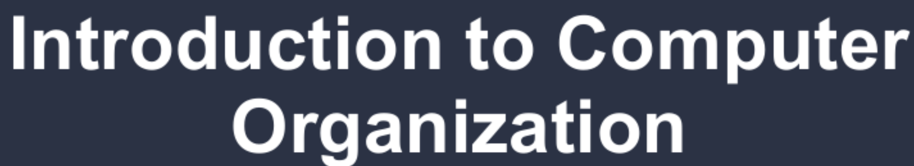


Introduction to Computer Organization

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

December 2013

Copyright Notice

Copyright ©2008, ©2009, ©2010, ©2011, ©2012, ©2013 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

Preface **xiv**

1 Introduction **1**
1.1 Computer Subsystems . . . 3
1.2 How the Subsystems Interact . . . 4

2 Data Storage Formats **6**
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 `gdb` . . . 17
2.7 ASCII Character Code . . . 21
2.8 `write` and `read` Functions . . . 23
2.9 Exercises . . . 26

3 Computer Arithmetic **30**
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

4 Logic Gates **61**
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

5 Logic Circuits 91
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

6 Central Processing Unit 129
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

7 Programming in Assembly Language 147
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

8 Program Data – Input, Store, Output 171
8.1 Calling `write` 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 `printf` and `scanf` in 64-bit Mode . . . 195
8.5 Designing the Local Variable Portion of the Call Stack . . . 197
8.6 Using `syscall` 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

9 Computer Operations 208
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 `mov` Instruction . . . 227
9.3.7 The `add` Instruction . . . 229
9.4 Instructions Introduced Thus Far . . . 230
9.4.1 Instructions . . . 231
9.4.2 Addressing Modes . . . 231
9.5 Exercises . . . 232

10 Program Flow Constructs 235
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 `while` Loop . . . 242
10.2 Binary Decisions . . . 250
10.2.1 Short-Circuit Evaluation . . . 259
10.2.2 Conditional Move . . . 260
10.3 Instructions Introduced Thus Far . . . 261
10.3.1 Instructions . . . 262
10.3.2 Addressing Modes . . . 263
10.4 Exercises . . . 263

11 Writing Your Own Functions 267
11.1 Overview of Passing Arguments . . . 267
11.2 More Than Six Arguments, 64-Bit Mode . . . 274
11.3 Interface Between Functions, 32-Bit Mode . . . 284
11.4 Instructions Introduced Thus Far . . . 287
11.4.1 Instructions . . . 287
11.4.2 Addressing Modes . . . 288
11.5 Exercises . . . 288

12 Bit Operations; Multiplication and Division 290
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 . . . 323
12.6.2 Addressing Modes . . . 324
12.7 Exercises . . . 325

13 Data Structures 327
13.1 Arrays . . . 327
13.2 `structs` (Records) . . . 333
13.3 `structs` as Function Arguments . . . 338
13.4 Structs as C++ Objects . . . 343
13.5 Instructions Introduced Thus Far . . . 353
13.5.1 Instructions . . . 354
13.5.2 Addressing Modes . . . 355
13.6 Exercises . . . 356

14 Fractional Numbers 358
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 . . . 370
14.5.3 3DNow! Floating Point . . . 375
14.6 Comments About Numerical Accuracy . . . 376
14.7 Instructions Introduced Thus Far . . . 376
14.7.1 Instructions . . . 376
14.7.2 Addressing Modes . . . 379
14.8 Exercises . . . 379

15 Interrupts and Exceptions 383
15.1 Hardware Interrupts . . . 384
15.2 Exceptions . . . 385
15.3 Software Interrupts . . . 385
15.4 CPU Response to an Interrupt or Exception . . . 386
15.5 Return from Interrupt/Exception . . . 387
15.6 The `syscall` and `sysret` Instructions . . . 387
15.7 Summary . . . 390
15.8 Instructions Introduced Thus Far . . . 390
15.8.1 Instructions . . . 390
15.8.2 Addressing Modes . . . 393
15.9 Exercises . . . 393

16 Input/Output **394**
16.1 Memory Timing 394
16.2 I/O Device Timing 395
16.3 Bus Timing 395
16.4 I/O Interfacing 396
16.5 I/O Ports 397
16.6 Programming Issues 398
16.7 Interrupt-Driven I/O 409
16.8 I/O Instructions 409
16.9 Exercises 410

A Reference Material **411**
A.1 Basic Logic Gates 411
A.2 Register Names 412
A.3 Argument Order in Registers 412
A.4 Register Usage 413
A.5 Assembly Language Instructions Used in This Book 414
A.6 Addressing Modes 416

B Using GNU `make` to Build Programs **417**

C Using the `gdb` Debugger for Assembly Language **423**

D Embedding Assembly Code in a C Function **429**

E Exercise Solutions **434**
E.2 Data Storage Formats 434
E.3 Computer Arithmetic 442
E.4 Logic Gates 453
E.5 Logic Circuits 456
E.6 Central Processing Unit 457
E.7 Programming in Assembly Language 459
E.8 Program Data – Input, Store, Output 464
E.9 Computer Operations 467
E.10 Program Flow Constructs 475
E.11 Writing Your Own Functions 487
E.12 Bit Operations; Multiplication and Division 494
E.13 Data Structures 509
E.14 Fractional Numbers 533
E.15 Interrupts and Exceptions 537

Bibliography **539**

Index **540**

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 4.26. 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
4.16 A “don’t care” cell on a Karnaugh map. 77
4.17 Karnaugh map for `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/subtracter. 96
5.6 Circuit for a 3×8 decoder with enable. 98
5.7 Full adder implemented with 3×8 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 $Control$ 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 20×2^{20} address decoder. 127
5.37 Addressing 1 MB of memory with two 10×2^{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. 272
11.2 Arguments 7 – 9 are passed on the stack to the `sumNine` function. 277
11.3 Arguments and local variables in the stack frame, `sumNine` function. 278
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. 396

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 3×8 decoder with $enable$. 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 `syscall` 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 `je` instruction. 241
10.5 Instructions to double data size. 246

11.1 Argument register save area in stack frame. 272

12.1 Bit patterns (in binary) of the ASCII numerals and the corresponding 32-bit `ints`. 308
12.2 Register usage for the `mul` instruction. 309
12.3 Register usage for the div instruction. 315
12.4 Instructions to set up the dividend registers. 317

14.1 MXCSR status register. 367
14.2 SSE scalar floating point conversion instructions. 368
14.3 Some SSE floating point a4rithmetic and data movement instructions. . . 368
14.4 x87 Status Word. 371
14.5 A sampling of x87 floating point instructions. 373

15.1 Some system call codes for the `syscall` instruction. 389

Listings

2.1 Using printf 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 gdb to view CPU registers. 140
7.1 A “null” program (C). 149
7.2 A “null” program (gcc assembly language). 149
7.3 A “null” program (programmer assembly language). 150
7.4 A “null” program (gcc 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 (gcc 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 write system call function (C). 172
8.2 “Hello world” program using the write system call function (gcc assembly language). 172
8.3 A C implementation of a stack. 177
8.4 Save and restore the contents of the rbx and r12 – r15 registers. 182
8.5 Echoing characters entered from the keyboard (gcc assembly language). 189
8.6 Echoing characters entered from the keyboard (programmer assembly language). 193
8.7 Calling printf and scanf to write and read formatted I/O (C). 195
8.8 Calling printf and scanf to write and read formatted I/O (gcc assembly language). 195
8.9 Calling printf and scanf to write and read formatted I/O (programmer assembly language). 196
8.10 Some local variables (C). 198
8.11 Some local variables (gcc 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 syscall instruction. 202
8.15 Displaying four characters on the screen using the write system call function in assembly language. 204
9.1 Assignment to a register variable (C). 209
9.2 Assignment to a register variable (gcc 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 (gcc 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).247
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). 252
10.9 General structure of an `if-else` construct. 254
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. 264
11.1 Passing arguments to a function (C). 269
11.2 Accessing arguments in the `sumInts` function from Listing 11.1 (`gcc` assembly language). 270
11.3 Accessing arguments in the `sumInts` function from Listing 11.1 (programmer assembly language) 273
11.4 Passing more than six arguments to a function (C). 274
11.5 Passing more than six arguments to a function (`gcc` assembly language). 276
11.6 Passing more than six arguments to a function (programmer assembly language). 281
11.7 Passing more than six arguments to a function (`gcc` assembly language, 32-bit). 285
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). 321
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. . . . 367
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). . . . 373
14.5 Use float for Loop Control Variable? 379
14.6 Are floats accurate? 380
14.7 Casting integer to float in C. . . . 381
14.8 Casting integer to float in assembly language. . . . 381
15.1 Using `syscall` to `cat` a file. . . . 387
16.1 Sketch of basic I/O functions using memory-mapped I/O — C version. . . . 398
16.2 Memory-mapped I/O in assembly language. . . . 401
16.3 Sketch of basic I/O functions, isolated I/O — C version. . . . 404
16.4 Isolated I/O in assembly language. . . . 405
B.1 An example of a `Makefile` for an assembly language program with one source file. . . . 418
B.2 An example of a `Makefile` for a program with both C and assembly language source files. . . . 419
B.3 `Makefile` variables. . . . 419
B.4 Incomplete `Makefile`. . . . 421
D.1 Embedding an assembly language instruction in a C function (C). . . . 429
D.2 Embedding an assembly language instruction in a C function `gcc` assembly language. . . . 430
D.3 Embedding more than one assembly language instruction in a C function and specifying a register (C). . . . 431
D.4 Embedding more than one assembly language instruction in a C function and specifying a register (`gcc` assembly language). . . . 432

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 and be done on any of the common Linux distributions that has 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 387).

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 539. 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`. 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`. I welcome your feedback (`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 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 LaTeX 2ε to typeset and draw the figures for this book. The fonts are DejaVu (`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.

Three students deserve special thanks, David Tran, Zack Gold, and Jim O'Hara. They used this book in a class taught by Mike Lyle at Santa Rosa Junior College, David in Fall 2010, Zack in Fall 2011, and Jim in Fall 2013. All three 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, LaTeX 2ε, 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:

General concept	***C/C++ implementation***
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.

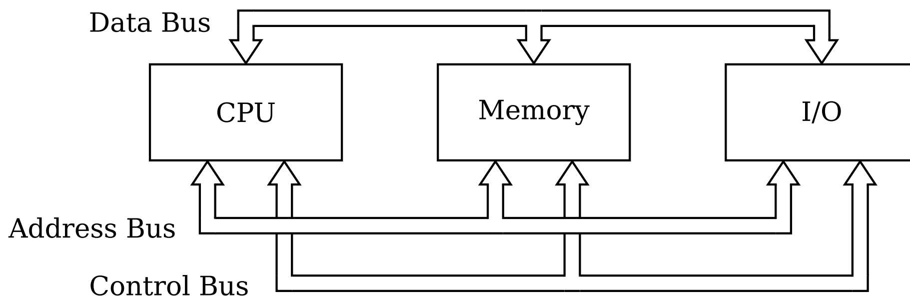


Figure 1.1: Subsystems of a computer. The CPU, Memory, and I/O subsystems communicate with one another via the three buses.

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`

"*Bi*nary dig*it*" 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`.

Four binary digits (bits)	One hexadecimal digit
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,

$$0110\ 1100\ 0010\ 1010_2 = 6c2a_{16} \tag{2.1}$$

and

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

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, lowercase 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.

	Prefix	**Example**	**32-bit pattern** (binary)
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`	`00 000 000 000 000 000 000 000 001 010 011`

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, . . . , 1, 0, -1, -2, ... 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

$$1 \times 100 + 2 \times 10 + 3 \times 1 \tag{2.3}$$

or

$$1 \times 10^2 \times 10^1 + 3 \times 10^0 \tag{2.4}$$

The right-most digit (3 in Equation2.4) 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:

$$d_{n-1} \times r^{n-1} + d_{n-2} \times r^{n-2} + \ldots d_1 \times r^1 + d_0 \times r^0 \tag{2.5}$$

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

$$d_{n-1} \times 2^{n-1} + d_{n-2} \times 2^{n-2} + \ldots d_1 \times 2^1 + d_0 \times 2^0 \tag{2.6}$$

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

$$\mathtt{1} \times 2^7 + \mathtt{0} \times 2^6 + \mathtt{1} \times 2^5 + \mathtt{0} \times 2^4 + \mathtt{0} \times 2^3 + \mathtt{1} \times 2^2 + \mathtt{0} \times 2^1 + \mathtt{1} \times 2^0 \tag{2.7}$$

If we evaluate this expression in decimal, we get

$$128 + 0 + 32 + 0 + 0 + 4 + 1 + 1 = 165_{10} \tag{2.8}$$

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:

$$N = d_{n-1} \times 2^{n-1} + d_{n-2} \times 2^{n-2} + \ldots + d_1 \times 2^1 + d_0 \times 2^0 \tag{2.9}$$

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

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

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,

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

Dividing both sides of Equation 2.11 by two:

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

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 ⇐ theInteger;
2 while quotient ≠ 0 do
3     nextBit ⇐ quotient % 2;
4     quotient ⇐ 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

$$\begin{aligned}
123_{10} &= d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0\\
&= \texttt{01111011}_2\\
&= \texttt{7b}_{16}
\end{aligned}$$

□

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 124th 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 957th byte from the beginning of memory will always remain the 957th 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 957th 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*.

Address	Contents	Address	Contents
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 `bffffed0` in memory. If the grade is "A", the programmer would set the bit pattern at location `bffffed0` to $\texttt{00}_{16}$. If the grade is "C", the programmer would set the bit pattern at location `bffffed0` to $\texttt{02}_{16}$. 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 `bffffed0` 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.[1] 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**):

[1] 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 `int`s, `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 `int`s 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`.[2]
- `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"
```

[2] 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...do
(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 *) 0x7fffffffe058
(gdb) print &aFloat
$4 = (float *)  0x7fffffffe05c
```

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 0x7fffffffe058
0x7fffffffe058: 19088743
(gdb) x/1fw 0x7fffffffe05c
0x7fffffffe05c: 19088.7422
```

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

```
(gdb) x/1xw 0x7fffffffe058
0x7fffffffe058:     0x01234567
(gdb) x/4xb 0x7fffffffe058
0x7fffffffe058:     0x67    0x45    0x23    0x01
```

The display of the `anInt` variable in hexadecimal, which is located at memory address `0x7fffffffe058`, 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 (`0x7fffffffe059`). So this row displays each of the bytes stored at the four memory addresses `0x7fffffffe058`, `0x7fffffffe059`, `0x7fffffffe05a`, and `0x7fffffffe05b`.

```
(gdb) x/1fw 0x7fffffffe05c
0x7fffffffe05c:     19088.7422
(gdb) x/1xw 0x7fffffffe05c
0x7fffffffe05c:     0x4695217c
(gdb) x/4xb 0x7fffffffe05c
0x7fffffffe05c:     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 `0x7fffffffe058` above. We can rearrange this display to show the bit patterns at each of the four locations:

```
7fffffffe058: 67
7fffffffe059: 45
7fffffffe05a: 23
7fffffffe05b: 01
```

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

```
7fffffffe058: 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 0x7fffffffe058
0x7fffffffe058:     0x01234567
(gdb) x/4xb 0x7fffffffe058            [BIG ENDIAN COMPUTER, NOT OURS!]
0x7fffffffe058:     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` ... `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

bit pat.	char	bit pat.	char	bit pat.	char	bit pat.	char
00	NUL (Null)	20	(Space)	40	@	60	‘
01	SOH (Start of Hdng)	21	!	41	A	61	a
02	STX (Start of Text)	22	"	42	B	62	b
03	ETX (End of Text)	23	#	43	C	63	c
04	EOT (End of Transmit)	24	$	44	D	64	d
05	ENQ (Enquiry)	25	%	45	E	65	e
06	ACK (Acknowledge)	26	&	46	F	66	f
07	BEL (Bell)	27	’	47	G	67	g
08	BS (Backspace)	28	(	48	H	68	h
09	HT (Horizontal Tab)	29	)	49	I	69	i
0a	LF (Line Feed)	2a	*	4a	J	6a	j
0b	VT (Vertical Tab)	2b	+	4b	K	6b	k
0c	FF (Form Feed)	2c	,	4c	L	6c	l
0d	CR (Carriage Return)	2d	-	4d	M	6d	m
0e	SO (Shift Out)	2e	.	4e	N	6e	n
0f	SI (Shift In)	2f	/	4f	O	6f	o
10	DLE (Data-Link Escape)	30	0	50	P	70	p
11	DC1 (Device Control 1)	31	1	51	Q	71	q
12	DC2 (Device Control 2)	32	2	52	R	72	r
13	DC3 (Device Control 3)	33	3	53	S	73	s
14	DC4 (Device Control 4)	34	4	54	T	74	t
15	NAK (Negative ACK)	35	5	55	U	75	u
16	SYN (Synchronous idle)	36	6	56	V	76	v
17	ETB (End of Trans. Block)	37	7	57	W	77	w
18	CAN (Cancel)	38	8	58	X	78	x
19	EM (End of Medium)	39	9	59	Y	79	y
1a	SUB (Substitute)	3a	:	5a	Z	7a	z
1b	ESC (Escape)	3b	;	5b	[	7b	{
1c	FS (File Separator)	3c	<	5c	\	7c	|
1d	GS (Group Separator)	3d	=	5d	]	7d	}
1e	RS (Record Separator)	3e	>	5e	^	7e	~
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.

Address	Contents
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`.[3] 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
```

[3] 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 Foundat
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`
b) `1000 1001 1010 1011`
c) `1111 1110 1101 1100`
d) `0000 0010 0101 0000`

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

a) `83af`
b) `9001`
c) `aaaa`
d) `5555`

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

a) `ffffffff`
b) `7fff58b7def0`
c) 1111_2
d) 1111_{16}
e) 00000000_2
f) 00000000_{16}

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

a) eight bits
b) thirty-two bits
c) sixty-four bits
d) ten bits
e) twenty 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`
b) `01010101`
c) `11110000`
d) `00001111`
e) `10000000`
f) `01100011`
g) `01111011`
h) `11111111`

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

a) `1010101111001101`
b) `0001001000110100`
c) `1111111011011100`
d) `0000011111010000`
e) `1000000000000000`
f) `0000010000000000`
g) `1111111111111111`
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`
b) `50`
c) `ff`
d) `89`
e) `64`
f) `0c`
g) `11`
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`
b) `ffff`
c) `0400`
d) `1111`
e) `8888`
f) `0190`
g) `abcd`
h) `5555`

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

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

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

a) `1024`
b) `1000`
c) `32768`
d) `32767`
e) `256`
f) `65535`
g) `2005`
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 `0x2fffeb96`. 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 `int`s?

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 `anInt` 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.[1] 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:

$$\begin{array}{rrl} 1 & 1 & \longleftarrow carries \\ & 67 & \longleftarrow x \\ + & 79 & \longleftarrow y \\ \hline & 46 & \longleftarrow sum \end{array}$$

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

[1] 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_i ⇐ (x_i + y_i) % 10 ;          // mod operation
3     carry ⇐ (x_i + y_i) / 10 ;          // div operation
4     i ⇐ i + 1;
```

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     sum_i ⇐ (x_i + y_i) % 2 ;          // mod operation
3     carry ⇐ (x_i + y_i) / 2 ;          // div operation
4     i ⇐ i + 1;
```

Example 3-a

Compute the sum of x = `10101011` and y = `11001101`.

Solution:

```
  0  0001  111    ⟵ carries
     1010  1011   ⟵ x
+    0100  1101   ⟵ y
  ---------------
     1111  1000   ⟵ sum
```

This is how the algorithm was applied.

ones place:
sum_0 = `(1 + 1) % 2 = 0`
carry = `(1 + 1) / 2 = 1`

twos place:
  sum_1 = `(1 + 1 + 0) % 2 = 0`
  carry = `(1 + 1 + 0) / 2 = 1`
fours place:
  sum_2 = `(1 + 0 + 1) % 2 = 0`
  carry = `(1 + 0 + 1) / 2 = 1`
eights place:
  sum_3 = `(1 + 1 + 1) % 2 = 1`
  carry = `(1 + 1 + 1) / 2 = 1`
sixteens place:
  sum_4 = `(1 + 0 + 0) % 2 = 1`
  carry = `(1 + 0 + 0) / 2 = 0`
thirty-twos place:
  sum_5 = `(0 + 1 + 0) % 2 = 1`
  carry = `(0 + 1 + 0) / 2 = 0`
sixty-fours place:
  sum_6 = `(0 + 0 + 0) % 2 = 1`
  carry = `(0 + 0 + 0) / 2 = 0`
one hundred twenty-eights place:
  sum_7 = `(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   $sum_i \Leftarrow (x_i + y_i)$ % radix ; `// mod operation`
3   carry $\Leftarrow (x_i + y_i)$ / 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,... and we start from the right. So the binary number 1011 represents:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \tag{3.1}$$

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

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

From Table 2.1 on page 7 we see that $\texttt{1011}_2$ = $\texttt{b}_{16}$, 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 = `0xabcd` and y = `0x6089`.

