Computer Organization and Assembly Language: Embedded Systems Perspective

Ben Lee
Oregon State University
School of Electrical Engineering and Computer Science

Contents

Contents **2**

1 Introduction **17**
1.1 The Role of Computers in Modern Society 17
1.2 Spectrum of computers and their processors 18

2 Assembly Language Fundamentals **21**
2.1 High-Level Language to Machine Language 21
2.2 How Do We Speak the Language of the Machine 22
2.3 Instruction Set Architecture 23
2.3.1 Operations in the ISA 24
2.3.2 Number of Operands per Instruction 27
2.3.3 Operand Locations and How They are Specified . . . 29
2.3.4 Operand Type and Size 31
2.4 Instruction Format 31
2.5 A pseudo-ISA 33

3 Computer Organization Fundamentals **37**
3.1 Memory 38
3.2 Microoperations 39
3.3 Organization of the pseudo-CPU 43
3.3.1 Instruction Format 44
3.3.2 Registers 45
3.3.3 Bus-Register Connections 45
3.3.4 Instruction Cycle 47
3.3.5 Extensions to the pseudo-ISA/CPU 58

4 AVR Assembly Language **67**
4.1 Why Microcontrollers? 67
4.2 General Characteristics 68

4.2.1 Program and Data Memories 69
4.3 Registers and Memory 70
4.3.1 Registers 70
4.4 Addressing Modes 73
4.5 Instructions 77
4.5.1 Data Transfer Instructions 78
4.5.2 Arithmetic and Logic Instructions 83
4.5.3 Control Transfer Instructions 83
4.5.4 I/O Instructions 85
4.5.5 Bit and Bit-test Instructions 86
4.6 Assembly to Machine Instruction Mapping 87
4.7 Assembler Directives 92
4.8 Expressions 95
4.8.1 Operands 96
4.8.2 Operators 96
4.8.3 Functions 96
4.9 Assembly Coding Techniques 97
4.9.1 Code Structure 97
4.9.2 ATmega128 Definition File 99
4.10 Control Flow 99
4.10.1 IF Statement 100
4.10.2 IF-ELSE Statement 102
4.10.3 Loops 102
4.10.4 Functions and Subroutines 104
4.11 Anatomy of an Assembly Program 110

5 Input/Output 115
5.1 AVR I/O Capabilities 115
5.2 I/O Ports 115
5.3 Interrupts 126
5.3.1 AVR Interrupt Facility 127
5.4 Timers/Counters 138
5.4.1 TCNT0 and TCNT2 139
5.4.2 TCNT1 145
5.4.3 TCNT Interrupt Mask and Interrupt Flag Registers . 149
5.5 Analog-to-Digital Converter 149
5.6 Universal Synchronous Asynchronous Receiver/Transmitter . 150

6 Embedded C **151**
6.1 Basics of Embedded C 151
6.1.1 Comments 151
6.1.2 Identifiers 151
6.1.3 Reserve Words 152
6.1.4 Variables 152
6.1.5 Data Types 155
6.1.6 Constants 155
6.1.7 Preprocessors 156
6.1.8 Enumeration 162
6.1.9 Pointers 162
6.2 Operators 165
6.3 Control Statements 168
6.3.1 `if` Statements 168
6.3.2 `if/else` Statements 168
6.3.3 `if/else/else if` Statements 168
6.3.4 `while` Loop 168
6.3.5 `do/while` Loop 169
6.3.6 `do/while` Loop 169
6.3.7 `for` Loop 169
6.3.8 Conditional Statement 170
6.3.9 `switch/case` statement 170
6.4 Functions 170
6.4.1 Library Functions 171
6.4.2 Function Structure 171
6.4.3 Function Ordering 172
6.4.4 `#include` 173
6.4.5 Recursive Functions 173
6.5 Arrays and Structures 174
6.5.1 Arrays 174
6.5.2 Multidimensional Arrays 174
6.5.3 Structures 175
6.5.4 Nested Structures 176
6.6 Accessing I/O Registers 177
6.6.1 Bit-level access to I/O Registers 179
6.7 Accessing EEPROM 180
6.8 Using Interrupts 180

7 Digital Components **185**
7.1 Multiplexers . . . 186
7.2 Decoders . . . 188
7.3 Memory Elements . . . 190
7.3.1 Latches . . . 191
7.3.2 Flip-Flops . . . 193
7.3.3 Edge-Triggered D Flip-Flop with Enable . . . 196
7.4 Registers . . . 197
7.4.1 n-bit Register . . . 197
7.4.2 Shift Registers . . . 197
7.5 Memory . . . 198
7.5.1 Static RAM (SRAM) . . . 201
7.5.2 Building Bigger and Wider Memories . . . 204
7.6 Register File . . . 206
7.7 Arithmetic and Logic Unit and Address Adder . . . 207

8 Microarchitecture Design **209**
8.1 Microarchitecture . . . 209
8.2 Instruction Format . . . 210
8.3 Basic Components in a Datapath . . . 212
8.3.1 Registers . . . 212
8.3.2 Program and Data Memories . . . 214
8.3.3 Sign-Extension and Zero-Fill Units . . . 214
8.3.4 ALU . . . 215
8.3.5 Alignment Unit . . . 217
8.3.6 Register File . . . 217
8.3.7 Address Adder . . . 218
8.3.8 Multiplexers . . . 220
8.4 Multi-cycle Implementation . . . 221
8.4.1 Fetch Stage . . . 222
8.4.2 Execute Stage . . . 223
8.5 Execution of More Complex Instructions . . . 235
8.6 Control Unit Design . . . 237
8.6.1 Opcode Encoding . . . 237
8.6.2 Control and Alignment Unit . . . 239
8.6.3 Sequence Control . . . 248
8.6.4 Register Address Logic . . . 252
8.7 Pipeline Implementation . . . 256

9 Arithmetic and Logic Unit 259
9.1 Number Systems . . . 259
9.1.1 Sign-Magnitude Representation . . . 260
9.1.2 1's-complement Representation . . . 262
9.1.3 2's-complement Representation . . . 264
9.2 Shift Operations . . . 266
9.2.1 Logical Shifts . . . 266
9.2.2 Arithmetic Shifts . . . 266
9.2.3 Circular Shifts . . . 267
9.3 Basic ALU Design . . . 267
9.3.1 Full Adder . . . 267
9.3.2 n-bit Ripple Carry Adder . . . 267
9.3.3 n-bit ALU . . . 270
9.3.4 Condition Codes . . . 275
9.3.5 Fast Adders . . . 276
9.4 Multiplication . . . 280
9.4.1 Add-and-Shift . . . 280
9.4.2 Booth Algorithm . . . 284
9.4.3 Modified Booth Algorithm (Bit-pair recoding) . . . 287
9.4.4 Array Multipliers . . . 289
9.4.5 Implementing Large Multiplication using Smaller Multiplications . . . 289
9.4.6 Partial Product Reduction (Reduction of Summands) 291
9.5 Division . . . 291
9.5.1 Restoring . . . 293
9.5.2 Non-Restoring . . . 294
9.6 Floating-Point Numbers . . . 296
9.6.1 IEEE 754 Floating-Point Standard . . . 296
9.6.2 Four Basic Operations . . . 299
9.6.3 Rounding . . . 302
9.6.4 Example: Fractional Division Mantissas . . . 304

A AVR Instruction Set Summary 307
A.1 Arithmetic and Logic Instructions . . . 307
A.2 Data Transfer Instructions . . . 308
A.3 Branch Instructions . . . 309
A.4 Bit and Bit-test Instructions . . . 310
A.5 MCU Control Instructions . . . 311
A.6 AVR Opcode Encoding . . . 312
A.7 AVR Opcodes . . . 313

B AVR Studio 4 **323**
B.1 Startup Tutorial 323
B.1.1 Installation 323
B.1.2 Project Creation 324
B.1.3 Project Simulation 325
B.2 Simulation Tips 326
B.2.1 Line-By-Line Debugging 326
B.2.2 Workspace Window 327
B.2.3 Memory Windows 328
B.3 Debugging Strategies 328

C Microcontroller Programming with PonyProg2000 **331**
C.1 Installing PonyProg2000 331
C.1.1 PonyProg2000 Setup 332
C.1.2 Setup the AVR Microcontroller Board 332
C.2 Uploading a Program 333

Index **336**

List of Figures

1.1 Desktop versus Embedded systems. 19

2.1 Simplified hierarchical view of software and hardware and the role of ISA. 23
2.2 An example instruction that refers to operands in both memory and register. 29
2.3 Indirect addressing. 30
2.4 Instruction format. 32
2.5 Instruction format of the hypothetical ISA. 33
2.6 A simple C program. 35
2.7 An equivalent assembly program for code in Figure 2.6. . . . 36

3.1 Computer Organization. 38
3.2 Simple data transfer between two n-bit registers. 40
3.3 n-bit data transfer between two different sized registers. . . . 40
3.4 Data transfer of high-byte of R2 to low-byte of R1. 41
3.5 Simultaneous data transfer of $R3$ to $R2$ and $R2$ to $R1$. 42
3.6 Simultaneous data transfer of $R2$ to $R1$ and $R1$ to $R2$. 42
3.7 Hypothetical CPU. 44
3.8 Instruction Format for the pseudo-CPU. 45
3.9 Bus-Register Connections. 46
3.10 Fetch Cycle. 48
3.10 Fetch Cycle (cont.). 49
3.11 Operation of `LDA x`. 50
3.12 Microoperations for `LDA x`. 52
3.13 Operation of `STA x`. 53
3.14 Microoperations for `STA x`. 54
3.15 Operation of `ADD x`. 55
3.16 Microoperations for `ADD x`. 56

3.17 Operation of `J x`. 57
3.18 Microoperation for `J x`. 57
3.19 Microoperation for `BNE x`. 58
3.20 The concept of a pointer. 59
3.21 The concept of indirection. 60
3.22 LDA indirect. 61
3.22 LDA indirect (cont.). 62
3.23 LDA indirect with pre-decrement. 63
3.24 Simple CPU decrement capability with MDR. 64
3.25 Pseudo-CPU with a temporary register. 65

4.1 The most popular 8-bit processor. Source: EETimes and Embedded Systems Design Magazine 2006 Embedded Market Survey 68
4.2 Some AVR-based products. 69
4.3 The block diagram of Atmega128. 70
4.4 AVR memory organization. 71
4.5 AVR GPRs. 72
4.6 Status Register. 73
4.7 Register Addressing mode. 74
4.8 Direct addressing modes. 75
4.9 Indirect addressing modes. 76
4.10 Indirect addressing modes with pre-decrement and post-increment. 78
4.11 Program memory constant addressing. 79
4.12 Program memory addressing modes. 79
4.13 Relative Program Memory constant addressing. 80
4.14 Machine instruction mapping for `ADD R15, R16`. 88
4.15 Machine instruction mapping for `LD R16, Y`. 88
4.16 Machine instruction mapping for `LDI R30, $F0`. 89
4.17 Machine instruction mapping for `LDD R4, Y+2`. 89
4.18 Machine instruction mapping for `IN R25, $16`. 90
4.19 Machine instruction mapping for `BREQ label`. 90
4.20 Stack manipulation for subroutine call and return. 91
4.21 Machine instruction mapping for `CALL label`. 92
4.22 Contents of the Program Memory. 112
4.23 `lpm` instruction. 113

5.1 I/O ports in AVR processor. 116
5.2 A single pin of a port. 117
5.3 Reading and writing to a port. 118

5.4 Connection of whiskers and wheels to PORTB and PORTE in Tekbot. 120
5.5 Code for Tekbot movement. 121
5.5 Code for Tekbot movement (cont.). 122
5.6 Triggering of a whisker. 124
5.7 The process of handling interrupts. 128
5.8 I-bit in the SREG. 129
5.9 I/O Pins. 129
5.10 Controlling interrupts. 131
5.11 An example code. 133
5.12 Controlling interrupts. 133
5.13 Interrupt-based code for Tekbot. 135
5.13 Interrupt-based code for Tekbot 136
5.14 Initializing interrupts. 137
5.15 Initializing the stack. 137
5.16 Block diaragram of Timer/Counter0. 139
5.17 Timing diagram of normal mode. 141
5.18 Example timing diagram in CTC mode. 143
5.19 Example timing diagram in PWM mode. 144
5.20 Time Counter Control Register 0. 144
5.21 Block diagram of Timer/Clock2. 147
5.22 Time Counter Control Register 1. 148
5.23 TIMSK and TIFR registers. 150

6.1 Program in Data Memory. 183

7.1 Digital components in the AVR microarchitecture. 186
7.2 2-to-1 MUX. 187
7.3 n-bit 2-to-1 Multiplexer. 187
7.4 Examples of 4-to-1 and 8-to-1 Muxes. 188
7.5 Cascaded MUXs. 189
7.6 2-to-4 decoder. 190
7.7 Decoder front-end for memories. 190
7.8 S-R latch. 191
7.9 Operations of S-R latch. 192
7.10 $\overline{S}$-$\overline{R}$ latch. 192
7.11 S-R latch with enable. 193
7.12 D latch with enable. 194
7.13 Negative-edge triggered D-FF. 195
7.14 Functional behavior of a negative-edge-triggered D flip-flop. . 195

7.15 Positive-edge triggered D-FF. 196
7.16 Positive-edge triggered D flip-flop with high enable. 196
7.17 Register with load enable. 198
7.18 4-bit shift register. 199
7.19 Bidirectional shift register with parallel load. 200
7.20 Logic diagram of a SRAM cell. 201
7.21 2^n by b-bit SRAM structure. 202
7.22 Read operation. 203
7.23 Write operation. 203
7.24 Implementing 256K×8-bit RAM using 64K×8-bit RAMs. . . 205
7.25 Implementing 256K×16-bit RAM using 256K×8-bit RAMs. . 206
7.26 Internal structure of the 2 read-port, 2 write-port register file. 207

8.1 AVR instruction formats. 211
8.2 Basic 2-stage microarchitecture 213
8.3 Program and Data Memories. 214
8.4 Sign-extension unit. 215
8.5 Zero-fill unit. 215
8.6 Arithmetic and logic Unit (ALU). 216
8.7 Alignment Unit. 218
8.8 2 read-port, 2 write-port Register File. 219
8.9 16-bit Address Adder. 219
8.10 Fetch and Execute stages. 221
8.11 Multi-cycle implementation. 221
8.12 Fetch cycle. 222
8.13 The portion of the basic datapath utilized by Arithmetic and Logic instructions. 224
8.14 The portion of the datapath for 1-cycle 8-bit Data transfer. . 226
8.15 The portion of the datapath for 1-cycle 16-bit Data transfer. 226
8.16 Part of the datapath utilized by `IN` and `OUT` instructions. . . 227
8.17 EX1 of register indirect for loads and stores. 229
8.18 EX1 of register indirect with displacement for loads and stores. 229
8.19 EX1 for register indirect with pre-decrement and post-increment for loads and stores. 231
8.20 EX2 for loads and stores. 232
8.21 EX1 of PC-relative branch instruction. 233
8.22 Micro-operations for direct jump instructions. 234
8.23 Enhanced 2-stage microarchitecture 235
8.24 Opcode encoding for instructions in Table 8.9. 238
8.25 Control signals for the enhanced AVR datapath. 240

8.26 Control signals for the Fetch stage. 241
8.27 Control signals required in EX1 for `ADD Rd,Rr` instruction. . 242
8.28 Control signals required in EX1 for `ORI Rd,K` instruction. . . 243
8.29 Control signals required in EX1 for `BREQ k` instruction. . . . 244
8.30 Control signals required in EX1 for `LD Rd,Y` and `ST Y,Rr` instructions. 246
8.31 Control signals required in EX2 for `LD Rd,X` and `LD X,Rr` instructions. 247
8.32 Control signals required for `CALL k` instruction. 249
8.32 Control signals required for `CALL k` instruction. 251
8.33 The Finite State Machine control for the multi-cycle datapath.252
8.34 The Finite State Machine control for the datapath. 254
8.35 RAL Mapping for `ADD` and `ORI`. 254
8.36 RAL Mapping for `LD` and `ST`. 255
8.37 RAL Mapping for `CALL`. 257
8.38 Register Address Logic. 257
8.39 Pipeline implementation of the datapath. 258

9.1 Information in a computer. 260
9.2 Full adder and its truth table and K-map. 268
9.3 Full adder implementation. 269
9.4 Ripple Carry Adder. 269
9.5 Ripple Carry Adder delay. 270
9.6 A complementer with its Function Table. 271
9.7 Ripple Carry Adder with a complementer. 271
9.8 Bit-slice ALU. 272
9.9 Bit-slice arithmetic operations. 273
9.10 Bit-slice logic operations. 274
9.11 Setting condition codes 275
9.12 Two conditions that cause overflow. 276
9.13 n-bit Carry Look-ahead Adder 278
9.14 A cascaded CLA 278
9.15 A block CLA 279
9.16 Hardware implementation of Add-and-Shift multiplier. 282
9.17 Array multiplier. 290
9.18 A large multiplier implemented with small multipliers 291
9.19 Single-precision IEEE 754 floating-point format. 297
9.20 Underflow and overflow ranges in single-precision IEEE 754 floating-point format. 298
9.21 Double-precision IEEE 754 floating-point format. 299

9.22 Floating-point add operations. 300
9.23 Sticky, Rounding, and guard bits for rounding. 303

A.1 Category 1 opcode encoding. 312
A.2 Category 2 opcode encoding. 312
A.3 Category 3 opcode encoding. 320
A.4 Category 4 opcode encoding. 321

B.1 AVR Studio Project Creation. 324
B.2 I/O Window in Workspace. 327
B.3 Memory Window. 328

C.1 PonyProg2000 Interface Board Setup. 333
C.2 Fully Initialized PonyProg2000. 334
C.3 Successfully Loaded AVR Hex Program File in PonyProg2000. 335

List of Tables

3.1 Functional Requirement of Tri-State buffer. 47
3.2 Instructions in the pseudo-ISA. 50

4.1 Common Instruction Nomenclature. 83
4.2 Conditional Branch Summary. [1]Interchange Rd and Rr in the operation before the test, i.e., CP Rd,Rr ? CP Rr,Rd . . 84
4.3 Conditional Branch Summary. 87
4.4 Assembler Directives. 92
4.5 Expression Operators. 97
4.6 Line Formatting Rules. 98
4.7 Code Structure. 100

5.1 Interrupt vectors 132
5.2 Definitions for BOTTOM, MAX, and TOP 140
5.3 Clock Select bits for TCNT0 145
5.4 Description of Waveform Generation Mode Bits 145
5.5 Description of Compare Output Mode Bits (non-PWM mode) 145
5.6 Description of Compare Output Mode Bits, Fast PWM mode 146
5.7 Description of Wave Generation Mode Bits 148
5.8 Description of Compare Output Mode Bits, Fast PWM mode 149

6.1 Data Types. 155

8.1 Arithmetic and Logic Operations for the 8-bit ALU 216
8.2 Operations for the 16-bit Address Adder. 219
8.3 Micro-operations for the Fetch Stage 223
8.4 Micro-operations for Arithmetic and Logic Instructions 223
8.5 Micro-operations for Move and I/O Instructions 225
8.6 Micro-operations for Load and Store Instructions 228
8.7 Branch and Jump Instructions 230

8.8 Micro-operations for the Fetch Stage 236
8.9 AVR Instructions for Control Unit Design 237
8.10 Direct Subroutine Call. 246
8.11 Summary of control signals for instructions in Table 8.9 . . . 250
8.12 Finite state table for the multi-cycle implementation. 253

9.1 Booth recoding table. 285
9.2 Bit-pair recoding table. 288
9.3 Results of operations on special numbers. 299
9.4 Rounding modes in the IEEE 754 standard. 303
9.5 Errors in the Round to Nearest Even mode. 304

A.1 AVR Arithmetic and Logic Instructions 307
A.2 Data Transfer Instructions 308
A.3 AVR Arithmetic and Logic Instructions 309

Chapter 1

Introduction

{(BL): Work in progress...}

1.1 The Role of Computers in Modern Society

There is no doubt about the significance of computers in our daily lives. Imagine where we would be without personal computers (PCs)to handle our daily chores at work and home, and as a source of entertainment? Although the importance of computers in modern society is unmistakeable, there is another facet of computers that most people are vaguely aware of - *embedded systems* or *computers*. Unlike general-purpose desktop and laptop computers, embedded computers are designed to perform one or a few dedicated functions, and more importantly, are *embedded* as a part of a complete device that often include hardware and other mechanical parts. For example, remote keyless entry systems for automobiles have embedded systems that send special code to lock/unlock doors as well as other functionalities. Consumer electronics, medical equipment,

The meaning of embedded systems is hard to define exactly and has evolved over the years. For example, systems that control household appliances (e.g., microwave ovens, washing machines, dishwashers, etc.), automobiles (e.g., Anti-lock Braking System, rain sensor, Electronic Stability Control, etc.), and medical equipment (e.g., patient monitoring systems) still follow the classic definition of an embedded system that performs *special-purpose* tasks However, a more recent meaning of an embedded system is some combination of computer hardware and software, either fixed in capability or programmable, that is specifically designed for a particular kind of application device. These devices include Portable Media Players (PMPs)

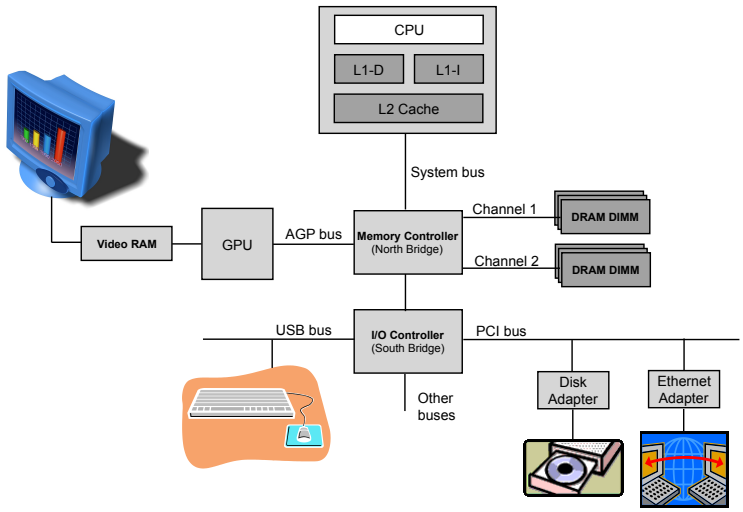
and mobile phones, and more recently, smartphones and pad devices with mobile operating systems (e.g., Google Android, Microsoft Windows Mobile, Apple iOS, RIM Blackberry, etc.) and ability to download and install *apps* for additional functionality. This increase in flexibility has blurred the distinction between special-purpose embedded systems and general purpose computers such as desktops and laptops. Therefore, a more accurate meaning is any system with a computer that is not a desktop or laptop is an embedded system.

So why are we focusing on embedded systems when most of us interact with desktops and laptops? Despite familiarity and popularity of PCs, more than 95% devices with computers are embedded systems. For example, a high-end car can have over 100 embedded processors controlling everything from electronic throttle to climate control. Even a typical house contains embedded system to control Heating, Ventilating, and Air Conditioning (HVAC)

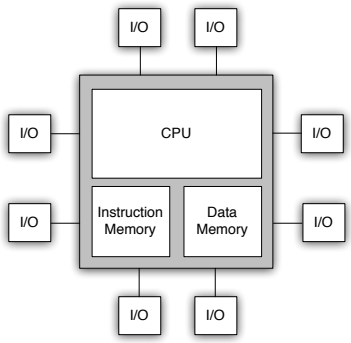
1.2 Spectrum of computers and their processors

The spectrum of computers varies as much as their intended applications. Desktop, laptops, and servers are design to handle multitude of applications from document processing to video conferencing. As such, these systems contains an array of peripheral devices controlled by a powerful processor. Figure 1.1(a) illustrates the organization of a typical desktop system (laptop systems are also fundamentally similar). At the time of writing of this book, such as system is controlled by a processor or *Central Processing Unit* (CPU) with level-1 (L1) and level-2 (L2) caches, and in some cases level-3 (L3) cache, and clocked at 2∼3 GHz. They also support expandable memory up to several Giagabytes and even have a separate chip, called Graphics Processing Unit (GPU) , to process graphics for displays. Besides the processor, another central part of a desktop system is a "chip set" containing a *Memory Controller* and *I/O Controller*, also referred to as North Bridge and South Bridge, respectively. These two controllers allow the high-speed part of the system, i.e., processor, memory, and graphics, to be interfaced to the slower I/O peripherals such as USB devices, hard-disks, and wired (i.e., Ethernet) and wireless network (WLAN) connections.

In contrast, an embedded system is a self-contained *System-on-Chip* (SoC) that contain all the basic components needed to control I/O devices. These components include a processor running at few to tens of MHz with Instruction and Data Memories of tens to hundreds of Kilobytes. Unlike



(a) Desktop system.



(b) Embedded system.

Figure 1.1: Desktop versus Embedded systems.

desktop systems, I/O devices they control are usually simple devices, such as sensors, motors, and other mechanical devices, and due to their size and speed are low power.

Talk about mobile devices

-Desktop, laptop, servers -Embedded system mobile systems microcontrollers

What are the thing readers need to know about computers? Its parts? (A Computer System) -Desktop systems vs. embedded systems (nice diagram)

What is happening inside your computer

Evolution of computing systems - Hardware - software - multimedia - communications, - game

Objectives for the book - Why is computer organization and assembly language programming important?

-what do we to know about solving a problem using computers? - Problem - Algorithm - Language - ISA - Microarchitecture - circuits - devices

What do we cover in this book

Roadmap for the rest of the book

Chapter 2

Assembly Language Fundamentals

2.1 High-Level Language to Machine Language

Most of us write programs using *high-level languages*, such as C/C++, Fortran, or Java, and rely on sophisticated software libraries that implement complex functions to simply our programming tasks. These programs are then compiled using another sophisticated program, i.e., compilers, to generate binary executables that processors understand. During execution, programs may also rely on run-time support provided by operating systems (OSs), such as system calls, or syscalls, that provide services for process control, file and device management, and communication. This allows us to concentrate only on programming without worrying about how the hardware understands and processes our algorithmic intent. However, the knowledge needed to design and build such a hardware requires the understanding of how low-level commands or instructions translated from high-level languages are executed by the processor. These low-level instructions are called *machine instructions* and represent *machine code* or *machine language*, which is the language the hardware understands.

Machine instructions consist of 0s and 1s and thus are hard to understand and program by humans. A much more readable form of machine language, called *assembly language*, uses *mnemonics* to refer to machine code instructions. Mnemonics are a symbolic representation of the machine codes and other data needed to program a particular processor architecture, and thus make it easier for programmers to remember individual machine instructions, registers, memory locations, etc. An assembly language is unique to

each processor manufacturer, and thus, unlike high-level languages, is not portable.

Even though compilers (or an interpreter in the case of Java) do all the hard work to translate high-level languages to a machine language specific to a particular processor, the concept of *assembly language programming* is important for a number of reasons. First, the characteristics of assembly instructions, and thus machine instructions, strongly influence processor design. Therefore, a processor designer must be well versed in assembly programming and understand the features of assembly instructions to design and implement efficient and fast processors. Second, compiler writers must understand assembly language programming to map machine independent intermediate representation to machine dependent code. Third, writing software that interacts directly with the hardware, such as in device drivers and interrupt facilities, requires a clear understanding of I/O operations provided by an assembly language. Fourth, writing real-time applications that need precise timing and responses, such as simulations, flight navigation systems, and medical equipment, requires greater visibility and control over processing details that only assembly languages can provide. Finally, all programmers can write better programs by knowing how their software is executed by a processor.

Therefore, this chapter discusses the fundamental concepts of assembly language, starting with how assembly instructions are represented, what and how much information is contained in the instructions, and how these design choices affect the instruction format, and thus code density and the power of the instructions. Later in Chapter 4, we will study in detail the assembly language of the AVR microcontroller. We will then see in Chapter 8 how the characteristics of AVR assembly language influence the design of a processor.

2.2 How Do We Speak the Language of the Machine

Figure 2.1 shows the hierarchical relationship between software and hardware. Applications written in high-level languages rely on compilers to generate binaries and, if necessary, request run-time services from an operating system. Applications can also be written in assembly language that are *assembled* into binaries. Regardless of whether high-level or assembly language is used, the resulting binary executables are in the format that the processor hardware can understand and execute.

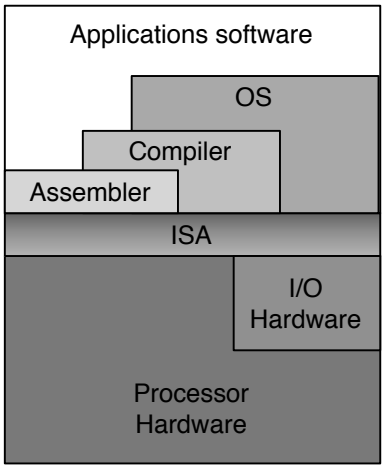


Figure 2.1: Simplified hierarchical view of software and hardware and the role of ISA.

The *Instruction Set Architecture* (ISA) is the portion of the processor visible to the programmer or the compiler. The concept of ISA is important because it defines the *basic set of instructions* performed by the processor and the only information that the processor understands. Thus, assembly programs are made up of instructions in the ISA. Therefore, ISA basically acts as an *interface* either for programmers to directly write or for software to transform processor-independent programs into processor-dependent programs, or *machine language programs*.

2.3 Instruction Set Architecture

Now that the important layer or gateway to the processor hardware has been exposed, we are ready to discuss the design of ISAs and how it affects programmers or compilers as well as processor design.

Assembly programmers and compiler writers write software based on existing ISAs, e.g., x86, AVR, ARM, etc. However, how are ISAs designed and what is the thought process that goes into coming up with an ISA? A designer of a new ISA has to contend with a variety of factors that affect programming as well as speed, cost, and complexity of processor hardware. These factors include

- Operations provided in the instruction set;
- Number of explicit operands named per instruction;
- Location of the operands and how operand locations are specified; and
- Type and size of operands.

The following subsections discuss each of these factors in more detail.

2.3.1 Operations in the ISA

What should be the set of operations provided in an ISA? This choice can be akin to buying a car. There are so many choices including the type of brand and model, and the options available in the car. For the design of an ISA, some important criteria are

- Functional completeness;
- Efficiency (power of the instruction); and
- Simplicity of hardware design and/or programming.

Functional completeness refers to the need for an ISA to provide a comprehensive set of instructions to implement any given program. The degree of functional completeness will vary from one ISA to another. For example, should an ISA have instructions to handle floating-point operations? The answer depends on the purpose of the processor. If applications running on a processor require extensive floating-point operations, then it may be advantageous from a performance standpoint to provide floating-point instructions with accompanying hardware (e.g., floating-point unit). This will in turn improve speed but increase complexity and cost.

Efficiency or *power* of an instruction refers to what a single assembly instruction can accomplish. Some instructions are very powerful and can implement complex tasks. Other instructions are simple and require more instructions to accomplish the same task. The tradeoff between two options depends on the applications of the processor. For example, a processor designed for simple tasks, such as low-end embedded processors for motor control and light and rain sensors, will not need to be high-speed and thus can be programmed with more instructions. On the other hand, processors designed for personal computers require powerful instructions running at high speeds.

Simplicity of hardware design and/or programming is directly related to the two previous mentioned factors. From a programmer's point-of-view, complex instructions will make programming easier, while simple hardware leads to more complex programs and vice versa.

Based on the aforementioned discussions, the right tradeoff between instruction efficiency and simplicity in hardware design may not be obvious at this point. The answer lies in the characteristics of programs and how often these complex instructions are encountered. When complex instructions are frequently encountered, it may be better to dedicate hardware to handle these instructions. However, years of research on processor design have also shown that instruction sets based on the *Reduced Instruction Set Computer* (RISC) leads to better performance with minimal affect on programming. The RISC concept is based on the fact that simple and common instructions provide higher performance when this simplicity allows for much faster execution of each instruction. Therefore, with few exceptions (e.g., x86), most modern microprocessors are RISC-based.

Categories of Instructions

The categories of instructions offered by an ISA define the functional completeness of a processor. The different types of instructions that an ISA may have are:

- Data transfer
- Arithmetic
- Logical
- Control transfer
- I/O
- System
- Floating point
- Decimal
- String

Among these different types of instructions, most processors provide the first five instruction categories (data transfer, arithmetic, logical, control

transfer, and I/O). *Data transfer* instructions cause information to be copied from one location to another either in the processor's internal memory (registers) or in the external main memory. A processor without these instructions would be rendered useless since data required by applications reside in the memory (possibly after being transferred from a storage device, such as a flash drive or a hard disk, to memory). *Arithmetic* instructions perform operations on numeric data (e.g., add, subtract, etc.). *Logical* instruction perform Boolean and other non-numerical operations (e.g., AND, OR, NOT, etc.). A processor without arithmetic and logic instructions would defeat the whole purpose of having a computer in the first place. *Control transfer* instructions change the sequence of program execution (jump and conditional branches). These instructions are crucial for implementing conditional statements, such as IF-THEN-ELSE and CASE statements provided in high-level languages. Last but not least are *I/O* instructions, which cause information to be transferred between the processor or its main memory and external I/O devices (e.g., display, keyboard, mouse, printer, sensors, etc.). Without these instructions, computers would not be able to communicate with the outside world.

