
COMPUTER ORGANIZATION AND DESIGN FUNDAMENTALS

Examining Computer Hardware from the Bottom to the Top

David Tarnoff

Revised First Edition

SML Exhibit 1015

Computer Organization and Design Fundamentals

by David Tarnoff

Copyright © 2005-2007 by David L. Tarnoff. All rights reserved.

Published with the assistance of Lulu.com

This book was written by David L. Tarnoff who is also responsible for the creation of all figures contained herein.

Cover design by David L. Tarnoff

Cover cartoons created by Neal Kegley

Printing History:

July 2005: First edition.

January 2006: Minor corrections to first edition.

July 2007: Added text on Gray code, DRAM technologies, Mealy machines, XOR boolean rules, signed BCD, and hard drive access times. Also made minor corrections.

Legal Notice:

The 3Com[®] name is a registered trademark of the 3Com Corporation.

The Apple[®] name and iTunes[®] name are registered trademarks of Apple Computer, Inc.

The Dell[®] name is a registered trademark of Dell, Inc.

The Intel[®] name, Pentium[®] 4 Processor Extreme Edition, Hyper-Threading Technology[™], and Hyper-Pipelined Technology[™] are registered trademarks of the Intel Corporation.

PowerPC[®] is a registered trademark of International Business Machines Corporation.

The Microsoft[®] name is a registered trademark of the Microsoft Corporation.

While every precaution has been taken to ensure that the material contained in this book is accurate, the author assumes no responsibility for errors or omissions, or for damage incurred as a result of using the information contained in this book.

Please report any errors found to the author at tarnoff@etsu.edu. In addition, suggestions concerning improvements or additions to the text are encouraged. Please direct such correspondence to the author.

TABLE OF CONTENTS

Preface.....	xxi
Chapter One: Digital Signals and Systems	1
1.1 Should Software Engineers Worry About Hardware?	1
1.2 Non-Digital Signals.....	3
1.3 Digital Signals.....	4
1.4 Conversion Systems.....	6
1.5 Representation of Digital Signals	7
1.6 Types of Digital Signals.....	9
1.6.1 Edges.....	9
1.6.2 Pulses.....	9
1.6.3 Non-Periodic Pulse Trains	10
1.6.4 Periodic Pulse Trains.....	11
1.6.5 Pulse-Width Modulation	13
1.7 Unit Prefixes	15
1.8 What's Next?	16
Problems.....	16
Chapter Two: Numbering Systems	17
2.1 Unsigned Binary Counting.....	17
2.2 Binary Terminology.....	20
2.3 Unsigned Binary to Decimal Conversion	20
2.4 Decimal to Unsigned Binary Conversion	23
2.5 Binary Representation of Analog Values.....	25
2.6 Sampling Theory.....	31
2.7 Hexadecimal Representation.....	34
2.8 Binary Coded Decimal.....	36
2.9 Gray Codes.....	37
2.10 What's Next?	40
Problems.....	41
Chapter Three: Binary Math and Signed Representations	43
3.1 Binary Addition.....	43
3.2 Binary Subtraction	45
3.3 Binary Complements.....	46
3.3.1 One's Complement	46
3.3.2 Two's Complement.....	47
3.3.3 Most Significant Bit as a Sign Indicator	50
3.3.4 Signed Magnitude	51

3.3.5 MSB and Number of Bits.....	51
3.3.6 Issues Surrounding the Conversion of Binary Numbers.....	52
3.3.7 Minimums and Maximums	55
3.4 Floating Point Binary.....	57
3.5 Hexadecimal Addition	61
3.6 BCD Addition	64
3.7 Multiplication and Division by Powers of Two.....	65
3.8 Easy Decimal to Binary Conversion Trick.....	67
3.9 Arithmetic Overflow.....	67
3.10 What's Next?	69
Problems	69
Chapter Four: Logic Functions and Gates.....	71
4.1 Logic Gate Basics	71
4.1.1 NOT Gate.....	72
4.1.2 AND Gate	72
4.1.3 OR Gate.....	73
4.1.4 Exclusive-OR (XOR) Gate	74
4.2 Truth Tables.....	75
4.3 Timing Diagrams for Gates	79
4.4 Combinational Logic	80
4.5 Truth Tables for Combinational Logic	83
4.6 What's Next?.....	86
Problems	87
Chapter Five: Boolean Algebra	89
5.1 Need for Boolean Expressions.....	89
5.2 Symbols of Boolean Algebra.....	90
5.3 Boolean Expressions of Combinational Logic	92
5.4 Laws of Boolean Algebra	95
5.5 Rules of Boolean Algebra.....	96
5.5.1 NOT Rule.....	96
5.5.2 OR Rules.....	96
5.5.3 AND Rules.....	97
5.5.4 XOR Rules.....	98
5.5.5 Derivation of Other Rules.....	99
5.6 Simplification.....	101
5.7 DeMorgan's Theorem	103
5.8 What's Next?.....	106
Problems	107

Chapter Six: Standard Boolean Expression Formats.....	109
6.1 Sum-of-Products	109
6.2 Converting an SOP Expression to a Truth Table	110
6.3 Converting a Truth Table to an SOP Expression	112
6.4 Product-of-Sums	114
6.5 Converting POS to Truth Table	115
6.6 Converting a Truth Table to a POS Expression	118
6.7 NAND-NAND Logic	119
6.8 What's Next?	122
Problems.....	123
Chapter Seven: Karnaugh Maps	125
7.1 The Karnaugh Map	125
7.2 Using Karnaugh Maps	129
7.3 "Don't Care" Conditions in a Karnaugh Map.....	137
7.4 What's Next?	138
Problems.....	139
Chapter Eight: Combinational Logic Applications	141
8.1 Adders	141
8.2 Seven-Segment Displays.....	147
8.3 Active-Low Signals.....	151
8.4 Decoders.....	152
8.5 Multiplexers	155
8.6 Demultiplexers	157
8.7 Integrated Circuits.....	159
8.8 What's Next?	163
Problems.....	164
Chapter Nine: Binary Operation Applications	165
9.1 Bitwise Operations.....	165
9.1.1 Clearing/Masking Bits	167
9.1.2 Setting Bits	171
9.1.3 Toggling Bits.....	171
9.2 Comparing Bits with XOR.....	173
9.3 Parity	174
9.4 Checksum.....	175
9.5 Cyclic Redundancy Check	179
9.5.1 CRC Process.....	185
9.5.2 CRC Implementation	187
9.6 Hamming Code	188

9.7 What's Next? 199

Problems 199

Chapter Ten: Memory Cells 203

10.1 New Truth Table Symbols 203

10.1.1 Edges/Transitions 203

10.1.2 Previously Stored Values 204

10.1.3 Undefined Values 204

10.2 The S-R Latch 205

10.3 The D Latch 209

10.4 Divide-By-Two Circuit 212

10.5 Counter 213

10.6 Parallel Data Output 214

10.7 What's Next? 215

Problems 216

Chapter Eleven: State Machines 217

11.1 Introduction to State Machines 217

11.1.1 States 217

11.1.2 State Diagrams 218

11.1.3 Errors in State Diagrams 222

11.1.4 Basic Circuit Organization 222

11.2 State Machine Design Process 225

11.3 Another State Machine Design: Pattern Detection 234

11.4 Mealy Versus Moore State Machines 237

11.5 What's Next? 238

Problems 239

Chapter Twelve: Memory Organization 241

12.1 Early Memory 241

12.2 Organization of Memory Device 242

12.3 Interfacing Memory to a Processor 244

12.3.1 Buses 244

12.3.2 Memory Maps 248

12.3.3 Address Decoding 250

12.3.4 Chip Select Hardware 255

12.4 Memory Mapped Input/Output 259

12.5 Memory Terminology 260

12.5.1 Random Access Memory 260

12.5.2 Read Only Memory 261

12.5.3 Static RAM versus Dynamic RAM 261

12.5.4 Types of DRAM and Their Timing	263
12.5.5 Asynchronous vs. Synchronous Memory	266
12.6 What's Next?	267
Problems.....	267
Chapter Thirteen: Memory Hierarchy	269
13.1 Characteristics of the Memory Hierarchy	269
13.2 Physical Characteristics of a Hard Drive	269
13.2.1 Hard Drive Read/Write Head.....	270
13.2.2 Data Encoding	272
13.2.3 Hard Drive Access Time	275
13.2.4 S.M.A.R.T.	278
13.3 Organization of Data on a Hard Drive	279
13.4 Cache RAM.....	284
13.4.1 Cache Organization	286
13.4.2 Dividing Memory into Blocks	287
13.4.3 Cache Operation.....	289
13.4.4 Cache Characteristics	290
13.4.5 Cache Mapping Functions.....	290
13.4.6 Cache Write Policy	299
13.5 Registers.....	300
13.6 What's Next?	300
Problems.....	301
Chapter Fourteen: Serial Protocol Basics.....	303
14.1 OSI Seven-Layer Network Model	303
14.2 Serial versus Parallel Data Transmission.....	304
14.3 Anatomy of a Frame or Packet	306
14.4 Sample Protocol: IEEE 802.3 Ethernet.....	308
14.5 Sample Protocol: Internet Protocol.....	310
14.6 Sample Protocol: Transmission Control Protocol.....	313
14.7 Dissecting a Frame.....	317
14.8 Additional Resources	320
14.9 What's Next?	322
Problems.....	322
Chapter Fifteen: Introduction to Processor Architecture.....	325
15.1 Organization versus Architecture.....	325
15.2 Components	325
15.2.1 Bus.....	325
15.2.2 Registers	326

15.2.3 Flags.....	327
15.2.4 Buffers.....	328
15.2.5 The Stack.....	329
15.2.6 I/O Ports.....	331
15.3 Processor Level.....	332
15.4 CPU Level.....	333
15.5 Simple Example of CPU Operation.....	334
15.6 Assembly and Machine Language.....	338
15.7 Big-Endian/Little-Endian.....	345
15.8 Pipelined Architectures.....	346
15.9 Passing Data To and From Peripherals.....	350
15.9.1 Memory-Mapped I/O.....	351
15.9.2 Polling.....	353
15.9.3 Interrupts.....	354
15.9.4 Direct Memory Access.....	355
15.9.5 I/O Channels and Processors.....	356
15.10 What's Next?.....	357
Problems.....	357
Chapter Sixteen: Intel 80x86 Base Architecture.....	359
16.1 Why Study the 80x86?.....	359
16.2 Execution Unit.....	360
16.2.1 General Purpose Registers.....	361
16.2.2 Address Registers.....	362
16.2.3 Flags.....	363
16.2.4 Internal Buses.....	365
16.3 Bus Interface Unit.....	365
16.3.1 Segment Addressing.....	366
16.3.2 Instruction Queue.....	370
16.4 Memory versus I/O Ports.....	371
16.5 What's Next?.....	372
Problems.....	373
Chapter Seventeen: Intel 80x86 Assembly Language.....	375
17.1 Assemblers versus Compilers.....	375
17.2 Components of a Line of Assembly Language.....	376
17.3 Assembly Language Directives.....	378
17.3.1 SEGMENT Directive.....	378
17.3.2 .MODEL, .STACK, .DATA, and .CODE Directives.....	380
17.3.3 PROC Directive.....	381

17.3.4 END Directive.....	382
17.3.5 Data Definition Directives	382
17.3.6 EQU Directive.....	383
17.4 80x86 Opcodes.....	385
17.4.1 Data Transfer.....	385
17.4.2 Data Manipulation.....	386
17.4.3 Program Control.....	387
17.4.4 Special Operations	390
17.5 Addressing Modes.....	391
17.5.1 Register Addressing	391
17.5.2 Immediate Addressing.....	392
17.5.3 Pointer Addressing.....	392
17.6 Sample 80x86 Assembly Language Programs.....	393
17.7 Additional 80x86 Programming Resources	397
17.8 What's Next?	398
Problems.....	398
Index	401

TABLE OF FIGURES

1-1	Sample Digital System.....	3
1-2	Continuous Analog Signal with Infinite Resolution	4
1-3	Sample of Discrete Measurements Taken Every 0.1 Sec.....	4
1-4	Samples Taken of an Analog Signal	5
1-5	Slow Sampling Rate Missed an Anomaly.....	5
1-6	Poor Resolution Resulting in an Inaccurate Measurement	5
1-7	Block Diagram of a System to Capture Analog Data	6
1-8	Representation of a Single Binary Signal	8
1-9	Representation of Multiple Digital Signals.....	8
1-10	Alternate Representation of Multiple Digital Signals.....	9
1-11	Digital Transition Definitions	10
1-12	Pulse Waveforms	10
1-13	Non-Periodic Pulse Train.....	10
1-14	Periodic Pulse Train	11
1-15	Periodic Pulse Train with Different Pulse Widths	11
1-16	Periodic Pulse Train with 25% Duty Cycle	13
2-1	Counting in Decimal	17

2-2	Counting in Binary.....	18
2-3	Binary-Decimal Equivalents from 0 to 17	19
2-4	Values Represented By Each of the First 8 Bit Positions	21
2-5	Sample Conversion of 10110100_2 to Decimal.....	21
2-6	Decimal to Unsigned Binary Conversion Flow Chart	24
2-7	Sample Analog Signal of Sound	26
2-8	Effects of Number of Bits on Roundoff Error	32
2-9	Aliasing Effects Due to Slow Sampling Rate	33
2-10	Eight Binary Values Identifying Rotating Shaft Position	38
2-11	Example of a Position Encoder	38
2-12	Conversion from Unsigned Binary to Gray Code.....	39
3-1	Four Possible Results of Adding Two Bits.....	44
3-2	Four Possible Results of Adding Two Bits with Carry.....	44
3-3	Two's Complement Short-Cut.....	49
3-4	Converting a Two's Complement Number to a Decimal	53
3-5	IEEE Standard 754 Floating-Point Formats.....	59
3-6	Duplicate MSB for Right Shift of 2's Complement Values.....	66
4-1	Basic Format of a Logic Gate	71
4-2	Basic Logic Symbols	72
4-3	Operation of the NOT Gate.....	72
4-4	Operation of a Two-Input AND Gate	73
4-5	Operation of a Two-Input OR Gate	74
4-6	Operation of a Two-Input XOR Gate	74
4-7	Sample Three-Input Truth Table.....	75
4-8	Listing All Bit Patterns for a Four-Input Truth Table.....	76
4-9	Inverter Truth Table	77
4-10	Two-Input AND Gate Truth Table	77
4-11	Two-Input OR Gate Truth Table	77
4-12	Two-Input XOR Gate Truth Table.....	78
4-13	Three-Input AND Gate Truth Table With Don't Cares	78
4-14	Sample Timing Diagram for a Three-Input AND Gate	79
4-15	Sample Timing Diagram for a Three-Input OR Gate	79
4-16	Sample Timing Diagram for a Three-Input XOR Gate	79
4-17	Sample Combinational Logic.....	80
4-18	Combinational Logic for a Simple Security System.....	80
4-19	Truth Table for Simple Security System of Figure 4-18	81
4-20	"NOT" Circuits	82
4-21	Schematic "Short-Hand" for Inverted Inputs	82

4-22	Sample of Multi-Level Combinational Logic	83
4-23	Process of Passing Inputs Through Combinational Logic	83
4-24	Steps That Inputs Pass Through in Combinational Logic.....	84
4-25	All Combinations of Ones and Zeros for Three Inputs.....	84
4-26	Step (a) in Sample Truth Table Creation	85
4-27	Step (b) in Sample Truth Table Creation	85
4-28	Step (c) in Sample Truth Table Creation	86
4-29	Step (d) in Sample Truth Table Creation	86
5-1	Schematic and Truth Table of Combinational Logic.....	89
5-2	Boolean Expression for the AND Function	90
5-3	Boolean Expression for the OR Function	91
5-4	Boolean Expression for the NOT Function.....	91
5-5	Circuit Representation of the Boolean Expression $1+0+1$	91
5-6	Sample of Multi-Level Combinational Logic	92
5-7	Creating Boolean Expression from Combinational Logic	93
5-8	Examples of the Precedence of the NOT Function	93
5-9	Example of a Conversion from a Boolean Expression	94
5-10	Commutative Law for Two Variables OR'ed Together	95
5-11	Schematic Form of NOT Rule	96
5-12	Rules of Boolean Algebra	101
5-13	Application of DeMorgan's Theorem.....	105
5-14	Schematic Application of DeMorgan's Theorem.....	106
6-1	Sample Sum-of-Products Binary Circuit	110
6-2	Samples of Single Product (AND) Truth Tables	111
6-3	Sample of a Sum-of-Products Truth Table	111
6-4	Conversion of an SOP Expression to a Truth Table	112
6-5	Sample Product-of-Sums Binary Circuit	115
6-6	Samples of Single Sum (OR) Truth Tables.....	115
6-7	Sample of a Product-of-Sums Truth Table	116
6-8	Sample Sums With Multiple Zero Outputs.....	117
6-9	Conversion of a POS Expression to a Truth Table	118
6-10	Circuit Depiction of DeMorgan's Theorem.....	120
6-11	OR Gate Equals a NAND Gate With Inverted Inputs.....	120
6-12	OR-to-NAND Equivalency Expanded to Four Inputs	120
6-13	Sample SOP Circuit	121
6-14	Sample SOP Circuit with Output OR Gate Replaced	121
6-15	Sample SOP Circuit Implemented With NAND Gates.....	122
7-1	2-by-2 Karnaugh Map Used with Two Inputs	126

7-2	Mapping a 2-Input Truth Table to Its Karnaugh Map	126
7-3	Three-Input Karnaugh Map	127
7-4	Four-Input Karnaugh Map	127
7-5	Identifying the Products in a Karnaugh Map	130
7-6	Karnaugh Map with Four Adjacent Cells Containing '1'	130
7-7	Sample Rectangle in a Three-Input Karnaugh Map.....	133
7-8	Karnaugh Map with a "Don't Care" Elements	138
7-9	Karnaugh Map with a "Don't Care" Elements Assigned	138
8-1	Four Possible Results of Adding Two Bits.....	141
8-2	Block Diagram of a Half Adder.....	142
8-3	Four Possible States of a Half Adder	142
8-4	Logic Circuit for a Half Adder.....	143
8-5	Block Diagram of a Multi-bit Adder.....	144
8-6	Block Diagram of a Full Adder.....	144
8-7	Sum and Carryout Karnaugh Maps for a Full Adder.....	145
8-8	Logic Circuit for a Full Adder	146
8-9	Seven-Segment Display	147
8-10	Displaying a '1' with a 7-Segment Display	147
8-11	A Seven-Segment Display Displaying a Decimal '2'	148
8-12	Block Diagram of a Seven-Segment Display Driver	148
8-13	Segment Patterns for all Hexadecimal Digits	149
8-14	Seven Segment Display Truth Table	149
8-15	Karnaugh Map for Segment 'e'.....	150
8-16	Karnaugh Map for Segment 'e' with Rectangles	150
8-17	Logic Circuit for Segment e of 7-Segment Display.....	151
8-18	Labeling Conventions for Active-Low Signals	152
8-19	Sample Circuit for Enabling a Microwave	153
8-20	Sample Circuit for Delivering a Soda.....	153
8-21	Truth Table to Enable a Device for A=1, B=1, & C=0.....	154
8-22	Digital Circuit for a 1-of-4 Decoder	154
8-23	Digital Circuit for an Active-Low 1-of-4 Decoder	155
8-24	Truth Table for an Active-Low 1-of-8 Decoder	155
8-25	Block Diagram of an Eight Channel Multiplexer	156
8-26	Truth Table for an Eight Channel Multiplexer	156
8-27	Logic Circuit for a 1-Line-to-4-Line Demultiplexer.....	158
8-28	Truth Table for a 1-Line-to-4-Line Demultiplexer	159
8-29	Examples of Integrated Circuits.....	159
8-30	Pin-out of a Quad Dual-Input NAND Gate IC (7400).....	160
8-31	Sample Pin 1 Identifications	160

8-32	Generic Protoboard	161
8-33	Generic Protoboard Internal Connections	161
8-34	Sample Circuit Wired on a Protoboard	162
8-35	Schematic Symbol of a Light-Emitting Diode (LED)	162
8-36	LED Circuit	163
8-37	Switch Circuit.....	163
9-1	Graphic of a Bitwise Operation Performed on LSB	166
9-2	Bitwise AND of 01101011_2 and 11011010_2	166
9-3	Three Sample Bitwise ANDs	168
9-4	Possible Output from a Motion Detector	173
9-5	A Difference in Output Indicates an Error	173
9-6	Simple Error Detection with an XOR Gate.....	174
9-7	Sample Block of Data with Accompanying Datasums	176
9-8	Small Changes in Data Canceling in Checksum.....	179
9-9	Example of Long Division in Binary	181
9-10	Example of Long Division Using XOR Subtraction.....	182
9-11	Sample Code for Calculating CRC Checksums.....	189
9-12	Venn Diagram Representation of Hamming Code	192
9-13	Example Single-Bit Errors in Venn Diagram	192
9-14	Example of a Two-Bit Error	193
9-15	Using Parity to Check for Double-Bit Errors.....	194
10-1	Symbols for Rising Edge and Falling Edge Transitions	204
10-2	Sample Truth Table Using Undefined Output	204
10-3	Primitive Feedback Circuit using Inverters.....	205
10-4	Operation of a NAND Gate with One Input Tied High	206
10-5	Primitive Feedback Circuit Redrawn with NAND Gates	206
10-6	Only Two Possible States of Circuit in Figure 10-5	206
10-7	Operation of a Simple Memory Cell	207
10-8	Operation of a Simple Memory Cell (continued).....	208
10-9	S-R Latch	209
10-10	S-R Latch Truth Table	209
10-11	Block Diagram of the D Latch	209
10-12	Edge-Triggered D Latch Truth Tables	211
10-13	Transparent D Latch Truth Tables	211
10-14	Divide-By-Two Circuit	212
10-15	Clock and Output Timing in a Divide-By-Two Circuit	212
10-16	Cascading Four Divide-By-Two Circuits	213
10-17	Counter Implemented with Divide-By-Two Circuits	213