Solution:

```
1  011     ←— carries
   abcd    ←— x
+  6089    ←— y
   ----
   0c56    ←— sum
```

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:
  sum_0 = `(d + 9) % 16 = 6`
  carry = `(d + 9) / 16 = 1`
sixteens place:
  sum_1 = `(1 + c + 8) % 16 = 5`
  carry = `(1 + c + 8) / 16 = 1`
two hundred fifty-sixes place:
  sum_2 = `(1 + b + 0) % 16 = c`
  carry = `(1 + b + 0) / 16 = 0`
four thousand ninety-sixes place:
  sum_3 = `(0 + a + 6) % 16 = 0`
  carry = `(0 + a + 6) / 16 = 1`

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 ≤ x_i then
 3         difference_i ⇐ x_i − y_i;
 4         borrow ⇐ 0;
 5     else
 6         j ⇐ i + 1;
 7         while x_j = 0 do
 8             j ⇐ j + 1;
 9         for j to i do
10             x_j ⇐ x_j - 1;
11             j ⇐ j - 1;
12             x_j ⇐ x_j + radix;
13     i ⇐ i + 1;
```

This algorithm is not as complicated as it first looks.

Example 3-c

Subtract y = 10101011 from x = 11001101.

Solution:

```
0  0100  010     ←— borrows
   1100  1101    ←— x
-  1010  1011    ←— y
   ----------
   0010  0010    ←— difference
```

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:
 difference_0 = 1 - 1 = 0
twos place:
 Borrow from the fours place in the minuend.
 The borrow becomes 2 in the twos place.
 difference_1 2 - 1 = 1
fours place:
 Since we borrowed 1 from here, the minuend has a 0 left.
 difference_2 = 0 - 0 = 0
eights place:
 difference_3 = 1 - 1 = 0
sixteens place:
 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.
difference_5 = `2 - 1 = 1`

sixty-fours place:
Since we borrowed 1 from here, the minuend has a 0 left.
difference_6 = `0 - 0 = 0`

one hundred twenty-eights place:
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:

$$\begin{array}{rl l}
1 & 101 & \longleftarrow borrows \\
 & 6089 & \longleftarrow x \\
- & \text{ab5d} & \longleftarrow y \\
\hline
 & \text{b52c} & \longleftarrow difference
\end{array}$$

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 $\rightarrow$ `CF` = 0
- borrow from outside $\rightarrow$ `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:

$$\begin{array}{rcccr}
0100_2 & = & 0\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0 & = & 4_{10} \\
+\,0010_2 & = & 0\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 & = & +\,2_{10} \\
\hline
0110_2 & = & 0\times2^3 + 1\times2^2 + 1\times2^1 + 0\times2^0 & = & 6_{10}
\end{array}$$

and `CF` = 0.

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

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

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:

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

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):

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

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.

Four binary digits (bits)	**One hexadecimal digit**	**Decimal equivalent**
`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

$$-8_{10} \leq x \leq +7_{10} \tag{3.3}$$

or

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

The last observation can be generalized for n bits to:

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

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

$$x + (-x) = 2^n \tag{3.6}$$

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:

$$-x = 2^n - x \tag{3.7}$$

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

$$-1_{10} = \texttt{100000000}_2 - \texttt{00000001}_2 = \texttt{11111111}_2 \tag{3.8}$$

or in hexadecimal:

$$-1_{16} = \texttt{100}_{16} - \texttt{01}_{16} = \texttt{ff}_{16} \tag{3.9}$$

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:

$$-x - 1 = 2^n - x - 1 \tag{3.10}$$

Rearranging a little:

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

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:

$$2^8 - 1 = \texttt{11111111}_2 \tag{3.12}$$

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

$$2^n - 1 - x = \texttt{111}\ldots\texttt{111}_2 - x \tag{3.13}$$

where $\texttt{111}\ldots\texttt{111}_2$ 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:

$$\texttt{11111111}_2 - \texttt{00000001}_2 = \texttt{11111110}_2 \tag{3.14}$$

or in hexadecimal:

$$\texttt{f}_{16} - \texttt{01}_{16} = \texttt{fe}_{16} \tag{3.15}$$

Another (simpler) way to look at this is

$$2^n - 1 - x = \text{“flip all the bits in x”} \tag{3.16}$$

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

$$-x - 1 = \text{the reduced radix complement of x} \tag{3.17}$$

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

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

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 ⇐ x';
2 x ⇐ 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 $\texttt{5678}_{16}$ 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:

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

□

Example 3-e

The 16-bit integer $\texttt{8765}_{16}$ 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.

$$\begin{aligned}\text{Take the one's complement} &\Rightarrow 789\text{a}_{16} \\ \text{Add one} &\Rightarrow 789\text{b}_{16}\end{aligned}$$

Compute the decimal equivalent.

$$\begin{aligned}\texttt{789b}_{16} &= 7 \times 4096 + 8 \times 256 + 9 \times 16 + 11 \times 1 \\ &= 28672 + 2048 + 144 + 11 \\ &= +30875_{10}\end{aligned}$$

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

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

□

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

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

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

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

□

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, $\texttt{15}_{16}$ and $\texttt{6f}_{16}$, we get:

```
carry ⟶   0   1 ⟵ penultimate carry
              0001  0101     ⟵ x
          +   0110  1111     ⟵ y
              ----------
              1000  0100     ⟵ sum
```

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:

$$\texttt{OF} = \texttt{CF} \;\hat{}\; \text{penultimate carry} \tag{3.19}$$

where '^' is the exclusive or operator. In the above example

$$\begin{aligned} \texttt{OF} &= \texttt{0} \;\hat{}\; \texttt{1} \\ &= \texttt{1} \end{aligned} \tag{3.20}$$

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:

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

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:

```
carry ⟶ 0   0   ⟵ penultimate carry
            0  ...   ⟵ x
        +   1  ...   ⟵ y
            -------
            1  ...   ⟵ sum
```

   this addition would produce `OF = 0 ^ 0 = 0`.

2. If the penultimate carry is one:

```
carry ⟶ 1   1   ⟵ penultimate carry
            0  ...   ⟵ x
        +   1  ...   ⟵ y
            -------
            0  ...   ⟵ sum
```

   this addition would produce `OF = 1 ^ 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:

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

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

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

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

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:

$$x = \mathtt{0}\ldots$$
$$y = \mathtt{0}\ldots$$

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:

```
carry ⟶ 0   0  ⟵ penultimate carry
            0  ...  ⟵ x
        +   0  ...  ⟵ y
            ----------
            0  ...  ⟵ sum
```

   this addition would produce `OF = 0 ^ 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:

```
carry ⟶ 0   1  ⟵ penultimate carry
            0  ...  ⟵ x
        +   0  ...  ⟵ y
            ----------
            1  ...  ⟵ sum
```

   this addition would produce `OF = 0 ^ 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:

$$x = \mathtt{1}\ldots$$
$$y = \mathtt{1}\ldots$$

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:

```
carry ⟶ 1  0   ⟵ penultimate carry
           1  ...  ⟵ x
       +   1  ...  ⟵ y
           ------
           0  ...  ⟵ sum
```

   this addition would produce `OF = 1 ^ 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:

```
carry ⟶ 1  1   ⟵ penultimate carry
           1  ...  ⟵ x
       +   1  ...  ⟵ y
           ------
           1  ...  ⟵ sum
```

   this addition would produce `OF = 1 ^ 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 to 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.

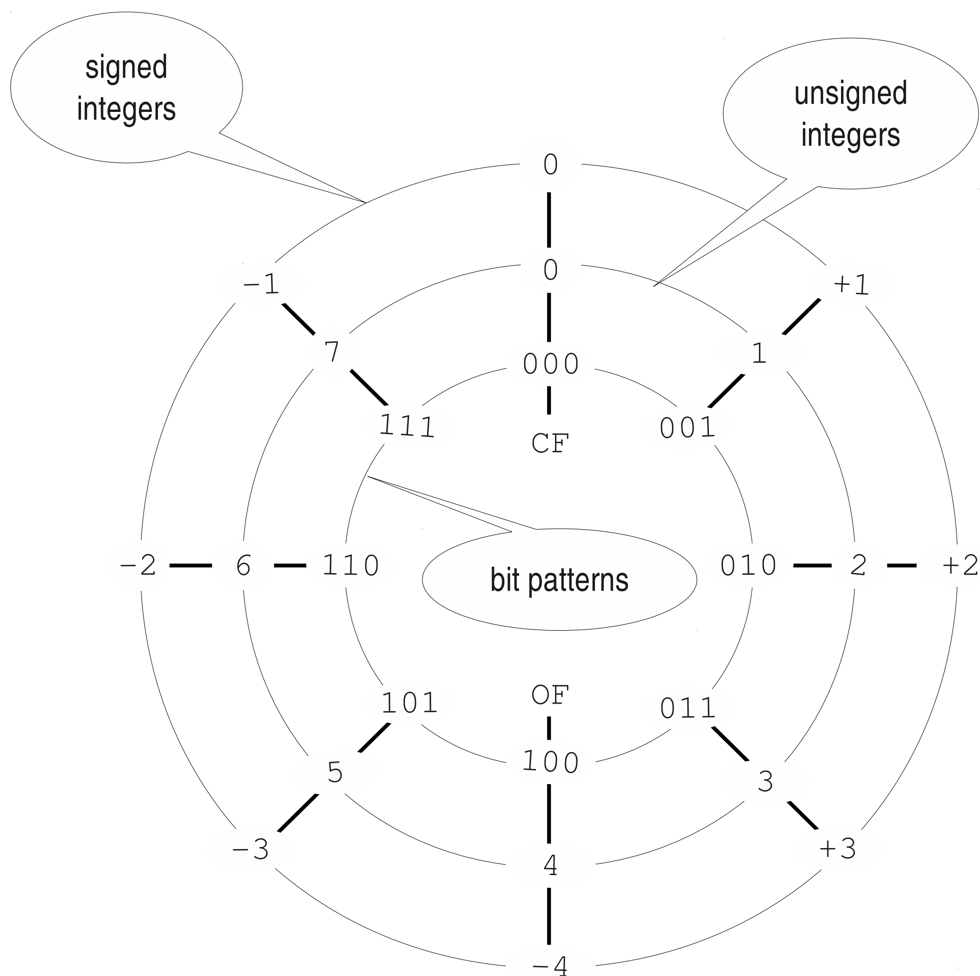


Figure 3.1: “Decoder Ring” for three-bit signed and unsigned integers. Move clockwise when adding numbers, counter-clockwise when subtracting. Crossing over `000` sets the `CF` to one, indicating an error for unsigned integers. Crossing over `100` sets the `OF` to one, indicating an error for signed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

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, `OF` = `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.

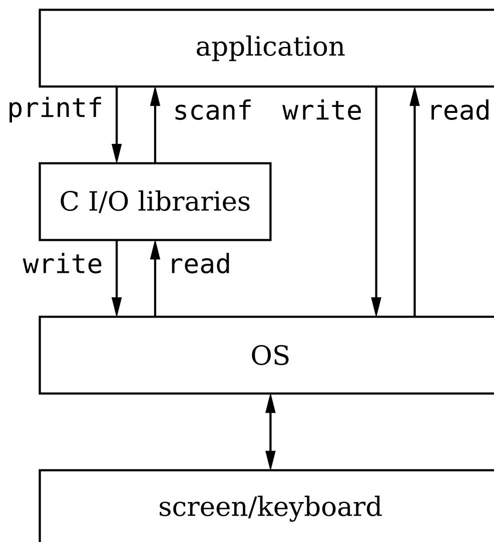


Figure 3.2: Relationship of I/O libraries to application and operating system. An application can use functions in the I/O libraries to convert between keyboard/screen chars and basic data types, or it can directly use the `read` /`write` system calls to transfer raw bytes.

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 «.[2] 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.

[2] In C++ the » and « operators have been overloaded for use with the input and output streams.

```c
/*
 * mulDiv.c
 * Asks user to enter an integer. Then prompts user to enter
 * a power of two to multiply the integer, then another power
 * of two to divide. Assumes that user does not request more
 * than 30 as the power of 2.
 * Bob Plantz - 4 June 2009
 */

#include <stdio.h>

int main(void)
{
    int x;
    int leftShift, rightShift;

    printf("Enter an integer: ");
    scanf("%i", &x);

    printf("Multiply by two raised to the power: ");
    scanf("%i", &leftShift);
    printf("%i x %i = %i\n", x, 1 << leftShift, x << leftShift);

    printf("Divide by two raised to the power: ");
    scanf("%i", &rightShift);
    printf("%i / %i = %i\n", x, 1 << rightShift, x >> rightShift);