Whether or not the other types of instructions are needed depends on the application of the processor. *System* instructions are used for operating system calls. Syscalls provide an interface between a user program and the operating system to provide services such as dynamic memory management and I/O operations. It is obvious that syscalls are necessary for processors on personal computers that run Windows or Mac OS X operating systems. However, embedded systems designed to control say engine idling does not require an operating system and thus syscalls. The same is true for *floating-point* instructions. An embedded system designed to process only integer data will not require floating-point instructions. If these instruction are needed, they can be implemented using existing instructions at a cost of higher programming complexity and lower speed. *Decimal* instructions directly manipulate decimal numbers. Typically, decimal numbers are first converted to binary numbers before performing arithmetic operations, and then converted back to decimal numbers before they are stored in a file or printed on a display. The conversion between decimal and binary is time consuming. Therefore, having instructions that can directly manipulate decimal numbers would be beneficial in terms of speed. However, decimal operations are different from binary operations, and thus the processor has to be augmented with special hardware to handle such instructions. The same is true for *string* instructions that allow manipulations of characters. Nonetheless, encoding these instructions into the ISA increases the complex-

ity of the processor and may not be worth it if these operations are rarely used.

2.3.2 Number of Operands per Instruction

The second factor in ISA design is the number of operands supported by the instruction. An *operand* refers to the data to be manipulated or operated on by an instruction. The number of operands associated with each instruction can be considered in terms of the following issues:

- Control circuit complexity (decoding);
- Storage required for instructions (code density);
- Power of instructions; and
- Number of instructions required to perform a given task.

In order to illustrate this trade-off, consider the following example add instruction where `x`, `y`, and `z` represent addresses of memory locations or registers.

- 4-address instruction

$$\texttt{add z, x, y, } \textit{goto } \texttt{q} \tag{2.1}$$

`x` and `y` represent the addresses of the two source operands and `z` represents the address of the destination and is equivalent to the operation $\texttt{z} \leftarrow \texttt{x} + \texttt{y}$. In addition to the add operations, this instruction also defines a *target address* `q`, which will be the address of the next instruction to be fetched and executed. This 4-address instruction format is the most powerful but explicitly defines `x`, `y`, `z`, and `q` within a fixed size instruction.

- 3-address instruction

$$\texttt{add z, x, y, (q is implied)} \tag{2.2}$$

This format is almost identical to the 4-address instruction format discussed above, but omits `q`. The reasoning behind this is simple. Most programming languages are *imperative*. That is, statements in high-level languages and thus instructions in assembly language are executed in step-wise, sequential manner. Therefore, after the current instruction is executed, the likelihood that the instruction after the current instruction will be executed is high.

Thus, rather than explicitly defining `q` in each instruction, this information is maintained in a special register called *Program Counter* (PC) and its content is incremented after each instruction execution. The 3-address instruction format is typically used in high-end microprocessors for personal computers and mobile computing platforms, such as cell-phones, Portable Media Players (e.g., iPod), and tablet computers (e.g., iPad).

- 2-address instruction

$$\texttt{add x, y} \tag{2.3}$$

This format is similar to the 3-address instruction format discussed above, but both destination and one of the source operand is defined by `x` and is equivalent to the operation $\texttt{x} \leftarrow \texttt{x} + \texttt{y}$. The 2-address instruction format is typically used in mid-range microcontrollers, e.g., AVR.

- 1-address instruction: Accumulator-based architecture

$$\texttt{add x} \tag{2.4}$$

This format defines only one explicit operand. The second operand as well as destination are *implicitly* defined by a special register called the *Accumulator* (AC). Thus, this instruction performs the operation $\texttt{AC} \leftarrow \texttt{AC} + \texttt{x}$. The disadvantage of this instruction format is that AC is involved in every operation, and thus additional instructions are needed to move the data in AC to any other register or memory location. This leads to a larger number of instructions to accomplish the same task than using either 3-address or 2-address instruction formats. The advantage is that the length of the instruction format is shorter. This instruction format is very common in low-end microcontrollers.

- 0-address instruction: Stack-based architecture

$$\texttt{add} \tag{2.5}$$

This is a unique format in that no operands are explicitly defined. Instead, a stack is implied where an operation is performed by popping the top two locations of the stack representing the two source operands, and the result is pushed onto the stack. There are no processors that use this instruction format. However, the concept of stack-based arithmetic is used in HP calculators with Reverse Polish Notation (RPN).

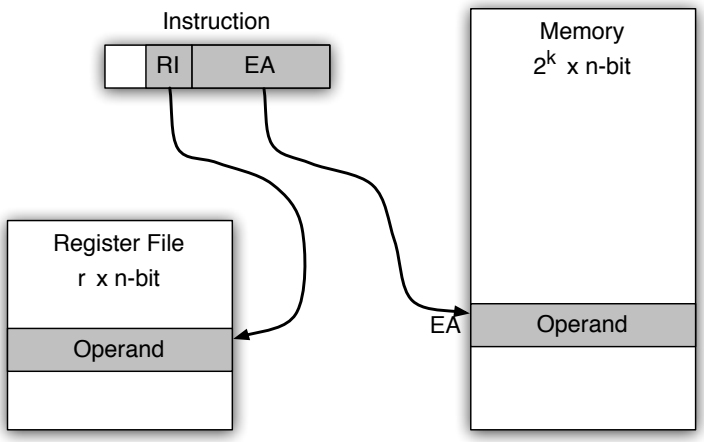
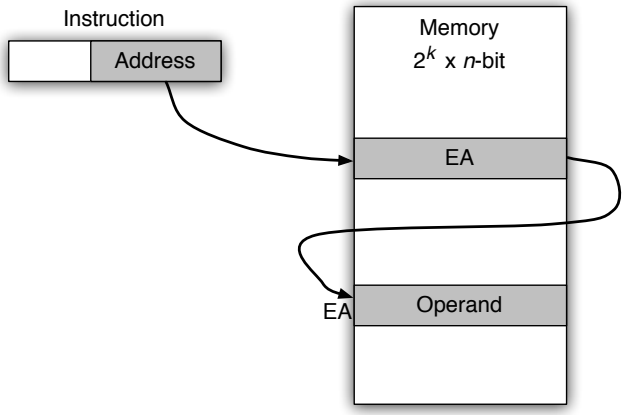


Figure 2.2: An example instruction that refers to operands in both memory and register.

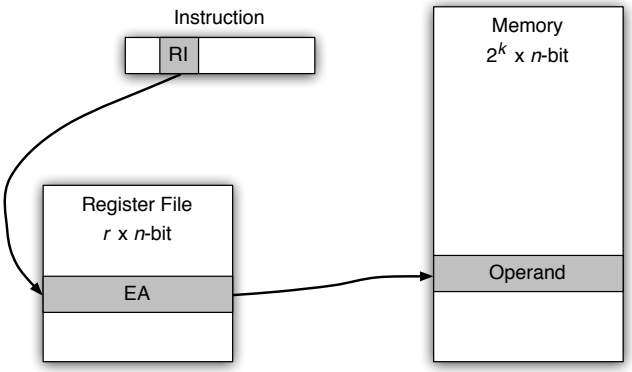
2.3.3 Operand Locations and How They are Specified

An operand can reside in either a register or a memory location. Figure 2.2 shows an example instruction format that refers to operands in both a register and a memory location. The instruction refers to the operand in memory by its k-bit address, known as its *effective address* (EA), and the operand in the register file by its $log_(2)r$-bit register address, known as a *register identifier* (RI). Therefore, both EA and RI have to be encoded within the fixed size instruction, which is typically 16 bits or 32 bits.

Operand locations can be specified in a number of different ways. The way in which operands are identified is called an *addressing mode*. For example, consider again the example shown in Figure 2.2. When an effective address encoded in an instruction directly refers to an operand in memory, it is called *direct* or *absolute addressing*. When a register identifier encoded in an instruction refers to a register in a register file, it is called *register direct addressing* or simply *register addressing*. In addition to these two addressing modes, there are other ways instructions can refer to operands. Figure 2.3(a) shows an example of *memory indirect addressing*. In this example, the effective address is in a memory location instead of being encoded in the instruction. Figure 2.3(b) shows an example of *register indirect addressing*, where the effective address is in a register. These figures illustrate the important distinction between direct addressing and indirect addressing;



(a) Memory indirect addressing.



(b) Register indirect addressing.

Figure 2.3: Indirect addressing.

an effective address in indirect addressing can be thought of as a pointer to either an array or a structure and any element within these data structures can be accessed by manipulating the effective address. Note that this would not be possible with direct addressing since EA and/or RI encoded within the instruction would have to be modified. This would amount to a *self modifying code*, where the code modifies its own instructions while executing, which is not a recommended coding practice.

There are also variations of indirect addressing modes, including *pre-*

decrement, *post-increment*, and *displacement*, which allow the effective address to be decremented, incremented, and added with a displacement, respectively. These addressing modes as well as others will be discussed in more detail when we discuss the AVR Assembly Language in Chapter 4.

2.3.4 Operand Type and Size

Operand size can be 8-bit, 16-bit, 32-bit, or 64-bit depending on the processor. Note that we are talking about operand size, not instruction size, which again is typically either 16-bit or 32-bit. When manufacturers refer to n-bit processors, n refers to the size of the operand. For example, the 8-bit AVR microcontroller discussed in Chapter 4 has an operand size of 8 bits. Obviously, the larger the operand size, the more powerful the instructions. For example, a 32-bit processor that support 32-bit operands will typically have 32-bit registers and 32-bit memory locations. Therefore, 32-bit data can be manipulated with a single instruction.

The reason 8-bit and 16-bit processors exist is that most applications that these processor were designed for do not require large operand sizes. For example, embedded systems for temperature sensor or analog-to-digital conversion only require 8-bit or 16-bit data depending on the desired resolution. Now, this does not mean 8-bit or 16-bit processors cannot support 32-bit operations. It just means that more instructions are needed by these processors to perform an operation that can be done with a single 32-bit instruction.

2.4 Instruction Format

Based on the aforementioned discussions, a variety of information has to be conveyed to the processor through assembly instructions. *Instruction format* is the interface between program languages and processor hardware, and its layout is composed of fields of binary numbers. Figure 2.4 shows an example of a generic instruction format, which for modern processors is typically either 16-bit or 32-bit wide. It consists of an *opcode field* and a number of *address fields*, each representing a specific item needed by the instruction, such as register identifier, memory address, constant, etc.

The opcode field specifies an operation, such as add, subtract, shift, branch, etc. Each address field specifies the location of an operand either in a register or a memory location. The size of the opcode field is dictated by the number of operations the ISA supports. For example, an opcode field of k bits can encode up to 2^k different operations. The similar is true for

Figure 2.4: Instruction format.

address fields. For example, if the memory size is 4K (i.e., $2^{12} = 4,096$) words, then the number of bits to specify an address is 12 bits.

It should be pretty obvious by now that there is a limit on the number of opcodes and addresses an instruction format of either 16 or 32 bits can support. For example, with a 16-bit instruction format, if an address field directly references or points to a memory location and the opcode field is 4 bits, the instruction format can only support one address with a memory size of $2^{12} = 4,096$ words and 16 different operations. Even allocating one more bit to the opcode field to support 32 operations reduces the addressable memory size down to 2K (or 2^{11}) words.

Instruction formats that supports either a 2-address or a 3-address format get around this limitation by employing a *register file*, which is an array of registers. The size of the register file is typically 16 or 32 entries, which significantly reduces the number of bits for the address fields. For example, a 16-bit instruction format with a register file containing 16 entries and opcode field size of 4 bits, can support up to 3 addresses. Even with a 32-entry register file, up to 2 addresses can be supported with a couple of bits to spare. For a 32-bit instruction format, there is ample room to support up to 3 addresses with room to spare other encoding possibilities.

A typical ISA supports well over one hundred different instructions. This will require 6 or more bits for the opcode field, which cannot be supported by static encoding schemes discussed above. Thus, another technique, called *opcode extension*, is use to expand the encoding space within the instruction format. The basic idea is to have one (or a number of) opcode pattern(s) to indicate to the processor decoder that other bits in the instruction format are used to encode additional instructions. Therefore, the size the opcode field and location of its bits vary depending on the type of instructions. We will see an example of this when the ISA of the AVR processor is discussed in detail in Chapter 4.

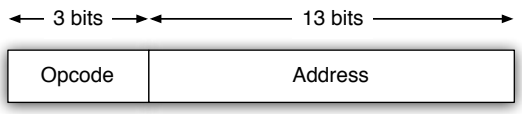


Figure 2.5: Instruction format of the hypothetical ISA.

2.5 A pseudo-ISA

In order to illustrate the concepts discussed in this Chapter, this section discusses a design of a simple, pseudo-ISA. The instruction format for the hypothetical ISA is shown in Figure 2.5, which is an 1-address instruction format containing an opcode field and an address field. For the sake of discussion, let us assume the instruction format is 16 bits wide. Our hypothetical ISA will provide only *eight* instructions, and thus, the number of bits in the opcode field is 3 bits (i.e., $2^3 = 8$). The rest of the 13 bits is allocated for the address field, which allows up to $2^{13} = 8,192$ memory locations to be directly addressed. Since the instruction format is 16 bits, each memory location and the accumulator (AC) are also 16 bits.

Our hypothetical ISA supports the following set of instructions:

- Data transfer Instructions
  - `LDA` (Load Accumulator): Loads a memory word to accumulator
    - `LDA x`; `x` is a location in memory
  - `STA` (Store Accumulator): Stores the contents of accumulator to memory
    - `STA x`; `x` is a location in memory
- Arithmetic and Logical Instructions
  - `ADD` (Add to accumulator): Adds the content of memory word specified by the effective address to the value of the accumulator.
    - `ADD x`; `x` is a location in memory
  - `SUB` (Subtract from accumulator): Subtracts the content of memory word specified by the effective address from the value of the accumulator.
    - `ADD x`; `x` is a location in memory

  - `NAND` (logical NAND to accumulator): Performs logical NAND between the content of memory word specified by the effective address and the value of the accumulator
    * `NAND x`; `x` is a location in memory
  - `SHFT` (Shift): Shifts the content of AC by one bit to the left. The bit shifted in is 0.

- Control Transfer
  - `J` (Jump): Transfers the program control to the instruction specified by the address.
    * `J x`; jump to instruction in memory location `x`
  - `B`*cc* (Branch Conditionally): Transfers the program control to the instruction specified by the address based on condition cc.
    * `BNZ x`; jump to instruction in memory location `x` if content of AC is not zero

Despite the fact that there are only eight instructions, our hypothetical ISA is functionally complete. The pair of data transfer instructions `LDA` and `STA` allow operands or data to be transferred between the memory and AC. The combination of ALU instructions `ADD`, `NAND`, and `SHFT` allows for coding of any arbitrary arithmetic and logic functions. For example, a subtract operation can be performed by adding the two's complement of the second operand to the first operand (see Chapter 9.3). A multiply operation can be performed by successive add and shift operations (see Chapter 9.4). In terms of logic operations, NAND is functionally complete and thus any logic operation can be perform using NAND. `J` and `BNZ` instructions allow for control transfer.

You may have noticed that I/O instructions are conspicuously absent. It would have been ideal to add a pair of `INP` and `OUT` instructions for I/O operations. However, since the 3-bit opcode field does not allow room for any more instructions, we instead opt for using `LDA` and `STA` instructions to perform *memory-mapped* I/O operations. The basic idea of memory-mapped I/O is to use the same address bus to address both memory and I/O devices, instead of having a separate, dedicated port for I/O. This is in contrast to port-mapped I/O, where a special class of instructions, such as `INP` and `OUT`, are used for performing I/O.

Now that we have discussed the operations of the eight instructions, let us write a small assembly program using our hypothetical ISA. Figure 2.6

```
;
; A simple C program
;
main()
{       int     A = 83, B = -23, C = 0;
        C = 4*A + B;

}
```

Figure 2.6: A simple C program.

shows an example C program that multiplies variable `A` by 4 and adds it to variable `B` and assigns the result to variable (`C`).

The equivalent assembly program is also shown in Figure 2.7, which consists of mnemonics, a data section, and assembly directives. Mnemonics represent the code for the assembly program and consist of `LDA`, `SHFT`, `ADD`, `STA`, and `B` instructions. The *data section* defines data elements in memory and consists of the three `.DEC` assembly directives. *Assembly directives* are special instructions that are executed by the assembler at assembly time, not by the processor at run-time. There are several types of assembly directives in the assembly program. For example, the assembly program starts with an `.ORG` directive and ends with an `.END` directive. The `.ORG` directive defines the starting location of the code and data section in memory. Thus, `.ORG 0` indicates the first instruction in the code, i.e., `LDA A`, will be located at memory location 0. The `.END` directive indicates the end of the program.

The C statement `C = 4*A + B;` is implemented by the sequence of assembly instructions `LDA`, `SHFT`, `ADD`, and `STA`. The `LDA A` instruction loads the variable `A` from memory to AC. The two `SHFT` instructions multiply the variable `A` by 4. The `ADD B` instruction adds the variable `B` to the contents of AC (i.e., `A`) and stores the result back into AC. Finally, the `STA C` instruction stores the result of the add operation to variable `C` in memory.

At this point, there may be some confusion about what is meant by 'a variable in memory?'. Variables `A`, `B`, and `C` were declared in the C program by the statement `int A = 83, B = -23, C = 0;`, which states that these are of type integer (16 bits) and variables `A` and `B` are initialized to 83 and -23. This allows the complier to appropriately allocate locations in memory and assign decimal values. This is achieved in assembly language by using directives. In the example assembly program, the `.DEC` directive allocates a

```
;
; Equivalent assembly program
;
      .ORG  0     ; Origin of the program is location 0
      LDA   A     ; Load operand from location A to AC
      SHFT        ; Multiply A by 2
      SHFT        ; Multiply 2*A by 2
      ADD   B     ; Add operand from location B to AC and store result back in AC
      STA   C     ; Store result in AC to location C
Loop: J     Loop  ; Loop forever
A:    .DEC  83    ; Decimal operand A
B:    .DEC  -23   ; Decimal operand B
C:    .DEC  0     ; Initial value of location C
      .END        ; End of symbolic program
```

Figure 2.7: An equivalent assembly program for code in Figure 2.6.

memory location pointed to by a label and stores the initialized value. For example, the line `A: .DEC 83` states that the memory location labeled `A` is initialized with a decimal value 83. The label `A` can be anything as long as the assembly programer or the compiler writer is aware of the fact that the label `A` in the assembly program is referring to the variable A in the C program.

You may have noticed that the last instruction in the code (`J Loop`) is an unconditional branch to itself resulting in an infinite loop. You may wonder why we would write a program with an infinite loop instead of having a special instruction that would halt the program. The answer to this question is that processors are always executing instructions and never stay idle. Moreover, they can be 'woken up' by external events through the *interrupt handling facility* (See Chapter 5). For example, a word processor program is always executing some instructions in the background even when you don't type any words. However, as soon as you type a character, it interrupts the processor and the program comes out of its dormant state.

Chapter 3

Computer Organization Fundamentals

Computer organization or *computer architecture* defines how the various digital components are organized, interconnected, and inter-operate in order to implement a computer system.

Computer architecture design consists of the following subcategories: instruction set architecture, microarchitecture, and system design. The concept of instruction set architecture (ISA) has already been discussed in Chapter 2. *Microarchitecture* involves a lower level, more concrete and detailed, description of the system that involves how the constituent parts of the system are interconnected and how they interoperate in order to implement an ISA. *System design* involves all of the other hardware components within a computer system.

Figure 3.1 shows a typical computer system, which consists of processor or *Central Processing Unit* (CPU), memory, and Input/Output (I/O). All computer systems, whether they are general-purpose computers, such as desktops and laptops, or embedded systems, such as media players and motor controllers, are basically designed this way. The memory holds instructions and data. I/O is is used to communicate with the outside world (e.g., sensors, motors, network interfaces, printer, disk, display, mouse, etc.). The CPU contains datapath and control unit. A *datapath* is a collection of functional blocks or components, such as registers, buses, arithmetic logic units (ALUs), etc., that performs data processing operations. The *control unit* is responsible for managing the flow of information among various components within the datapath as well as between datapath and memory or I/O. Chapter 5 provides a detailed discussion on I/O operations. This chap-

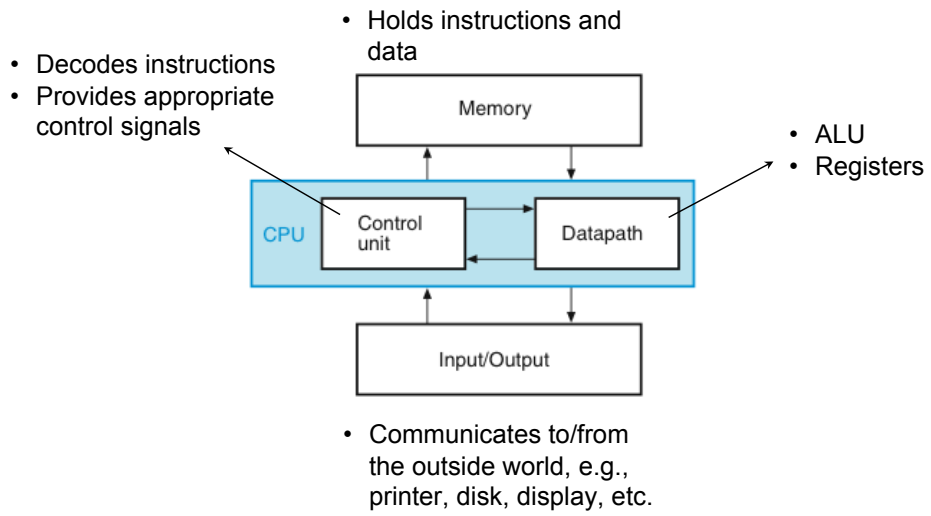


Figure 3.1: Computer Organization.

ter discusses the fundamental concepts in microarchitecture design and is meant to provide a basis for more detailed discussions in Chapters 4 and 8.

3.1 Memory

The term *memory* can have a number of different meanings. A memory is either connected externally to a CPU or integrated into the CPU chip. Memory is also referred to as *random access memory* (RAM), which allows instructions or data to be accessed in any order, and any piece of information is returned in a constant amount of time regardless of its physical location. This is in contrast to magnetic or optical disks where access time of a data depends on its location on the disk (i.e., track and sector).

A memory can be organized as *separate* instruction and data memories, or *unified* to hold both. It is organized into consecutive addressable memory words, where a memory word can have different meanings. For example, a memory word can mean the size of the information accessed by the CPU (i.e., CPU register size). For example, many high-end embedded processors have 32-bit memory words, which accommodate both instructions and data that are 32 bits. In contrast, some embedded processors have different memory word sizes for instructions and data. For example, AVR microcontroller, which will be discussed in detail in Chapters 8, has 16-bit memory word size

for instructions, while memory word size for data is 8 bits.

Memory can also be organized in a hierarchical fashion. For example, a memory can be augmented by a small, fast *cache memory*, and there can be multiple levels of caches, e.g., level-1 (L1), level-2 (L2), and level-3 (L3) caches. Memory can also be supplemented with larger but slower magnetic (i.e., hard-disk) and solid-state (i.e., flash drive) storage devices. The number of levels and the complexity of the memory hierarchy depends on the computer system and its applications. For example, personal computers have a memory hierarchy consisting of all of the levels discussed thus far. In contrast, embedded systems controlled by simple microcontrollers may only have one or two levels of memory hierarchy (e.g., register file and memory).

3.2 Microoperations

A CPU executes an assembly instruction by performing a sequence of *micro-operations*. A micro-operation is a basic operation performed on information (instruction and data) stored in registers or in memory in a single clock cycle, or *tick*. The sequence of micro-operations required to implement an assembly instruction greatly depends on the microarchitecture. More importantly, there must be no ambiguity in describing the set of micro-operations. To this end, *Register Transfer Language* (RTL) description is used to represent registers and specify the operations on their contents. RTL uses a set of expressions and statements that resemble statements used in *Hardware Description Language* (HDL) and programming languages. This notation allows clear and concise specifications of part or all of a complex digital system, such as a processor.

The most fundamental operation is data transfer from one register to another represented using the *replacement operator* ($\leftarrow$). For example, the statement

$$R1 \leftarrow R2,$$

denotes that the content of register $R2$ is transferred to register $R1$. Thus, $R1$ represents the *destination* register and $R2$ represents the *source* register. Note that this is a copy operation and thus the content of $R2$ is not destroyed.

Figure 3.2 illustrates the data transfer operation. There are several important assumptions being implied with the RTL description. First, all the registers involved in the data transfer are *clocked*, which allows multiple data transfers to occur simultaneously and synchronously. Second, the number of bits in both registers are equal and thus the number of bits being transferred

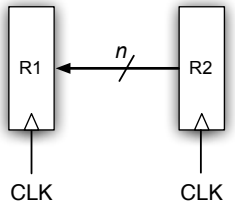


Figure 3.2: Simple data transfer between two n-bit registers.

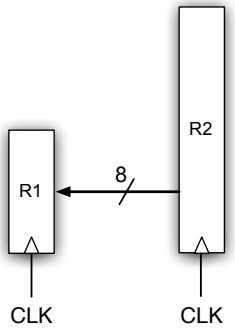


Figure 3.3: n-bit data transfer between two different sized registers.

does not have to be explicitly specified. Third, all n bits are transferred, i.e., parallel data transfer. Lastly, the order of bits are adhered to, i.e., i^{th}-bit of $R2$ is transferred to i^{th}-bit of $R1$.

A data being transferred can be specified in a number of ways. Instead of transferring all the bits of a register, a subgroup of bits and even individual bits can be specified using parenthesis. For example, the following statement indicates the transfer of bits 7 through 0 of $R2$ to $R1$.

$$R1 \leftarrow R2(7...0),$$

Note that the size of $R1$ is equal to the size of data being transferred. This type of operation is required when the size of $R2$ is different from $R1$. Figure 3.3 illustrates an example where $R2$ is a 16-bit register and $R1$ is an 8-bit register. The above micro-operation results in the transfer of the lower byte of $R2$ to $R1$.

A register can also be segmented into two halves and defined as low (L) and high (H) parts. The following statement specifies the transfer of

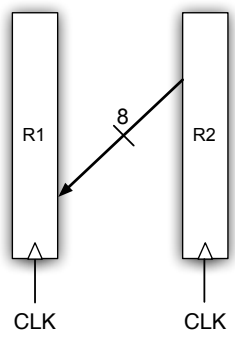


Figure 3.4: Data transfer of high-byte of R2 to low-byte of R1.

upper-half of $R2$ to lower-half of $R1$:

$$R1(L) \leftarrow R2(H).$$

This is illustrate in Figure 3.4.

A register content can be subdivided into fields of unspecified number of bits. The following two unrelated statements specify transfers of operation code portion (opcode) and address portion of $R2$ to $R1$.

$$\begin{aligned} &R1 \leftarrow R2(opcode) \\ &and \\ &R1 \leftarrow R2(address) \end{aligned}$$

Register transfers can also be performed simultaneously, which are indicated by a comma (,). The following statements show three different examples:

$$\begin{aligned} &R1 \leftarrow R3(opcode), R2 \leftarrow R3(address) \\ &R1 \leftarrow R2, R2 \leftarrow R3 \\ &R1 \leftarrow R2, R1 \leftarrow R2 \end{aligned}$$

Figure 3.5 illustrates a simultaneous data transfers from $R3$ to $R2$ and $R2$ to $R1$. In this example, the output of $R2$ is available as the input for $R1$ and the output of $R3$ is available as the input for $R2$. Therefore, when the three registers are simultaneously clocked, the input to each respective registers become latched.

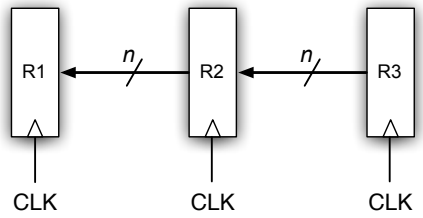


Figure 3.5: Simultaneous data transfer of $R3$ to $R2$ and $R2$ to $R1$.

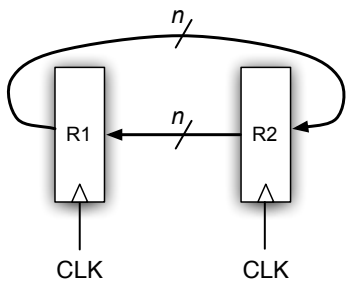


Figure 3.6: Simultaneous data transfer of $R2$ to $R1$ and $R1$ to $R2$.

Figure 3.6 illustrates a similar example. In this example, the output of $R2$ is available as the input for $R1$ and at the same time the output of $R1$ is available as the input for $R2$. Thus, these contents are latched on to the respective registers at the edge of clock.

In contrast to registers, a memory contains an array of data. Thus, square brackets are used to indicates a particular location within memory ($M[\ ]$). For example, the following statement illustrates the transfer of a data pointed to by the address in register $R2$ to $R1$:

$$R1 \leftarrow M[R2].$$

Basic arithmetic, logic, and shift operations are defined using typical operators found in high-level languages. The following statements show several different examples:

$$R0 \leftarrow R1 + R2$$
$$R0 \leftarrow R1 \vee R2$$
$$R0 \leftarrow R1 \oplus R2$$
$$R1 \leftarrow sl\ R2$$

The first three operations represent addition (+), logical NAND ($\vee$), and logical Exclusive-OR (EOR) ($\oplus$). The last operation represents *shift left* (*sl*), which shifts the n bits of *R2* to the left by one bit.

There are data transfers that occur only when certain conditions are satisfied, not every clock cycle. These cases can be represented by *conditional statements*. The following statement transfers the content of *R2* to *R1*:

$$\text{if } (cond) \text{ then } R1 \leftarrow R2,$$

where *cond* represents the condition to be satisfied.

3.3 Organization of the pseudo-CPU

In order to understand what computer organization is, in particular microarchitecture, we need to consider how the various components (memory, registers, ALU, busses, and control units) are "put together". Fig. 3.7 illustrates a very simple way to connect these components using a shared *Internal data Bus*. Obviously, microarchitecture of commercial processors are much more complex, but nevertheless the simple CPU shown in Fig. 3.7 is sufficient to implement the instructions of the pseudo-ISA discussed in Chapter 2 as well as many other assembly instructions. Moreover, implementing the pseudo-ISA on the pseudo-CPU provides a foundation for understanding how more complicated ISAs and microarchitectures work.

As can be seen from the figure, the *AC* register serves as one of the inputs as well as the output for the ALU. The other input to the ALU can be any of the registers connected to the Internal Data Bus. The *IR* register is to latch the opcode of an instruction or possibly an entire instruction so that the *CU* can decode the opcode and provide an appropriate set of control signals to execute an instruction. Note that the *PC* register has a dedicated adder, which eliminates the need to use the ALU and extra transfers between registers to perform *PC* increment. Finally, the set of *MDR* and *MAR* allow access to and from memory.

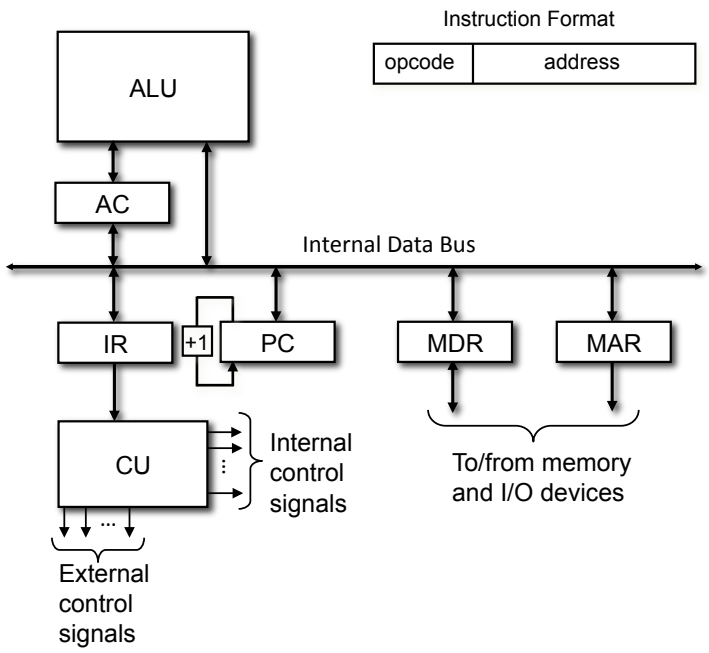


Figure 3.7: Hypothetical CPU.

3.3.1 Instruction Format

The instruction format for the pseudo-CPU is shown in Fig. 3.8. It represents an 1-address instruction format and consists of an operation code (opcode) field and an address field. The opcode field specifies an operation to be performed by an instruction, while the address field contains the address of the operand, referred to as the *effective address* (EA). Therefore, instructions that require two operands, e.g., $x + y$, one operand is in memory pointed to by EA in the address field, second operand is in AC, and opcode defines the type of operation to be performed on the two operands. Although we have not discussed the size of the instruction format, the number of bits in the opcode field defines how many unique operations can be supported by an ISA[1], and the number of bits in the address field dictates the size of the memory address space. These issues will be elaborated more in the latter part of the chapter.

[1]ISAs can also use a technique known as *opcode extension* to increase the number of supported operations.

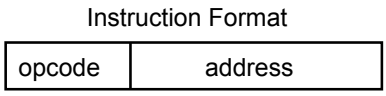


Figure 3.8: Instruction Format for the pseudo-CPU.