10-18	Output of Binary Counter Circuit	214
10-19	Output Port Data Latch Circuitry.....	215
11-1	Adding Memory to a Digital Logic Circuit	217
11-2	States of a Traffic Signal System.....	218
11-3	States of a Light Bulb.....	218
11-4	State Diagram for Light Bulb State Machine.....	218
11-5	Complete State Diagram for Light Bulb State Machine	219
11-6	Block Diagram of an Up-Down Binary Counter	220
11-7	State Diagram for a 3-Bit Up-Down Binary Counter	221
11-8	Sample of a Reset Indication in a State Diagram.....	221
11-9	Block Diagram of a State Machine	223
11-10	Initial State of the Push Button Light Control	226
11-11	Transitions from State 0 of Push Button Circuit.....	226
11-12	B=0 Transition from State 0 of Push Button Circuit	227
11-13	B=1 Transition from State 0 of Push Button Circuit	227
11-14	B=0 Transition from State 1 of Push Button Circuit	227
11-15	B=1 Transition from State 1 of Push Button Circuit	228
11-16	Transitions from State 2 of Push Button Circuit.....	228
11-17	Final State Diagram for Push Button Circuit.....	229
11-18	Block Diagram for Push Button Circuit.....	230
11-19	K-Maps for S_1' , S_0' , and L of Push Button Circuit.....	232
11-20	Finished Push Button Circuit	232
11-21	Revised Truth Table and K Map for Push Button Circuit	233
11-22	Identifying the Bit Pattern "101" in a Bit Stream	234
11-23	State Diagram for Identifying the Bit Pattern "101"	235
11-24	Next State and Output Truth Tables for Pattern Detect.....	236
11-25	K-Maps for S_1' , S_0' , and P of Pattern Detect Circuit	237
11-26	Final Circuit to Identify the Bit Pattern "101"	237
11-27	Basic Configuration of a Mealy Machine	238
11-28	Sample State Diagram of a Mealy Machine	238
11-29	Output Truth Table for Sample Mealy Machine.....	239
12-1	Diagram of a Section of Core Memory.....	241
12-2	Basic Organization of a Memory Device.....	243
12-3	Basic Processor to Memory Device Interface.....	245
12-4	Two Memory Devices Sharing a Bus	246
12-5	Three Buffers Trying to Drive the Same Output	248
12-6	Sample Memory Maps.....	249
12-7	Full Address with Enable Bits and Device Address Bits.....	251

12-8	IPv4 Address Divided into Subnet and Host IDs	254
12-9	Sample Chip Select Circuit for a Memory Device	256
12-10	Some Types of Memory Mapped I/O Configurations	260
12-11	Basic Addressing Process for a DRAM	264
12-12	Organization of DRAM	265
12-13	Example of an FPM Transfer	265
12-14	Example of an EDO Transfer	266
13-1	Block Diagram of a Standard Memory Hierarchy	269
13-2	Configuration of a Hard Drive Write Head	271
13-3	Sample FM Magnetic Encoding	273
13-4	Sample MFM Magnetic Encoding	274
13-5	RLL Relation between Bit Patterns and Polarity Changes	274
13-6	Sample RLL Magnetic Encoding	275
13-7	Components of Disk Access Time	277
13-8	Relation between Read/Write Head and Tracks	279
13-9	Organization of Hard Disk Platter	280
13-10	Illustration of a Hard Drive Cylinder	281
13-11	Equal Number of Bits per Track versus Equal Sized Bits	282
13-12	Comparison of Sector Organizations	282
13-13	Cache Placement between Main Memory and Processor	285
13-14	L1 and L2 Cache Placement	285
13-15	Split Cache Organization	286
13-16	Organization of Cache into Lines	287
13-17	Division of Memory into Blocks	288
13-18	Organization of Address Identifying Block and Offset	289
13-19	Direct Mapping of Main Memory to Cache	291
13-20	Direct Mapping Partitioning of Memory Address	292
13-21	Fully Associative Partitioning of Memory Address	295
13-22	Set Associative Mapping of Main Memory to Cache	297
13-23	Effect of Cache Set Size on Address Partitioning	298
14-1	Sample Protocol Stack using TCP, IP, and Ethernet	307
14-2	Layout of an IEEE 802.3 Ethernet Frame	308
14-3	Layout of an IP Packet Header	311
14-4	Layout of a TCP Packet Header	314
14-5	Position and Purpose of TCP Control Flags	315
14-6	Layout of a TCP Pseudo Header	316
14-7	Simulated Raw Data Capture of an Ethernet Frame	317
15-1	Sample Code Using Conditional Statements	328

15-2	Block Diagram of a System Incorporating a Buffer	329
15-3	Generic Block Diagram of a Processor System	332
15-4	Generic Block Diagram of Processor Internals.....	333
15-5	Generic Block Diagram of a Typical CPU	334
15-6	Decoded Assembly Language from Table 15-6	343
15-7	Non-Pipelined Execution of Five Instructions.....	348
15-8	Pipelined Execution of Five Instructions	348
15-9	Sample Memory Mapped Device Circuit	352
15-10	Basic Operation of an ISR	355
16-1	Block Diagram of 80x86 Execution Unit (EU)	360
16-2	Block Diagram of 80x86 Bus Interface Unit (BIU).....	366
16-3	Segment/Pointer Relation in the 80x86 Memory Map	368
17-1	Format of a Line of Assembly Language Code	377
17-2	Format and Parameters Used to Define a Segment.....	379
17-3	Format of the .MODEL Directive.....	380
17-4	Format and Parameters Used to Define a Procedure	381
17-5	Format and Parameters of Some Define Directives.....	383
17-6	Example Uses of Define Directives	384
17-7	Format and Parameters of the EQU Directive	384
17-8	Sample Code with and without the EQU Directive	384
17-9	Format and Parameters of the MOV Opcode.....	385
17-10	Format and Parameters of the IN and OUT Opcodes	385
17-11	Format and Parameters of the ADD Opcode	386
17-12	Format and Parameters of NEG, NOT, DEC, and INC	386
17-13	Format and Parameters of SAR, SHR, SAL, and SHL.....	387
17-14	Example of a JMP Instruction.....	387
17-15	Example of a LOOP Instruction.....	389
17-16	Sample Organization of a Procedure Call.....	390
17-17	Examples of Register Addressing	392
17-18	Examples of Immediate Addressing	392
17-19	Examples of an Address being used as an Operand.....	393
17-20	Skeleton Code for a Simple Assembly Program.....	393
17-21	Code to Assign Data Segment Address to DS Register.....	394
17-22	Code to Inform O/S that Program is Terminated.....	395
17-23	Skeleton Code with Code Added for O/S Support	395
17-24	Data Defining Directives for Example Code	396
17-25	Step-by-Step Example Operation Converted to Code	396
17-26	Final Code for Example Assembly Language Program.....	397

TABLE OF TABLES

1-1	Unit Prefixes.....	15
2-1	Converting Binary to Decimal and Hexadecimal	35
2-2	Converting BCD to Decimal.....	36
2-3	Derivation of the Four-Bit Gray Code	40
3-1	Representation Comparison for 8-bit Binary Numbers	57
3-2	Hexadecimal to Decimal Conversion Table.....	62
3-3	Multiplying the Binary Value 1001_2 by Powers of Two.....	65
8-1	Addition Results Based on Inputs of a Full Adder	144
8-2	Sum and Carryout Truth Tables for a Full Adder	145
9-1	Truth Table for a Two-Input XOR Gate	172
9-2	Addition and Subtraction Without Carries or Borrows.....	181
9-3	Reconstructing the Dividend Using XORs	183
9-4	Second Example of Reconstructing the Dividend.....	184
9-5	Data Groupings and Parity for the Nibble 1011_2	190
9-6	Data Groupings with a Data Bit in Error	190
9-7	Data Groupings with a Parity Bit in Error	191
9-8	Identifying Errors in a Nibble with Three Parity Bits.....	191
9-9	Parity Bits Required for a Specific Number of Data Bits	195
9-10	Membership of Data and Parity Bits in Parity Groups	197
11-1	List of States for Push Button Circuit	230
11-2	Next State Truth Table for Push Button Circuit.....	231
11-3	Output Truth Table for Push Button Circuit	231
11-4	Revised List of States for Push Button Circuit	233
11-5	List of States for Bit Pattern Detection Circuit	236
12-1	The Allowable Settings of Four Chip Selects	247
12-2	Sample Memory Sizes versus Required Address Lines.....	251
15-1	Conditional Jumps to be Placed After a Compare	337
15-2	Conditional Jumps to be Placed After an Operation	338
15-3	Numbered Instructions for Imaginary Processor	340
15-4	Assembly Language for Imaginary Processor	340
15-5	Operand Requirements for Imaginary Processor	341
15-6	A Simple Program Stored at Memory Address 1000_{16}	342
15-7	Signal Values for Sample I/O Device	351
15-8	Control Signal Levels for I/O and Memory Transactions.....	353

16-1 Summary of Intel 80x86 Bus Characteristics 360

16-2 Summary of the 80x86 Read and Write Control Signals..... 372

17-1 Memory Models Available for use with .MODEL 381

17-2 Summary of 80x86 Conditional Jumps..... 388

17-3 80x86 Instructions for Modifying Flags 390

Our discussion so far has focused on logic design as it applies to hardware implementation. Frequently software design also requires the use of binary logic. This section presents some higher-level binary applications, ones that might be found in software. These applications are mostly for error checking and correction, but the techniques used should not be limited to these areas.

9.1 Bitwise Operations

Most software performs data manipulation using mathematical operations such as multiplication or addition. Some applications, however, may require the examination or manipulation of data at the bit level. For example, what might be the fastest way to determine whether an integer is odd or even?

The method most of us are usually taught to distinguish odd and even values is to divide the integer by two discarding any remainder then multiply the result by two and compare it with the original value. If the two values are equal, the original value was even because a division by two would not have created a remainder. Inequality, however, would indicate that the original value was odd. Below is an if-statement in the programming language C that would have performed this check.

```
if(((iVal/2)*2) == iVal)
    // This code is executed for even values
else
    // This code is executed for odd values
```

Let's see if we can't establish another method. As we discussed in Chapter 3, a division by two can be accomplished by shifting all of the bits of an integer one position to the right. A remainder occurs when a one is present in the rightmost bit, i.e., the least significant bit. A zero in this position would result in no remainder. Therefore, if the LSB is one, the integer is odd. If the LSB is zero, the integer is even. This is shown with the following examples.

$$35_{10} = 00100011_2$$

$$93_{10} = 01011101_2$$

$$124_{10} = 01111100_2$$

$$30_{10} = 00011110_2$$

This reduces our odd/even detection down to an examination of the LSB. The question is can we get the computer to examine only a single bit, and if we can, will it be faster than our previous example?

There is in fact a way to manipulate data at the bit level allowing us to isolate or change individual bits. It is based on a set of functions called **bitwise operations**, and the typical programming language provides operators to support them.

The term bitwise operation refers to the setting, clearing, or toggling of individual bits within a binary number. To do this, all processors are capable of executing logical operations (AND, OR, or XOR) on the individual pairs of bits within two binary numbers. The bits are paired up by matching their bit position, performing the logical operation, then placing the result in the same bit position of the destination value.

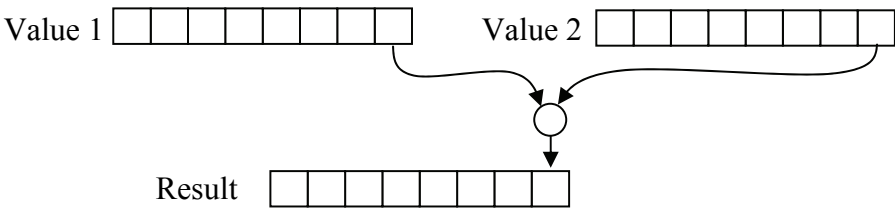


Figure 9-1 Graphic of a Bitwise Operation Performed on LSB

As an example, Figure 9-2 presents the bitwise AND of the binary values 01101011_2 and 11011010_2 .

Value 1	0	1	1	0	1	0	1	1
Value 2	1	1	0	1	1	0	1	0
Resulting AND	0	1	0	0	1	0	1	0

Figure 9-2 Bitwise AND of 01101011_2 and 11011010_2

Remember that the output of an AND is one if and only if all of the inputs are one. In Figure 9-2, we see that ones only appear in the result in columns where both of the original values equal one. In a C program, the bitwise AND is identified with the operator '&'. The example in Figure 9-2 can then be represented in C with the following code.

```
int iVal1 = 0b01101011;  
int iVal2 = 0b11011010;  
int result = iVal1 & iVal2;
```

Note that the prefix '0b' is a non-standard method of declaring a binary integer and is not supported by all C compilers. If your compiler does not support this type of declaration, use the hex prefix '0x' and declare iVal1 to be 0x6B and iVal2 to be 0xDA. As for the other bitwise operators in C, '|' (pipe) is the bitwise OR operator, '^' (caret) is the bitwise XOR operator, and '~' (tilde) is the bitwise NOT operator.

Typically, bitwise operations are intended to manipulate the bits of a single variable. In order to do this, we must know two things: what needs to be done to the bits and which bits to do it to.

As for the first item, there are three operations: clearing bits to zero, setting bits to one, and toggling bits from one to zero and from zero to one. Clearing bits is taken care of with the bitwise AND operation while setting bits is done with the bitwise OR. The bitwise XOR will toggle specific bits.

A bit mask is a binary value that is of the same length as the original value. It has a pattern of ones and zeros that defines which bits of the original value are to be changed and which bits are to be left alone.

The next three sections discuss each of the three types of bitwise operations: clearing bits, setting bits, and toggling bits.

9.1.1 Clearing/Masking Bits

Clearing individual bits, also known as bit masking, uses the bitwise AND to clear specific bits while leaving the other bits untouched. The mask that is used will have ones in the bit positions that are to be left alone while zeros are in the bit positions that need to be cleared.

This operation is most commonly used when we want to isolate a bit or a group of bits. It is the perfect operation for distinguishing odd and even numbers where we want to see how the LSB is set and ignore the remaining bits. The bitwise AND can be used to clear all of the bits except the LSB. The mask we want to use will have a one in the LSB and zeros in all of the other positions. In Figure 9-3, the results of three bitwise ANDs are given, two for odd numbers and one for an even number. By ANDing a binary mask of 00000001₂, the odd numbers have a non-zero result while the even number has a zero result.

This shows that by using a bitwise AND with a mask of 00000001₂, we can distinguish an odd integer from an even integer. Since bitwise

operations are one of the fastest operations that can be performed on a processor, it is the preferred method. In fact, if we use this bitwise AND to distinguish odd and even numbers on a typical processor, it can be twice as fast as doing the same process with a right shift followed by a left shift and over ten times faster than using a divide followed by a multiply.

35 ₁₀ (odd)	0	0	1	0	0	0	1	1
Odd/Even Mask	0	0	0	0	0	0	0	1
Bitwise AND Result	0	0	0	0	0	0	0	1
<hr/>								
93 ₁₀ (odd)	0	1	0	1	1	1	0	1
Odd/Even Mask	0	0	0	0	0	0	0	1
Bitwise AND Result	0	0	0	0	0	0	0	1
<hr/>								
30 ₁₀ (even)	0	0	0	1	1	1	1	0
Odd/Even Mask	0	0	0	0	0	0	0	1
Bitwise AND Result	0	0	0	0	0	0	0	0

Figure 9-3 Three Sample Bitwise ANDs

Below is an if-statement in the programming language C that uses a bitwise AND to distinguish odd and even numbers.

```
if(!(iVal&0b00000001))
    // This code is executed for even values
else
    // This code is executed for odd values
```

The bitwise AND can also be used to clear specific bits. For example, assume we want to separate the nibbles of a byte into two different variables. The following process can be used to do this:

- Copy the original value to the variable meant to store the lower nibble, then clear all but the lower four bits
- Copy the original value to the variable meant to store the upper nibble, then shift the value four bits to the right. (See Section 3.7, "Multiplication and Division by Powers of Two," to see how to shift right using C.) Lastly, clear all but the lower four bits.

This process is demonstrated below using the byte 01101101₂.

Isolating the lower nibble

Original value	0	1	1	0	1	1	0	1
Lower nibble mask	0	0	0	0	1	1	1	1
Resulting AND	0	0	0	0	1	1	0	1

Isolating the upper nibble

Original value	0	1	1	0	1	1	0	1
Shift right 4 places	0	0	0	0	0	1	1	0
Lower nibble mask	0	0	0	0	1	1	1	1
Resulting AND	0	0	0	0	0	1	1	0

The following C code will perform these operations.

```
lower_nibble = iVal & 0x0f;
upper_nibble = (iVal>>4) & 0x0f;
```

Example

Using bitwise operations, write a function in C that determines if an IPv4 address is a member of the subnet 192.168.12.0 with a subnet mask 255.255.252.0. Return a true if the IP address is a member and false otherwise.

Solution

An IPv4 address consists of four bytes or octets separated from one another with periods or "dots". When converted to binary, an IPv4 address becomes a 32 bit number.

The address is divided into two parts: a subnet id and a host id. All of the computers that are connected to the same subnet, e.g., a company or a school network, have the same subnet id. Each computer on a subnet, however, has a unique host id. The host id allows the computer to be uniquely identified among all of the computers on the subnet.

The subnet mask identifies the bits that represent the subnet id. When we convert the subnet mask in this example, 255.255.252.0, to binary, we get 11111111.11111111.11111100.00000000.

The bits that identify the subnet id of an IP address correspond to the positions with ones in the subnet mask. The positions with zeros in the subnet mask identify the host id. In this example, the first 22 bits of any IPv4 address that is a member of this subnet should be the same,

170 Computer Organization and Design Fundamentals

specifically they should equal the address 192.168.12.0 or in binary 11000000.10101000.00001100.00000000.

So how can we determine if an IPv4 address is a member of this subnet? If we could clear the bits of the host id, then the remaining bits should equal 192.168.12.0. This sounds like the bitwise AND. If we perform a bitwise AND on an IPv4 address of this subnet using the subnet mask 255.255.252.0, then the result must be 192.168.12.0 because the host id will be cleared. Let's do this by hand for one address inside the subnet, 192.168.15.23, and one address outside the subnet, 192.168.31.23. First, convert these two addresses to binary.

192.168.15.23 = 11000000.10101000.00001111.00010111
192.168.31.23 = 11000000.10101000.00011111.00010111

Now perform a bitwise AND with each of these addresses to come up with their respective subnets.

IP Address	11000000.10101000.00001111.00010111
Subnet mask	11111111.11111111.11111100.00000000
Bitwise AND	11000000.10101000.00001100.00000000

IP Address	11000000.10101000.00011111.00010111
Subnet mask	11111111.11111111.11111100.00000000
Bitwise AND	11000000.10101000.00011100.00000000

Notice that the result of the first bitwise AND produces the correct subnet address while the second bitwise AND does not. Therefore, the first address is a member of the subnet while the second is not.

The code to do this is shown below. It assumes that the type `int` is defined to be at least four bytes long. The left shift operator '`<<`' used in the initialization of `sbnt_ID` and `sbnt_mask` pushes each octet of the IP address or subnet mask to the correct position.

```
int subnetCheck(int IP_address)
{
    int sbnt_ID = (192<<24)+(168<<16)+(12<<8)+0;
    int sbnt_mask = (255<<24)+(255<<16)+(252<<8)+0;
    if((sbnt_mask & IP_address) == sbnt_ID)
        return 1;
    else return 0;
}
```

9.1.2 Setting Bits

Individual bits within a binary value can be set to one using the bitwise logical OR. To do this, OR the original value with a binary mask that has ones in the positions to be set and zeros in the positions to be left alone. For example, the operation below sets bit positions 1, 3, and 5 of the binary value 10010110_2 . Note that bit position 1 was already set. Therefore, this operation should have no affect on that bit.

Original value	1	0	0	1	0	1	1	0
Mask	0	0	1	0	1	0	1	0
Bitwise OR	1	0	1	1	1	1	1	0

In a C program, the bitwise OR is identified with the operator '|'.

Example

Assume that a control byte is used to control eight sets of lights in an auditorium. Each bit controls a set of lights as follows:

bit 7 – House lighting	bit 3 – Emergency lighting
bit 6 – Work lighting	bit 2 – Stage lighting
bit 5 – Aisle lighting	bit 1 – Orchestra pit lighting
bit 4 – Exit lighting	bit 0 – Curtain lighting

For example, if the house lighting, exit lighting, and stage lighting are all on, the value of the control byte should be 10010100_2 . What mask would be used with the bitwise OR to turn on the aisle lighting and the emergency lighting?