    return 0;
}
```

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]	carry[(i-1)]	z[i]	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`*th* bit of `x`; `carry[(i-1)]` is the carry from adding the `(i-1)`*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]	~x[i]
0	1
1	0

Figure 3.4: Truth tables showing bitwise C/C++ operations. `x[i]` is the `i`*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

and

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

or

x	y	x `||` y
0	0	0
0	non-zero	1
non-zero	0	1
non-zero	non-zero	1

complement

x	!x
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
1	0011 0001	0000 0000 0000 0000 0000 0000 0000 0001
2	0011 0010	0000 0000 0000 0000 0000 0000 0000 0010
3	0011 0011	0000 0000 0000 0000 0000 0000 0000 0011
4	0011 0100	0000 0000 0000 0000 0000 0000 0000 0100
5	0011 0101	0000 0000 0000 0000 0000 0000 0000 0101
6	0011 0110	0000 0000 0000 0000 0000 0000 0000 0110
7	0011 0111	0000 0000 0000 0000 0000 0000 0000 0111
8	0011 1000	0000 0000 0000 0000 0000 0000 0000 1000
9	0011 1001	0000 0000 0000 0000 0000 0000 0000 1001
a	0110 0001	0000 0000 0000 0000 0000 0000 0000 1010
b	0110 0010	0000 0000 0000 0000 0000 0000 0000 1011
c	0110 0011	0000 0000 0000 0000 0000 0000 0000 1100
d	0110 0100	0000 0000 0000 0000 0000 0000 0000 1101
e	0110 0101	0000 0000 0000 0000 0000 0000 0000 1110
f	0110 0110	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.

Sign	BCD code (four bits)
+	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:

decimal	Gray code
0	0
1	1

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

Gray code
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:

decimal	Gray code
0	00
1	01
2	11
3	10

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

Gray code
00
01
11
10
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:

decimal	Gray code
0	000
1	001
2	011
3	010
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).

Decimal	Gray code
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`
b) `aa`
c) `f0`
d) `0f`
e) `80`
f) `63`
g) `7b`

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`
b) `edcc`
c) `fedc`
d) `07d0`
e) `8000`
f) `0400`
g) `ffff`
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
b) -1
c) -10
d) 88
e) 127
f) -16
g) -32
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
b) -1024
c) -1
d) 32767
e) -256
f) -32768
g) -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`
b) `55 + f0`
c) `80 + 7b`
d) `63 + 7b`
e) `0f + ff`
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`
b) `1234 + fedc`
c) `8000 + 8000`
d) `0400 + ffff`
e) `07d0 + 782f`
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 $\overline{x}$. 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*:

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z \tag{4.1}$$
$$x + (y + z) = (x + y) + z \tag{4.2}$$

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:

$$x \cdot 1 = x \tag{4.3}$$
$$x + 0 = x \tag{4.4}$$

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

- AND and OR are *commutative*:

$$x \cdot y = y \cdot x \tag{4.5}$$
$$x + y = y + x \tag{4.6}$$

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:

$$x \cdot 0 = 0 \tag{4.7}$$
$$x + 1 = 1 \tag{4.8}$$

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:

$$x \cdot x' = 0 \tag{4.9}$$
$$x + x' = 1 \tag{4.10}$$

  Complement does not exist in elementary algebra.

- AND and OR are *idempotent*:

$$x \cdot x = x \tag{4.11}$$
$$x + x = x \tag{4.12}$$

  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*:

$$x \cdot (y + z) = x \cdot y + x \cdot z \tag{4.13}$$
$$x + y \cdot z = (x + y) \cdot (x + z) \tag{4.14}$$

  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*:

$$(x')' = x \tag{4.15}$$

  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.

$$(x \cdot y)' = x' + y' \tag{4.16}$$
$$(x + y)' = x' \cdot y' \tag{4.17}$$

  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 "$\cdot$" 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 "$\cdot$" operator is omitted.

A Boolean expression is created from the numbers 0 and 1, and literals. Literals can be combined using either the "$\cdot$" 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,

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

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),

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

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

$$F_1(x,y,z) = x \cdot y \cdot z \tag{4.20}$$

$$F_2(x,y,z) = x + y + z \tag{4.21}$$

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:

			$minterm$	$maxterm$
x	y	z	$F_1 = (x \cdot y \cdot z)$	$F_2 = (x + y + z)$
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:

$$F_1(x, y) = x \cdot y' + x' \cdot y + x \cdot y \tag{4.22}$$

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.

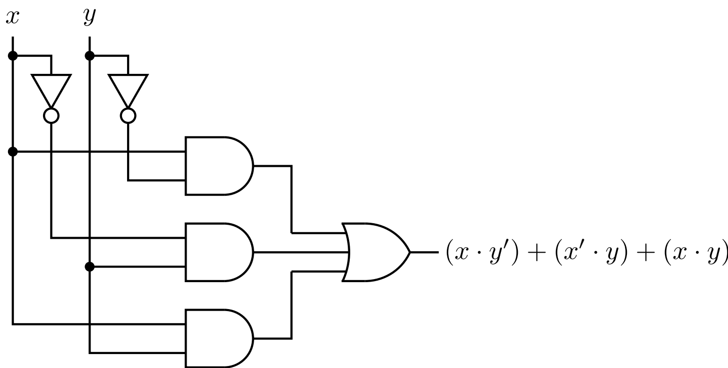


Figure 4.4: Hardware implementation of the function in Equation 4.22.

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 third term, and then rearrange a bit:

$$F_1(x, y) = x \cdot y' + x \cdot y + x' \cdot y + x \cdot y \tag{4.23}$$

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

$$F_1(x,y) = x \cdot (y' + y) + y \cdot (x' + x) \tag{4.24}$$

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

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

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:

$$F_2(x,y) = (x + y') \cdot (x' + y) \cdot (x' + y') \tag{4.26}$$

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.

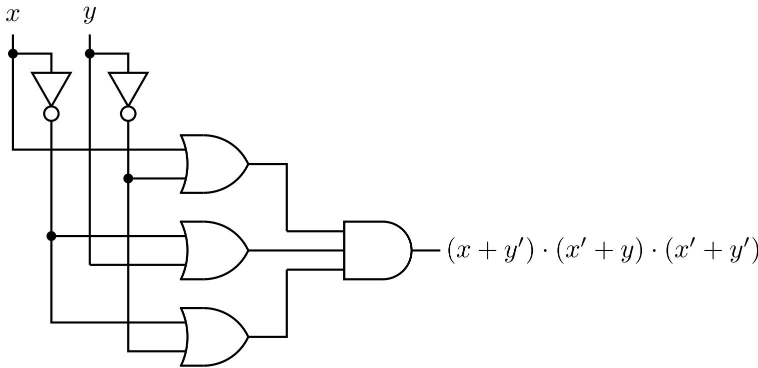


Figure 4.5: Hardware implementation of the function in Equation 4.26.

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

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

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

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

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:

$$F(w,x,y,z) = w' \cdot x' \cdot y' \cdot z' + w' \cdot x' \cdot y \cdot z' + w' \cdot x \cdot y' \cdot z' + w' \cdot x \cdot y \cdot z' \\ + w \cdot x' \cdot y' \cdot z' + w \cdot x' \cdot y \cdot z' + w \cdot x \cdot y' \cdot z' + w \cdot x \cdot y \cdot z'$$

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

$F(x,y)$	$y=0$	$y=1$
$x=0$	m_0	m_1
$x=1$	m_2	m_3

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 = 0	y = 1
x = 0		1
x = 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 = 0	y = 1
x = 0		1
x = 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:

$$F_1(x, y) = x + y \tag{4.31}$$

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

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.

$F(x,y,z)$ $x \backslash yz$	00	01	11	10
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.

$F(w,x,y,z)$ $wx \backslash yz$	00	01	11	10
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

$$F(x, y, z) = x' \cdot y' \cdot z' + x' \cdot y' \cdot z + x' \cdot y \cdot z' + x \cdot y' \cdot z' + x \cdot y \cdot z' + x \cdot y \cdot z$$

Solution:
First we draw the Karnaugh map:

$F(x,y,z)$	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

$F(x,y,z)$	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).

$$F(x,y,z) = x' \cdot y' \cdot z' + x' \cdot y' \cdot z + x' \cdot y \cdot z' + x \cdot y' \cdot z' + x \cdot y \cdot z' + x \cdot y \cdot z$$

Solution:
Using the Karnaugh map zeros,

$F(x,y,z)$ $x \backslash yz$	00	01	11	10
0			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

$$F(w,x,y,z) = w' \cdot x' \cdot y' \cdot z' + w' \cdot x' \cdot y \cdot z' + w' \cdot x \cdot y' \cdot z + w' \cdot x \cdot y \cdot z + w \cdot x \cdot y' \cdot z + w \cdot x \cdot y \cdot z + w \cdot x' \cdot y' \cdot z' + w \cdot x' \cdot y \cdot z'$$

Solution:
Using the groupings on the Karnaugh map,

$F(w,x,y,z)$ $wx \backslash yz$	00	01	11	10
00	1			1
01		1	1	
11		1	1	
10	1			1

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,

$$F_1(x, y, z) = (x' \cdot y' \cdot z') \tag{4.32}$$

$$F_2(x, y, z) = (x + y + z) \tag{4.33}$$

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.

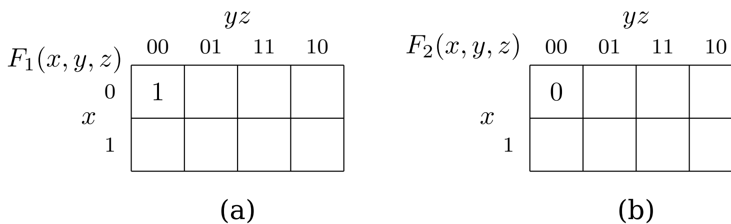


Figure 4.12: Comparison of one minterm (a) versus one maxterm (b) on a Karnaugh map.

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 is in each cell corresponding to a maxterm.

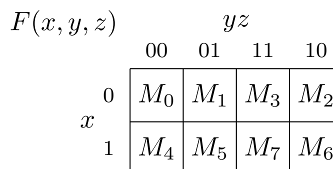


Figure 4.13: Mapping of three-variable maxterms on a Karnaugh map.

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.

$F(w,x,y,z)$

wx \ yz	00	01	11	10
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 4.14: Mapping of four-variable minterms on a Karnaugh map.

Example 4-e

Find a minimal product of sums for the function

$$F(x,y,z) = (x+y+z)\cdot(x+y+z')\cdot(x+y'+z') \cdot (x'+y+z)\cdot(x'+y'+z')$$

Solution:
This expression includes maxterms 0, 1, 3, 4, and 7. These appear in a Karnaugh map:

$F(x,y,z)$

x \ yz	00	01	11	10
0	0	0	0	
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.

$F(x,y,z)$

x \ yz	00	01	11	10
0	0	0	0	
1	0		0	

From this Karnaugh map it is very easy to write the function as a minimal product of sums:

$$F(x,y,z) = (x+y)\cdot(y+z)\cdot(y'+z')$$

□

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:

$$x \oplus y = x\cdot y' + x'\cdot y$$

This is the simplest form of the XOR operation. It requires two AND gates, two NOT gates, and an OR gate for realization.

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.

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.

$F(x,y)$	$y=0$	$y=1$
$x=0$		1
$x=1$	1	$\times$

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

$F(x,y)$	$y=0$	$y=1$
$x=0$		1
$x=1$	1	$\times$

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.

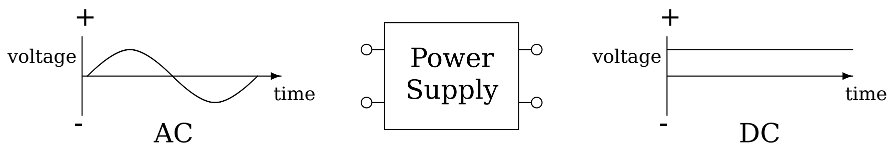


Figure 4.18: AC/DC power supply.

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

$$v = i\ R \tag{4.34}$$

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 Ω is used to

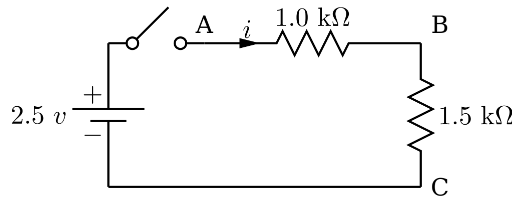


Figure 4.19: Two resistors in series.

indicate ohms, and $\mathrm{k}\Omega$ 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:

$$\begin{aligned} R &= 1.0\ \mathrm{k}\Omega + 1.5\ \mathrm{k}\Omega \\ &= 2.5 \times 10^3 \mathrm{ohms} \end{aligned} \tag{4.35}$$

The amount of current can be determined from the application of Equation 4.34. Solving for i,

$$\begin{aligned} i &= \frac{v}{R} \\ &= \frac{2.5 \text{ volts}}{2.5 \times 10^3 \text{ ohms}} \\ &= 1.0 \times 10^{-3} \text{ amps} \\ &= 1.0 \text{ ma} \end{aligned} \tag{4.36}$$

where “ma” means “milliamps.”

We can now use Equation 4.34 to determine the voltage difference between points A and B.

$$\begin{aligned} v_{AB} &= i\ R \\ &= 1.0 \times 10^{-3} \text{ amps} \times 1.0 \times 10^3 \text{ ohms} \\ &= 1.0 \text{ volts} \end{aligned} \tag{4.37}$$

Similarly, the voltage difference between points B and C is

$$\begin{aligned} v_{BC} &= i\ R \\ &= 1.0 \times 10^{-3} \text{ amps} \times 1.5 \times 10^3 \text{ ohms} \\ &= 1.5 \text{ volts} \end{aligned} \tag{4.38}$$

Figure 4.20 shows the same two resistors connected in parallel. In this case, the

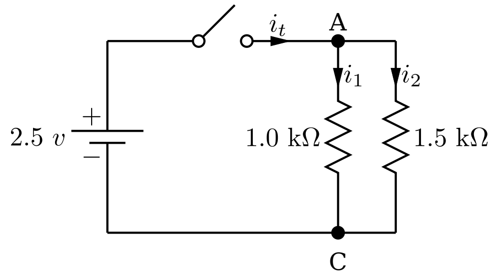


Figure 4.20: Two resistors in parallel.

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,

$$\begin{aligned} i_1 &= \frac{v}{R_1} \\ &= \frac{2.5 \text{ volts}}{1.0 \times 10^3 \text{ ohms}} \\ &= 2.5 \times 10^{-3} \text{ amps} \\ &= 2.5 \text{ ma} \end{aligned} \tag{4.39}$$

and

$$\begin{aligned} i_2 &= \frac{v}{R_2} \\ &= \frac{2.5 \text{ volts}}{1.5 \times 10^3 \text{ ohms}} \\ &= 1.67 \times 10^{-3} \text{ amps} \\ &= 1.67 \text{ ma} \end{aligned} \tag{4.40}$$

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,

$$\begin{aligned} i_t &= i_1 + i_2 \\ &= 2.5 \text{ ma} + 1.67 \text{ ma} \\ &= 4.17 \text{ ma} \end{aligned} \tag{4.41}$$

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

$$v = \frac{1}{C} \int_0^t i \, \mathrm{d}t \tag{4.42}$$

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}.

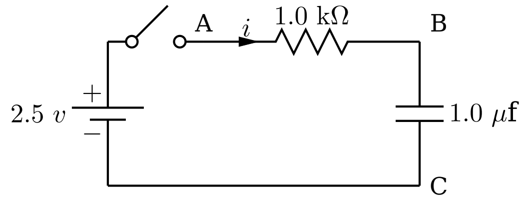


Figure 4.21: Capacitor in series with a resistor; v_{AB} is the voltage across the resistor and v_{BC} is the voltage across the capacitor.

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,

$$2.5 = i\ R + v_{BC} \tag{4.43}$$

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

$$\begin{aligned} i_{\text{initial}} &= \frac{2.5 \text{ volts}}{1.0 \text{ k}\Omega} \\ &= 2.5 \text{ ma} \end{aligned} \tag{4.44}$$

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

$$\begin{aligned} R\ C &= 1.0 \times 10^3 \text{ ohms} \times 1.0 \times 10^{-6} \text{ farads} \\ &= 1.0 \times 10^{-3} \text{ seconds} \\ &= 1.0 \text{ msec.} \end{aligned} \tag{4.45}$$

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

$$v_{BC} = 2.5\ (1 - \mathrm{e}^{-t/10^{-3}}) \tag{4.46}$$

This is shown in Figure 4.22. At the time $t = 1.0$ millisecond (one time constant), the voltage across the capacitor is

$$\begin{aligned} v_{BC} &= 2.5\ (1 - \mathrm{e}^{-10^{-3}/10^{-3}}) \\ &= 2.5\ (1 - \mathrm{e}^{-1}) \\ &= 2.5 \times 0.63 \\ &= 1.58 \text{ volts} \end{aligned} \tag{4.47}$$

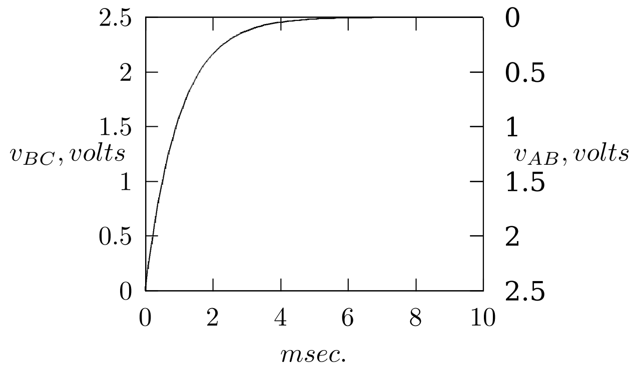


Figure 4.22: Capacitor charging over time in the circuit in Figure 4.21. The left-hand y-axis shows voltage across the capacitor, the right-hand voltage across the resistor.

After 6 time constants of time have passed, the voltage across the capacitor has reached

$$\begin{aligned} v_{BC} &= 2.5\ (1 - \mathrm{e}^{-6\times10^{-3}/10^{-3}}) \\ &= 2.5\ (1 - \mathrm{e}^{-6}) \\ &= 2.5 \times 0.9975 \\ &= 2.49 \text{ volts} \end{aligned} \tag{4.48}$$

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 (∼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

$$v = L\ \frac{\mathrm{d}i}{\mathrm{d}t} \tag{4.49}$$

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

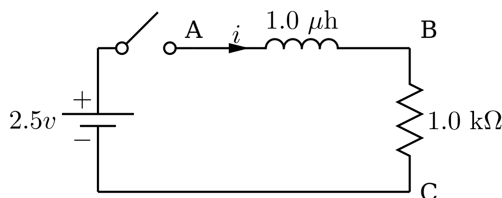


Figure 4.23: Inductor in series with a resistor.

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

$$\begin{aligned}\frac{L}{R} &= \frac{1.0 \times 10^{-6} \text{ henrys}}{1.0 \times 10^{3} \text{ ohms}} \\ &= 1.0 \times 10^{-9} \text{ seconds} \\ &= 1.0 \text{nanoseconds}\end{aligned} \tag{4.50}$$

When the switch is closed, the voltage that develops across the inductor over time is given by

$$v_{AB} = 2.5 \times \mathrm{e}^{-t/10^{-9}} \tag{4.51}$$

This is shown in Figure 4.24. Note that after about 6 nanoseconds (6 time constants) the

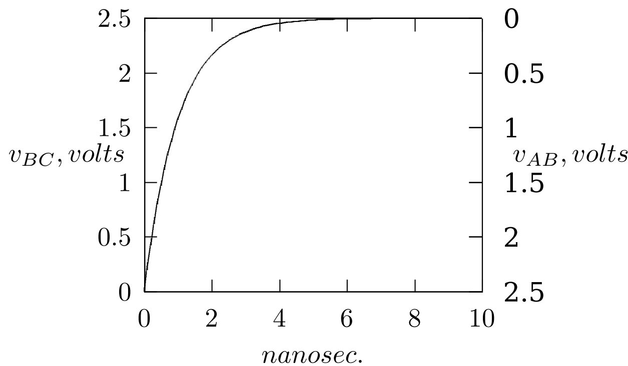


Figure 4.24: Inductor building a magnetic field over time in the circuit in Figure 4.23. The left-hand y-axis shows voltage across the resistor, the right-hand voltage across the inductor.

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

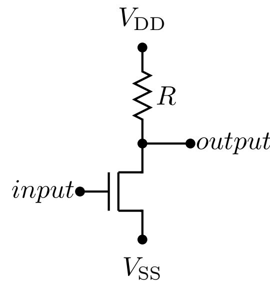


Figure 4.25: A single n-type MOSFET transistor switch.

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

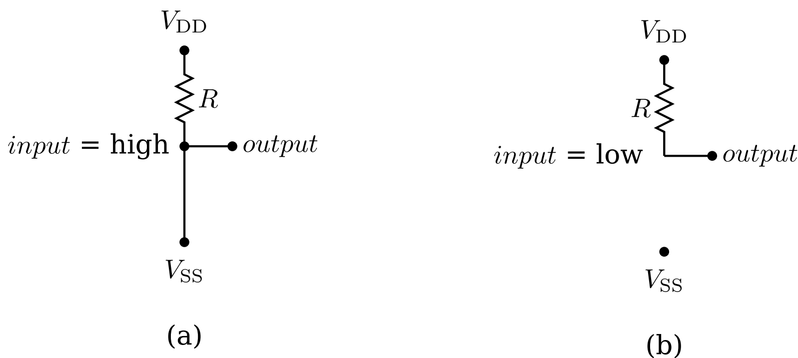


Figure 4.26: Single transistor switch equivalent circuit; (a) switch closed; (b) switch open.

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

$$i = \frac{V_{DD} - V_{SS}}{R} \tag{4.52}$$

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

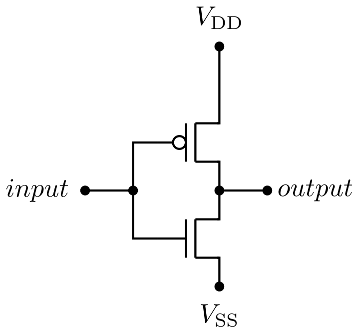
input	*output*
low	*high*
high	*low*

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 larger 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.



input	*output*
0	1
1	0

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.

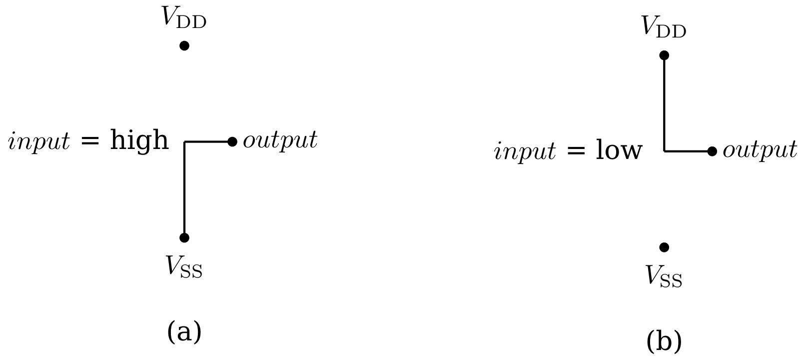
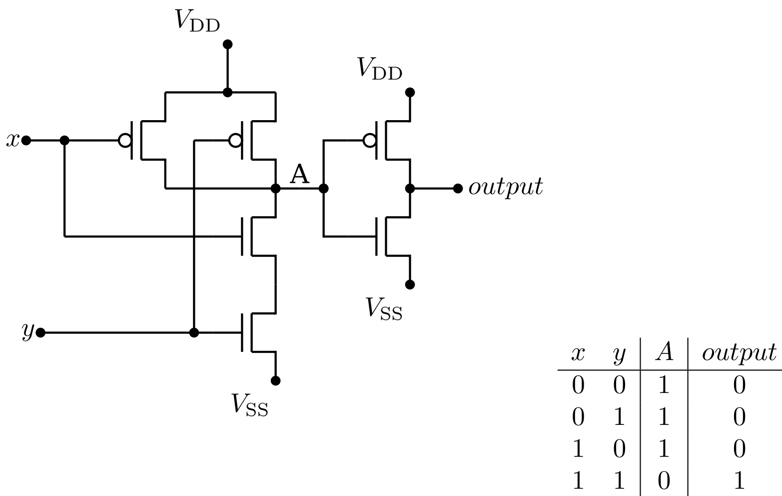


Figure 4.28: CMOS inverter equivalent circuit; (a) pull up open and pull down closed; (b) pull up closed and pull down open.

Figure 4.29 shows an AND gate implemented with CMOS transistors. (See Exercise



x	y	A	*output*
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 4.29: CMOS AND circuit.

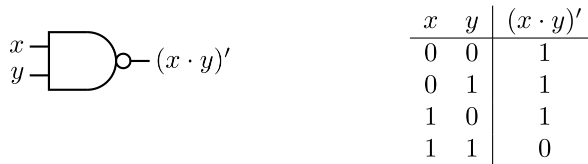
4-12.) Notice that the signal at point A is NOT(x AND y). The circuit from point A to the output is a NOT gate. It requires two fewer transistors than the AND operation. We will examine the implications of this result in Section 4.5.

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.

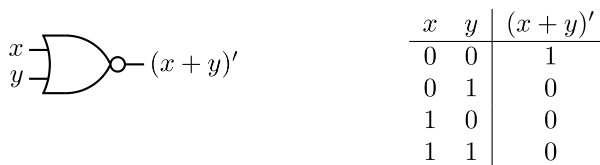
- NAND — a binary operator; the result is 0 if and only if both operands are 1; otherwise the result is 1. We will use $(x \cdot y)'$ to designate the NAND operation. It is also common to use the '↑' symbol or simply "NAND". The hardware symbol for the NAND gate is shown in Figure 4.30. The inputs are 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.

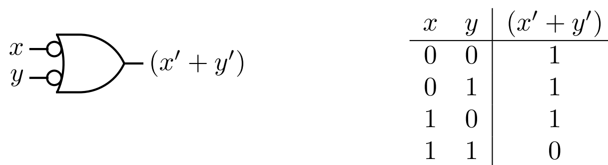
- NOR — a binary operator; the result is 0 if at least one of the two operands is 1; otherwise the result is 1. We will use $(x + y)'$ to designate the NOR operation. It is also common to use the '↓' symbol or simply "NOR". The hardware symbol for the NOR gate is shown in Figure 4.31. The inputs are 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.

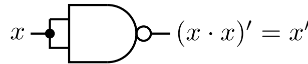


Figure 4.33: A NOT gate built from a NAND gate.

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.

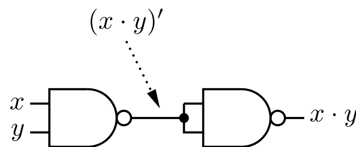


Figure 4.34: An AND gate built from two NAND gates.

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.

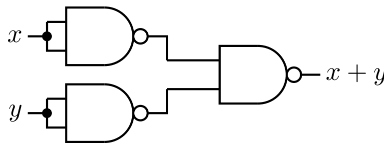


Figure 4.35: An OR gate built from three NAND gates.

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

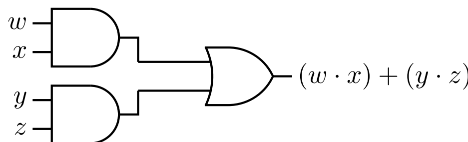


Figure 4.36: The function in Equation 4.55 using two AND gates and one OR gate.

(Equation 4.15) it is clear that the circuit in Figure 4.37 is equivalent to the one in Figure 4.36.

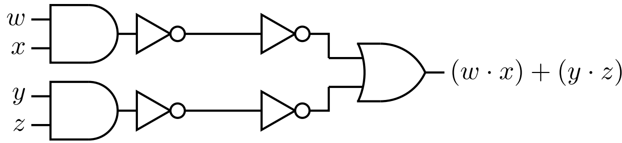


Figure 4.37: The function in Equation 4.55 using two AND gates, one OR gate and four NOT gates.

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.

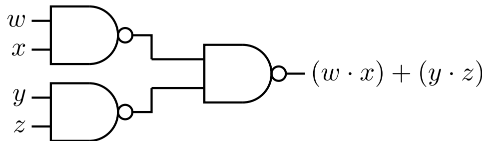


Figure 4.38: The function in Equation 4.55 using only three NAND gates.

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.

$F(x, y, z)$

z \ 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.

$F(x, y, z)$

y \ 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 ith bit of the multiple bit value, x; y_i is the ith bit of the multiple bit value, y; Sum_i is the ith 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 ith bit of the multiple bit value, x; y_i is the ith bit of the multiple bit value, y; Sum_i is the ith 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:

Sum_i	x_iy_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.

$$\begin{aligned} Sum_i(Carry_i, x_i, y_i) &= Carry_i' \cdot x_i' \cdot y_i + Carry_i' \cdot x_i \cdot y_i' \\ &+ \ Carry_i \cdot x_i' \cdot y_i' + Carry_i \cdot x_i \cdot y_i \end{aligned} \tag{5.1}$$

In the Karnaugh map for carry:

$Carry_{i+1}$	x_iy_i = 00	01	11	10
$Carry_i$ = 0			1	
1		1	1	1

we can see three groupings:

$Carry_{i+1}$	$x_i y_i$			
$Carry_i$	00	01	11	10
0			1	
1		1	1	1

These groupings yield a three-term function that defines when $Carry_{i+1} = 1$:

$$Carry_{i+1} = x_i \cdot y_i + Carry_i \cdot x_i' \cdot y_i + Carry_i \cdot x_i \cdot y_i' \tag{5.2}$$

Equations 5.1 and 5.2 lead directly to the circuit for an adder in Figure 5.1.

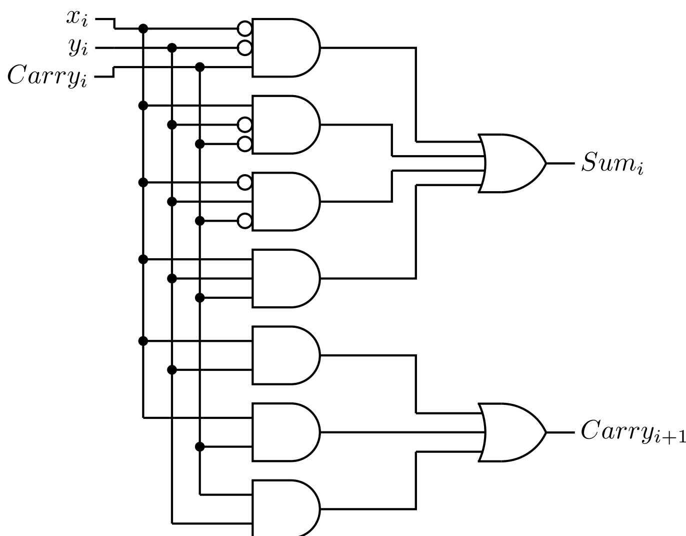


Figure 5.1: An adder circuit.

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.

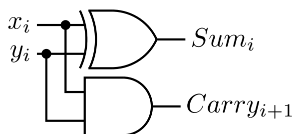


Figure 5.2: A half adder circuit.

Instead of using Karnaugh maps, we will perform some algebraic manipulations on Equation 5.1. Using the distribution rule, we can rearrange:

$$\begin{aligned} Sum_i(Carry_i, x_i, y_i) &= Carry_i' \cdot (x_i' \cdot y_i + x \cdot y_i') + Carry_i \cdot (x_i' \cdot y_i' + x_i \cdot y_i) \\ &= Carry_i' \cdot (x_i \oplus y_i) + Carry_i \cdot (x_i' \cdot y_i' + x_i \cdot y_i) \end{aligned} \tag{5.3}$$

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

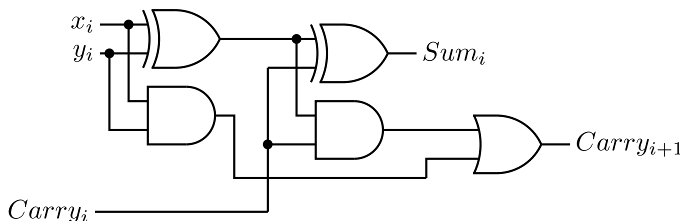


Figure 5.3: Full adder using two half adders.

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 ith 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

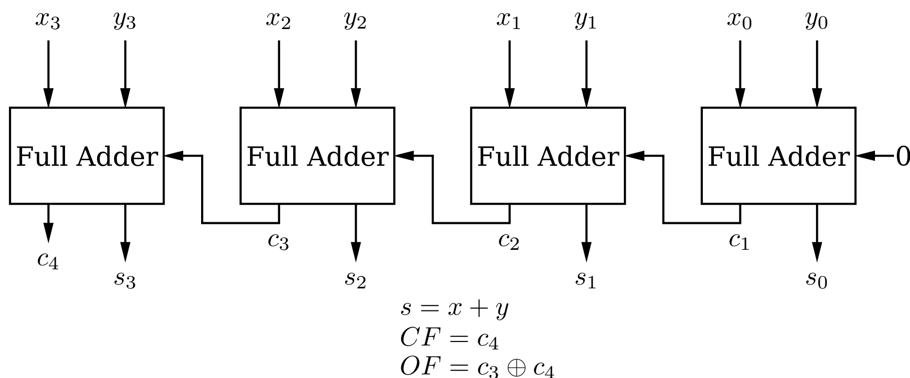


Figure 5.4: Four-bit adder.

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:

$$\begin{aligned} x - y &= x + (\text{2's complement of } y) \\ &= x + [(y\text{'s bits flipped}) + 1] \end{aligned} \tag{5.6}$$

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 0`*s*`0`*d* `1000 1001 11`*ss 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:

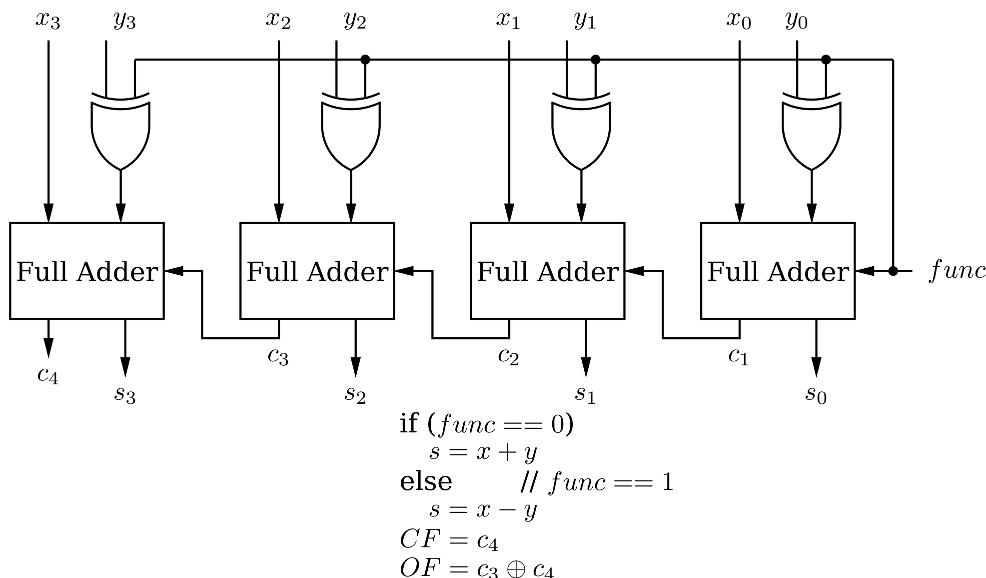


Figure 5.5: Four-bit adder/subtracter.