3.3.2 Registers

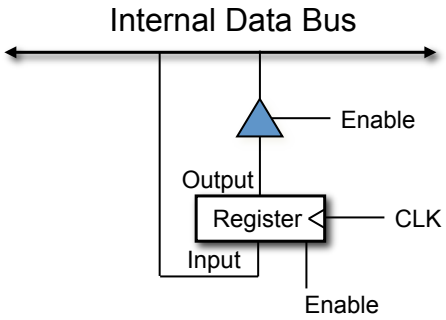
The pseudo-CPU has several important registers to hold important information associated with instruction fetch and execute.

- *PC* (Program Counter) - holds the address of the next instruction to be fetched from memory
- *MAR* (Memory Address Register) - holds the address of the next instruction or data to be fetched from memory.
- *MDR* (Memory Data Register) - holds the information (word) to be sent to/from memory.
- *AC* (Accumulator) - a special register that holds the data to be manipulated by the ALU.
- *IR* (Instruction Register) - holds the instruction to be decoded by the Control Unit (CU).

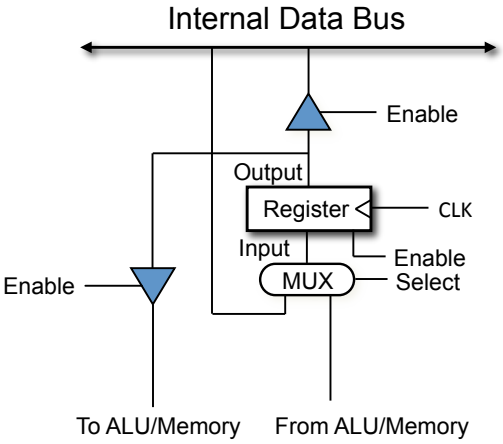
3.3.3 Bus-Register Connections

Before discussing the the basic data transfers between registers in the pseudo-CPU, some explanations are needed about how theses registers are connected to the Internal Data Bus. This is because these registers share the same bus and thus additional logic is required to make sure only a pair of registers (sender/receiver) is communicating at any one time.

Fig. 3.9(a) shows a more detailed implementation of a *single bit* of a single-input, single-output register (such as PC and IR). An n-bit version would simply have n of these connected in parallel and control by a single common control signal. As can be seen, a register is clocked (CLK) as well as enabled using an Enable signal. The CLK signal comes from a global clock and synchronizes all of the operations in the processor. On the other hand, Enable signals are generated by the Control Unit (CU), which depends on the type of instructions being executed. In order to isolate the output of a register from the Internal Data Bus, a tri-state buffer is used. Table



(a) Single-input, single-output.



(b) Multi-input, multi-output.

Figure 3.9: Bus-Register Connections.

3.1 show the functional requirement for the tri-state buffer. A tri-state act as an open-circuit (Hi-Z) unless enabled. Thus, whether or not an output of a register appears on the Internal Data Bus is controlled by the Enable signal. Fig. 3.9(b) shows a multi-input, multi-output register (such as AC, PC, MDR and MAR). In addition to tri-state buffers controlling the output fan-out, a multiplexor (MUX) controls the input fan-in. In the example shown in Figure 3.9(b), MUX controls whether the data latched onto the register is from either Internal Data Bus or ALU/Memory.

Table 3.1: Functional Requirement of Tri-State buffer.

Input	Enable	Output
x	0	Hi-Z
0	1	0
1	1	1

3.3.4 Instruction Cycle

An *instruction cycle* consists of a series of steps, referred to as *micro-operations*, required to *fetch* and *execute* one assembly instruction. A micro-operation represents what a microarchitecture can accomplish in one clock cycle. The steps required to fetch an instruction, referred to as the *fetch cycle*, is identical for all instructions. In contrast, each instruction execution, referred to the *execute cycle*, has a unique set of micro-operations. This is because the sequence and the number of micro-operations required for an instruction depends on its complexity (e.g., add vs. multiply) and hardware availability (multiplier vs. no multiplier hardware).

Fetch Cycle

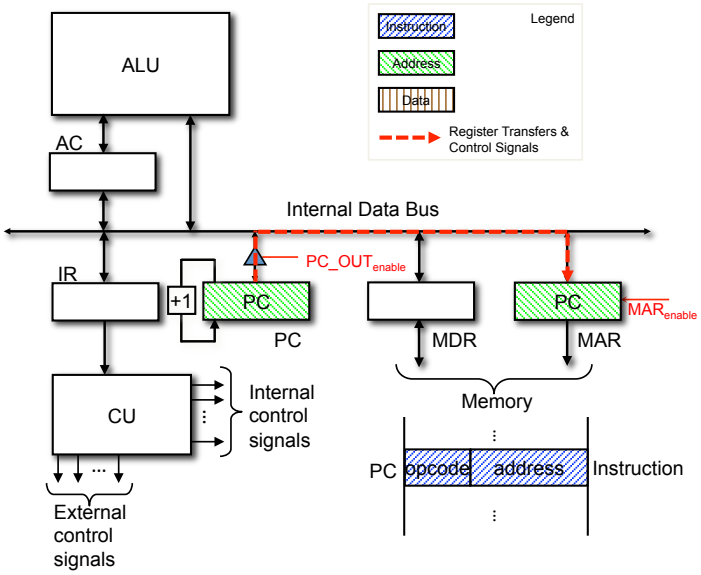
The Fetch Cycle is defined by the following sequence of micro-operations using RTL description:

1. MAR ← PC
2. MDR ← M[MAR], PC ← PC + 1
3. IR ← MDR(opcode), MAR ← MDR(address)

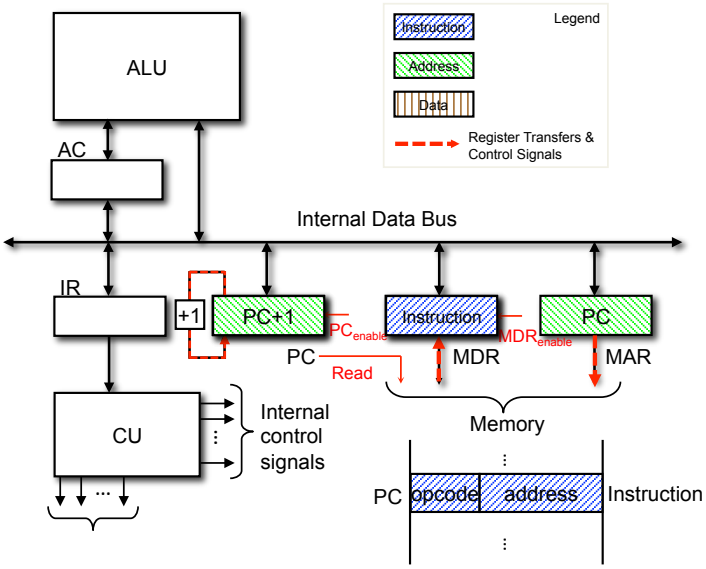
Figure 3.10 illustrates the sequence of micro-operations required for the fetch cycle.

In Cycle 1, the content of PC is moved to MAR, which allows the address in MAR to point to the current instruction to be fetched from memory. This is achieved by having the control unit (CU) provide (1) PC_OUT_{enable} signal to the tri-state buffer, which causes the content of the PC to appear on the Internal Data Bus, and (2) MAR_{enable} signal to MAR. Thus, at the end of the clock cycle, the content of PC is latched onto MAR.

In Cycle 2, M[MAR] refers the memory location pointed to by the address in MAR. Therefore, the instruction in M[MAR] is read from memory and latched onto MDR. The control unit provides *Read* control signal to memory and MDR_{enable} signal to the MDR register. At same time PC is incremented

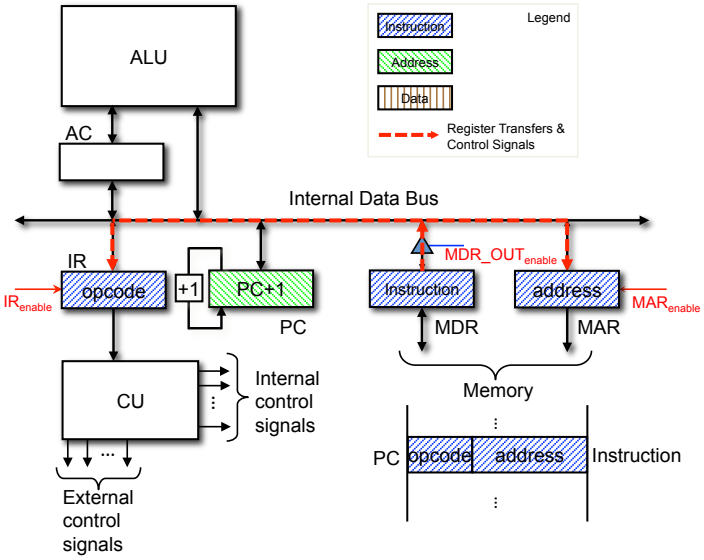


(a) Cycle 1: MAR←PC.



(b) Cycle 2: MDR←M[MAR], PC←PC+1

Figure 3.10: Fetch Cycle.



(c) Cycle 3: IR←MDR(opcode), MAR←MDR(address).

Figure 3.10: Fetch Cycle (cont.).

and latched (via PC_{enable}), which then points to instruction to be fetched and executed in the next instruction cycle. These two operations can be done concurrently because both reading from memory and incrementing PC does not require the use of the Internal data Bus, and thus does not conflict.

Although the instruction has been fetched into the processor by the end of Cycle 2, the processor does not know what this instruction is. Therefore, the opcode portion of the instruction is moved to IR in Cycle 3. In addition, the address portion of the instruction is moved to MAR. This sets up the MAR register to the point to the operand needed by the instruction. This is achieved by allowing the content of MDR to appear on the Internal Data Bus (MDR_OUT_{enable}) and enable the latching of IR (IR_{enable}) and MAR (MAR_{enable}). Note that not all instructions require an operand (e.g., branch instruction). However, performing this operation in the Fetch Cycle saves cycles in the Execute Cycle.

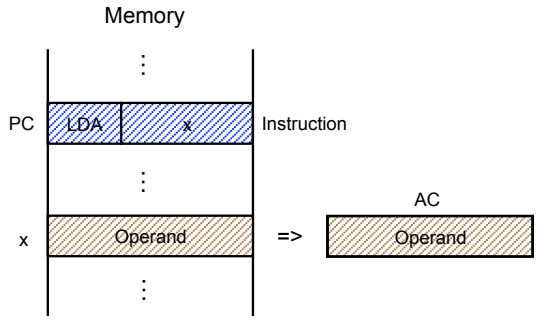


Figure 3.11: Operation of `LDA x`.

Execute Cycle

Unlike the Fetch Cycle, the Execute Cycle depends on the instruction. Thus, this section will discuss the execution of each instruction in the pseudo-ISA shown in Table 3.2, which was defined in Chapter 2.5.

Table 3.2: Instructions in the pseudo-ISA.

Category	Instruction	Description
Data transfer	`LDA x`	Load accumulator
	`STA x`	Store accumulator
Arithmetic & logic	`ADD x`	Add to accumulator
	`SUB x`	Subtract from accumulator
	`NAND x`	Logical NAND to accumulator
	`SHFT`	Shift accumulator
Control transfer	`J x`	Jump
	`BNE x`	Branch conditionally

The following example show the sequence of micro-operations required for `LDA x`.

Example 3.1. *Execute cycle for `LDA x`.*

Figure 3.11 illustrates the operation of `LDA x`. The effective address, `x`, points to a memory location that contains the operand to be loaded into AC. Note that the old content of the AC is overwritten. The `LDA x` instruction can be implemented by the following sequence of micro-operations:

1. MDR ← M[MAR]
2. AC ← MDR

Figure 3.12 illustrates the sequence of micro-operations for `LDA x`. After the fetch cycle, *MAR* contains the address of the operand, i.e., the effective address, to be read from memory. Therefore, in Cycle 1, the operand in memory location pointed to by *MAR*, i.e., M[MAR], is transferred to *MDR*. In Cycle 2, the operand in *MDR* is transferred to *AC*.

Example 3.2. *Execute cycle for `STA x`.*

Figure 3.13 illustrates the operation of `STA x`. The `STA x` instruction can be implemented by the following sequence of micro-operations:

1. MDR ← AC
2. M[MAR] ← MDR

Fig. 3.14 illustrates the execute cycle for `STA x`, which is basically an inverse operation of `LDA x`. In Cycle 1, the content of *AC* is transferred to *MDR*. In Cycle 2, the content of *MDR* is written to the memory location pointed *MAR*, which is the EA *x*. In Cycle 2, the operand in *MDR* is transferred to *AC*.

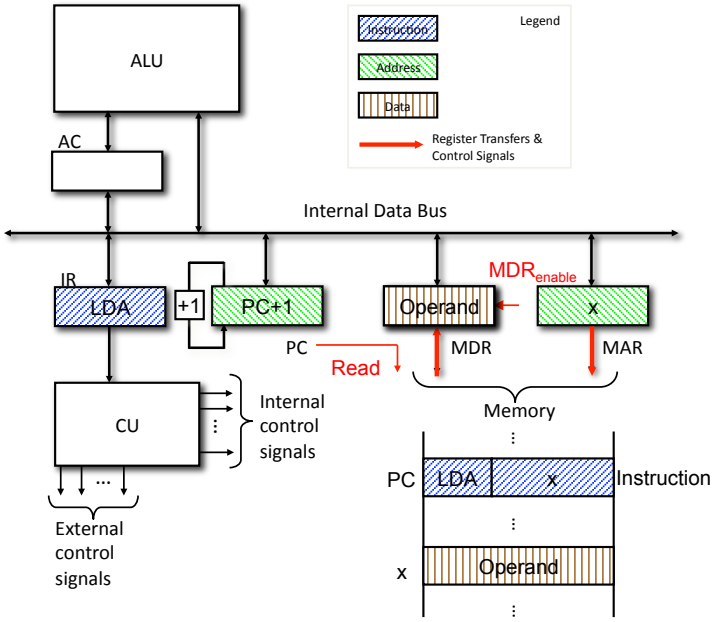
Example 3.3. *Execute cycle for `ADD x`.*

Figure 3.15 illustrates the operation of `ADD x`. The effective address `x` points to one of the operand (i.e., Operand1), while the other operand is contained in *AC*. The result of the add operation is stored back in *AC*. The `ADD x` instruction can be implemented by the following sequence of micro-operations:

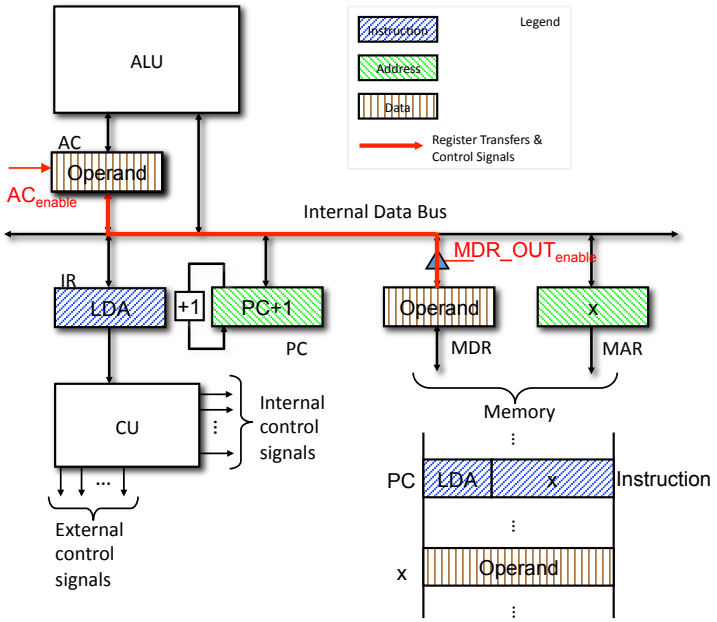
1. MDR ← M[MAR]
2. AC ← AC + MDR

Fig. 3.16 illustrates the execute cycle for `ADD x`. In Cycle 1, the operand in memory location pointed to by *MAR*, i.e., M[MAR], is transferred to *MDR*. In Cycle 2, the content of *MDR* is added with the content of *AC* (i.e., Operand2), and the result is latched to the *AC*.

Example 3.4. *Execute cycle for `SUB x`.*



(a) Cycle 1: MDR←M[MAR].



(b) Cycle 2: AC←MDR.

Figure 3.12: Microoperations for `LDA x`.

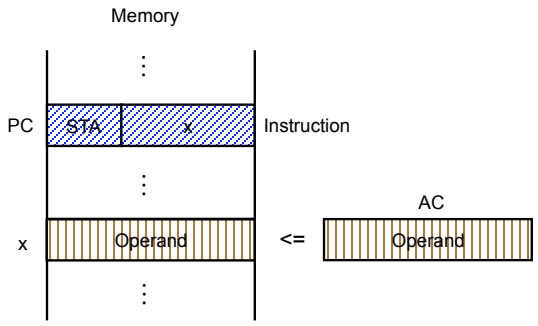


Figure 3.13: Operation of `STA x`.

`SUB x` is similar to `ADD x`, except the operation performed is subtract. The effective address `x` points to the operand that is subtracted from the operand contained in *AC*. The result of the subtract operation is stored back in *AC*. The `SUB x` instruction can be implemented by the following sequence of micro-operations:

1. MDR ← M[MAR]
2. AC ← AC - MDR

Example 3.5. *Execute cycle for `NAND x`.*

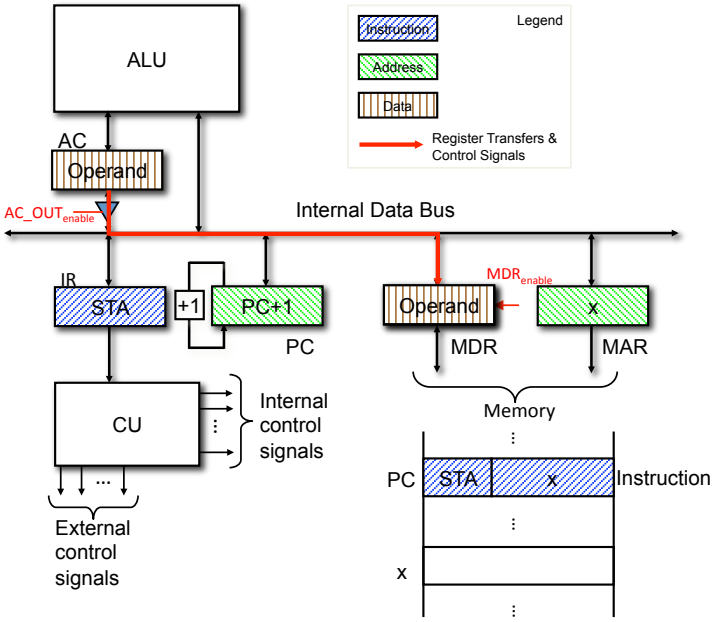
`NAND x` is also similar to `ADD x` and `SUB x`, except the operation performed is bit-wise logical NAND. The `NAND x` instruction can be implemented by the following sequence of micro-operations:

1. MDR ← M[MAR]
2. AC ← AC ∧ MDR

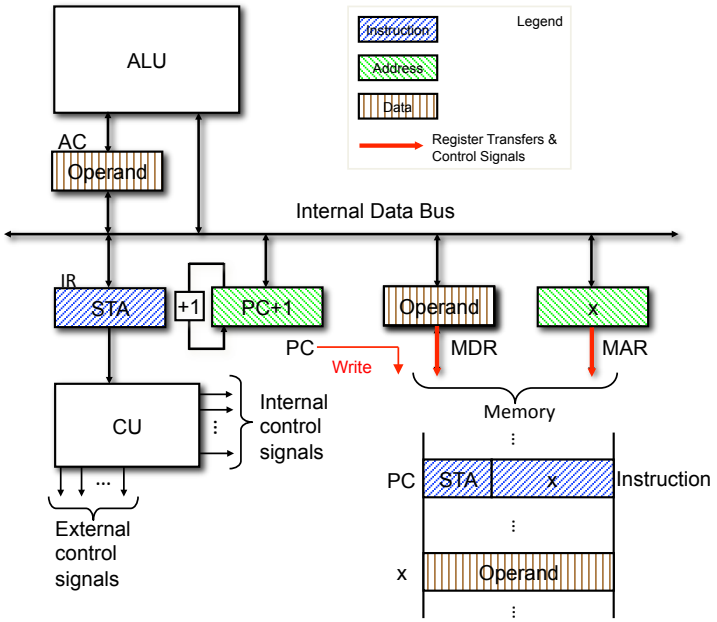
Example 3.6. *Execute cycle for `SHFT`.*

Unlike the three previous instructions, `SHFT` is a unary operation involving only the content of the *AC*. The `SHFT` instruction can be implemented by the following micro-operation, where *sl* means shift left by one bit:

1. AC ← *sl* AC



(a) Cycle 1: MDR ← AC.



(b) Cycle 2: M[MAR] ← MDR.

Figure 3.14: Microoperations for `STA x`.

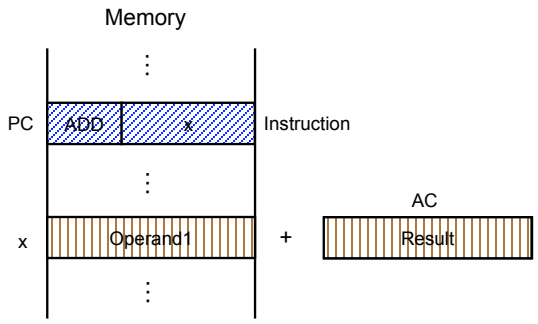


Figure 3.15: Operation of `ADD x`.

Note that the `SHFT` instruction can also be described as

1. AC(16...1) ← AC(15...0), AC(0) ← 0

Example 3.7. *Execute cycle for `J x`.*

Figure 3.17 illustrates the operation of `J x`. Address `x` represent the *target address* for the jump. Thus, the next instruction to be fetched from memory is contained in the location pointed to by `x`. The `J x` instruction can be implemented by the following micro-operation:

1. PC ← MDR(address)

or

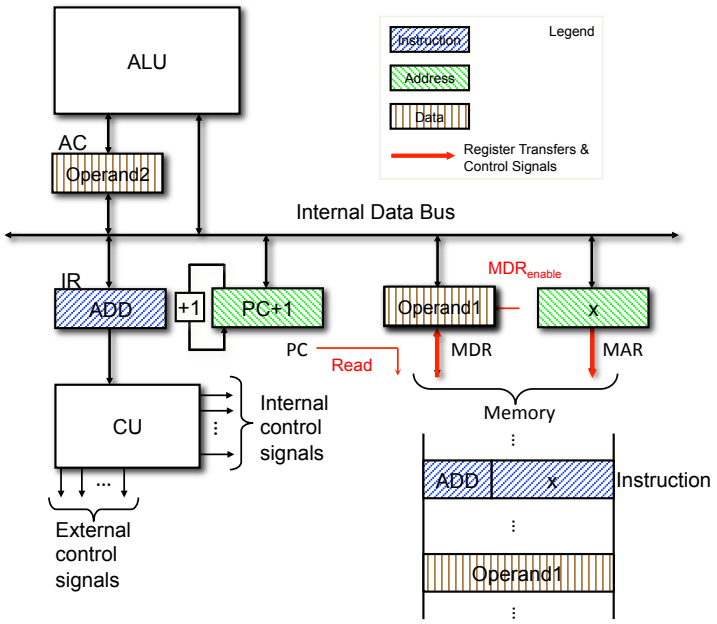
1. PC ← MAR

Fig. 3.18 illustrates the execute cycle for `J x`. The target address x, which is already in *MDR* after the fetch cycle, is transferred to the PC. This way, the fetch and execute cycle is performed on the *next instruction*.

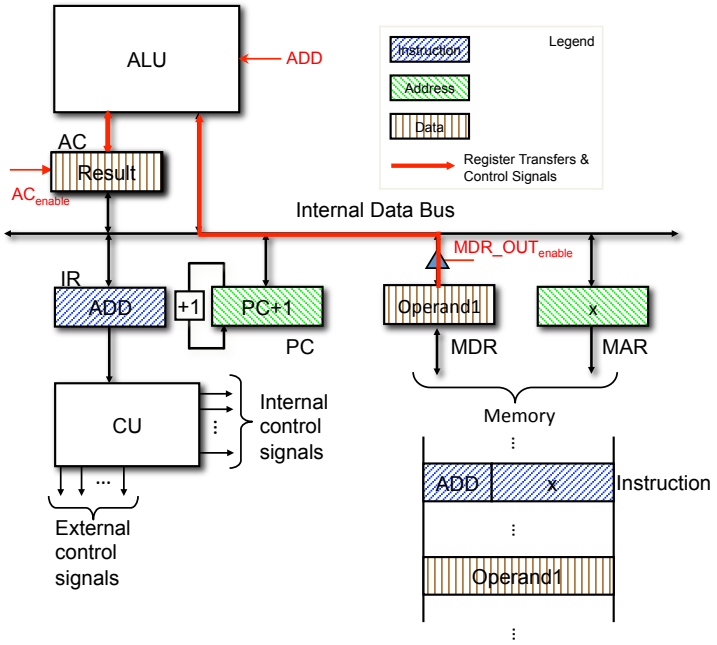
Example 3.8. *Execute cycle for `BNE x`.*

In contrast to `J`, which is an unconditional branch, `BNE x` is a *conditional branch*. The `BNE x` instruction can be implemented by the following micro-operation:

1. If (Z != 1) then PC ← MDR(address)



(a) Cycle 1: MDR ← M[MAR].



(b) Cycle 2: AC ← AC + MDR.

Figure 3.16: Microoperations for `ADD x`.

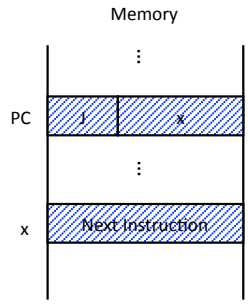


Figure 3.17: Operation of `J x`.

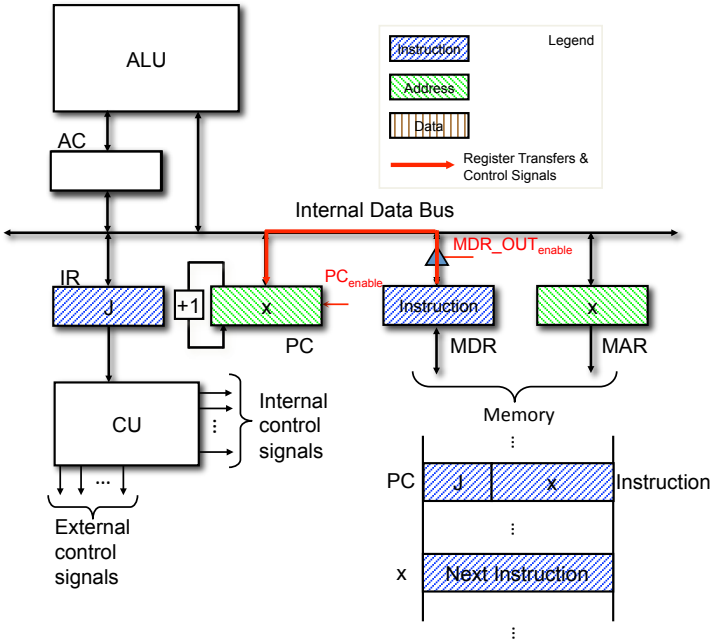


Figure 3.18: Microoperation for `J x`.

Fig. 3.19 illustrates the execute cycle for `BNE x`. Similar to unconditional branches, the address x serves as the *target address* for the branch. The `BNE` instruction tests the Z-flag (or Zero-flag), which is one of a number of *condition codes* generated by the ALU after every arithmetic operation.

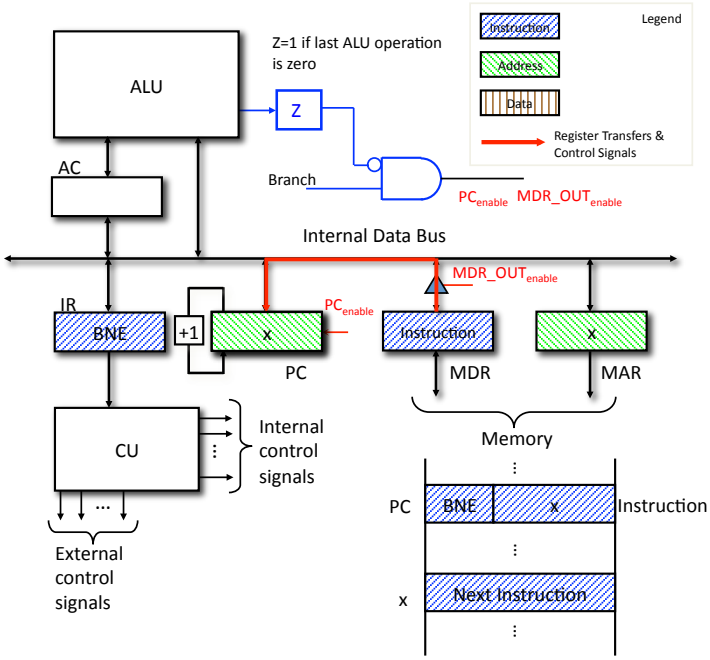


Figure 3.19: Microoperation for `BNE x`.

The other condition codes include negative (N), overflow (V), and carry (C) flags (see Chapter 9). If Z-flag is not set (i.e., the result is not zero) and the current instruction is branch, which is indicated by the *Branch* signal from the CU, the *address* field of the instruction in *MDR* is transferred to *PC*.

3.3.5 Extensions to the pseudo-ISA/CPU

We have discussed thus far how the pseudo-ISA can be implemented on the pseudo-CPU. The pseudo-CPU, despite its simplicity, is capable of much more. Thus, this subsection will discuss how the pseudo-ISA can be extended to support additional instructions as well as differing addressing modes.

Example 3.9. *Suppose we want to implement "load accumulator indirect" (`LDA (x)`).*

`LDA (x)` is an example of *indirect addressing mode*. Note that parentheses '(' and ')' distinguish indirect addressing from direct addressing. Addressing

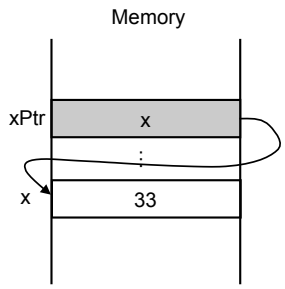


Figure 3.20: The concept of a pointer.

modes in assembly language provide different ways to access operands. We will discuss more about addressing modes in Chapter 4. But for now, indirect addressing mode provides a way to *indirectly* reference a value, also referred to as *indirection*, and thus implement *pointers*.

Pointers are variables that contain addresses as their values. For example, consider the following declaration in C/C++:

```
int main()
{     int x, *xPtr;

      x = 33;
      xPtr = &x;
}
```

The variable `x` is of type integer, and `*xPtr` represents a pointer to type `int`. The first statement assigns '33' to the variable `x`. In the second statement, the address of variable `x` is assigned to pointer variable `xPtr` (where `&` represents the address operator). Figure 3.20 shows the relationship between `x` and `xPtr` in memory. As can be seen from the figure, the value 33 is *directly* referenced by address `x`, while `xPtr` *indirectly* references the variable `x` whose value is 33.

The importance of pointers in programming is that they can be manipulated. For example, consider an array `A` of `n` elements, i.e., `A[n]`. Based on the above concept of pointers, `A[i]` is equivalent to `*(A+i)`. Therefore, we can perform arithmetic operations on `A` to index off of the first element of the array `A`, i.e., `A[0]`.

Now that we have discussed pointers and indirection, let us resume the discussion of `LDA (x)`. Figure 3.21 illustrates the operation of `LDA (x)`. As can be seen from the figure, the *Operand* is pointed to by the *EA* in

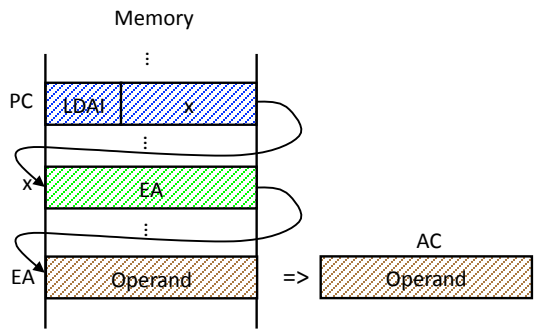


Figure 3.21: The concept of indirection.

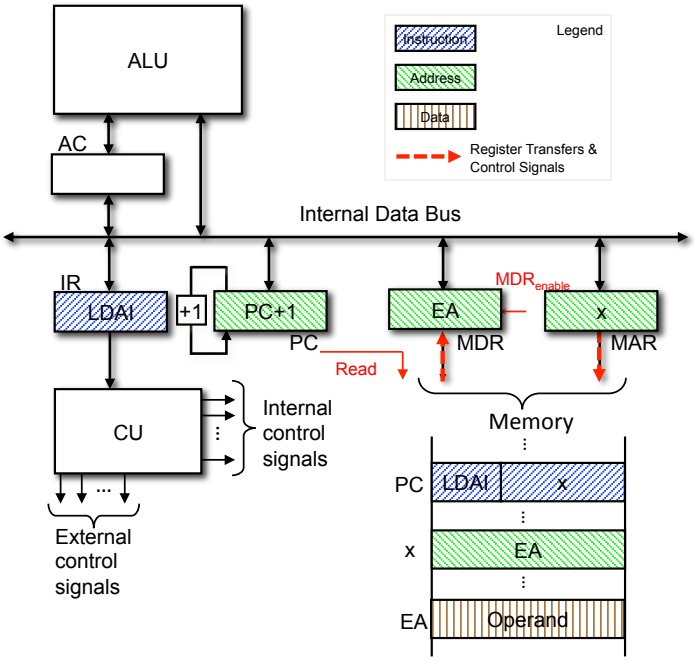
memory, which is in turn pointed to by address x in the instruction format. Therefore, the Execute cycle requires accessing the memory twice, first for *EA* and second for the *Operand*. The sequence of micro-operations for `LDA (x)` is shown below.