Solution

The bitwise OR uses a mask where a one is in each position that needs to be turned on and zeros are placed in the positions meant to be left alone. To turn on the aisle lighting and emergency lighting, bits 5 and 3 must be turned on while the remaining bits are to be left alone. This gives us a mask of 00101000_2 .

9.1.3 Toggling Bits

We can also toggle or switch the value of individual bits from 1 to 0 or vice versa. This is done using the bitwise XOR. Let's begin our discussion by examining the truth table for a two-input XOR.

Table 9-1 Truth Table for a Two-Input XOR Gate

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

If we cover up the bottom two rows of this truth table leaving only the rows where A=0 visible, we see that the value of B is passed along to X, i.e., if A=0, then X equals B. If we cover up the rows where A=0 leaving only the rows where A=1 visible, it looks like the inverse of B is passed to X, i.e., if A=1, then X equals the inverse of B. This discussion makes a two-input XOR gate look like a programmable inverter. If A is zero, B is passed through to the output untouched. If A is one, B is inverted at the output.

Therefore, if we perform a bitwise XOR, the bit positions in the mask with zeros will pass the original value through and bit positions in the mask with ones will invert the original value. The example below uses the mask 00101110₂ to toggle bits 1, 2, 3, and 5 of a binary value while leaving the others untouched.

Original value	1	0	0	1	0	1	1	0
Mask	0	0	1	0	1	1	1	0
Bitwise XOR	1	0	1	1	1	0	0	0

Example

Assume a byte is used to control the warning and indicator lights on an automotive dashboard. The following is a list of the bit positions and the dashboard lights they control.

- | | |
|----------------------------|------------------------------|
| bit 7 – Oil pressure light | bit 3 – Left turn indicator |
| bit 6 – Temperature light | bit 2 – Right turn indicator |
| bit 5 – Door ajar light | bit 1 – Low fuel light |
| bit 4 – Check engine light | bit 0 – High-beams light |

Determine the mask to be used with a bitwise XOR that when used once a second will cause the left and right turn indicators to flash when the emergency flashers are on.

Solution

The bitwise XOR uses a mask with ones in the positions to be toggled and zeros in the positions to be left alone. To toggle bits 3 and 2 on and off, the mask should have ones only in those positions. Therefore, the mask to be used with the bitwise XOR is 00001100_2 .

9.2 Comparing Bits with XOR

This brings us to our first method for detecting errors in data: comparing two serial binary streams to see if they are equal. Assume that one device is supposed to send a stream of bits to another device. An example of this might be a motion detector mounted in an upper corner of a room. The motion detector has either a zero output indicating the room is unoccupied or a one output indicating that something in the room is moving. The output from this motion detector may look like that shown in Figure 9-4.

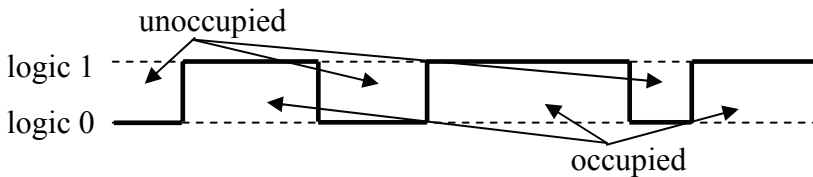


Figure 9-4 Possible Output from a Motion Detector

To verify the output of the motion detector, a second motion detector could be mounted in the room so that the two separate outputs could be compared to each other. If the outputs are the same, the signal can be trusted; if they are different, then one of the devices is in error. At this point in our discussion, we won't know which one.

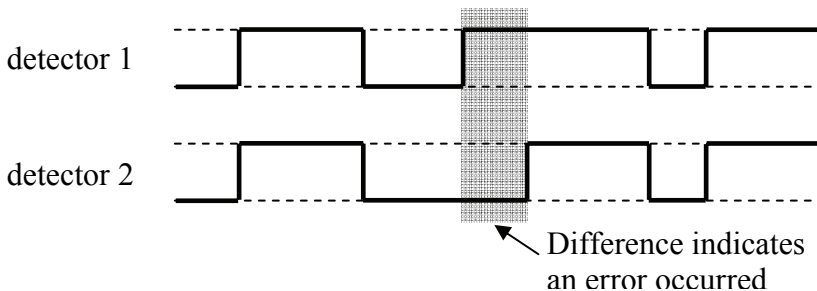


Figure 9-5 A Difference in Output Indicates an Error

A two-input XOR gate can be used here to indicate when an error has occurred. Remember that the output of a two-input XOR gate is a zero if both of the inputs are the same and a one if the inputs are different. This gives us a simple circuit to detect when two signals which should be identical are not.

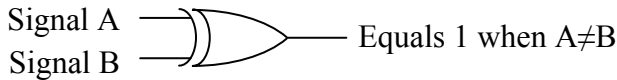


Figure 9-6 Simple Error Detection with an XOR Gate

This circuit will be used later in this chapter to support more complex error detection and correction circuits.

9.3 Parity

One of the most primitive forms of error detection is to add a single bit called a parity bit to each piece of data to indicate whether the data has an odd or even number of ones. It is considered a poor method of error detection as it sometimes doesn't detect multiple errors. When combined with other methods of error detection, however, it can improve their overall performance.

There are two primary types of parity: odd and even. Even parity means that the sum of the ones in the data element and the parity bit is an even number. With odd parity, the sum of ones in the data element and the parity bit is an odd number. When designing a digital system that uses parity, the designers decide in advance which type of parity they will be using.

Assume that a system uses even parity. If an error has occurred and one of the bits in either the data element or the parity bit has been inverted, then counting the number of ones results in an odd number. From the information available, the digital system cannot determine which bit was inverted or even if only one bit was inverted. It can only tell that an error has occurred.

One of the primary problems with parity is that if two bits are inverted, the parity bit appears to be correct, i.e., it indicates that the data is error free. Parity can only detect an odd number of bit errors.

Some systems use a parity bit with each piece of data in memory. If a parity error occurs, the computer will generate a non-maskable interrupt, a condition where the operating system immediately discontinues the execution of the questionable application.

Example

Assume the table below represents bytes stored in memory along with an associated parity bit. Which of the stored values are in error?

Data								Parity
1	0	0	1	0	1	1	0	0
0	0	1	1	1	0	1	0	1
1	0	1	1	0	1	0	1	1
0	1	0	1	1	0	0	1	0
1	1	0	0	0	1	0	1	1

Solution

To determine which data/parity combinations have an error, count the number of ones in each row. The rows with an odd sum have errors while the rows with an even sum are assumed to contain valid data.

Data								Parity	
1	0	0	1	0	1	1	0	0	4 ones – even → no error
0	0	1	1	1	0	1	0	1	5 ones – odd → Error!
1	0	1	1	0	1	0	1	1	6 ones – even → no error
0	1	0	1	1	0	0	1	0	4 ones – even → no error
1	1	0	0	0	1	0	1	1	5 ones – odd → Error!

9.4 Checksum

For digital systems that store or transfer multiple pieces of data in blocks, an additional data element is typically added to each block to provide error detection for the block. This method of error detection is common, especially for the transmission of data across networks.

One of the simplest implementations of this error detection scheme is the *checksum*. As a device transmits data, it takes the sum of all of the data elements it is transmitting to create an aggregate sum. This sum is called the *datasum*. The overflow carries generated by the additions are either discarded or added back into the *datasum*. The transmitting device then sends a form of this *datasum* appended to the end of the block. This new form of the *datasum* is called the *checksum*.

As the data elements are received, they are added a second time in order to recreate the *datasum*. Once all of the data elements have been received, the receiving device compares its calculated *datasum* with the checksum sent by the transmitting device. The data is considered error

free if the receiving device's datasum compares favorably with the transmitted checksum. Figure 9-7 presents a sample data block and the datasums generated both by discarding the two carries and by adding the carries to the datasum.



Figure 9-7 Sample Block of Data with Accompanying Datasums

Upon receiving this transmission, the datasum for this data block must be calculated. Begin by taking the sum of all the data elements.

$$3F_{16} + D1_{16} + 24_{16} + 5A_{16} + 10_{16} + 32_{16} + 89_{16} = 259_{16}$$

The final datasum is calculated by discarding any carries that went beyond the byte width defined by the data block (59_{16}) or by adding the carries to the final sum ($59_{16} + 2 = 5B_{16}$). This keeps the datasum the same width as the data. The method of calculating the datasum where the carries are added to the sum is called the **one's complement sum**.

The checksum shown for the data block in Figure 9-7 is only one of a number of different possible checksums for this data. In this case, the checksum was set equal to the expected datasum. If any of the data elements or if the checksum was in error, the datasum would not equal the checksum. If this happens, the digital system would know that an error had occurred. In the case of a network data transmission, it would request the data to be resent.

The only difference between different implementations of the checksum method is how the datasum and checksum are compared in order to detect an error. As with parity, it is the decision of the designer as to which method is used. The type of checksum used must be agreed upon by both the transmitting and receiving devices ahead of time. The following is a short list of some of the different types of checksum implementations:

- A block of data is considered error free if the datasum is *equal* to the checksum. In this case, the checksum element is calculated by taking the sum of all of the data elements and discarding any carries, i.e., setting the checksum equal to the datasum.

- A block of data is considered error free if the sum of the datasum and checksum results in a binary value with all ones. In this case, the checksum element is calculated by taking the 1's complement of the datasum. This method is called a *1's complement checksum*.
- A block of data is considered error free if the sum of the datasum and checksum results in a binary value with all zeros. In this case, the checksum element is calculated by taking the 2's complement of the datasum. This method is called a *2's complement checksum*.

As shown earlier, the basic checksum for the data block in Figure 9-7 is 59_{16} (01011001_2). The 1's complement checksum for the same data block is equal to the 1's complement of 59_{16} .

$$1's \text{ complement of } 59_{16} = 10100110_2 = A6_{16}$$

The 2's complement checksum for the data block is equal to the 2's complement of 59_{16} .

$$2s \text{ complement of } 59_{16} = 10100111_2 = A7_{16}$$

Example

Determine if the data block and accompanying checksum below are error free. The data block uses a 1's complement checksum.

Data						Checksum
06 ₁₆	00 ₁₆	F7 ₁₆	7E ₁₆	01 ₁₆	52 ₁₆	31 ₁₆

Solution

First, calculate the datasum by adding all the data elements in the data block.

$$\begin{array}{rcl}
 06_{16} & & 06_{16} \\
 + 00_{16} & \nearrow & + F7_{16} \\
 \hline
 06_{16} & & FD_{16} \\
 & \nearrow & + 7E_{16} \\
 & & \hline
 & & 17B_{16} \\
 & \nearrow & + 01_{16} \\
 & & \hline
 & & 7C_{16} \\
 & \nearrow & + 52_{16} \\
 & & \hline
 & & CE_{16}
 \end{array}$$

This gives us a datasum of CE_{16} . If we add this to the checksum 31_{16} we get $CE_{16} + 31_{16} = FF_{16}$, which tells us the data block is error free.

There is a second way to check this data. Instead of adding the datasum to the checksum, you can use the datasum to recalculate the

checksum and compare the result with the received checksum. Taking the 1's complement of CE_{16} gives us:

$$CE_{16} = 11001110_2$$

$$1\text{'s complement of } CE_{16} = 001100001_2 = 31_{16}$$

Example

Write a C program to determine the basic checksum, 1's complement checksum, and 2's complement checksum for the data block 07_{16} , 01_{16} , 20_{16} , 74_{16} , 65_{16} , 64_{16} , $2E_{16}$.

Solution

Before we get started on this code, it is important to know how to take a 1's complement and a 2's complement in C. The 1's complement uses a bitwise not operator '~'. By placing a '~' in front of a variable or constant, the bitwise inverse or 1's complement is returned. Since most computers represent negative numbers with 2's complement notation, the 2's complement is calculated by placing a negative sign in front of the variable or constant.

The code below begins by calculating the datasum. It does this with a loop that adds each value from the array of data values to a variable labeled *datasum*. After each addition, any potential carry is stripped off using a bitwise AND with 0xff. This returns the byte value.

Once the datasum is calculated, the three possible checksum values can be calculated. The first one is equal to the datasum, the second is equal to the bitwise inverse of the datasum, and the third is equal to the 2's complement of the datasum.

```
int datasum=0;
int block[] = {0x07, 0x01, 0x20, 0x74,
               0x65, 0x64, 0x2E};

// This for-loop adds all of the data elements
for(int i=0; i < sizeof(block)/sizeof(int); i++)
    datasum += block[i];

// The following line discards potential carries
datasum &= 0xff;
// Compute each of the three types of checksums
int basic_checksum = datasum;
int ones_compl_checksum = 0xff & (~datasum);
int twos_compl_checksum = 0xff & (-datasum);
```

If we execute this code with the appropriate output statements, we get the following three values for the checksums.

```
The basic checksum is 93
The 1's complement checksum is 6c
The 2's complement checksum is 6d
```

9.5 Cyclic Redundancy Check

The problem with using a checksum for error correction lies in its simplicity. If multiple errors occur in a data stream, it is possible that they may cancel each other out, e.g., a single bit error may subtract 4 from the checksum while a second error adds 4. If the width of the checksum character is 8 bits, then there are $2^8 = 256$ possible checksums for a data stream. This means that there is a 1 in 256 chance that multiple errors may not be detected. These odds could be reduced by increasing the size of the checksum to 16 or 32 bits thereby increasing the number of possible checksums to $2^{16} = 65,536$ or $2^{32} = 4,294,967,296$ respectively.

Assume Figure 9-8 represents a segment of an integer number line where the result of the checksum is identified. A minor error in one of the values may result in a small change in the checksum value. Since the erroneous checksum is not that far from the correct checksum, it is easy for a second error to put the erroneous checksum back to the correct value indicating that there hasn't been an error when there actually has been one.

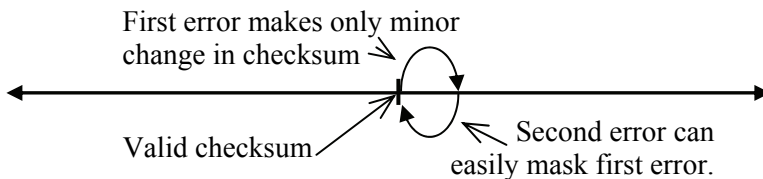


Figure 9-8 Small Changes in Data Canceling in Checksum

What we need is an error detection method that generates vastly different values for small errors in the data. The checksum algorithm doesn't do this which makes it possible for two bit changes to cancel each other in the sum.

A cyclic redundancy check (CRC) uses a basic binary algorithm where each bit of a data element modifies the checksum across its

12.1 Early Memory

Every year new memory technologies are developed promising faster response and higher throughput. This makes it difficult to maintain a printed document discussing the latest advances in memory technologies. Although this chapter does present some basic memory technologies and how they are used to improve performance, the focus is on memory organization and interfacing with the processors.

One of the earliest types of computer memory was called **magnetic core memory**. It was made by weaving fine copper wires through tiny rings of magnetic material in an array. Figure 12-1 shows the basic arrangement of core memory.

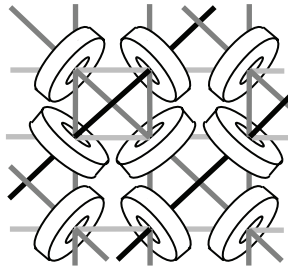


Figure 12-1 Diagram of a Section of Core Memory

Much like recording music to a magnetic tape, when electrical current was sent through the center of one of the magnetic rings, it polarized it with a magnetic charge. Each of these rings could have a charge that flowed clockwise or counter-clockwise. One direction was considered a binary 1 while the other was considered a binary 0.

The horizontal and vertical wires of the core memory were used to write data to a specific ring. By putting half the current necessary to polarize the magnetic material on one of the horizontal wires and the same level of current on one of the vertical wires, the ring where the two wires intersected had enough total current to modify the ring's polarity. The polarity of the remaining rings would be left unaltered.

The diagonal wires, called sense wires, were used to read data. They could detect when the polarity on one of the rings was changed. To read data, therefore, the bit in question would be written to with the horizontal and vertical wires. If the sense wire detected a change in polarity, the bit that had been stored there must have been opposite from the one just written. If no polarity change was detected, the bit written must have been equal to the one stored in that ring.

Magnetic core memory looks almost like fabric, the visible rings nestled among a lacework of glistening copper wires. It is for these reasons, however, that it is also impractical. Since the rings are enormous relative to the scale of electronics, a memory of 1024 bytes (referred to as a 1K x 8 or "1K by 8") had physical dimensions of approximately 8 inches by 8 inches. In addition, the fine copper wires were very fragile making manufacturing a difficult process. A typical 1K x 8 memory would cost thousands of dollars. Therefore, magnetic core memory disappeared from use with the advent of transistors and memory circuits such as the latch presented in Chapter 10.

12.2 Organization of Memory Device

Modern memory has the same basic configuration as magnetic core memory although the rings have been replaced with electronic memory cells such as the D-Latch. The cells are arranged so that each row represents a memory location where a binary value would be stored and the columns represent different bits of those memory locations. This is where the terminology "1K x 8" used in Section 12.1 comes from. Memory is like a matrix where the number of rows identifies the number of memory locations in the memory and the number of columns identifies the number of bits in each memory location.

To store to or retrieve data from a memory device, the processor must place a binary number called an address on special inputs to the memory device. This address identifies which row of the memory matrix or array the processor is interested in communicating with, and enables it.

Once a valid address is placed on the address lines, the memory cells from that row are connected to bi-directional connections on the memory device that allow **data** either to be stored to or read from the latches. These connections are called the data lines. Three additional lines, **chip select**, **read enable**, and **write enable**, are used to control the transaction.

Figure 12-2 presents the basic organization of a memory device.

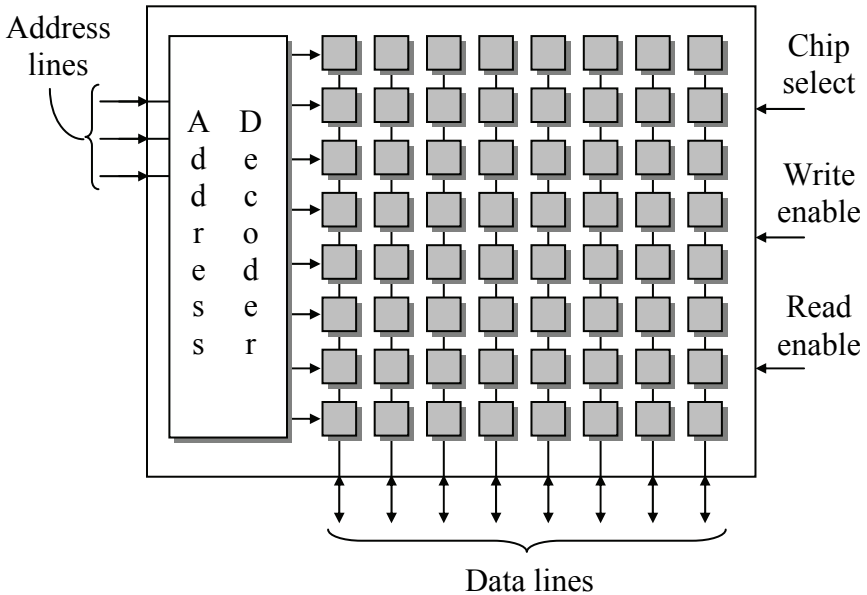


Figure 12-2 Basic Organization of a Memory Device

Remember from Chapter 8 that a decoder with n inputs has 2^n outputs, exactly one of which will be active for each unique pattern of ones and zeros at its input. For example, an active-low 2-input decoder will have four outputs. A different output will equal zero for each unique input pattern while all of the other inputs will be ones.

An **address decoder** selects exactly one row of the memory array to be active leaving the others inactive. When the microprocessor places a binary number onto the address lines, the address decoder selects a single row in the memory array to be written to or read from. For example, if $011_2 = 3_{10}$ is placed on the address lines, the fourth row of the memory will be connected to the data lines. The first row is row 0.

The processor uses the inputs **read enable** and **write enable** to specify whether it is reading data from or writing data to the selected row of the memory array. These signals are active low. When read enable is zero, we are reading data from memory, and when write enable is zero, we are writing data to memory. These two signals should never be zero at the same time.

Sometimes, the read enable and write enable signals are combined into a single line called $R/\overline{W}$ (pronounced "read write-bar"). In this case, a one on $R/\overline{W}$ initiates a data read while a zero initiates a write.

If latches are used for the memory cells, then the data lines are connected to the *D* inputs of the memory location latches when data is written, and they are connected to the *Q* outputs when data is read.

The last input to the memory device shown in Figure 12-2 is the **chip select**. The chip select is an active low signal that enables and disables the memory device. If the chip select equals zero, the memory activates all of its input and output lines and uses them to transfer data. If the chip select equals one, the memory becomes idle, effectively disconnecting itself from all of its input and output lines. The reason for this is that the typical memory device shares the address and data lines of a processor with other devices.

Rarely does a processor communicate with only one memory device on its data lines. Problems occur when more than one device tries to communicate with the processor over shared lines at the same time. It would be like ten people in a room trying to talk at once; no one would be able to understand what was being said.

The processor uses digital logic to control these devices so that only one is talking or listening at a time. Through individual control of each of the chip select lines to the memory devices, the processor can enable only the memory device it wishes to communicate with. The processor places a zero on the chip select of the memory device it wants to communicate with and places ones on all of the other chip select inputs.

The next section discusses how these chip selects are designed so that no conflicts occur.