```
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	1	0	0	0	0
0	1	0	1	0	0	0	0	1	0	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 ith

$enable$	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
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 3×8 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,

$$\begin{aligned} y_3 &= x_2' \cdot x_1 \cdot x_0 \\ &= m_3 \end{aligned} \tag{5.7}$$

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 ith output bit is equal to the ith minterm for the n input bits.

The 3×8 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):

$$Sum_i(Carry_i, x_i, y_i) = m_1 + m_2 + m_4 + m_7 \tag{5.8}$$

And for the Carry expression:

$$Carry_{i+1}(Carry_i, x_i, y_i) = m_3 + m_5 + m_6 + m_7 \tag{5.9}$$

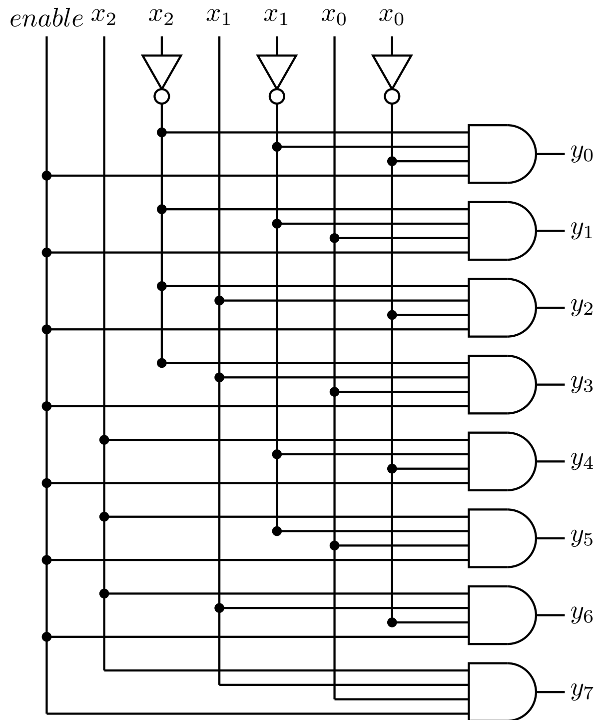


Figure 5.6: Circuit for a 3×8 decoder with enable.

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 3×8 decoder and two 4-input OR gates, as shown in Figure 5.7.

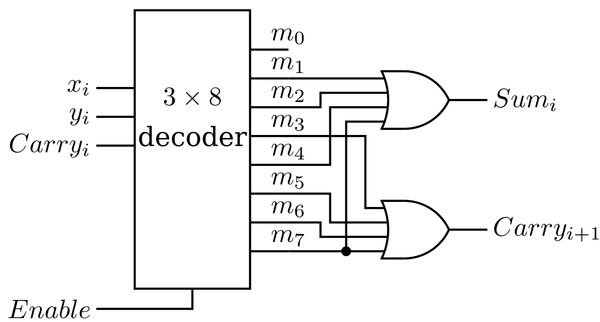


Figure 5.7: Full adder implemented with 3×8 decoder. This is for one bit slice. An n-bit adder would require n of these circuits.

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

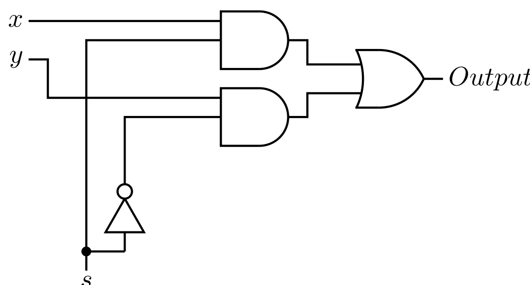


Figure 5.8: A 2-way multiplexer.

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,

$$Output = s_0' \cdot s_1' \cdot w + s_0' \cdot s_1 \cdot x + s_0 \cdot S_1' \cdot y + s_0 \cdot s_1 \cdot z \quad (5.10)$$

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.

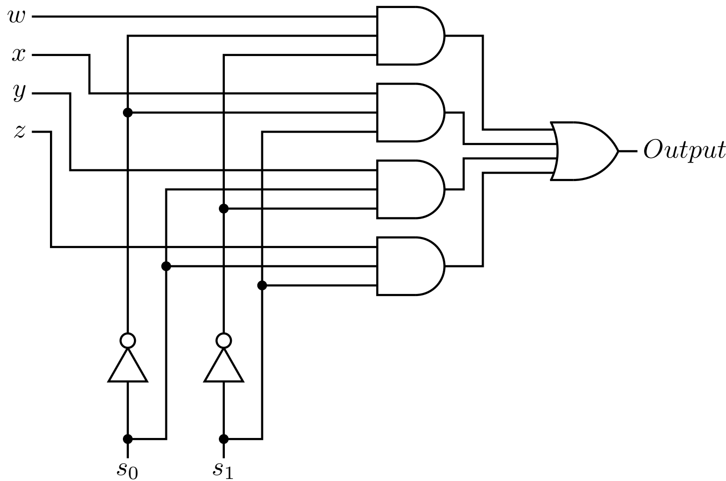


Figure 5.9: A 4-way multiplexer.

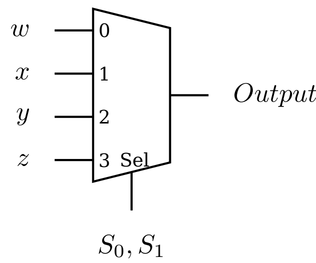


Figure 5.10: Symbol for a 4-way multiplexer.

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.

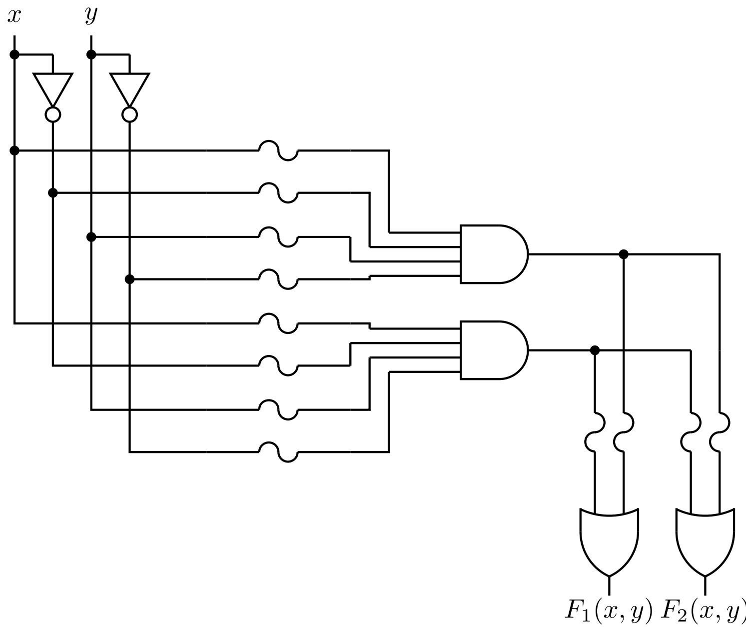


Figure 5.11: Simplified circuit for a programmable logic array. The “S” shaped line at the inputs to each gate represent fuses. The fuses are “blown” to remove that input.

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

$$w' \cdot y \cdot z$$

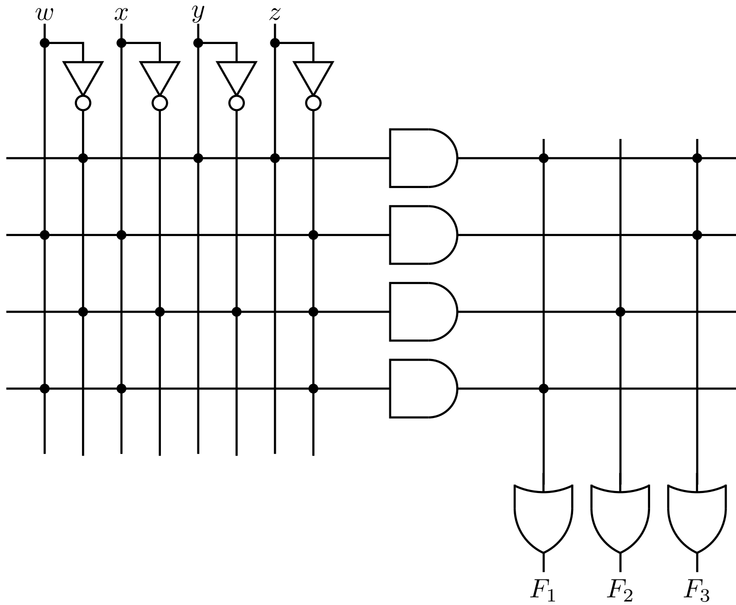


Figure 5.12: Programmable logic array schematic. The horizontal lines to the AND gate inputs represent multiple wires — one for each input variable and its complement. The vertical lines to the OR gate inputs also represent multiple wires — one for each AND gate output. The dots represent connections.

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:

$$F_1(w, x, y, z) = w' \cdot y \cdot z + w \cdot x \cdot z' \tag{5.11}$$
$$F_2(w, x, y, z) = w' \cdot x' \cdot y' \cdot z' \tag{5.12}$$
$$F_3(w, x, y, z) = w' \cdot y \cdot z + w \cdot x \cdot z' \tag{5.13}$$

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_1a_0'	10
a_1a_0	11

And the OR gate plane has been programmed to store the four characters (in ASCII code):

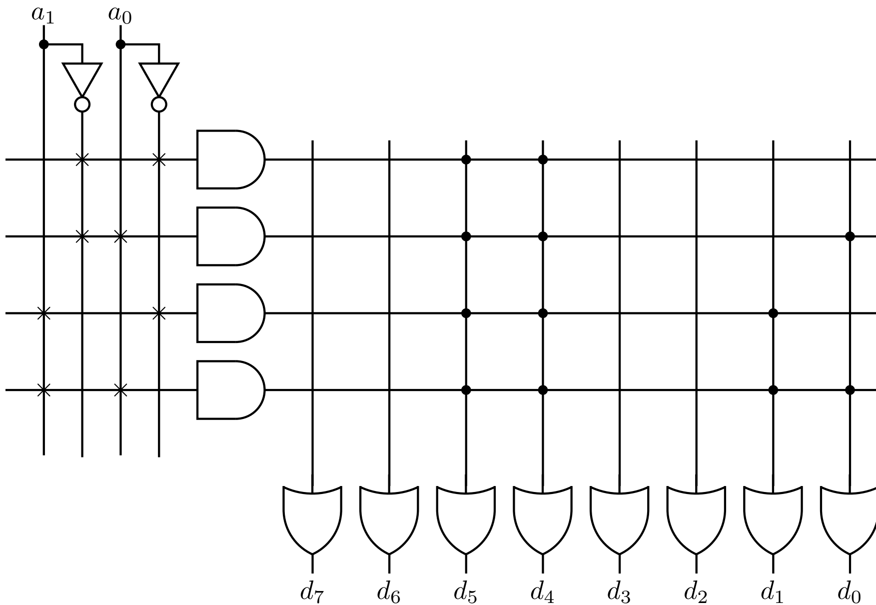


Figure 5.13: Eight-byte Read Only Memory (ROM). The "×" connections represent permanent connections. Each AND gate can be thought of as producing an address. The eight OR gates produce one byte. The connections (dots) in the OR plane represent the bit pattern stored at the address.

minterm	address	contents
$a_1'a_0'$	00	$'0'$
$a_1'a_0$	01	$'1'$
a_1a_0'	10	$'2'$
a_1a_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 "×" 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

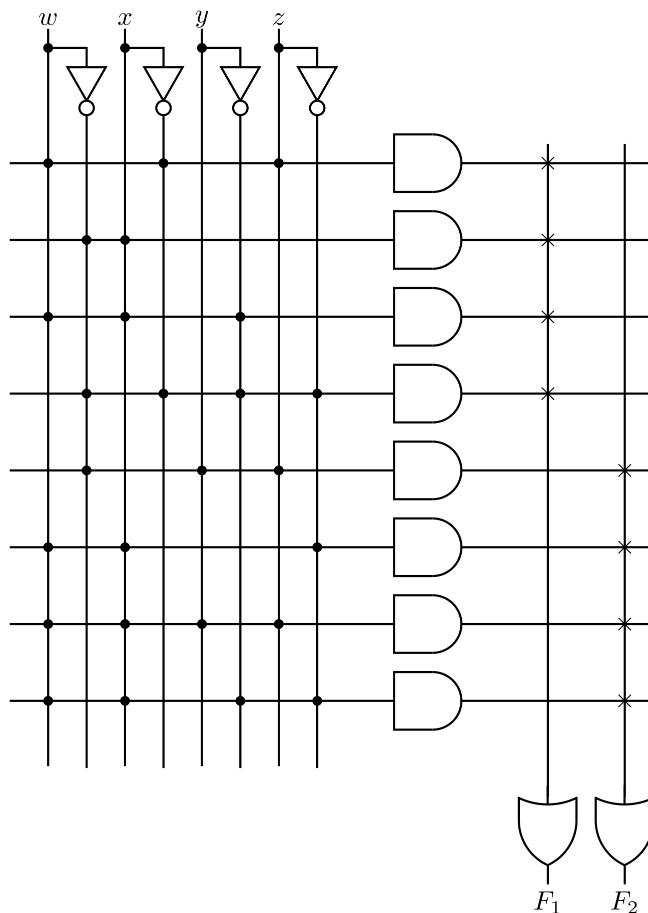


Figure 5.14: Two-function Programmable Array Logic (PAL). The "×" connections represent permanent connections. Each AND gate can be thought of as producing an address. The eight OR gates produce one byte. The connections (dots) in the OR plane represent the bit pattern stored at the address.

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.

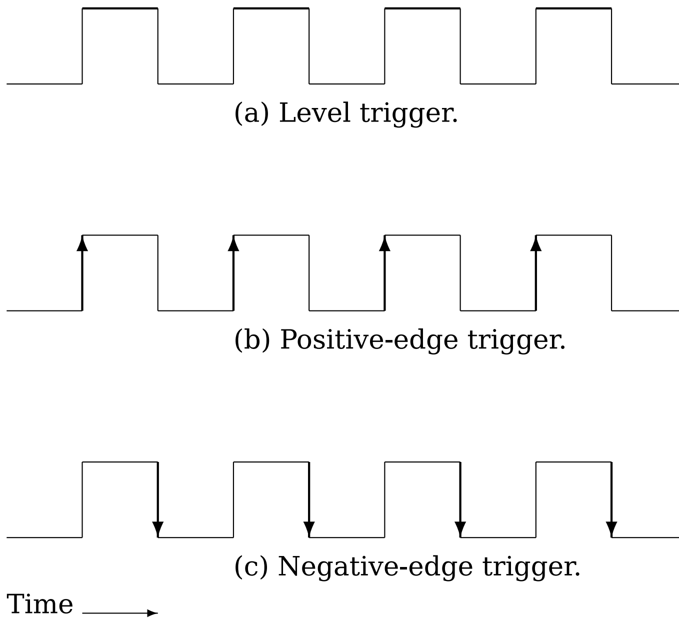


Figure 5.15: Clock signals. (a) For level-triggered circuits. (b) For positive-edge triggering. (c) For negative-edge triggering.

