
18 *     data_value is stored at top of stack
19 */
20 void push(int data_value)
21 {
22     stackPointer++;
23     *stackPointer = data_value;
24 }
25
26 /*
27 * precondition:
28 *     stackPointer points to data element at top of stack
29 * postcondtion:
30 *     data element at top of stack is copied to *data_location
31 *     address in stackPointer is decremented by four
32 */
33 void pop(int *data_location)
34 {
35     *data_location = *stackPointer;
36     stackPointer--;
37 }
38
39 int main(void)
```

```

40 {
41     int x = 12;
42     int y = 34;
43     int z = 56;
44     printf("Start with the stack pointer at %p", (void *)stackPointer);
45     printf(", and x = %i, y = %i, and z = %i\n", x, y, z);
46
47     push(x);
48     push(y);
49     push(z);
50     x = 100;
51     y = 200;
52     z = 300;
53     printf("Now the stack pointer is at %p", (void *)stackPointer);
54     printf(", and x = %i, y = %i, and z = %i\n", x, y, z);
55     pop(&z);
56     pop(&y);
57     pop(&x);
58
59     printf("And we end with the stack pointer at %p", (void *)stackPointer);
60     printf(", and x = %i, y = %i, and z = %i\n", x, y, z);
61
62     return 0;
63 }

```

---

**8 -3** Use gdb to examine the values in the rbp and rsp registers just before the first and just before the last instructions are executed.

**8 -4** This exercise shows that the text strings and local variables are stored in different areas of memory.

**8 -6**

```

1 # int2hex.s
2 # Prompts user to enter an integer, then displays its hex equivalent
3 # Bob Plantz - 22 June 2009
4
5 # Stack frame
6     .equ    anInt,-4
7     .equ    localSize,-16
8 # Read only data
9     .section .rodata
10 prompt:
11     .string "Enter an integer number: "
12 scanFormat:
13     .string "%i"
14 printFormat:
15     .string "%i = %x\n"
16 # Code
17     .text          # switch to text segment
18     .globl  main
19     .type   main, @function
20 main:
21     pushq   %rbp      # save caller's base pointer

```

```

22      movq    %rsp, %rbp      # establish our base pointer
23      addq    $localSize, %rsp  # for local variable
24
25      movl    $prompt, %edi   # address of prompt text string
26      movl    $0, %eax       # no floating point args.
27      call    printf        # invoke printf function
28
29      leaq    anInt(%rbp), %rsi # place to store integer
30      movl    $scanFormat, %edi # address of scanf format string
31      movl    $0, %eax       # no floating point args.
32      call    scanf         # invoke scanf function
33
34      movl    anInt(%rbp), %edx # the integer
35      movl    anInt(%rbp), %esi # two copies
36      movl    $printfFormat, %edi # address of printf text string
37      movl    $0, %eax       # no floating point args.
38      call    printf        # invoke printf function
39
40      movl    $0, %eax       # return 0
41      movq    %rbp, %rsp     # delete local variables
42      popq    %rbp          # restore caller's base pointer
43      ret                 # back to calling function

```

**8 -7**

```

1 # assignSeveral.s
2 # Assigns values to four chars and four ints and prints them.
3 # Bob Plantz - 22 June 2009
4
5 # Stack frame
6      .equ    a,-1
7      .equ    b,-2
8      .equ    c,-3
9      .equ    d,-4
10     .equ    w,-8
11     .equ    x,-12
12     .equ    y,-16
13     .equ    z,-20
14     .equ    arg7,0
15     .equ    arg8,8
16     .equ    arg9,16
17     .equ    localSize,-48
18 # Read only data
19     .section .rodata
20 format:
21     .string "The values are %c, %i, %c, %i, %c, %i, %c, and %i\n"
22 # Code
23     .text
24     .globl  main
25     .type   main, @function
26 main:
27     pushq   %rbp           # save calling function's base pointer
28     movq    %rsp, %rbp      # establish our base pointer
29     addq    $localSize, %rsp # allocate memory for local variables

```

```

30
31     movb    $'A', a(%rbp)      # initialize chars
32     movb    $'B', b(%rbp)
33     movb    $'C', c(%rbp)
34     movb    $'D', d(%rbp)
35     movl    $12, w(%rbp)      # and ints
36     movl    $34, x(%rbp)
37     movl    $45, y(%rbp)
38     movl    $67, z(%rbp)

39
40     movslq  z(%rbp), %rax   # load z
41     movq    %rax, arg9(%rsp) # and place on stack
42     movzbq d(%rbp), %rax   # load d
43     movq    %rax, arg8(%rsp) # place on stack
44     movslq  y(%rbp), %rax   # load y
45     movq    %rax, arg7(%rsp) # place on stack
46     movl    y(%rbp), %r9d
47     movzbl  c(%rbp), %r9d   # load args into regs.
48     movl    x(%rbp), %r8d
49     movzbl  b(%rbp), %ecx
50     movl    w(%rbp), %edx
51     movzbl  a(%rbp), %esi
52     movl    $format, %edi    # format string
53     movl    $0, %eax         # no floating point values
54     call    printf

55
56     movq    %rbp, %rsp      # delete local variables
57     popq    %rbp            # restore calling function's base point
58     movl    $0, %eax         # return 0
59     ret

```

## E.9 Computer Operations

**9 -1** Your numbers may differ.

instruction	n bytes	rax	rsi	rbp	rsp
pushq %rbp	1	7f940f088a60	7fff24330778	0	7fff172a9618
movl %rsp, %rbp	3	7f940f088a60	7fff24330778	0	7fff172a9610
movl \$0xabcd1234,%esi	5	7f940f088a60	abcd1234	7fff172a9610	7fff172a9610
movl \$0, %eax	5	0	abcd1234	7fff172a9610	7fff172a9610
movl %rbp, %rsp	3	0	abcd1234	7fff172a9610	7fff172a9610
popq %rbp	1	0	abcd1234	0	7fff172a9618
ret					

**9 -3** Comments have been added. Your results may vary depending on version changes in gcc.

```

1      .file  "addAndSubtract3.c"
2      .section .rodata
3 .LC0:
4      .string "Enter two integers: "
5 .LC1:
6      .string "%i %i"

```

```

7 .LC2:
8     .string "sum = %i, difference = %i\n"
9     .text
10    .globl main
11    .type main, @function
12 main:
13     pushq %rbp
14     movq %rsp, %rbp
15     subq $16, %rsp
16     movl $.LC0, %edi
17     movl $0, %eax
18     call printf
19     leaq -12(%rbp), %rdx
20     leaq -16(%rbp), %rax
21     movq %rax, %rsi
22     movl $.LC1, %edi
23     movl $0, %eax
24     call __isoc99_scanf
25     movl -16(%rbp), %eax      # load w
26     movl %eax, -8(%rbp)       # y = w;
27     movl -12(%rbp), %eax      # load x
28     addl %eax, -8(%rbp)       # y += x;
29     movl -16(%rbp), %eax      # load w
30     movl %eax, -4(%rbp)       # z = w;
31     movl -12(%rbp), %eax      # load x
32     subl %eax, -4(%rbp)       # z -= x;
33     movl -4(%rbp), %edx
34     movl -8(%rbp), %eax
35     movl %eax, %esi
36     movl $.LC2, %edi
37     movl $0, %eax
38     call printf
39     movl $0, %eax
40     leave
41     ret
42     .size main, .-main
43     .ident "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
44     .section .note.GNU-stack,"",@progbits

```

- 9 -4** The assembly language program in Listing 9.6 uses esi for the y variable and edx for the z variable. If there is overflow, the call to printf changes the contents of these registers. So when the results are displayed y and/or z are incorrect.

```

1 # addAndSubtract3.s
2 # Gets two integers from user, then
3 # performs addition and subtraction
4 # Bob Plantz - 23 June 2009
5 # Stack frame
6     .equ w,-16
7     .equ x,-12
8     .equ y,-8
9     .equ z,-4

```

```
10          .equ    localSize, -16
11 # Read only data
12          .section .rodata
13 prompt:      .string "Enter two integers: "
14 getData:     .string "%i %i"
15 warning:     .string "Overflow has occurred.\n"
16 display:     .string "sum = %i, difference = %i\n"
17
18
19
20
21 # Code
22          .text
23          .globl main
24          .type main, @function
25 main:
26          pushq %rbp      # save caller's base pointer
27          movq %rsp, %rbp  # establish our base pointer
28          addq $localSize, %rsp  # for local vars
29
30          movl $prompt, %edi  # prompt user
31          movl $0, %eax      # no floats
32          call printf
33
34          leaq x(%rbp), %rdx  # &x
35          leaq w(%rbp), %rsi  # &w
36          movl getData, %edi  # get user data
37          movl $0, %eax      # no floats
38          call scanf
39
40 #####
41 # These three instructions could replace the four that follow
42 # this sequence. They work because mov does not affect eflags.
43 # But changes in the code may introduce an instruction before
44 # the jno that does affect eflags, thus breaking the code.
45 #          movl w(%rbp), %eax  # load w
46 #          addl y(%rbp), %eax  # add y
47 #          movl %eax, y(%rbp)  # y = w + x
48 #####
49          movl w(%rbp), %eax  # load w
50          movl %eax, y(%rbp)  # y = w
51          movl x(%rbp), %eax  # load x
52          addl %eax, y(%rbp)  # y = w + x
53          jno nOver1        # skip warning if no OF
54          movl $warning, %edi  ##### changes edi
55          movl $0, %eax
56          call printf       ##### may change several registers
57 nOver1:
58          movl w(%rbp), %eax  # load w
59          movl %eax, z(%rbp)  # z = w
60          movl x(%rbp), %eax  # load x
61          subl %eax, z(%rbp) # z = w - x
```

```

62      jno    nOver2      # skip warning if no OF
63      movl    $warning, %edi
64      movl    $0, %eax
65      call    printf
66 nOver2:
67      movl    z(%rbp), %edx # load z
68      movl    y(%rbp), %esi # and y
69      movl    $display, %edi # display results
70      movl    $0, %eax      # no floats
71      call    printf
72
73      movl    $0, %eax      # return 0 to OS
74      movq    %rbp, %rsp    # restore stack pointer
75      popq    %rbp          # restore caller's base pointer
76      ret

```

---

## 9 -7 GAS LISTING Exercise\_9-6.s page 1

```

1      # Exercise_9-6.s
2      # This is not a program. It is a group of
3      # instructions to hand-assemble.
4      # Bob Plantz - 27 June 2009
5          .text
6          .globl main
7          main:
8 0000 55          pushq  %rbp
9 0001 4889E5      movq    %rsp, %rbp
10
11 0004 B9EFCDAB  movl    $0x89abcdef, %ecx   # a)
11     89
12 0009 66B8CDAB  movw    $0xabcd, %ax      # b)
13 000d B030          movb    $0x30, %al      # c)
14 000f B431          movb    $0x31, %ah      # d)
15 0011 4D89C7          movq    %r8, %r15     # e)
16 0014 4588CA          movb    %r9b, %r10b    # f)
17 0017 4589DC          movl    %r11d, %r12d    # g)
18 001a 48BEF42C          movq    $0x7fffec9b2cf4, %rsi # h)
18     9BECFF7F
18     0000
19
20 0024 B8000000  movl    $0, %eax
20     00
21 0029 4889EC  movq    %rbp, %rsp
22 002c 5D          popq    %rbp
23 002d C3          ret
24

```

---

**9 -8 GAS LISTING Exercise\_9-7.s page 1**

```

1          # Exercise_9-7.s
2          # This is not a program. It is a group of
3          # instructions to hand-assemble.
4          # Bob Plantz - 27 June 2009
5          .text
6          .globl main
7          main:
8 0000 55      pushq %rbp
9 0001 4889E5  movq %rsp, %rbp
10
11 0004 81C1EFCD    addl $0x89abcdef, %ecx # a)
11     AB89
12 000a 6605CDAB    addw $0xabcd, %ax # b)
13 000e 0430        addb $0x30, %al # c)
14 0010 80C431        addb $0x31, %ah # d)
15 0013 4D01E7        addq %r12, %r15 # e)
16 0016 664501C2    addw %r8w, %r10w # f)
17 001a 4400CE        addb %r9b, %sil # g)
18 001d 01F7        addl %esi, %edi # h)
19
20 001f B8000000    movl $0, %eax
20     00
21 0024 4889EC        movq %rbp, %rsp
22 0027 5D            popq %rbp
23 0028 C3            ret
24

```

**9 -9 GAS LISTING Exercise\_9-8.s page 1**

```

1          # Exercise_9-8.s
2          # This is not a program. It is an experiment
3          # to determine the machine code for pushl.
4          # Bob Plantz - 27 June 2009
5          .text
6          .globl main
7          main:
8 0000 55      pushq %rbp
9 0001 4889E5  movq %rsp, %rbp
10
11 0004 50      pushq %rax
12 0005 51      pushq %rcx
13 0006 52      pushq %rdx
14 0007 53      pushq %rbx
15 0008 56      pushq %rsi
16 0009 57      pushq %rdi
17 000a 4150      pushq %r8
18 000c 4151      pushq %r9

```

```

19 000e 4152      pushq   %r10
20 0010 4153      pushq   %r11
21 0012 4154      pushq   %r12
22 0014 4155      pushq   %r13
23 0016 4156      pushq   %r14
24 0018 4157      pushq   %r15
25
26 001a 415F      popq    %r15
27 001c 415E      popq    %r14
28 001e 415D      popq    %r13
29 0020 415C      popq    %r12
30 0022 415B      popq    %r11
31 0024 415A      popq    %r10
32 0026 4159      popq    %r9
33 0028 4158      popq    %r8
34 002a 5F         popq    %rdi
35 002b 5E         popq    %rsi
36 002c 5B         popq    %rbx
37 002d 5A         popq    %rdx
38 002e 59         popq    %rcx
39 002f 58         popq    %rax
40
41 0030 B8000000  movl    $0, %eax
41     00
42 0035 4889EC    movq    %rbp, %rsp
43 0038 5D         popq    %rbp
44 0039 C3         ret
45

```

**9 -10** See solution to Exercise 9

**9 -11** GAS LISTING Exercise\_9-10.s page 1

```

1          # Exercise_9-10.s
2          # This is not a program. I used the machine code fro
3          # listing to create Exercise 9-9.
4          # Uses a drill and kill approach to learning
5          # how to disassemble machine code
6          # Bob Plantz - 27 June 2009
7          .text
8          .globl main
9          main:
10 0000 55        pushq   %rbp
11 0001 4889E5    movq    %rsp, %rbp
12
13          #a
14 0004 B0AB      movb    $0xab, %al
15 0006 B4CD      movb    $0xcd, %ah
16 0008 41B0EF    movb    $0xef, %r8b
17 000b 41B701    movb    $0x01, %r15b

```

```

18
19          #b
20 000e 40B723      movb   $0x23, %dil
21 0011 40B634      movb   $0x34, %sil
22 0014 B256        movb   $0x56, %dl
23 0016 B678        movb   $0x78, %dh
24
25          #c
26 0018 B83412CD    movl   $0xabcd1234, %eax
26     AB
27 001d BBABCD12    movl   $0x3412cdab, %ebx
27     34
28 0022 41B90000    movl   $0x0, %r9d
28     0000
29 0028 41BE7B00    movl   $0x7b, %r14d
29     0000
30
31          #d
32 002e 66B8CDAB    movw   $0xabcd, %ax
33 0032 66BBBACD    movw   $0xcdba, %bx
34 0036 66B93412    movw   $0x1234, %cx
35 003a 66BA2143    movw   $0x4321, %dx
36
37          #e
38 003e 88C4        movb   %al, %ah
39 0040 88C8        movb   %cl, %al
40 0042 8808        movb   %cl, (%rax)
41 0044 88480A      movb   %cl, 10(%rax)
42 0047 8A08        movb   (%rax), %cl
43 0049 8A480A      movb   10(%rax), %cl
44
45          #f
46 004c 89C3        movl   %eax, %ebx
47 004e 6689D8      movw   %bx, %ax
48 0051 4889CA      movq   %rcx, %rdx
49 0054 4589C6      movl   %r8d, %r14d
50
51          #g
52 0057 04AB        addb   $0xab, %al
53 0059 80C4CD      addb   $0xcd, %ah
GAS LISTING Exercise_9-10.s page 2

```

```

54 005c 80C3EF      addb   $0xef, %bl
55 005f 80C701      addb   $0x01, %bh
56
57          #h
58 0062 80C123      addb   $0x23, %cl
59 0065 80C534      addb   $0x34, %ch
60 0068 80C256      addb   $0x56, %dl
61 006b 80C678      addb   $0x78, %dh
62

```