1. MDR ← M[MAR]
2. MAR ← MDR
3. MDR ← M[MAR]
4. AC ← MDR

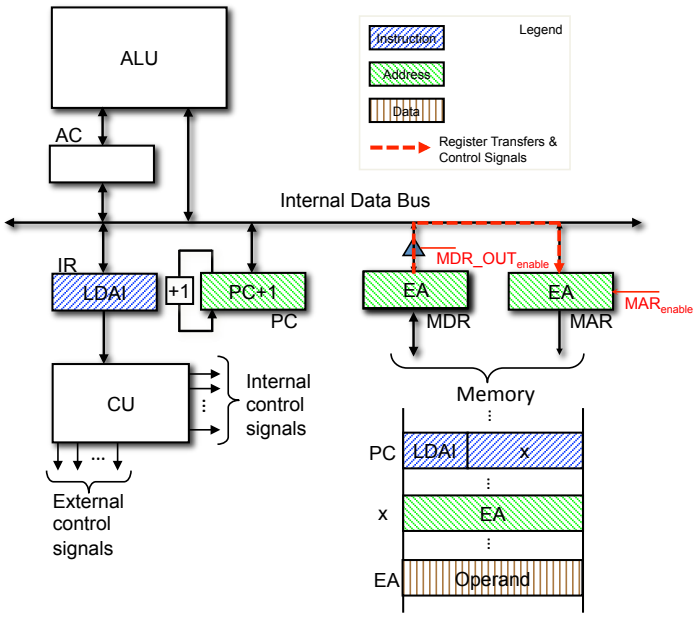
Figure 3.22 illustrates the sequence of micro-operations for `LDA (x)`. Cycle 1 involves accessing the memory location x to read in the *EA*. *EA* is then transferred to *MAR* in Cycle 2 so that it can be used to read in *Operand* in Cycle 3. Finally, *Operand* in *MDR* transferred to *AC* in Cycle 4.

Example 3.10. *Suppose we want to implement "load accumulator indirect with pre-decrement", `LDA -(x)`.*

The operations of `LDA -(x)` is illustrated in Figure 3.23. This instruction is similar to `LDA (x)`, except the content of memory location pointed by x, i.e., EA+1, is decremented and stored back *before Operand* is read from memory. Load accumulator indirect with pre-decrement is another useful addressing mode that can be used, for example, to access elements of an array one by one.

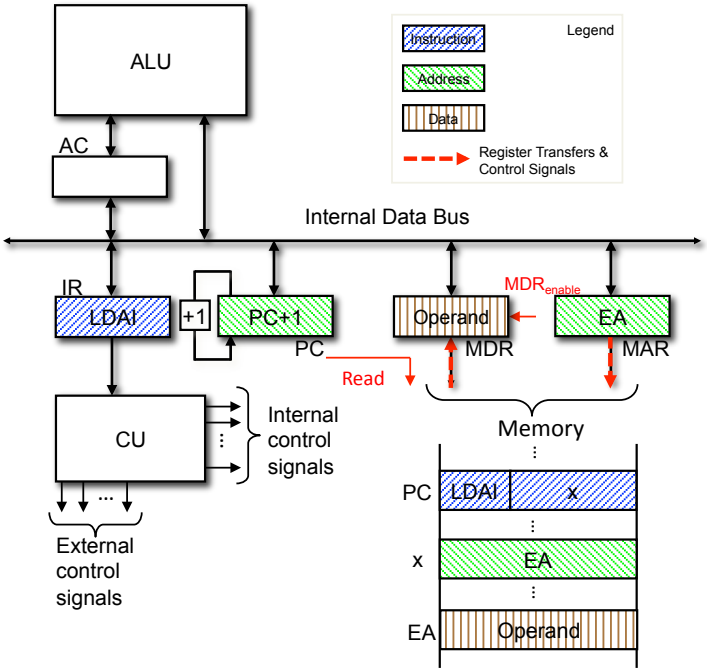


(a) Cycle 1: MDR ← M[MAR].

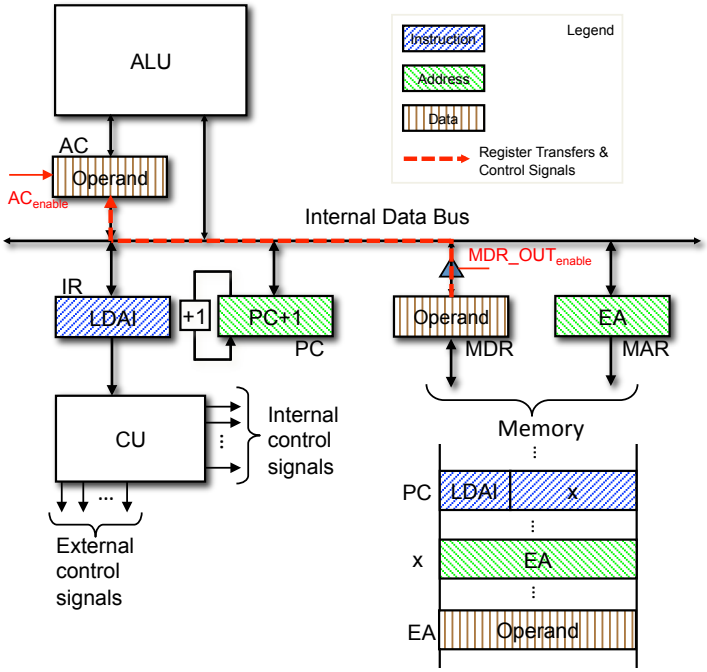


(b) Cycle 2: MAR ← MDR.

Figure 3.22: LDA indirect.



(c) Cycle 3: MDR ← M[MAR].



(d) Cycle 4: AC ← AC + MDR.

Figure 3.22: LDA indirect (cont.).

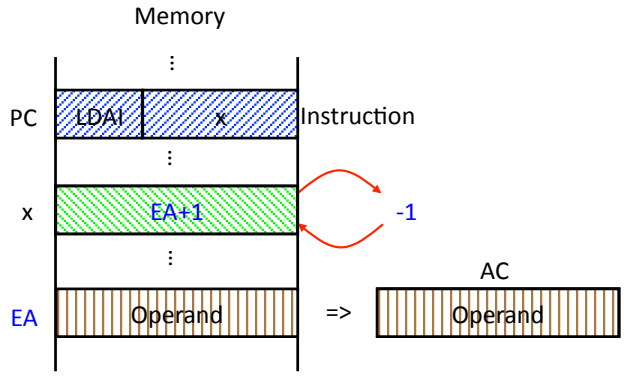


Figure 3.23: LDA indirect with pre-decrement.

This example illustrates different design choices that can be made and how they affect the microarchitecture and performance. There are a couple of ways to include the predecrement capability into the pseudo-CPU shown in Figure. 3.7. In the first method, a decrementer is integrated into the *MDR*, much like the *PC*. This allows $EA+1$ to be decremented directly in *MDR* and simplifies the overall design. Figure 3.24 shows such a modification. Based on this design, the sequence of micro-operations for the Execute Cycle is shown below.

1. MDR ← M[MAR] ; Read effective address (EA)
2. MDR ← MDR - 1 ; Decrement EA+1
3. M[MAR] ← MDR ; Store it back in x
4. MAR ← MDR ;
5. MDR ← M[MAR] ; Read operand
6. AC ← MDR ; Transfer operand to AC

As can be see, Steps 1 and 4-5 are identical the `LDA (x)` instruction. The only additional micro-operations required are decrementing $EA+1$ in Step 2 and storing it back into memory location pointed by x. Note that Steps 4 and 5 can be done in the same cycle. This is because all of the registers are clocked, and thus moving the content of MDR into MAR occurs at the end of the clock cycle and does not affect the *MAR* during writing of *MDR*

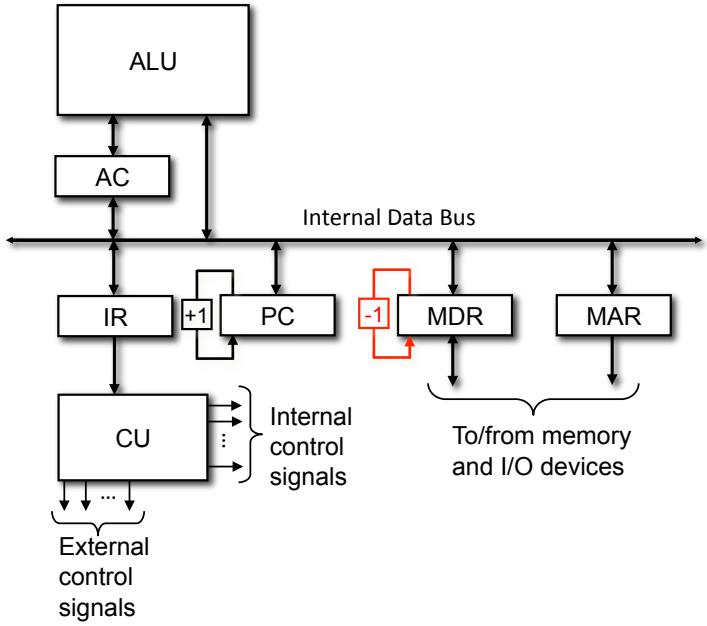


Figure 3.24: Simple CPU decrement capability with MDR.

into memory during the same cycle. Based on this, the optimized sequence of microoperations is shown below:

1. MDR ← M[MAR] ; Read effective address (EA)
2. MDR ← MDR - 1 ; Decrement EA+1
3. M[MAR] ← MDR, MAR ← MDR ; Store it back in x ;
4. MDR ← M[MAR] ; Read operand
5. AC ← MDR ; Transfer operand to AC

One of the problems with the first method is that an n-bit adder hardware is needed to implement the decrementer. As will be seen in Chapter 9, an adder requires a significant amount of logic and may not be worth the investment in hardware depending on how often indirect with pre-decrement addressing modes are encountered. Therefore, in the second method, we will use AC to perform pre-decrement since the ALU already has this capability.

Based on this design, the sequence of micro-operations for the Execute Cycle is shown below:

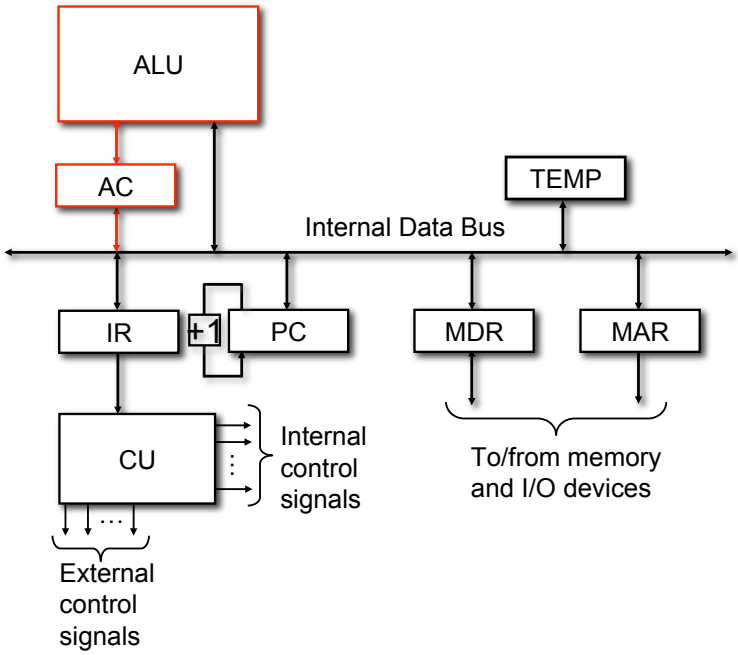


Figure 3.25: Pseudo-CPU with a temporary register.

1. MDR ← M[MAR] ; Read effective address (EA)
2. AC ← MDR
3. AC ← AC - 1 ; Decrement EA
4. MDR ← AC
5. M[MAR] ← MDR, MAR ← MDR ; Store it back in x
6. MDR ← M[MAR] ; Read operand
7. AC ← MDR ; Transfer operand to AC

Cycles 2-3 decrement $EA+1$, and cycles 4-5 store EA back into the memory location pointed to by x. As can be seen, this implementation requires 7 versus 5 cycles for the design with the dedicated decrementer, which results in longer delay but less amount of hardware.

Chapter 4

AVR Assembly Language

4.1 Why Microcontrollers?

A *microcontroller* is a small computer system optimized for hardware control that encapsulates the processor core, memory, clock, timers, and I/O ports on a single piece of silicon. Unlike microprocessors in a personal computer, microcontrollers are designed for small, dedicated applications. For example, they are used to automatically control devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, and toys. By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control hardware devices.

Microcontrollers are optimized for hardware control in the sense that their Instruction Set Architectures (ISAs) provide a set of instructions to easily control I/O devices. These instructions include writing/reading to/from as well as setting, clearing, and testing each bit of a parallel I/O register or *port*.

The majority of microcontrollers in use today are embedded in other machinery, such as automobiles, telephones, appliances, and peripherals for computer systems. Such system are called *embedded systems*. While some embedded systems are very sophisticated, many have small memory and program size, with no operating system, and low software complexity. Typical I/O devices they control include switches, relays, solenoids, LEDs, small or custom LCD displays, radio frequency devices, and sensors for temperature, humidity, light level, etc. Embedded systems usually have no keyboard, screen, disks, printers, or other recognizable I/O devices of a personal com-

CMP
United Business Media

58

2006 State of Embedded Market Survey

Consideration of 8-bit chip families

Vendor	2006 %	2005 %	Vendor	2006 %	2005 %
Atmel AVR	32	33	Rabbit 2000, 3000	11	16
Microchip PIC 18	30	31	Dallas/Maxim 80xx	10	15
Freescale HC05, HC08, HC11	27	25	Philips P80x, P87x, P89x	10	11
Microchip PIC 14/16	26	28	Renesas	10	11
Intel 80xx, '251	21	22	Cypress PSoC	9	11
Microchip PIC 10/12	17	21	Cygnal/SiLabs 80xx	8	11
Zilog Z8, Z80, Z180, eZ80	16	19	NEC K0	5	5
Microchip PIC 17	15	18	Infineon C500	3	6
Atmel 80xx	13	16	National COP8	3	4
TI TMS370, 7000	13	14	Ubicom SX	1	5
STMicro ST6, ST7, ST8	12	11	Other	2	3
Xilinx PicoBlaze	12	14			

EETIMES
Embedded Systems Design

53. Which of the following 8-bit chip families would you consider for your next embedded project?

N = 662

Figure 4.1: The most popular 8-bit processor. Source: EETimes and Embedded Systems Design Magazine 2006 Embedded Market Survey

puter (PC), and may lack human interaction devices of any kind.

Although microprocessors for PCs get most of the spotlight, microcontrollers are more prevalent and ubiquitous in our lives. For example, the ratio of embedded devices to PCs is greater than 100. A typical house may contain over 50 embedded processors. A high-end car can have over 80 embedded processors. In fact, embedded systems account for the most of the worlds production of microprocessors!

This chapter presents assembly programming for one of the most widely used embedded processors, Atmel *AVR 8-bit microcontrollers* (see Figure 4.1). AVR consists of a family of processors with different Program and Data Memory sizes, number of I/O ports, clock speed, and data sizes for a variety of embedded solutions. Figure 4.2 show some examples of AVR-based products.

4.2 General Characteristics

Our discussion of the AVR architecture in this chapter will be based on ATmega128. Fig. 4.3 shows the block diagram of the ATmega128 micro-

Figure 4.2: Some AVR-based products.

controller. It consists of a CPU core, I/O ports, and extensive peripheral features, such as timers/counters, analog-to-digital converter (ADC), serial universal synchronous/asynchronous receiver/transmitter (USARTs), serial interface, etc.

4.2.1 Program and Data Memories

The AVR architecture has separate program and data memories. This is shown in Figure 4.4. The size of the *Program Memory* for ATmega128 is 64K (K=1,024) memory words, where the size of each memory word is equal to the instruction format length of 16 bits. Therefore, the Program Memory size is 128 Kbytes. The Program Memory is implemented using non-volatile flash memory, which retains its contents even after the power is turned off. The *Data Memory* contains 256 bytes of *General Purpose Registers* (GPRs) (see Section 4.3) and I/O registers plus 4 Kbytes internal *Static RAM* (SRAM). The Data Memory is also expandable up 64 Kbytes using external memory.

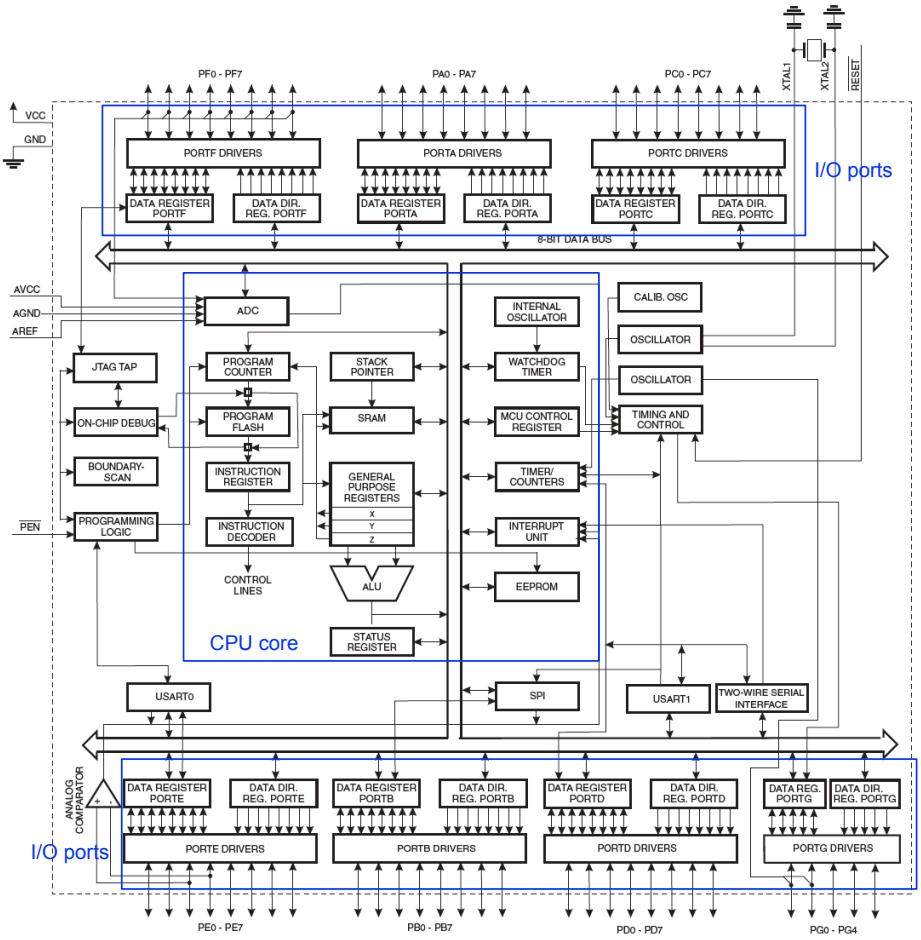


Figure 4.3: The block diagram of Atmega128.

4.3 Registers and Memory

As discussed in Chapter 2, one of the most important aspects of assembly programming is knowing where the data to be manipulated is located in registers and memories. The following two subsections discuss the registers and memories available in the AVR architecture.

4.3.1 Registers

The AVR architecture has the following set of registers:

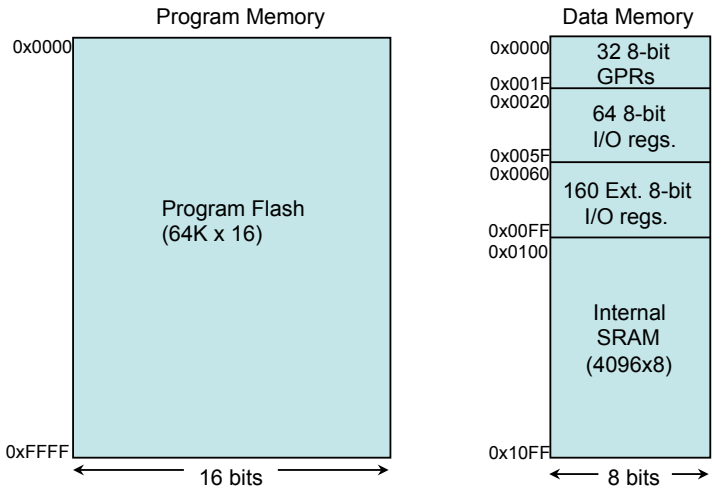


Figure 4.4: AVR memory organization.

- 32 8-bit General Purpose Registers (GPRs)
- 16-bit X-, Y-, and Z-register
- Program Counter (PC)
- 16-bit Stack Pointer (SP)
- 8-bit Status Register (SREG)
- 7 8-bit I/O registers (ports)

Figure 4.5 shows *General Purpose Registers* (GPRs), which serve as a small storage space used by the processor to quickly access and perform operations on both data and addresses. The 32 GPRs are referenced in assembly programs as `R31-R0`.

X-register, *Y-register*, and *Z-register* are *address registers* used as pointers to Data Memory address space. One of these registers, Z-register, can also be used to access Program Memory address space. These address registers are also mapped to the last six registers of GPRs (see Figure 4.5). That is, X-register is mapped to the register pair `R27:R26`, Y-register is mapped to the register pair `R29:R28`, and Z-register is mapped to the register pair `R31:R30`. Therefore, address registers are 16-bit wide allowing all of the Data Memory space (64 Kbytes) to be accessed.

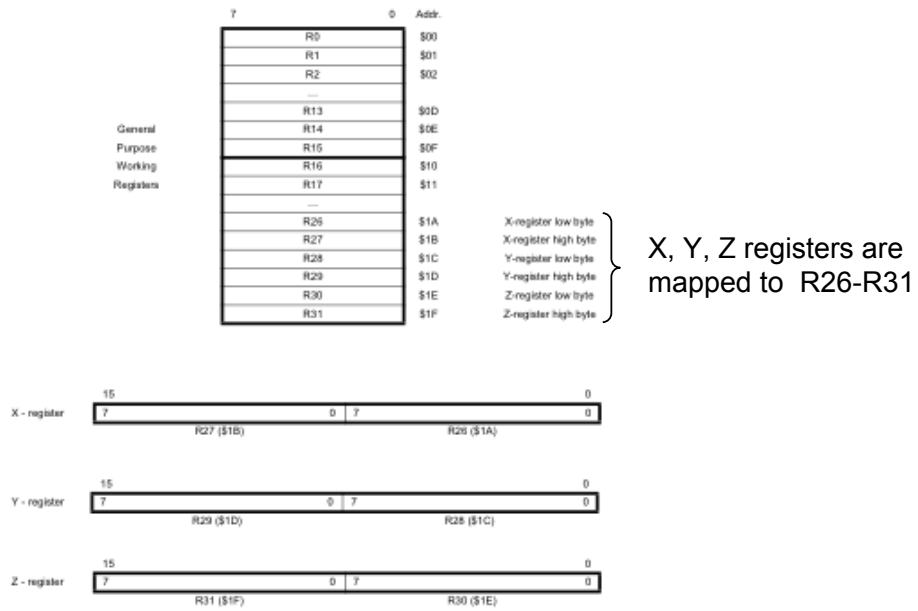


Figure 4.5: AVR GPRs.

Program Counter (PC) is a special register that points to either the current instruction being executed or the next instruction to be executed. PC is 16-bit wide so that the entire 64K memory words in the Program Memory can be accessed.

Stack Pointer (SP) is used to point to the top of the stack. A stack is a data structure that implements *last-in, first-out* (LIFO) function. For example, a stack is used to store information about the active subroutines of a program.

Status Register (SREG) contains a collection of *condition codes*, or flags, to indicate the current status of the processor. The format of SREG is shown in Fig 4.6 and contains the following flags.

- I - Global Interrupt Enable
- T - Bit Copy Storage
- H - Half Carry Flag
- S - Sign Bit
- V - 2s Complement Overflow Flag

I	T	H	S	V	N	Z	C

Figure 4.6: Status Register.

- N - Negative Flag
- Z - Zero Flag
- C - Carry Flag

These flags are set/reset according to the outcome of an instruction execution.

There are also I/O registers that are used to perform I/O operations. Chapter 5 will provide a detailed discussion of the I/O capabilities in the AVR architecture.

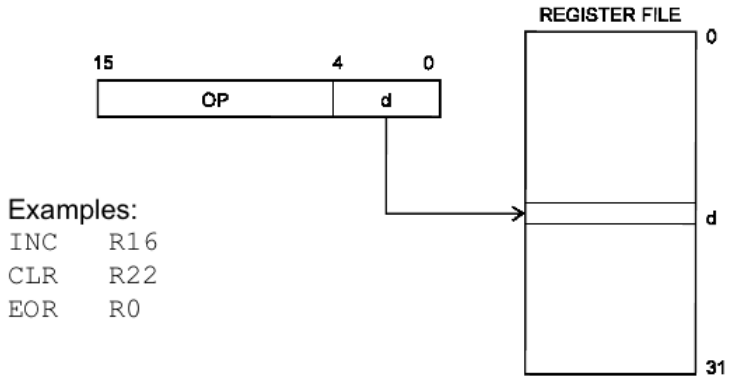
4.4 Addressing Modes

Addressing Modes define the way operands are accessed. Different addressing modes gives programmers flexibility to implement pointers to memory, counters for loop control, indexing of data, and program relocation.

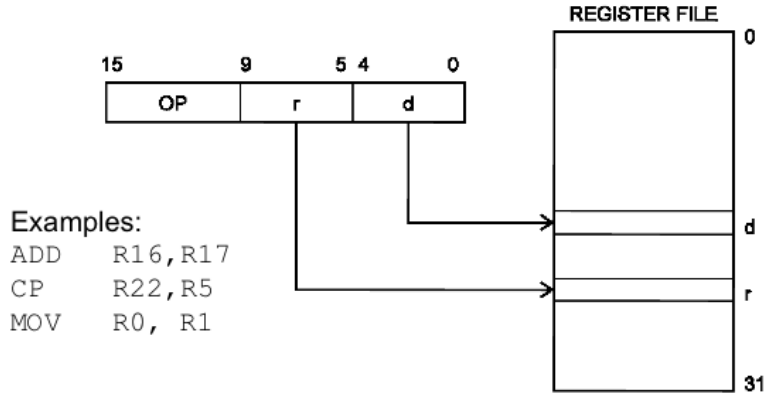
There are several addressing modes provided in AVR:

- Register (with 1 and 2 registers)
- Direct
- Indirect
  - w/Displacement
  - Pre-decrement
  - Post-increment
- Relative

Fig. 4.7 shows examples of *register addressing* modes. These addressing modes are used to manipulate data in GPRs. Fig. 4.7(a) illustrates operations on a single operand. For example, AVR assembly instructions `INC` (*Increment*) and `CLR` (*Clear*) only require a single register. `EOR` (*Exclusive OR*) is a Boolean operation that typically requires two operands, however, it can also be used with only one register with the implication that the other



(a) Register addressing with one register.

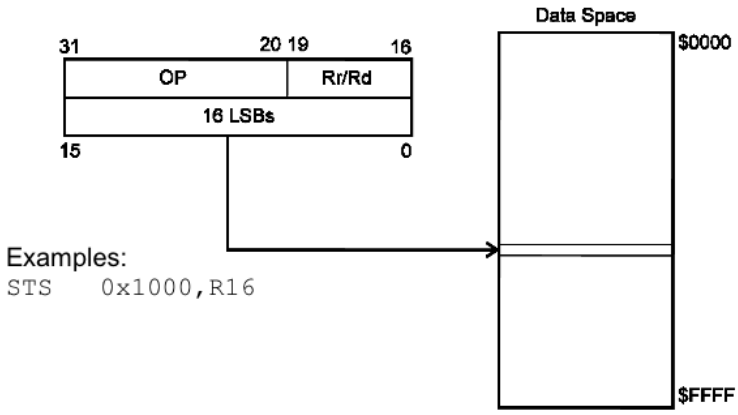


(b) Register addressing with two registers.

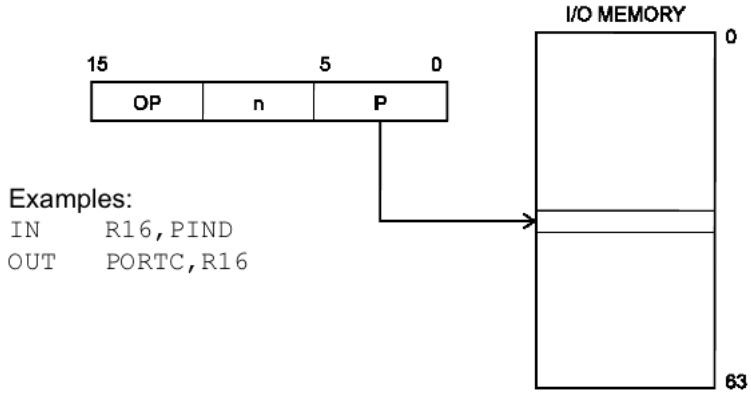
Figure 4.7: Register Addressing mode.

registers is the same register. Thus, `EOR R0` and `EOR R0,R0` are equivalent and both of these instructions clear register `R0`. Figure 4.7(b) illustrates operations on two operands. The AVR ISA uses a 2-address instruction format. Therefore, both left and right (or first and second) registers serve as input operands and the result of an operation is stored in the left (or first) register, which is consistent with assignment statements in high-level languages.

Besides register addressing, the rest of the addressing modes provides different ways to access data in either Data Memory or Program Memory.



(a) Direct addressing.

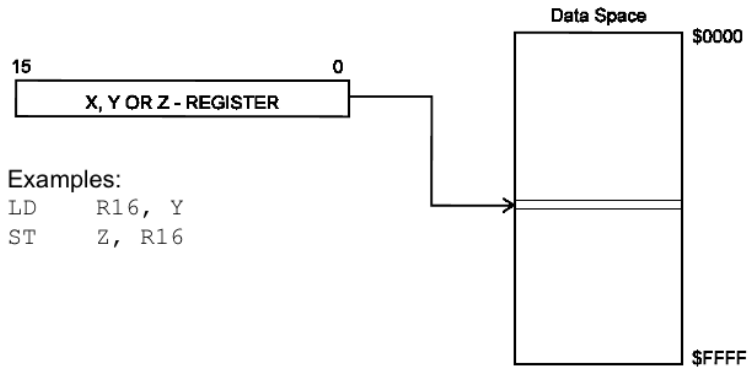


(b) I/O direct addressing.

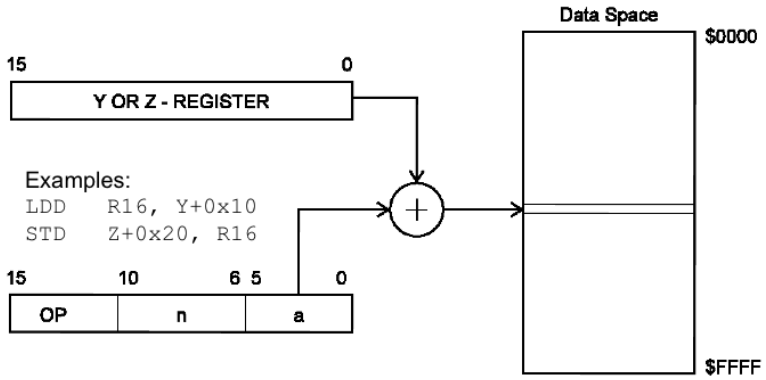
Figure 4.8: Direct addressing modes.

Fig. 4.8 shows examples of *direct addressing* mode. Figure 4.8(a) illustrates instructions that access a word in Data Memory. The instruction format of these instructions is 32 bits and is stored in two consecutive 16-bit words in Program Memory, with the second 16-bit word representing *effective address*, or the address of the operand. This allows instructions to directly access all 2^{16}=64K words in Data Memory.

I/O operations also use direct addressing. However, since there are only 64 I/O registers mapped to the Data Memory space, $log_2 64 = 6$ bits for the address is directly included in the instruction format. For the extended I/O



(a) Indirect addressing.



(b) Indirect addressing with displacement.

Figure 4.9: Indirect addressing modes.

registers, instructions with direct addressing capability, such as `LDS` and `STS` will have to be used.

It is obvious from Figure 4.8(a) that frequent use of direct addressing increases code size. Therefore, *indirect addressing* allows an effective address to be put into an address registers X, Y, or Z, and implements these instructions with 16-bit instruction format. Figure 4.9 shows examples of indirect addressing mode. The most common way to use indirect addressing is with `LD` (*load indirect*) and `ST` (*store indirect*) instructions. This is illustrated in Fig. 4.9(a).

A variation of indirect addressing is to allow for *displacements*. Fig. 4.9(b)

illustrates `LDD` (*load indirect with displacement*) and `STD` (*store indirect with displacement*) instructions. These instructions are useful for accessing data structures. For example, an address register acts as based pointer for a structure, and displacement would represent an offset to a member of the data structure.

Another variation of indirect addressing is to have *pre-decrement* and *post-increment* capabilities. This is shown in Fig. 4.10. As the name suggests, pre-decrement decrements an address register and then is used to access an operand. On the other hand, post-increment uses an address register to access operand and then the address register is incremented. Pre-decrement and post-increment are useful for accessing array elements one-by-one (either first to last or last to first) without having to separately increment address registers.