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 $Q = 1$

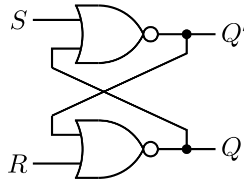


Figure 5.16: NOR gate implementation of an SR latch.

($\Leftrightarrow Q' = 0$) 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

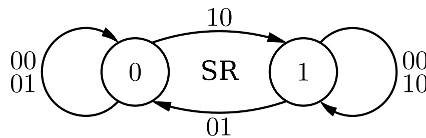


Figure 5.17: State diagram for an SR latch. There are two possible inputs, 00 or 01, that cause the latch to remain in state 0. Similarly, 00 or 10 cause it to remain in state 1. Since the output is simply the state, it is not shown in this state diagram. Notice that the input 11 is not allowed, so it 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 $(0 \cdot 1)' = 1$.

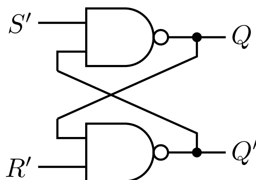


Figure 5.18: NAND gate implementation of an S’R’ latch.

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.

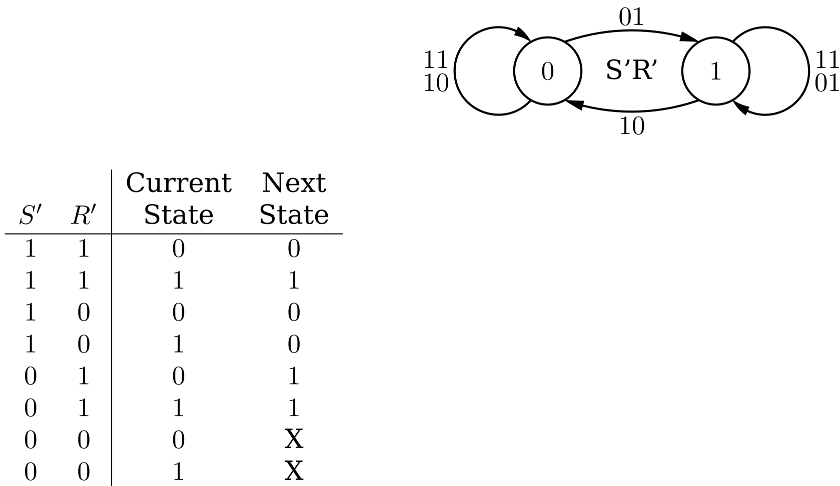
If $Q = 1$ and $Q' = 0$, the output of the lower NAND gate is $(1 \cdot 0)' = 1$. This is fed back to the input of the upper NAND gate to give $(0 \cdot 1)' = 1$ as its output. The feedback from the output of the upper NAND gate to the input of the lower maintains its output as $(1 \cdot 1)' = 0$. Thus, $Q = Q' = 1$, 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 $'$ 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

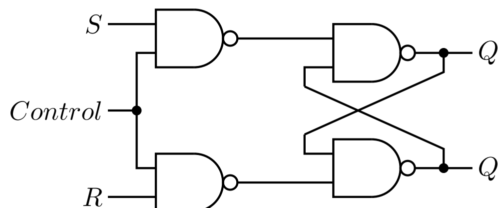


Figure 5.20: SR latch with $Control$ input.

NAND gates remain at 1 as long as $Control = 0$. Table 5.4 shows the state behavior of the SR latch with control.

Control	*S*	*R*	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.

Control	*D*	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

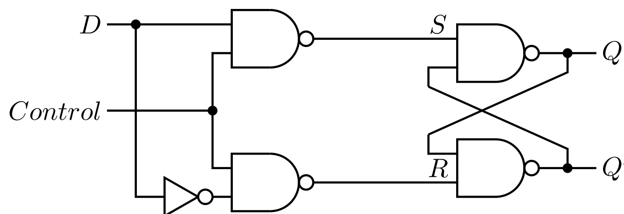


Figure 5.21: D latch constructed from an SR latch.

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.

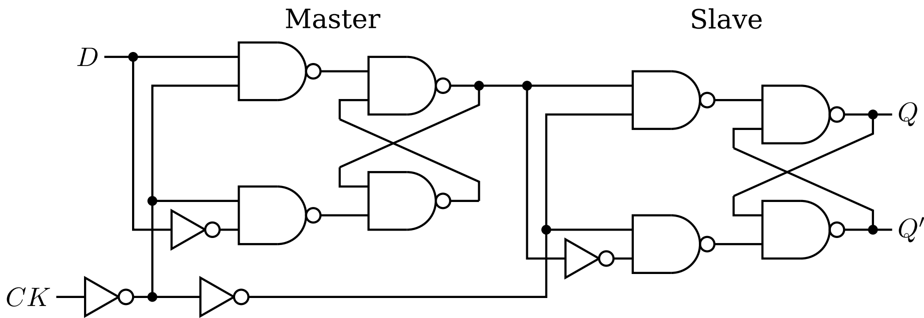


Figure 5.22: D flip-flop, positive-edge triggering.

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

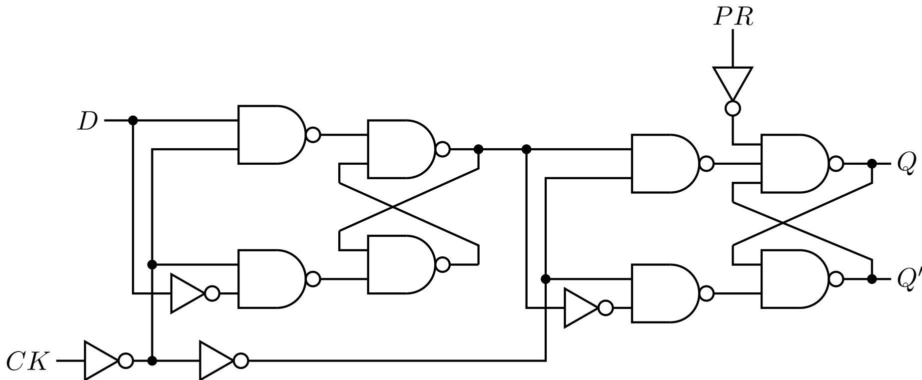


Figure 5.23: D flip-flop, positive-edge triggering with asynchronous preset.

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.

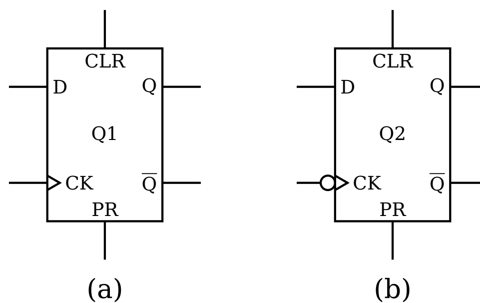
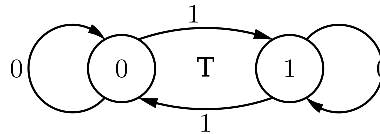


Figure 5.24: Symbols for D flip-flops. Includes asynchronous clear (CLR) and preset (PR). (a) Positive-edge triggering; (b) Negative-edge triggering.

Hardware designers typically use $\overline{Q}$ instead of Q'. It is common to label the circuit as "Q*n*," with *n* = 1, 2,... 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.

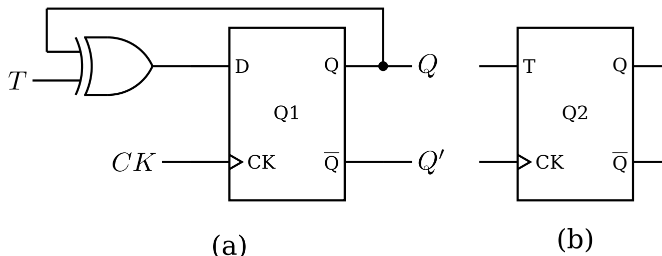
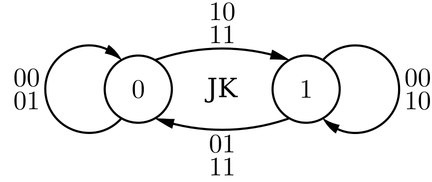


Figure 5.26: T flip-flop. (a) Circuit using a D flip-flop. (b) Symbol for a T flip-flop.

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.

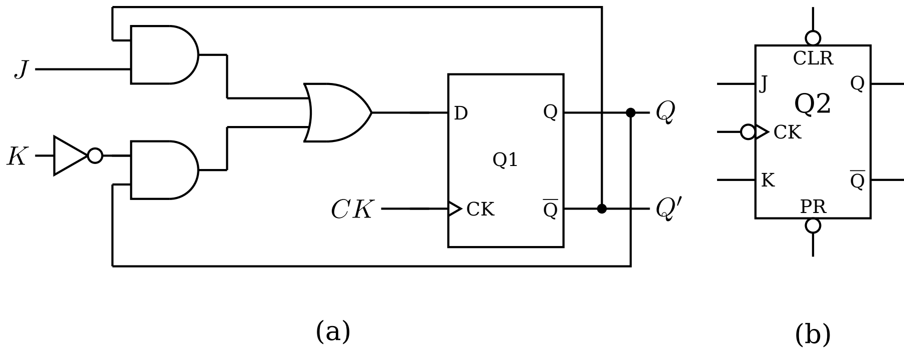


Figure 5.28: JK flip-flop. (a) Circuit using a D flip-flop. (b) Symbol for a JK flip-flop with asynchronous CLR and PR inputs.

5.4 Designing Sequential Circuits

We will now consider a more general set of steps for designing sequential circuits.[1] 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 $\log_2$ 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

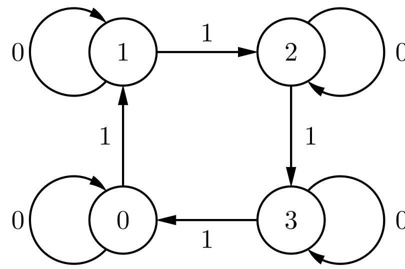
Design a counter that has an *Enable* input. When $Enable = 1$ it increments through the sequence 0, 1, 2, 3, 0, 1,... with each clock tick. $Enable = 0$ causes the counter to remain in its current state.

Solution:

1. First we create a state table and state diagram:

[1] I wish to thank Dr. Lynn Stauffer for her valuable suggestions for this section.

	$Enable = 0$	$Enable = 1$
Current	Next	Next
n	n	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 $n = n_1 n_0$, giving the state table:

		$Enable = 0$		$Enable = 1$	
Current		Next		Next	
n_1	n_0	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.

		$Enable = 0$						$Enable = 1$					
Current		Next						Next					
n_1	n_0	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 $Enable$.

$J_0(E, n_1, n_0)$

E \ $n_1 n_0$	00	01	11	10
0		X	X	
1	1	X	X	1

$K_0(E, n_1, n_0)$

E \ $n_1 n_0$	00	01	11	10
0	X			X
1	X	1	1	X

$J_1(E, n_1, n_0)$

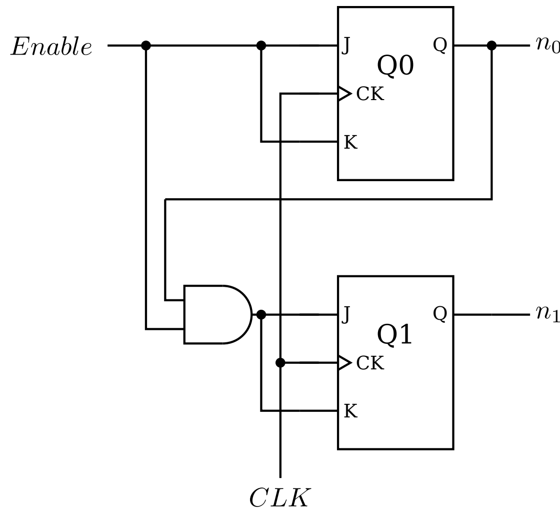
E \ $n_1 n_0$	00	01	11	10
0			X	X
1		1	X	X

$K_1(E, n_1, n_0)$

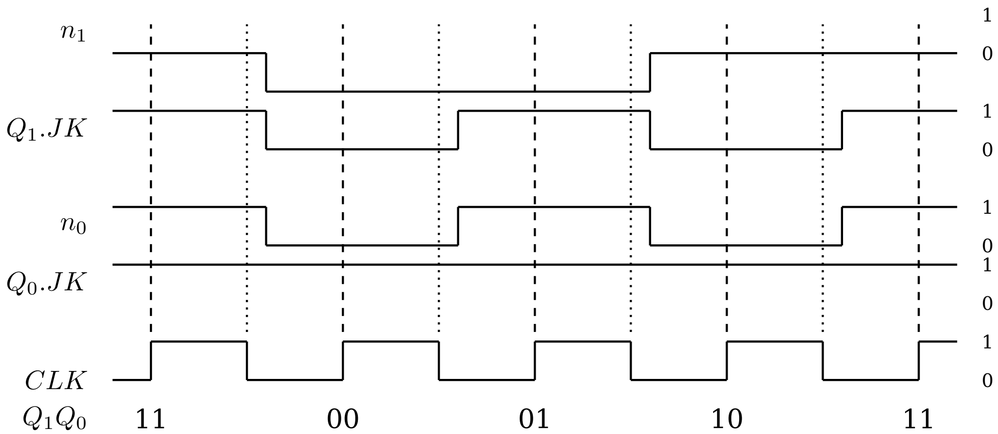
E \ $n_1 n_0$	00	01	11	10
0	X	X		
1	X	X	1	

$$J_0(E, n_1, n_0) = E$$
$$K_0(E, n_1, n_0) = E$$
$$J_1(E, n_1, n_0) = E \cdot n_0$$
$$K_1(E, n_1, n_0) = E \cdot n_0$$

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 ith JK flip-flop, and n_i is its output. (Recall that $J = K$ in this design.) When the ith 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.

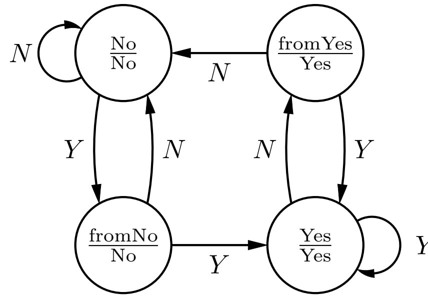
In this next example we will design a circuit to implement a prediction circuit.

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		$Taken = No$ Next		$Taken = Yes$ Next	
State	Prediction	State	Prediction	State	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_1s_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:

Current		*Input = 0* *Next*		*Input = 1* *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.

Current		*Input = 0* *Next*						*Input = 1* *Next*					
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	1	0	0	0	X	X	1	1	1	1	X	X	0
1	0	0	0	X	1	0	X	1	1	X	0	1	X
1	1	1	0	X	0	X	1	1	1	X	0	X	0

5. We use Karnaugh maps to derive equations for the JK flip-flop inputs.

$J_0(In, s_1, s_0)$

In \ s_1s_0	00	01	11	10
0		X	X	
1	1	X	X	1

$K_0(In, s_1, s_0)$

In \ s_1s_0	00	01	11	10
0	X	1	1	X
1	X			X

$J_1(In, s_1, s_0)$

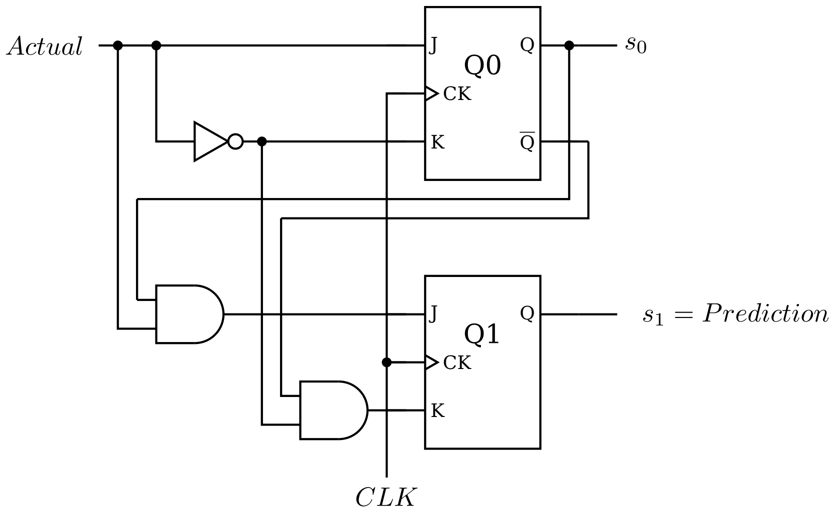
In \ s_1s_0	00	01	11	10
0			X	X
1		1	X	X

$K_1(In, s_1, s_0)$

In \ s_1s_0	00	01	11	10
0	X	X		1
1	X	X		

$$J_0(In, s_1, s_0) = In$$
$$K_0(In, s_1, s_0) = In'$$
$$J_1(In, s_1, s_0) = In \cdot s_0$$
$$K_1(In, s_1, s_0) = In' \cdot s_0'$$

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_is 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.

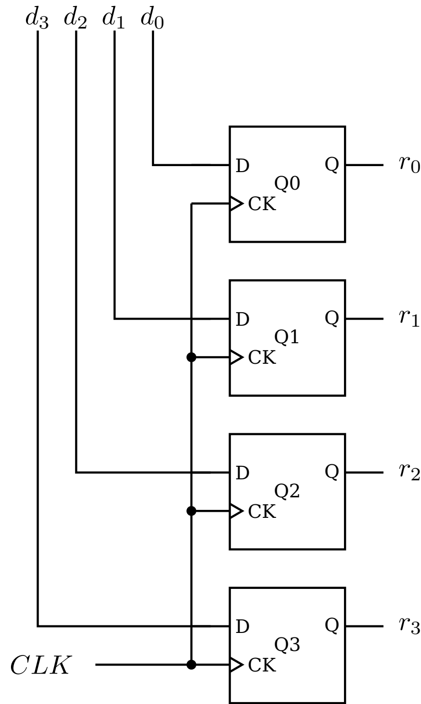


Figure 5.29: A 4-bit register. A D flip-flop is used to hold each bit. The state of the ith bit is set by the value of d_i at each clock tick. The 4-bit value stored in the register is $r = r_3r_2r_1r_0$.

Three bits are required to address one of the registers ($2^3 = 8$). If the 8-bit output from a 3×8 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 $r0_i$ – $r7_i$ are the ith bits from each of eight registers, $r0$ – $r7$. 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.

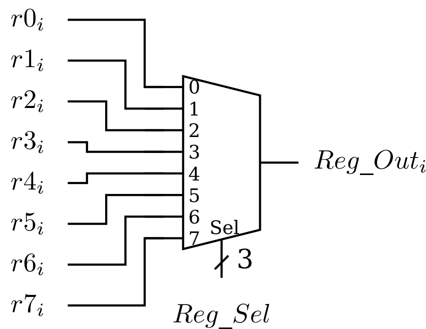


Figure 5.31: 8-way mux to select output of register file. This only shows the output of the ith bit. n are required for n-bit registers. Reg_Sel is a 3-bit signal that selects one of the eight inputs.

- 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

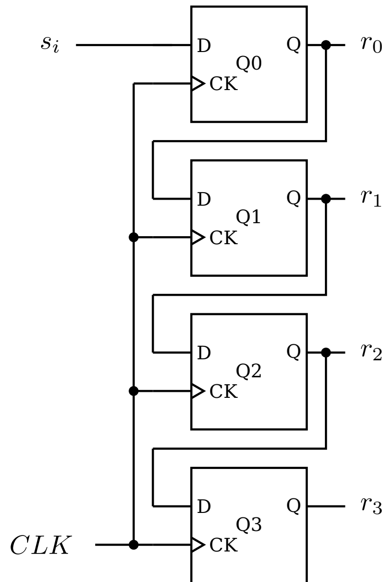


Figure 5.32: Four-bit serial-to-parallel shift register. A D flip-flop is used to hold each bit. Bits arrive at the input, s_i, one at a time. The last four input bits are available in parallel at r_3 – r_0.

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, $r_3, \ldots, r_0$.

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

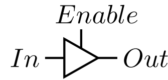


Figure 5.33: Tri-state buffer.

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 2×4 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

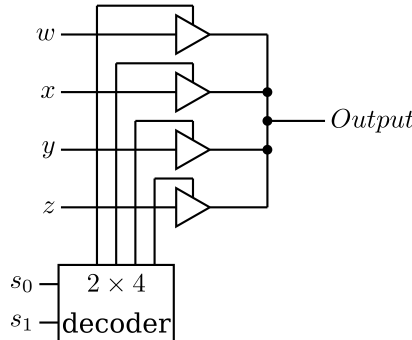


Figure 5.34: Four way multiplexer built from tri-state buffers. $Output$ = w, x, y, or z, depending on which one is selected by $s_1 s_0$ fed into the decoder. Compare with Figure 5.9, page 100.

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, $r_3 r_2 r_1 r_0$.