```

63          #i
64 006e 053412CD      addl    $0xabcd1234, %eax
64      AB
65 0073 81C3ABCD      addl    $0x3412cdab, %ebx
65      1234
66 0079 81C1D4C3      addl    $0xa1b2c3d4, %ecx
66      B2A1
67 007f 81C2A1B2      addl    $0xd4c3b2a1, %edx
67      C3D4
68
69          #o
70 0085 05AB0000      addl    $0xab, %eax
70      00
71 008a 83C301        addl    $0x1, %ebx
72 008d 83C100        addl    $0x0, %ecx
73 0090 81C2FF00      addl    $0xff, %edx
73      0000
74
75          #k
76 0096 6605CDAB      addw    $0xabcd, %ax
77 009a 6681C3BA      addw    $0xcdba, %bx
77      CD
78 009f 6681C134      addw    $0x1234, %cx
78      12
79 00a4 6681C221      addw    $0x4321, %dx
79      43
80
81          #l
82 00a9 6605AB00      addw    $0xab, %ax
83 00ad 6683C301      addw    $0x1, %bx
84 00b1 6683C100      addw    $0x0, %cx
85 00b5 6681C2FF      addw    $0xff, %dx
85      00
86
87          #m
88 00ba 00C4           addb    %al, %ah
89 00bc 4100C2         addb    %al, %r10b
90 00bf 00CA           addb    %cl, %dl
91 00c1 4500C1         addb    %r8b, %r9b
92
93          #n
94 00c4 01C3           addl    %eax, %ebx
95 00c6 6601D8           addw    %bx, %ax
96 00c9 4801CA           addq    %rcx, %rdx
97 00cc 4501C6           addl    %r8d, %r14d
98
99 00cf B8000000      movl    $0, %eax
99      00

```

GAS LISTING Exercise\_9-10.s page 3

```
100 00d4 4889EC      movq    %rbp, %rsp
```

101 00d7 5D	popq %rbp
102 00d8 C3	ret

---

## E.10 Program Flow Constructs

**10 -1**

instruction	n bytes	offset	total	decimal
7462	2	62	64	+100
749a	2	9a	9c	-100
0f8426010000	6	00000126	0000012c	+300
0f84cefeffff	6	fffffece	fffffed4	-300

**10 -2** Looking at the listing file:

```

18 0009 EB03      jmp    here1
19 000b 83F601    xorl   $1, %esi  # no jump, turn of bit
20          here1:           0
21 000e 488D0425  leaq   here2, %rax
21          00000000

```

the second byte in the `jmp here1` instruction is `03`, which is the number of bytes to the `here1` location.

Single-stepping through the program with `gdb` and examining the contents of `rax`, `rip`, and `pointer` shows that `jmp *%rax` and `jmp *pointer` use the full address, not just an offset.

**10 -3** The program will probably crash. When the `write` function is called, it returns the number of characters written. Return values are placed in `eax`. Hence, the address is overwritten. In general, it is safer to use variables in the stack frame if their values must remain the same after another function is called.

**10 -4**

```

1 # numerals.s
2 # Displays the numerals on screen
3 # Bob Plantz - 27 June 2009
4 # useful constant
5     .equ    STDOUT,1
6 # stack frame
7     .equ    theNumeral,-1
8     .equ    localSize,-16
9 # read only data
10    .section .rodata
11 newline:
12     .byte   '\n'
13 # code
14     .text
15     .globl  main
16     .type   main, @function

```

```

17 main:
18     pushq %rbp          # save caller's base pointer
19     movq %rsp, %rbp      # establish ours
20     addq $localSize, %rsp # local vars.
21
22     movb $'0', theNumeral(%rbp) # initial numeral
23 loop:
24     movl $1, %edx         # one character
25     leaq theNumeral(%rbp), %rsi # in this mem location
26     movl $STDOUT, %edi
27     call write
28
29     incb theNumeral(%rbp) # next char
30     cmpb $'9', theNumeral(%rbp) # over 9 yet?
31     jbe loop             # no, keep going
32
33 allDone:
34     movl $1, %edx         # do a newline for user
35     movl $newline, %esi
36     movl $STDOUT, %edi
37     call write
38
39     movl $0, %eax         # return 0;
40
41     movq %rbp, %rsp        # delete local vars.
42     popq %rbp             # restore caller's base pointer
43     ret                  # return to caller

```

**10 -5**

```

1 # alphaUpper.s
2 # Displays the upper case alphabet on screen
3 # Bob Plantz - 27 June 2009
4 # useful constant
5     .equ STDOUT,1
6 # stack frame
7     .equ theLetter,-1
8     .equ localSize,-16
9 # read only data
10    .section .rodata
11 newline:
12    .byte '\n'
13 # code
14    .text
15    .globl main
16    .type main, @function
17 main:
18     pushq %rbp          # save caller's base pointer
19     movq %rsp, %rbp      # establish ours
20     addq $localSize, %rsp # local vars.
21
22     movb $'A', theLetter(%rbp) # initial alpha
23 loop:
24     movl $1, %edx         # one character

```

```

25      leaq    theLetter(%rbp), %rsi # in this mem location
26      movl    $STDOUT, %edi
27      call    write
28
29      incb    theLetter(%rbp) # next char
30      cmpb    '$Z', theLetter(%rbp) # over Z yet?
31      jbe     loop      # no, keep going
32
33 allDone:
34      movl    $1, %edx      # do a newline for user
35      movl    $newline, %esi
36      movl    $STDOUT, %edi
37      call    write
38
39      movl    $0, %eax      # return 0;
40
41      movq    %rbp, %rsp    # delete local vars.
42      popq    %rbp        # restore caller's base pointer
43      ret     # return to caller

```

**10 -6**

```

1 # alphaLower.s
2 # Displays the lower case alphabet on screen
3 # Bob Plantz - 27 June 2009
4 # useful constant
5     .equ   STDOUT,1
6 # stack frame
7     .equ   theLetter,-1
8     .equ   localSize,-16
9 # read only data
10    .section .rodata
11 newline:
12     .byte  '\n'
13 # code
14     .text
15     .globl main
16     .type  main, @function
17 main:
18     pushq %rbp          # save caller's base pointer
19     movq  %rsp, %rbp    # establish ours
20     addq  $localSize, %rsp # local vars.
21
22     movb  '$a', theLetter(%rbp) # initial alpha
23 loop:
24     movl  $1, %edx      # one character
25     leaq   theLetter(%rbp), %rsi # in this mem location
26     movl   $STDOUT, %edi
27     call   write
28
29     incb   theLetter(%rbp) # next char
30     cmpb   '$z', theLetter(%rbp) # over z yet?
31     jbe    loop      # no, keep going
32

```

```

33 allDone:
34     movl    $1, %edx      # do a newline for user
35     movl    $newline, %esi
36     movl    $STDOUT, %edi
37     call    write
38
39     movl    $0, %eax      # return 0;
40
41     movq    %rbp, %rsp    # delete local vars.
42     popq    %rbp          # restore caller's base pointer
43     ret                 # return to caller

```

---

**10 -7**

```

1 /*
2 * whileLoop.c
3 * While loop multiplication.
4 *
5 * Bob Plantz - 27 June 2009
6 */
7
8 #include<stdio.h>
9
10 int main ()
11 {
12     int x, y, z;
13     int i;
14
15     printf("Enter two integers: ");
16     scanf("%i %i", &x, &y);
17     z = x;
18     i = 1;
19     while (i < y)
20     {
21         z += x;
22         i++;
23     }
24     printf("%i * %i = %i\n", x, y, z);
25     return 0;
26 }

```

---

With version 4.7.0 of gcc and no optimization (-O0), they both use the same assembly language for the loop:

```

28     jmp     .L2
29 .L3:
30     movl    -16(%rbp), %eax
31     addl    %eax, -8(%rbp)
32     addl    $1, -4(%rbp)
33 .L2:
34     movl    -12(%rbp), %eax
35     cmpl    %eax, -4(%rbp)
36     jl     .L3

```

**10 -8** After the program executes, the system prompt is displayed twice because the “return key” is still in the standard in buffer. This can be fixed by reading two characters.

```
1 /*  
2  * yesNola.c  
3  * Prompts user to enter a y/n response.  
4  *  
5  * Bob Plantz - 27 June 2009  
6 */  
7  
8 #include <unistd.h>  
9  
10 static char response[2];  
11  
12 int main(void)  
13 {  
14     register char *ptr;  
15  
16     ptr = "Save changes? ";  
17  
18     while (*ptr != '\0')  
19     {  
20         write(STDOUT_FILENO, ptr, 1);  
21         ptr++;  
22     }  
23  
24     read (STDIN_FILENO, response, 2);  
25  
26     if (*response == 'y')  
27     {  
28         ptr = "Changes saved.\n";  
29         while (*ptr != '\0')  
30         {  
31             write(STDOUT_FILENO, ptr, 1);  
32             ptr++;  
33         }  
34     }  
35     else  
36     {  
37         ptr = "Changes discarded.\n";  
38         while (*ptr != '\0')  
39         {  
40             write(STDOUT_FILENO, ptr, 1);  
41             ptr++;  
42         }  
43     }  
44     return 0;  
45 }
```

**10 -10**

```
1 # others.s  
2 # Displays all printable characters other than numerals
```

```
3 # and letters.
4 # Bob Plantz - 27 June 2009
5 # useful constants
6 .equ STDOUT,1
7 .equ SPACE, ' ' # lowest printable character
8 .equ SQUIGGLE, '~' # highest printable character
9 # stack frame
10 .equ theChar,-1
11 .equ localSize,-16
12 # read only data
13 .section .rodata
14 newline:
15 .byte '\n'
16 # code
17 .text
18 .globl main
19 .type main, @function
20 main:
21 pushq %rbp # save caller's base pointer
22 movq %rsp, %rbp # establish ours
23 addq $localSize, %rsp # local vars.
24
25 movb $SPACE, theChar(%rbp) # initial char
26 loop:
27 cmpb $SQUIGGLE, theChar(%rbp) # all chars?
28 ja allDone # yes, we're done
29
30 cmpb '$0', theChar(%rbp) # numeral?
31 jb print # no, print it
32 cmpb '$9', theChar(%rbp)
33 jbe noPrint # yes, don't print it
34 cmpb '$A', theChar(%rbp) # upper case?
35 jb print # no, print it
36 cmpb '$Z', theChar(%rbp)
37 jbe noPrint # yes, don't print it
38 cmpb '$a', theChar(%rbp) # lower case?
39 jb print # no, print it
40 cmpb '$z', theChar(%rbp)
41 jbe noPrint
42 print:
43 movl $1, %edx # one character
44 leaq theChar(%rbp), %rsi # in this mem location
45 movl $STDOUT, %edi # standard out
46 call write
47 noPrint:
48 incb theChar(%rbp) # next char
49 jmp loop # check at top of loop
50
51 allDone:
52 movl $1, %edx # do a newline for user
53 movl $newline, %esi
54 movl $STDOUT, %edi
```

```

55      call    write
56
57      movl    $0, %eax      # return 0;
58
59      movq    %rbp, %rsp    # delete local vars.
60      popq    %rbp        # restore caller's base pointer
61      ret                 # return to caller

```

---

**10 -11**

```

1 # incChars.s
2 # Prompts user to enter a text string, then changes each
3 # character to the next higher one.
4 # Bob Plantz - 27 June 2009
5 # useful constants
6      .equ    STDIN,0
7      .equ    STDOUT,1
8      .equ    SPACE,' '
9      .equ    SQUIGGLE,'~'   # highest printable character
10 # stack frame
11     .equ    theString,-256
12     .equ    localSize,-256
13 # read only data
14     .section .rodata
15 prompt:
16     .string "Enter a string of characters: "
17 msg:
18     .string "Incrementing each character: "
19 newline:
20     .byte   '\n'
21 # code
22     .text
23     .globl  main
24     .type   main, @function
25 main:
26     pushq  %rbp          # save caller's base pointer
27     movq   %rsp, %rbp    # establish ours
28     addq   $localSize, %rsp # local vars.
29
30     movl   $prompt, %esi  # prompt user
31 promptLup:
32     cmpb   $0, (%esi)    # end of string?
33     je     getString    # yes, get user input
34     movl   $1, %edx       # no, one character
35     movl   $STDOUT, %edi
36     call   write
37     incl   %esi          # next char
38     jmp    promptLup    # check at top of loop
39
40 getString:
41     leaq   theString(%rbp), %rsi # place to put user input
42     movl   $1, %edx        # one character
43     movl   $STDIN, %edi
44     call   read

```

```

45 readUp:
46     cmpb    '$\n', (%rsi)      # end of input?
47     je      incChars        # yes, process the string
48     incq    %rsi             # next char
49     movl    $1, %edx         # one character
50     movl    $STDIN, %edi
51     call    read
52     jmp    readUp          # check at top of loop
53
54 incChars:
55     movb    $0, (%rsi)      # null character for C string
56     leaq    theString(%rbp), %rsi # pointer to the string
57 incLoop:
58     cmpb    $0, (%rsi)      # end of string?
59     je      doDisplay       # yes, display the results
60     incb    (%rsi)          # change character
61     cmpb    $SQUIGGLE, (%rsi) # did we go too far?
62     jbe    okay            # no
63     movb    $SPACE, (%rsi)   # yes, wrap to beginning
64 okay:
65     incq    %rsi             # next char
66     jmp    incLoop          # check at top of loop
67
68 doDisplay:
69     movl    $msg, %esi       # print message for user
70 dispLoop:
71     cmpb    $0, (%esi)      # end of string?
72     je      showString      # yes, show results
73     movl    $1, %edx         # no, one character
74     movl    $STDOUT, %edi
75     call    write
76     incl    %esi             # next char
77     jmp    dispLoop         # check at top of loop
78
79 showString:
80     leaq    theString(%rbp), %rsi # pointer to the string
81 showLoop:
82     cmpb    $0, (%rsi)      # end of string?
83     je      allDone          # yes, get user input
84     movl    $1, %edx         # no, one character
85     movl    $STDOUT, %edi
86     call    write
87     incq    %rsi             # next char
88     jmp    showLoop          # check at top of loop
89
90 allDone:
91     movl    $1, %edx         # do a newline for user
92     movl    $newline, %esi
93     movl    $STDOUT, %edi
94     call    write
95
96     movl    $0, %eax          # return 0;

```

```

97      movq    %rbp, %rsp      # delete local vars.
98      popq    %rbp          # restore caller's base pointer
99      ret                 # return to caller

```

**10 -12**

```

1 # decChars.s
2 # Prompts user to enter a text string, then changes each
3 # character to the next lower one.
4 # Bob Plantz - 27 June 2009
5 # useful constants
6     .equ    STDIN,0
7     .equ    STDOUT,1
8     .equ    SPACE,' '
9     .equ    SQUIGGLE,'~' # highest printable character
10 # stack frame
11    .equ    theString,-256
12    .equ    localSize,-256
13 # read only data
14    .section .rodata
15 prompt:
16    .string "Enter a string of characters: "
17 msg:
18    .string "Decrementing each character: "
19 newline:
20    .byte   '\n'
21 # code
22    .text
23    .globl  main
24    .type   main, @function
25 main:
26    pushq   %rbp           # save caller's base pointer
27    movq    %rsp, %rbp      # establish ours
28    addq    $localSize, %rsp # local vars.
29
30    movl    $prompt, %esi    # prompt user
31 promptLup:
32    cmpb    $0, (%esi)      # end of string?
33    je    (getString)      # yes, get user input
34    movl    $1, %edx         # no, one character
35    movl    $STDOUT, %edi
36    call    write
37    incl    %esi            # next char
38    jmp     promptLup       # check at top of loop
39
40 getString:
41    leaq    theString(%rbp), %rsi # place to put user input
42    movl    $1, %edx            # one character
43    movl    $STDIN, %edi
44    call    read
45 readLup:
46    cmpb    '$\n', (%rsi)      # end of input?
47    je     decChars           # yes, process the string
48    incq    %rsi              # next char

```

```

49      movl    $1, %edx          # one character
50      movl    $STDIN, %edi
51      call    read
52      jmp     readUp         # check at top of loop
53
54 decChars:
55      movb    $0, (%rsi)       # null character for C string
56      leaq    theString(%rbp), %rsi # pointer to the string
57 decLoop:
58      cmpb    $0, (%rsi)       # end of string?
59      je      doDisplay       # yes, display the results
60      decb    (%rsi)
61      cmpb    $SPACE, (%rsi)   # did we go too far?
62      jae    okay            # no
63      movb    $SQUIGGLE, (%rsi) # yes, wrap to beginning
64 okay:
65      incq    %rsi             # next char
66      jmp     decLoop        # check at top of loop
67
68 doDisplay:
69      movl    $msg, %esi        # print message for user
70 dispLoop:
71      cmpb    $0, (%esi)       # end of string?
72      je      showString      # yes, show results
73      movl    $1, %edx
74      movl    $STDOUT, %edi
75      call    write
76      incl    %esi             # next char
77      jmp     dispLoop       # check at top of loop
78
79 showString:
80      leaq    theString(%rbp), %rsi # pointer to the string
81 showLoop:
82      cmpb    $0, (%rsi)       # end of string?
83      je      allDone          # yes, get user input
84      movl    $1, %edx
85      movl    $STDOUT, %edi
86      call    write
87      incq    %rsi             # next char
88      jmp     showLoop        # check at top of loop
89
90 allDone:
91      movl    $1, %edx          # do a newline for user
92      movl    $newline, %esi
93      movl    $STDOUT, %edi
94      call    write
95
96      movl    $0, %eax          # return 0;
97      movq    %rbp, %rsp
98      popq    %rbp             # restore caller's base pointer
99      ret                  # return to caller

```

**10 -13**