The addressing modes discussed thus far deal with how Data Memory is accessed. There are also addressing modes that dictate how *constants* or *immediate values* are accessed in Program Memory and PC is updated to implement flow of control.

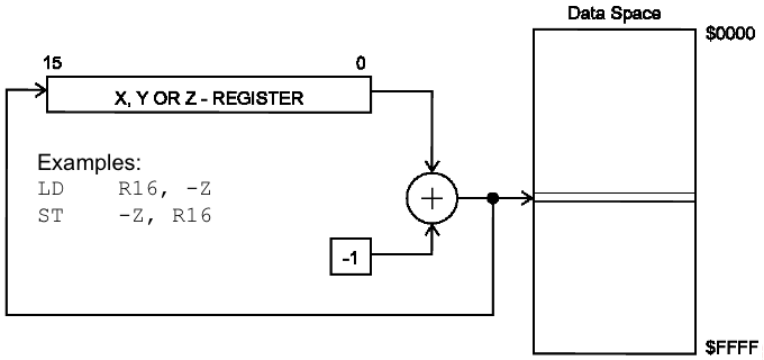
Fig. 4.11 illustrates the *Program Memory Constant addressing*. Either 8-bit or 16-bit constants stored in the Program Memory can be accessed using the `LPM` (*load Program Memory*) instruction.

Fig. 4.12 shows *Program Memory addressing*, which affect how *target address* is generated for *jump* (`JMP` and `IJMP`) and *subroutine call* (`CALL` and `ICALL`) instructions. Target address for instructions that use *direct Program Memory addressing* are 32 bits long with the second 16 bits representing the target address. In contrast, *indirect Program Memory addressing* uses the Z-register to hold the target address.

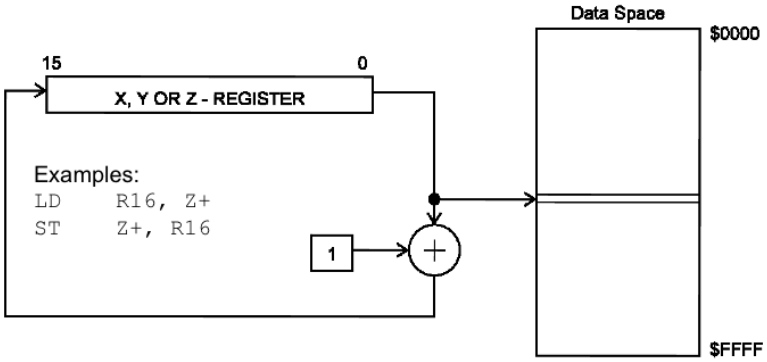
Fig. 4.13 shows *Relative Program Memory addressing*. This addressing mode uses an address, which is one more than the PC value for the current instruction (i.e., PC+1), and adds the signed 12-bit *displacement* to generate the target address. The two AVR assembly instructions that use this addressing mode is `RJMP` (*relative jump*) and `RCALL` (*relative call*). This type of instructions is also refer to as *PC-relative* jump. The signed 12-bit displacement is represented as a two's complement number and allows the displacement to be between $-2^{11} = -2,048$ and $2^{11} - 1 = 2,047$.

4.5 Instructions

AVR has 133 different instructions. These instructions fall into four categories; (1) Data Transfer, (2) Arithmetic and Logic, (3) Control Transfer



(a) Indirect addressing with pre-decrement.



(b) Indirect addressing with post-increment.

Figure 4.10: Indirect addressing modes with pre-decrement and post-increment.

(branch/jump), and (4) Bit and Bit-test.

4.5.1 Data Transfer Instructions

The majority of instructions in any assembly program is *data transfer* instructions. These instructions essentially move data from one area in memory to another or from memory to GPRs, and vice versa. Moreover, location of data to be moved is dictated by the addressing modes.

The `MOV` instruction transfers data between two registers, and uses the following syntax:

- `MOV` *dest*, *src* (*Move between registers*)

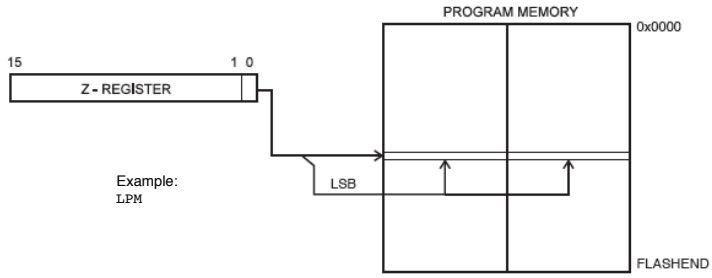
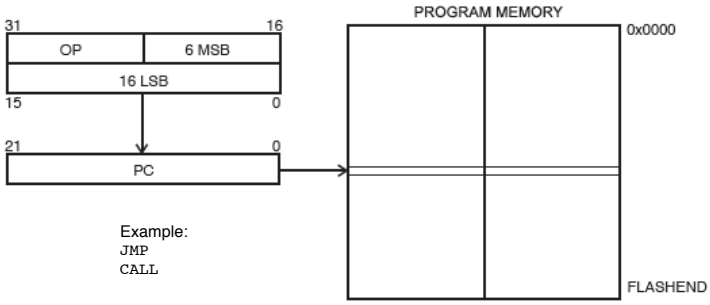
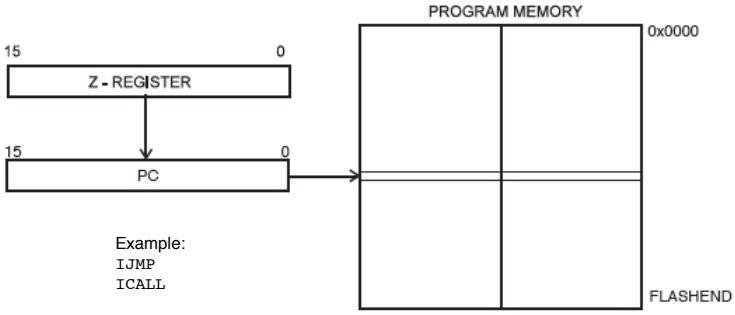


Figure 4.11: Program memory constant addressing.



(a) Direct Program Memory addressing.



(b) Indirect Program Memory addressing.

Figure 4.12: Program memory addressing modes.

- $dest = \texttt{Rd}\ (0 \leq \texttt{d} \leq 31)$
- $src = \texttt{Rr}\ (0 \leq \texttt{r} \leq 31)$

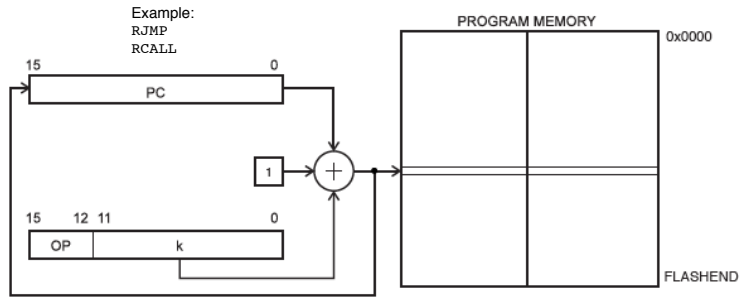


Figure 4.13: Relative Program Memory constant addressing.

Both `Rr` and `Rd` can be any one of the 32 GPRs. This is a copy operation, thus the content of `Rr` is not destroyed. This instruction is an example of register addressing.

The `LD` instruction loads either 8-bit data from Data Memory, and uses the following syntax:

- `LD` *dest*, *src* (*Load Indirect*)
  - *dest* = `Rd` ($0 \leq$ `d` $\leq$ `31`)
  - *src* = `X`, `X+`, `-X`, `Y`, `Y+`, `Y`, `Y+`, `-Y`, `Z`, `Z+`, `-Z`

There are several variations of the load instruction:

- `LDD` *dest*, *src* (*Load Indirect with Displacement*)
  - *dest* = `Rd` ($0 \leq$ `d` $\leq$ `31`)
  - *src* = `Y+`displacement, `Z+`displacement
- `LDI Rd`, *immediate* (*Load Immediate*)
  - *dest* = `Rd` ($16 \leq$ `d` $\leq$ `31`)
  - *immediate* can be
    - Binary → `0b00001010`
    - Decimal → `10`
    - Hexadecimal → `0x0A`, `$0A`
- `LDS` *dst*, *src* (*Load Direct from Data Memory*)
  - *dest* = `Rd` ($0 \leq$ `d` $\leq$ `31`)

- *src* = `k` (16-bit addresss)

`LDI` is simply a way to move an immediate (or constant) value into a register. Only `LDI` instruction supports immediate addressing. Also note that the destination register (`Rd`) for this instruction must be in the upper 16 GPRs (`R31-R16`).

The following example loop shows how `LDI` can be used. Suppose the loop executes 16 iterations. This can be implemented by loading a register with the value 16 and then decrementing it after each iteration. When the register value reaches zero, the program exits the loop. Since the loop count 16 is a constant, this value can be loaded into a register using immediate addressing. The following code demonstrates this example.

```
      ldi   r22, 16     ; Load  immediate value 16 into r22
Loop:
      breq  Exit        ; If zero, exit loop
      adc   r0, r1      ; Do something
      dec   r22         ; Decrement r22
      rjmp  Loop        ; Jump to loop
Exit:
      inc   r0          ; Continue on with program
```

Note that `16` is a decimal number. Replacing `16` with `$10` or `0x10` would be equivalent.

`LDS` uses direct addressing to move 8-bit data from Data Memory to a GPR. Direct addressing requires 16-bit address encoded in the second half of the instruction format to access the data.

`ST` instruction stores data to Data Memory, and uses the following syntax.

- `ST` *dest*, *src* (*Store Indirect*)
  - *src* = `Rr` ($0 \leq$ `d` $\leq$ `31`)
  - *dst* = `X`, `X+`, `-X`, `Y`, `Y+`, `Y`, `Y+`, `-Y`, `Z`, `Z+`, `-Z`

There are a couple of variations of the store instruction:

- `STD` *dest*, *src* (*Store Indirect with Displacement*)
  - *src* = `Rr` ($0 \leq$ `d` $\leq$ `31`)
  - *dst* = `Y+`displacement, `Z+`displacement
- `STS` *dst*, *src* (*Store Direct to Data Memory*)

  - *dest* = `k` (16-bit address)
  - *src* = `Rd` ($0 \leq$ `d` ≤ 31)

The last category of data transfer instructions is `LPM` (*Load from Program Memory*):

- `LPM` *dest*, *src*
  - *dest* = `Rd` ($0 \leq$ `d` ≤ 31)
  - *src* = `Z`, `Z+`
- `LPM`
  - *dest* = `R0` (implied)
  - *src* = `Z` (implied)

The `LPM` instruction is different from load instructions in that it is used to access Program Memory rather than Data Memory.

`PUSH` and `POP` instructions push and pop data to and from the stack, respectively, and use the following syntax:

- `PUSH` *src*
  - *src* = `Rr` ($0 \leq$ `r` ≤ 31)
- `POP` *dest*
  - *dest* = `Rd` ($0 \leq$ `d` ≤ 31)

The discussion on stack operations requires a special treatment. A stack implements last-in, first-out (LIFO) and is one of the most important data structures in computer science. For example, many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low-level code. Stacks are also used to support subroutine calls and returns and parameter passing. Even calculators that use reverse polish notation (RPN) rely on a stack.

The following code demonstrates how an immediate value $32 is pushed onto the stack:

```
ldi   r22, $32          ; Load the immediate value $32 into r22
push  r22
```

Table 4.1: Common Instruction Nomenclature.

Ending Letter	Meaning	Description
C	Carry	The operation involves the carry (C) bit
I	Immediate	The operation involves an immediate value that is passed as the second argument.
W	Word	The operation is a 16-bit operation.
S	Signed	The operation handles signed numbers
S/U	Signed/Unsigned	The operation handles both signed and unsigned.

4.5.2 Arithmetic and Logic Instructions

Almost all of the *arithmetic and logic* instructions operate on two 8-bit operands and can modify all of the condition codes in the `SREG`. The following is a breakdown of the available instructions:

- Addition: `ADD`, `ADC`, `ADIW`
- Subtraction: `SUB`, `SUBI`, `SBC`, `SBCI`, `SBIW`
- Logic: `AND`, `ANDI`, `OR`, `ORI`, `EOR`
- Compliment: `COM`, `NEG`
- Register Bit Manipulation: `SBR`, `CBR`
- Register Manipulation: `INC`, `DEC`, `TST`, `CLR`, `SER`
- Multiplication: `MUL`, `MULS`, `MULSU`
- Fractional Multiplication: `FMUL`, `FMULS`, `FMULSU`

Table 4.1 explains the common nomenclature to the naming of the instructions.

4.5.3 Control Transfer Instructions

Control transfer instructions are used to implement logical decisions and flow of control within a program. Typically, around 20% of the instructions in a program are control transfer instructions. A control transfer instruction a basically an instruction modifies the Program Counter (PC) and redirect where the next instruction is fetched. Without these instructions it would not be possible to implement `if` − `then` − `else`, `case` statements, and subroutine calls/returns in high-level langauges. There are two types of control transfer instructions; conditional and unconditional branches.

Table 4.2: Conditional Branch Summary. [1]Interchange Rd and Rr in the operation before the test, i.e., CP Rd,Rr ? CP Rr,Rd

Test	Boolean	Mnemonic	Complementary	Boolean	Mnemonic	Comment
Rd>Rr	Z·(N⊕V)=0	BRLT(1)	Rd≤ Rr	Z+(N⊕V)=1	BRGE*	Signed
Rd≥Rr	(N⊕V)=0	BRGE	Rd<Rr	(N⊕V) = 1	BRLT	Signed
Rd= Rr	Z=1	BREQ	Rd≠Rr	Z=0	BRNE	Signed
Rd≤Rr	Z+(N⊕V)=1	BRGE(1)	Rd>Rr	Z·(N⊕V)=0	BRLT*	Signed
Rd<Rr	(N⊕V)=1	BRLT	Rd≥Rr	(N⊕V)=0	BRGE	Signed
Rd>Rr	C+Z=0	BRLO(1)	Rd≤Rr	C+Z=1	BRSH*	Unsigned
Rd≥Rr	C=0	BRSH/BRCC	Rd<Rr	C=1	BRLO/BRCS	Unsigned
Rd=Rr	Z=1	BREQ	Rd≠Rr	Z=0	BRNE	Unsigned
Rd≤Rr	C+Z=1	BRSH(1)	Rd>Rr	C+Z=0	BRLO*	Unsigned
Rd<Rr	C=1	BRLO/BRCS	Rd≥Rr	C=0	BRSH/BRCC	Unsigned
Carry	C=1	BRCS	No carry	C=0	BRCC	Simple
Negative	N=1	BRMI	Positive	N=0	BRPL	Simple
Overflow	V=1	BRVS	No overflow	V=0	BRVC	Simple
Zero	Z=1	BREQ	Not zero	Z=0	BRNE	Simple

Conditional Branches

Conditional branches will modify the PC if the corresponding condition is met. In AVR, the condition is determined by condition codes in the Status Register (`SREG`) bits. For example, the `BRNE` (Branch Not Equal), instruction will look at the Zero (Z) flag of the SREG. If Z = 0, then the branch is taken, otherwise the branch is not taken. Note that these condition codes are modified based on the outcome of an instruction *before* the branch.

There are numerous instructions that can modify the `SREG` bits. Most arithmetic and logic instructions can affect all of the `SREG` bits. But what are more commonly used is the *compare* instructions, (`CP`, `CPC`, `CPI`, `CPSE`). These compare instructions subtract the two corresponding registers, and depending on its outcome, modify the `SREG`. Note that unlike arithmetic instructions, these compare instructions do not update any registers.

Table 4.2 shows a summary of all conditional tests, the corresponding instruction, and what bits in `SREG` are manipulated to determine the condition. The following example shows how conditional branch and compare instructions are used to implement control flow.

Consider the following C-code that tests two registers R0 and R1 and if the condition is true executes the statement:

```
if (R0 <= R1)
{
        Do Something;
```

}

The condition "<=" in the above code represents "less than equal to". This condition corresponds to the entry with Rd Rr as the test in Table 4.2, where Rd and Rr unsigned values. Therefore, the AVR instruction `BRSH`, which stands for "Branch if same or higher", can be used to implement "<=" . However, the footnote (1) next to it says that Rd and Rr need to be switched in the compare instruction. Thus, the following shows the equivalent assembly code:

```
      CP    R1, R0      ; Compare R1 with R0
      BRSH  TEST        ; Jump to test if R0 <= R1
      RJMP  NEXT        ; Jump to remaining code
TEST: ...do something...
NEXT: ...continue on...
```

Conditional branches also include the skip instructions (`SBRC`, `SBRS`, `SBIC`, `SBIS`). These instructions will skip the next instruction if the condition is met. These instructions can be very useful in, for example, determining whether or not to call a function. Note that the skip instructions are limited to only bit testing on registers or specific areas in I/O Memory.

Unconditional Branches

Unconditional branches directly modify the PC. These instructions are known as jumps because they cause the program to "jump" to another location in Program Memory. There are several types of jump instructions (`RJMP`, `IJMP`, `EIJMP`, `JMP`), but the most common one is *relative jump* (`RJMP`) because it is implemented as a 16-bit instruction and does not require a address register to hold the target address.

There are also special unconditional branch instructions known as *subroutine calls* (`RCALL`, `ICALL`, `EICALL`, `CALL`). Subroutine calls work just like jump instructions, except they also push the address of the current instruction plus 1 (i.e., PC+1), referred to as the *return address*, on to the stack before making the jump. There is also a corresponding *return* instruction (`RET`), that pops the return address from the stack and loads it into the PC. These instructions are used to implement functions in AVR assembly.

4.5.4 I/O Instructions

I/O instructions allow data to be read from or written I/O registers.

- `IN/OUT` (*input/output*)

- `IN Rd, A`
- `OUT A, Rr`

I/O instructions will be discussed in detail in Chapter 5.

4.5.5 Bit and Bit-test Instructions

Bit and bit-test instructions manipulate or test the individual bits within an 8-bit register. There are three types of bit and bit-test instructions; *Shift and Rotate*, *Bit Manipulation*, and *SREG Manipulation*.

Shift and Rotate

Shifting a register involves shifting all the bits one spot to either the left or the right. The AVR ISA categorizes register shifts as *shifts* and *rotates*. Shifting will just shift the last bit out to carry (C) bit and shift in a 0 to the first bit. Rotating will shift out the last bit to the carry bit and shift in the carry bit to the first bit. Therefore, rotating a register will not lose any bits while shifting a register will loose the last bit. The instruction mnemonics are `LSL`, `LSR`, `ROL`, and `ROR` for Logical Shift Left, Logical Shift Right, Rotate Left Through Carry, and Rotate Right Through Carry respectively.

There are also some special Shifting Instructions, Arithmetic Shift Right (`ASR`) and Swap Nibbles (`SWAP`). Arithmetic Shift Right behaves like a Logical Shift Right except it does not shift out to the carry bit. Instead, `ASR` will shift right anywhere from 1 to 7 spaces. Swap Nibbles will swap the upper and lower 4-bits with each other.

Bit Manipulation

Bit Manipulation Instructions allow programmer to manipulate individual bits within a register by either setting it to 1 or clearing it to 0, the individual bits. There are three instruction pairs that manipulate the SREG, an I/O Register, or a GPR through the T flag in the SREG. `BSET` and `BCLR` will set and clear, respectively, any bit within the SREG register. `SBI` and `CBI` will set and clear, respectively, any bit in an I/O register. `BST` will store any bit in a GPR to the T flag in the SREG, and `BLD`loads the value of the T flag in the SREG to any bit in a GPR.

Table 4.3: Conditional Branch Summary.

Bit	Bit Name	Set Bit	Clear Bit
C	Carry Bit	`SEC`	`CLC`
N	Negative Flag	`SEN`	`CLN`
Z	Zero Flag	`SEZ`	`CLZ`
I	Global Interrupt Flag	`SEI`	`CLI`
S	Signed Test Flag	`SES`	`CLS`
V	Twos Complement OVF Flag	`SEV`	`CLV`
T	T Flag	`SET`	`CLT`
H	Half Carry Flag	`SEH`	`CLH`

SREG Manipulation

Although the instructions `SBI` and `CBI` allow a programmer to set and clear any bit in the SREG, there are additional instructions that set and clear specific bits within the SREG. This is useful when the programmer does not want to keep track of which bit in the SREG is for what. Table 4.3 table shows the mnemonics for each set and clear instruction pair.

4.6 Assembly to Machine Instruction Mapping

Up until now, we have discussed assembly programming using mnemonics, or symbolic representation of instructions. Mnemonics make it easier for programmers to remember names of the instructions, registers, and memory locations using labels. These assembly mnemonics are then assembled into machine instructions consisting of 0s and 1 s that the processor understands. Therefore, this section discusses how assembly instructions are mapped to machine instructions. Understanding this process is important for a couple reasons. First, when the computer was first developed back in 1947, it was programmed using machine instructions (via mechanical switches) until assemblers and compilers came along! Second, the processor decodes and executes the machine instructions; therefore, understanding how various information of an instruction is encoded is crucial for designing and developing processor hardware to decode and execute these instructions.

Figure 4.14 shows the machine instruction mapping for

```
ADD   R15,R16     ; R15 <- R15 + R16
```

The 16-bit instruction format is subdivided into four 4-bit parts. The bit pattern '1001 11' represents the opcode for the `ADD` instruction, and the destination register identifier `ddddd` and source register identifier `rrrrr` are

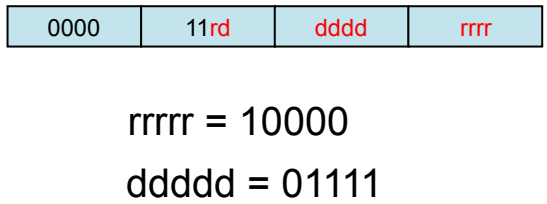


Figure 4.14: Machine instruction mapping for `ADD R15, R16`.

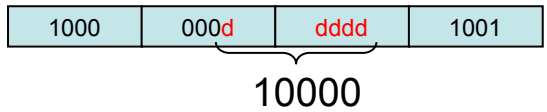


Figure 4.15: Machine instruction mapping for `LD R16, Y`.

located in the instruction format as shown in the figure. Since the destination and source register identifiers are `R15` and `R16`, `ddddd` and `rrrrr` will be '10000' and '01111', respectively. Note that all Arithmetic and Logic instructions that require two registers follow this convention, and the only thing that will be different is the opcode.

Figure 4.15 shows the machine instruction mapping for

```
LD R16, Y          ; R16 <- M(Y)
```

The location of the destination register identifier `ddddd` is the same as the case with two registers, and the rest of the bits represent the opcode for the `LD` instruction. Thus, the opcode also implies that the address register `Y` is being used. Since the destination register identifier is `R16`, `ddddd` will be '10000'.

Figure 4.16 shows the machine instruction mapping for

```
LDI R30, $F0       ; R30 <- $F0
```

Again, the location of the destination register identifier `ddddd` is the same as `LD`, except that the most significant bit does not exist. The reason for this is that the instruction format has to also support the two hexadecimal values `$F0`. Therefore, the instruction format forgoes the most significant bit, which is implied and is always 1, to accommodate the immediate value. The immediate value is then represented by the constant field `KKKK KKKK`, which is '1111 0000'.

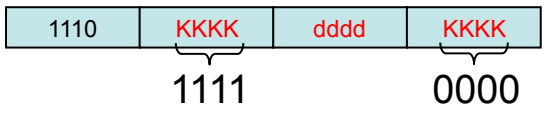


Figure 4.16: Machine instruction mapping for `LDI R30, $F0`.

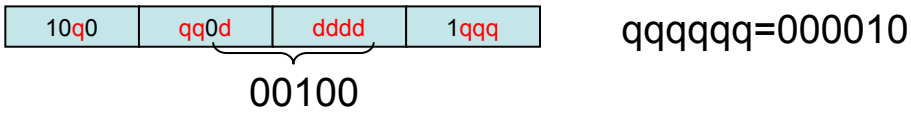


Figure 4.17: Machine instruction mapping for `LDD R4, Y+2`.

Figure 4.17 shows the machine instruction mapping for `LDD R4, Y+2`.

```
LDD R4,Y+2; R4 <- M(Y+2)
```

Again, the location of the destination register identifier `ddddd` is the same as `LD`. The 6-bit displacement field `qqqqqq` is spread across the instruction format and can represent any number between 0 and 63. Therefore, the displacement `2` is encoded as '000010'.

Figure 4.18 shows the machine instruction mapping for

```
IN R25, $16 ; R25 <- M($16+32)
```

Again, the location of the destination register identifier `ddddd` is the same as `LD` and `LDD`. The bits `AAAAAA` flank the `ddddd` field and represent any I/O register identifier between 0 and 63. Therefore, the I/O register identifier `$16` is encoded as '01 0110'. One clarification needs to made with regards to the instruction description in the comment field. `M($16+32)` indicates that I/O register identified is offset by 32. This is because the I/O register address space start after the 32 GPRs (see Figure 4.4). Therefore, even though the programmer can directly reference any one of the 64 I/O registers, its physical location in Data Memory is offset by 32.

Figure 4.19 shows the machine instruction mapping for

```
BEQ label   ; if (Z=1) then PC <- PC + 1 + k
```

This instruction is a *PC-relative branch*, thus the target address of the branch is calculated by adding the difference between the target address

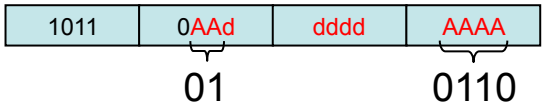


Figure 4.18: Machine instruction mapping for `IN R25, $16`.

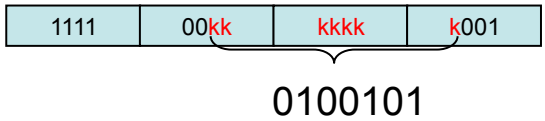


Figure 4.19: Machine instruction mapping for `BREQ label`.

of `label` and the address of the next instruction (i.e., PC+1), and adding to PC+1. This way, only the offset has to be stored in the instruction. This is illustrated in the following code. The address of the instruction after `BREQ` is `$0234`, while the target address of `BREQ`, i.e., `SKIP`, is `$0259`. Thus, the offset is `$0259-$0234` = `$0025`. Then, the six LSBs are stored in `kk kkkk k`, which is '01 0010 1'.

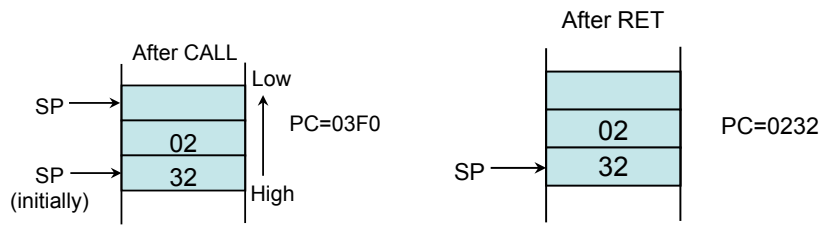
```
/Address/          /Code/
...                ...
0232               CP    R0 R0
0233               BREQ  SKIP
0234               {next instruction}
...                ...
0259        SKIP:
...                ...
```

Figure 4.21 shows the machine instruction mapping for

```
Call label  ; STACK <- PC, PC <- k,
```

This is a 32-bit instruction and holds the 16-bit target address, i.e., `label`, in the second half of the instruction. The `CALL` instruction is one of the few 32-bit instructions in the AVR Instruction Set Architecture. Note that the additional `k` bits in the first half of the instruction format allows for future expansion of the Data Memory.

The `CALL` instruction has another important requirement, which may not be obvious by looking at the instruction mapping, and that is to save



(a) Return address pushed onto the stack.

(b) Return address is popped from the stack.

Figure 4.20: Stack manipulation for subroutine call and return.

the *return address* of the subroutine call. In order to illustrate this process, consider the following code.

```
/Address/          /Code/
...                ...
0230               CALL  SUBR
0232               {next instruction}
...                ...
03F0        SUBR:
...                ...
...                {my subroutine}
...                ...
                   RET
```

`CALL` is used in conjunction with the `RET` (*return from subroutine*) instruction to implement subroutine calls and returns. Therefore, after the subroutine (`SUBR`) is executed, `RET` instruction jumps to the instruction after the `CALL`, which is referred to the *return address*. This requires `CALL` to save the return address onto the stack before jumping to the subroutine. Figure 4.20(a) illustrates the process of saving the return address onto the stack. In the above code, the return address is `$0232`, and thus the processor automatically pushes the lower and the upper bytes onto the stack. This way, when `RET` is executed as shown in Figure 4.20(b), the higher and lower bytes of the return address will be popped from the stack and move to PC. Note that the return address still remain in memory but is irrelevant relative to the stack.

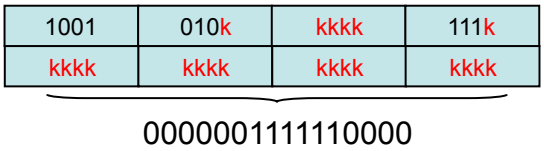


Figure 4.21: Machine instruction mapping for `CALL label`.

Table 4.4: Assembler Directives.

Directive	Description
`.BYTE`	Reserve byte to a variable
`.CSEG`	Code Segment
`.DB`	Define constant byte(s)
`.DEF`	Define a symbolic name on a register
`.DEVICE`	Define which device to assemble for
`.DSEG`	Data Segment
`.DW`	Define constant words
`.ENDMACRO`	End macro
`.EQU`	Set a symbol equal to an expression
`.ESEG`	EEPROM segment
`.EXIT`	Exit from a file
`.INCLUDE`	Read source from another file
`.LIST`	Turn listfile generation on
`.LISTMAC`	Turn macro expression on
`.MACRO`	Begin Macro
`.NOLIST`	Turn listfile generation off
`.ORG`	Set program origin
`.SET`	Set a symbol to an expression

4.7 Assembler Directives

Assembler directives are special instructions that are executed before the code is assembled and directs the assembler about information it needs to know in order to carry out the assembly process, e.g., code location, constant definitions, storage definitions, etc.. These instructions are denoted by a preceding dot, i.e., `.EQU`. The directives are not assembled into assembly instructions. Instead, they are used to adjust the location of the program in memory, define macros, initialize memory, and so on. Table 4.4 provides an overview of the directives supported by the AVR Assembler.

The following discusses the most commonly used directives.

`.ORG` - Set program origin

The `.ORG` directive sets the program counter (PC) to an absolute value. The value to set is given as a parameter. If an `.ORG` directive is given within a Code Segment, then it is the Program memory counter that is set. If the directive is preceded by a label (on the same source line), the label will be given the value of the parameter. The default value of the Code location counter is zero when assembling is started. Note that the Program memory location counter counts words and not bytes.

Syntax: `.ORG` *expression*

Example:

```
      .ORG  $0000       ; rjmp is located in Program Memory $0000
      rjmp   main       ; Jump to the main section of code}
      .ORG  $0042       ; main starts at location $0042
main:
      mov   r1, r0      ; Do something
```

`.CSEG` - Code Segment

The `.CSEG` directive defines the start of a Code Segment. An assembler file can contain multiple Code Segments, which are concatenated into one Code Segment when assembled. The example code below shows how `.CSEG` directive is used.

Syntax: `.CSEG`

Example:

```
      .DSEG             ; Start Data Segment
vartab:
      .BYTE 4           ; Reserve 4 bytes in SRAM
      .CSEG
const:
      .DW   2           ; Write 0x0002 in Program Memory
      mov   r1, r0      ; Do something
```

`.DB` - Define constant byte(s)

The `.DB` directive reserves memory resources in the Program Memory. In order to refer to the reserved locations, a label should precede the `.DB` directive.

The `.DB` directive takes a list of expressions, delimited by commas, and must contain at least one expression. Each expression must evaluate to a number between 128 and 127 since each expression is represented by 8-bits.

If the `.DB` directive is used in a Code Segment and the expression list contains more than one expression, the expressions are packed so that two bytes are placed in each Program Memory word. If the expression list contains an odd number of expressions, the last expression will be placed in a Program Memory word of its own, even if the next line in the assembly code contains a `.DB` directive. The following example code shows the use of `.DB` directives to place constants and characters in the Program Memory.

Syntax: `LABEL: .DB expressionlist`

Example:

```
      .CSEG
consts:
      .DB   0, 255, 0b01010101, -128, $AA
text:
      .DB   "Hello World!"
```

`.DEF` - Set a symbolic name on a register

The `.DEF` directive allows registers to be referred to through symbols. This allows programmers referred to registers with familiar symbolic names rather than register numbers. A register can have several symbolic names attached to it. A symbolic name can be redefined later in the program. The following example code shows how registers `R16` and `R0` can be assigned to symbolic names `temp` and `ior`, respectively, and used in the program.

Syntax: `.DEF Symbol = Register`

Example:

```
      .DEF  temp = R16
      .DEF  ior = R0
      ldi   temp, $F0         ; Load 0xF0 into temp register
      in    ior, $3f          ; Read SREG into ior register
text:
      eor   temp, ior         ; Exclusive or temp and ior
```

`.EQU` - Set a symbol equal to an expression

The `.EQU` directive assigns a value to a label. A *label* is a symbolic name assigned to an address of an instruction or a constant (or group of constants) in Program memory, an address of memory location in Data Memory, or any constant. This label can then be used in later expressions. A label assigned to a value by the `.EQU` directive is a constant and can not be changed or redefined. The following example shows how the symbolic name io_offset is assigned to the value $23, and then used in an another expression (porta = io_offset + 2) to be used in the body of the code.