12.3 Interfacing Memory to a Processor

The previous section presented the input and output lines for a memory device. These lines are shared across all of the devices that communicate with the processor. If you look at the electrical traces across the surface of a motherboard, you should see collections of traces running together in parallel from the processor to then from one memory device to the next. These groups of wires are referred to as the **bus**, which is an extension of the internal structure of the processor. This section discusses how the memory devices share the bus.

12.3.1 Buses

In order to communicate with memory, a processor needs three types of connections: data, address, and control. The **data lines** are the electrical connections used to send data to or receive data from

memory. There is an individual connection or wire for each bit of data. For example, if the memory of a particular system has 8 latches per memory location, i.e., 8 columns in the memory array, then it can store 8-bit data and has 8 individual wires with which to transfer data.

The **address lines** are controlled by the processor and are used to specify which memory location the processor wishes to communicate with. The address is an unsigned binary integer that identifies a unique location where data elements are to be stored or retrieved. Since this unique location could be in any one of the memory devices, the address lines are also used to specify which memory device is enabled.

The **control lines** consist of the signals that manage the transfer of data. At a minimum, they specify the timing and direction of the data transfer. The processor also controls this group of lines. Figure 12-3 presents the simplest connection of a single memory device to a processor with n data lines and m address lines.

Unfortunately, the configuration of Figure 12-3 only works with systems that have a single memory device. This is not very common. For example, a processor may interface with a BIOS stored in a non-volatile memory while its programs and data are stored in the volatile memory of a RAM stick. In addition, it may use the bus to communicate with devices such as the hard drive or video card. All of these devices share the data, address, and control lines of the bus. (BIOS stands for Basic Input/Output System and it is the low-level code used to start the processor when it is first powered up.)

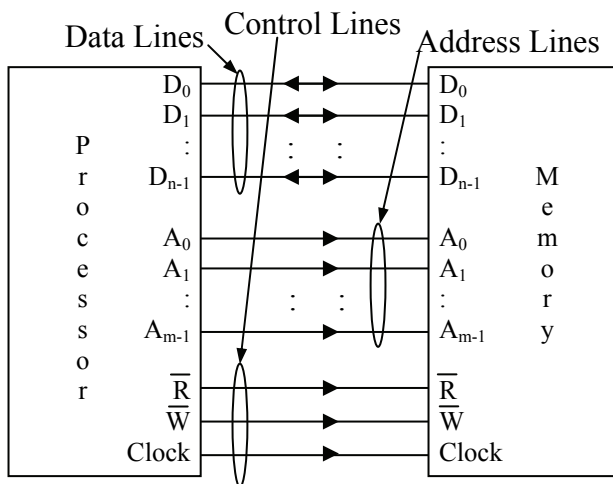


Figure 12-3 Basic Processor to Memory Device Interface

A method had to be developed to allow a processor to communicate to multiple memory devices across the same set of wires. If this wasn't done, the processor would need a separate set of data, address, and control lines for each device placing an enormous burden on circuit board designers for routing wires.

By using a bus, the processor can communicate with exactly one device at a time even though it is physically connected to many devices. If only one device on the bus is enabled at a time, the processor can perform a successful data transfer. If two devices tried to drive the data lines simultaneously, the result would be lost data in a condition called **bus contention**.

Figure 12-4 presents a situation where data is being read from memory device 1 while memory device 2 remains "disconnected" from the bus. Disconnected is in quotes because the physical connection is still present; it just doesn't have an electrical connection across which data can pass.

Notice that Figure 12-4 shows that the only lines disconnected from the bus are the data lines. This is because bus contention only occurs when multiple devices are trying to output to the same lines at the same time. Since only the microprocessor outputs to the address and control lines, they can remain connected.

In order for this scheme to work, an additional control signal must be sent to each of the memory devices telling them when to be connected to the bus and when to be disconnected. This control signal is called a **chip select**.

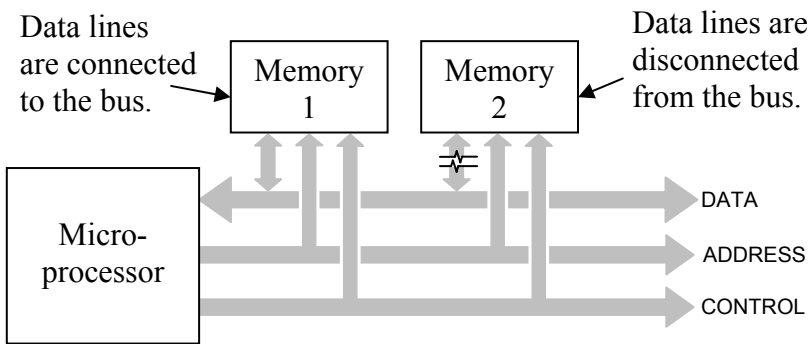


Figure 12-4 Two Memory Devices Sharing a Bus

A chip select is an active low signal connected to the enable input of the memory device. If the chip select is high, the memory device remains idle and its data lines are disconnected from the bus. When the processor wants to communicate with the memory device, it pulls that device's chip select low thereby enabling it and connecting it to the bus.

Each memory device has its own chip select, and at no time do two chip selects go low at the same time. For example, Table 12-1 shows the only possible values of the chip selects for a system with four memory devices.

Table 12-1 The Allowable Settings of Four Chip Selects

	CS ₀	CS ₁	CS ₂	CS ₃
Only memory device 0 connected	0	1	1	1
Only memory device 1 connected	1	0	1	1
Only memory device 2 connected	1	1	0	1
Only memory device 3 connected	1	1	1	0
All devices disconnected	1	1	1	1

The disconnection of the data lines is performed using *tristate outputs* for the data lines of the memory chips. A tristate output is digital output with a third state added to it. This output can be a logic 1, a logic 0, or a third state that acts as a high impedance or open circuit. It is like someone opened a switch and nothing is connected.

This third state is controlled by the chip select. When the active low chip select equals 1, data lines are set to high impedance, sometimes called the **Z state**. A chip select equal to 0 causes the data lines to be active and allow input or output.

In Figure 12-5a, three different outputs are trying to drive the same wire. This results in bus contention, and the resulting data is unreadable. Figure 12-5b shows two of the outputs breaking their connection with the wire allowing the first output to have control of the line. This is the goal when multiple devices are driving a single line. Figure 12-5c is the same as 12-5b except that the switches have been replaced with tristate outputs. With all but one of the outputs in a Z state, the top gate is free to drive the output without bus contention.

The following sections describe how memory systems are designed using chip selects to take advantage of tristate outputs.

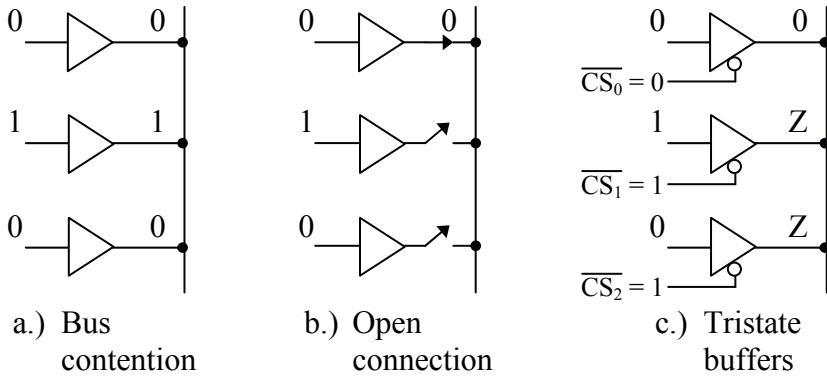


Figure 12-5 Three Buffers Trying to Drive the Same Output

12.3.2 Memory Maps

Think of memory as several filing cabinets where each folder can contain a single piece of data. The size of the stored data, i.e., the number of bits that can be stored in a single memory location, is fixed and is equal to the number of columns in the memory array. Each piece of data can be either code (part of a program) or data (variables or constants used in the program). Code and data are typically stored in the same memory, each piece of which is stored in a unique address or row of memory.

Some sections of memory are assigned to a predefined purpose which may place constraints on how they are arranged. For example, the BIOS from which the computer performs its initial startup sequence is located at a specific address range in non-volatile memory. Video memory may also be located at a specific address range.

System designers must have a method to describe the arrangement of memory in a system. Since multiple memory devices and different types of memory may be present in a single system, hardware designers need to be able to show what addresses correspond to which memory devices. Software designers also need to have a way to show how the memory is being used. For example, which parts of memory will be used for the operating system, which parts will be used to store a program, or which parts will be used to store the data for a program.

System designers describe the use of memory with a **memory map**. A memory map represents a system's memory with a long, vertical column. It is meant to model the memory array where the rows correspond to the memory locations. Within the full range of addresses

are smaller partitions where the individual resources are present. Figure 12-6 presents two examples of memory maps.

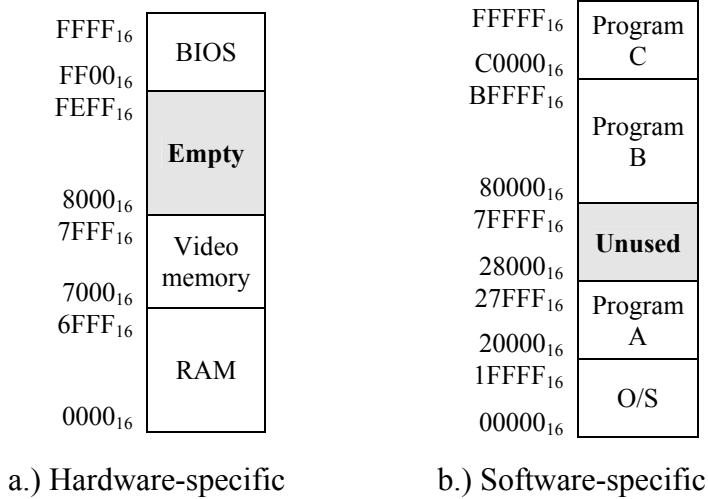


Figure 12-6 Sample Memory Maps

The numbers along the left side of the memory map represent the addresses corresponding to each memory resource. The memory map should represent the full address range of the processor. This full address range is referred to as the processor's **memory space**, and its size is represented by the number of memory locations in the full range, i.e., 2^m where m equals the number of address lines coming out of the processor. It is up to the designer whether the addresses go in ascending or descending order on the memory map.

As an example, let's calculate the memory space of the processor represented by the memory map in Figure 12-6b. The top address for this memory map is $FFFFF_{16} = 1111\ 1111\ 1111\ 1111_2$. Since the processor accesses its highest address by setting all of its address lines to 1, we know that this particular processor has 20 address lines. Therefore, its memory space is $2^{20} = 1,048,576_{10} = 1\text{ Meg}$. This means that all of the memory resources for this processor must be able to fit into 1 Meg without overlapping.

In the next section, we will see how to compute the size of each partition of memory using the address lines. For now, however, we can determine the size of a partition in memory by subtracting the low

address from the high address, then adding one to account for the fact that the low address itself is a memory location. For example, the range of the BIOS in Figure 12-6a starts at $\text{FF00}_{16} = 65,280_{10}$ and goes up to $\text{FFFF}_{16} = 65,535_{10}$. This means that the BIOS fits into $65,535 - 65,280 + 1 = 256$ memory locations.

It is vital to note that there is an exact method to selecting the upper and lower addresses for each of the ranges in the memory map. Take for example the memory range for Program A in Figure 12-6b. The lower address is 20000_{16} while the upper address is $27FFF_{16}$. If we convert these addresses to binary, we should see a relationship.

$$\begin{aligned} 20000_{16} &= 0010\ 0000\ 0000\ 0000\ 0000_2 \\ 27FFF_{16} &= 0010\ 0111\ 1111\ 1111\ 1111_2 \end{aligned}$$

It is not a coincidence that the upper five bits of these two addresses are identical while the remaining bits go from all zeros in the low address to all ones in the high address. Converting the high and the low address of any one of the address ranges in Figure 12-6 should reveal the same characteristic.

The next section shows how these most significant address bits are used to define which memory device is being selected.

12.3.3 Address Decoding

Address decoding is a method for using an address to enable a unique memory device while leaving all other devices idle. The method described here works for many more applications than memory though. It is the same method that is used to identify which subnet a host computer is connected to based on its IP address.

All address decoding schemes have one thing in common: the bits of the full address are divided into two groups, one group that is used to identify the memory device and one group that identifies the memory location within the selected memory device. In order to determine how to divide the full address into these two groups of bits, we need to know how large the memory device is and how large the memory space is. Once we know the size of the memory device, then we know the number of bits that will be required from the full address to point to a memory location within the memory device.

Just as we calculated the size of the memory space of a processor, the size of the memory space of a device is calculated by raising 2 to a power equal to the number of address lines going to that device. For

example, a memory device with 28 address lines going into it has $2^{28} = 256$ Meg locations. This means that 28 address bits from the full address must be used to identify a memory location within that device. All of the remaining bits of the full address will be used to enable or disable the device. It is through these remaining address bits that we determine where the memory will be located within the memory map.

Table 12-2 presents a short list of memory sizes and the number of address lines required to access all of the locations within them. Remember that the memory size is simply equal to 2^m where m is the number of address lines going into the device.

Table 12-2 Sample Memory Sizes versus Required Address Lines

Memory size	Number of address lines	Memory size	Number of address lines
1 K	10	256 Meg	28
256 K	18	1 Gig	30
1 Meg	20	4 Gig	32
16 Meg	24	64 Gig	36

The division of the full address into two groups is done by dividing the full address into a group of most significant bits and least significant bits. The block diagram of an m -bit full address in Figure 12-7 shows how this is done. Each bit of the full address is represented with a_n where n is the bit position.

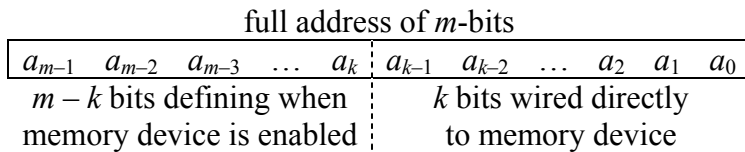


Figure 12-7 Full Address with Enable Bits and Device Address Bits

The bits used to enable the memory device are always the most significant bits while the bits used to access a memory location within the device are always the least significant bits.

Example

A processor with a 256 Meg address space is using the address $35E3C03_{16}$ to access a 16 Meg memory device.

- How many address lines are used to define when the 16 Meg memory space is enabled?
- What is the bit pattern of these enable bits that enables this particular 16 Meg memory device?
- What is the address within the 16 Meg memory device that this address is going to transfer data to or from?
- What is the lowest address in the memory map of the 16 Meg memory device?
- What is the highest address in the memory map of the 16 Meg memory device?

Solution

First, we need to determine where the division in the full address is so that we know which bits go to the enable circuitry and which are connected directly to the memory device's address lines. From Table 12-2, we see that to access 256 Meg, we need 28 address lines. Therefore, the processor must have 28 address lines coming out of it.

The memory device is only 16 Meg which means that it requires 24 address lines to uniquely identify all of its addresses.

a_{27}	a_{26}	a_{25}	a_{24}		a_{23}	a_{22}	...	a_2	a_1	a_0
4 bits that enable memory device					24 bits going to address lines of memory device					

Therefore, the **four most significant address lines** are used to enable the memory device.

By converting $35E3C03_{16}$ to binary, we should see the values of each of these bit positions for this memory location in this memory device.

$$35E3C03_{16} = 0011\ 0101\ 1110\ 0011\ 1100\ 0000\ 0011_2$$

The four most significant bits of this 28-bit address are **0011₂**. This, therefore, is the bit pattern that will enable this particular 16 Meg memory device: $a_{27} = 0$, $a_{26} = 0$, $a_{25} = 1$, and $a_{24} = 1$. Any other pattern

of bits for these four lines will disable this memory device and disallow any data transactions between it and the processor.

The 16 Meg memory device never sees the most significant four bits of this full address. The only address lines it ever sees are the 24 that are connected directly to its address lines: a_0 through a_{23} . Therefore, the address the memory device sees is:

$$0101\ 1110\ 0011\ 1100\ 0000\ 0011_2 = 5E3C03_{16}$$

As for the highest and lowest values of the full address for this memory device, we need to examine what the memory device interprets as its highest and lowest addresses. The lowest address occurs when all of the address lines *to the memory device* are set to 0. The highest address occurs when all of the address lines *to the memory device* are set to 1. Note that this does not include the four most significant bits of the full address which should stay the same in order for the memory device to be active. Therefore, from the standpoint of the memory map which uses the full address, the lowest address is the four enable bits set to 0011_2 followed by 24 zeros. The highest address is the four enable bits set to 0011_2 followed by 24 ones.

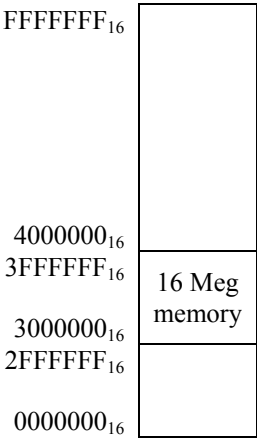
	4 bits that enable memory device				24 bits going to address lines of memory device							
	a_{27}	a_{26}	a_{25}	a_{24}	a_{23}	a_{22}	...	a_2	a_1	a_0		
Highest address	0	0	1	1	1	1	...	1	1	1		
Lowest address	0	0	1	1	0	0	...	0	0	0		

Therefore, from the perspective of the memory map, the lowest and highest addresses of this memory device are:

$$\text{Highest} = 0011\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 3FFFFFF_{16}$$

$$\text{Lowest} = 0011\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 3000000_{16}$$

The following memory map shows how this 16 Meg memory is placed within the full range of the processor's memory space. The full address range of the processor's memory space is determined by the fact that there are 28 address lines from the processor. Twenty-eight ones is $FFFFFFFF_{16}$ in hexadecimal and 28 zeros is 0000000_{16} .



The method for resolving the subnet of an IP address is the same as enabling a specific memory device within a processor's memory space. When configuring a computer to run on a network that uses the Internet Protocol version 4 addressing scheme, it must be assigned a 32-bit address that uniquely identifies it among all of the other computers on that network. This 32-bit address serves a second purpose though: it identifies the sub-network or subnet that this computer is a member of within the entire network. A subnet within the entire IP network is equivalent to a memory device within the memory space of a processor.

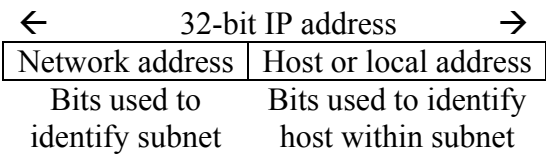


Figure 12-8 IPv4 Address Divided into Subnet and Host IDs

According to IPv4 standard, there are four classes of addressing, Class A, Class B, Class C, and Class D. Each of these classes is defined by the number of bits that are assigned to identify the subnet and how many bits are left for the host ID. For example, a Class A subnet uses 8 bits to identify the subnet. This leaves 24 bits to identify the host within the subnet. Therefore, a Class A network can ideally contain a maximum of $2^{24} = 16,777,216$ hosts. The actual number of hosts is two less. Two addresses for every subnet are reserved: one for a broadcast address and one for the subnet itself.

A Class C network uses 24 bits to identify the subnet and 8 bits to identify the host within the subnet. Therefore, a Class C network can have at most $2^8 - 2 = 254$ machines on it, far fewer than a Class A. The drawback of a Class A network, however, is that if the entire network were assigned to Class A subnets, then there would ideally only be room for $2^8 = 256$ subnets. Whenever the number of bits used to identify the subnet is increased, the number of possible subnets is increased while the number of hosts within a subnet is decreased.

Example

The IPv4 address 202.54.151.45 belongs to a Class C network. What are the subnet and the host ids of this address?

Solution

First, IPv4 addresses are represented as four bytes represented in decimal notation. Therefore, let's convert the IP address above into its 32-bit binary equivalent.

$$202_{10} = 11001010_2$$

$$54_{10} = 00110110_2$$

$$151_{10} = 10010111_2$$

$$45_{10} = 00101101_2$$

This means that the binary address of 202.54.151.45 is:

$$11001010.00110110.10010111.00101101$$

Remember that the Class C network uses the first twenty-four bits for the subnet id. This gives us the following value for the subnet id.

$$\text{Subnet id}_{202.54.151.45} = 110010100011011010010111_2$$

Any IPv4 address with the first 24 bits equal to this identifies a host in this subnet.

The host id is taken from the remaining bits.

$$\text{Host id}_{202.54.151.45} = 00101101_2$$

12.3.4 Chip Select Hardware

What we need is a circuit that will enable a memory device whenever the full address is within the address range of the device and

disable the memory device when the full address falls outside the address range of the device. This is where those most significant bits of the full address come into play.

Remember from our example where we examined the addressing of a 16 Meg memory device in the 256 Meg memory space of a processor that the four most significant bits needed to remain 0011_2 . In other words, if the four bits a_{27} , a_{26} , a_{25} , and a_{24} equaled 0000_2 , 0001_2 , 0010_2 , 0100_2 , 0101_2 , 0110_2 , 0111_2 , 1000_2 , 1001_2 , 1010_2 , 1011_2 , 1100_2 , 1101_2 , 1110_2 , or 1111_2 , the 16 Meg memory device would be disabled. Therefore, we want a circuit that is active when $a_{27} = 0$, $a_{26} = 0$, $a_{25} = 1$, and $a_{24} = 1$. This sounds like the product from an AND gate with a_{27} and a_{26} inverted. Chip select circuits are typically active low, however, so we need to invert the output. This gives us a NAND gate.