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 20×2^{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

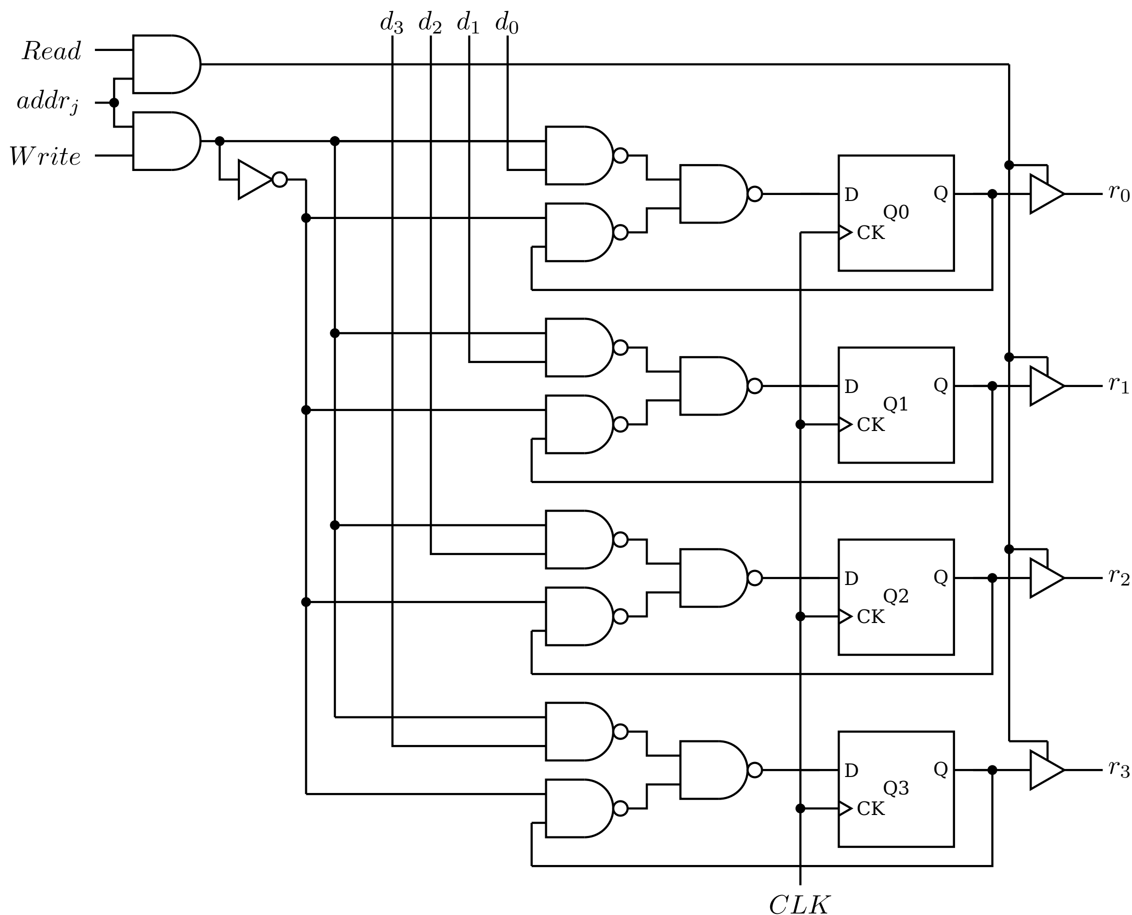


Figure 5.35: 4-bit memory cell. Each bit is output through a tri-state buffer. $addr_j$ is one output from a decoder corresponding to an address.

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 a 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.

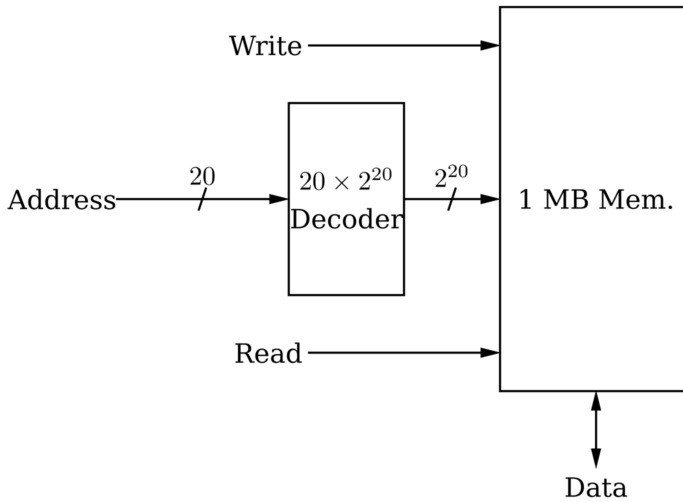


Figure 5.36: Addressing 1 MB of memory with one 20×2^{20} address decoder. The short line through the connector lines indicates the number of bits traveling in parallel in that connection.

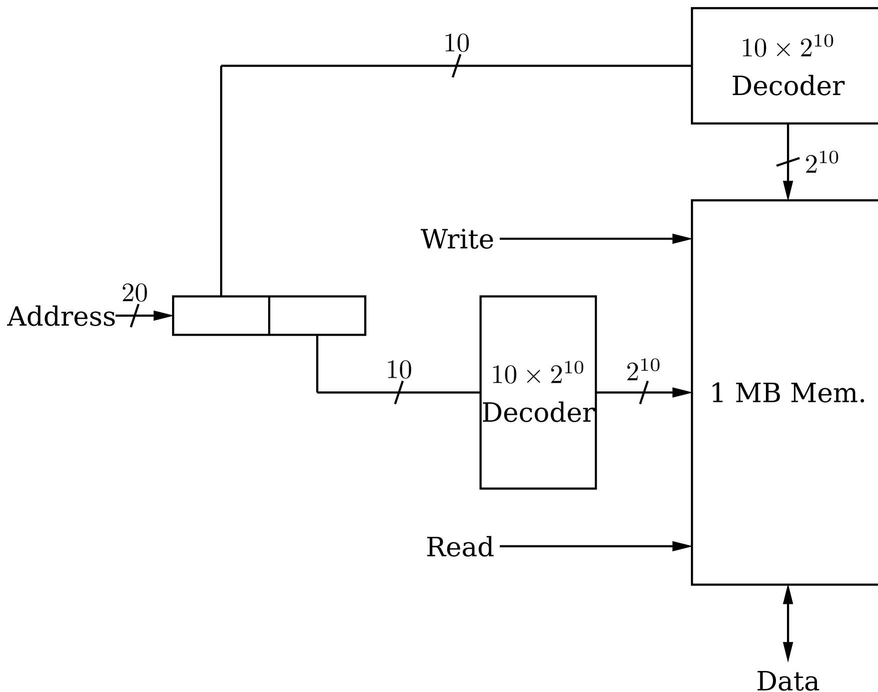


Figure 5.37: Addressing 1 MB of memory with two 10×2^{10} address decoders.

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

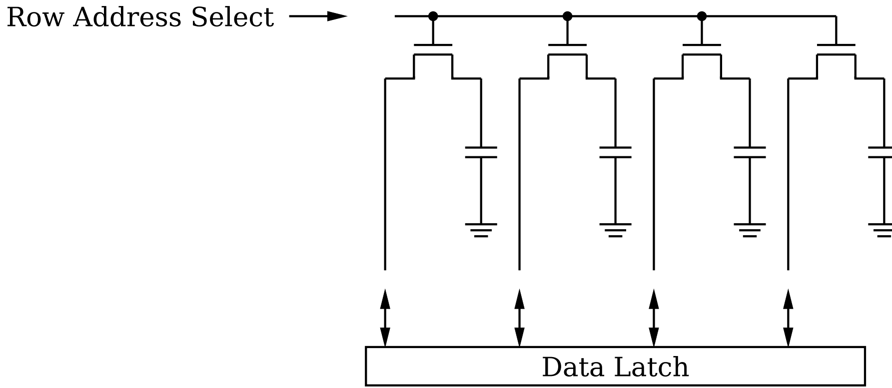


Figure 5.38: Bit storage in DRAM.

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/subtracter.

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 3×8 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.

<table>
<tr><th>Mode</th><th>Submode</th><th>Operating System</th><th>Default Address (bits)</th><th>Default <code>int</code> (bits)</th></tr>
<tr><td rowspan="3">IA-32e or Long</td><td>64-bit</td><td rowspan="3">64-bit</td><td>64</td><td rowspan="2">32</td></tr>
<tr><td rowspan="2">Compatibility</td><td>32</td></tr>
<tr><td>16</td><td>16</td></tr>
<tr><td rowspan="3">Legacy</td><td>Protected</td><td rowspan="2">32-bit</td><td>32</td><td>32</td></tr>
<tr><td>Virtual-8086</td><td rowspan="2">16</td><td rowspan="2">16</td></tr>
<tr><td>Real</td><td>16-bit</td></tr>
</table>

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*.

6.1 CPU Overview

An overall block diagram of a typical CPU is shown in Figure 6.1. The subsystems are

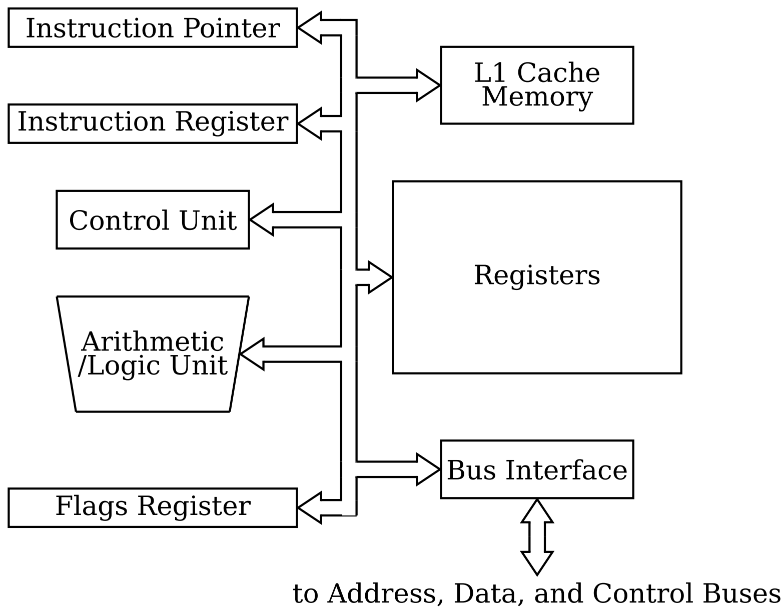


Figure 6.1: CPU block diagram. The CPU communicates with the Memory and I/O subsystems via the Address, Data, and Control buses. See Figure 1.1 (page 3).

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 (**GPR**s)
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 (**MSR**s)		
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.

A pictorial representation of the naming of each portion of the general-purpose registers is shown in Figure 6.2.

bits 63-0	**bits 31-0**	**bits 15-0**	**bits 15-8**	**bits 7-0**
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	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.

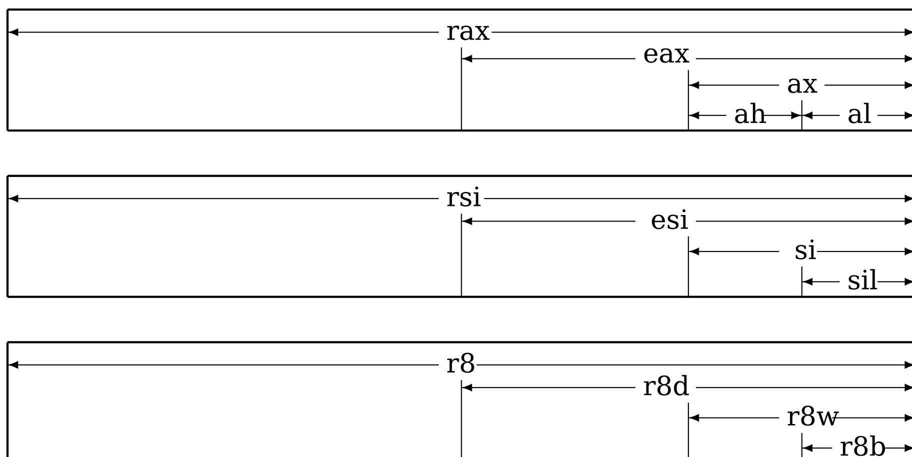


Figure 6.2: Graphical representation of general purpose registers. The three shown here are representative of the pattern of all the general purpose registers.

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 *x*`h` 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

Register	Special usage	Called function preserves contents
`rax`	1*st* function return value.	No
`rbx`	Optional base pointer.	Yes
`rcx`	Pass 4*th* argument to function.	No
`rdx`	Pass 3*rd* argument to function; 2*nd* function return value.	No
`rsp`	Stack pointer.	Yes
`rbp`	Optional frame pointer.	Yes
`rdi`	Pass 1*st* argument to function.	No
`rsi`	Pass 2*nd* argument to function.	No
`r8`	Pass 5*th* argument to function.	No
`r9`	Pass 6*th* 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`[1], always points to the next instruction to be executed. As explained in Section 6.4 (page 136), every time an instruction is fetched,

[1] 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.

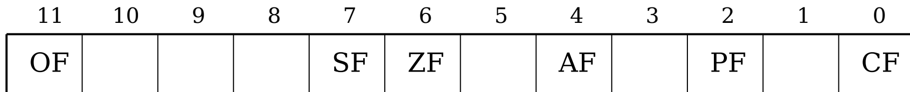


Figure 6.3: Condition codes portion of the `rflags` register. The high-order 32 bits (32 – 63) are reserved for other use and are not shown here. Neither are bits 12 – 31, which are for system flags (see [3]).

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.

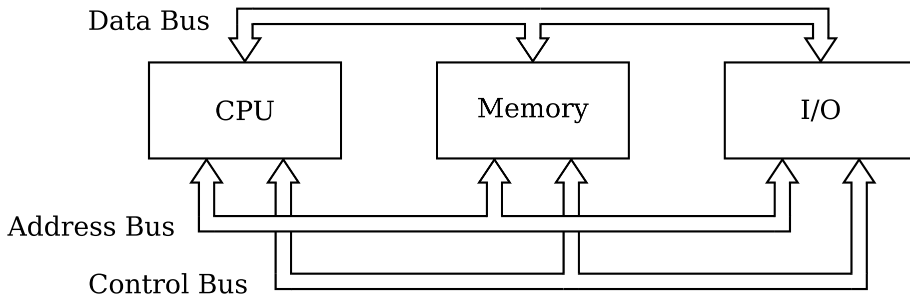


Figure 6.4: Subsystems of a computer. The CPU, Memory, and I/O subsystems communicate with one another via the three busses. (Repeat of Figure 1.1.)

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 `0x7fffd9a43cef` are in the state:

```
0x7fffd9a43cef: 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 `0x7fffd9a43cef`, the control unit then

1. places `0x7fffd9a43cef` 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 `0x7fffd9a43cef` will be changed to the state:

```
0x7fffd9a43cef: 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.

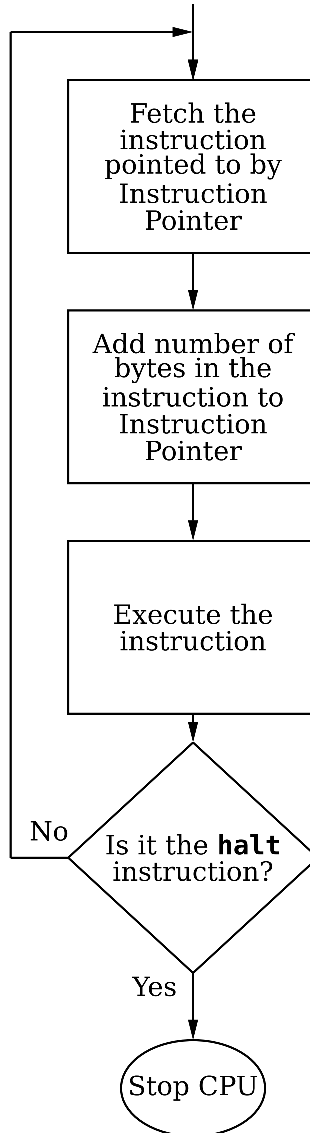


Figure 6.5: The instruction execution cycle.

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. [2]

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 that 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

[2] 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.

```c
/*
 * gdbExample1.c
 * Subtracts one from user integer.
 * Demonstrate use of gdb to examine registers, etc.
 * Bob Plantz - 5 June 2009
 */

#include <stdio.h>

int main(void)
{
    register int wye;
    int *ptr;
    int ex;

    ptr = &ex;
    ex = 305441741;
    wye = -1;
    printf("Enter an integer: ");
    scanf("%i", ptr);
    wye += *ptr;
    printf("The result is %i\n", wye);

    return 0;
}
```

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...d
(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 *) 0x7ffffffe044
```

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 0x7ffffffe044
 0x7ffffffe044: 305441741
```

I verify that the value assigned to the `ex` variable is stored at location `0x7ffffffe044`.

```
(gdb) x/1xw 0x7ffffffe044
0x7fff504c473c:     0x1234abcd
```

I examine the same integer in hexadecimal format.

```
 (gdb) x/4xb  0x7ffffffe044
0x7ffffffe044:     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 `0x7ffffffe044`,*
- *`0xab` is stored in the byte at address `0x7ffffffe045`,*
- *`0x34` is stored in the byte at address `0x7ffffffe046`, and*
- *`0x12` is stored in the byte at address `0x7ffffffe047`.*

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 0x7ffffffe044
0x7ffffffe044:     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 0x7fffffffe044, and*
- *0x1234 is stored in the two bytes starting at address 0x7fffffffe046.*

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 *)  0x7fffffffe044
(gdb) print &ptr
$4 = (int **) 0x7fffffffe048
```

Look carefully at the ptr variable. It is located at address 0x7fffffffe048 and it contains another address, 0x7fffffffe044, 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            0x7fffffffe044 140737488347204
rbx            0xffffffff 4294967295
rcx            0x4005e0 4195808
rdx            0x7fffffffe158 140737488347480
rsi            0x7fffffffe148 140737488347464
rdi            0x1 1
rbp            0x7fffffffe060 0x7fffffffe060
rsp            0x7fffffffe040 0x7fffffffe040
r8             0x400670 4195952
r9             0x7ffff7de9740 140737351948096
r10            0x7fffffffdec0 140737488346816
r11            0x7ffff7a3e680 140737348101760
r12            0x400480 4195456
r13            0x7fffffffe140 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`.[3] (Recall that `int`s 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.
```

[3] 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.

a) What is the address of the first instruction that will be executed when you enter the `n` command?

b) 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 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          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.

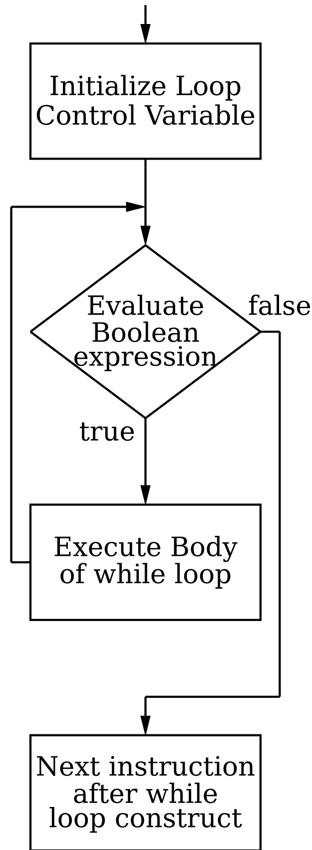


Figure 10.1: Flow chart of a `while` loop. The large diamond represents a binary decision that leads to two possible paths, "true" or "false." Notice the path that leads back to the top of the `while` loop after the body has been executed.

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

`cmp`*s* *source, destination*

where *s* denotes the size of the operand:

s	**meaning**	**number of bits**
`b`	byte	8
`w`	word	16
`l`	longword	32
`q`	quadword	64

Intel® Syntax: `cmp` *destination, source*

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

`test`*s* *source*, *destination*

where *s* denotes the size of the operand:

s	**meaning**	**number of bits**
b	byte	8
w	word	16
l	longword	32
q	quadword	64

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

`j`*cc* *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.

instruction	action	condition codes
`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	$ZF = 1$
`jg`	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	$SF = 1$
`jz`	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.

instruction	meaning	immediately after a `cmp` ...
`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.

instruction	meaning	immediately after a `cmp` ...
`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*. [1]

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 `j`*`cc`* 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

[1] 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    $0xfffffffe, %esi # no jump, turn off bit 0
21 here1:
22         leaq    here2, %rax
23         jmp     *%rax
24         andl    $0xfffffffd, %esi # no jump, turn off bit 1
25 here2:
26         leaq    here3, %rax
27         movq    %rax, pointer
28         jmp     *pointer
29         andl    $0xfffffffb, %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:

`movs`*sd* *source*, *destination*

where *s* denotes the size of the source operand and *d* the size of the destination operand.

(Use the *s* column for *d*.)

s	**meaning**	**number of bits**
`b`	byte	8
`w`	word	16
`l`	longword	32
`q`	quadword	64

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:

`movz`*sd* *source*, *destination*

where *s* denotes the size of the source operand and *d* the size of the destination operand.

(Use the *s* column for *d*.)

s	**meaning**	**number of bits**
`b`	byte	8
`w`	word	16
`l`	longword	32
`q`	quadword	64

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, shown in Table 10.5. The doubling operation includes sign extention into the affected higher-order portion of the register.

AT&T syntax	Intel® syntax	start	result
cbtw	cbw	byte in al	word in ax
cwtl	cwde	word in ax	long in eax
cltq	cdqe	lonq in eax	quad in rax

Table 10.5: Instructions to double data size. These instructions do not specify any operands, but they may change the rax register.

Notice that these instructions do not explicitly specify any operands, but they may change the rax register. 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    # loop body processing
19    # ------
20         subl    $1, count(%rbp)  # change loop control variable
21         jmp     whileLoop        # back to top
22 whileDone:
23    # 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          .equu   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 instructions 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,

`inc`*s* *source*

where *s* denotes the size of the operand:

s	meaning	number of bits
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 35 of the program in Listing 10.5, `incl` is used to add one to the address stored in memory:

```
35          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.