```
1 # echoN.s
2 # Prompts user to enter a single character.
3 # The character is echoed. If it is a numeral, say N,
4 # it is echoed N+1 times
5 # Bob Plantz - 27 June 2009
6 # useful constants
7     .equ    STDIN,0
8     .equ    STDOUT,1
9 # stack frame
10    .equ    count,-8
11    .equ    response,-4
12    .equ    localSize,-16
13 # read only data
14     .section .rodata
15 instruct:
16     .ascii  "A single numeral, N, is echoed N+1 times, other charac-
17     .asciz  "are\n echoed once. 'q' ends program.\n\n"
18 prompt:
19     .string "Enter a single character: "
20 msg:
21     .string "You entered: "
22 bye:
23     .string "End of program.\n"
24 newline:
25     .byte   '\n'
26 # code
27     .text
28     .globl  main
29     .type   main, @function
30 main:
31     pushq   %rbp          # save caller's base pointer
32     movq    %rsp, %rbp      # establish ours
33     addq    $localSize, %rsp # local vars
34
35     movl    $instruct, %esi  # instruct user
36 instructLup:
37     cmpb    $0, (%esi)      # end of string?
38     je      runLoop        # yes, run program
39     movl    $1, %edx         # no, one character
40     movl    $STDOUT, %edi
41     call    write
42     incl    %esi            # next char
43     jmp    instructLup     # check at top of loop
44
45 runLoop:
46     movl    $prompt, %esi   # prompt user
47 promptLup:
48     cmpb    $0, (%esi)      # end of string?
49     je      getChar        # yes, get user input
50     movl    $1, %edx         # no, one character
51     movl    $STDOUT, %edi
```

```
52         call    write
53         incl    %esi          # next char
54         jmp     promptLup   # check at top of loop
55
56 getChar:
57         leaq    response(%rbp), %rsi # place to put user input
58         movl    $2, %edx        # include newline
59         movl    $STDIN, %edi
60         call    read
61
62         movb    response(%rbp), %al  # get input character
63         cmpb    $'q', %al       # if 'q'
64         je      allDone        # end program
65 # Otherwise, set up count loop
66         movl    $1, count(%rbp) # assume not numeral
67         cmpb    $'0', %al       # check for numeral
68         jb      echoLoop
69         cmpb    $'9', %al
70         ja      echoLoop
71         andl    $0xf, %eax      # numeral, convert to int
72         incl    %eax          # echo N+1 times
73         movl    %eax, count(%rbp) # save counter
74 echoLoop:
75         movl    $msg, %esi      # pointer to the string
76 msgLoop:
77         cmpb    $0, (%esi)      # end of string?
78         je      doChar        # yes, show character
79         movl    $1, %edx
80         movl    $STDOUT, %edi
81         call    write
82         incl    %esi          # next char
83         jmp     msgLoop       # check at top of loop
84
85 doChar:
86         movl    $1, %edx      # one character
87         leaq    response(%rbp), %rsi # in this mem location
88         movl    $STDOUT, %edi
89         call    write
90
91         movl    $1, %edx      # and a newline
92         movl    $newline, %esi
93         movl    $STDOUT, %edi
94         call    write
95
96         decl    count(%rbp)    # count--
97         jne     echoLoop      # continue if more to do
98         jmp     runLoop        # else get next character
99
100 allDone:
101        movl    $bye, %esi      # ending message
102 doneLup:
103        cmpb    $0, (%esi)      # end of string?
```

```

104      je      cleanUp          # yes, get user input
105      movl    $1, %edx        # no, one character
106      movl    $STDOUT, %edi
107      call    write
108      incl    %esi           # next char
109      jmp     doneUp         # check at top of loop
110
111  cleanUp:
112      movl    $0, %eax        # return 0;
113      movq    %rbp, %rsp        # delete local vars.
114      popq    %rbp           # restore caller's base pointer
115      ret                 # return to caller

```

---

## E.11 Writing Your Own Functions

### 11 -3

```

1 # helloworld.s
2 # Hello world program to test writeStr function
3 # Bob Plantz - 27 June 2009
4
5 hiworld:
6     .string "Hello, world!\n"
7
8     .text
9     .globl main
10
11 main:
12     pushq  %rbp            # save caller base pointer
13     movq    %rsp, %rbp        # establish our base pointer
14
15     movl    $hiworld, %edi   # address of string to print
16     call    writeStr         # write it
17
18     movl    $0, %eax          # return 0;
19     movq    %rbp, %rsp        # delete local variables
20     popq    %rbp           # restore caller base pointer
21     ret

```

---

```

1 # writeStr.s
2 # Writes a C-style text string to the standard output (screen).
3 # Bob Plantz - 27 June 2009
4
5 # Calling sequence:
6 #         rdi <- address of string to be written
7 #         call    writestr
8 # returns number of characters written
9
10 # Useful constant
11     .equ    STDOUT,1
12 # Stack frame, showing local variables and arguments
13     .equ    stringAddr,-16

```

```

14      .equ    count, -4
15      .equ    localSize, -16
16
17      .text
18      .globl  writeStr
19      .type   writeStr, @function
20 writeStr:
21      pushq   %rbp          # save base pointer
22      movq   %rsp, %rbp     # new base pointer
23      addq   $localSize, %rsp # local vars. and arg.
24
25      movq   %rdi, stringAddr(%rbp) # save string pointer
26      movl   $0, count(%rbp) # count = 0;
27 writeLoop:
28      movq   stringAddr(%rbp), %rax # get current pointer
29      cmpb   $0, (%rax)      # at end yet?
30      je     done           # yes, all done
31
32      movl   $1, %edx        # no, write one character
33      movq   %rax, %rsi       # points to current char
34      movl   $STDOUT, %edi     # on the screen
35      call   write
36      incl   count(%rbp)    # count++;
37      incl   stringAddr(%rbp) # stringAddr++;
38      jmp    writeLoop      # and check for end
39 done:
40      movl   count(%rbp), %eax # return count
41      movq   %rbp, %rsp       # restore stack pointer
42      popq   %rbp            # restore base pointer
43      ret

```

---

**11 -4**

```

1 # echoString.s
2 # Prompts user to enter a string, then echoes it.
3 # Bob Plantz - 27 June 2009
4 # stack frame
5      .equ    theString, -256
6      .equ    localSize, -256
7 # read only data
8      .data
9 usrprmppt:
10     .string "Enter a text string:\n"
11 usrmsg:
12     .string "You entered:\n"
13 newline:
14     .string "\n"
15 # code
16     .text
17     .globl  main
18     .type   main, @function
19 main:
20     pushq   %rbp          # save caller base pointer
21     movq   %rsp, %rbp      # establish our base pointer

```

```

22      addq    $localSize, %rsp  # local vars.
23
24      movl    $usrprmpt, %edi   # tell user what to do
25      call    writeStr
26
27      leaq    theString(%rbp), %rdi # place for user response
28      call    readLn
29
30      movl    $usrmsg, %edi     # echo for user
31      call    writeStr
32      leaq    theString(%rbp), %rdi
33      call    writeStr
34
35      movl    $newline, %edi    # some formatting for user
36      call    writeStr
37
38      movl    $0, %eax          # return 0;
39      movq    %rbp, %rsp        # delete local variables
40      popq    %rbp             # restore caller base pointer
41      ret

```

---

```

1 # readLnSimple.s
2 # Reads a line (through the '\n' character from standard input. Deletes
3 # the '\n' and creates a C-style text string.
4 # Bob Plantz - 27 June 2009
5
6 # Calling sequence:
7 #         rdi <- address of place to store string
8 #         call    readLn
9 # returns number of characters read (not including NUL)
10
11 # Useful constant
12     .equ    STDIN,0
13 # Stack frame, showing local variables and arguments
14     .equ    stringAddr,-16
15     .equ    count,-4
16     .equ    localSize,-16
17
18     .text
19     .globl  readLn
20     .type   readLn, @function
21 readLn:
22     pushq   %rbp           # save base pointer
23     movq    %rsp, %rbp       # new base pointer
24     addq    $localSize, %rsp # local vars. and arg.
25
26     movq    %rdi, stringAddr(%rbp) # save string pointer
27     movl    $0, count(%rbp)      # count = 0;
28
29     movl    $1, %edx          # read one character
30     movq    stringAddr(%rbp), %rsi # into storage area
31     movl    $STDIN, %edi       # from keyboard

```

```

32          call    read
33 readLoop:
34         movq   stringAddr(%rbp), %rax # get pointer
35         cmpb   $'\n', (%rax)        # return key?
36         je     endOfString       # yes, mark end of string
37         incq   stringAddr(%rbp) # no, move pointer to next byte
38         incl   count(%rbp)      # count++;
39         movl   $1, %edx        # get another character
40         movq   stringAddr(%rbp), %rsi # into storage area
41         movl   $STDIN, %edi      # from keyboard
42         call   read
43         jmp    readLoop        # and look at it
44
45 endOfString:
46         movq   stringAddr(%rbp), %rax # current pointer
47         movb   $0, (%rax)        # mark end of string
48
49         movl   count(%rbp), %eax # return count;
50         movq   %rbp, %rsp        # restore stack pointer
51         popq   %rbp            # restore base pointer
52         ret                 # back to OS

```

See above for `writeStr`.

**11 -5** Note: Some students will try to create a nested loop, the outer one being executed twice. But the display messages are not nearly as nice, unless the student uses some "goto" statements. In my opinion, two separate change case loops is better software engineering because it allows maximum flexibility in the user messages. The user will generally complain about what is seen on the screen, not the cleverness of the code.

```

1 # changeCase.s
2 # Prompts user to enter a string, echoes it, changes case of alpha
3 # characters, displays them, changes them back, then displays result.
4 # Bob Plantz - 27 June 2009
5
6 # Stack frame
7         .equ    response,-256
8         .equ    localSize,-256
9         .data
10 usrprmppt:
11         .string "Enter a text string:\n"
12 usrmsg:
13         .string "You entered:\n"
14 chngmsg:
15         .string "Changing the case gives:\n"
16 newline:
17         .string "\n"
18
19         .text
20         .globl  main
21         .type   main, @function
22 main:

```

```
23      pushq %rbp          # save caller base pointer
24      movq %rsp, %rbp      # establish our base pointer
25      addq $localSize, %rsp # local vars
26
27      movl $usrprmp, %edi  # tell user what to do
28      call writeStr
29
30      movl $256, %esi       # max number of chars
31      leaq response(%rbp), %rdi # place to store them
32      call readLn
33
34      movl $usrmsg, %edi    # echo for usr
35      call writeStr
36
37      leaq response(%rbp), %rdi
38      call writeStr
39
40      movl $newline, %edi   # some formatting for user
41      call writeStr
42
43      leaq response(%rbp), %rax  # address of user's text string
44 changeCaseLup:
45      cmpb $0, (%rax)        # end of string
46      je showChange          # yes, show what we've done
47      cmpb $'A', (%rax)       # no, see if it's an alpha character
48      jb notAlpha            # lower than 'A'
49      cmpb $'Z', (%rax)       # check if it's upper case
50      jbe isAlpha             # it is
51      cmpb $'a', (%rax)       # now check lower case range
52      jb notAlpha            # notAlpha
53      cmpb $'z', (%rax)
54      ja notAlpha
55 isAlpha:
56      xorb $0x20, (%rax)      # flip the case bit
57 notAlpha:
58      incq %rax              # next character
59      jmp changeCaseLup      # and check for end to string
60
61 showChange:
62      movl $chngmsg, %edi     # tell user about it
63      call writeStr
64
65      leaq response(%rbp), %rdi # show the changes
66      call writeStr
67
68      movl $newline, %edi      # some formatting for user
69      call writeStr
70
71      leaq response(%rbp), %rax  # address of user's text string
72 restoreLup:
73      cmpb $0, (%rax)        # end of string
74      je showOrig             # yes, we're back to original
```

```

75      cmpb    $'A', (%rax)      # no, see if it's an alpha character
76      jb     notLetter       # lower than 'A'
77      cmpb    $'Z', (%rax)      # check if it's upper case
78      jbe     isLetter        # it is
79      cmpb    $'a', (%rax)      # now check lower case range
80      jb     notLetter       # flip the case bit
81      cmpb    $'z', (%rax)
82      ja     notLetter
83 isLetter:
84      xorb    $0x20, (%rax)
85 notLetter:
86      incq    %rax           # next character
87      jmp     restoreLup     # and check for end to string
88
89 showOrig:
90      movl    $usrmsg, %edi      # show original version
91      call    writeStr
92
93      leaq    response(%rbp), %rdi # should be restored
94      call    writeStr
95
96      movl    $newline, %edi      # some formatting for user
97      call    writeStr
98
99      movl    $0, %eax          # return 0;
100     movq    %rbp, %rsp          # delete local variables
101     popq    %rbp             # restore caller base pointer
102     ret

```

See above for `writeStr` and `readLn`.

## 11 -6

```

1 # echoString2.s
2 # Prompts user to enter a string, then echoes it.
3 # Bob Plantz - 27 June 2009
4 # stack frame
5     .equ    theString,-256
6     .equ    localSize,-256
7 # Length of the array. Do not make this larger than 255.
8 # I have used a small number to test readLn for removing
9 # extra characters from the keyboard buffer.
10    .equ    arrayLength,4
11 # read only data
12    .data
13 usrprmppt:
14     .string "Enter a text string:\n"
15 usrmsg:
16     .string "You entered:\n"
17 newline:
18     .string "\n"
19 # code
20     .text
21     .globl  main

```

22 main:

```

23      pushq  %rbp          # save caller base pointer
24      movq  %rsp, %rbp    # establish our base pointer
25      addq  $localSize, %rsp # local vars.

26
27      movl  $usrprmppt, %edi # tell user what to do
28      call  writeStr

29
30      movl  $arrayLength, %esi # "length" of array
31      leaq  theString(%rbp), %rdi # place for user response
32      call  readLn

33
34      movl  $usrmmsg, %edi   # echo for user
35      call  writeStr
36      leaq  theString(%rbp), %rdi
37      call  writeStr

38
39      movl  $newline, %edi   # some formatting for user
40      call  writeStr

41
42      movl  $0, %eax        # return 0;
43      movq  %rbp, %rsp      # delete local variables
44      popq  %rbp            # restore caller base pointer
45      ret

```

---

1 # readLn.s

```

2 # Reads a line (through the '\n' character from standard input. Deletes
3 # the '\n' and creates a C-style text string.
4 # Bob Plantz - 27 June 2009

5
6 # Calling sequence:
7 #     rsi <- length of char array
8 #     rdi <- address of place to store string
9 #     call    readLn
10 # returns number of characters read (not including NUL)

11
12 # Useful constant
13     .equ    STDIN,0
14 # Stack frame, showing local variables and arguments
15     .equ    maxLength,-24
16     .equ    stringAddr,-16
17     .equ    count,-4
18     .equ    localSize,-32

19
20     .text
21     .globl  readLn
22     .type   readLn, @function

23 readLn:
24     pushq  %rbp          # save base pointer
25     movq  %rsp, %rbp    # new base pointer
26     addq  $localSize, %rsp # local vars. and arg.

```

27

```

28      movq    %rsi, maxLength(%rbp) # save max storage space
29      movq    %rdi, stringAddr(%rbp) # save string pointer
30
31      movl    $0, count(%rbp)      # count = 0;
32      subq    $1, maxLength(%rbp) # leave room for NUL char
33
34      movl    $1, %edx          # read one character
35      movq    stringAddr(%rbp), %rsi # into storage area
36      movl    $STDIN, %edi        # from keyboard
37      call    read
38 readLoop:
39      movq    stringAddr(%rbp), %rax # get pointer
40      cmpb    $'\n', (%rax)       # return key?
41      je     endOfString         # yes, mark end of string
42      movl    count(%rbp), %eax   # current count
43      cmpl    %eax, maxLength(%rbp) # is caller's array full?
44      je     skipStore          # yes, store any more chars
45
46      incq    stringAddr(%rbp) # no, move pointer to next byte
47      incl    count(%rbp)       # count++;
48 skipStore:
49      movl    $1, %edx          # get another character
50      movq    stringAddr(%rbp), %rsi # into storage area
51      movl    $STDIN, %edi        # from keyboard
52      call    read
53      jmp    readLoop          # and look at it
54
55 endOfString:
56      movq    stringAddr(%rbp), %rax # current pointer
57      movb    $0, (%rax)           # mark end of string
58
59      movl    count(%rbp), %eax   # return count;
60      movq    %rbp, %rsp          # restore stack pointer
61      popq    %rbp              # restore base pointer
62      ret                  # back to OS

```

See above for writeStr.

## E.12 Bit Operations; Multiplication and Division

### 12 -1

```

1 # binary2int.s
2 # Prompts the user to enter an integer in binary, then displays
3 # it in decimal.
4 # Bob Plantz - 12 June 2008
5
6 # Stack frame
7      .equ    theInt,-40
8      .equ    buffer,-36
9      .equ    localSize,-48
10 # Read only data
11      .section .rodata

```

```

12 prompt:
13         .asciz "Please enter an integer in binary: "
14 displayFmt:
15         .asciz "In decimal: %d\n"
16 # Code
17         .text
18         .globl main
19         .type main, @function
20 main:
21         pushq %rbp          # save frame pointer
22         movq %rsp, %rbp      # new frame pointer
23         addq $localSize, %rsp # local vars.
24
25 # Tell user what to do.
26         movl $prompt, %edi    # prompt user
27         call writeStr
28
29 # Get binary number
30         movl $36, %esi        # max number of chars
31         leaq buffer(%rbp), %rax # place for user input
32         movq %rax, %rdi
33         call readLn          # get user input string
34
35 # convert text string from zeros and ones to int format
36         leaq buffer(%rbp), %rdi # start of string
37         movl $0, %esi           # use for int
38 convertloop:
39         movb (%rdi), %al       # get char
40         cmpb $0, %al           # null char?
41         je done_convert        # yes, done with conversion
42         andb $0x0f, %al         # no, convert char to 8-bit byte
43         shll $1, %esi           # make room for it
44         orb %al, %sil          # add it in
45         incq %rdi              # next char
46         jmp convertloop        # and do the next one
47 done_convert:
48         movl %esi, theInt(%rbp) # store result
49
50 # display in decimal
51         movl theInt(%rbp), %esi # int to display
52         movl $displayFmt, %edi  # format string
53         movl $0, %eax
54         call printf
55
56         movl $0, %eax           # return 0
57         movq %rbp, %rsp          # restore stack pointer
58         popq %rbp               # restore frame pointer
59         ret                      # back to OS

```

See Section E.11 for `writeStr` and `readLn`.