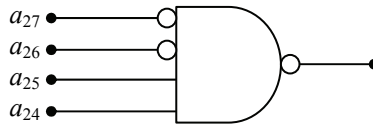


Figure 12-9 Sample Chip Select Circuit for a Memory Device

So the process of designing a chip select is as follows:

- Using the memory space of the processor and the size of the memory device, determine the number of bits of the full address that will be used for the chip select.
- Using the base address where the memory device is to be located, determine the values that the address lines used for the chip select are to have.
- Create a circuit with the address lines for the chip select going into the inputs of a NAND gate with the bits that are to be zero inverted.

Example

Using logic gates, design an active low chip select for a 1 Meg BIOS to be placed in the 1 Gig memory space of a processor. The BIOS needs to have a starting address of $1E00000_{16}$.

Solution

First of all, let's determine how many bits are required by the 1 Meg BIOS. We see from Table 12-2 that a 1 Meg memory device requires

20 bits for addressing. This means that the lower 20 address lines coming from the processor must be connected to the BIOS address lines. Since a 1 Gig memory space has 30 address lines ($2^{30} = 1 \text{ Gig}$), then $30 - 20 = 10$ address lines are left to determine the chip select.

Next, we figure out what the values of those ten lines are supposed to be. If we convert the starting address to binary, we get:

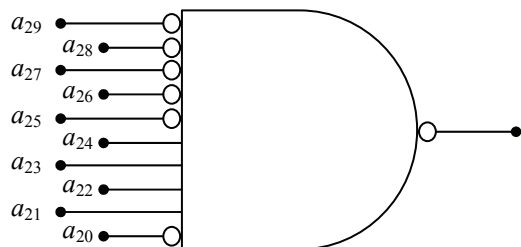
$$1\text{E}00000_{16} = 00\ 0001\ 1110\ 0000\ 0000\ 0000\ 0000\ 0000$$

Notice that enough leading zeros were added to make the address 30 bits long, the appropriate length in a 1 Gig memory space.

We need to assign each bit a label. We do this by labeling the least significant bit a_0 , then incrementing the subscript for each subsequent position to the left. This gives us the following values for each address bit. (a_{18} through a_2 have been deleted in the interest of space.)

a_{29}	a_{28}	a_{27}	a_{26}	a_{25}	a_{24}	a_{23}	a_{22}	a_{21}	a_{20}		a_{19}	a_{18}	...	a_1	a_0
0	0	0	0	0	1	1	1	1	0		0	0	...	0	0

Bits a_{20} through a_{29} are used for the chip select.



Example

What is the largest memory device that can be placed in a memory map with a starting address of $\text{A}40000_{16}$?

Solution

This may seem like a rather odd question, but it actually deals with an important aspect of creating chip selects. Notice that for every one of our starting addresses, the bits that go to the chip select circuitry can be ones or zeros. The bits that go to the address lines of the memory device, however, **must all be zero**. This is because the first address in any memory device is 0_{10} . The ending or highest address will have all ones going to the address lines of the memory device.

Let's begin by converting the address $A40000_{16}$ to binary.

$$A40000_{16} = 1010\ 0100\ 0000\ 0000\ 0000\ 0000_2$$

If we count the zeros starting with the least significant bit and moving left, we see that there are 18 zeros before we get to our first one. This means that the largest memory device we can place at this starting address has 18 address lines. Therefore, the largest memory device we can start at this address has $2^{18} = 256\text{ K}$ memory locations.

Example

True or False: $B000_{16}$ to $CFFF_{16}$ is a valid range for a single memory device.

Solution

This is much like the previous example in that it requires an understanding of how the address lines going to the chip select circuitry and the memory device are required to behave. The previous example showed that the address lines going to the memory device must be all zero for the starting or low address and all ones for the ending or high address. The address lines going to the chip select, however, must all remain constant.

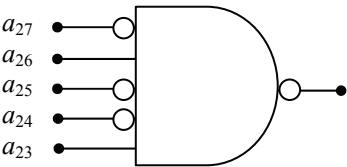
Let's begin by converting the low and the high addresses to binary.

	a_{15}		a_{13}		a_{11}		a_9		a_7		a_5		a_3		a_1		a_0
	a_{14}		a_{12}		a_{10}		a_8		a_6		a_4		a_2				
Low	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
High	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

Note that it is impossible to make a vertical division through both the high and the low addresses where all of the bits to the left are the same for both the high and the low addresses while every bit to the right goes from all zeros for the low address to all ones for the high address. Since we cannot do this, we cannot make a chip select for this memory device and the answer is *false*.

Example

What is the address range of the memory device that is enabled with the chip select shown?



Solution

To begin with, the addressing can be determined from the subscripts of the address lines identified in the figure. The address lines coming out of the processor go from a_0 (always assumed to be the least significant bit of the address) to a_{27} . This means that the processor has 28 address lines and can access a memory space of $2^{28} = 256$ Meg.

The chip select only goes low when all of the inputs to the NAND gate (after the inverters) equal 1. This means that $a_{27} = 0$, $a_{26} = 1$, $a_{25} = 0$, $a_{24} = 0$, and $a_{23} = 1$. We find the lowest address by setting all of the remaining bits, a_{22} through a_0 , to zero and we find the highest address by setting all of the remaining bits to 1. This gives us the following binary addresses.

	a_{27}	a_{26}	a_{25}	a_{24}	a_{23}	a_{22}	a_{21}	...	a_1	a_0
High address	0	1	0	0	1	1	1	...	1	1
Low address	0	1	0	0	1	0	0	...	0	0

When we convert these values to hexadecimal, we get:

High address = $0100\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 4FFFFFF_{16}$

Low address = $0100\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 4800000_{16}$

12.4 Memory Mapped Input/Output

Some devices do not contain a memory array, yet their interface to the processor uses data lines and control lines just like a memory device. For example, an analog-to-digital converter (ADC) reads an analog value and converts it to a digital number that the processor can use. The processor reads this digital value from ADC exactly the same way that it would read a value it had stored in a memory device.

The ADC may also require parameters to be sent to it from the processor. These parameters might include the method it uses for conversion, the time it waits between conversions, and which analog input channels are active. The processor sets these values by writing to the ADC in the same way it would store data to a memory device.

The practice of interfacing an input/output (I/O) device as if it was a memory device is called **memory mapping**. Just like the bus interface for a memory device, the memory mapped interface to a bus uses a chip select to tell the device when it's being accessed and data lines to pass

data between the device and the processor. Some memory mapped I/O even use a limited number of address lines to identify internal registers. In addition, I/O devices use the write enable and read enable signals from the processor to determine whether data is being sent to the device or read from it. Some devices may only support writing (purely output) while others may only support reading (purely input).

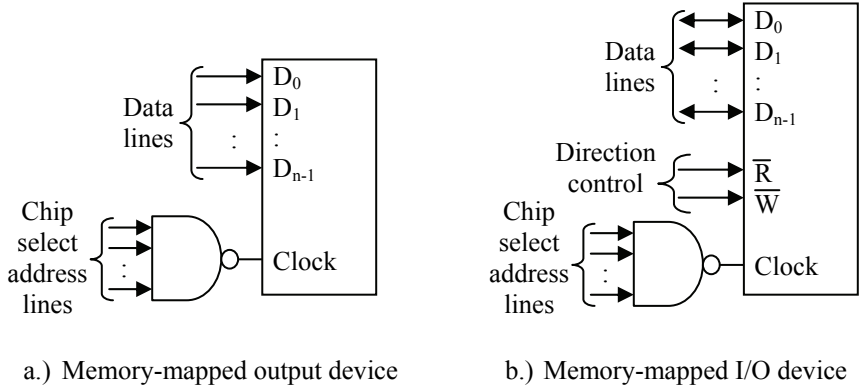


Figure 12-10 Some Types of Memory Mapped I/O Configurations

12.5 Memory Terminology

There are many different purposes for memory in the operation of a computer. Some memory is meant to store data and programs only while the computer is turned on while other memory is meant to be permanent. Some memory contains application code while other memory is meant to store the low-level driver code to control devices such as an IDE interface or a video card. Some memory may have a larger capacity while other memory may be faster.

In order to understand what memory technologies to apply to which processor operation, we need to understand a little bit more about the technologies themselves. This section discusses some of the terminology used to describe memory.

12.5.1 Random Access Memory

The term **Random Access Memory** (RAM) is typically applied to memory that is easily read from and written to by the microprocessor. In actuality, this is a misuse of this term. For a memory to be random access means that any address can be accessed at any time. This is to differentiate it from storage devices such as tapes or hard drives where

the data is accessed sequentially. We will discuss hard drive technologies in Chapter 13.

In general, RAM is the main memory of a computer. Its purpose is to store data and applications that are currently in use. The operating system controls the use of this memory dictating when items are to be loaded into RAM, where they are to be located in RAM, and when they need to be removed from RAM. RAM is meant to be very fast both for reading and writing data. RAM also tends to be volatile in that as soon as power is removed, all of the data is lost.

12.5.2 Read Only Memory

In every computer system, there must be a portion of memory that is stable and impervious to power loss. This kind of memory is called **Read Only Memory** or ROM. Once again, this term is a misnomer. If it was not possible to write to this type of memory, we could not store the code or data that is to be contained in it. It simply means that without special mechanisms in place, a processor cannot write to this type of memory. If through an error of some sort, the processor tries to write to this memory, an error will be generated.

The most common application of ROM is to store the computer's BIOS. Since the BIOS is the code that tells the processor how to access its resources upon powering up, it must be present even when the computer is powered down. Another application is the code for embedded systems. For example, it is important for the code in your car's computer to remain even if the battery is disconnected.

There are some types of ROM that the microprocessor can write to, but usually the time needed to write to them or the programming requirements needed to do so make it unwise to write to them regularly. Therefore, these memories are still considered read only.

In some cases, the processor cannot write to a ROM under any circumstances. For example, the code in your car's computer should never need to be modified. This ROM is programmed before it is installed. To put a new program in the car's computer, the old ROM is removed and discarded and a new ROM is installed in its place.

12.5.3 Static RAM versus Dynamic RAM

For as long as memory has existed, scientists and engineers have experimented with new technologies to make RAM faster and to cram more of it into a smaller space, two goals that are typically at odds.

Nowhere is this more obvious than in the two main classifications of RAM: Static RAM (SRAM) and Dynamic RAM (DRAM).

SRAM is made from an array of latches such as the D-latch we studied in Chapter 10. Each latch can maintain a single bit of data within a single memory address or location. For example, if a memory stores eight bits per memory address, then there are eight latches for a single address. If this same memory has an address space of 256 K, then there are $2^{18} \cdot 8 = 2^{21} = 2,097,152$ latches in the device.

Latches are not small devices as logic circuits go, but they are very fast. Therefore, in the pursuit of the performance goals of speed and size, SRAMs are better adapted to speed. In general, SRAMs:

- store data in transistor circuits similar to D-latches;
- are used for very fast applications such as RAM caches (discussed in Chapter 13);
- tend to be used in smaller memories which allows for very fast access due to the simpler decoding logic; and
- are volatile meaning that the data remains stored only as long as power is available.

There are circuits that connect SRAMs to a back up battery that allows the data to be stable even with a loss of power. These batteries, about the size of a watch battery, can maintain the data for long periods of time much as a battery in a watch can run for years. On the negative side, the extra battery and circuitry adds to the overall system cost and takes up physical space on the motherboard

A bit is stored in a DRAM using a device called a capacitor. A capacitor is made from a pair of conductive plates that are held parallel to each other and very close together, but not touching. If an electron is placed on one of the plates, its negative charge will force an electron on the other plate to leave. This works much like the north pole of a magnet pushing away the north pole of a second magnet.

If enough electrons are deposited on the one plate creating a strong negative charge, enough electrons will be moved away from the opposite plate creating a positive charge. Like a north pole attracting the south pole of a second magnet, the charges on these two plates will be attracted to each other and maintain their charge. This is considered a logic '1'. The absence of a charge is considered a logic '0'.

Since a capacitor can be made very small, DRAM technology is better adapted to high density memories, i.e., cramming a great deal of bits into a small space.

Capacitors do have a problem though. Every once in a while, one of the electrons will escape from the negatively charged plate and land on the positively charged plate. This exchange of an electron decreases the overall charge difference on the two plates. If this happens enough, the stored '1' will disappear. This movement of electrons from one plate to the other is referred to as *leakage current*.

The electrons stored on the plates of the capacitors are also lost when the processor reads the data. It requires energy to read data from the capacitors, energy that is stored by the position of the electrons. Therefore, each read removes some of the electrons.

In order to avoid having leakage current or processor reads corrupt the data stored on the DRAMs, i.e., turning the whole mess to zeros, additional logic called refresh circuitry is used that periodically reads the data in the DRAM then restores the ones with a full charge of electrons. This logic also recharges the capacitors each time the processor reads from a memory location. The refresh circuitry is included on the DRAM chip making the process of keeping the DRAM data valid transparent to the processor. Although it adds to the cost of the DRAM, the DRAM remains cheaper than SRAM.

The refresh process involves disabling memory circuitry, then reading each word of data and writing it back. This is performed by counting through the rows. The process does take time thus slowing down the apparent performance of the memory.

In general, DRAMs:

- have a much higher capacity due to the smaller size of the capacitor (The RAM sticks of your computer's main memory are DRAMs.);
- will "leak" charge due to the nature of capacitors eventually causing the data to disappear unless it is refreshed periodically;
- are much cheaper than SRAM; and
- are volatile meaning that the data is fixed and remains stored only as long as power is available.

12.5.4 Types of DRAM and Their Timing

The basic DRAM storage technology is unchanged since first RAM chips, but designers have used alternate techniques to improve the

effective performance of the DRAM devices. For example, the number of pins that electrically connect the DRAM to the system can be quite large, one of the largest culprits being addressing. A 1 Gbyte memory, for example, requires 30 pins for addressing. If the number of pins could be reduced, the memory would have a smaller footprint on the circuit board and a more reliable connection.

The number of address lines can be cut in half by presenting the memory address to the DRAM in two stages. During the first stage, the first half of the address is presented on the address lines and stored or latched in the memory device. The second stage presents the last half of the address on the same pins. Once it receives both halves of the address, the DRAM can process the request.

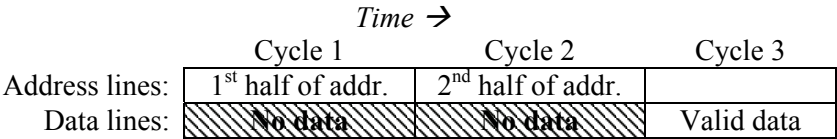


Figure 12-11 Basic Addressing Process for a DRAM

This allows the addressable space of the DRAM to be twice that which could be supported by the address pins. Unfortunately, it comes at the cost of the delay of the second addressing cycle. It will be shown later, however, that this can be turned into an advantage.

The presentation of the address in two stages is viewed as a logical reorganization of the memory into three-dimensions. The upper half of the address identifies a row. The lower half of the address identifies a column. The intersection of the row and column is where the data is stored, the multiple bits of the storage location is the third dimension. This concept of rows and columns is represented in Figure 12-12 for a memory with four data bits per addressable location.

To support the two-stage delivery of the address, two additional active-low control signals are needed. The first is called *row address select* or $\overline{RAS}$. This signal is used to strobe the row address into the DRAM's row address latch. The second, column address select or $\overline{CAS}$, strobes the column address into the DRAM's column address latch. This requires two additional input pins on the DRAM.

By using the two-stage addressing of a DRAM, the addition of a single address line will quadruple the memory size. This is because an

additional address line doubles both the number of rows and the number of columns.

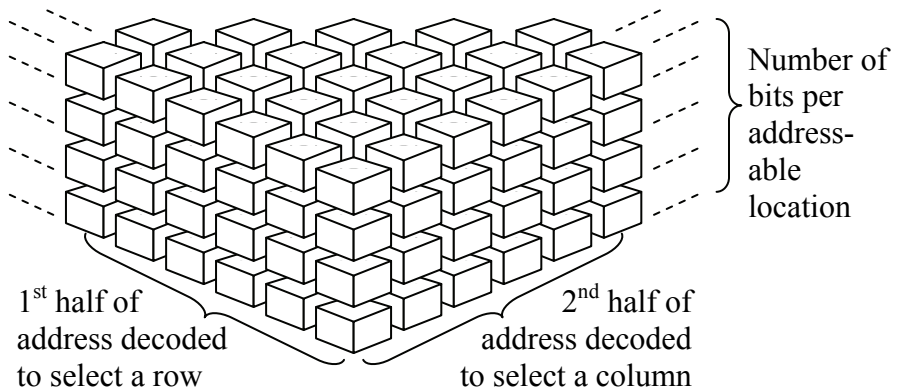


Figure 12-12 Organization of DRAM

Now let's get back to the issue of the delay added by a second address cycle. Most data transfers to and from main memory take the form of block moves where a series of instructions or data words are pulled from memory as a group. (For more information on memory blocks, see the section on caches in Chapter 13.)

If the processor needs a block from memory, the first half of the address should be the same for all the items of the block. Because of this, the memory access process begins with the row address then uses only the column address for subsequent retrievals. This is called Fast Page Mode (FPM), the data of a single row being referred to as a page.

The $\overline{RAS}$ line is held low as long as the row address is valid. Figure 12-13 presents an example of FPM for a memory block of size four.

		Time →							
Address:	Row addr	Col. addr. 0		Col. addr. 1		Col. addr. 2		Col. addr. 3	
	No data	No data	Data word 0	No data	Data word 1	No data	Data word 2	No data	Data word 3

Figure 12-13 Example of an FPM Transfer

FPM requires both the row and column addresses to be presented for the first cycle only. For subsequent addresses, only the column address is needed thereby saving one address cycle. Furthermore, since the decoding circuitry for the column address only uses half of the bits of the full address, the time it takes to decode each individual column address is shorter than it would have been for the full address.

The memory access time can be further shorted by having the processor present the column address for the next data element while the current data element is being returned on the data lines. This overlap of the DRAM returning a word of data while the processor is writing the column address for the next word is called Extended Data-Out (EDO). For data reads within the same page, EDO results in a savings of approximately 10 ns for each read. Figure 12-14 presents an example of EDO for a memory block of size four.

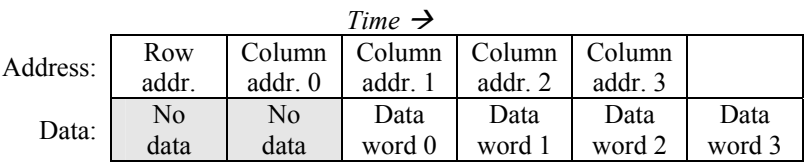


Figure 12-14 Example of an EDO Transfer

If the processor needs to fetch a sequence of four data words from memory, Burst EDO (BEDO) can further speed up the process by using the initial column address, and then using a counter to step through the next three addresses. This means that the processor would only be required to send the row address and column address once, then simply clock in the four sequential data words.

12.5.5 Asynchronous versus Synchronous Memory Interfacing

In all logic circuits, there is a delay between the time that inputs are set and the outputs appear. The inputs of a memory include address lines and control lines while the data lines can be either inputs or outputs. When a processor sets the inputs of a memory device, it has to wait for the memory to respond. This is called *asynchronous* operation.

Asynchronous operation makes it difficult to design the timing of motherboards. The processor has to run slow enough to account for the slowest type and size of memory that is expected to be used. One

memory may be ready with the data before the processor expects it while a different memory may take longer.

Some processors, however, are designed so that the memory follows a precise timing requirement governed by a clock that is added to the bus to keep everything attached in lock-step. This type of interface is referred to as *synchronous* and the memory that uses this type of interface is referred to as synchronous DRAM or SDRAM.

The main benefit derived from controlling the memory interface with a clock is that the memory's internal circuitry can be set up in what is referred to as a pipelined structure. Pipelining is discussed in Chapter 15, but at this point it is sufficient to say that pipelining allows a device to perform multiple operations at the same time. For example, an SDRAM might be able to simultaneously output data to the processor and receive the next requested address. Overlapping functions such as these allows the memory to appear faster.

One improvement to SDRAM allows two words of data to be written or read during a single clock cycle. This means that every time the processor accesses the memory it transfers two words, one on the rising edge of the clock and one on the falling edge. This type of memory is referred to as Double Data Rate SDRAM or DDR SDRAM. DDR2 and DDR3 double and quadruple the DDR rate by allowing four and eight data words respectively to be transferred during a single clock pulse. This is made possible by taking advantage of the fact that the when a DRAM row is accessed, simply counting through the columns retrieves consecutive bytes of data which can be buffered and send out the data lines as quickly as the processor can receive them.

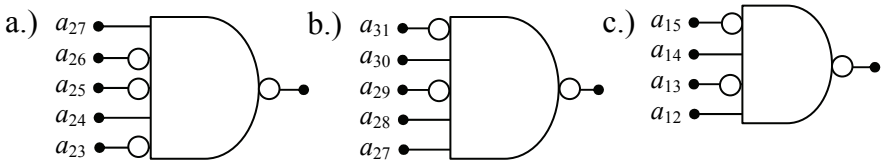
12.6 What's Next?

This chapter has only examined a small part of the information storage solutions used in a computer system. In the next section, we will discuss the operation and purpose of all levels of data storage by examining the different characteristics of each. The major characteristics that will be examined are speed, size, and whether data is lost with a power loss. Each level has specific needs and therefore is implemented with different technologies.