Syntax: `.EQU label = expression`

Example:

```
.EQU  io_offset = $23
.EQU  porta = io_offset + 2
clr   r2                ; Clear R2
out   porta, r2         ; Write to Port A
```

`.INCLUDE` - Include another file

The `.INCLUDE` directive tells the Assembler to start reading from a specified file. The Assembler then assembles the specified file until the end of file (EOF) or the `.EXIT` directive is encountered. An include file may itself contain `.INCLUDE` directives. The following example code indicates the use of `iodefs.asm` file.

Syntax: `.INCLUDE` *filename*

Example:

```
                        ; iodefs.asm
.EQU  sreg = $3F        ; Status register
.EQU  sphigh = $3E      ; Stack pointer high
.EQU  splow = $3D       ; Stack pointer low
                        ; incdemo.asm
.INCLUDE    "iodefs.asm"; Include I/O definitions
in    r0, sreg          ; Read status register
```

4.8 Expressions

As we saw in the last few code examples, the assembler supports expressions. Expressions can consist of operands, operators, and functions.

4.8.1 Operands

The following operands can be used:

- User defined labels that are given the value of the location counter at the place they appear.
- User defined constants defined by the `.EQU` directive.
- Integer constants: constants can be given in several formats, including
  - Decimal (default): 10, 255
  - Hexadecimal (two notations): 0x0a, $0a, 0xff, $ff
  - Binary: 0b00001010, 0b11111111
- PC the current value of the Program Counter

4.8.2 Operators

The assembler supports a number of operators shown in Table 4.5. These operators can be commonly associated with C/C++ operators. Note that these operations are done only during compilation and are not used in place of the AVR Instructions.

4.8.3 Functions

The following functions are defined:

- `LOW(expression)` returns the low-byte of an expression
- `HIGH(expression)` returns the high-byte of an expression
- `BYTE2(expression)` is the same function as `HIGH`
- `BYTE3(expression)` returns the third byte of an expression
- `BYTE4(expression)` returns the fourth byte of an expression
- `LWRD(expression)` returns bits 0-15 of an expression
- `HWRD(expression)` returns bits 16-31 of an expression
- `PAGE(expression)` returns bits 16-21 of an expression
- `EXP2(expression)` returns $2^{expression}$
- `LOG2(expression)` returns the integer part of log_2(expression)

Table 4.5: Expression Operators.

Symbol	Description
!	Logical Not
~	Bitwise Not
-	Unary Minus
*	Multiplication
/	Division
%	Modulo
+	Addition
-	Subtraction
<<	Shift left
>>	Shift right
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Not equal
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
&&	Logic AND
||	Logic OR
?	Conditional operator

4.9 Assembly Coding Techniques

For many of you this may be the first time you are writing assembly programs. This section presents general guidelines to produce well-structured codes that can be easily understood and debugged.

4.9.1 Code Structure

It is important to create and maintain a consistent code structure throughout the program. Assembly language in general can be very confusing, and spending several hours trying to find a specific problem area in a piece of code can become quite frustrating. A well-structured program will ease this confusion and make the program very readable to yourself and to others.

So what does a well-structured assembly program look like? Structure includes everything that is typed in the program, where certain parts of the program are located, how an instruction looks within a line, etc. There are several ways to write out the code on the paper, but the most important part

Table 4.6: Line Formatting Rules.

Column	Tab Length	Includes	Comments
1	1	Pre-compiler Directives, Labels	If a label is longer than one tab length, then the instruction mnemonic goes on the next line. No instructions must be placed on the same line as a pre-compiler directive.
2	1	Directive Parameters, Instruction Mnemonic	It is common for Directive Parameters to exceed one tab length.
3	2	Instruction Parameters	If Instruction Parameters exceed two tab lengths, then place the comment on the previous line.
4	3	Comments	Comments should only be used in the fourth column, which is roughly four tab lengths from the start of the line. Unless the code is blatantly obvious, place a comment on every line of code. Exceptions are Header Comments, they must start at the beginning of the line

is to be consistent. If you start writing your code in one fashion, maintain that fashion through out the remainder of the program. Varying between different styles can be quite confusing and make the code unreadable.

The one style that I recommend is using the four-column method. If this style is used consistently throughout the program, the program should look like four columns. A column is usually separated with one or two tabs depending on how long the data strings are. In general, Table 4.6 describes what goes into each column, the tab lengths, and exception rules.

The following is an example of well-formatted code using the rules from Table 4.6.

```
; XOR Block of Data

.include    m128def.inc     ; Include definition file

.def  tmp = r15               ; Temporary Register
.def  xor = r6                ; XOR Register
.equ  addr1 = $1500           ; Beginning Address
.equ  addr2 = $2000           ; Ending Address
```

```
                                ; This code segment will XOR all the
                                ; bytes of data between the two address
                                ; ranges.
      .org  $0000               ; Set the program starting address
INIT:
      ldi   XL, low(addr1)      ; Load low byte of start address in X
      ldi   XH, high(addr1)     ; Load high byte of start address in X
FETCH:                          ; Code wont fit, create a new line
      ld    tmp, X+             ; Load data from address into tmp
      eor   xor, tmp            ; XOR tmp with xor register, store in xor
      cpi   XL, low(addr2)      ; Compare high byte of X with End Address
      brne  FETCH               ; If low byte is not equal, then get next
      cpi   XH, high(addr2)     ; Compare low byte of X with End Address
      brne  FETCH               ; If high byte is not equal then get next
DONE:
      rjmp        DONE          ; Infinite done loop
```

The next part to proper code structure is code placement. Certain sections of code should be placed in certain areas. This alleviates confusion and allows the contents to be ordered and navigable. Table 4.7 describes the order in which certain code segments are to be placed.

By following these simple structure rules, the code will be more readable and understandable.

4.9.2 ATmega128 Definition File

A definition file contains addresses and values for common I/O registers and special registers within a specific processor. For example, every Atmel AVR processor contains SREG, bit its location in memory may not be the same across different processors. Thus, a definition file can contain common names for I/O registers, such as `SREG` and `SPH`, so that programmers do not have memorize or look up each I/O register or processor specific registers. The definition file for ATmega128 is `m128def.inc`, which contains lots of .EQU and a few .DEF directives, as well as other useful information, such as the last address in SRAM (`RAMEND`). `m128def.inc` is included with AVR Studio4 when you download and install the program.

4.10 Control Flow

Conditional branch and jump instruction are essential for implementing control flow in a program. Therefore, this section discusses examples of how

Table 4.7: Code Structure.

Header Comments	Title, Author, Date, and Description
Definition Includes	Specific Device Definition Includes, i.e. m128def.inc
Register Renaming	Register renaming and variable creation, i.e. .def tmp = r0
Constant Declaration	Constant declarations and creation, i.e. .equ addr = $2000
Interrupt Vectors	See Section 4.2 Interrupt Vectors
Initialization Code	Any initialization code goes here
Main Code	The heart of the program.
Subroutines	Any subroutine that is created follows the main code.
ISRs	Any Interrupt Subroutines will go here.
Data	Any hard coded data is best placed here, i.e. .DB hello
Additional Code Includes	Finally, if there is any additional source code includes, will go last.

C/C++ control-flow expression are coded in assembly.

4.10.1 IF Statement

IF statement is probably the simplest control statement in programming, and is written as

```
if (expr)
        statement
```

If *expr* is true, then *statement* is executed; otherwise, *statement* is skipped. This is true for assembly as well. For example, consider the following C code.

```
if (n >= 3)
{
        expr++;
        n = expr;
}
```

Although there many different ways to write equivalent assembly code, the code below shows one possibility:

```
.def  n = r16
.def  expr = r1
.equ  cmp = 3

...
      cpi   n, cmp            ; Compare value
IF:
      brsh  EXEC              ; If n >= 3 then branch to NEXT
      rjmp  NEXT              ; Jump to NEXT since expression is false
EXEC:
      inc   expr              ; increment expr
      mov   n, expr           ; Set n = expr
NEXT: ...                     ; continue on with code
```

The `cpi n, cmp` instruction compares the value in `r16` with the immediate value `3`, which then appropriately sets the condition codes depending on the outcome. The `brsh` (*branch if greater than or equal*) instruction branches to the label `EXEC` if `r16` is greater than equal to `3`; otherwise, skips to the next instruction (i.e., `rjmp NEXT`). Note that we could have also used `brcc`(*branch if carry cleared*) instead of `brsh`.

Although this assembly code behaves like the C code, it is not optimal in terms of code size and/or execution time. One way to optimize this code is to simply use the complementary Boolean expression `brlo` (*branch if less than*). This is shown below:

```
.def  n = r16
.def  expr = r1
.equ  cmp = 3
...
      cpi   n, cmp            ; Compare value
IF:
      brlo  NEXT              ; If n >= 3 is false then skip code
      inc   expr              ; increment expr
      mov   n, expr           ; Set n = expr
NEXT: ...                     ; Continue on with the code
```

This assembly program uses one less branch statement and one less line of code, and it is also easier to read and understand. You may wonder how much of a speed improvement is achieved by removing one instruction from each side of the branch. As a stand-alone statement, it may not make a

much of a difference. However, if the IF statement is nested in a loop that executes many times, its execution time can be significantly reduced. Note that we could have also used `brcs`(*branch if carry set*) instead of `brlo`.

4.10.2 IF-ELSE Statement

This is very similar to the IF statement, except it has an additional else statement. Consider the following example C code.

```
if (n == 5)
        expr++;
else
        n = expr;
```

The equivalent AVR assembly code is shown below:

```
.def  n = r16
.def  expr = r1
.equ  cmp = 5
...
      cpi   n, cmp            ; Compare value
IF:
      brne  ELSE              ; Goto ELSE statement since expression is false
      inc   expr              ; Execute the IF statement
      rjmp  NEXT              ; Continue on with code
ELSE:
      mov   n, expr           ; Execute the ELSE statement
NEXT: ...                     ; Continue on with code
```

Again, this code uses the complimentary Boolean expression `brne` (*branch not equal*) is used instead of `breq` (*branch equal*), which makes the code more compact and run faster.

4.10.3 Loops

Loops can be implemented using FOR, WHILE, and DO statements.

FOR statement

The FOR statement is used to execute code iteratively, and are commonly used to process an array data. For example, the following code iterates 10 times.

```
for (n = 0; n < 10; n++)
        sum += n;
```

The following is the equivalent assembly code.

```
.def  n = r16
.def  sum = r3
.equ  max = 10
...
      ldi   n, max      ; Initialize n to max
FOR:
      add   sum, n      ; sum += n
      dec   n           ; decrement n
      brne  FOR         ; repeat loop if n does not equal 0
NEXT: ...               ; rest of code
```

WHILE statement

The WHILE statement is also commonly used to create loops, and has the form:

```
while (expr)
        statement
next statement
```

First *expr* is evaluated, if it true, then the statement is executed, and control is passed back to the beginning of the WHILE loop. The effect of this is that the body of the WHILE loop, namely the statement, is executed repeatedly until *expr* is false. At that point, control is passed to next statement. Consider the following C code example:

```
while (n < 10) {
        sum += n;
        n++;
}
```

The equivalent AVR assembly code is shown below:

```
.def  n = r16
.def  sum = r3
.equ  limit = 10
...
WHIL: cpi   n, limit    ; Compare n with limit
      brsh  NEXT        ; If n >= limit, goto NEXT
```

```
      add   sum, n      ; sum += n
      inc   n           ; n++
      rjmp  WHIL        ; Go back to beginning of WHILE loop
NEXT: ...               ; Continue on with code
```

DO Statement

The DO statement can be considered a variant of the WHILE statement. Instead testing at the top of the loop, it is performed at the bottom. The following is an example:

```
do {
        sum += n;
        n--;
} while (n > 0);
```

The assembly code for the DO statement is also very similar to the WHILE statement.

```
.def  n = r16
.def  sum = r3
.equ  limit = 0
...
DO:   add   sum, n      ; sum += n
      dec   n           ; n++
      brne  DO          ; since n is unsigned, brne is same expr
NEXT: ...               ; Continue on with code
```

In this code `dec` is called before the `brne` instruction, and thus the `cpi` instruction is not needed.

4.10.4 Functions and Subroutines

AVR Assembly language supports *functions* and *subroutines*, which allows a programmer to drastically reduce the size and complexity of the code. Because of their importance, this topic is given its own subsection. This section will cover the general topics of functions and subroutines, including how to create one, when to create one, and how to use one.

Definitions

A function or a subroutine can generally be thought as reusable code. Reusable code is any segment of code that can be used over and over again

throughout the program without duplications. You can think of functions and subroutines as they are implemented in a C program. A function is created outside of the main program and then is later called (perhaps multiple times) within the main program.

Reusable code within an assembly program can be thought of as either a function or a subroutine. Both are very similar in terms of how they are implemented, but have subtle differences. A subroutine is a reusable piece of code that requires no input from the main program. Generally, the state of the program is saved upon entering the subroutine and is restored before leaving the subroutine. A function, on the other hand, is involved within the main code and requires some interaction. This usually means that some registers or other memory locations have to be initialized before the function is called. In addition, a function will most likely alter the state of the program.

In general, a subroutine must not alter the state of a running program and care must be taken to ensure the proper operation of a subroutine. For a function, the main program must initialize data for the function to work properly and must be able to handle any changes caused by the function.

Operational Overview

A subroutine or function is implemented via `CALL`, `RCALL`, `ICALL`, or `EICALL` instructions and is matched with the `RET` instruction to return to the address of the instruction after the subroutine call, referred to as the *return address*. A function or a subroutine is preceded by a label that signifies its name. When `CALL` is executed, the processor first pushes the return address onto the stack. This is important to realize since it means that the stack must be initialized before functions or subroutines can be used. The `CALL` instruction will then jump to the address specified by label used as the parameter. The next instruction to be executed will then be the first instruction within the subroutine or function. Upon exiting the subroutine or function, the `RET` instruction must be called. `RET` pops the return address from the stack and loads it into the PC. Thus, the next instruction to be executed is the instruction after the `CALL` instruction.

It is important to keep track of what is pushed onto and popped off of the stack. If within a subroutine or function, data is not popped correctly, the RET instruction can pop the wrong data values for the address and thus the program will not function correctly. Additionally, never exit a subroutine or function via another jump instruction other than RET. Doing so will cause the data in the stack to never be popped and thus the stack will become out

of sync.

So when does a programmer decide to create a subroutine or function? This can often be a tough call and can get really complicated. The general rule of thumb is that a subroutine or function is needed when the programmer finds that the same segment of code is being written in several places within the program. This can be very troublesome since an error within the code segment results in several hours of work trying to fix every instance of the code segment. On the other hand, if the program were to use a function or subroutine, then fixing the code is easy since there is only one instance and it is in a common location.

Implementation

In the last section, we briefly talked about how a function or a subroutine works. This section will give detailed explanations on how to implement both the function and the subroutine.

The first thing to do for any function and subroutine is to initialize the stack. This can be done in four lines of code at the beginning of the program. Optimally, the stack should be initialized for any program. Here is the code:

```
.include m128def.inc

INIT:                                ; Initial the stack pointer
      ldi   r16, low(RAMEND)         ; Load the lo byte of the rams end addr
      out   SPL, r16                 ; Set the Stack Pointer Low register
      ldi   r16, high(RAMEND)        ; Load the hi byte of the rams end addr
      out   SPH, r16                 ; Set the Stack Pointer High register
```

After this point, any function or subroutine will be oriented correctly with regard to the stack.

The subroutine does not require any outside influence for its performance. Therefore it is a good idea to save the state of the program before executing the subroutine. This means that certain registers must be pushed to the stack in the beginning of the subroutine and popped just before the subroutine ends. It is important to remember to pop registers in reverse order from which they where pushed. These registers include the SREG (essentially the state of the program) and any general purpose registers that are used throughout subroutine.

A good example of a routine is a wait loop that will wait for a specific amount of time. For our case, we will want the wait subroutine to wait for 1000 cycles (not including the subroutine overhead cycles.)

```
.def  ocnt = r16              ; Outer loop count variable
.def  icnt = r17              ; Inner loop count variable
WAIT:                         ; Wait subroutine
      push  icnt              ; Save icnt register
      push  ocnt              ; Save ocnt register
      in    ocnt, SREG        ; Get the SREG value
      push  ocnt              ; Save the value of the SREG
      ldi   ocnt, 55          ; Loop outer loop 55 times
WTL1:
      ldi   icnt, 5           ; Loop inner loop 5 times
WTL2:
      dec   icnt              ; Decrement inner loop counter
      brne  WTL2              ; Continue through inner loop
      dec   ocnt              ; Decrement outer loop counter
      brne  WTL1              ; Continue through outer loop
      ldi   ocnt, 3           ; This next loop just uses 9 cycles
WTL3:
      dec   ocnt              ; that is need for the 1000 cycles
      brne  WTL3              ; Repeat last loop 3 times
      nop                     ; We still come up 1 cycle short
      pop   ocnt              ; Get the SREG value
      out   SREG, ocnt        ; Restore the value of the SREG
      pop   ocnt              ; Restore ocnt register
      pop   icnt              ; Restore icnt register
      ret                     ; Return from subroutine
```

We are going to stray off topic for a second to discuss this code. From the time first LDI instruction is called to the last NOP, there are exactly 1000 cycles of execution. To calculate this, you must take into consideration the number of cycles it takes to execute each instruction. With the exception of the branches, every instruction takes one cycle. The branches take 1 cycle if false and 2 cycles if true. With this in mind, the main loop follows the equation ((3*icnt + 3)*ocnt . The optimal values for icnt and ocnt are 5 and 55 respectively. This yields the total number of cycles for the main to be 990 cycles. This is unfortunately 10 cycles short of our goal. We could just shove 10 consecutive NOPs at the end, but instead opted for a second small loop. This second loop follows the equation 3*ocnt and with a value of ocnt being 3 yields 9 cycles. Therefore, with the addition of a single NOP instruction, our total number of cycles is 1000 cycles.

Now back to the topic of subroutines. As you can see in the example code, the very first thing we do is push the SREG to the stack. Additionally, we also push the registers icnt and ocnt since they are used within the subroutine. We then execute the main subroutine code. This followed up by

popping the data from the stack in the reverse order that it is pushed. And finally, we leave the subroutine with the RET instruction. Another item to notice is that PUSH and POP only deal with the general-purpose registers. Therefore, if we wanted to push an I/O Register, such as the SREG, we must first load it into a general-purpose register. Since we were already using the ocnt register within the subroutine, we used it to temporarily hold the SREG value. We made sure the ocnt register was first pushed to the stack so that we didnt loose any data that might have been there.

So now how does a programmer use a subroutine once it has been created? Well, this is easier than it sounds. Since a label precedes the subroutine, we can just make a call to the label with one of the CALL instructions.

```
MAIN:
      ldi   r16, 4            ; Set r16 to 4
LOOP:
      rcall WAIT              ; Call our WAIT routine
      dec   r16               ; Decrement r16
      brne  LOOP              ; Call the wait statement 4 times
      ...                     ; Additional code
      rcall WAIT              ; Call our WAIT routine
      ...                     ; Even more code
DONE:
      rjmp  DONE              ; Program complete
```

As you can see, this program will call the WAIT routine in several places. Now if the routine did not exist, then we would have to write all that code twice in several places. While this is not hard with modern technologies such as cut and paste, the might still initially be incorrect. This means that we now have to search through the rest of the code and this can quickly become a headache. By now, you should be able to understand how subroutines can make your programming experience better.

Function Implementation

A function is a bit different from a subroutine in the fact that it alters the state of the program. Unlike the subroutine, the function will most likely need to be initialized prior to the function call. This is how to create input to a function. Also the function will most likely modify some registers to provide output. With the exceptions of the input and output, a function behaves just like a routine. Any registers that used within a function but do not correspond to any input or output, must still be pushed and popped from the stack.

We will use a common example of a function that takes two 8-bit numbers and multiplies them together. Since an 8-bit multiplication results in a 16-bit number, the result will be stored in the same registers that were used for the input. The two input registers are the multiplier (`mplr`) and the multiplicand (`mplc`). The high byte of the result will be stored in mplr and the low byte will be stored in the mplc. The basic multiplication algorithm will be to repeatedly add the multiplicand to itself for the amount specified in the multiplicand. So here is the function.

```
.def  mplr = r18              ; The multiplier
.def  mplc = r19              ; The multiplicand
.def  resh = r1               ; The high byte of the result
.def  resl = r2               ; The low byte of the result
.def  zero = r0               ; Zero register that always contains 0

MUL:                          ; The multiplication function
      push  resh              ; Save the state of resh
      push  resl              ; Save the state of resl
      push  zero              ; Save the zero register
      clr   zero              ; Enforce the 0 in the zero register
      clr   resh              ; Clear the result high byte
      clr   resl              ; Clear the result low byte
      cpi   mplr, 0           ; Initially check mplr for 0
      breq  EXIT              ; Check for end condition
ADD:
      add   resl, mplc        ; Add multiplicand to result
      adc   resh, zero        ; This just add the carry bit, if any
      dec   mplr              ; Decrement mplr
      brne  ADD               ; Repeat loop if mplr is not 0
EXIT:
      mov   mplr, resh        ; Move high byte of result to mplr
      mov   mplc, resl        ; Move low byte of result to mplc
      pop   zero              ; Restore zero register
      pop   resl              ; Restore resl register
      pop   resh              ; Restore resh register
      ret                     ; Return from function
```

As you can see, the general structure and layout is very similar to a subroutine. The only difference is that we did not push and pop the registers mplr and mplc, since they are used for input and output. You will also note the SREG was not saved. Sometimes, a function will want to return the value of the SREG and therefore it is not necessarily vital to save it.

Now we will see how to utilize a function. The process is very similar to subroutine except that the input values need to be initialized before the

function is called. It is also important to note that register renaming for the functions input and outputs should be done at the beginning of the program with the programs renamed registers. Like the subroutine, any register that is used internally can be renamed before the instance of the function. So here is an example of how to use the function we created in the above example.

```
.def  mplr = r18
.def  mplc = r19
INIT:
      ldi   r16, high(RAMEND) ; Initialize the stack pointer high byte
      out   SPH, r16
      ldi   r16, low(RAMEND)  ; Initialize the stack pointer low byte
      out   SPL, r16
MAIN: ...                     ; Other code in the main program
      ldi   mplr, 25          ; Load 25 into multiplier
      ldi   mplc, 93          ; Load 93 into multiplicand
      rcall MUL               ; Multiply 93 * 25
      st    X+, mplr          ; Do something with the result
      st    X, mplc           ; by storing it into memory
      ...                     ; Additional Code
      mov   mplr, r5          ; Setup another multiplication function
      mov   mplc, r8          ; with the registers r5 and r8
      rcall MUL               ; Multiply r8 * r5
      mov   r0, mplr          ; Do something with the result
      mov   r1, mplc          ; by storing the result into r0:r1
      ...                     ; More code
DONE:
      rjmp  DONE              ; Program complete
```

As you can see, we initialize the function by storing the value we want to multiply into the `mplr` and `mplc`. We then call the multiplication function, `MUL`. And finally we handle the results. What values you want to multiply and what to do with the result depends on the program. By now, you should have a good understanding as how to create and use a function.

4.11 Anatomy of an Assembly Program

Now that we have had some basic ideas about how to program in AVR assembly, this section discusses what an assembly program looks like in memory.

Consider an example program show below, which adds eight numbers stored in Program Memory. Fig. 4.22 shows the layout of the machine code in Program Memory.

```
.include    m128def.inc
      .org  $0000
      rjmp  Init_addr
      .org  $000B
Init_addr:
      ldi   ZL,low(Count<<1)  ;
      ldi   ZH,high(Count<<1) ;
      lpm   R16,Z             ; R16 =8
      ldi   ZL,low(Nums<<1)   ;
      ldi   ZH,high(Nums<<1)  ; Z points to 12
Init_acc:
      clr   R1                ; Accumulate the result in R1(H) and R0(L)
      clr   R0                ;
Loop:
      lpm   R2,Z+             ; Load data to R2 and post inc. pointer
      add   R0, R2            ; Add R2 to R0(L)
      brcc  Skip              ; No carry, skip next step
      inc   R1                ; Add carry R1(H)
Skip:
      dec   R16               ; Count down the # words to add
      brne  Loop              ; If not done loop
Done:
      jmp   Done              ; Done.  Loop forever.
Nums:
      .DB   12, 24, 0x3F, 255, 0b00001111, 2, 21, 6
Count:
      .DB   8
```

Figure 4.22 shows the layout of the machine code represented as hexadecimal for the above assembly code in Program Memory. Note that that upper and lower bytes of the 16-bit instruction format are flipped. That is, unlike how we view the instruction format from most significant bit (MSB) to least significant bit (LSB), or from most significant hexadecimal digit to least significant digit, the actual order in Program Memory is reversed. This is because AVR uses little-endian, where the lower byte of instruction comes first. In addition, due to limited space, two memory words are shown per line or row rather just one.

The machine code consists of four sections: relative jump instruction, pointer initialization, main loop, and data.

The first section of the machine code is the `rjmp` instruction, which performs a relative jump to label `Init_addr` upon reset. `rjmp` is located at address `$0000` defined by directive `.org $0000`. (thus PC+1 = $0001) and The target address of `rjmp` is the `Init_addr`, which is located at address

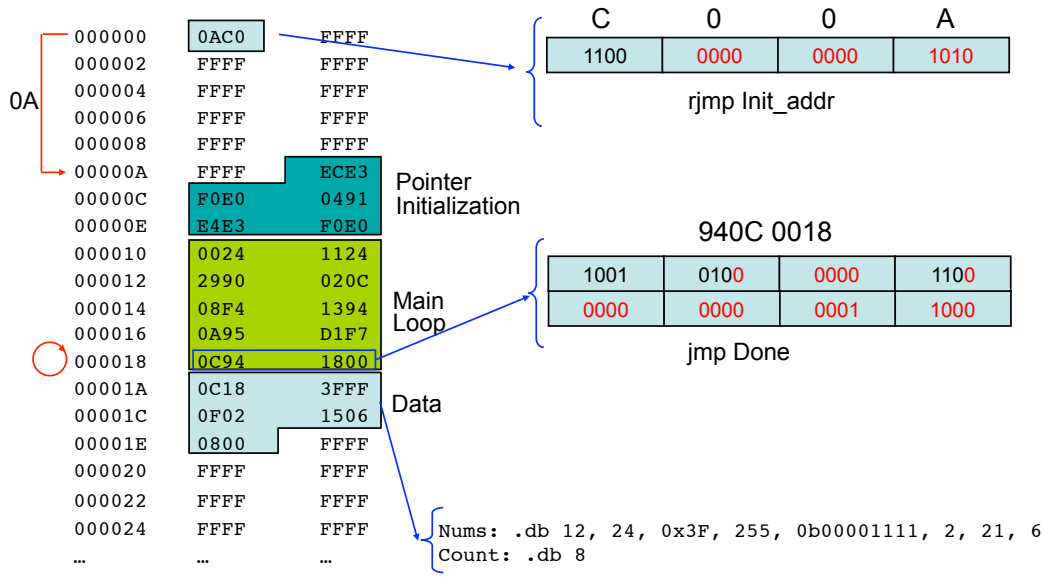


Figure 4.22: Contents of the Program Memory.

$000B. Since `rjmp` is a PC-relative jump, the displacement is calculated by subtracting the address of `rjmp` plus 1 (i.e., `$0001` from the target address, i.e., `$000B-$0001=$000A`. Then, the first 12 bits (or 3 hexadecimal digits) of the displacement are stored in the instruction.

The second section of the code consists of pointer initialization, which starts at address location `$000B`.

The main loop starts at location `$0010` and ends with a 32-bit `jmp` instruction that loops back to itself. The `jmp` instruction uses direct program addressing and thus the second 16-bit of the instruction holds the address of the label `Done` (`$0018`). Note that the `jmp` instruction has six additional bits for address in the first 16-bit of the instruction. These bits leave room for possible expansion to the 64K Program Memory address space.

The last section of the machine code contains data. The eight numbers to be accumulated starts at label `Nums` and the number `8` is located at label `Count`, which are located at Program Memory addresses `$001A` and `$001E`, respectively. In order to access the either numbers, Z-register is used to first point to label `Count`. This is achieved using the following instructions:

```
Init_addr:
```

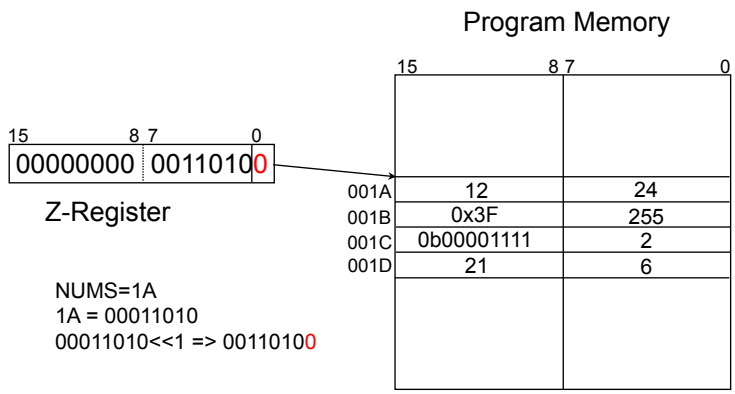


Figure 4.23: `lpm` instruction.

```
ldi   ZL,low(Count<<1)  ;
ldi   ZH,high(Count<<1) ;
```

As discussed in Sec. 4.8.3, `low()` returns the low-byte of the address generated by the expression `Count<<1`. The operator `<<` means shift left by one bit. Thus, `Count<<1` takes the address of label `Count`, which is `$001E` (0000 0000 0001 1110_2), and shifts it left by one bit resulting in `$003C` (0000 0000 0011 1100_2). The low-byte `$3C` is then stored in the low-byte of the Z-register. The same applies to the second instruction except the high-byte of the address is loaded into the high-byte of the Z-register. Thus, after the execution of these two instructions, the Z-register contains `$003C`.

The `lpm r16,Z` instruction is used load the constant value 8 in to `r16`. How the `lpm` instruction distinguishes between the first byte and the second byte will be explained shortly,

As can be seen from the code above, the same is done for the label `Nums`. However, the challenge is accessing data from Program Memory that is on 2-byte boundaries, yet each data is only a byte! Figure 4.23 illustrates this situation where the eight numbers are stored in consecutive bytes in Program Memory. The key is the Program Memory constant addressing mode discussed in Section 4.4 (see Fig. 4.11). The expression `Nums<<1` shifts in zero to the LSB, which allows the first (left) byte to be distinguished from the second (right) byte. After the Z-register is initialized to point to the first 8-bit data of the eight-element data array, the `lpm R2,Z+` instruction is used to load each data element and then increment the Z-register to point to the next 8-bit data.

Chapter 5

Input/Output

Input/Output (I/O) refers to the communication between a processor and external devices. Microcontrollers for embedded systems typically have simple I/O interfaces to communicate with relatively simple devices. In contrast, high-end microprocessors for laptops, desktops, and servers have pretty complex I/O devices, such as keyboards, mouse, display, printers, etc., or other computer systems, which in turn have microcontrollers in them. Therefore, this chapter discusses I/O functionalities of microcontrollers, in particular, AVR microcontrollers.

Types of I/Os handled depends on the application. A typical application may involve a *sensor* that monitors the some physical condition, such as temperature or rotational speed of a wheel (e.g., Anti-lock Braking System), as input signals, which can then be processed by a microcontroller. Some I/O devices (e.g., timers/counters and A/D and D/A converters) are implemented within the microcontroller, while others (e.g., keypad, LEDs, and displays) are implemented separately.

5.1 AVR I/O Capabilities

AVR processors support I/O functionalities through dedicated I/O ports and interrupt facility, which are controlled by special I/O registers.

5.2 I/O Ports

I/O ports are used to communicate with external devices. AVR Atmega128 has six 8-bit ports and one 5-bit port (53 I/O lines) as shown in Fig. 5.1. These ports are basically registers and are referred to as Port A-H *Data*

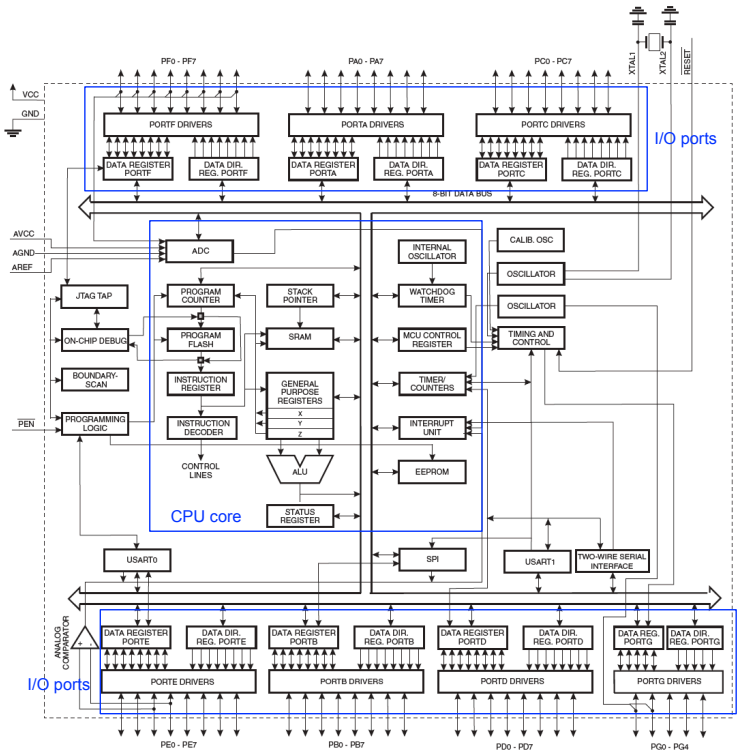


Figure 5.1: I/O ports in AVR processor.