```
2 # Converts decimal int to binary
3 # Bob Plantz - 27 June 2009
4
5 # Stack frame
6     .equ    myInt,-44
7     .equ    counter,-40
8     .equ    buffer,-36
9     .equ    localSize,-48
10 # Read only data
11     .section .rodata
12 prompt:
13     .string "Enter an integer: "
14 format:
15     .string "%i"
16 msg1:
17     .string "The stored number is "
18 msg2:
19     .string " in binary.\n"
20 # Code
21     .text
22     .globl main
23     .type main, @function
24 main:
25     pushq %rbp          # save frame pointer
26     movq %rsp, %rbp      # new frame pointer
27     addq $localSize, %rsp # local vars.
28
29     movl $prompt, %edi    # prompt user
30     call writeStr
31     leaq myInt(%rbp), %rsi # get user's int
32     movl $format, %edi
33     movl $0, %eax
34     call scanf
35
36 # Generate text string of ones and zeros
37     leaq buffer(%rbp), %rax # place for text string
38     movl $32, counter(%rbp) # 32 bits
39 convertloop:
40     shll $1, myInt(%rbp) # move high order bit to CF
41     jnc zero             # it's zero
42     movb '$1', (%rax)    # one character
43     jmp cont              # go on
44 zero:
45     movb '$0', (%rax)    # zero character
46 cont:
47     incq %rax            # next char position
48     decl counter(%rbp)   # counter--
49     jg convertloop        # keep going until counter == 0
50
51     movb $0, (%rax)      # store NULL
52
53 # display in binary
```

```

54      movl    $msg1, %edi      # nice message for user
55      call    writeStr
56
57      leaq    buffer(%rbp), %rdi # address of our string
58      call    writeStr
59
60      movl    $msg2, %edi      # nice message for user
61      call    writeStr
62
63      movl    $0, %eax        # return 0
64      movq    %rbp, %rsp      # restore stack pointer
65      popq    %rbp          # restore frame pointer
66      ret                 # back to OS

```

---

See Section E.11 for `writeStr`.

### 12 -3

```

1 # multiply.s
2 # Gets two 16-bit integers from user and computes their product.
3 # Bob Plantz - 27 June 2009
4
5 # Stack frame
6     .equ    multiplier,-8
7     .equ    multiplicand,-4
8     .equ    localSize,-16
9 # Read only data
10    .section .rodata
11 prompt:
12     .string "Enter an integer (0 - 65535): "
13 printfmat:
14     .string "%hu times %hu = %u\n"
15 scanfmat:
16     .string "%hu"
17 # Code
18     .text
19     .globl  main
20     .type   main, @function
21 main:
22     pushq   %rbp          # save frame pointer
23     movq    %rsp, %rbp      # new frame pointer
24     addq    $localSize, %rsp # local vars.
25
26 # prompt user
27     movl    $prompt, %edi    # message address
28     movl    $0, %eax
29     call    printf
30
31 # get first integer
32     leaq    multiplicand(%rbp), %rsi # place to store it
33     movl    $scanfmat, %edi # scanf formatting string
34     movl    $0, %eax
35     call    scanf
36

```

```

37 # get second integer
38     movl    $prompt, %edi      # message address
39     movl    $0, %eax
40     call    printf
41
42     leaq    multiplier(%rbp), %rsi # place to store it
43     movl    $scanfformat, %edi # scanf formatting string
44     movl    $0, %eax
45     call    scanf
46
47 # now multiply them
48     movw    multiplier(%rbp), %si # pass by value
49     movw    multiplicand(%rbp), %di
50     call    mul16
51
52 # at this point, the 32-bit result is in eax
53     movl    %eax, %ecx          # result
54     movl    multiplier(%rbp), %edx # one number
55     movl    multiplicand(%rbp), %esi # other number
56     movl    $printfformat, %edi # printf formatting string
57     movl    $0, %eax
58     call    printf
59
60     movl    $0, %eax          # return 0
61     movq    %rbp, %rsp        # restore stack pointer
62     popq    %rbp            # restore base pointer
63     ret                 # back to OS

```

---

```

1 # mul16.s
2 # Multiplies two 16-bit integers and returns 32-bit result
3 # Bob Plantz - 27 June 2009
4
5 # Calling sequence
6 #           si <- multiplier
7 #           di <- multiplicand
8 #           call mul16
9 #Code
10    .text
11    .globl  mul16
12    .type   mul16, @function
13 mul16:
14    pushq   %rbp             # save frame pointer
15    movq    %rsp, %rbp       # new frame pointer
16
17    movw    %si, %ax          # move for multiply
18    mulw    %di
19
20    sal     $16, %edx         # shift high-order part of answer
21                                # into high-order part of the register
22    andl    $0xffff, %eax     # make sure high-order part of eax is c
23    orl     %edx, %eax        # make 32-bit result for return
24

```

```
25      movq    %rbp, %rsp      # restore stack pointer
26      popq    %rbp          # restore frame pointer
27      ret                 # back to caller
```

---

**12 -4**

```
1 # divide.s
2 # Gets two 32-bit integers from user and computes quotient
3 # of the first divided by the second.
4 # Bob Plantz - 27 June 2009
5
6 # Stack frame
7     .equ    divisor,-8
8     .equ    dividend,-4
9     .equ    localSize,-16
10 # Read only data
11     .section .rodata
12 prompt:
13     .asciz "Enter an integer (0 - 4294967295): "
14 printfformat:
15     .asciz "%u div %u = %u\n"
16 scanformat:
17     .asciz "%u"
18 # Code
19     .text
20     .globl main
21     .type main, @function
22 main:
23     pushq %rbp           # save frame pointer
24     movq  %rsp, %rbp      # new frame pointer
25     addq  $divisor, %rsp  # local vars.
26
27 # prompt user
28     movl  $prompt, %edi   # message address
29     movl  $0, %eax
30     call  printf
31
32 # get first integer
33     leaq   dividend(%rbp), %rsi  # place to store it
34     movl   $scanformat, %edi    # scanf formatting string
35     movl   $0, %eax
36     call   scanf
37
38 # get second integer
39     movl  $prompt, %edi   # message address
40     movl  $0, %eax
41     call  printf
42
43     leaq   divisor(%rbp), %rsi  # place to store it
44     movl   $scanformat, %edi    # scanf formatting string
45     movl   $0, %eax
46     call   scanf
47
48 # now divide them
```

```

49      movl    divisor(%rbp), %esi # pass by value
50      movl    dividend(%rbp), %edi
51      call    div32
52
53 # at this point, the 32-bit result is in eax
54      movl    %eax, %ecx      # result
55      movl    divisor(%rbp), %edx # numerator
56      movl    dividend(%rbp), %esi # denominator
57      movl    $printfformat, %edi # printf formatting string
58      movl    $0, %eax
59      call    printf
60
61      movl    $0, %eax        # return 0
62      movq    %rbp, %rsp      # restore stack pointer
63      popq    %rbp          # restore frame pointer
64      ret                 # back to OS

```

---

```

1 # div32.s
2 # divides two 32-bit integers and returns 32-bit quotient
3 # Bob Plantz - 27 June 2009
4
5 # Calling sequence
6 #       esi <- divisor
7 #       edi <- dividend
8 #       call div32
9 # Code
10      .text
11      .globl  div32
12      .type   div32, @function
13 div32:
14      pushq   %rbp        # save base pointer
15      movq    %rsp, %rbp  # new base pointer
16
17      movl    $0, %edx     # clear for divide
18      movl    %edi, %eax
19      divl    %esi        # div is in eax
20
21      movq    %rbp, %rsp  # restore stack pointer
22      popq    %rbp        # restore base pointer
23      ret                 # return quotient

```

---

**12 -5**

```

1 # modulo.s
2 # Gets two 32-bit integers from user and computes remainder
3 # of the first divided by the second.
4 # Bob Plantz - 27 June 2009
5
6 # Stack frame
7      .equ    divisor,-8
8      .equ    dividend,-4
9      .equ    localSize,-16
10 # Read only data

```

```
11         .section  .rodata
12 prompt:    .asciz  "Enter an integer (0 - 4294967295): "
13 printformat: .asciz  "%u mod %u = %u\n"
14 scanformat: .asciz  "%u"
15
16 # Code
17         .text
18         .globl  main
19         .type   main, @function
20
21 main:
22     pushq   %rbp          # save frame pointer
23     movq    %rsp, %rbp      # new frame pointer
24     addq    $localSize, %rsp # local vars.
25
26
27 # prompt user
28     movl    $prompt, %edi   # message address
29     movl    $0, %eax
30     call    printf
31
32 # get first integer
33     leaq    dividend(%rbp), %rsi  # place to store it
34     movl    $scanformat, %edi    # scanf formatting string
35     movl    $0, %eax
36     call    scanf
37
38 # get second integer
39     movl    $prompt, %edi   # message address
40     movl    $0, %eax
41     call    printf
42
43     leaq    divisor(%rbp), %rsi  # place to store it
44     movl    $scanformat, %edi    # scanf formatting string
45     movl    $0, %eax
46     call    scanf
47
48 # now divide them
49     movl    divisor(%rbp), %esi  # pass by value
50     movl    dividend(%rbp), %edi
51     call    div32
52
53 # at this point, the 32-bit result is in eax
54     movl    %eax, %ecx        # result
55     movl    divisor(%rbp), %edx # numerator
56     movl    dividend(%rbp), %esi # denominator
57     movl    $printformat, %edi  # printf formatting string
58     movl    $0, %eax
59     call    printf
60
61     movl    $0, %eax          # return 0
62     movq    %rbp, %rsp        # restore stack pointer
```

---

```

63      popq    %rbp          # restore frame pointer
64      ret           # back to OS


---


1 # mod32.s
2 # divides two 32-bit integers and returns 32-bit remainder
3 # Bob Plantz - 27 June 2009
4
5 # Calling sequence
6 #       esi <- divisor
7 #       edi <- dividend
8 #       call div32
9 # Code
10     .text
11     .globl  div32
12     .type   div32, @function
13 div32:
14     pushq   %rbp          # save base pointer
15     movq    %rsp, %rbp    # new base pointer
16
17     movl    $0, %edx      # clear for divide
18     movl    %edi, %eax
19     divl    %esi          # remainder is in edx
20     movl    %edx, %eax    # return remainder
21
22     movq    %rbp, %rsp    # restore stack pointer
23     popq    %rbp          # restore base pointer
24     ret                 # return quotient

```

---

**12 -6**


---

```

1 # decimal2unt.s
2 # Prompts the user to enter an integer in decimal, then converts
3 # it to int format.
4 # Bob Plantz - 27 June 2009
5
6 # Constant
7     .equ    bufferSize,12
8
9 # Stack frame
10    .equ    buffer,-16
11    .equ    theInt,-4
12    .equ    localSize,-16
13
14 # Read only data
15    .section .rodata
16 prompt:   .asciz  "Please enter an integer in decimal: "
17 format:    .asciz  "You entered %i\n"
18
19 # Code
20     .text
21     .globl  main

```

---

```

24      .type   main, @function
25 main:
26      pushq   %rbp          # save frame pointer
27      movq    %rsp, %rbp    # new frame pointer
28      addq    $localSize, %rsp # local vars.
29
30 # Tell user what to do.
31      movq    $prompt, %rdi   # prompt user
32      call    writeStr
33
34 # Get decimal number
35      movl    $buffSize, %esi  # max number of chars
36      leaq    buffer(%rbp), %rdi # place for user input
37      call    readLn         # get user input string
38
39 # convert the string to int format
40      leaq    theInt(%rbp), %rsi # place to store the int
41      leaq    buffer(%rbp), %rdi # user's string
42      call    dec2uInt
43
44      movl    theInt(%rbp), %esi # display results
45      movq    $format, %rdi
46      movl    $0, %eax
47      call    printf
48
49      movl    $0, %eax        # return 0
50      movq    %rbp, %rsp      # restore stack pointer
51      popq    %rbp          # restore base pointer
52      ret                  # back to OS

```

```

1 # dec2uInt.s
2 # Converts string of numerals to decimal unsigned int
3 # Bob Plantz - 13 June 2009
4
5 # Calling sequence
6 #     rsi <- address of place to store the int
7 #     rdi <- address of string
8 #     call dec2uInt
9 #     returns 0
10 # Code
11      .text
12      .globl  dec2uInt
13      .type   dec2uInt, @function
14 dec2uInt:
15      pushq   %rbp          # save caller frame ptr
16      movq    %rsp, %rbp    # our stack frame
17
18      movl    $0, %eax       # subtotal = 0
19 loop:
20      movb    (%rdi), %cl    # get current character
21      cmpb    $0, %cl        # end of string?
22      je      done           # yes, all done

```

```

23      andl    $0xf, %ecx      # no, convert char to int
24      imull   $10, %eax      # 10 x subtotal
25      addl    %ecx, %eax      # add current int to subtotal
26      incq    %rdi          # move pointer
27      jmp     loop          # and check again
28 done:
29      movl    %eax, (%rsi)    # store the int
30
31      movl    $0, %eax        # return 0
32      movq    %rbp, %rsp      # delete local vars
33      popq    %rbp          # restore caller frame ptr
34      ret

```

---

See Section E.11 for `writeStr` and `readLn`.

## 12 -7

```

1 # addConstant.s
2 # Prompts the user to enter an integer in decimal, converts
3 # it to int format, adds a constant, then displays result.
4 # Bob Plantz - 28 June 2009
5
6 # useful constant
7 theConstant = 12345
8
9 # Stack frame
10     .equ    theInt,-16
11     .equ    buffer,-12
12     .equ    localSize,-16
13 # Read only data
14     .section .rodata
15 prompt:
16     .asciz "Please enter an integer in decimal: "
17 msg:
18     .asciz "The result is: "
19 endl:
20     .asciz "\n"
21 # Code
22     .text
23     .globl main
24     .type main, @function
25 main:
26     pushq %rbp          # save frame pointer
27     movq %rsp, %rbp      # new frame pointer
28     addq $localSize, %rsp # local vars.
29
30 # Tell user what to do.
31     movl $prompt, %edi    # prompt user
32     call writeStr
33
34 # Get decimal number
35     movl $12, %esi        # allow up to 11 chars
36     leaq buffer(%rbp), %rdi # place for user input
37     call readLn           # get user input string

```

```

38
39 # convert the string to int format
40     leaq    theInt(%rbp), %rsi # place to store the int
41     leaq    buffer(%rbp), %rdi # user's string
42     call    dec2uInt
43
44 # add the constant
45     addl    $theConstant, theInt(%rbp)
46
47 # convert the int to string format
48     movl    theInt(%rbp), %esi # the result
49     leaq    buffer(%rbp), %rdi
50     call    uInt2dec          # do conversion
51
52 # now display for user
53     movl    $msg, %edi       # nice message for user
54     call    writeStr
55
56     leaq    buffer(%rbp), %rdi
57     call    writeStr
58
59     movl    $endl, %edi      # some formatting
60     call    writeStr
61
62     movl    $0, %eax         # return 0
63     movq    %rbp, %rsp        # restore stack pointer
64     popq    %rbp             # restore frame pointer
65     ret                  # back to OS

```

---

```

1 # uInt2dec.s
2 # Converts unsigned int to corresponding unsigned decimal string
3 # Bob Plantz - 13 June 2009
4
5 # Calling sequence
6 #           esi <- value of the int
7 #           rdi <- address of string
8 #           call uInt2dec
9 #           returns zero
10
11 # Stack frame
12     .equ    array,-12
13     .equ    localSize,-16
14 # Read only data
15     .section .rodata
16 ten:   .long   10
17 # Code
18     .text
19     .globl  uInt2dec
20     .type   uInt2dec, @function
21 uInt2dec:
22     pushq   %rbp            # save callers frame ptr
23     movq    %rsp, %rbp       # our stack frame

```