Problems

1. What is the largest memory that can have a starting or lowest address of 160000_{16} ?

2. What are the high and low addresses of the memory ranges defined by each of the chip selects shown below?



3. What is the processor memory space for each chip select in problem 2?
4. What is the memory device size for each chip select in problem 2?
5. How many 16 K memories can be placed (without overlapping) in the memory space of a processor that has 24 address lines?
6. Using logic gates, design an active low chip select for the memory device described in each of the following situations.
- a.) A 256 K memory device starting at address 280000_{16} in a 4 Meg memory space
- b.) A memory device in the range 30000_{16} to $37FFF_{16}$ in a 1 Meg memory space.
7. How many latches are contained in a SRAM that has 20 address lines and 8 data lines?
8. True or false: DRAM is faster than SRAM.
9. True or false: DRAM is cheaper per bit than SRAM.
10. True or false: More DRAM can be packed into the same area (higher density) than SRAM.
11. Which is usually used for smaller memories, DRAM or SRAM?
12. When data is passed from a memory chip to the processor, what values do the bus signals $\overline{R}$ and $\overline{W}$ have?
13. What is the subnet and host id of the Class C IPv4 address 195.164.39.2?
14. Taking into account the addresses for the subnet and broadcast, how many hosts can be present on a Class C IPv4 subnet?

13.1 Characteristics of the Memory Hierarchy

We've discussed the organization and operation of RAM, but RAM is only one level of the group of components used to store information in a computer. The hard drive, for example, stores all of the data and code for the computer in a non-volatile format, and unless a file has been opened, this information can only be found on the hard drive.

Even though the hard drive stores all of the data necessary to operate the computer, other storage methods are needed. This is for a number of reasons, most notably the hard drive is slow and running programs from it would be impossible. When the processor needs data or applications, it first loads them into main memory (RAM).

Main memory and the hard drive are two levels of the computer's **memory hierarchy**. A memory hierarchy is an organization of storage devices that takes advantage of the characteristics of different storage technologies in order to improve the overall performance of a computer system. Figure 13-1 presents the components of the standard memory hierarchy of a computer. Each of these components and their function in the hierarchy is described in this chapter.

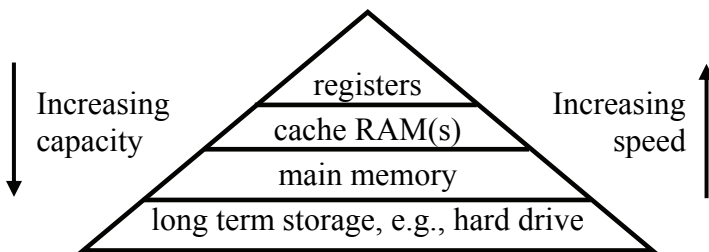


Figure 13-1 Block Diagram of a Standard Memory Hierarchy

13.2 Physical Characteristics of a Hard Drive

At the bottom of the hierarchy is long-term, high-capacity storage. This type of storage is slow making a poor choice for the processor to use for execution of programs and data access. It is, however, necessary to provide computer systems with high capacity, non-volatile storage.

Hard drives are the most cost-effective method of storing data. In the mid-1980's, a 30 Megabyte hard drive could be purchased for around \$300 or about \$10 per MB. In 2007, retailers advertised a 320 Gigabyte SATA Hard drive for around \$80 or about \$0.00025 per MB. In other words, the cost to store a byte of data is almost 1/40,000th cheaper today than it was a little over two decades ago.

Hard drives store data in well-organized patterns of ones and zeros across a thin sheet of magnetic material. This magnetic material is spread either on one or both sides of a lightweight, rigid disk called a **substrate**. The substrate needs to be lightweight because it is meant to spin at very high speeds. The combination of magnetic material and substrate is called a **platter**.

The more rigid the substrate is, the better the reliability of the disk. This was especially true when the mechanisms that were used to read and write data from and to the disks were fixed making them prone to scraping across the substrate's surface if the substrate was not perfectly flat. The condition where the read-write mechanism comes in contact with the disk is called a "crash" which results in magnetic material being scraped away from the disk.

Substrates used to be made from aluminum. Unfortunately, extreme heat sometimes warped the aluminum disk. Now glass is used as a substrate. It improves on aluminum by adding:

- better surface uniformity which increases reliability;
- fewer surface defects which reduces read/write errors;
- better resistance to warping;
- better resistance to shock; and
- the ability to have the read/write mechanism ride closer to the surface allowing for better data density.

13.2.1 Hard Drive Read/Write Head

Data is recorded to the platter using a conductive coil called a head. Older drives and floppy drives use the same head for reading the data too. The head is shaped like a "C" with the gap between the ends positioned to face the magnetic material. A coil of wire is wrapped around the portion of the head that is furthest from the magnetic material. Figure 13-2 shows the configuration of this type of head.

In order to write data, an electrical current is passed through the wire creating a magnetic field within the gap of the head close to the disk.

This field magnetizes the material on the platter in a specific direction. Reversing the current would polarize the magnetic material in the opposite direction. By spinning the platter under the head, patterns of magnetic polarization can be stored in circular paths on the disk. By moving the head along the radius, nested circular paths can be created. The magnetized patterns on the platter represent the data.

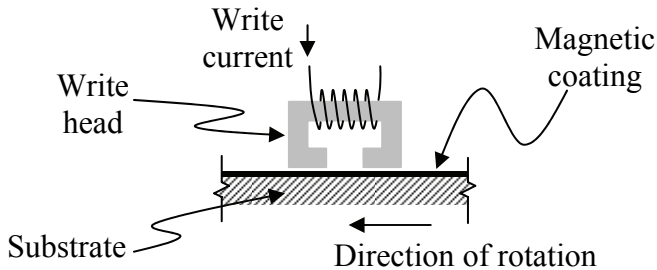


Figure 13-2 Configuration of a Hard Drive Write Head

It is possible to use the same head to read data back from the disk. If a magnetized material *moves* past a coil of wire, it produces a small current. This is the same principle that allows the alternator in your car to produce electricity. The direction of the current generated by the disk's motion changes if the direction of the magnetization changes. In this way, the same coil that is used to write the data can be used to read it. Just like the alternator in your car though, if the disk is not spinning, no current is generated that can be used to read the data.

Newer hard drives use two heads, one for reading and one for writing. The newer read heads are made of a material that changes its resistance depending on the magnetic field that is passing under it. These changes in resistance affect a current that the hard drive controller is passing through the read head during the read operation. In this way, the hard drive controller can detect changes in the magnetic polarization of the material directly under the read head.

There is another characteristic of the read/write head that is important to the physical operation of the hard drive. As was stated earlier, the area that is polarized by the head is equal to the gap in the write head. To polarize a smaller area thereby increasing the data density, the gap must be made smaller. To do this, the distance between the head and the platter must be reduced. Current technology allows heads to "fly" at less than three micro inches above the platter surface.

When the magnetic material is deposited on a flexible substrate such as a floppy diskette or a cassette tape, the flex in the material makes it possible for the head to come in contact with the substrate without experiencing reliability problems. This is not true for hard disks. Since the platters are rigid and because the platters spin at thousands of rotations per minute, any contact that the head makes with the platter will result in magnetic material being scraped off. In addition, the heat from the friction will eventually cause the head to fail.

These two issues indicate that the read/write head should come as close to the platters as possible without touching. Originally, this was done by making the platter as flat as possible while mounting the head to a rigid arm. The gap would hopefully stay constant. Any defects or warpage in the platter, however, would cause the head to crash onto the platter resulting in damaged data.

A third type of head, the **Winchester head** or "flying head" is designed to float on a cushion of air that keeps it a fixed distance from the spinning platter. This is done by shaping the head into an airfoil that takes advantage of the air current generated by the spinning platter. This means that the head can operate much closer to the surface of the platter and avoid crashing even if there are imperfections.

13.2.2 Data Encoding

It might seem natural to use the two directions of magnetic polarization to represent ones and zeros. This is not the case, however. One reason for this is that the controllers detect the *changes* in magnetic direction, not the direction of the field itself. Second, large blocks of data that are all ones or all zeros would be difficult to read because eventually the controller might lose track or synchronization of where one bit ended and the next began.

The typical method for storing data to a platter involves setting up a clock to define the bit positions, and watching how the magnetic field changes with respect to that clock. Each period of the clock defines a single bit time, e.g., if a single bit takes 10 nanoseconds to pass under the read-write head when the platter is spinning, then a clock with a period of 10 nanoseconds, i.e., a frequency of $(10 \times 10^{-9})^{-1} = 100 \text{ MHz}$ is used to tell the controller when the next bit position is coming.

Originally, a magnetic field change at the beginning and middle of a bit time represented a logic one while a magnetic field change only at the beginning represented a logic zero. This method was referred to as **Frequency Modulation (FM)**. Figure 13-3 uses an example to show the

relationship between the bit-periods, the magnetic field changes, and the data stored using FM encoding.

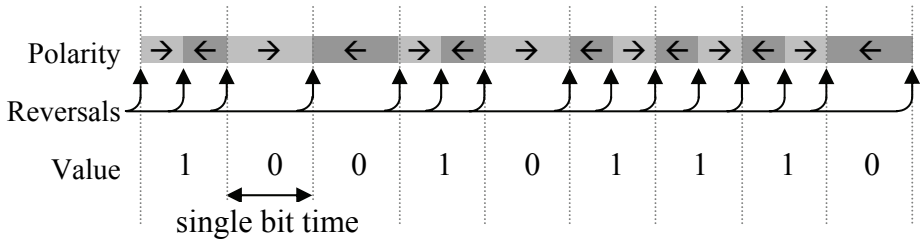


Figure 13-3 Sample FM Magnetic Encoding

To store a one using FM encoding, the polarization of the magnetic field must change twice within the space of a bit. This means that in order to store a single bit, FM encoding takes twice the width of the smallest magnetic field that can be written to the substrate. If the maximum number of polarity changes per bit could be reduced, more data could be stored to the same disk.

Modified Frequency Modulation (MFM) does this by changing the way in which the magnetic polarization represents a one or a zero. MFM defines a change in polarization in the middle of a bit time as a one and no change in the middle as a zero. If two or more zeros are placed next to each other, a change in polarization is made between each of the bit times. This is done to prevent a stream zeros from creating a long block of unidirectional polarization. Figure 13-4 uses an example to show the relationship between the bit-periods, the magnetic field changes, and the data stored using MFM encoding.

For MFM encoding, the longest period between polarity changes occurs for the bit sequence 1-0-1. In this case, the polarity changes are separated by two bit periods. The shortest period between polarity changes occurs when a one follows a one or a zero follows a zero. In these cases, the polarity changes are separated by a single bit period. This allows us to double the data density over FM encoding using the same magnetic surface and head configuration. The hard drive controller, however, must be able to handle the increased data rate.

Run Length Limited (RLL) encoding uses polarity changes to define sequences of bits rather than single bits. By equating different patterns of polarity changes to different sequences of ones and zeros, the density of bits stored to a platter can be further increased. There is a

science behind generating these sequences and their corresponding polarity changes. It is based on satisfying the following requirements:

- to ensure enough polarity changes to maintain bit synchronization;
- to ensure enough bit sequences are defined so that any sequence of ones and zeros can be handled; and
- to allow for the highest number of bits to be represented with the fewest number of polarity changes.

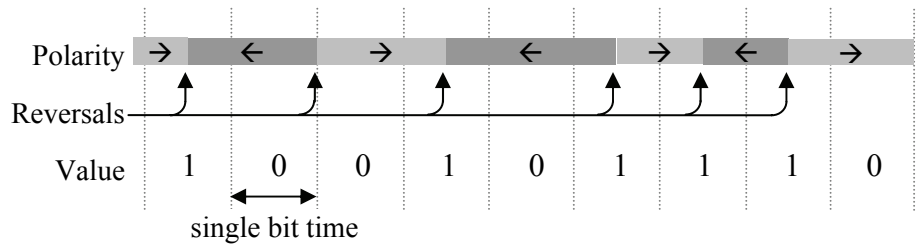


Figure 13-4 Sample MFM Magnetic Encoding

Figure 13-5 presents a sample set of RLL encoding polarity changes and the bit sequences that correspond to each of them. Any pattern of ones and zeros can be represented using this sample set of sequences.

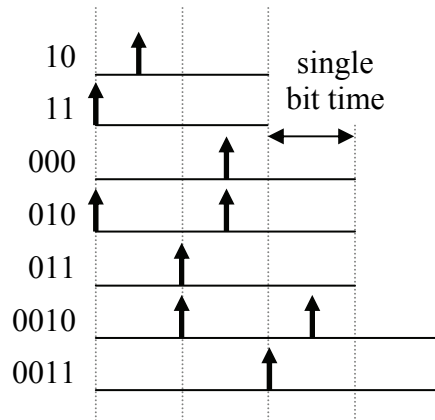


Figure 13-5 RLL Relation between Bit Patterns and Polarity Changes

Now the shortest period between polarity changes is one and a half bit periods producing a 50% increased density over MFM encoding. Figure 13-6 presents the same sample data with RLL encoding.

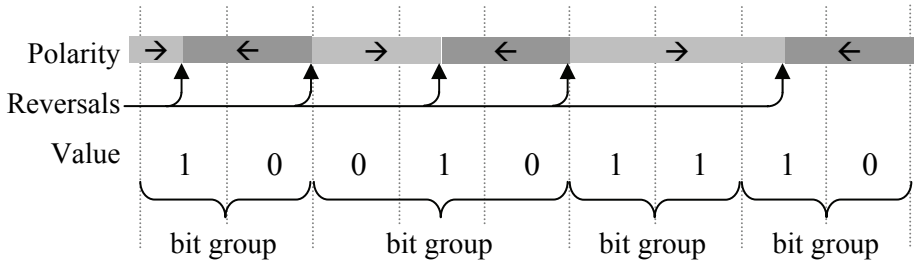


Figure 13-6 Sample RLL Magnetic Encoding

Improved encoding methods have been introduced since the development of RLL that use digital signal processing and other methods to realize better data densities. These methods include Partial Response, Maximum Likelihood (PRML) and Extended PRML (EPRML) encoding. A discussion of the details of these methods is beyond the scope of this text because it depends on a better understanding of sampling theory and electrical principles.

13.2.3 Hard Drive Access Time

There are a number of issues affecting the latency between a device requesting data and the hard drive responding with the data. Some of these issues depend on the current state of the system while others depend on the physical design of the drive and the amount of data being requested. There are four basic aspects to hard drive access time: queuing time, seek time, rotational latency, and transfer time.

After an initial request is made to a hard drive, the system must wait for the hard drive to become available. This is called queuing time. The hard drive may be busy serving another request or the bus or I/O channel that the hard drive uses may be busy serving another device that shares the link. In addition, the system's energy saving features may have powered down the drive meaning that an additional delay is incurred waiting for the drive to spin up.

The second aspect, seek time, is the amount of time it takes to get the read/write head from its current track to the desired track. Seek time is dependent on many things. First, it depends on the distance between the current and desired tracks. In addition, mechanical movement of any sort requires a ramping up before attaining maximum speed and a ramping down to avoid overshooting the desired target position. It is for these reasons that manufacturers publish a typical seek time.

Seek times have improved through the use of lighter components and better head positioning so that shorter seek distances are needed. As of this writing, the typical seek time for a hard drive is around 8 ms while higher performance drives might be as low as 4 ms. The heads used in CDROMs are heavier, and therefore, the seek time of a CDROM is longer than that of a hard drive. Older fixed head designs used multiple heads (one per track), each of which was stationary over its assigned track. In this case, the seek time was minimal, limited to the amount of time it took to electrically switch to the desired head.

Once the head has been positioned over the desired track, the drive must wait for the platters to rotate to the sector containing the requested data. This is called rotational latency. The worst case occurs when the start of the desired sector has just passed under the head when the drive begins looking for the data. This requires almost a full rotation of the platters before the drive can begin transferring the data. We can use the following calculation to determine the time required for a platter in a 7200 RPM drive to make a full rotation.

$$\frac{1 \text{ minute}}{7200 \text{ rotations}} \times \frac{60 \text{ seconds}}{1 \text{ minute}} = 8.3 \text{ ms per rotation}$$

If we make the assumption that on average the desired sector will be one half of a rotation away from the current position, then the average rotational latency should be half the time it takes for a full rotation. This means that for a 7200 RPM drive, the estimated rotational latency should be about 4.2 milliseconds.

Queuing time, seek time, and rotational latency are somewhat random in nature. Transfer time, however, is more predictable. Transfer time is the time it takes to send the requested data from the hard drive to the requesting device. Theoretically, the maximum transfer time equals the amount of time it takes for the data to pass beneath the head. If there are N sectors per track, then the amount of time it takes to retrieve a single sector can be calculated as shown below.

$$\text{Theoretical transfer time for a sector} = (N \times \text{rotational speed})^{-1}$$

Figure 13-7 presents a graphical representation of seek time, rotational latency, and transfer time.

As an example, let's calculate the amount of time it would take to read a 1 Mbyte file from a 7200 RPM drive with a typical 8 ms seek time that has 500 sectors per track each of which contains 512 bytes.

Begin by determining how many sectors the file requires. Dividing 1 Mbyte (1×10^6 bytes) by 512 bytes/sector shows that 1954 sectors will be required. There are two things to note about this number. First, $1 \times 10^6 \div 512$ actually equals 1953.125, but since files are stored on sector boundaries, the last sector must be partially empty. Second, authorities such as NIST, IEEE, and IEC have recommend Mbyte to represent 10^6 instead of 2^{20} as is used in the section on memory.

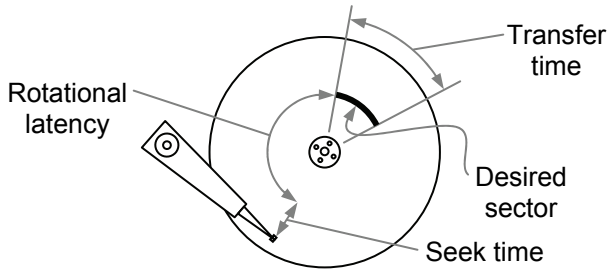


Figure 13-7 Components of Disk Access Time

Next, let's determine the transfer time for a single sector. If it takes 8.3 milliseconds for a complete revolution of the spindle, then during that time 500 sectors pass beneath the head. This means that a sector passes beneath the head every $8.3 \text{ ms/rotation} \div 500 \text{ sectors/rotation} = 16.7 \text{ microseconds/sector}$. This can also be calculated using the expression presented above.

$$\begin{aligned} \text{Transfer time for a sector} &= (500 \times 7200 \text{ RPM} \times 1/60 \text{ minutes/sec})^{-1} \\ &= 1/60000 \text{ seconds/sector} \\ &= 0.0000167 \text{ seconds/sector} \end{aligned}$$

Transferring 1954 sectors should take $1954 \times 16.7 = 32.6 \text{ ms}$. By adding the average seek time and rotational latency, we can predict the total access time not counting queuing time.

$$\text{Time to access 1 Mbyte file} = 8 \text{ ms} + 4.2 \text{ ms} + 32.6 \text{ ms} = 44.8 \text{ ms}$$

There are a few problems with this example. First, it will be discussed later how not all hard drives have a constant number of sectors per track. Therefore, the use of 500 sectors per track in the problem statement is not accurate. Second, this example assumes that

the sectors of the file have been stored in consecutive locations. This is referred to as *sequential access*. Unfortunately sequential access is not always possible. File system fragmentation where the blocks of a file end up scattered across dissociated sectors of the hard disk hurts disk performance because each discontinuity in the order of the sectors forces the drive to incur an additional seek time and rotational latency.

Lastly, with 500 sectors per track, the almost 2000 sectors of this file will need to be stored on at least four tracks. Each time a drive is forced to switch tracks, an additional seek time and rotational latency will be incurred. In the case of our example, 12.2 ms must be added for each of the three additional tracks bringing the access time to 81.4 ms.

If the sectors of the file are stored randomly across the tracks of the platters, individual seek times and rotational latencies will be required for each sector. This is referred to as *random access*. In this case, each sector will take $8 \text{ ms} + 4.2 \text{ ms} + 0.017 \text{ ms} = 12.217 \text{ ms}$ to retrieve. Multiplying this by 1954 sectors means that retrieving the 1 Mbyte file in random access will take 23.872 seconds.

13.2.4 Self-Monitoring, Analysis & Reporting Technology System

A hard drive crash rarely comes without a warning. The user may be unaware of any changes in their hard drive's operation preceding a mechanical failure, but there are changes. For example, if a hard drive's platters are taking longer to get up to full speed, it may be that the bearings are going bad. A hard drive that has been experiencing higher than normal operating temperatures may also be about to fail.

Newer drives now support a feature referred to as ***Self-Monitoring Analysis and Reporting Technology*** (SMART). SMART enabled drives can provide an alert to the computer's BIOS warning of a parameter that is functioning outside of its normal range. This usually results in a message to the user to replace the drive before it fails.

SMART attribute values are stored in the hard drive as integers in the range from 1 to 253. Lower values indicate worse conditions. Depending on the parameter and the manufacturer, different failure thresholds are set for each of the parameters. The parameters measured vary from drive to drive with each drive typically monitoring about twenty. The following is a sample of some types of measurements:

- ***Power On Hours***: This indicates the age of the drive.
- ***Power Cycle Count***: This also might be an indication of age.