`dec`*s* *source*

where *s* denotes the size of the operand:

s	meaning	number of bits
b	byte	8
w	word	16
l	longword	32
q	quadword	64

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.

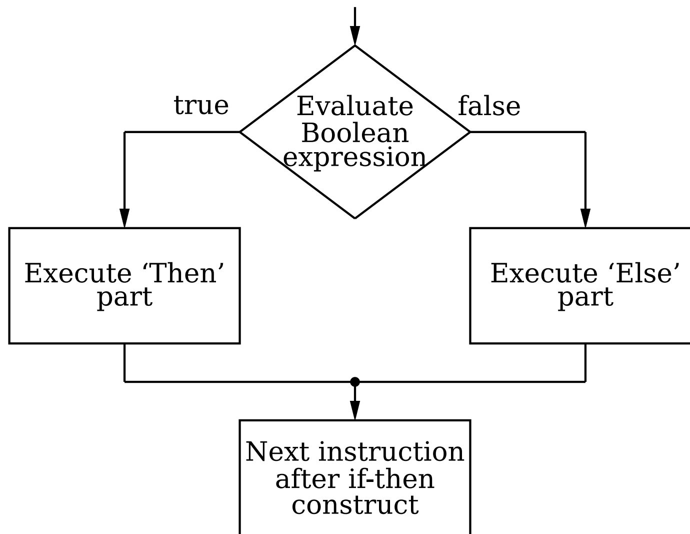


Figure 10.2: Flow chart of `if-else` construct. The large diamond represents a binary decision that leads to two possible paths, "true" or "false." Notice that either the "then" block or the "else" block is executed, but not both. Each leads to the end of the `if-else` construct.

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          .text
10         .globl  main
11         .type   main, @function
12 main:
13         pushq   %rbp
14         movq    %rsp, %rbp
15         subq    $16, %rsp
16         movq    $.LC0, -8(%rbp)
17         jmp     .L2
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:                             # else part,
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
68 .L7:                             # after if-else statement
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.

```
# generalIf-else.s
# general structure of an if-else (not a program)
#
#    if (response == 'y')
#    {
#        then part
#    }
#    else
#    {
#        else part
#    }
#
# Bob Plantz - 10 June 2009

        cmpb    $'y', response(%rbp) # check conditions
        jne     noChange        # false, go to else part
   # ------
   # "then" part processing
   # ------
        jmp     allDone         # go to end of if-else
noChange:
   # ------
   # "else" part processing
   # ------
allDone:
   # 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.

```
# yesNo2.s
# Prompts user to enter a y/n response.
# Bob Plantz - 12 June 2009

# Useful constants
        .equ    STDIN,0
        .equ    STDOUT,1
# Stack frame
        .equ    response,-1
        .equ    ptr,-16
        .equ    localSize,-16
# Read only data
        .section  .rodata
queryMsg:
        .string "Save changes? "
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:                             # "else" part
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
72  .L7:                                # end of if-else construct
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" ($57_{10} = \texttt{39}_{16}$). 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 that the ascii code for the numeral "0" ($48_{10} = \texttt{30}_{16}$). 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 of the boolean tests is executed one at a time from left to right. 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, cmov*cc*, for simple if constructs like this. The general format is

cmov*cc*   *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 `cmov`*cc* 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:

opcode	***source***	***destination***	***action***	**page**
`cbtw`			convert byte to word, `al` → `ax`	246
`cwtl`			convert word to long, `ax` → `eax`	246
`cltq`			convert long to quad, `eax` → `rax`	246
`cmov`*cc*	*%reg/mem*	*%reg*	conditional move	260
`mov`*s*	$*imm*/%*reg*	*%reg/mem*	move	156
`mov`*s*	*mem*	*%reg*	move	156
`movs`*ss*	$*imm*/%*reg*	*%reg/mem*	move, sign extend	245
`movz`*ss*	$*imm*/%*reg*	*%reg/mem*	move, zero extend	245
`pop`*w*		*%reg/mem*	pop from stack	181
`push`*w*	$*imm*/%*reg/mem*		push onto stack	181

s = `b`, `w`, `l`, `q`; *w* = `l`, `q`; *cc* = condition codes

arithmetic/logic:

opcode	***source***	***destination***	***action***	**page**
`add`*s*	$*imm*/%*reg*	*%reg/mem*	add	214
`add`*s*	*mem*	*%reg*	add	214
`cmp`*s*	$*imm*/%*reg*	*%reg/mem*	compare	237
`cmp`*s*	*mem*	*%reg*	compare	237
`dec`*s*	*%reg/mem*		decrement	249
`inc`*s*	*%reg/mem*		increment	248
`lea`*w*	*mem*	*%reg*	load effective address	191
`sub`*s*	$*imm*/%*reg*	*%reg/mem*	subtract	215
`sub`*s*	*mem*	*%reg*	subtract	215
`test`*s*	$*imm*/%*reg*	*%reg/mem*	test bits	238
`test`*s*	*mem*	*%reg*	test bits	238

s = `b`, `w`, `l`, `q`; *w* = `l`, `q`

program flow control:

opcode	***location***	***action***	**page**
`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
`j`*cc*	*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
You entered:  5
You entered:  5
You entered:  5
You entered:  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. They 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.

```
/*
 * addProg.c
 * Adds two integers
 * Bob Plantz - 13 June 2009
 */

#include <stdio.h>
#include "sumInts1.h"

int main(void)
{
    int x, y, z;
    int overflow;

    printf("Enter two integers: ");
    scanf("%i %i", &x, &y);
    overflow = sumInts(x, y, &z);
    printf("%i + %i = %i\n", x, y, z);
    if (overflow)
        printf("*** Overflow occurred ***\n");

    return 0;
}
```

```
/*
 * sumInts1.h
 * Returns N + (N-1) + ... + 1
 * Bob Plantz - 4 June 2008
 */

#ifndef SUMINTS1_H
#define SUMINTS1_H
int sumInts(int, int, int *);
#endif
```

```
/*
 * sumInts1.c
 * Adds two integers and outputs their sum.
 * Returns 0 if no overflow, else returns 1.
 * Bob Plantz - 13 June 2009
 */

```

```
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.

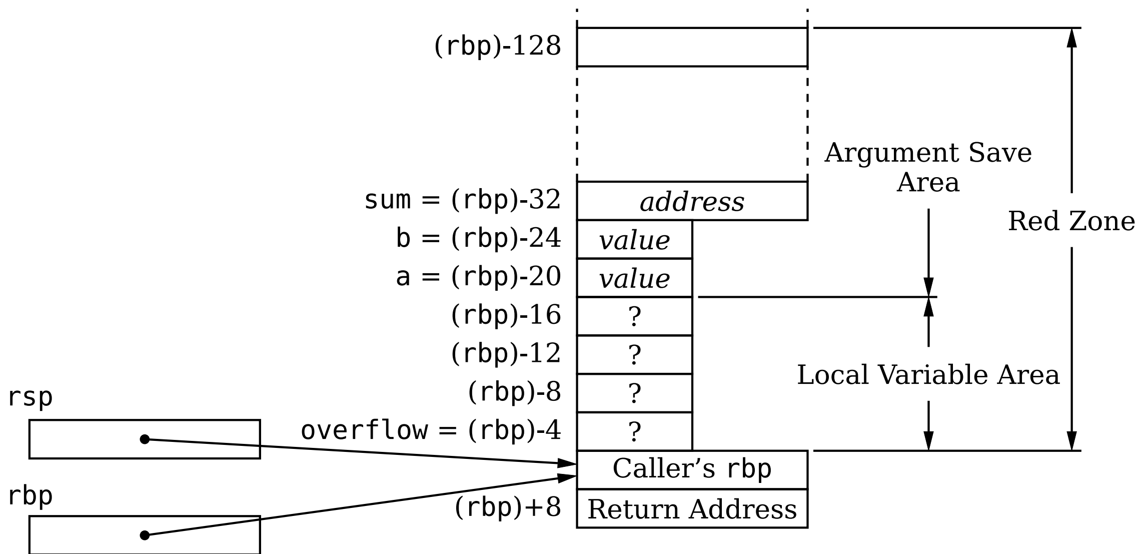


Figure 11.1: Arguments and local variables in the stack frame, `sumInts` function. The two input values and the address for the output are passed in registers, then stored in the Argument Save Area by the called function. Since this is a leaf function, the Red Zone is used for this function's stack frame.

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.

```
/*
 * nineInts1.c
 * Declares and adds nine integers.
 * Bob Plantz - 13 June 2009
 */
#include <stdio.h>
#include "sumNine1.h"

int main(void)
{
    int total;
    int a = 1;
    int b = 2;
    int c = 3;
    int d = 4;
    int e = 5;
    int f = 6;
    int g = 7;
    int h = 8;
    int i = 9;

    total = sumNine(a, b, c, d, e, f, g, h, i);
    printf("The sum is %i\n", total);
    return 0;
}
```

```
/*
 * sumNine1.h
 * Computes sum of nine integers.
 * Bob Plantz - 13 June 2009
 */
#ifndef SUMNINE_H
#define SUMNINE_H
int sumNine(int one, int two, int three, int four, int five,
        int six, int seven, int eight, int nine);
#endif
```

```
/*
 * sumNine1.c
 * Computes sum of nine integers.
 * Bob Plantz - 13 June 2009
 */
```

```c
#include <stdio.h>
#include "sumNine1.h"

int sumNine(int one, int two, int three, int four, int five,
            int six, int seven, int eight, int nine)
{
    int x;

    x = one + two + three + four + five + six
            + seven + eight + nine;
    printf("sumNine done.\n");
    return x;
}
```

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.

```asm
        .file   "nineInts1.c"
        .section        .rodata
.LC0:
        .string "The sum is %i\n"
        .text
        .globl  main
        .type   main, @function
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $80, %rsp
        movl    $1, -40(%rbp)
        movl    $2, -36(%rbp)
        movl    $3, -32(%rbp)
        movl    $4, -28(%rbp)
        movl    $5, -24(%rbp)
        movl    $6, -20(%rbp)
        movl    $7, -16(%rbp)
        movl    $8, -12(%rbp)
        movl    $9, -8(%rbp)
        movl    -20(%rbp), %r9d   # f is 6th argument
        movl    -24(%rbp), %r8d   # e is 5th argument
        movl    -28(%rbp), %ecx   # d is 4th argument
        movl    -32(%rbp), %edx   # c is 3rd argument
        movl    -36(%rbp), %esi   # b is 2nd argument
        movl    -40(%rbp), %eax   # load a
        movl    -8(%rbp), %edi    # load i
        movl    %edi, 16(%rsp)    # insert on stack
        movl    -12(%rbp), %edi   # load h
        movl    %edi, 8(%rsp)     # insert on stack
        movl    -16(%rbp), %edi   # load g
        movl    %edi, (%rsp)      # insert on stack
        movl    %eax, %edi        # a is 1st argument
        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

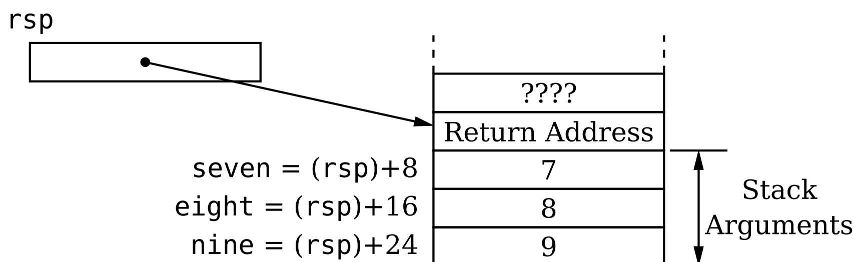


Figure 11.2: Arguments 7 – 9 are passed on the stack to the `sumNine` function. State of the stack when control is first transfered to this function.

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.

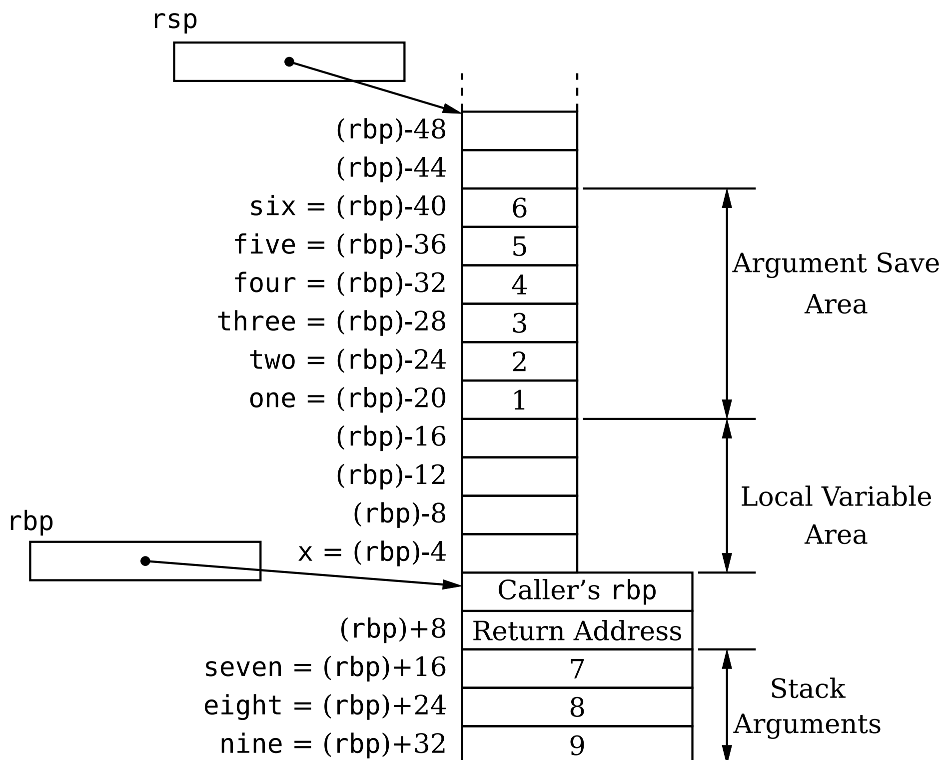


Figure 11.3: Arguments and local variables in the stack frame, sumNine function. The first six arguments are passed in registers but saved in the stack frame. Arguments beyond six are passed in the portion of the stack frame that is created by the calling function.

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`:

```
        movl    $.LC0, %edi
        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:

```
        movl    -24(%rbp), %eax  # load two
        movl    -20(%rbp), %edx  # load one, subtotal
        addl    %eax, %edx       # add two to subtotal
        movl    -28(%rbp), %eax  # load three
        addl    %eax, %edx       # add to subtotal
        movl    -32(%rbp), %eax  # load four
        addl    %eax, %edx       # add to subtotal
        movl    -36(%rbp), %eax  # load five
        addl    %eax, %edx       # add to subtotal
        movl    -40(%rbp), %eax  # load six
        addl    %eax, %edx       # add to subtotal
        movl    16(%rbp), %eax   # load seven
        addl    %eax, %edx       # add to subtotal
        movl    24(%rbp), %eax   # load eight
        addl    %eax, %edx       # add to subtotal
        movl    32(%rbp), %eax   # load nine
        addl    %edx, %eax       # add to subtotal
        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.

```
# nineInts2.s
# Demonstrate how integral arguments are passed in 64-bit mode.
# Bob Plantz - 13 June 2009
# Bob Plantz - 5 November 2013 - deleted unneeded register usage (lines 48 -

# Stack frame
#   passing arguments on stack (rsp)
#     need 3x8 = 24 -> 32 bytes
        .equ    seventh,0
```

```
10         .equ    eighth,8
11         .equ    ninth,16
12 #   local vars (rbp)
13 #     need 10x4 = 40 -> 48 bytes
14         .equ    i,-4
15         .equ    h,-8
16         .equ    g,-12
17         .equ    f,-16
18         .equ    e,-20
19         .equ    d,-24
20         .equ    c,-28
21         .equ    b,-32
22         .equ    a,-36
23         .equ    total,-40
24         .equ    localSize,-80
25 # Read only data
26         .section  .rodata
27 format:
28         .string "The sum is %i\n"
29 # Code
30         .text
31         .globl  main
32         .type   main, @function
33 main:
34         pushq   %rbp              # save caller's base pointer
35         movq    %rsp, %rbp        # establish ours
36         addq    $localSize, %rsp  # space for local variables
37                                   #  + argument passing
38         movl    $1, a(%rbp)       # initialize local variables
39         movl    $2, b(%rbp)       # etc...
40         movl    $3, c(%rbp)
41         movl    $4, d(%rbp)
42         movl    $5, e(%rbp)
43         movl    $6, f(%rbp)
44         movl    $7, g(%rbp)
45         movl    $8, h(%rbp)
46         movl    $9, i(%rbp)
47
48         movl    i(%rbp), %eax       # load i
49         movl    %eax, ninth(%rsp)   #   9th argument
50         movl    h(%rbp), %eax       # load h
51         movl    %eax, eighth(%rsp)  #   8th argument
52         movl    g(%rbp), %eax       # load g
53         movl    %eax, seventh(%rsp) #   7th argument
54         movl    f(%rbp), %r9d       # f is 6th
55         movl    e(%rbp), %r8d       # e is 5th
56         movl    d(%rbp), %ecx       # d is 4th
57         movl    c(%rbp), %edx       # c is 3rd
58         movl    b(%rbp), %esi       # b is 2nd
59         movl    a(%rbp), %edi       # a is 1st
60         call    sumNine
61         movl    %eax, total(%rbp)   # total = nineInts(...)
```

```
62
63          movl    total(%rbp), %esi
64          movl    $format, %edi
65          movl    $0, %eax
66          call    printf
67
68          movl    $0, %eax           # return 0;
69          movq    %rbp, %rsp         # delete locals
70          popq    %rbp               # restore caller's base pointer
71          ret                        # back to OS
```

```
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.

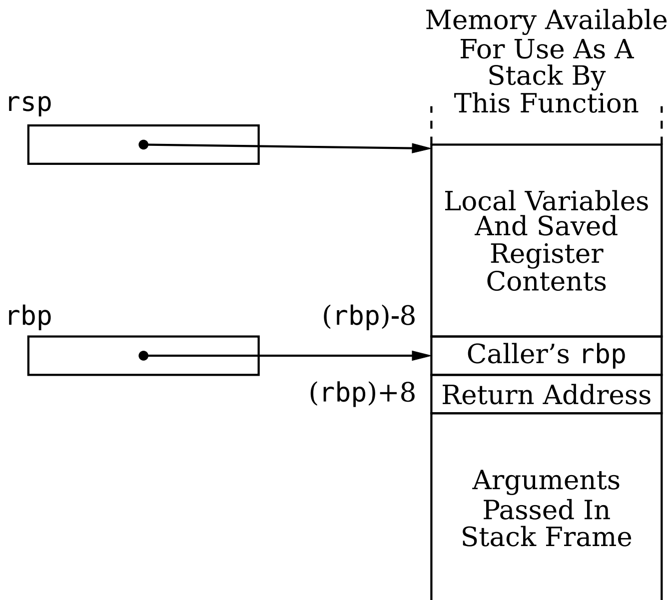


Figure 11.4: Overall layout of the stack frame.

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.
   (b) Establish a new frame pointer at the current top of stack by copying `rsp` to `rbp`.
   (c) 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.
   (d) 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.
   (e) If the function calls another function, save the arguments passed in registers on the stack.
3. *Within* the *called function*:
   (a) `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.
   (b) Arguments passed in registers to the function and saved on the stack are accessed by negative offsets from the frame pointer, `rbp`.
   (c) Arguments passed on the stack to the function are accessed by positive offsets from the frame pointer, `rbp`.
   (d) Local variables are accessed by negative offsets from the frame pointer, `rbp`.
4. When *leaving* the *called function*:
   (a) Place the return value, if any, in `eax`.
   (b) Restore the values in the `rbx`, `rbp`, `rsp`, and `r12` – `r15` registers from the register save area in the stack frame.
   (c) 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"
37          .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.

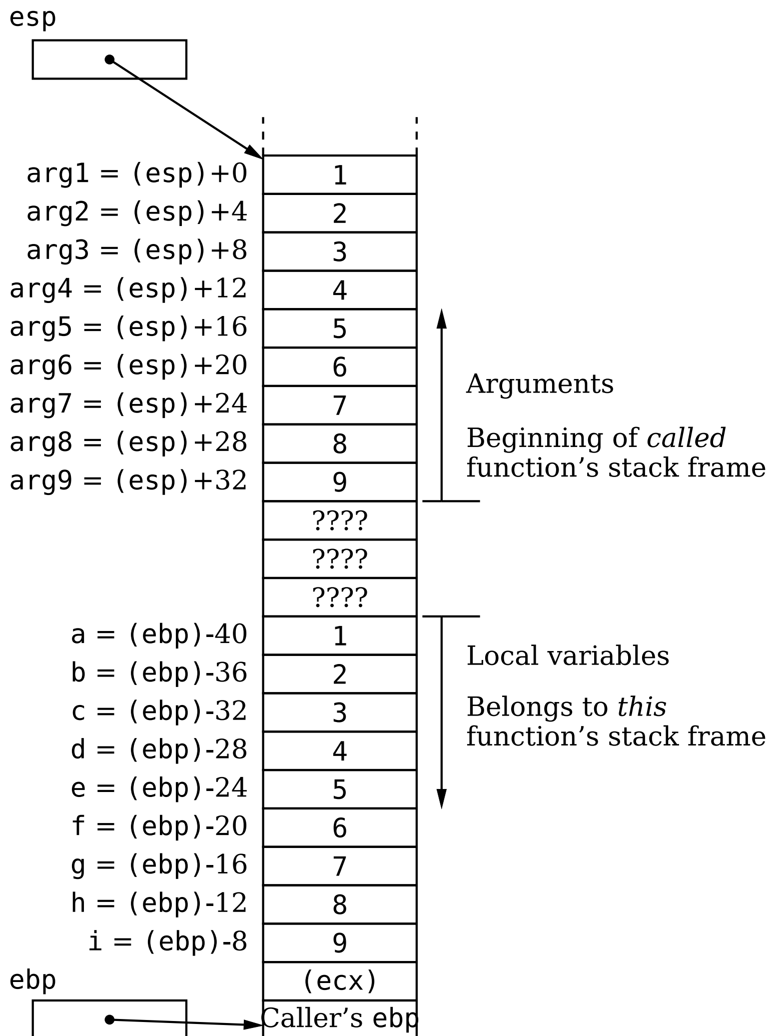


Figure 11.5: Calling function's stack frame, 32-bit mode. Local variables are accessed relative to the frame pointer (`ebp` register). In this example, they are all 4-byte values. Arguments are accessed relative to the stack pointer (`esp` register). Arguments are passed in 4-byte blocks.

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:

opcode	***source***	***destination***	***action***	**page**
`cbtw`			convert byte to word, `al` → `ax`	246
`cwtl`			convert word to long, `ax` → `eax`	246
`cltq`			convert long to quad, `eax` → `rax`	246
`cmov`*cc*	%*reg*/*mem*	%*reg*	conditional move	260
`mov`*s*	$*imm*/%*reg*	%*reg*/*mem*	move	156
`mov`*s*	*mem*	%*reg*	move	156
`movs`*ss*	$*imm*/%*reg*	%*reg*/*mem*	move, sign extend	245
`movz`*ss*	$*imm*/%*reg*	%*reg*/*mem*	move, zero extend	245
`pop`*w*		%*reg*/*mem*	pop from stack	181
`push`*w*	$*imm*/%*reg*/*mem*		push onto stack	181

s = `b`, `w`, `l`, `q`; *w* = `l`, `q`; *cc* = condition codes

arithmetic/logic:

opcode	***source***	***destination***	***action***	**page**
`add`*s*	$*imm*/%*reg*	%*reg*/*mem*	add	214
`add`*s*	*mem*	%*reg*	add	214
`cmp`*s*	$*imm*/%*reg*	%*reg*/*mem*	compare	237
`cmp`*s*	*mem*	%*reg*	compare	237
`dec`*s*	%*reg*/*mem*		decrement	249
`inc`*s*	%*reg*/*mem*		increment	248
`lea`*w*	*mem*	%*reg*	load effective address	191
`sub`*s*	$*imm*/%*reg*	%*reg*/*mem*	subtract	215
`sub`*s*	*mem*	%*reg*	subtract	215
`test`*s*	$*imm*/%*reg*	%*reg*/*mem*	test bits	238
`test`*s*	*mem*	%*reg*	test bits	238

s = `b`, `w`, `l`, `q`; *w* = `l`, `q`

program flow control:

opcode	***location***	***action***	**page**
`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
`j`*cc*	*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

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`

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.

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

2 -1 a) `4567`
b) `89ab`
c) `fedc`
d) `0250`

2 -2 a) `1000 0011 1010 1111`
b) `1001 0000 0000 0001`
c) `1010 1010 1010 1010`
d) `0101 0101 0101 0101`

2 -3 a) 32
b) 48
c) 4
d) 16
e) 8
f) 32

2 -4 a) 2
b) 8
c) 16
d) 3
e) 5
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` b) `85` c) `240` d) `15` e) `128` f) `99` g) `123` h) `255`

2 -7 a) `43981` b) `4660` c) `65244` d) `2000` e) `32768` f) `1024` g) `65535` 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` b) `80` c) `255` d) `137` e) `100` f) `12` g) `17` 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` b) `65535` c) `1024` d) `4369` e) `34952` f) `400` g) `43981` h) `21845`

2 -10 a) `64` b) `7b` c) `0a` d) `58` e) `ff` f) `10` g) `20` h) `80`

2 -11 a) `0400` b) `03e8` c) `8000` d) `7fff` e) `0100` f) `ffff` g) `07d5` 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 `int`s is 0 – 4,294,967,295, so four bytes will be required. If the storage area begins at byte number `0x2fffeb96`, the number will also occupy bytes number `0x2fffeb97`, `0x2fffeb98`, `0x2fffeb99`.

2 -18