Register (PORTA-H). Each port can be used send/receive 8-bit data or each I/O line or *pin* can be configured to send/receive a single bit. Since each I/O line can be can be read or write, Port A-H *Data Direction Register* (DDRA-H) is used to control whether these lines are input or output. Both PORTx and DDRx are mapped to the I/O Register address space in Data Memory, e.g., PORTA is mapped to $3B and DDRA is mapped to $3A.

Fig 5.2 shows the simplified internal structure of a single I/O pin. Each I/O pin, Pxn, is connected to a pair of latches PORTxn and PINxn, and a control latch DDRxn, where the suffix 'x' represents the port name (A-H) and the suffix 'n' is the bit position (7-0) within a port. PORTxn is used to send data to the pins and PINxn is used to receive data from the pins. PORTxn latch has an additional purpose, which will be discussed in Sec. 5.3. DDRxn controls the direction of the pins, either input or output. Internal control signals for the three tri-state buffers control when the data in PORTxn, PINxn, and DDRxn appear on the data bus.

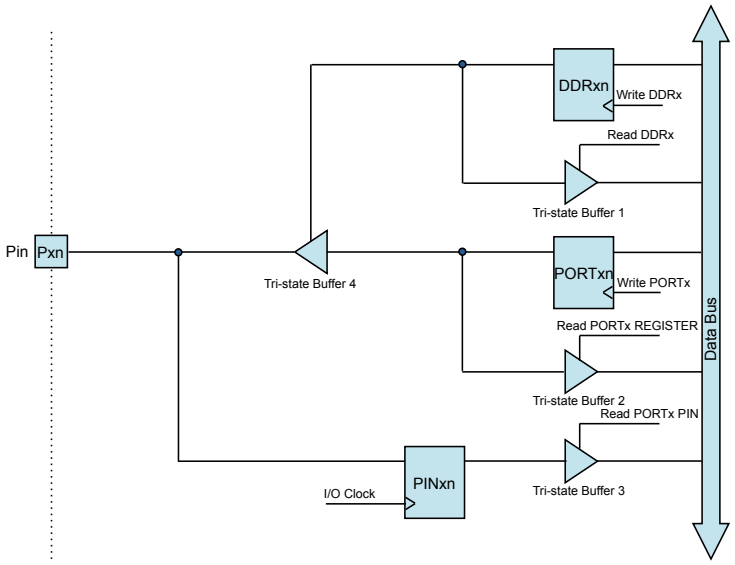
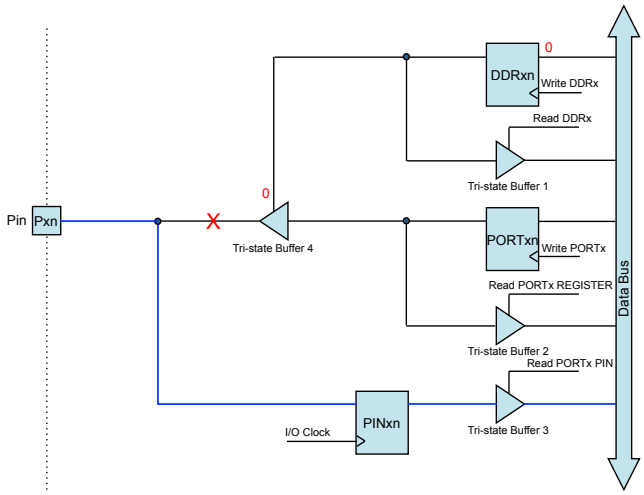


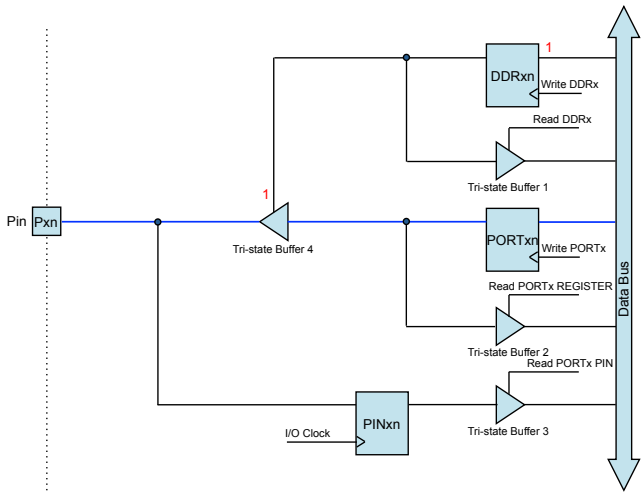
Figure 5.2: A single pin of a port.

Fig. 5.3 illustrates the configuring of an I/O line for input and output. In Fig. 5.3(a), writing 0 on DDRxn causes the output of the tri-state buffer between Pxn and PORTxn to high-impedance state allowing the signal on Pxn to be latched to PINxn. Fig. 5.3(b) shows how the pin can be configured for output. Writing a 1 into DDRxn enables the tri-state buffer, which provides a direct connection between PORTxn and Pxn.

The code below shows how PORTA can be configured for input and output. Figure 5.3 illustrates this process. In Figure 5.3(a), a 0 is written to DDxn to disable the tri-state buffer, which provides a direct path between Pxn and PINxn. This allows any signal on the pin to be latched onto PINxn at the end of each I/O clock. In Figure 5.3(b), an 1 is written to DDRxn to enable the tri-state buffer, which provides a direct path between PORTxn and Pxn. This allows any value written to PORTxn to appear on the pin. Note that the I/O instruction `out` is used to move the content of `r16` to DDRA, which is one of the 64 I/O registers. The `mov` (*Move between registers*) instruction cannot be used in this case because it only moves data between two General Purpose Registers (GPRs). In addition, input is performed by reading from PINA using the `in` (*In Port*) instruction, and output is performed by writing to PORTA using the `out` (*Out Port*) instruction.



(a) Configuring for input.



(b) Configuring for output.

Figure 5.3: Reading and writing to a port.

```
; Assembly code for setting PORTA for input
      ldi   r16, $00
      out   DDRA, r16    ; Write 0s to DDRA to set up for input
      ...
      in    r16, PINA    ; Then use this instruction to read from PINA
```

```
; Assembly code for setting PORTA for output
      ldi   r16, $ff
      out   DDRA, r16   ; Write 1s to DDRA setup for output
      ...
      out   PORTA, r16  ; Then use this instruction to write to PORTA
```

Now that we have discussed how to configure the I/O ports and perform input and output, we are ready to discuss a more elaborate example of an I/O operation. *TekBots* are programmable robots developed at School of Electrical Engineering and Computer Science, Oregon State University. Tekbots are used by several universities to help students learn some of the fundamental concepts that are found in the fields of computer and electrical engineering. Figure 5.4 shows a picture of Tekbot, which is controlled by a AVR microcontroller.

Example 5.1. *Tekbot has two whiskers (left and right) that detect bumps. Right whisker input connected to PORTE pin 4. Left whisker input connected to PORTE pin 5. Detection of a bump initiates a routine to turn the Tekbot around. Right/Left engine enable connected to PORTB pins 4/7, where 0=go, 1=stop. Right/Left engine direction connected to PORTB pins 5/6, where 1=forward, 0=backward.*

The code shown in Figure 5.5 controls the basic functionalities of Tekbots. The program consists of five main parts: (1) Interrupt Vectors, (2) Program Initialization, (3) Main Program, (4) HitRight subroutine, and (5) Wait subroutine. Note that HitLeft subroutine, which is similar to HitRight subroutine, has been omitted to save space.

Interrupt Vectors

The very first part of the program contains two `.org` directives to (1) allow `rjmp INIT` to be the first instruction executed when the Tekbot is turned on and (2) place the rest of the code starting at location `$0046`. When Tekbot is turned on (or reset), the AVR microcontroller by default sets PC to `$0000` (see Section 5.3.1). This causes the processor to fetch and execute the instruction located at `$0000`, which is the `rjmp INIT` instruction. The `rjmp` instruction unconditionally jumps to label `INIT` that contains the code for initializing the I/O registers. As we will see in Section 5.3.1, the first 70 ($46) locations in the Program Memory are dedicated for interrupts, called *interrupt vectors* (see Section 5.3), and thus placing any code in this address

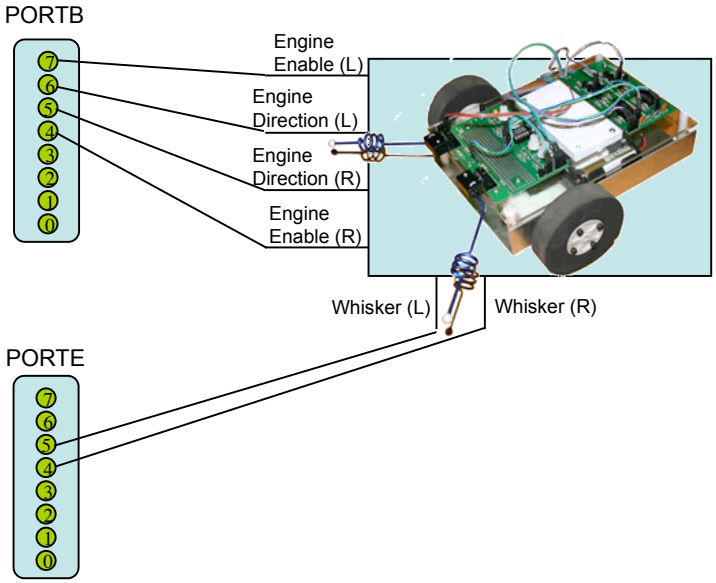


Figure 5.4: Connection of whiskers and wheels to PORTB and PORTE in Tekbot.

space may cause unwanted behavior. Thus, `.org $0046` causes the rest of the program to be placed after the interrupt vectors.

Program Initialization

When Tekbot, and thus the processor, is turned on, ports connected to the wheels and whiskers have to be appropriately configured. As shown in Figure 5.4, the engine enable and engine direction signals for the left and right wheels are connected to bits 7-4 of PORTB. These pins need to be set for output to control the engines. Note that one of the first things that needs to be done during initialization is to set up the stack. The *stack* is needed to store return addresses of subroutine calls and store and restore register values. This part of the code has been omitted to simplify the discussion, and will be discussed in more detailed in Section 5.3. For now, we will assume the stack has been set up.

The first two instructions set the bits 7-4 of the `DDRB` register to 1s ensuring the corresponding bits in `PORTB` will function as outputs. The next two instructions clear the `PORTB` register, which ensures that the engines

```
;-----------------------------------------------------------
; Interrupt Vectors
;-----------------------------------------------------------
      .org  $0000             ; Reset and Power On Interrupt
      rjmp  INIT              ; Jump to program initialization
      .org  $0046             ; End of Interrupt Vectors
;-----------------------------------------------------------
; Program Initialization
;-----------------------------------------------------------
INIT:
;; Initialize Stack ;;
; Initialize Port B for output
      ldi   r16, 0b11110000   ; Set Port B Directional Register
      out   DDRB, r16         ; for output
      ldi   r16, $00          ; Initialize Port B to 0s
      out   PORTB, r16        ; Port B outputs low
; Initialize Port E for inputs
      ldi   r16, $00          ; Set Port E Directional Register
      out   DDRE, r16         ; for input
      ldi   r16, 0b00110000   ; Activate pull-up registers
      out   PORTE, r16        ;
; Initialize TekBot Forward Movement
      ldi   r16, 0b01100000   ; Load Move Forward Command
      out   PORTB, r16        ; Send command to motors

;-------------------------------------------------------------
; Main Program
;-------------------------------------------------------------
MAIN:
      in    r16, PINE         ; Get whisker input from Port E
      andi  r16, 0b00110000   ; Mask out other bits
      cpi   r16, 0b00010000   ; Check for Right Whisker input
      brne  NEXT              ; Continue with next check
      rcall HitRight          ; Call the subroutine HitRight
      rjmp  MAIN              ; Continue with program
NEXT:
      cpi   r16, 0b00100000   ; Check for Left Whisker input
      brne  MAIN              ; No Whisker input, continue
      rcall HitLeft           ; Call subroutine HitLeft
      rjmp  MAIN              ; Continue through main

;---------------------------------------------------------------
; Sub:      HitRight
; Desc:     Handles functionality of the TekBot when the right whisker
;           is triggered.
;---------------------------------------------------------------
```

Figure 5.5: Code for Tekbot movement.

```
HitRight:
;;Save mpr, wait, SREG registers
; Move Backwards for a second
      ldi   r16, $00          ; Load Move Backwards command
      out   PORTB, r16        ; Send command to port
      ldi   waitcnt, Wtime    ; Wait for 1 second (WTime=100, waitcnt=r17)
      rcall Wait              ; Call wait function
; Turn left for a second
      ldi   r16, 0b01000000   ; Load Turn Left Command (PORTB, pin 6)
      out   PORTB, r16        ; Send command to port
      ldi   waitcnt, WTime    ; Wait for 1 second
      rcall Wait              ; Call wait function
; Move Forward again
      ldi   r16, 0b01100000   ; Load Move Forward command
      out   PORTB, r16        ; Send command to port
;;Restore mpr, wait, SREG registers
      ret                     ; Return from subroutine

;-----------------------------------------------------------------
; Sub:      Wait
; Desc:     A wait loop that is 16 + 159975*waitcnt cycles or roughly
;           waitcnt*10ms.  Just initialize wait for the specific amount
;           of time in 10ms intervals. Here is the general equation
;           for the number of clock cycles in the wait loop:
;                 ((3 * ilcnt + 3) * olcnt + 3) * waitcnt + 13 + call
;-----------------------------------------------------------------
Wait:
;; Save wait, r18, r19 registers
OLoop:
      ldi   r19, 224          ; (1) load outer-loop counter register
MLoop:
      ldi   r18, 237          ; (1) load inner-loop counter register
ILoop:
      dec   r18               ; (1) decrement ilcnt
      brne  Iloop             ; (2/1) Continue Inner Loop
      dec   r19               ; (1) decrement olcnt
      brne  Mloop             ; (2/1) Continue Outer Loop
      dec   waitcnt           ; (1) Decrement wait
      brne  OLoop             ; (2/1) Continue Wait loop
;; restore wait, r18, r19 registers
      ret                     ; Return from subroutine
```

Figure 5.5: Code for Tekbot movement (cont.).

are disabled. Note that when reset occurs, all I/O registers are set to their default values, which is either 0 or N/A (i.e., random) depending on the I/O register. For the Atmega128 processor, `PORTD` and `PORTE` are reset to 0. Thus, explicitly clearing the `PORTB` register may seem redundant. However, many of the older AVR processors do not reset the ports and thus these two instruction provide backward compatibility.

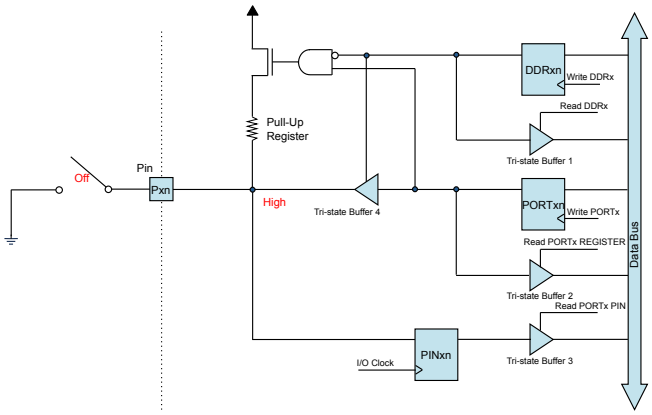
The same is true for initializing `PORTE`, except it will be set for input. However, detecting when whiskers are triggered requires the use of *pull-up register*. A whisker is nothing more than a passive switch. That is, when a whisker is hit, it turns on a switch; otherwise the switch is off. However, turing on and off the switch need to be translated into 0 and 1, so that the processor can detect this on its I/O pin. Figure 5.6 shows how pull-up register is used to latch 1 or 0 onto the PINxn when the switch is turned off or on, respectively

Figure 5.6(a) shows that when the whisker is not triggered (i.e.., the switch is off), the voltage at the input of PINxn is very close to the supply voltage, which is considered high. When the whisker is triggered and the switch closes, the voltage is pull down to ground, which is considered low. Therefore, when the whisker is triggered, 0 is latched onto PINxn; otherwise, 1 is latch onto PINxn. The pull-up registers is activated by writing a 1 onto PORTxn together with setting DDRxn to 0 for input. This cause the pass transistor located between the supply voltage and the pull-register to turn on.

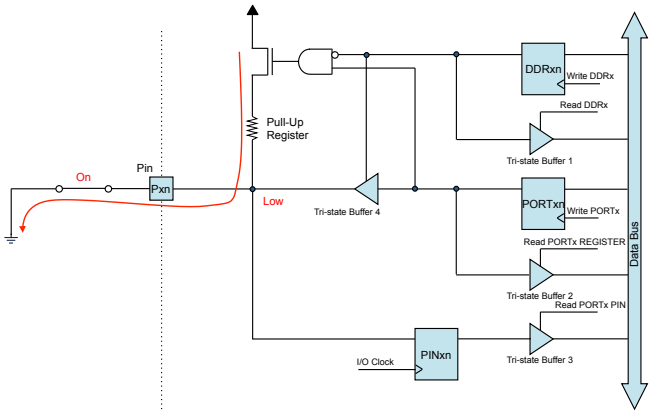
Once the ports are configured, writing $60 or 0b01100000 to `PORTB` causes the Tekbot to move forward. This is because the 6th and 5th bits that control engine direction are set to 1s and 7th and 4th bits that control direction are set to 0s.

Main Program

The Main Program is simply a loop that checks whether the right or left whisker has been triggered. If a whisker is hit, the corresponding PINE latch will be cleared, i.e., bit 4 and bit 5 for the right and left whiskers, respectively. Therefore, the Main Program checks the two whiskers in succession by (1) loading in the content of PINE, (2) masking out unrelated bits, and (3) comparing to see if the corresponding bit is zero, and if so, execute `HitRight` or `HitLeft` subroutine.



(a) Configuring for input.



(b) Configuring for output.

Figure 5.6: Triggering of a whisker.

HitRight

When the right whisker is hit, the Tekbot moves backward for a second, then turns left for a second, and then moves forward. This is achieved by first setting both wheels to move backwards (i.e., `PORTB6`=0 and `PORTB5`=0), and then setting the left wheel to move forward and the right wheel to move backward (i.e., `PORTB6`=1 and `PORTB5`=0), and finally setting both wheels to move forward (i.e., `PORTB6`=1 and `PORTB5`=1). In addition, moving backwards and turning left each requires a duration of one second. This is achieved by calling the `HitRight` subroutine.

Wait

The `Wait` subroutine is specifically coded to execute for approximately one second (0.99962 sec to be exact!). The code structure is a triply nested loop where each iteration of the outer-most loop takes around 10 ms to execute. Thus, since the value of `WTime` that was passed to register `waitcnt`is 100, the triply-nested loop executes for `waitcnt`×10 ms = 1sec. Now how do we know how long each iteration takes to execute? The beauty of assembly programming is that the programmer knows (or can find out) the number clock cycles it takes to execute each instruction. Moreover, the *clock cycle time* or clock period is known based on the clock frequency used by the processor. For example, ATmega128 uses a 16 MHz clock rate, which leads to a clock cycle time of $62.5ns$.

Now going back to our `Wait` subroutine, the number of cycles each instruction takes in the triply-nested loop is know (indicated in parenthesis). For example, `ldi` and `dec` instructions each require one cycle to execute. On the other hand, the `brne` instruction requires one cycle if the branch is not taken, and two cycles if the branch is taken. Thus, the total number of cycles required to executed the triply-nested loop, T_{cyc} is given by the following equation:

$$T_{cyc} = (((((3 \times ilcnt) - 1 + 4) \times mlcnt) - 1 + 4) \times waitcnt) - 1 \qquad (5.1)$$

$(3 \times ilcnt) - 1$ portion of the equation represents the number of cycles for the inner-loop. This is because as long as the branch for the inner loop is taken, each iteration takes 1 + 2 cycles. Thus, the total number cycles required for the `ilcnt` iteration is $3 \times ilcnt$). The -1 term comes from the fact that the very last iteration of the inner-loop will be false and thus not taken, which requires only 1 cycle rather than 2. The middle-loop consists of the execution time for the inner-loop plus the `ldi r18,237` instruction before the inner-loop and `ldi r18,237` and `brne MLoop` after the inner-loop. These three instructions require 4 cycles as long as `brne MLoop` is taken. Therefore, $(((3 \times ilcnt) - 1 + 4) \times mlcnt) - 1$ represents the execution time for the middle loop with the assumption that the last branch instruction is not taken. The execution for the outer loop is evaluated in a similar manner that results in Equation 5.1. Finally, 7 cycles are required to execute `rcall Wait` (3 cycles) and `ret` (4 cycles). Therefore, the time required to execute the subroutine call to `Wait` and then return is 15,993,906 cycles × 62.5 ns =0.99962 sec.

5.3 Interrupts

In the code for Tekbot movement shown in Figure 5.5, the loop in the Main Program constantly checks whether a whisker is triggered. This method of checking for an external event is called *busy waiting*. Unfortunately, a processor busy-waiting expends all its processing power waiting for events to occur. A better method is to used interrupts. An *interrupt* is an unexpected event (in terms of timing) causing a change in the normal flow of the instruction execution. This allows the processor to execute a program and when an external event occurs suspend the program and service the external event and then resume the execution of the program.

An external event can come in many forms. Some examples are

- I/O device ready to accept more output
- USART (Universal Synchronous Asynchronous Receiver/Transmitter) transmit buffer empty.
- I/O device has input ready
- Key pressed on keyboard.
- Tekbot whiskers triggered.
- A timer generates interrupt, e.g., every 20 ms.

Note that interrupts are different from *traps*, which are caused by either events within the processor or by software. Some examples of traps are

- Overflow
- Divide by zero
- Undefined opcode
- SWI (software interrupt)

The main difference between interrupts and traps is that the latter is synchronous with respect to instruction execution, while the former is asynchronous. This means a trap is either caused or generated by an instruction, whereas an interrupt can occur at any time due to an external event. Moreover, interrupt handlers (service routines do not always return to the original program.

5.3.1 AVR Interrupt Facility

AVR processor has 35 interrupt sources:

- 8 external interrupt requests
- 4 Timers/Counters with compare and overflow
- ADC complete
- USART Tx/Rx complete
- Reset
- $\cdots$ and many more

External interrupts are triggered by pins INT7-INT0, and thus allow up to eighth I/O devices to be connected to the microcontroller. In contrast to external interrupts, Timers/Counters are integrated into the microcontroller and thus generate internal interrupts. *Analog-to-Digital Converter* (ADC), as the name suggest, converts an analog input signal to a 10-bit binary value. *Universal Synchronous Asynchronous Receiver/Transmitter* (USART) is a transceiver that translates data between parallel and serial forms (i.e., from bytes to bits and vice versa), and thus implements a serial port (e.g., RS-232) *Reset* is another external interrupt that gets triggered when the processor is powered on.

Now that we have discussed the sources of interrupt for the AVR microcontroller, we are ready to discuss how interrupts from these sources are handled. Figure 5.7 illustrates the interrupt handing process. First, the processor checks for an interrupt at the end of each instruction execution. If there is no interrupt, the processor fetches and executes the next instruction. If an interrupt occurred, the processor pushes the return address, which is the address of the next instruction to be fetched and executed, onto the stack. Next, the source of the interrupt is identified, which allows the appropriate *Interrupt Service Routine* (ISR) to be executed. After the ISR is executed, it returns to the point where the interrupt occurred and the user program execution resumes.

The following discusses each of these steps in detail.

Interrupt Detection

Before using the interrupt facility of AVR microcontroller, it needs to be configured. The first thing that needs to be done is to turn on the interrupts.

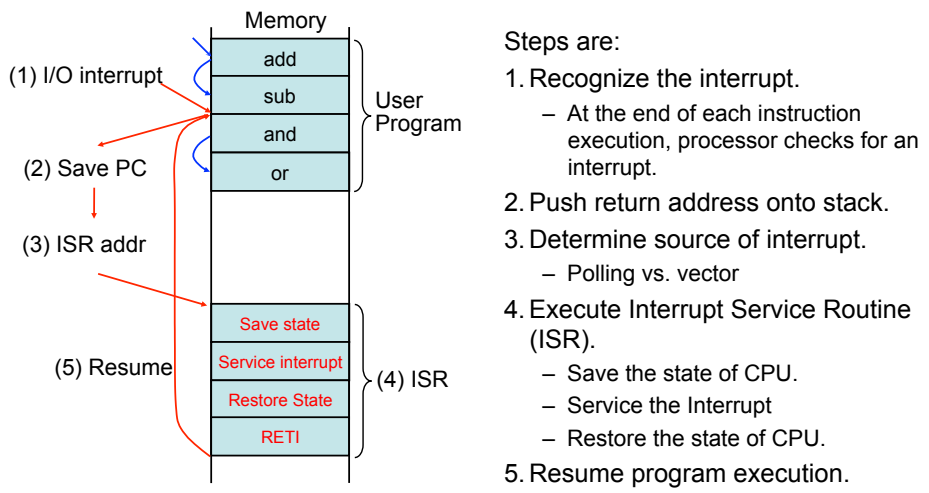


Figure 5.7: The process of handling interrupts.

This is achieved by setting the MSB of the `SREG`, called the *Global Interrupt Enable* (I-bit), shown in Fig. 5.8. As the name suggests, I-bit needs to be set to allow interrupts to be detected. AVR has special instructions that allow the I-bit to be directly set (`SEI`) or cleared (`CLI`). In addition, the I-bit is cleared after an interrupt is detected and set by *Return From Interrupt* (`RETI`) instruction. Therefore, the default behavior is to not allow other interrupts from occurring while an interrupt is being serviced. This behavior can easily be changed by clearing the I-bit. For example, different interrupts can be prioritized allowing a higher priority device interrupt a lower priority device.

Interrupts can be either external or internal. *External* interrupts are generated by I/O devices connected to and triggered by pins INT7 - INT0. As shown in Fig. 5.9, INT7-4 are connected to PORTE pins 7-4, while INT3-0 are connected to PORTD pins 3-0. On the other hand, *internal* interrupts are generated by timers and counters within the microcontroller. As the names suggest, timers/counters are used to specify some time interval before the processor is notified (i.e., interrupted).

We first discuss the detection of external interrupts. AVR has the following set of I/O registers that control how, when, and whether external interrupts are detected:

- External Interrupt Flag Register (*EIFR*)

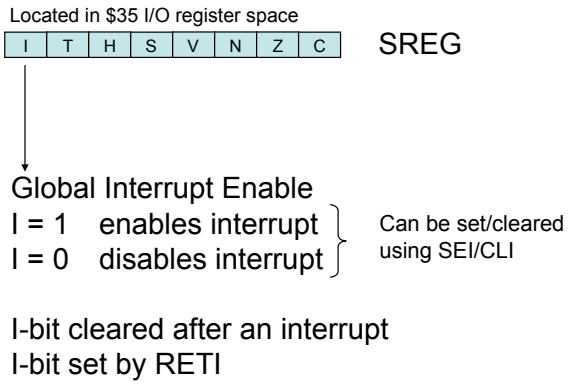


Figure 5.8: I-bit in the SREG.

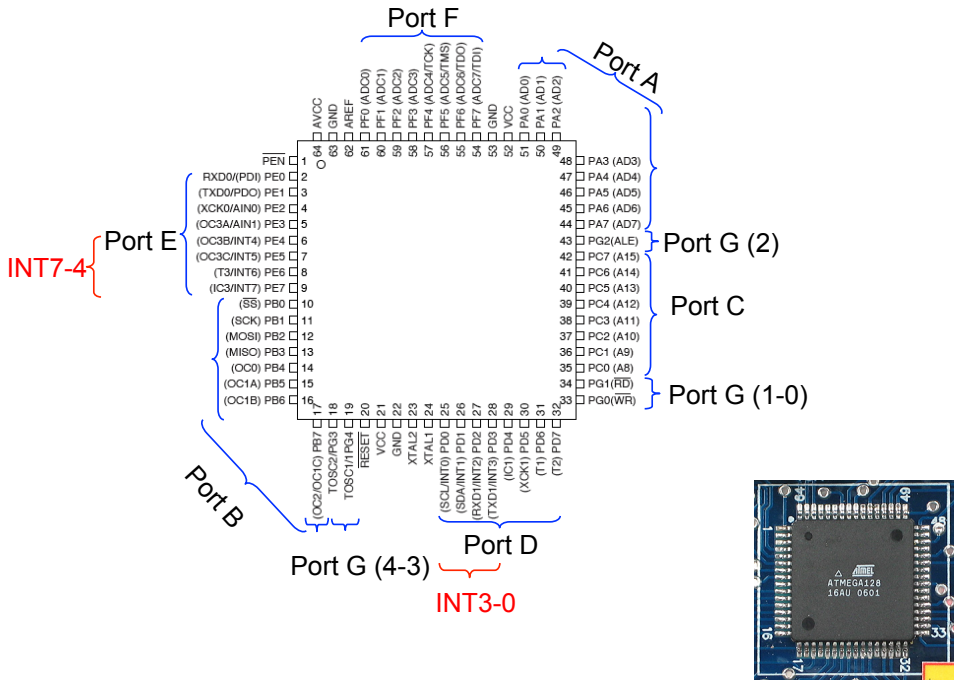


Figure 5.9: I/O Pins.

- External Interrupt Mask Register (*EIMSK*)
- External Interrupt Control Registers (*EICRA* & *EICRB*)

The eight external interrupts INT7 - INT0 are detected by the processor by latching their signals to *External Interrupt Flag Register* (EIFR). These signals can be masked using *External Interrupt Mask Register* (*EIMSK*), which basically allows each interrupts to be detected or not detected. The interrupts can be triggered by a falling edge, rising edge, or low level input signal by setting up by *External Interrupt Control Registers* (EICR), which consists of a pair of 8-bit registers *EICRA* and *EICRB*.

Fig. 5.12 illustrates conceptually how these registers are organized. EIFR on the bottom of the figure latches interrupts from INT7-INT0. EIFR detects one or more interrupts and then transfers the control to an *Interrupt Service Routine* (ISR). How the transfer of control is done will be discussed in Sec. 5.3.1.

The *EIMSK* register is used to mask out unwanted interrupts. This can be thought of as a safety measure to prevent an unwanted event (e.g., static electricity) from triggering an interrupt.

A pair of bits controls how an external interrupt is detected, which are referred to as *Interrupt Sense Control* bit 1 and bit 0 (ISCn1 and ISCn0). Thus, there are 16 ISCn bits spread across EICRA and EICRB. EICRA senses interrupts for INT3-INT0, while EICRB sense interrupts for INT7-INT4. EICRA has three options available for controlling how INT3-INT0 are detected; low-level (00), falling edge (10), and rising edge (11). On the hand, EICRB provides one additional option for INT7-INT4,i.e., any level change (01).

Internal interrupts consist mainly of Timers/Counters. They are

- Two 8-bit *Timer/Counter0* and *2* (TCNT0 & 2).
- Two 8-bit *Output Compare Register 0* and *2* (OCR0 & 2)
- Two 16-bit *Timer/Counter1* and *3* (TCNT1 & 3).
- Two 16-bit *Output Compare Register 1* and *3* (OCR1 & 3) is used to compare the counter value.

These TCNT registers are used in conjunction with their corresponding OCR registers to generate an interrupt if TCNT register counts up to a value store in the OCR registers. Section 5.4 will discuss the variety of ways the Timers/Counter can be used.

Determining the Source of Interrupt

When an interrupt occurs, each interrupt source is mapped to a vector, i.e., an address in the program memory. This allows the flow control to be

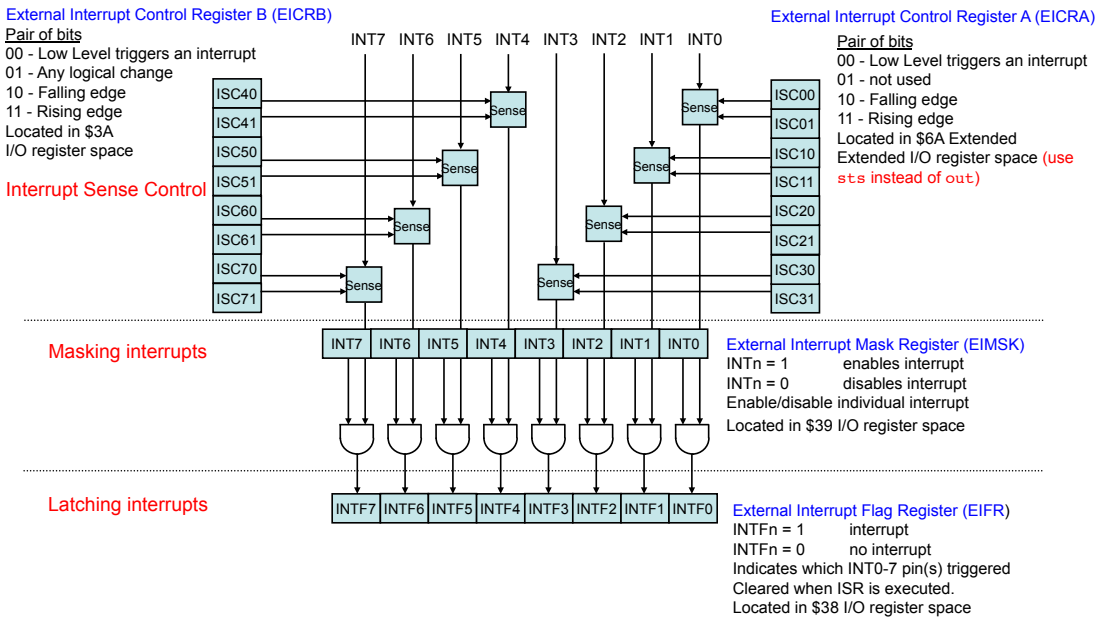


Figure 5.10: Controlling interrupts.

redirected to an appropriate vector depending the source of the interrupt. Table 5.1 shows the *interrupt vector table* and how various interrupt sources are mapped to the program memory.