- **Spin Up Time:** A longer spin up time may indicate a problem with the assembly that spins the platters.
- **Temperature:** Higher temperatures also might indicate a problem with the assembly that spins the platters.
- **Head Flying Height:** A reduction in the flying height of a Winchester head may indicate it is about to crash into the platters.

There are still unpredictable failures such as the failure of an IC or a failure caused by a catastrophic event such as a power surge, but now the user can be forewarned of most mechanical failures.

13.3 Organization of Data on a Hard Drive

The width of a hard drive's read/write head is much smaller than that of the platter. This means that there are a number of non-overlapping positions for the read/write head along the platter's radius. By allowing the movable read/write head to be positioned at intervals along the radius of the disk, information can be recorded to any of a number of concentric circles on the magnetic material. Each one of these circles is called a **track**. A typical hard drive disk contains thousands of tracks per inch (TPI) on a single side of a platter, each track being the width of the read/write head. Figure 13-8 shows how these tracks correspond to the movement and size of the read/write head.

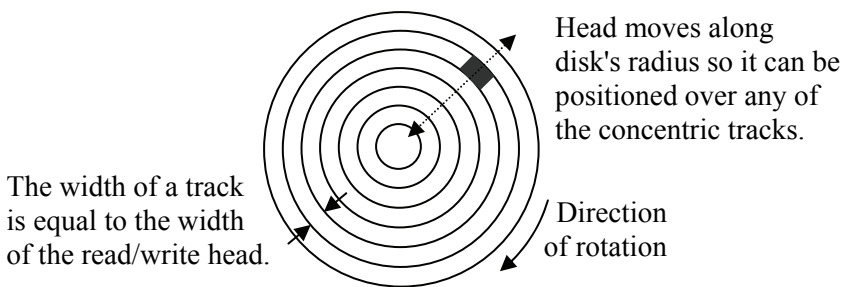


Figure 13-8 Relation between Read/Write Head and Tracks

A small gap called an **intertrack gap** is placed between the tracks to avoid interference from neighboring data. Reducing this gap allows for more data to be stored on a disk, but it also increases the risk of having data corrupted when data from an adjacent track bleeds over.

Each track is divided into sections of around 512 bytes apiece. These sections are called **sectors**. A platter may have sectors that are

fixed in size for the whole platter or they may have variable amounts of data depending on their location on the platter relative to the center of rotation. There are typically hundreds of sectors per track.

In addition to the gaps left between the tracks, gaps are also left between the sectors. These gaps allow for a physical separation between the blocks of data and are typically used to help the hard drive controller when reading from or writing to the disk. These gaps are called **intersector gaps**. Figure 13-9 shows the relationship of these gaps to the tracks and sectors.

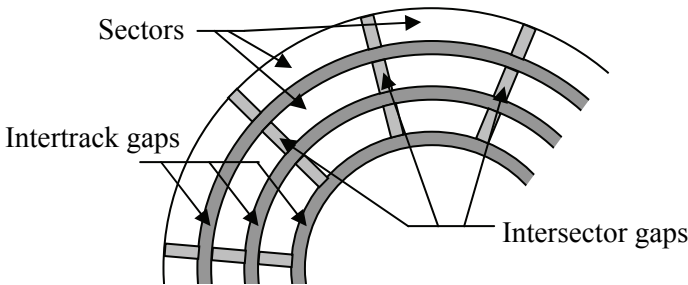


Figure 13-9 Organization of Hard Disk Platter

One way to increase the capacity of a hard drive is to increase the number of surfaces upon which the magnetic material is placed. The first way to do this is to place magnetic material on both sides of the platter. When this is done, a second read-write head must be placed on the opposite side of the platter to read the second magnetic surface. By using the same organization of sectors and tracks, this doubles the capacity of the hard drive.

A second method for increasing capacity is to mount multiple platters on a single spindle, the axis around which all of the platters rotate. Each additional magnetic surface adds to the capacity of the drive, and as with putting magnetic material on both sides of a single platter, all magnetic surfaces have the same organization of sectors and tracks, each sector lining up with the ones above it and below it. Every additional magnetic surface requires an additional read-write head.

All of the heads of a hard drive are locked together so that they are reading from the exact same location on each of their respective surfaces. Therefore, each track on each surface that is the same distance from the spindle can be treated as a unit because the hard drive controller is accessing them simultaneously. The set of all tracks, one

from each surface, that are equidistant from the spindle are referred to as a **cylinder**. This virtual entity is depicted in Figure 13-10.

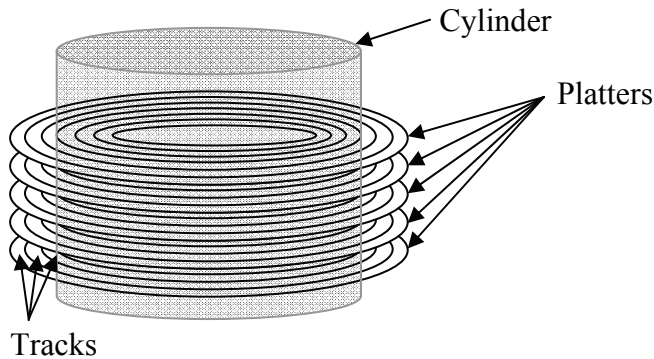


Figure 13-10 Illustration of a Hard Drive Cylinder

Using this information, we can develop a method for calculating the capacity of a hard drive. In general, the capacity of a hard drive equals the number of bytes per sector multiplied by the number of sectors per track multiplied by the number of cylinders multiplied by 2 if the platters have magnetic material on both sides and finally multiplied by the number of platters.

Figure 13-9 shows a platter that has the same number of sectors per track regardless of the radius of the track. From this figure, it can be seen that the physical size of a sector becomes smaller as its distance from the spindle is reduced. Since the number of bits per sector is constant, the size of a bit is also reduced.

Because the smallest allowable size for a bit is dictated by the size of the read-write head, the number of bits per track is limited by the number of bits that can fit on the smallest track, the one closest to the spindle. Because of this limitation, the outside tracks waste space when the bits become wider than is required by the head. Regardless of where the head is positioned, bits will pass under the head at a constant rate. This arrangement is called **constant angular velocity (CAV)**.

A better use of the space on the platter can be realized by letting the width of all bits be defined by the width of the read-write head regardless of the track position. This allows for more bits to be stored on the outer tracks. This tighter arrangement of bits can be seen in the comparison of CAV Figure 13-11a with the equal sized bits in Figure 13-11b.

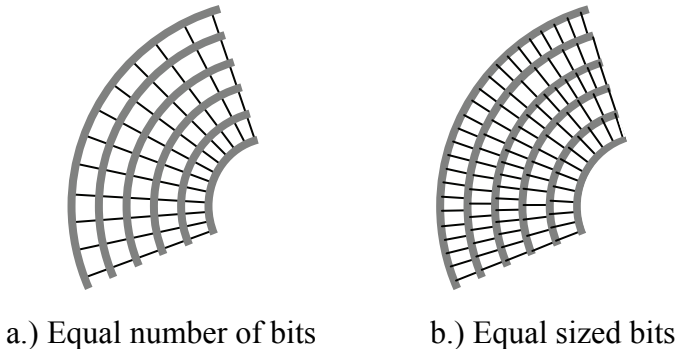


Figure 13-11 Equal Number of Bits per Track versus Equal Sized Bits

The problem with doing this is that as the read-write head moves to the outer tracks, the rate at which the bits pass under the head increases dramatically over that for the smallest track. This contrasts with the fixed number of bits per track which has the same data rate regardless of the position of the read-write head. This means that the hard drive with the equal sized bits requires a more complex controller.

Regardless of how the bits are arranged on the platters, the number of *bits per sector* must remain constant for all tracks. Since partial sectors are not allowed, additional bits cannot be added to tracks further from the spindle until a full sector's worth of bits can be added. This creates "zones" where groups of neighboring tracks have the same number of sectors, and therefore, the same number of bits. This method is called **Zone Bit Recording (ZBR)**. Figure 13-12 compares CAV with ZBR.

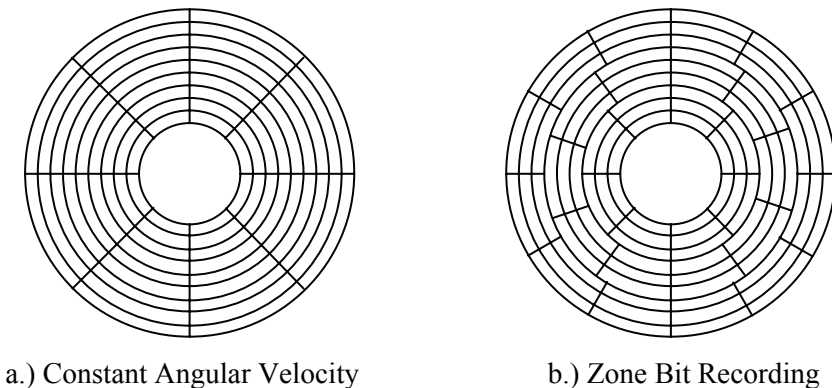


Figure 13-12 Comparison of Sector Organizations

This brings us to the next layer of data organization on a hard drive: formatting. Formatting is the process of setting or clearing bits on the platters in an effort to organize and locate the files that will be stored on the hard drive. The methods used by hard drives to organize data can also be found on other rotational storage media such as floppy disks and Zip® disks.

Every hard drive utilizes two types of formatting: low-level and O/S-level. **Low-level formatting (LLF)** depends on the mechanics of the hard drive and its controller. It divides the platters into usable subsections by defining the tracks and sectors. In addition to defining the divisions, it also stores digital information allowing the controller to properly access the data. This additional information includes:

- Synchronization fields that provide timing and positional information to the controller;
- ID information to allow the controller to identify which track and sector its heads are currently positioned over; and
- Error correcting codes to detect and correct errors in the data.

LLF is driven by the mechanical operation of the hard drive and is independent of the operating system that uses it. At one time, an LLF could be performed with a computer's BIOS, but because of the complexity of modern hard drives, LLF is now performed at the factory only. The closest function currently available is a utility called a **zero fill**. This function erases all data on a hard drive by filling the sectors with zeroes.

O/S-level formatting is used to create a file system so that the operating system can find and load data from the hard drive. This includes information such as the folder structure, file attributes, and on which sectors the files are stored.

There is a level of logical hard drive organization between low-level formatting and O/S level formatting called **partitioning**. Partitioning uses a table contained on the hard drive that defines individual, non-overlapping "logical drives," i.e., drives that look like separate drives themselves, but in actuality are all contained on a single set of platters.

One of the original uses of partitioning was to divide a hard drive into smaller logical units when the hard drives that manufacturers produced became too large for a BIOS or operating system to handle. For example, to install a 2 Gigabyte hard drive on a system where the

BIOS was only capable of seeing 512 Megabyte drives, the hard drive has to be logically divided into at least four drives.

Another application of partitioning is if different O/S-level formatting is needed on a single hard drive. If, for example, a user wishes to load both Windows and Linux on the same hard drive, three logical drives would be needed, one with a Windows format, one with a Linux native format, and one with a Linux swap format.

At one time, performance benefits could be realized with effective partitioning, but this is no longer true with advances in hard drive design.

13.4 Cache RAM

Even with increases in hard drive performance, it will never be practical to execute programs or access data directly from these mechanical devices. They are far too slow. Therefore, when the processor needs to access information, it is first loaded from the hard drive into main memory where the higher performance RAM allows fast access to the data. When the processor is finished with the data, the information can either be discarded or used to update the hard drive.

Because of its expense, the capacity of a computer's main memory falls short of that of its hard drive. This should not matter though. Not all of the data on a hard drive needs to be accessed all of the time by the processor. Only the currently active data or applications need to be in RAM. Additional performance improvements can be realized by taking this concept to another level.

Remember from our discussion in Chapter 12 that there are two main classifications of RAM: static RAM (SRAM) and dynamic RAM (DRAM). SRAM is faster, but that speed comes at a price: it has a lower density and it is more expensive. Since main memory needs to be quite large and inexpensive, it is implemented with DRAM.

Could, however, the same relation that exists between main memory and a hard drive be realized between a small block of SRAM and a large main memory implemented in DRAM? Main memory improves the performance of the system by loading only the information that is currently in use from the hard drive. If a method could be developed where the code that is in immediate use could be stored in a small, fast SRAM while code that is not quite as active is left in the main memory, the system's performance could be improved again.

Due to the nature of programming, instructions that are executed within a short period of time tend to be clustered together. This is due

primarily to the basic constructs of programming such as loops and subroutines that make it so that when one instruction is executed, the chances of it or its surrounding instructions being executed again in the near future are very good. Over a short period of time, a cluster of instructions may execute over and over again. This is referred to as the ***principle of locality***. Data also behaves according to this principle due to the fact that related data is often defined in consecutive locations.

To take advantage of this principle, a small, fast SRAM is placed between the processor and main memory to hold the most recently used code and data under the assumption that they will most likely be used again soon. This small, fast SRAM is called a ***RAM cache***.



Figure 13-13 Cache Placement between Main Memory and Processor

The reason the SRAM of the cache needs to be small is that larger address decoder circuits are slower than small address decoder circuits. The larger the memory is, the more complex the address decoder circuit. The more complex the address decoder circuit is, the longer it takes to select a memory location based on the address it received. Therefore, making a memory smaller makes it faster.

It is possible to take this concept a step further by placing an even smaller SRAM between the cache and the processor thereby creating two levels of cache. This new cache is typically contained inside of the processor. By placing the new cache inside the processor, the wires that connect the two become very short, and the interface circuitry becomes more closely integrated with that of the processor. Both of these conditions along with the smaller decoder circuit result in even faster data access. When two caches are present, the one inside the processor is referred to as a level 1 or ***L1 cache*** while the one between the L1 cache and memory is referred to as a level 2 or ***L2 cache***.



Figure 13-14 L1 and L2 Cache Placement

The **split cache** is another cache system that requires two caches. In this case, a processor will use one cache to store code and a second cache to store data. Typically, this is to support an advanced type of processor architecture such as pipelining where the mechanisms that the processor uses to handle code are so distinct from those used for data that it does not make sense to put both types of information into the same cache.

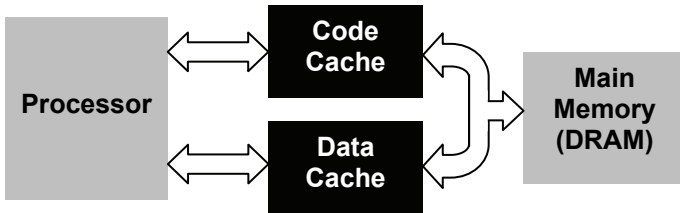


Figure 13-15 Split Cache Organization

13.4.1 Cache Organization

The success of caches is due primarily to the principle of locality. This suggests that when one data item is loaded into a cache, the items close to it in memory should be loaded too. For example, if a program enters a loop, most of the instructions that make up that loop will be executed multiple times. Therefore, when the first instruction of a loop is loaded into the cache, time will be saved if its neighboring instructions are loaded at the same time. That way the processor will not have to go back to main memory for subsequent instructions.

Because of this, caches are typically organized so that when one piece of data or code is loaded, the **block** of neighboring items is loaded too. Each block loaded into the cache is identified with a number called a **tag** that can be used to determine the original addresses of the data in main memory. This way, when the processor is looking for a piece of data or code (hereafter referred to as a word), it only needs to look at the tags to see if the word is contained in the cache.

The each block of words and its corresponding tag are combined in the cache to form a **line**. The lines are organized into a table much like that shown in Figure 13-16. It is important to note that when a word from within a block of words is needed from main memory, the whole block is moved into one of the lines of the cache along with the tag used to identify from where it came.

13.4.2 Dividing Memory into Blocks

Main memory stores all of its words in sequential addresses. The cache, however, has no sequential order. Therefore, it is the addressing scheme of main memory that is used to define the blocks of words and the method for locating them. The definition of blocks in main memory is logical only; it has no effect on how the words are stored.

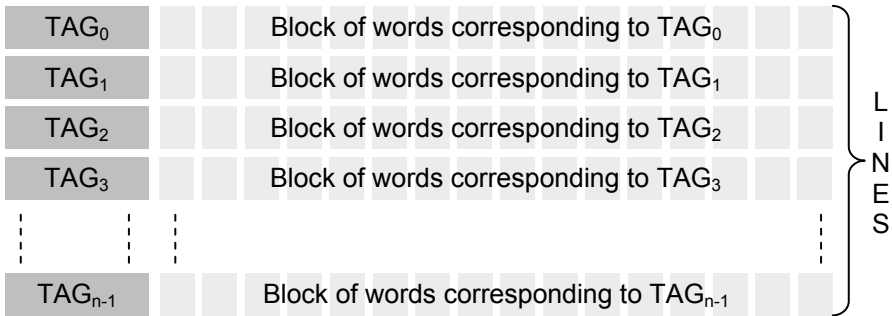


Figure 13-16 Organization of Cache into Lines

The full main memory address defines a specific memory location within memory. For example, a unique twenty-bit address such as $3E9D1_{16} = 0011\ 1110\ 1001\ 1101\ 0001_2$ points to exactly one memory location within a 1 Meg memory space.

If we "hide" the last bit of the address, i.e., that bit could be a one or a zero, then the resulting address could refer to one of two possible locations, $3E9D1_{16}$ ($0011\ 1110\ 1001\ 1101\ 0001_2$) or $3E9D0_{16}$ ($0011\ 1110\ 1001\ 1101\ 0000_2$). If we hide the last two bits, then the last two bits could be 00_2 , 01_2 , 10_2 , or 11_2 . Therefore, the address could be referring to one of the following four possible sequential locations:

$$\begin{aligned} 3E9D0_{16} &= 0011\ 1110\ 1001\ 1101\ 0000_2 \\ 3E9D1_{16} &= 0011\ 1110\ 1001\ 1101\ 0001_2 \\ 3E9D2_{16} &= 0011\ 1110\ 1001\ 1101\ 0010_2 \\ 3E9D3_{16} &= 0011\ 1110\ 1001\ 1101\ 0011_2 \end{aligned}$$

This is how a block is defined. By removing a small group of bits at the end of an address, the resulting identifier points to a group of memory locations rather than a specific address. Every additional bit

that is removed doubles the size of the group. This group of memory locations is what is referred to as a block.

The number of words in a block is defined by the number of bits removed from the end of the address to create the block identifier. For example, when one bit is removed, a block contains two memory locations. When two bits are removed, a block contains four memory locations. In the end, the size of a block, k , is defined by:

$$k = 2^w \tag{13.1}$$

where w represents the number of bits "removed". Figure 13-17 shows an example of a 1 Meg memory space divided into four word blocks.

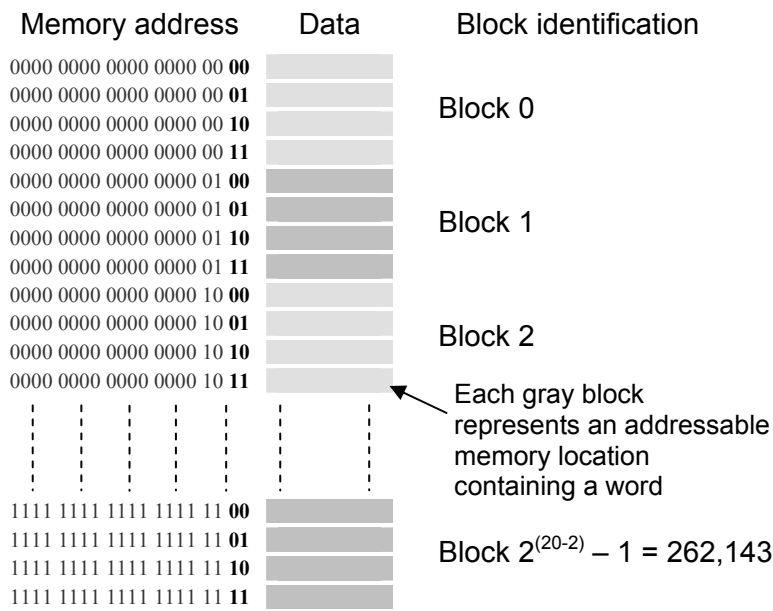


Figure 13-17 Division of Memory into Blocks

To organize memory into blocks, a memory address is divided into two logical sets of bits, one to represent the block number and one to identify the word's position or offset within the block. The memory address for the example in Figure 13-17 uses the most significant 18 bits to identify the block and the last two bits to identify a word's position within the block. Figure 13-18 presents this division using the address 10100101011010010110_2 ($A5696_{16}$).

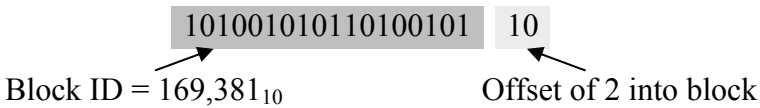


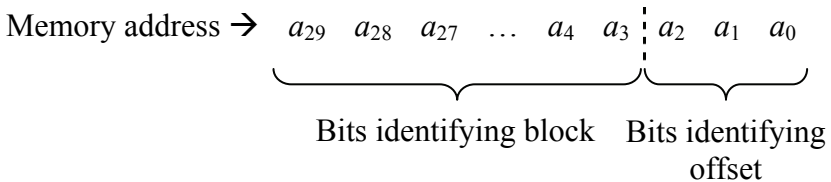
Figure 13-18 Organization of Address Identifying Block and Offset

Example

How many blocks of 8 words are there in a 1 Gig memory space?