```

24      addq    $localSize, %rsp  # local vars.
25
26      leaq    array(%rbp), %rcx # ptr to local array
27      movb    $0, (%rcx)        # null byte
28
29      movl    %esi, %eax       # the number to conv.
30 charLup:
31      movl    $0, %edx          # high-order = 0
32      divl    ten              # divide by ten
33      orb     $0x30, %dl         # convert to ascii
34      incq    %rcx              # next location
35      movb    %dl, (%rcx)        # store the character
36      cmpl    $0, %eax          # anything left?
37      jne     charLup          # yes, do it
38
39 copyLup:
40      cmpb    $0, (%rcx)        # NUL char?
41      je      allDone          # yes, copy it
42      movb    (%rcx), %dl         # get achar
43      movb    %dl, (%rdi)        # store it
44      incq    %rdi              # move pointers
45      decq    %rcx
46      jmp     copyLup          # and check again
47
48 allDone:
49      movb    (%rcx), %dl         # get NUL char
50      movb    %dl, (%rdi)        # and store it
51      movl    $0, %eax          # return count;
52
53      movq    %rbp, %rsp          # delete local vars.
54      popq    %rbp              # restore caller frame ptr
55      ret

```

See Section E.11 for `writeStr` and `readLn`.

## 12 -8

```

1 # addConstant2.s
2 # Prompts the user to enter an integer in decimal, converts
3 # it to int format, adds a constant, then displays result.
4 # Bob Plantz - 28 June 2009
5
6 # useful constant
7 theConstant = -12345
8
9 # Stack frame
10     .equ   theInt,-16
11     .equ   buffer,-12
12     .equ   localSize,-16
13 # Read only data
14     .section .rodata
15 prompt:
16     .asciz "Please enter an integer in decimal: "
17 msg:

```

```
18     .asciz "The result is: "
19 endl:    .asciz "\n"
20
21 # Code
22     .text
23     .globl main
24     .type main, @function
25 main:
26     pushq %rbp          # save frame pointer
27     movq %rsp, %rbp      # new frame pointer
28     addq $localSize, %rsp # local vars.
29
30 # Tell user what to do.
31     movl $prompt, %edi    # prompt user
32     call writeStr
33
34 # Get decimal number
35     movl $12, %esi        # allow up to 11 chars
36     leaq buffer(%rbp), %rdi # place for user input
37     call readLn           # get user input string
38
39 # convert the string to int format
40     leaq theInt(%rbp), %rsi # place to store the int
41     leaq buffer(%rbp), %rdi # user's string
42     call dec2sInt
43
44 # add the constant
45     addl $theConstant, theInt(%rbp)
46
47 # convert the int to string format
48     movl theInt(%rbp), %esi # the result
49     leaq buffer(%rbp), %rdi
50     call sInt2dec          # do conversion
51
52 # now display for user
53     movl $msg, %edi        # nice message for user
54     call writeStr
55
56     leaq buffer(%rbp), %rdi
57     call writeStr
58
59     movl $endl, %edi       # some formatting
60     call writeStr
61
62     movl $0, %eax          # return 0
63     movq %rbp, %rsp         # restore stack pointer
64     popq %rbp              # restore frame pointer
65     ret                     # back to OS
```

---

1 # dec2sInt.s

2 # Converts string of numerals to decimal int, signed version

3 # Bob Plantz - 13 June 2009

```

4
5 # Calling sequence
6 #         rsi <- address of place to store the int
7 #         rdi <- address of string
8 #         call dec2sInt
9 #         returns 0

10
11 # Stack frame
12     .equ    negFlag,-4
13     .equ    localSize,-16
14 # Code
15     .text
16     .globl  dec2sInt
17     .type   dec2sInt, @function

18 dec2sInt:
19     pushq  %rbp          # save caller frame ptr
20     movq   %rsp, %rbp    # our stack frame
21     addq   $localSize, %rsp # space for local var

22
23     movl   $0, negFlag(%rbp) # assume false

24
25     cmpb   $'-' , (%rdi)   # minus sign?
26     jne    checkPlus      # no, check for plus sign
27     movl   $1, negFlag(%rbp) # negFlag = true;
28     incq   %rdi           # skip minus sign
29     jmp    doIt           # and do the conversion

30 checkPlus:
31     cmpb   $'+', (%rdi)   # plus sign?
32     jne    doIt           # no, ready for conversion
33     incq   %rdi           # skip plus sign

34 doIt:
35     call   dec2uInt       # arguments are correct
36                     # absolute value now stored
37     cmpl   $0, negFlag(%rbp) # negative?
38     je     done            # no, all done
39     negl   (%rsi)          # change sign

40 done:
41     movl   $0, %eax        # return 0
42     movq   %rbp, %rsp      # delete local vars
43     popq   %rbp             # restore caller frame ptr
44

```

---

```

1 # sInt2dec.s
2 # Converts signed int to corresponding signed decimal string
3 # Bob Plantz - 13 June 2009
4
5 # Calling sequence
6 #         esi <- value of the int
7 #         rdi <- address of string
8 #         call sInt2dec
9 #         returns zero
10 # Code

```

```

11      .section  .rodata
12 ten:    .long    10
13
14      .text
15      .globl  sInt2dec
16      .type   sInt2dec, @function
17 sInt2dec:
18      pushq   %rbp          # save callers frame ptr
19      movq    %rsp, %rbp    # our stack frame
20
21      cmpl    $0, %esi      # >= 0?
22      jge     positive    # yes, just convert it
23      movb    $'-', (%rdi)  # store minus sign
24      incq    %rdi          # and move the pointer
25      negl    %esi          # negate the number
26 positive:
27      call    uInt2dec    # arguments are correct
28
29      movl    $0, %eax      # return 0
30      movq    %rbp, %rsp    # delete local vars.
31      popq    %rbp          # restore caller frame ptr
32      ret

```

See above for uInt2dec and dec2uInt. See Section E.11 for writeStr and readLn.

## E.13 Data Structures

**13 -1**

```

1 # arrayIndex.s
2 # Allocates an int array and stores element number in each element.
3 # Bob Plantz - 29 June 2009
4
5 # number of elements in the array
6 nInts = 25
7 # Stack frame
8      .equ    rbxSave,intArray-8
9      .equ    intArray,-4*nInts
10     .equ    index,intArray-8 # 8 bytes to be consistent
11                      # with indexed addressing
12     .equ    localSize,index
13 # Read only data
14     .section .rodata
15 msg:
16     .string "The array contains:\n"
17 endl:
18     .string "\n"
19 # Code
20     .text
21     .globl  main
22     .type   main, @function
23 main:

```

```

24      pushq %rbp          # save caller frame pointer
25      movq %rsp, %rbp     # set our frame pointer
26      addq $localSize, %rsp # local variables
27      andq $-16, %rsp     # 16-byte alignment
28      movq %rbx, rbxSave(%rbp) # save reg for OS
29
30      movq $0, index(%rbp) # index = 0
31      leaq intArray(%rbp), %rbx # the array
32
33 # store values in the array
34 storeUp:
35      movq index(%rbp), %rax # get index value
36      cmpq $nInts, %rax    # all filled?
37      jae display         # yes, display it
38
39      movl %eax, (%rbx,%rax,4) # no, store index
40
41      incq index(%rbp)     # index++
42      jmp storeUp         # do rest of elements
43
44 display:
45      movq $0, index(%rbp) # restart at beginning
46 displayUp:
47      movq index(%rbp), %rax # get index value
48      cmpq $nInts, %rax    # any more?
49      jae done             # no, all done
50
51      movl (%rbx,%rax,4), %edi # yes, display it
52      call putInt
53
54      movl $endl, %edi       # do it in a column
55      call writeStr
56
57      incq index(%rbp)     # index++
58      jmp displayUp        # do rest of elements
59
60 done:
61      movl $0, %eax         # return 0;
62      movq rbxSave(%rbp), %rbx # restore reg
63      movq %rbp, %rsp        # remove local vars
64      popq %rbp             # restore caller frame ptr
65      ret                  # back to OS

```

---

```

1 #.putInt.s
2 # writes a signed int to standard out
3 # Bob Plantz - 28 June 2009
4
5 # Calling sequence
6 #      edi <- value of the int
7 #      call putInt
8
9 # Stack frame

```

```

10      .equ    buffer,-12
11      .equ    localSize,-16
12 # Code
13      .text
14      .globl .putInt
15      .type  .putInt, @function
16 putInt:
17      pushq   %rbp          # save callers frame ptr
18      movq   %rsp, %rbp      # our stack frame
19      addq   $localSize, %rsp # local vars.
20
21      movl   %edi, %esi      # number to convert
22      leaq   buffer(%rbp), %rdi # place to store string
23      call   sInt2dec        # do the conversion to string
24
25      leaq   buffer(%rbp), %rdi # place where string stored
26      call   writeStr         # write it
27
28      movl   $0, %eax        # return 0
29      movq   %rbp, %rsp      # delete local vars.
30      popq   %rbp            # restore caller frame ptr
31      ret

```

See Section E.12 for `sInt2dec`. See Section E.11 for `writeStr`.

### 13 -2

```

1 # sumInts.s
2 # Prompts user for 10 integers, stores them in an array, then
3 # displays their sum.
4 # Bob Plantz - 28 June 2009
5
6 # number of elements in the array
7      .equ    nInts,10
8 # Stack frame
9      .equ    rbxSave,total-8
10     .equ    total,index-4
11     .equ    index,intArray-8
12     .equ    intArray,-4*nInts
13     .equ    localSize,rbxSave
14 # Read only data
15     .section .rodata
16 prompt:
17     .string "Enter an integer: "
18 msg:
19     .string "The sum is: "
20 endl:
21     .string "\n"
22 # Code
23     .text
24     .globl  main
25     .type   main, @function
26 main:
27     pushq   %rbp          # save caller frame pointer

```

```
28      movq    %rsp, %rbp      # set our frame pointer
29      addq    $localSize, %rsp # local variables
30      andq    $-16, %rsp     # 16-byte boundary
31      movq    %rbx, rbxSave(%rbp) # save reg for OS
32
33      movq    $0, index(%rbp)  # index = 0
34      leaq    intArray(%rbp), %rbx # the array
35
36 # store user values in the array
37 storeUp:
38      cmpq    $nInts, index(%rbp)    # all filled?
39      jae     sum                  # yes, sum them
40
41      movl    $prompt, %edi       # no, prompt user
42      call    writeStr
43
44      movq    index(%rbp), %rax   # get index value
45      leaq    (%rbx,%rax,4), %rdi # place to store int
46      call    getInt
47
48      incq    index(%rbp)       # index++
49      jmp    storeUp          # do rest of elements
50
51 sum:
52      movq    $0, index(%rbp)  # restart at beginning
53      movl    $0, total(%rbp) # init total
54 sumUp:
55      cmpl    $nInts, index(%rbp) # all summed?
56      jae     display           # yes, display total
57
58      movq    index(%rbp), %rax   # get index value
59      movl    (%rbx,%rax,4), %eax # no, add current
60      addl    %eax, total(%rbp)
61
62      incq    index(%rbp)       # index++
63      jmp    sumUp            # do rest of elements
64
65 display:
66      movl    $msg, %edi        # tell user about it
67      call    writeStr
68
69      movl    total(%rbp), %edi # and show the sum
70      call    putInt
71
72      movl    $endl, %edi
73      call    writeStr
74
75      movq    rbxSave(%rbp), %rbx # restore reg
76      movl    $0, %eax           # return 0;
77      movq    %rbp, %rsp         # remove local vars
78      popq    %rbp              # restore caller frame ptr
79      ret                 # back to OS
```

---

```

1 # getInt.s
2 # reads an int from standard in
3 # Bob Plantz - 28 June 2009
4
5 # Calling sequence
6 #         rdi <- pointer where to store the int
7 #         call getInt
8 #         returns 0
9
10 # Stack frame
11     .equ    outPtr,-24
12     .equ    buffer,-12
13     .equ    localSize,-32
14 # Code
15     .text
16     .globl  getInt
17     .type   getInt, @function
18 getInt:
19     pushq  %rbp          # save callers frame ptr
20     movq   %rsp, %rbp    # our stack frame
21     addq   $localSize, %rsp # local vars.
22
23     movq   %rdi, outPtr(%rbp) # save argument
24
25     movl   $12, %esi       # max number of chars
26     leaq   buffer(%rbp), %rdi # place where string stored
27     call   readLn         # read it
28
29     movq   outPtr(%rbp), %rsi # place to store number
30     leaq   buffer(%rbp), %rdi # address of string
31     call   dec2sInt        # convert string to int
32
33     movl   $0, %eax        # return 0
34     movq   %rbp, %rsp      # delete local vars.
35     popq   %rbp            # restore caller frame ptr
36     ret

```

---

See above for putInt. See Section E.12 for dec2sInt. See Section E.11 for writeStr and readLn.

**13 -3**


---

```

1 # averageInts
2 # Prompts user for 10 integers, stores them in an array, then
3 # displays their average.
4 # Bob Plantz - 29 June 2009
5
6 # number of elements in the array
7     .equ    nInts,10
8 # Stack frame
9     .equ    rbxSave,total-8
10    .equ    total,index-4
11    .equ    index,intArray-8

```

---

```
12         .equ    intArray,-4*nInts
13         .equ    localSize,rbxSave
14 # Read only data
15         .section .rodata
16 prompt:   .string "Enter an integer: "
17 msg:       .string "The sum is: "
18 endl:     .string "\n"
19
20 endl:     .string "\n"
21
22 # Code
23         .text
24         .globl main
25         .type main, @function
26 main:
27         pushq %rbp          # save caller frame pointer
28         movq %rsp, %rbp      # set our frame pointer
29         addq $localSize, %rsp # local variables
30         andq $-16, %rsp      # 16-byte boundary
31         movq %rbx, rbxSave(%rbp) # save reg for OS
32
33         movq $0, index(%rbp) # index = 0
34         leaq intArray(%rbp), %rbx # the array
35
36 # store user values in the array
37 storeLup:
38         cmpq $nInts, index(%rbp) # all filled?
39         jae sum             # yes, sum them
40
41         movl $prompt, %edi    # no, prompt user
42         call writeStr
43
44         movq index(%rbp), %rax # get index value
45         leaq (%rbx,%rax,4), %rdi # place to store int
46         call getInt
47
48         incq index(%rbp)     # index++
49         jmp storeLup        # do rest of elements
50
51 sum:
52         movq $0, index(%rbp) # restart at beginning
53         movl $0, total(%rbp) # init total
54 sumLup:
55         cmpl $nInts, index(%rbp) # all summed?
56         jae display          # yes, display total
57
58         movq index(%rbp), %rax # get index value
59         movl (%rbx,%rax,4), %eax # no, add current
60         addl %eax, total(%rbp)
61
62         incq index(%rbp)     # index++
63         jmp sumLup          # do rest of elements
```

```

64
65 display:
66     movl    $msg, %edi      # tell user about it
67     call    writeStr
68
69
70     movl    total(%rbp), %eax  # compute the average
71     movl    $0, %edx          # create 64-bit dividend
72     cmpl    $0, %eax          # is it negative?
73     jge     pos              # no
74     movl    $-1, %edx         # sign extend dividend
75 pos:
76     movl    $nInts, %ebx      # get divisor
77     idivl   %ebx             # signed division
78
79     movl    %eax, %edi        # and show the average
80     call   .putInt
81
82     movl    $endl, %edi
83     call    writeStr
84
85     movq    rbxSave(%rbp), %rbx # restore reg
86     movl    $0, %eax          # return 0;
87     movq    %rbp, %rsp          # remove local vars
88     popq    %rbp              # restore caller frame ptr
89     ret                  # back to OS

```

See above for `putInt` and `getInt`. See Section E.11 for `writeStr` and `readLn`.

### 13 -8

```

1 # structFields.s
2 # Stores user input values in three structs and echoes them
3 # Bob Plantz - 28 June 2009
4
5     .include "structDef.h"
6 # Stack frame
7     .equ    buffer,z-12
8     .equ    z,y-structSize
9     .equ    y,x-structSize
10    .equ   x,-structSize
11    .equ    localSize,buffer
12 # Read only data
13    .section .rodata
14 userPrompt:
15    .string "Enter data for the three structs.\n"
16 echoMsg:
17    .string "You entered:\n"
18 endl:
19    .string "\n"
20 # Code
21    .text
22    .globl  main
23    .type   main, @function

```

```

24 main:
25     pushq %rbp          # save frame pointer
26     movq %rsp, %rbp      # our frame pointer
27     addq $localSize, %rsp # local variables
28     andq $-16, %rsp      # stack alignment
29
30     movl $userPrompt, %edi # tell user what to do
31     call writeStr
32
33     leaq x(%rbp), %rdi   # the x struct
34     call getData          # get values from user
35
36     leaq y(%rbp), %rdi   # the y struct
37     call getData          # get values from user
38
39     leaq z(%rbp), %rdi   # the z struct
40     call getData          # get values from user
41
42 # give the user a message
43     movl $echoMsg, %edi   # start display
44     call writeStr
45
46     leaq x(%rbp), %rdi   # the x struct
47     call putData          # show values to user
48
49     leaq y(%rbp), %rdi   # the y struct
50     call putData          # show values to user
51
52     leaq z(%rbp), %rdi   # the z struct
53     call putData          # show values to user
54
55     movl $endl, %edi      # do a newline
56     call writeStr
57
58     movl $0, %eax          # return 0;
59     movq %rbp, %rsp        # remove local vars
60     popq %rbp              # restore caller's frame ptr
61     ret                   # back to OS

```

---