```
 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
```

2 -19

```
number  letter grade
  0          A
  1          B
  2          C
  3          D
  4          F
```

2 -26

```
/*
 * echoDecHexAddr.c
 * Asks user to enter a number in decimal or hexadecimal
 * then echoes it in both bases, also showing where values
 * are stored.
 *
 * Bob Plantz - 19 June 2009
 */

#include <stdio.h>

int main(void)
{
    int x;
    unsigned int y;

    while(1)
    {
        printf("Enter a decimal integer: ");
        scanf("%i", &x);
        if (x == 0) break;

        printf("Enter a bit pattern in hexadecimal: ");
        scanf("%x", &y);
        if (y == 0) break;

        printf("%i is stored as %#010x at %p, and\n", x, x, &x);
        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

```
/*
 * echoString1.c
 * Echoes a string entered by user.
 *
 * Bob Plantz - 19 June 2009
 */

#include <unistd.h>
#include <string.h>

int main(void)
{
    char aString[200];
    char *stringPtr = aString;

    write(STDOUT_FILENO, "Enter a text string: ",
         strlen("Enter a text string: "));  // prompt user

    read(STDIN_FILENO, stringPtr, 1);       // get first character
    while (*stringPtr != '\n')              // look for end of line
    {
        stringPtr++;                        // move to next location
        read(STDIN_FILENO, stringPtr, 1);   // get next characte
    }

    // now echo for user
    write(STDOUT_FILENO, "You entered:\n",
         strlen("You entered:\n"));
    stringPtr = aString;
    do
    {
        write(STDOUT_FILENO, stringPtr, 1);
        stringPtr++;
    } while (*stringPtr != '\n');
    write(STDOUT_FILENO, stringPtr, 1);

    return 0;
}
```

2 -31

```
/*
 * echoString2.c
 * Echoes a string entered by user. Converts input
 * to C-style string.
 * Bob Plantz - 19 June 2009
 */

#include <stdio.h>
#include <unistd.h>
#include <string.h>

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
```

```
1  /*
2   * readLn.c
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 #include <unistd.h>
17 #include "readLn.h"
18
19 int readLn(char *stringAddr, int maxLength)
20 {
21     int count = 0;
22     maxLength--;           // allow space for NUL
23     read(STDIN_FILENO, stringAddr, 1);
24     while (*stringAddr != '\n')
25     {
26         if (count < maxLength)
27         {
28             count++;
29             stringAddr++;
30         }
31         read(STDIN_FILENO, stringAddr, 1);
32     }
33     *stringAddr = '\0';    // terminate C string
34
35    return count;
36 }
```

E.3 Computer Arithmetic

3 -1 four

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:

$$\begin{array}{rcl} number & & 32bits(hex) \\ 48 & \longrightarrow & 00000048 \\ +27 & \longrightarrow & 00000027 \\ \hline 75 & & 0000007f \end{array}$$

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.

$$\begin{array}{rcl} number & & 4bits \\ (+4) & \longrightarrow & 0100 \\ +\ (+5) & \longrightarrow & 0101 \\ \hline (-7) & \longleftarrow & 1001 \end{array}$$

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.

$$\begin{array}{rcl} number & & 4bits \\ (-4) & \longrightarrow & 1100 \\ +\ (-5) & \longrightarrow & 1011 \\ \hline (+7) & \longleftarrow & 0111 \end{array}$$

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`
b) `-86`
c) `-16`
d) `+15`
e) `-128`
f) `+99`
g) `+123`

3 -8
a) `+4660`
b) `-4660`
c) `-292`
d) `+2000`
e) `-32768`
f) `+1024`
g) `-1`
h) `+30767`

3 -9
a) `64`
b) `ff`
c) `f6`
d) `58`
e) `7f`
f) `f0`
g) `e0`
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` ⇒ unsigned right
`OF = 0` ⇒ signed right

b) `45`
`CF = 1` ⇒ unsigned wrong
`OF = 0` ⇒ signed right

c) `fb`
`CF = 0` ⇒ unsigned right
`OF = 0` ⇒ signed right

d) `de`
`CF = 0` ⇒ unsigned right
`OF = 1` ⇒ signed wrong

e) `0e`
`CF = 1` ⇒ unsigned wrong
`OF = 0` ⇒ signed right

f) `00`
`CF = 1` ⇒ unsigned wrong
`OF = 1` ⇒ signed wrong

3 -12

a) `0000`
`CF = 1` ⇒ unsigned wrong
`OF = 0` ⇒ signed right

b) `1110`
`CF = 1` ⇒ unsigned wrong
`OF = 0` ⇒ signed right

c) `0000`
`CF = 1` ⇒ unsigned wrong
`OF = 1` ⇒ signed wrong

d) `03ff`
`CF = 1` ⇒ unsigned wrong
`OF = 0` ⇒ signed right

e) `7fff`
`CF = 0` ⇒ unsigned right
`OF = 0` ⇒ signed right

f) `7fff`
`CF = 1` ⇒ unsigned wrong
`OF = 1` ⇒ 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
16 #ifndef HEX2INT_H
17 #define HEX2INT_H
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	1	0
1	1	1

x		$x + 0$
0	0	0
1	0	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		$x \cdot 0$
0	0	0
1	0	0

x		$x + 1$
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:

$F(x, y, z)$

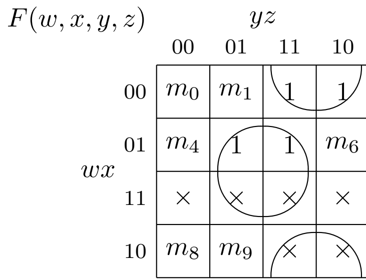
z \ xy	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

4 -10 Minterms:

$F(x, y, z)$

y \ xz	00	01	11	10
0	m_0	m_1	m_5	m_4
1	m_2	m_3	m_7	m_6

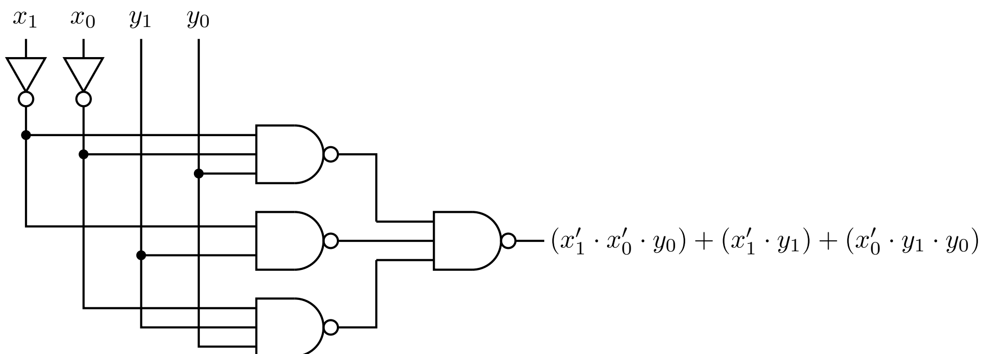
4 -11 The prime numbers correspond to the minterms m_2, m_3, m_5, and m_7. The minterms m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, m_{15} cannot occur so are marked “don’t care” on the Karnaugh map.



$$F(w, x, y, z) = x \cdot z + x' \cdot y$$

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		$Enable = 0$ Next						$Enable = 1$ Next					
n_1	n_0	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	1	0	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.

$J_0(E, n_1, n_0)$

E \ n_1n_0	00	01	11	10
0		1	1	
1	1			1

$K_0(E, n_1, n_0)$

E \ n_1n_0	00	01	11	10
0	1			1
1		1	1	

$J_1(E, n_1, n_0)$

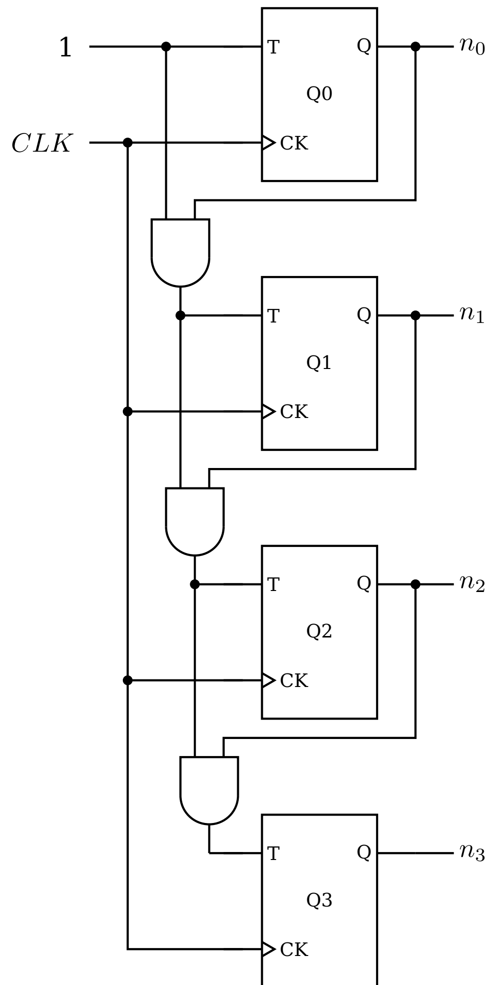
E \ n_1n_0	00	01	11	10
0			1	1
1		1		1

$K_1(E, n_1, n_0)$

E \ n_1n_0	00	01	11	10
0	1	1		
1	1		1	

$$
\begin{aligned}
J_0 &= E' \cdot n_0 + E \cdot n_0' \\
K_0 &= E' \cdot n_0' + E \cdot n_1 \\
J_1 &= E' \cdot n_1 + n_1 \cdot n_0' + E \cdot n_1' \cdot n_0 \\
K_1 &= E' \cdot n_1' + n_1' \cdot n_0' + E \cdot n_1 \cdot n_0
\end{aligned}
$$

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

```
/*
 * endian.c
 * Determines endianess. If endianess cannot be determined
 * from input value, defaults to "big endian"
 * Bob Plantz - 22 June 2009
 */

#include <stdio.h>

int main(void)
{
```

```
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 *) 0x7ffff74f473c
(gdb) x/4xb 0x7ffff74f473c
```

```
0x7ffff74f473c: 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

```
# f.s
# Does nothing but return zero to caller.
# Bob Plantz - 22 June 2009

        .text
        .globl  f
        .type   f, @function
f:
        pushq   %rbp            # save caller's frame pointer
        movq    %rsp, %rbp      # establish ours

        movl    $0, %eax        # return 0;

        movq    %rbp, %rsp      # delete local vars.
        popq    %rbp            # restore caller's frame pointer
        ret                     # return to caller
```

7 -2

```
# g.s
# Does nothing but return to caller.
# Bob Plantz - 22 June 2009

        .text
        .globl  g
        .type   g, @function
g:
        pushq   %rbp            # save caller's frame pointer
        movq    %rsp, %rbp      # establish ours

# A function that returns void has "garbage" in eax.

        movq    %rbp, %rsp      # delete local vars.
        popq    %rbp            # restore caller's frame pointer
        ret                     # return to caller
```

7 -3

```
# h.s
# Does nothing but return 123 to caller.
# Bob Plantz - 22 June 2009
```

```
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:
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          .equu   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.

```
/*
 * yesNo1a.c
 * Prompts user to enter a y/n response.
 *
 * Bob Plantz - 27 June 2009
 */

#include <unistd.h>

static char response[2];

int main(void)
{
    register char *ptr;

    ptr = "Save changes? ";

    while (*ptr != '\0')
    {
        write(STDOUT_FILENO, ptr, 1);
        ptr++;
    }

    read (STDIN_FILENO, response, 2);

    if (*response == 'y')
    {
        ptr = "Changes saved.\n";
        while (*ptr != '\0')
        {
            write(STDOUT_FILENO, ptr, 1);
            ptr++;
        }
    }
    else
    {
        ptr = "Changes discarded.\n";
        while (*ptr != '\0')
        {
            write(STDOUT_FILENO, ptr, 1);
            ptr++;
        }
    }
    return 0;
}
```

10 -10

```
# others.s
# 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,' '          # lowest printable character
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 readLup:
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     readLup         # 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,' '     # lowest printable character
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     readLup             # 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)              # change character
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            # 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 -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          .equu    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 charact
17         .asciz  "are\nechoed 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            # no, one character
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     doneLup         # 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                     # back to caller
```

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   usrprmpt:
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 usrprmpt:
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    $usrprmpt, %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
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
81         cmpb    $'z', (%rax)
82         ja      notLetter
83 isLetter:
84         xorb    $0x20, (%rax)       # flip the case bit
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    arrayLngth,4
11 # read only data
12         .data
13 usrprmpt:
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    $usrprmpt, %edi     # tell user what to do
28         call    writeStr
29 
30         movl    $arrayLngth, %esi # "length" of array
31         leaq    theString(%rbp), %rdi # place for user response
32         call    readLn
33 
34         movl    $usrmsg, %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
```

Index

activation record, 268
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, 329
  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`, 341
  `.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, 396
  control, 4, 135
  data, 4, 135
  synchronous, 396

timing, 395
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, 398
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, 77
ELF, 152
ELF:section, 152
ELF:segment, 153
endian
big, 20
little, 20, 142
exception processing cycle, 386
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, 370
`fp`*n* registers, 370
limitation, 363
range, 361
stack, 371
x87, 365
fractional values, 358
FSM, 104
function
called, 283
calling, 283

designing, 197
epilogue, 155
prologue, 155
writing, 201
functions
32-bit mode, 284
64-bit mode, 274

gate
AND, 61
NAND, 87
NOR, 87
NOT, 62
OR, 62
XOR, 76
gate descriptor, 384
`gdb`, 17
commands, 17, 140, 423
Gray code, 56

handler, 384
Harvard architecture, 4
hexadecimal, 6, 7
human convenience, 16

I/O, 3
devices, 4
isolated, 397
memory-mapped, 397, 398
programming, 4
IDE, 147
identifier, 151, 152
IEEE 754, 363
exponent bias, 363
hidden bit, 363
size, 363
if-else, 250
impedance, 78
implicit argument, 346
Input/Output, 3
instruction
`add`, 214, 219
`and`, 290
`call`, 173
`cbtw`, 246, 317
`cmp`, 237
`dec`, 249
`div`, 315
`idiv`, 317
`imul`, 310
`in`, 397, 398
`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
`cmov`*sf*, 260
`cvtsi2sd`, 370
`in`, 397
`iret`, 387
`out`, 398
`syscall`, 387
`sysret`, 387
instruction set architecture, 1
integer
signed decimal, 37
unsigned decimal, 36
Integrated Development Environment, 147
interrupt handler, 384, 409
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, 346
Memory, 3, 11
memory
    data allocation, 173
    timing, 394
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++, 346
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, 268
    by reference, 268, 338
    by value, 268, 338
    updates, 268
penultimate carry, 43
pipeline, 118
PLD, 100
positional notation, 8
`printf`
    calling, 195
`printf`, 14
    conversion codes, 14
privilege level, 385
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, 271
register
  general-purpose, 132
  names, 132
register file, 121
registers, 121, 131
`register` storage class, 139
repetition, 235
return address, 174
return value, 155, 268
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, 260
SIB byte, 226
sign-extension, 229
significand, 358
SIMD, 365
Single Instruction, Multiple Data, 365
SRAM, 125
SR latch
  Reset, 106
  Set, 106
SSE, 365
  scalar instructions, 367
  vector instructions, 367
stack, 159
  discipline, 177
  operations, 176
  overflow, 177
  pointer, 180
  underflow, 177
  viewing, 183
stack frame, 188, 268
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, 272
variables
  local, 188
vector, 365, 386
von Neumann bottleneck, 4

`while` statement, 236
`write`, 23, 49

x86 architecture, 1