Solution

Eight words require three bits to uniquely identify their position within a block. Therefore, the last three bits of the address represent the word's offset into the block. Since a 1 Gig (2^{30}) address space uses 30 address lines, there are $30 - 3 = 27$ remaining bits in the address. These bits are used to identify the block. Below is a diagram of the logical organization of the address.



13.4.3 Cache Operation

When the processor needs a word from memory, it first checks the cache. The circuits used to access the same word from main memory may be activated simultaneously so no time is lost in case the data is not available in the cache, a condition known as a **miss**. If the search of the cache is successful, then the processor will use the cache's word and disregard the results from main memory. This is referred to as a **hit**.

In the case of a miss, the entire block containing the requested word is loaded into a line of the cache, and the word is sent to the processor. Depending on the design of the cache/processor interface, the word is either loaded into the cache first and then delivered to the processor or it is loaded into the cache and sent to the processor at the same time. In the first case, the cache is in control of the memory interface and lies between memory and the processor. In the second case, the cache acts like an additional memory on the same bus with the main memory.

13.4.4 Cache Characteristics

The cache system used by a processor is defined by six traits:

- the size of the cache;
- the size of a block, which when combined with the size of the cache defines the number of lines;
- the number of caches (i.e., multiple levels or a split cache);
- the mapping function (the link between a block's address in memory and its location in the cache);
- the replacement algorithm (the method used to figure out which block to remove from the cache in order to free up a line); and
- the write policy (how the processor writes data to the cache so that main memory eventually gets updated).

As far as the size of a cache is concerned, designers need to perform a balancing act to determine the best size cache for a system. The larger a cache is, the more likely it is that the processor will find the word it needs in the cache. The problem is that as a cache gets larger, the address decoding circuits also get larger and therefore slower. In addition, more complicated logic is required to search a cache because of the seemingly random way that the blocks are stored in it. Larger caches are also more expensive.

There is also a relationship between size of a block and the performance of the cache. As the block size goes up, the possibility of getting a hit when looking for data could go up due to more words being available within a region of active code. For a fixed cache size, however, as the block size increases, the number of blocks that can be stored in a cache goes down thereby potentially reducing the number of hits. A typical size of a block is fairly low, between 4 and 8 words.

13.4.5 Cache Mapping Functions

There are three main methods used to map a line in the cache to an address in memory so that the processor can quickly find a word: direct mapping, full associative mapping, and set associative mapping. Let's begin with *direct mapping*.

Assume main memory is divided up into n blocks and the cache has room to contain exactly m blocks. Because of the nature of the cache, m is much smaller than n . If we divide m into n , we should get an integer

which represents the number of times that the main memory could fill the cache with different blocks from its contents.

For example, if main memory is 128 Meg (2^{27}) and a block size is four words (2^2), then main memory contains $n = 2^{27-2} = 2^{25}$ blocks. If the cache for this system can hold 256 K (2^{18}) words, then $m = 2^{18-2} = 2^{16}$ blocks. Therefore, the main memory could fill the cache $n/m = 2^{25}/2^{16} = 2^{25-16} = 2^9 = 512$ times.

Another way of putting it is this: the memory is much larger than a cache, so each line in the cache is responsible for storing one of many blocks from main memory. In the case of our example above, each line of the cache is responsible for storing one of 512 different blocks from main memory at any one time.

Direct mapping is a method used to assign each memory block in main memory to a specific line in the cache. If a line is already filled with a memory block when a new block needs to be loaded, the old block is discarded from the cache. Figure 13-19 shows how multiple blocks from our example are mapped to each line in the cache.

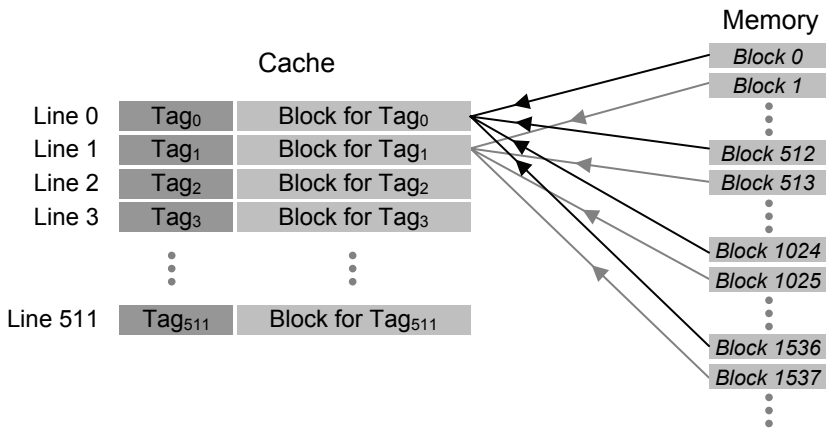


Figure 13-19 Direct Mapping of Main Memory to Cache

As with locating a word within a block, bits are taken from the main memory address to uniquely define the line in the cache where a block should be stored. For example, if a cache has $2^9 = 512$ lines, then a line would need 9 bits to be uniquely identified. Therefore, the nine bits of the address immediately to the left of the word identification bits would identify the line in the cache where the block is to be stored. The bits of

the address not used for the word offset or the cache line would be used for the tag. Figure 13-20 presents this partitioning of the bits.

Once the block is stored in the line of the cache, the tag is copied to the tag location of the line. From the cache line number, the tag, and the word position within the block, the original address of the word can be reconstructed.

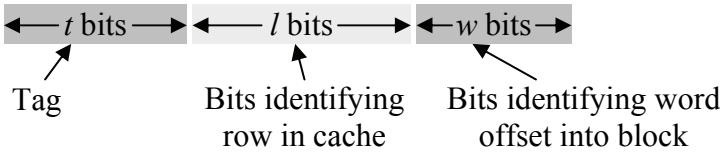


Figure 13-20 Direct Mapping Partitioning of Memory Address

In a nutshell, direct mapping breaks an address into three parts: t tag bits, l line bits, and w word bits. The word bits are the least significant bits identifying the specific word within a block of memory. The line bits are the next least significant identifying in which line of the cache the block will be stored. The remaining bits are stored with the block as the tag identifying where the block is located in main memory.

Example

Assume a cache system has been designed such that each block contains 4 words and the cache has 1024 lines, i.e., the cache can store up to 1024 blocks. What line of the cache is supposed to hold the block that contains the word from the twenty-bit address $3A456_{16}$? In addition, what is the tag number that will be stored with the block?

Solution

Start by dividing the address into its word id, line id, and tag bits. Since $4=2^2$, then the two least significant bits identify the word, i.e., $w = 2$. Since the cache has $1024=2^{10}$ lines, then the next 10 bits identify the line number where the data is supposed to be stored in the cache, i.e., $l = 10$. The remaining $t = 20 - w - l = 8$ bits are the tag bits. This partitions the address $3A456_{16} = 00111010010001010110_2$ as follows:

00111010	0100010101	10
tag bits	line id bits	word id bits

Therefore, the block from address $3A454_{16}$ to $3A457_{16}$ will be stored in line $0100010101_2 = 277_{10}$ of the cache with the tag 00111010_2 .

Example

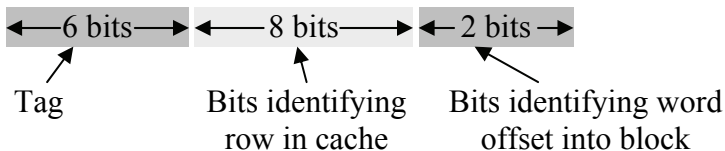
The first 10 lines of a 256 line cache are shown in the table below. Identify the address of the data that is shaded ($D8_{16}$). For this cache, a block contains 4 words. The tags are given in binary in the table.

Line #	Tag	word 00	word 01	word 10	word 11
0	110101	12	34	56	78
1	010101	54	32	6A	D3
2	000111	29	8C	ED	F3
3	001100	33	A2	2C	C8
4	110011	9A	BC	D8	F0
5	001101	33	44	55	66
6	010100	92	84	76	68
7	000100	FE	ED	00	ED
8	100000	00	11	22	33
9	101000	99	88	77	66

Solution

Start by finding the number of bits that represent each part of the address, i.e., the word id, the line id, and the tag. From the table, we can see that 2 bits represent the positions of each of the four words in a block and that 6 bits are used to represent the tag.

Since the cache has $256=2^8$ lines, then the line number in the cache is represented with 8 bits, and the address is partitioned as follows:



The shaded cell in the table has a tag number of 110011_2 . The line number is 4, which in 8 bit binary is 00000100_2 . Last of all, the word is in the third column which means that it is the 10_2 word within the block. (Remember to start counting from 00_2 .) Putting the tag, line id, and word id bits together gives us:

110011	00000100	10
tag bits	line id bits	word id bits

Therefore, the address that the shaded cell containing $D8_{16}$ came from is $1100\ 1100\ 0001\ 0010_2 = CC12_{16}$.

Example

Using the table from the previous example, determine if the data stored in main memory at address $101C_{16}$ is contained in this cache, and if it is, retrieve the data.

Solution

Converting $101C_{16}$ to binary gives us $0001\ 0000\ 0001\ 1100_2$. By using the breakdown of bits for the tag, line id, and word id, the binary value can be divided into its components.

000100	00000111	00
tag bits	line id bits	word id bits

From this we see that the line in the cache where this data should be stored is $00000111_2 = 7_{10}$. The tag currently stored in this line is 000100_2 which equals the tag from the above partitioned address. Therefore, the data from main memory address $101C_{16}$ is stored in this cache. If the stored tag did not match the tag pulled from the address, we would have known that the cache did not contain our address.

Lastly, we can find the data by looking at the offset 00_2 into the block at line 7. This gives us the value FE_{16} .

Example

Using the same table from the previous two examples, determine if the data from address 9827_{16} is in the cache.

Solution

Converting the hexadecimal address 9827_{16} to binary gives us $9827_{16} = 1001\ 1000\ 0010\ 0111_2$. By using the breakdown of bits for the tag, line id, and word id, we can divide this value into its components.

100110	00001001	11
tag bits	line id bits	word id bits

From this we see that the tag is 100110_2 , the line number is $00001001_2 = 9_{10}$, and the word offset into the block is 11_2 . Looking at line number 9 we see that the tag stored there equals 101000_2 . Since this does not equal 100110_2 , the data from that address is not contained in this cache, and we will have to get it from the main memory.

Fully associative mapping does not use line numbers at all. It divides the main memory address into two parts: the word id and a tag. In order to see if a block is stored in memory, the tag is pulled from the memory address and a search is performed through all of the lines of the cache to see if the block is present.

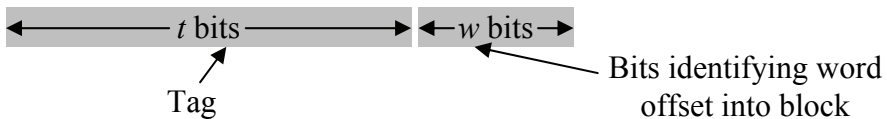


Figure 13-21 Fully Associative Partitioning of Memory Address

This method of searching for a block within a cache sounds like it might be a slow process, but it is not. Each line of the cache contains its own compare circuitry that is able to discern in an instant whether or not the block is contained at that line. With all of the lines performing this compare in parallel, the correct line is identified quickly.

This mapping algorithm is meant to solve a problem that occurs with direct mapping where two active blocks of memory map to the same line of the cache. When this happens, neither block of memory is allowed to stay in the cache long before it is replaced by the competing block. This results in a condition referred to as thrashing where a line in the cache goes back and forth between two or more blocks, usually replacing a block before the processor was through with it. Thrashing is avoided by allowing a block of memory to map to any line of the cache.

This benefit has a price, however. When a fully associative cache is full and the processor needs to load a new block from memory, a decision has to be made regarding which of the existing blocks is to be discarded. The selection method, known as a **replacement algorithm**,

should have a goal of replacing the block least likely to be needed by the processor in the near future.

There are numerous replacement algorithms, no one of which is significantly better than the others. In an effort to realize the fastest operation, each of these algorithms is implemented in hardware.

- **Least Recently Used (LRU)** – This method replaces the block that hasn't been read by the processor in the longest period of time.
- **First In First Out (FIFO)** – This method replaces the block that has been in cache the longest.
- **Least Frequently Used (LFU)** – This method replaces the block which has had fewest hits since being loaded into the cache.
- **Random** – This method randomly selects a block to be replaced. It has only slightly lower performance than LRU, FIFO, or LFU.

Example

The table below represents five lines from a cache that uses fully associative mapping with a block size of eight. Identify the address of the shaded data ($C9_{16}$).

Tag	Word id bits (in binary)							
	000	001	010	011	100	101	110	111
0110110110010 ₂	16	36	66	28	A1	3B	D6	78
0100011010101 ₂	54	C9	6A	63	54	32	00	D3
0001000111011 ₂	29	8C	ED	FD	29	54	12	F3
0001111001100 ₂	39	FA	B5	C1	33	9E	33	C8
1001100101101 ₂	23	4C	D2	40	6A	76	A3	F0

Solution

The tag for $C9_{16}$ is 0100011010101₂. When combining this with the word id of 001₂, the address in main memory from which $C9_{16}$ was retrieved is 0100011010101001₂ = $46A9_{16}$.

Example

Is the data from memory address $1E65_{16}$ contained in the table from the previous example?

Solution

For this cache, the last three bits identify the word and the rest of the bits act as the tag. Since $1E65_{16} = 0001111001100101_2$, then 101_2 is the word id and 0001111001100_2 is the tag. Scanning the rows shows that the fourth row contains this tag, and therefore the table contains the data in which we are interested. The word identified by 101_2 is $9E_{16}$.

The last mapping algorithm presented here is **set associative mapping**. Set associative mapping combines direct mapping with fully associative mapping by grouping together lines of a cache into sets. The sets are identified using a direct mapping scheme while the lines within each set are treated like a miniature fully associative cache where any block that is to be stored in the set can be stored to any line within the set. Figure 13-22 represents this arrangement with a sample cache that uses four lines to a set.

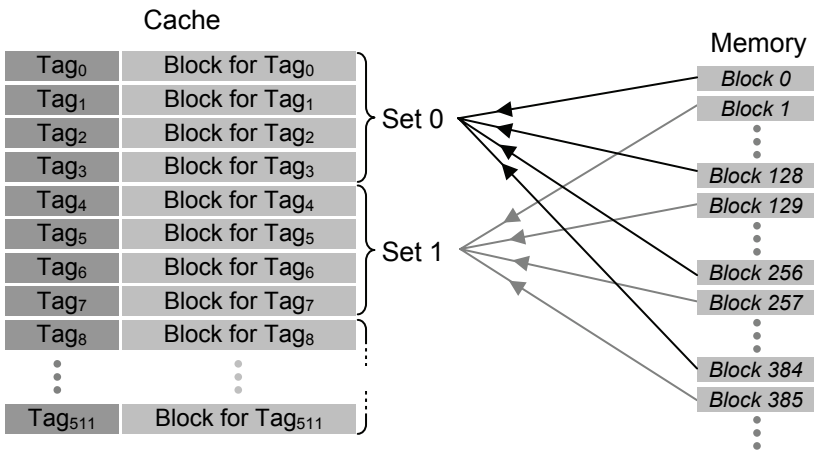


Figure 13-22 Set Associative Mapping of Main Memory to Cache

A set associative cache that has k lines per set is referred to as a k -way set associative cache. Since the mapping scheme uses the memory address just as direct mapping does, the number of lines contained in a set must be equal to an integer power of two, e.g., two, four, eight, sixteen, and so on.

Let's use an example to further explain set associative mapping. Assume that a system uses a cache with $2^9 = 512$ lines, a block of memory contains $2^3 = 8$ words, and the full memory space contains $2^{30} = 1$ Gig words. In a direct mapping scheme, this would leave $30 - 9 - 3$

= 18 bits for the tag. Note that the direct mapping method is equivalent to the set associative method where the set size is equal to one line.

By going from direct mapping to set associative with a set size of two lines per set, the number of sets is equal to half the number of lines. In the case of the cache with 512 lines, that would give us 256 sets of two lines each which would require eight bits from the memory address to identify the set. This would leave $30 - 8 - 3 = 19$ bits for the tag. By going to four lines per set, the number of sets is reduced to 128 sets requiring seven bits to identify the set and twenty bits for the tag.

Figure 13-23 shows how each time the number of lines per set in the example is doubled, the number of bits used to identify the set is reduced by one thereby increasing the number of tag bits by one.

Tag bits	Set ID bits	Word ID bits	
18 bits	9 bits	3 bits	Direct mapping (1 line/set)
19 bits	8 bits	3 bits	2-way set associative (2^1 lines/set)
20 bits	7 bits	3 bits	4-way set associative (2^2 lines/set)
21 bits	6 bits	3 bits	8-way set associative (2^3 lines/set)
⋮	⋮	⋮	
25 bits	2 bits	3 bits	128-way set associative (2^7 lines/set)
26 bits	1 bit	3 bits	256-way set associative (2^8 lines/set)
27 bits		3 bits	Fully associative (1 big set)

Figure 13-23 Effect of Cache Set Size on Address Partitioning

When a block from memory needs to be stored to a set already filled with other blocks, one of the replacement algorithms described for fully associative mapping is used. For a 2-way set associative cache, one of the easiest replacement algorithms to implement is the least recently used method. Each set contains a single bit identifying which of the two lines was used last. If the bit contains a zero, then the first line was used last and the second line is the one that should be replaced. If the bit contains a one, the first line should be replaced.

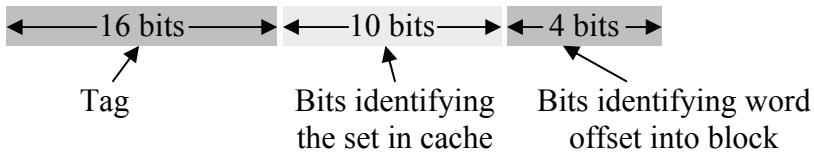
Example

Identify the set number where the block containing the address 29ABCDE8₁₆ will be stored. In addition, identify the tag and the lower

and upper addresses of the block. Assume the cache is a 4-way set associative cache with 4K lines, each block containing 16 words, with the main memory of size 1 Gig memory space.

Solution

First, we need to identify the partitioning of the bits in the memory address. A 1 Gig memory space requires 30 address lines. Four of those address lines will be used to identify one out of the 16 words within the block. Since the cache is a 4-way set associative cache, the number of sets equals 4K lines divided by four lines per set, i.e., $1K = 2^{10}$. Therefore, ten address lines will be needed to identify the set. The figure below represents the partitioning of the 30 bit address.



Converting $29ABCDE8_{16}$ to a 30-bit binary value gives us $00101001101010111100110111101000_2$. The first sixteen bits, 001010011010101111_2 , represent the tag. The next ten bits, $0011011110_2 = 222_{10}$, represent the set in which the block will be stored. The last four bits, $1000_2 = 8_{10}$, represent the word position within the block. The lowest address will be the one where word 0000_2 is stored and the highest address will be the one where the word 1111_2 is stored. Replacing the last four bits of $29ABCDE8_{16}$ with 0000_2 gives us a low address of $29ABCDE0_{16}$ while replacing them with 1111_2 gives us a high address of $29ABCDEF_{16}$.

13.4.6 Cache Write Policy

The last characteristic of caches discussed here is the **cache write policy**, i.e., the method used to determine how memory is updated when the cache is written to by the processor. There are a number of issues surrounding this topic, most of which apply to multi-processor systems which are beyond the scope of this text. Basically, when the cache is written to, the corresponding location in memory becomes obsolete.

One method for resolving this is to have the cache update memory every time the processor writes to the cache. This is called a **write through policy**. The problem with this is that it creates a great deal of

memory bus traffic, some of which might be unnecessary if the same memory location is being updated often. This policy also slows the write process since both memories are being updated at the same time.

A second method is to update main memory only when a line of the cache that has been modified to is about to be replaced with a new block from memory. This is called a ***write back policy***. This method runs into problems when more than one processor such as an I/O processor is sharing memory. Solutions include making shared memory non-cacheable, adding an additional processor to watch all of the caches and make appropriate updates when a write occurs, or giving all of the processors the ability to check other caches for updates.

13.5 Registers

At the top of the memory hierarchy is a set of memory cells called ***registers***. A register is a group of latches that have been combined in order to perform a special purpose. This group of latches may be used to store an integer, store an address pointing to memory, configure an I/O device, or indicate the status of a process. Whatever the purpose of the register is, all of the bits are treated as a unit.

Registers are contained inside the processor and are integrated with the circuitry used to perform the processor's internal operations. This integration places registers within millionths of a meter of the action resulting in very quick access times. In addition, the typical processor contains fewer than a hundred registers making decoding very simple and very fast. These two features combine to make registers by far the fastest memory unit in the memory hierarchy.

Because of the integral part they play in computer architecture, the details and applications of registers are presented in Chapter 15.

13.6 What's Next?

This chapter presented the system of memory components that serve the processor. This system will be revisited in Chapter 15 where a detailed examination of the organization of the components inside of the processor will be introduced.

Chapter 14 presents another important component supporting the processor: serial communications. Up to this point, our discussion of the processor has used the parallel bus to transfer data. The parallel bus has some drawbacks though. These include higher expense, lack of adaptability, and lower reliability. Chapter 14 presents the basic