```

1 # structDef.h
2 # Defines the struct field offsets.
3 # Bob Plantz - 28 June 2009
4
5 # struct definition
6     .equ aChar,0
7     .equ anInt,4
8     .equ structSize,8

```

---

```

1 # getData.s
2 # Gets user input values and stores them in a struct.
3 # Bob Plantz - 28 June 2009
4 # Calling sequence:

```

```

5 #           rdi <- address of struct
6 #           call putData
7
8 .include "structDef.h"
9 # Useful constant
10 .equ    STDOUT,1
11 # Stack frame
12 .equ    structPtr,-32
13 .equ    buffer,-2
14 .equ    localSize,-32
15 # Read only data
16 .section .rodata
17 charPrompt:
18     .string "Enter a single character: "
19 intPrompt:
20     .string "Enter an integer: "
21 # Code
22 .text
23 .globl  getData
24 .type   getData, @function
25 getData:
26     pushq %rbp          # save frame pointer
27     movq  %rsp, %rbp    # our frame pointer
28     addq  $localSize, %rsp # local var. and arg.
29     movq  %rdi, structPtr(%rbp) # save argument
30
31     movl  $charPrompt, %edi # prompt user for character
32     writeStr
33
34     movl  $2, %esi        # local buffer size
35     leaq   buffer(%rbp), %rdi # place to store input
36     call   readLn         # get user response
37
38     movb  buffer(%rbp), %al # first char entered
39     movq  structPtr(%rbp), %rdi
40     movb  %al, aChar(%rdi) # x.aChar = buffer[0]
41
42     movl  $intPrompt, %edi # prompt user for integer
43     writeStr
44     movq  structPtr(%rbp), %rdi
45     leaq   anInt(%rdi), %rdi # place for the int
46     call   getInt          # get user response
47
48     movl  $0, %eax        # return 0;
49     movq  %rbp, %rsp      # remove local vars
50     popq  %rbp            # restore caller's frame ptr
51     ret                  # back to caller

# putData.s
# Displays values stored in a struct.
# Bob Plantz - 28 June 2009
# Calling sequence:
```

```

5 #           rdi <- address of struct
6 #           call putData
7
8 .include "structDef.h"
9 # Useful constant
10 .equ    STDOUT,1
11 # Stack frame
12 .equ    structPtr,-16
13 .equ    localSize,-16
14 # Read only data
15 .section .rodata
16 charMsg: .string "The char is: "
17 intMsg:  .string "The int is: "
18 endl:    .string "\n"
19 # Code
20 .text
21 .globl  putData
22 .type   putData, @function
23
24 putData:
25     pushq %rbp          # save frame pointer
26     movq  %rsp, %rbp    # our frame pointer
27     addq  $localSize, %rsp # argument save area
28     movq  %rdi, structPtr(%rbp) # save struct addr.
29
30
31
32     movq  $charMsg, %rdi  # tell user about character
33     call  writeStr
34     movq  structPtr(%rbp), %rsi # the struct
35     movl  $1, %edx        # one byte
36     leaq  aChar(%rsi), %rsi  # address of the char
37     movl  $STDOUT, %edi
38     call  write
39     movl  $endl, %edi    # some nice formatting
40     call  writeStr
41
42
43     movl  $intMsg, %edi  # tell user about integer
44     call  writeStr
45     movq  structPtr(%rbp), %rdi # the struct
46     movl  anInt(%rdi), %edi
47     call  putInt        # display the integer
48     movl  $endl, %edi    # some nice formatting
49     call  writeStr
50
51     movl  $0, %eax       # return 0;
52     movq  %rbp, %rsp
53     popq  %rbp
54     ret

```

See above for `putInt` and `getInt`. See Section E.11 for `writeStr` and `readLn`.

**13 -9**

```
1 # totalCost.s
2 # Gets names and prices for three items and shows total cost
3 # Bob Plantz - 29 June 2009
4
5         .include "item.h"
6 # Stack frame
7         .equ    third,second-itemSize
8         .equ    second,first-itemSize
9         .equ    first,-itemSize
10        .equ   localSize,third
11 # Read only data
12         .section .rodata
13 endl:  .string "\n"
14 totalMsg:
15         .string "Their total cost is $"
16 # Code
17         .text
18         .globl main
19         .type  main, @function
20 main:
21         pushq %rbp          # save frame pointer
22         movq  %rsp, %rbp      # our frame pointer
23         addq  $localSize, %rsp # local variables
24         andq  $-16, %rsp      # 16-byte boundary
25
26 # get values into each of the struct variables
27         leaq   first(%rbp), %rdi # address of first struct
28         call   getItem          # gets the values
29         leaq   second(%rbp), %rdi # address of second struct
30         call   getItem
31         leaq   third(%rbp), %rdi # address of third struct
32         call   getItem
33
34 # display them
35         leaq   first(%rbp), %rdi # address of first struct
36         call   displayItem      # displays the values
37         leaq   second(%rbp), %rdi # address of second struct
38         call   displayItem
39         leaq   third(%rbp), %rdi # address of third struct
40         call   displayItem
41
42 # Now show their total cost
43         movl   $totalMsg, %edi  # message for user
44         call   writeStr
45
46         leaq   first(%rbp), %rsi # first item
47         movl   cost(%rsi), %edi # accumulate sum in eax
48         leaq   second(%rbp), %rsi # second item
49         addl   cost(%rsi), %edi # add to sum
50         leaq   third(%rbp), %rsi # third item
51         addl   cost(%rsi), %edi # add to sum
```

```
52      call    putInt          # argument in correct reg.
53
54      movl    $endl, %edi      # do a newline for user
55      call    writeStr
56
57      movl    $0, %eax         # return 0;
58      movq    %rbp, %rsp        # remove local vars
59      popq    %rbp             # restore caller's frame ptr
60      ret                 # back to OS
```

---

```
1 # item.h
2 # Fields and size of an item struct
3 # Bob Plantz - 29 June 2009
4
5     .equ    name,0
6     .equ    cost,52
7     .equ    itemSize,56
```

---

```
1 # displayItem.s
2 # displays an item
3 # Bob Plantz - 29 June 2009
4
5 # Calling sequence
6 #       rdi <- address of item struct
7 #       call getItem
8 #       returns void
9
10    .include "item.h"
11 # Stack frame
12    .equ    structPtr,-16
13    .equ    localSize,-16
14 # Read only data
15    .section .rodata
16 costMsg:
17    .string "Cost: $"
18 nameMsg:
19    .string "Name: "
20 endl:
21    .string "\n"
22 spacing:
23    .string "      "
24 # Code
25    .text
26    .globl  displayItem
27    .type   displayItem, @function
28 displayItem:
29    pushq   %rbp             # save caller's frame ptr
30    movq    %rsp, %rbp        # our stack frame
31    addq    $localSize, %rsp  # local vars
32    movq    %rdi, structPtr(%rbp) # save arg.
33
34    movl    $nameMsg, %edi    # name message
```

```

35      call    writeStr
36      movq    structPtr(%rbp), %rdi # struct address
37      leaq    name(%rdi), %rdi # get pointer to name
38      call    writeStr          # show name
39
40      movl    $spacing, %edi   # do some formatting
41      call    writeStr
42
43      movl    $costMsg, %edi   # cost message
44      call    writeStr
45      movq    structPtr(%rbp), %rdi # struct address
46      movl    cost(%rdi), %edi  # the integer
47      call    putInt          # write it
48
49      movl    $endl, %edi     # newline
50      call    writeStr
51
52      movq    %rbp, %rsp      # delete local vars.
53      popq    %rbp           # restore caller's frame ptr
54      ret

```

---

```

1 # getItem.s
2 # prompts user to enter an item name and it's cost
3 # Bob Plantz - 29 June 2009
4
5 # Calling sequence
6 #         rdi <- address of item struct
7 #         call getItem
8 #         returns void
9
10        .include "item.h"
11 # Stack frame
12        .equ    structPtr,-16
13        .equ    localSize,-16
14 # Read only data
15        .section .rodata
16 costMsg:
17        .string "Enter cost: $"
18 nameMsg:
19        .string "Name: "
20 # Code
21        .text
22        .globl  getItem
23        .type   getItem, @function
24 getItem:
25        pushq   %rbp          # save caller's frame ptr
26        movq    %rsp, %rbp     # our stack frame
27        addq    $localSize, %rsp # local vars.
28        movq    %rdi, structPtr(%rbp) # save arg.
29
30        movl    $nameMsg, %edi  # prompt for name
31        call    writeStr

```

---

```

32      movq   structPtr(%rbp), %rdi # struct address
33      leaq   name(%rdi), %rdi # pointer to name field
34      movl   $50, %esi        # max name length
35      call   readLn         # get name
36
37      movl   $costMsg, %edi  # prompt for cost
38      call   writeStr
39      movq   structPtr(%rbp), %rdi # struct address
40      leaq   cost(%rdi), %rdi # pointer to cost field
41      call   getInt          # get user input
42
43      movq   %rbp, %rsp      # delete local vars.
44      popq   %rbp            # restore caller's frame ptr
45      ret

```

---

See above for putInt and getInt. See Section E.11 for writeStr and readLn.

### 13 -10

```

1 # addInt2Frac.s
2 # creates a fraction and gets user values, then gets an
3 # integer from user and adds it to the fraction.
4 # Bob Plantz - 29 June 2009
5
6     .include "fraction.h"
7
8 # Stack frame
9     .equ    anInt,x-4
10    .equ    x,-fracSize
11    .equ    localSize,anInt
12
13 # Read only data
14     .section .rodata
15 prompt:
16     .string "Enter an integer: "
17 endl:
18     .string "\n"
19 # Code
20     .text
21     .globl main
22     .type main, @function
23 main:
24     pushq %rbp           # save frame pointer
25     movq %rsp, %rbp       # our frame pointer
26     addq $localSize, %rsp # local variables
27     andq $-16, %rsp       # align stack pointer
28
29     leaq x(%rbp), %rdi   # load address of object
30     call fraction         # call "constructor"
31
32     leaq x(%rbp), %rdi   # load address of object
33     call fractionGet      # get "member function"
34
35     movl $prompt, %edi    # ask user for a number

```

```

36      call    writeStr
37      leaq    anInt(%rbp), %rdi # and get it
38      call    getInt
39
40      movl    anInt(%rbp), %esi # value to add to fraction
41      leaq    x(%rbp), %rdi # load address of object
42      call    fractionAdd # add "member function"
43
44      leaq    x(%rbp), %rdi # load address of object
45      call    fractionDisplay # display "member function"
46
47      movl    $endl, %edi # formatting
48      call    writeStr
49
50      movl    $0, %eax # return 0;
51      movq    %rbp, %rsp # delete local vars.
52      popq    %rbp # restore base pointer for OS
53      ret     # back to caller (OS)

```

---

See above for getInt. See Section E.11 for writeStr.

### 13 -11

```

1 # addFrac2Frac.s
2 # creates two fractions and gets user values, then adds
3 # one to the other and displays the sum.
4 # Bob Plantz - 30 June 2009
5
6         .include "fraction.h"
7
8 # Stack frame
9         .equ    y,x-fracSize
10        .equ    x,-fracSize
11        .equ    localSize,y
12 # Read only data
13         .section .rodata
14 prompt:   .string "Enter two fractions:\n"
15 msg:       .string "Their sum is:\n"
16 endl:     .string "\n"
17
18 # Code
19         .text
20         .globl  main
21         .type   main, @function
22
23 main:
24         pushq   %rbp # save frame pointer
25         movq   %rsp, %rbp # our frame pointer
26         addq   $localSize, %rsp # local vars.
27         andq   $-16, %rsp # align stack pointer
28
29         leaq    x(%rbp), %rdi # pass address of object
30         call    fraction # to "constructor"

```

```

32      leaq    y(%rbp), %rdi      # pass address of object
33      call    fraction          # to "constructor"
34
35      movl    $prompt, %edi     # tell user what to do
36      call    writeStr
37
38      leaq    x(%rbp), %rdi      # get address of object
39      call    fractionGet       # get "member function"
40      leaq    y(%rbp), %rdi      # get address of object
41      call    fractionGet       # get "member function"
42
43      leaq    y(%rbp), %rsi      # address of argument
44      leaq    x(%rbp), %rdi      # address of object
45      call    fractionsAdd      # add "member function"
46
47      movl    $msg, %edi        # tell user what happened
48      call    writeStr
49      leaq    x(%rbp), %rdi      # get address of object
50      call    fractionDisplay   # display "member function"
51      movl    $endl, %edi        # formatting
52      call    writeStr
53
54      movq    %rbp, %rsp        # delete local vars.
55      popq    %rbp             # restore frame pointer
56      ret                  # back to caller

```

---

```

1 # fractionsAdd.s
2 # adds a fraction to this fraction
3 # Bob Plantz - 30 June 2009
4
5 # Calling sequence
6 #           rsi <- address of object to add
7 #           rdi <- address of this object
8 #           call   fractionsAdd
9 #           returns void
10
11      .include "fraction.h"
12 # Stack frame
13      .equ   localFraction,-fracSize
14      .equ   localSize,localFraction
15 # Code
16      .text
17      .globl fractionsAdd
18      .type  fractionAdd, @function
19 fractionsAdd:
20      pushq   %rbp            # save frame pointer
21      movq   %rsp, %rbp        # our frame pointer
22      addq   $localSize, %rsp  # for object address
23      andq   $-16, %rsp        # align stack pointer
24      leaq   localFraction(%rbp), %rcx  # pointer to local fraction
25
26      movl   den(%rsi), %eax  # multiply the denominators

```

```

27      mull    den(%rdi)
28      movl    %eax, den(%rcx) # new denominator
29
30      movl    num(%rdi), %eax # get this numerator
31      mull    den(%rsi)      # multiply by arg denominator
32      movl    %eax, num(%rcx) # and store in local num
33
34      movl    num(%rsi), %eax # get arg num
35      mull    den(%rdi)      # multiply by this den
36      addl    %eax, num(%rcx) # and add to local num
37
38      movl    num(%rcx), %eax # copy back to this object
39      movl    %eax, num(%rdi)
40      movl    den(%rcx), %eax
41      movl    %eax, den(%rdi)
42
43      movq    %rbp, %rsp      # delete local vars.
44      popq    %rbp          # restore frame pointer
45      ret                 # back to caller

```

See Section E.11 for `writeStr`.

### 13 -12

```

1 # addressBook.s
2 # Allows up to MAX address cards to be stored.
3 # Bob Plantz - 30 June 2009
4
5     .include "cardDef.h"
6 # Set MAX for the maximum number of cards
7     .equ    MAX,3
8 # Stack frame
9     .equ    count,index-4
10    .equ    index,cards-4
11    .equ    cards,buffer-(MAX*cardSize)
12    .equ    buffer,-32
13    .equ    localSize,count
14 # Read only data
15    .section .rodata
16 prompt:
17    .string "Command (Add, Delete, Show, List, Quit): "
18 addMsg:
19    .string "Add new person.\n"
20 delMsg:
21    .string "Delete last person.\n"
22 showMsg:
23    .string "Your addresses:\n"
24 listMsg:
25    .string "Here is the entire array:\n"
26 fullMsg:
27    .string "Address book is full.\n"
28 emptyMsg:
29    .string "Address book is empty.\n"
30 endl:

```

```

31          .string "\n"
32 # Code
33          .text
34          .globl main
35          .type main, @function
36 main:
37          pushq %rbp          # save frame pointer
38          movq %rsp, %rbp    # our frame pointer
39          addq $localSize, %rsp # local variables
40          andq $-16, %rsp     # align stack pointer
41
42 # construct the array objects
43          movl $0, index(%rbp) # start at beginning
44 constLup:
45          cmpl $MAX, index(%rbp) # end of list?
46          jae doProg          # yes, run the program
47          leaq cards(%rbp), %rdi # no, beginning of list
48          movl $cardSize, %eax   # length of each record
49          mull index(%rbp)      # times current location
50          addq %rax, %rdi       # points to current record
51          call card            # call constructor
52          incl index(%rbp)
53          jmp constLup
54
55 doProg:
56          movl $0, count(%rbp) # no people yet
57 runLoop:
58          movl $prompt, %edi    # tell user what to do
59          call writeStr
60
61          movl $32, %esi         # max number of chars
62          leaq buffer(%rbp), %rdi # place for user input
63          call readLn          # get user command
64          orb $0x20, buffer(%rbp) # make first char lower case
65
66          cmpb $'q', buffer(%rbp) # quit?
67          je allDone
68
69          cmpb $'a', buffer(%rbp) # check for add command
70          jne chkDel
71          cmpl $MAX, count(%rbp) # check for full list
72          jb doAdd             # there's space
73          movl $fullMsg, %edi   # array is full
74          call writeStr
75          jmp cont
76 doAdd:
77          movl $addMsg, %edi     # feedback for user
78          call writeStr
79          leaq cards(%rbp), %rdi # the array
80          movl $cardSize, %eax   # length of each record
81          mull count(%rbp)      # times number of records in array
82          addq %rax, %rdi       # points to next open space

```

```

83      call    cardGet          # fill it in
84      incl    count(%rbp)
85      jmp     cont
86
87 chkDel:
88      cmpb    '$d', buffer(%rbp) # check for delete command
89      jne     chkShow
90      cmpl    $0, count(%rbp)   # anybody on the list?
91      ja      doDel           # yes, delete last one
92      movl    $emptyMsg, %edi   # no, tell user
93      call    writeStr
94      jmp     cont
95 doDel:
96      movl    $delMsg, %edi    # feedback for user
97      call    writeStr
98      decl    count(%rbp)
99      jmp     cont
100
101 chkShow:
102     cmpb    '$s', buffer(%rbp) # check for show command
103     jne     chkList
104     cmpl    $0, count(%rbp)   # anybody on the list?
105     ja      doShow          # yes, show them
106     movl    $emptyMsg, %edi   # no, tell user
107     call    writeStr
108     jmp     cont
109 doShow:
110     movl    $showMsg, %edi    # feedback for user
111     call    writeStr
112
113     movl    $0, index(%rbp)   # start at beginning
114 showUp:
115     movl    count(%rbp), %eax
116     cmpl    %eax, index(%rbp)
117     jae     cont
118     leaq    cards(%rbp), %rdi
119     movl    $cardSize, %eax
120     mull    index(%rbp)
121     addq    %rax, %rdi
122     call    cardPut
123     incl    index(%rbp)
124     jmp     showUp
125
126 chkList:
127     cmpb    '$l', buffer(%rbp) # check for list command
128     jne     cont
129     movl    $listMsg, %edi    # feedback for user
130     call    writeStr
131
132     movl    $0, index(%rbp)   # start at beginning
133 listUp:
134     cmpl    $MAX, index(%rbp) # end of list?

```

```

135      jae    cont          # yes, next thing
136      leaq   cards(%rbp), %rdi  # the array
137      movl   $cardSize, %eax   # length of each record
138      mull   index(%rbp)     # times number of records in array
139      addq   %rax, %rdi      # points to current card
140      call   cardPut
141      incl   index(%rbp)
142      jmp    listUp
143 cont:
144      jmp    runLoop
145 allDone:
146      movl   $0, %eax        # return 0;
147      movq   %rbp, %rsp      # remove local vars
148      popq   %rbp           # restore caller's frame ptr
149      ret

```

---

```

1 # cardDef.h
2 # Defines the address card field offsets.
3 # Bob Plantz - 30 June 2009
4
5 # card definition
6     .equ   name,0
7     .equ   address,name+48
8     .equ   city,address+80
9     .equ   state,city+24
10    .equ   zip,state+20
11    .equ   cardSize,zip+6

```

---

```

1 # card.s
2 # card object default constructor.
3 # Bob Plantz - 30 June 2009
4 # Calling sequence:
5 #         rdi <- address of object
6 #         call card
7 #         returns void
8
9         .include "cardDef.h"
10 # Stack frame
11     .equ   thisPtr,-16
12     .equ   localSize,-16
13 # Read only data
14     .section .rodata
15 nameDefault:
16     .string "J. Doe"
17 addressDefault:
18     .string "123 Main St."
19 cityDefault:
20     .string "Middle Town"
21 stateDefault:
22     .string "Kansas"
23 zipDefault:
24     .string "12345"

```

```

25 # Code
26     .text
27     .globl card
28     .type card, @function
29 card:
30     pushq %rbp          # save frame pointer
31     movq %rsp, %rbp      # our frame pointer
32     addq $localSize, %rsp # for saving argument
33     movq %rdi, thisPtr(%rbp) # save it
34
35 # Copy default data into the fields
36     movl $nameDefault, %edx # name
37     movq thisPtr(%rbp), %rsi
38     leaq name(%rsi), %rsi # place to store string
39     movl $(address-name), %edi # max number of chars
40     call copyStr
41
42     movl $addressDefault, %edx # address
43     movq thisPtr(%rbp), %rsi
44     leaq address(%rsi), %rsi # place to store string
45     movl $(city-address), %edi # max number of chars
46     call copyStr
47
48     movl $cityDefault, %edx # city
49     movq thisPtr(%rbp), %rsi
50     leaq city(%rsi), %rsi # place to store string
51     movl $(state-city), %edi # max number of chars
52     call copyStr
53
54     movl $stateDefault, %edx # state
55     movq thisPtr(%rbp), %rsi
56     leaq state(%rsi), %rsi # place to store string
57     movl $(zip-state), %edi # max number of chars
58     call copyStr
59
60     movl $zipDefault, %edx # state
61     movq thisPtr(%rbp), %rsi
62     leaq zip(%rsi), %rsi # place to store string
63     movl $(cardSize-zip), %edi # max number of chars
64     call copyStr
65
66     movq %rbp, %rsp      # remove local vars
67     popq %rbp            # restore caller's frame ptr
68     ret                  # back to caller

```

---

```

1 # copyStr.s
2 # Copies a C-style text string.
3 #
4 # Calling sequence:
5 #     rdx <- address of source
6 #     rsi <- address of destination
7 #     edi <- maximum length to copy (including NULL)

```

```

8 #           call copyStr
9 #   returns number of chars copied, not including NULL.
10 # assumes maximum length is at least 1.
11 # Bob Plantz - 30 June 2009
12
13 # Code
14         .text
15         .globl  copyStr
16         .type   copyStr, @function
17 copyStr:
18         pushq   %rbp          # save frame pointer
19         movq    %rsp, %rbp    # our frame pointer
20
21         subl    $1, %edi      # allow room for NULL
22         movl    $0, %eax      # count = 0
23 loop:
24         cmpl    %eax, %edi    # any more space?
25         jle     done          # no, have to quit
26         movb    (%rdx), %cl   # yes, get a char
27         cmpb    $0, %cl       # NULL?
28         je      done          # yes, copy is done
29         movb    %cl, (%rsi)    # no, copy the char
30         incq    %rdx          # increment our pointers
31         incq    %rsi
32         incl    %eax          # and the counter
33         jmp    loop          # and check for more
34
35 done:   movb    $0, (%rsi)    # store NULL char
36
37         movq    %rbp, %rsp    # remove local vars
38         popq    %rbp          # restore caller's frame ptr
39         ret

```

```

1 # cardGet.s
2 # Gets user input values and stores them in a card object.
3 #
4 # Calling sequence:
5 #           rdi <- address of object
6 #           call cardGet
7 # Bob Plantz - 30 June 2009
8
9         .include "cardDef.h"
10 # Stack frame
11         .equ    thisPtr,-16
12         .equ    localSize,-16
13 # Read only data
14         .section .rodata
15 Prompt:   .string "Enter the data\n"
16 namePrompt: .string "           name: "
17 addressPrompt:

```

```
20           .string "      address: "
21 cityPrompt:
22           .string "      city: "
23 statePrompt:
24           .string "      state: "
25 zipPrompt:
26           .string "      zip code: "
27 # Code
28           .text
29           .globl cardGet
30           .type cardGet, @function
31 cardGet:
32     pushq %rbp          # save frame pointer
33     movq %rsp, %rbp      # our frame pointer
34     addq $localSize, %rsp # local vars.
35     movq %rdi, thisPtr(%rbp) # address of object
36
37     movl $Prompt, %edi    # tell user to enter data
38     call writeStr
39 # get user responses
40     movl $namePrompt, %edi # name
41     call writeStr
42     movl $(address-name), %esi # space allocated for name
43     movq thisPtr(%rbp), %rdi # our object
44     leaq name(%rdi), %rdi   # name field
45     call readLn
46
47     movl $addressPrompt, %edi # address
48     call writeStr
49     movl $(city-address), %esi # space allocated for address
50     movq thisPtr(%rbp), %rdi # our object
51     leaq address(%rdi), %rdi # address field
52     call readLn
53
54     movl $cityPrompt, %edi   # city
55     call writeStr
56     movl $(state-city), %esi # space allocated for city
57     movq thisPtr(%rbp), %rdi # our object
58     leaq city(%rdi), %rdi   # city field
59     call readLn
60
61     movl $statePrompt, %edi # state
62     call writeStr
63     movl $(zip-state), %esi # space allocated for state
64     movq thisPtr(%rbp), %rdi # our object
65     leaq state(%rdi), %rdi   # state field
66     call readLn
67
68     movl $zipPrompt, %edi   # zip code
69     call writeStr
70     movl $(cardSize-zip), %esi # space allocated for zip
71     movq thisPtr(%rbp), %rdi # our object
```

```
72     leaq    zip(%rdi), %rdi          # zip field
73     call    readLn
74
75     movl    $0, %eax      # return 0;
76     movq    %rbp, %rsp      # remove local vars
77     popq    %rbp          # restore caller's frame ptr
78     ret
```

---

```
1 # cardPut.s
2 # Displays a card object.
3 #
4 # Calling sequence:
5 #           rdi <- address of object
6 #           call cardPut
7 # Bob Plantz - 30 June 2009
8
9         .include "cardDef.h"
10 # Stack frame
11         .equ    thisPtr,-16
12         .equ    localSize,-16
13 # Read only data
14         .section .rodata
15 Msg:        .string "*** Address Card ***\n"
16 nameMsg:    .string "      name: "
17 addressMsg: .string "    address: "
18 cityMsg:    .string "      city: "
19 stateMsg:   .string "      state: "
20 zipMsg:     .string "  zip code: "
21 endl:       .string "\n"
22 # Code
23         .text
24         .globl  cardPut
25         .type   cardPut, @function
26 cardPut:
27         pushq   %rbp          # save frame pointer
28         movq    %rsp, %rbp      # our frame pointer
29         addq    $localSize, %rsp # local vars.
30         movq    %rdi, thisPtr(%rbp) # address of object
31
32         movl    $Msg, %edi      # tell user about data
33         call    writeStr
34
35         # show each field
36         movl    $nameMsg, %edi    # name
37         call    writeStr
```

```

45      movq    thisPtr(%rbp), %rdi # our object
46      leaq    name(%rdi), %rdi # name field
47      call    writeStr
48      movl    $endl, %edi      # next line (nb: I do them
49      call    writeStr        # individually so it's easier
50
51      movl    $addressMsg, %edi # address
52      call    writeStr
53      movq    thisPtr(%rbp), %rdi # our object
54      leaq    address(%rdi), %rdi # address field
55      call    writeStr
56      movl    $endl, %edi      # next line (nb: I do them
57      call    writeStr        # individually so it's easier
58
59      movl    $cityMsg, %edi   # city
60      call    writeStr
61      movq    thisPtr(%rbp), %rdi # our object
62      leaq    city(%rdi), %rdi # city field
63      call    writeStr
64      movl    $endl, %edi      # next line (nb: I do them
65      call    writeStr        # individually so it's easier
66
67      movl    $stateMsg, %edi  # state
68      call    writeStr
69      movq    thisPtr(%rbp), %rdi # our object
70      leaq    state(%rdi), %rdi # state field
71      call    writeStr
72      movl    $endl, %edi      # next line (nb: I do them
73      call    writeStr        # individually so it's easier
74
75      movl    $zipMsg, %edi   # zip
76      call    writeStr
77      movq    thisPtr(%rbp), %rdi # our object
78      leaq    zip(%rdi), %rdi # zip field
79      call    writeStr
80      movl    $endl, %edi      # next line (nb: I do them
81      call    writeStr        # individually so it's easier
82
83      movl    $0, %eax        # return 0
84      movq    %rbp, %rsp       # remove local vars
85      popq    %rbp            # restore caller's frame ptr
86      ret

```

See Section E.11 for `writeStr` and `readLn`.

## E.14 Fractional Numbers

14 -1

```

1  /*
2  * floatLoop.c
3  * shows how round off error breaks a loop control variable
4  * Bob Plantz - 1 July 2009

```

```
5  */
6
7 #include <stdio.h>
8
9 int main()
10 {
11     float number;
12     int counter = 10;
13
14     number = 0.5;
15     while ((number != 0.0) && (counter != 0))
16     {
17         printf("number = %f and counter = %i\n", number, counter);
18
19         number -= 0.1;    // change to 0.0625 to fix
20         counter -= 1;
21     }
22
23     return 0;
24 }
```

---

**14 -2**

```
1 /*
2 * floatRoundoff.c
3 * shows the effects of adding a small float to a large one.
4 * Bob Plantz - 1 July 2009
5 */
6
7 #include <stdio.h>
8
9 int main()
10 {
11     float fNumber = 2147483646.0;
12     int iNumber = 2147483646;
13
14     printf("Before adding the float is %f and the integer is %i\n",
15            fNumber, iNumber);
16     fNumber += 1.0;
17     iNumber += 1;
18     printf("After adding 1 the float is %f and the integer is %i\n",
19            fNumber, iNumber);
20
21     return 0;
22 }
```

---

**14 -5** The following program is provided for you to work with these conversions.

```
1 /*
2 * float2hex.c
3 * allows user to see bit pattern of a float
4 * Bob Plantz - 1 July 2009
5 */
6
```

```

7 #include <stdio.h>
8
9 int main()
10 {
11     float number;
12     unsigned int *ptr = (unsigned int *)&number;
13     char ans[50];
14
15     *ans = 'y';
16     while ((*ans == 'y') || (*ans == 'Y'))
17     {
18         printf("Enter a decimal number: ");
19         scanf("%f", &number);
20         printf("%f => %#0x\n", number, *ptr);
21
22         printf("Continue (y/n)? ");
23         scanf("%s", ans);
24     }
25
26     return 0;
27 }
```

---

- |             |             |
|-------------|-------------|
| a) 3f800000 | e) c5435500 |
| b) bdcccccd | f) 3ea8f5c3 |
| c) 44faa000 |             |
| d) 3b800000 | g) 4048f5c3 |

**14 -6** The following program is provided for you to work with these conversion.

---

```

1 /*
2  * hex2float.c
3  * converts hex pattern to float
4  * Bob Plantz - 1 July 2009
5 */
6
7 #include <stdio.h>
8
9 int main()
10 {
11     unsigned int number;
12     float *ptr = (float *)&number;
13     char ans[50];
14
15     *ans = 'y';
16     while ((*ans == 'y') || (*ans == 'Y'))
17     {
18         printf("Enter a hex number: ");
19         scanf("%x", &number);
20         printf("%#0x => %f\n", number, *ptr);
21
22         printf("Continue (y/n)? ");
```

```

23         scanf("%S", ans);
24     }
25
26     return 0;
27 }
```

---

- |              |               |
|--------------|---------------|
| a) +2.0      | e) 100.03125  |
| b) -1.0      | f) 1.2        |
| c) +0.0625   | g) 123.449997 |
| d) -16.03125 | h) -54.320999 |

**14 -7** The bit pattern for +2.0 is 01000...0. Because IEEE 754 uses a biased exponent format, all the floating point numbers in the range 0.0 – +2.0 are within the bit pattern range 00000...0 – 01000...0. So half the positive floating point numbers are in the range 00000...0 – 00111...0, and the other half in the range 01000...0 – 01111...1.

The same argument applies to the negative floating point numbers.

**14 -8**

```

1      .file   "casting.c"
2      .section    .rodata
3 .LC0:    .string "Enter an integer: "
4 .LC1:    .string "%i"
5 .LC3:    .string "%i + %lf = %lf\n"
6
7 .text
8 .globl main
9 .type  main, @function
10
11 main:
12     pushq %rbp
13     movq %rsp, %rbp
14     subq $48, %rsp
15     movl $.LC0, %edi
16     movl $0, %eax
17     call printf
18     leaq -20(%rbp), %rax
19     movq %rax, %rsi
20     movl $.LC1, %edi
21     movl $0, %eax
22     call __isoc99_scanf
23     movabsq $4608218246714312622, %rax # y = 1.23;
24     movq %rax, -16(%rbp)          # store y
25     movl -20(%rbp), %eax        # load x
26     cvtsi2sd    %eax, %xmm0      # xmm0 = (double)x
27     addsd -16(%rbp), %xmm0      # xmm0 += y
28     movsd %xmm0, -8(%rbp)        # z = xmm0
29     movl -20(%rbp), %ecx
30     movq -8(%rbp), %rdx
31     movq -16(%rbp), %rax
```

```

33      movq    %rdx, -40(%rbp)
34      movsd   -40(%rbp), %xmm1
35      movq    %rax, -40(%rbp)
36      movsd   -40(%rbp), %xmm0
37      movl    %ecx, %esi
38      movl    $.LC3, %edi
39      movl    $2, %eax
40      call    printf
41      movl    $0, %eax
42      leave
43      ret
44      .size   main, .-main
45      .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
46      .section .note.GNU-stack,"",@progbits

```

---

## E.15 Interrupts and Exceptions

**15 -1**

```

1 # myCatC.s
2 # Writes a file to standard out
3 # Runs in C environment, but does not use C libraries.
4 # Bob Plantz - 1 July 2009
5
6 # Useful constants
7     .equ    STDIN,0
8     .equ    STDOUT,1
9     .equ    theArg,8
10    # from asm/unistd_64.h
11    .equ    READ,0
12    .equ    WRITE,1
13    .equ    OPEN,2
14    .equ    CLOSE,3
15    .equ    EXIT,60
16    # from bits/fcntl.h
17    .equ    O_RDONLY,0
18    .equ    O_WRONLY,1
19    .equ    O_RDWR,3
20    # Stack frame
21    .equ    aLetter,-16
22    .equ    fd, -8
23    .equ    localSize,-16
24    # Code
25    .text          # switch to text segment
26    .globl  main
27    .type   main, @function
28 main:
29    pushq   %rbp      # save caller's frame pointer
30    movq    %rsp, %rbp  # establish our frame pointer
31    addq    $localSize, %rsp  # for local variable
32
33    movl    $OPEN, %eax      # open the file

```

```
34     movq    theArg(%rsi), %rdi # the filename
35     movl    $0_RDONLY, %esi   # read only
36     syscall
37     movl    %eax, fd(%rbp)   # save file descriptor
38
39     movl    $READ, %eax
40     movl    $1, %edx         # 1 character
41     leaq    aLetter(%rbp), %rsi # place to store character
42     movl    fd(%rbp), %edi   # standard in
43     syscall               # request kernel service
44
45 writeLoop:
46     cmpl    $0, %eax        # any chars?
47     je      allDone         # no, must be end of file
48     movl    $1, %edx         # yes, 1 character
49     leaq    aLetter(%rbp), %rsi # place to store character
50     movl    $STDOUT, %edi    # standard out
51     movl    $WRITE, %eax
52     syscall               # request kernel service
53
54     movl    $READ, %eax
55     movl    $1, %edx
56     leaq    aLetter(%rbp), %rsi # place to store character
57     movl    fd(%rbp), %edi   # standard in
58     syscall               # request kernel service
59     jmp     writeLoop       # check the char
60 allDone:
61     movl    $CLOSE, %eax    # close the file
62     movl    fd(%rbp), %edi # file descriptor
63     syscall               # request kernel service
64     movq    %rbp, %rsp
65
66     popq    %rbp            # restore caller's frame pointer
67     movl    $EXIT, %eax
68     syscall
```

# Bibliography

- [1] Peter Abel. *IBM PC Assembly Language and Programming*, Fifth Edition. Prentice-Hall, 2001
- [2] *AMD64 Architecture Programmer's Manual, Volume 1: Application Programming*; <http://developer.amd.com/devguides.jsp>
- [3] *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*; <http://developer.amd.com/devguides.jsp>
- [4] *AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions Programming*; <http://developer.amd.com/devguides.jsp>
- [5] *AMD64 Architecture Programmer's Manual, Volume 4: 128-Bit Media Instructions*; <http://developer.amd.com/devguides.jsp>
- [6] *AMD64 Architecture Programmer's Manual, Volume 5: 64-Bit Media and x87 Floating-Point Instructions*; <http://developer.amd.com/devguides.jsp>
- [7] Jonathan Bartlett. *Programming from the Ground Up*. Bartlett Publishing, 2004
- [8] Barry B. Brey. *The Intel Microprocessors*, Fifth Edition. Prentice Hall, 2000
- [9] Randal E. Bryant and David R. O'Hallaron. *Computer Systems*. Prentice Hall, 2003
- [10] *C programming language standard ISO/IEC 9899:TC3*. Committee Draft, September 7, 2007.
- [11] Richard C. Detmer. *Introduction to 80x86 Assembly Language and Computer Architecture*. Jones and Bartlett Publishers, 2001
- [12] Jeff Duntemann. *Assembly Language Step-By-Step: Programming with DOS and Linux*, Second Edition. John Wiley & Sons, 2000
- [13] *ELF-64 Object File Format*, Version 1.5 Draft 2, 1998; <http://busybox.net/cgi-bin/viewcvs.cgi/trunk/docs/elf-64-gen.pdf>
- [14] *IA-32 Intel® 64 and IA-32 Architecture Software Developer's Manual, Volume 1: Basic Architecture*; <http://www.intel.com/products/processor/manuals/index.htm>
- [15] *IA-32 Intel® 64 and IA-32 Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference A-M*; <http://www.intel.com/products/processor/manuals/index.htm>
- [16] *IA-32 Intel® 64 and IA-32 Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference N-Z*; <http://www.intel.com/products/processor/manuals/index.htm>

- [17] *IA-32 Intel® 64 and IA-32 Architecture Software Developer's Manual, Volume 3A: System Programming Guide;*  
<http://www.intel.com/products/processor/manuals/index.htm>
- [18] *IA-32 Intel® 64 and IA-32 Architecture Software Developer's Manual, Volume 3B: System Programming Guide;*  
<http://www.intel.com/products/processor/manuals/index.htm>
- [19] Kip R. Irvine. *Assembly Language for Intel-Based Computers*, Fourth Edition. Prentice Hall, 2003
- [20] Bruce F. Katz. *Digital Design: From Gates to Intelligent Machines*. Da Vinci Engineering Press, 2006
- [21] John R. Levine. *Linkers & Loaders*. Elsevier Science & Technology Books, 1999
- [22] Mike Loukides and Andy Oram. *Programming with GNU Software*. O'Reilly, 1997
- [23] M. Morris Mano. *Digital Design, Third Edition*. Prentice Hall, 2002
- [24] Alan B. Marcovitz. *Introduction to Logic Design, Second Edition*. McGraw-Hill, 2005
- [25] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface AMD64 Architecture Processor Supplement, Draft Version 0.99*, December 7, 2007; <http://www.x86-64.org/documentation.html>
- [26] *Merriam-Webster's Online Dictionary*; <http://m-w.com>
- [27] Bob Neveln. *Linux Assembly Language Programming*. Prentice Hall, 2000
- [28] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Third Edition*. Morgan Kaufmann, 2005
- [29] Richard M. Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB*. GNU Press, 2003
- [30] Richard M. Stallman and Roland McGrath. *GNU Make*. GNU Press, 2002
- [31] William Stallings. *Computer Organization & Architecture: Designing for Performance*, Sixth Edition. Prentice Hall, 2002
- [32] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994
- [33] *System V Application Binary Interface, Intel386™ Architecture Processor Support, Fourth Edition*, The SCO Group, 1997;  
<http://www.sco.com/developers/devspecs/>
- [34] Andrew S. Tanenbaum. *Structured Computer Organization*, Fifth Edition. Prentice Hall, 2006
- [35] John von Neumann. *First Draft of a Report on the EDVAC* Moore School of Electrical Engineering, University of Pennsylvania, 1945

# Index

- activation record, 267
- active-low, 108
- adder
  - full, 92
  - half, 92
- addition,
  - binary, 31
  - hexadecimal, 32
- address, memory, 11
  - symbolic name for, 13
- addressing mode, 174
  - base register plus offset, 188
  - immediate data, 175, 228
  - indexed, 328
  - register direct, 174, 228
  - rip-relative, 240
- adjacency property, 70
- algebra
  - Boolean, 61
- alternating current, 78
- ALU, 131
- AND, 61
- antifuse, 100
- Arithmetic Logic Unit, 131
- array, 327
- ASCII, 21
- assembler, 163
- assembler directive, 151
  - .ascii, 173
  - .asciz, 173
  - .byte, 173
  - .equ, 194
  - .globl, 153
  - .include, 340
  - .long, 173
  - .quad, 173
  - .space, 173
  - .string, 173
  - .text, 152
  - .word, 173
- assembly language, 150
  - efficiency, 1
  - required, 1
- assembly language mnemonic, 151
- assignment operator, 208, 210
- asynchronous D flip-flop, 112
- AT&T syntax, 157
- base, 8
- base pointer, 135
- basic data types, 48
- BCD code, 55
- Binary Coded Decimal, 55
- binary point, 358
- bit, 7
- bit mask, 307
- bitwise logical operators, 51
- Boolean algebra, 61
- Boolean algebra properties
  - associative, 63
  - commutative, 63
  - complement, 64
  - distributive, 64
  - idempotent, 64
  - identity, 63
  - involution, 64
  - null, 63
- Boolean expressions
  - canonical product, 66
  - canonical sum, 65
  - maxterm, 66
  - minterm, 65
  - product of maxterms, 66
  - product of sums, 66
  - product term, 65
  - sum of minterms, 65
  - sum of products, 65
  - sum term, 66
- borrow, 34
- branch point, 118
- bus, 4, 135
  - address, 4, 135
  - asynchronous, 397
  - control, 4, 135
  - data, 4, 135
  - synchronous, 397

- timing, 396
- byte, 7
- C-style string, 23
- call stack, 176
- canonical product, 66
- canonical sum, 65
- Carry Flag, 30, 37, 45
- Central Processing Unit, 3, 129
- CF, 37, 45
- circuit
  - combinational, 91
- clock, 105
- clock generator, 105
- clock pulses, 105
- COBOL, 56
- comment field, 152
- comment line, 151
- compile, 148
- compiler-generated label, 158
- complement, 62
- condition codes, 135
- control characters, 21
- Control Unit, 131
- control unit, 6
- convert
  - binary to decimal, 9
  - binary to signed decimal, 40
  - hexadecimal to decimal, 9
  - signed decimal-to-binary, 41
  - unsigned decimal to binary, 9
- CPU, 3, 129
  - block diagram, 130
  - overview, 130
- current, 77
- data
  - storing in memory, 13
- data types, 13
- debugger, 17
- decimal fractions, 358
- decoder, 95
- DeMorgan's Law, 64
- device handler, 399
- division, 315
- D latch, 110
- do-while, 250
- don't care, 77
- DRAM, 126
- effective address, 191
- electronics, 77
  - AC, 78
- amp, 77
- ampere, 77
- battery, 78
- capacitance, 78
- capacitor, 80
- coulomb, 77
- DC, 78
- direct current, 78
- inductance, 78
- inductor, 82
- ohms, 79
- parallel, 80
- passive elements, 78
- power supply, 78
- resistance, 78
- resistor, 79
- series, 79
- time constant, 81
- transient, 78
- voltage, 77
- voltage level, 78
- volts, 77
- watt, 78
- ELF, 152
- ELF:section, 152
- ELF:segment, 153
- endián
  - big, 20
  - little, 20, 142
- exception processing cycle, 387
- Executable and Linking Format, 152
- finite state machine, 104
- fixed point, 359
- Flags Register, 131
- flip-flop
  - D, 111
  - JK, 114
  - T, 112
- floating point, 360
  - errors, 361
  - extended format, 371
  - f<sub>p</sub>n registers, 371
  - limitation, 363
  - range, 361
  - stack, 371
  - x87, 365
- fractional values, 358
- FSM, 104
- function
  - called, 281, 282
  - calling, 281

- designing, 197
- epilogue, 155
- prologue, 155
- writing, 201
- functions
  - 32-bit mode, 283
  - 64-bit mode, 273
- gate
  - AND, 61
  - NAND, 87
  - NOR, 87
  - NOT, 62
  - OR, 62
  - XOR, 76
- gate descriptor, 385
- gdb, 17
  - commands, 17, 140, 424
- Gray code, 56
- handler, 385
- Harvard architecture, 4
- hexadecimal, 6, 7
  - human convenience, 16
- I/O, 3
  - devices, 4
  - isolated, 398
  - memory-mapped, 398, 399
  - programming, 4
- IDE, 147
- identifier, 151, 152
- IEEE 754, 363
  - exponent bias, 364
  - hidden bit, 363
  - size, 363
- if-else, 250
- impedance, 79
- implicit argument, 345
- Input/Output, 3
- instruction
  - add, 214, 219
  - and, 290
  - call, 173
  - cbtw, 246
  - cmp, 237
  - dec, 249
  - div, 315
  - idiv, 317
  - imul, 310
  - in, 398, 399
  - inc, 248
  - ja, 239
  - jae, 239
  - jb, 239
  - jbe, 239
  - jg, 240
  - jge, 240
  - jl, 240
  - jle, 240
  - jmp, 241
  - lea, 191
  - leave, 192
  - mov, 156
  - movs, 245
  - movz, 245
  - mul, 309
  - neg, 322
  - or, 290
  - pop, 181
  - push, 181
  - ret, 192
  - sal, 302
  - sar, 301
  - shl, 302
  - shr, 301
  - sub, 215
  - syscall, 201
  - test, 238
  - xor, 290
- instruction execution cycle, 137
- instruction fetch, 137
- Instruction Pointer, 131
- instruction pointer, 134
- instruction prefixes, 225
- Instruction Register, 131
- instruction register, 136
- instructions
  - cmovef, 260
  - cvtsi2sd, 370
  - in, 398
  - iret, 388
  - out, 399
  - syscall, 388
  - sysret, 388
- instruction set architecture, 1
- integer
  - signed decimal, 37
  - unsigned decimal, 36
- Integrated Development Environment, 147
- interrupt handler, 385, 410
- invert, 62
- ISA, 1
- label field, 151

- least significant digit, 8  
library, I/O, 49  
line-oriented, 151  
line buffered, 25  
linker, 165  
listing file, 222  
literal, 65  
local variables, 193, 200, 201  
location, memory, 11  
logic  
    sequential, 103  
logical operators, 290  
logic circuit  
    combinational, 91  
    sequential, 103  
logic gate, 61  
Loop Control Variable, 236  
  
machine code, 220  
mantissa, 358  
master/slave, 111  
maxterm, 66  
Mealy machine, 105  
member data, 345  
member function, 345  
Memory, 3, 11  
memory  
    data allocation, 173  
    timing, 395  
memory segment:code, 152  
memory segment:data, 152  
memory segment:heap, 153  
memory segment:stack, 152  
memory segment:text, 152  
minimal product of sums, 68  
minimal sum of products, 68  
minterm, 65  
mnemonic, 150  
mode  
    32-bit, 129  
    64-bit, 129  
    compatibility, 129  
    IA-32e, 129  
    long, 129  
Moore machine, 105  
most significant digit, 8  
multiplexer, 99  
multiplication, 308  
mux, 99  
  
name mangling, 346  
NAND, 86, 87  
negating, 40  
  
negation, 322  
negative, 39  
NOR, 87  
normalize, 361  
NOT, 62  
number systems  
    binary, 6, 8  
    decimal, 6, 15  
    hexadecimal, 6, 7, 15  
    octal, 7  
  
object, 343  
object, C++, 345  
object file, 152  
octal, 7  
OF, 37, 43, 45  
offset, 240  
one's complement, 40  
operand field, 151  
operation field, 151  
OR, 62  
Overflow Flag, 30, 43, 45  
  
PAL, 103  
parity, 21  
    even, 21  
    odd, 21  
pass  
    by pointer, 267  
    by reference, 267, 338  
    by value, 267, 337  
    updates, 267  
penultimate carry, 43  
pipeline, 118  
PLD, 100  
positional notation, 8  
printf  
    calling, 195  
printf, 14  
    conversion codes, 14  
privilege level, 386  
procedural programming, 14  
product of maxterms, 66  
product of sums, 66  
product term, 65  
program, 4  
Programmable Array Logic, 103  
Programmable Logic Device, 100  
programming  
    bit patterns, 8  
pseudo op, 151  
  
radix, 8

RAM, 12  
Random Access Memory, 12  
read, 23, 49  
Read Only Memory, 102  
real number, 360  
record, 333  
reduced radix complement, 40  
red zone, 270  
register  
    general-purpose, 132  
    names, 132  
register file, 121  
registers, 121, 131  
register storage class, 139  
repetition, 235  
return address, 174  
return value, 155, 267  
REX, 225  
rflags, 30, 37, 43  
ROM, 12, 102  
round off, 359  
  
scalar, 365  
scanf  
    calling, 195  
scanf, 14  
    conversion codes, 14  
section:text, 152  
shift bits, 301  
    left, 302  
    right, 301  
shift register, 122  
short-circuit evaluation, 259  
SIB byte, 226  
sign-extension, 229  
significand, 358  
SIMD, 365  
Single Instruction, Multiple Data, 365  
SRAM, 125  
SR latch  
    Reset, 106  
    Set, 106  
SSE, 366  
    scalar instructions, 367  
    vector instructions, 367  
stack, 159  
    discipline, 177  
    operations, 176  
    overflow, 177  
    pointer, 180  
    underflow, 177  
    viewing, 183  
stack frame, 188, 267  
stack pointer, 135  
stack pointer address, 182  
stack protection, 296  
state, 91  
state diagram, 107  
state table, 106  
stdio.h, 14  
STDOUT\_FILENO, 24  
struct, 333  
    field, 333  
    overall size, 341  
subsystems, 3  
subtraction, 34  
    hexadecimal, 36  
sum of minterms, 65  
sum of products, 65  
sum term, 66  
switch, 6, 30  
system call, 23, 49, 171  
  
this pointer, 349  
time constant, 83  
toggle, 111  
transistor  
    drain, 84  
    gate, 84  
    source, 84  
tri-state buffer, 124  
truth table, 51, 61  
two's complement, 37  
    computing, 40  
    defined, 39  
two's complement code, 37  
type casting, 305  
  
unistd.h, 24  
  
variable  
    automatic, 192  
    static, 192  
variable argument list, 271  
variables  
    local, 188  
    vector, 365, 387  
von Neumann bottleneck, 4  
  
while statement, 236  
write, 23, 49  
  
x86 architecture, 1