RESET is mapped to *hardware reset* and has the highest priority. This is followed by the eight external interrupts (INT0-INT7). TIMER2 COMP represents Timer/Counter2 compare match condition. TIMER2 OVF represents Timer/Counter2 overflow condition. There are total 35 vectors for the 35 interrupt sources.

Each vector is allocated two program memory words (i.e., 32 bits), which allows either a jump or subroutine call to be placed to transfer the control flow to a desired *interrupt service routine* (ISR). Figure 5.11 shows and example of how interrupt vectors may be set up. In this example, when a reset condition becomes active, the processor detects the interrupt and loads the program counter with the address of the vector (i.e., $0000). This causes the processor to jump to a routine located at label `RESET`, which is usually an initialization routine to set up the microcontroller.

Table 5.1: Interrupt vectors

Vector. No.	Program Address	Source	Interrupt Definition
1	$0000	RESET	Hardware Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7
10	$0012	TIMER2 COMP	Timer/Counter2 Compare Match
11	$0014	TIMER2 OVF	Timer/Counter2 Overflow
12	$0016	TIMER1 CAPT	Timer/Counter1 Capture Event
13	$0018	TIMER1 COMPA	Timer/Counter1 Compare Match A
14	$001A	TIMER1 COMPB	Timer/Counter1 Compare Match B
15	$001C	TIMER1 OVF	Timer/Counter1 Overflow
16	$001E	TIMER0 COMP	Timer/Counter0 Compare Match
17	$0020	TIMER0 OVF	Timer/Counter0 Overflow
18	$0022	SPI, STC	SPI Serial Transfer Complete
19	$0024	USART0, RX	UASRT0, Rx Complete
20	$0026	USART0, UDRE	USART0 Data Register Empty
21	$0028	USART0, TX	USART0, Tx Complete
22	$002A	ADC	ADC Conversion Complete
23	$002C	EE READY	EEPROM Ready
24	$002E	ANALOG COMP	Analog Comparator
25	$0030	TIMER1 COMPC	Timer/Counter1 Compare Match C
26	$0032	TIMER3 CAPT	Timer/Counter3 Capture Event
27	$0034	TIMER3 COMPA	Timer/Counter3 Compare Match A
28	$0036	TIMER3 COMPB	Timer/Counter3 Compare Match B
29	$0038	TIMER3 COMPC	Timer/Counter3 Compare Match C
30	$003A	TIMER3 OVF	Timer/Counter3 Overflow
31	$003C	USART1, RX	USART1, Rx Complete
32	$003E	USART1, UDRE	USART1 Data Register Empty
33	$0040	USART1, TX	USART1, Tx Complete
34	$0042	TWI	Two-wire Serial Interface
35	$0044	SPM READY	Store Program Memory Ready

Saving and restoring Processor State

As mentioned before, an interrupts is an unexpected event from the point-of-view of the user program. Therefore, servicing an interrupt must be non-intrusive to the user program, i.e., the state of the processor must not be changed. This way, when the control returns to the user program after executing ISA, the state of the processor would be identical to the way it was before the interrupt was serviced The state of the processor is defined by PC, SR, and GPRs, and thus must be stored before and after the interrupt is serviced.

The PC serves as the pointer to the instruction that will be executed

```
Address     Label       Code
$0000                   JMP    RESET
$0002                   JMP   EXT_INT0    ; IRQ0
$0004                   JMP   EXT_INT1    ; IRQ1
...                           ...               ...
$0010                   JMP   EXT_INT7    ; IRQ7
$0012                   JMP   TIME2_COMP
$0014                   JMP   TIME2_OVF
...                           ...               ...
...                           ...               ...
$????       EXT_INT0:   { Your ISR for External Interrupt 0 }
```

Figure 5.11: An example code.

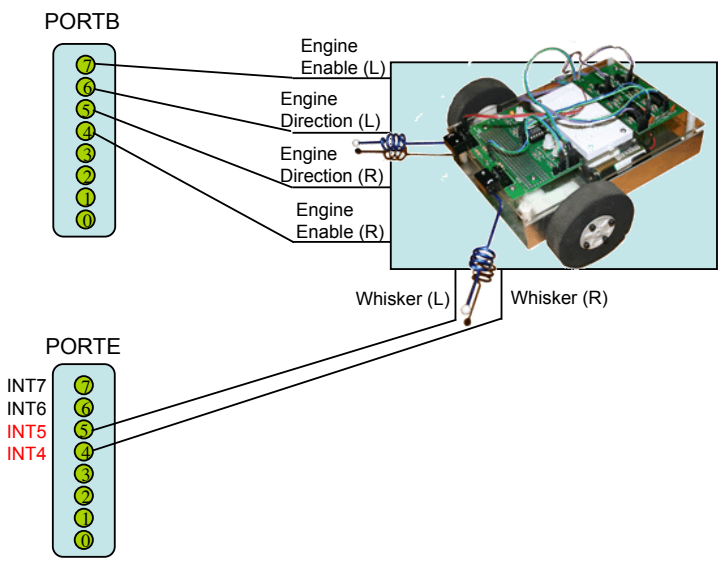


Figure 5.12: Controlling interrupts.

upon returning from an ISR. Therefore, the processor pushes PC onto the stack and clears I-bit in SREG, which disable the interrupt. After executing the ISR, `RETI` (*Return from interrupt*) automatically restores PC from stack and sets I-bit in SREG, which enables the interrupt. On the other hand GPRs and SREG may need to be saved if ISR uses them, and is the responsibility of the programmer. These registers are also saved on the stack

using using PUSH and POP instructions.

We conclude this subsection by presenting an interrupt base code shown in Figure 5.13 for controlling the Tekbot movements.

At a first glance, this code looks more complicated than the busy-waiting version. However, this has to do more with the extensive use of directives and expressions (see Chapter 4.8) than the code itself.

The code start with a series of .def and .equ directives that assign symbolic names to registers (e.g., mpr, waitcnt, ilcnt, and olcnt) and values to labels (e.g., WTime, WskrR, WskrL, etc.) to make it easier to keep track of the code. The last few .equ directives not only assign values to lables, but they also use expressions to evaluate these values. For example, consider the following directive:

```
.equ  MovFwd = (1<<EngDirR|1<<EngDirL)    ; Move Forwards Command
```

The expression (1<<EngDirR|1<<EngDirL) takes two binary numbers 00000001 and 00000001, and shifts them left (defined by the '<<' operator) by EngDirR and EngDirL bit positions, which in turn was assigned as 5 and 6, respectively. This results in binary numbers 00100000 and 01000000. Finally, these two numbers are logically ORed (defined by the '|' operator) to generate the binary number 01100000, which is assigned to MovFwd.

Next, the interrupt vectors for INT5 and INT4, which are connected to the left and right whiskers, respectively, are set up to transfer the control to the appropriate ISR. This is done by using .org $000A and .org $000C to place subroutine calls (rcall) to ISRs HitRight and HitLeft. Note that each of these subroutine call is immediately followed by the reti instruction, which sets the I-bit in the SREG and returns to the instruction after the interrupt was detected.

The Program Initialization sections starts by setting up a stack. As mentioned before, the stack is an important data structure for storing and restoring of return addresses of subroutine calls and register values. Figure 5.15 illustrates the process of setting up the stack. The expressions high(RAMEND) and low(RAMEND) evaluate the high and low bytes of the address of the very last location in Data Memory, i.e., location &10FF. This location is defined by label RAMEND (*end of memory*), which is defined in the m128def.inc include file. In addition, this include file also defines SPH (Stack Pointer high) and SPL (Stack Pointer low) as the the high and low bytes of the SP register. Therefore, SP initialization portion of the code moves 16-bits address that points to the last location in Data Memory into the SP register.

```
.include "m128def.inc"              ; Include definition file
;***********************************************************
;*    Internal Register Definitions and Constants
;***********************************************************
.def  mpr = r16                     ; Multipurpose register
.def  waitcnt = r17                 ; Wait Loop Counter
.def  ilcnt = r18                   ; Inner Loop Counter
.def  olcnt = r19                   ; Outer Loop Counter
.equ  WTime = 100                   ; Time to wait in wait loop
.equ  WskrR = 4                     ; Right Whisker Input Bit
.equ  WskrL = 5                     ; Left Whisker Input Bit
.equ  EngEnR = 4                    ; Right Engine Enable Bit
.equ  EngEnL = 7                    ; Left Engine Enable Bit
.equ  EngDirR = 5                   ; Right Engine Direction Bit
.equ  EngDirL = 6                   ; Left Engine Direction Bit
.equ  MovFwd = (1<<EngDirR|1<<EngDirL)    ; Move Forwards Command
.equ  MovBck = $00                  ; Move Backwards Command
.equ  TurnR = (1<<EngDirL)          ; Turn Right Command
.equ  TurnL = (1<<EngDirR)          ; Turn Left Command
.equ  Halt = (1<<EngEnR|1<<EngEnL)  ; Halt Command
;***********************************************************
;*    Start of Code Segment
;***********************************************************
      .cseg             ; Beginning of code segment
;-----------------------------------------------------------
; Interrupt Vectors
;-----------------------------------------------------------
      .org  $0000       ; Beginning of IVs
      rjmp  INIT        ; Reset interrupt
      .org  $000A {IRQ4 => pin4, PORTE}
      rcall HitRight          ; Call hit right function
      reti                    ; Return from interrupt
      .org  $000C {IRQ5 => pin5, PORTE}
      rcall HitLeft           ; Call hit left function
      reti                    ; Return from interrupt
      .org  $0046             ; End of Interrupt Vectors
;-----------------------------------------------------------
; Program Initialization
;-----------------------------------------------------------
INIT: ; The initialization routine
      ; Initialize Stack Pointer
      ldi   mpr, high(RAMEND)
      out   SPH, mpr
      ldi   mpr, low(RAMEND)
      out   SPL, mpr
```

Figure 5.13: Interrupt-based code for Tekbot.

```
      ; Initialize Port B for output
      ldi   mpr, (1<<EngEnL)|(1<<EngEnR)|(1<<EngDirR)|(1<<EngDirL)
      out   DDRB, mpr         ; Set the DDR register for Port B
      ldi   mpr, $00
      out   PORTB, mpr        ; Set the default output for Port B
      ; Initialize Port E for input
      ldi   mpr, (0<<WskrL)|(0<<WskrR)
      out   DDRE, mpr         ; Set the DDR register for Port E
      ldi   mpr, (1<<WskrL)|(1<<WskrR)
      out   PORTE, mpr        ; Set the Port E to Input with Hi-Z
      ; Initialize external interrupts
      ; Set the Interrupt Sense Control to falling edge
      ldi   mpr, (1<<ISC41)|(0<<ISC40)|(1<<ISC51)|(0<<ISC50)
      out   EICRB, mpr        ; Set INT4 & 5 to trigger on falling edge
      ldi   mpr, $00
      sts   EICRA, mpr        ; Use sts, EICRA in extended I/O space
      ; Set the External Interrupt Mask
      ldi   mpr, (1<<INT4)|(1<<INT5)
      out   EIMSK, mpr
      ; Turn on interrupts
      sei
;-----------------------------------------------------------
; Main Program
;-----------------------------------------------------------
MAIN:  ; The Main program
      ; Move Robot Forward
      ldi   mpr, MovFwd       ; Load FWD command
      out   PORTB, mpr        ; Send to motors
      rjmp  MAIN              ; Create an infinite while loop
                                    to signify the end of the program.
;-----------------------------------------------------------------
; Sub:      HitRight
; Desc:     Handles functionality of the TekBot when the right whisker
;           is triggered.
;----------------------------------------------------------------
HitRight:
      push  mpr               ; Save mpr register
      push  waitcnt           ; Save wait register
      in    mpr, SREG         ; Save program state
      push  mpr               ;
      ; Move Backwards for a second
      ldi   mpr, MovBck       ; Load Move Backwards command
      out   PORTB, mpr        ; Send command to port
      ldi   waitcnt, WTime    ; Wait for 1 second
      rcall Wait              ; Call wait function
      ; Turn left for a second
      ldi   mpr, TurnL        ; Load Turn Left Command
      out   PORTB, mpr        ; Send command to port
      ldi   waitcnt, WTime    ; Wait for 1 second
      rcall Wait              ; Call wait function
      pop   mpr               ; Restore program state
      out   SREG, mpr         ;
      pop   waitcnt           ; Restore wait register
      pop   mpr               ; Restore mpr
      ret                     ; Return from subroutine
```

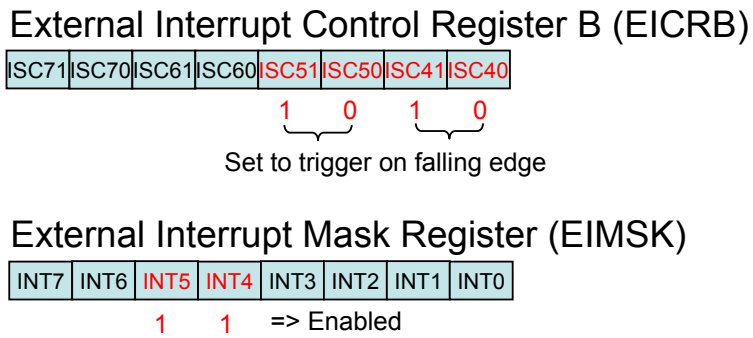


Figure 5.14: Initializing interrupts.

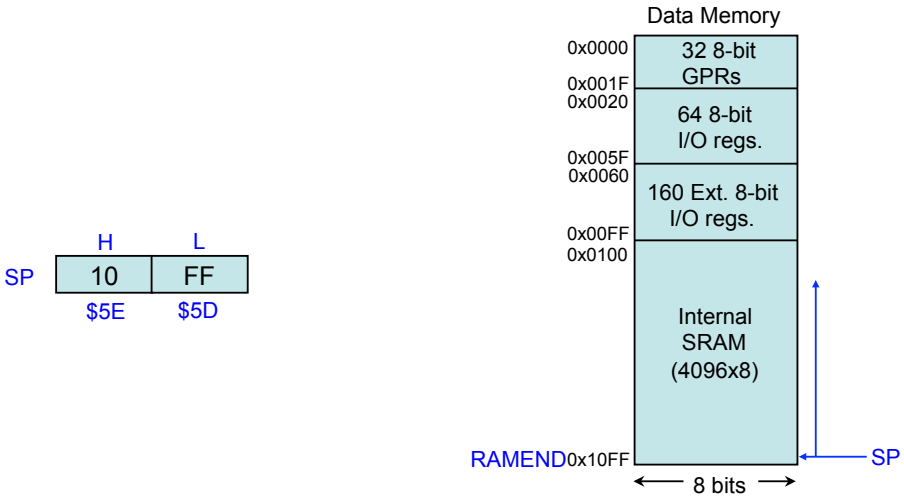


Figure 5.15: Initializing the stack.

The rest of the Program Initialization involves setting up the ports and interrupt handling. Initialization of ports are similar to the busy-waiting version, except that expressions are used to define bit patterns for Data Direction Registers `DDRB` and `DDRE`.

Initialization of the external interrupts are done by appropriately setting bits for `EICRA`, `EICRB`, and `EIMSK` registers. Figure **??** illustrates the initialization of INT5 and INT4 to trigger on a falling edge. The instructions shown below uses labels `ISC51`,`ISC50`, `ISC41`, and `ISC40` to define the bit

positions 3-0 within the `EICRB` register. Note that these labels are predefined in the `m128def.inc` include file. This combined with shift operations generates the bit pattern 00001010, which is then written to the `EICRB` register. As shown in Figure 5.12, this causes interrupt signals on INT5 and INT4 to be triggered on a falling edge.

```
ldi   mpr, (1<<ISC41)|(0<<ISC40)|(1<<ISC51)|(0<<ISC50)
out   EICRB, mpr        ; Set INT4 & 5 to trigger on falling edge
```

Similarly, the instructions shown below uses labels `INT5`,and `INT4` to define the 5th and 4th bit positions within the `EIMSK` register, to generate the bit pattern 00110000. Again, these labels are defined in the `m128def.inc` include file. When this value is written into the `EIMSK` register, it masks our any other interrupts except for INT5 and INT4. The last part of the Program Initialization is to enable the interrupt using `sei` (*set interrupt*).

```
ldi   mpr, (1<<INT4)|(1<<INT5)
out   EIMSK, mpr
```

The HitRight routine is basically same as the busy-wait version, but includes additional instructions to store the SREG and registers that were used by the main program before the interrupt occurred. This is achieved much pushing `mpr` (`r16`),`waitcnt` (`r17`), and `SREG` registers onto the stack. Note that pushing `SREG` onto the stack requires first moving it to a GPR (`mpr`) and then pushing it onto the stack. This is because the `SREG` register is located in the I/O register address space (locations $5E and $5D for SPH and SPL, respectively), and thus the I/O instruction `in` has to be used to move it to a GPR. Moreover, the `push` instruction only works on GPRs.

Finally, with the interrupt facility setup, the Main Program is much simpler than the busy-waiting version. It simply involves writing the bit pattern 0110000, which is evaluated based on the shift operation and directives defined for `EngDirL`, `EngDirR`, and `MovFwd`, into the PORTB register to move the Tekbot forward.

5.4 Timers/Counters

Timer/Counters are one of the most commonly used feature in a microcontroller.

The AVR microcontroller has two *8-bit Timer/Counters* (TCNT0 and TCNT2) and two *16-bit Timer/Counters* (TCNT1 and TCNT3). Each tick of the system clock (f_{CLK}) increments/decrements the contents of these

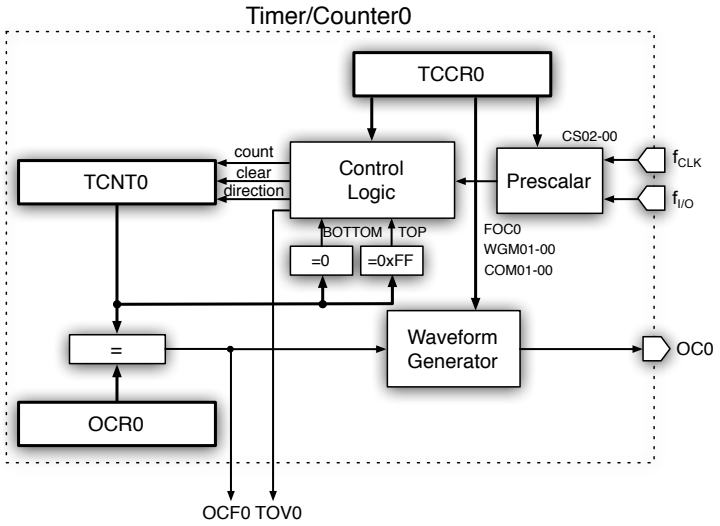


Figure 5.16: Block diaragram of Timer/Counter0.

registers. The most basic way to use a Timer/Counter is to write a value to it and then let it count up to its maximum number. When the count rolls over, an interrupt is generated. These Timers/Counters can also be used in conjunction with their corresponding *Output Compare Registers 0-3* (OCR0-3) to continually compare their contents with TCNT0-3. A match can be used to generate an *output compare interrupt* or a waveform output on one of the *Output Compare*(OC) pins.

5.4.1 TCNT0 and TCNT2

Figure 5.16 shows the block diagram of the Timer/Counter0. It consists of three user accessible registers: TCNT0, OCR0, and TCCR0. *TCNT0* is an 8-bit timer/counter, and is controlled by the *Control Logic* using three signals: the *count* signal enables the counting, the textitdirection signal selects between counting up and down, and the *clear* signal clears the Timer/-Counter to zero. The *Timer/Counter overflow 0* (TOV0) flag is set according to the mode of operation and can be used to generate a CPU interrupt. TCNT0 can also be used together with the *Output Compare Register 0* (OCR0) to continuously compare the two values, and if they are equal, a match is signaled by setting the *Output Compare Flag 0* (OCF0) at the next timer clock cycle.

The *Waveform Generator* uses the match signal to generate an output on the *Output Compare pin 0* (OC0) according to the operating mode. The *Prescalar* allows the I/O clock ($f_{I/O}$) to be scaled, which controls the rate at which a Timer/Counter is incremented. TCNT0 can also be asynchronously clocked from an external source. The exact behavior of the Timer/Counter depends on its mode of operation, which is set using the *Time Counter Control Register 0* (TCCR0).

TCNT2 is also an 8-bit Timer/Counter with features similar to TCNT0. The main difference between the two registers is that TCNT0 can be clocked by two external clock sources using *Timer Oscillator 1* (TOSC1) pin and *Timer Oscillator 2* (TOSC2) pin (which are bits 3 and 4 on Port G, respectively), while TCNT2 can be clocked by an external clock source on the T2 pin (which is the bit 6 on Port D). These external clock sources allow TCNT0 and TCNT2 to run asynchronously

There are several modes of operation with TCNT0. We will start with the discussion of the normal mode and then extend our discussion to other modes that provide more advanced capabilities. In addition, Table 5.2 shows the definitions used throughout this section of different values Timer/-Counter can have.

Table 5.2: Definitions for BOTTOM, MAX, and TOP

BOTTOM	The Timer/Counter reaches the BOTTOM when it becomes zero (i.e., 0x00).
MAX	The Timer/Counter reaches its MAX when it becomes 0xFF.
TOP	The Timer/Counter reaches the TOP when it becomes the highest value in the count sequence. The TOP value can be the value stored in the OCR0 register, or assigned to 0xFF (i.e., MAX). The assignment is dependent on the mode of operation.

Normal Mode

Figure 5.17 illustrates the basic operation in *normal mode*. The TCNT0 value in the figure illustrates the linear increase of the count starting from when TCNT0 is loaded with a *value* and when the counter rolls over (i.e., transitions from 0xFF to 0x00). When the counter rolls over, it interrupts the CPU by setting *Timer/Counter Overflow* flag (TOV0) in the *Timer/-Counter Interrupt Flag Register* (TIFR).

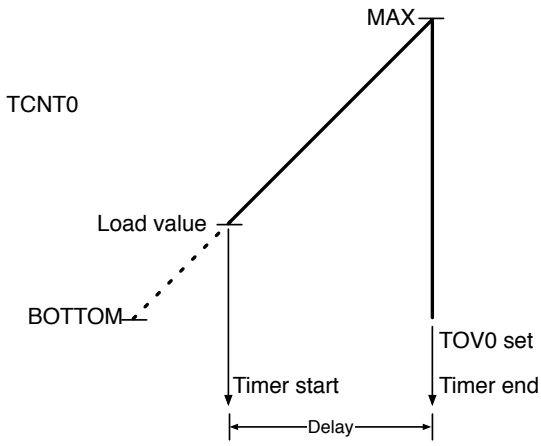


Figure 5.17: Timing diagram of normal mode.

The time or delay period between when TCNT0 is loaded with a value and TOV0 is set is given by the following equations:

$$Delay = \frac{(MAX - value) \cdot prescale}{f_{I/O}} \tag{5.2}$$

where *value* is the initial count, MAX represents 255 for TCNT0, $f_{I/O}$ represents the I/O clock, and *prescale* can be 8, 32, 64, 128, 256 or 1024 (see Table 5.3). The *prescale* can be adjusted using TCCR0, which will be discussed later.

Example 5.2. *Suppose we want to generate an interrupt after 5 ms using TCNT0 with $f_{I/O}$ given as 16 MHz.* {(BL): Need to look at this...}

16 MHz system clock leads to a clock period of $62.5ns$. In general, any combination of *value* and *prescale* that satisfies the following equation will work:

$$value = 255 - \frac{5ms}{prescale \times 62.5ns}$$

In general, we would like to use a prescale value that would lead to the highest resolution (i.e., lower prescale value) and then the period can be covered by the timer count. Since a prescale value of 256 only leads to 255×62.5 ns $= 4.08$ ms, a prescale of 1024 will be used. This leads to each tick being 64 μs. Therefore, the value to be loaded onto TCNT0 is $\lceil 255 - 78.1 \rceil = 177$, which leads to a delay period of 4.992 ms.

Clear Timer on Compare Match (CTC) Mode

In the *Clear Timer on Compare Match* (CTC) mode, the OCR0 register is used to control the counter resolution. The content of the OCR0 register defines the TOP value and thus its resolution. Moreover, TCNT0 is cleared to zero when TCNT0 and OCR0 match. An interrupt can be generated each time a match occurs by setting the *Output Compare Flag 0* (OCF0) in the *Timer/Counter Interrupt Flag Register* (TIFR) (see Figure 5.23). If the interrupt is enabled, an ISR can be used to adjust the TOP value. Moreover, with any interrupts, OCF0 can be enabled by setting the *Timer/Counter0 Output Compare Match Interrupt Enable* (OCIE0) bit in the *Timer/Counter Interrupt Mask* (TIMSK) register (see Figure 5.23).

Figure 5.18 shows an example CTC mode timing diagram. The TCNT0 value in the figure illustrates the linear increase in the counter value and short horizontal marks indicate when TCNT0 matches with OCR0, and the counter is cleared. The CTC mode can be used to generate a waveform on the *Output Compare 0* (OC0) pin, which is the pin 4 on PORT B (see Figure 5.9). This is done by setting the *Compare Output mode* bits (COM01=0 and COM00=0) in TCCR0. This causes the OC0 output to toggle its value each time TCNT0 and OCR0 match. It is important to note that OC0 pin is part of a port. Therefore, the data direction for the pin must be set to output to make signals visible on OC0 (see Section 5.2 for a description on how I/O pins are set for input/output). The frequency of the waveform generated in CTC mode, f_{CTC}, is controlled using the following equation:

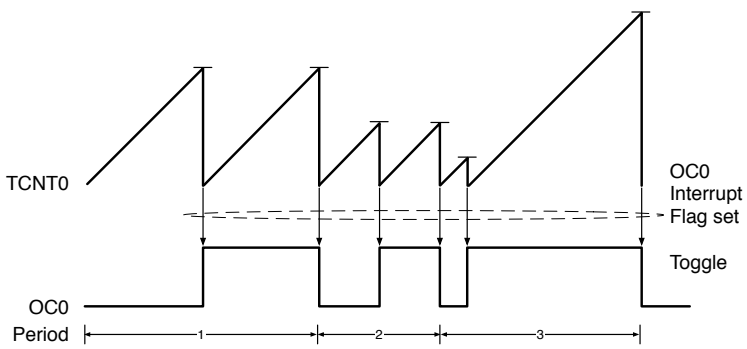


Figure 5.18: Example timing diagram in CTC mode.

$$f_{CTC} = \frac{f_{I/O}}{2 \cdot prescale \cdot (1 + OCR0)} \tag{5.3}$$

where $f_{I/O}$ is the I/O clock, *prescale* is the prescale factor defined in Table 5.3, and $OCR0$ is the value in the OCR0 register.

Fast Pulse Width Modulation (PWM) Mode

The *Fast Pulse Width Modulation* mode allows for generating high frequency Pulse Width Modulation (PWN) waves. Before discussing the details of this mode, some explanation is in order to understand the usefulness of this feature. *Pulse-width modulation* uses a rectangular pulse wave whose pulse width is modulated resulting in the variation of the average value of the waveform. This method is commonly used for driving motors, heaters, or lights in varying intensities or speeds.

Figure 5.19 shows an example Fast PWN mode timing diagram. In fast PWM mode (WGM01=1 and WGM00=1), the TCNT0 counts up from BOTTOM (0x00) to MAX (0xFF) then restarts at BOTTOM. For non-inverted PWM output (COM01=1 and COM00=0), the OC0 signal is cleared when TCNT0 and OCR0 match and is set when the counter transitions from 0xFF to 0x00. The inverse occurs using inverted PWM output (COM01=1 and COM00=1). The result is a waveform where its duty cycle can be varied by adjusting the values of OCR0.

The PWM frequency for the OC0 signal can be determined by using the following equation:

$$f_{PWM} = \frac{f_{I/O}}{prescale \cdot 256} \tag{5.4}$$

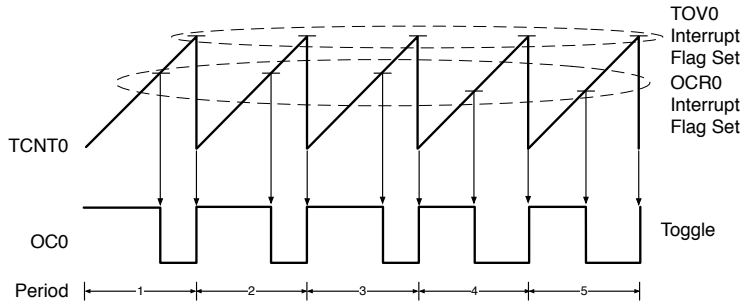


Figure 5.19: Example timing diagram in PWM mode.

7	6	5	4	3	2	1	0	
FOC0	WGM00	COM01	WGM01	COM00	CS02	CS01	CS00	TCCR0

Figure 5.20: Time Counter Control Register 0.

where $f_{I/O}$ is the I/O clock and *prescale* is the prescale factor defined in Table 5.3.

Phase Correct Pulse Width Modulation (PWM) Mode

{(BL): Work in progress...}

Time Counter Control Register

Controlling how TCNT0 behaves is done with the *Time Counter Control Register 0* (TCCR0) and the *Timer/Counter Interrupt Mask* (TIMSK) register. Figure 5.20 shows the format of the TCCR0 register.

Bits CS02, CS01, and CS00 allow internal frequencies to be derived or scaled from the I/O clock ($f_{I/O}$) using the *prescale* factor. This allows the rate at which a Timer/Counter is incremented/decremented to be controlled. Table 5.3 shows the eight possible choices.

Bits WGM01 and WGM00 define different modes of operation. This is shown in Table 5.4. In normal mode, the counting direction is alway up, and a value can be written to TCNT0 and when it counts up to MAX = 255, a timer overflow interrupt (i.e., TOV0=1) is generated. In CTC mode, the

Table 5.3: Clock Select bits for TCNT0

Clock Select bits			
CS02	CS01	CS00	Description
0	0	0	No clock source
0	0	1	$f_{I/O}$
0	1	0	$f_{I/O}/8$
0	1	1	$f_{I/O}/32$
1	0	0	$f_{I/O}/64$
1	0	1	$f_{I/O}/128$
1	1	0	$f_{I/O}/256$
1	1	1	$f_{I/O}/1024$

TOP value is determined by OCR0, and the behavior of OC0 is determined by bits COM01 and COM00 (see Table 5.8).

Table 5.4: Description of Waveform Generation Mode Bits

Mode	WGM01	WGM00	TOP	Update of OCR0 at	TOV0 Flag Set on
Normal	0	0	0xFF	Immediate	MAX
Phase Correct PWM	0	1	0xFF	TOP	BOTTOM
CTC	1	0	OCR0	Immediate	MAX
Fast PWM	1	1	0xFF	TOP	MAX

Bits COM01 and COM00 define the behavior of the Output Compare pin (OC0), which depends on whether the timer/counter is in non-PWM mode or fast PWM mode. Tables 5.5 and 5.6 show the behavior of OC0 in non-PWM mode (i.e., normal or CTC mode) and fast PWM mode, respectively.

Table 5.5: Description of Compare Output Mode Bits (non-PWM mode)

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected
0	1	Toggle OC0 on compare match
1	0	Clear OC0 on compare match
1	1	Set OC0 on compare match

5.4.2 TCNT1

TCNT1 is much more versatile and complex than TCNT0. In contrast to TCNT0, which is started, stopped, reset etc., TCNT1 is usually left running.

Table 5.6: Description of Compare Output Mode Bits, Fast PWM mode

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected
0	1	Reserved
1	0	Clear OC0 on compare match, set OC0 at TOP
1	1	Set OC0 on compare match, clear OC0 at TOP

This allows a variety of functions to be implemented. The block diagram of the Timer/Counter1 is shown in Figure 5.21.

The Timer/Counter1 consists of 16-bit TCNT1 register and three 16-bit Output Compare Registers (OCR1A, OCR1B, and OCR1C). The value of TCNT1 is continuously compared with OCR1A-C. A match will set the Output Compare Flag (OCF1A-C) at the next clock cycle. These flag generate an output compare interrupt if OCIE1A-C is enabled. These OCF1A-C signals are used by the Waveform Generator to generate an output based on the operating mode set by WGM13-WGM10 and COM1A bits using a pair of Timer/Counter1 Control Register A and B (TCCR1A and TCCR1B).

Similar to TCNT0, a *value* can be loaded onto TCNT1 in *normal mode*, which counts up to 63,535 (i.e., 0xFFFF) and when it rolls over (i.e., transition from 63,535 to 0) it interrupts the CPU by setting *Timer/Counter Overflow 1* flag (TOV1) in the *Timer/Counter Interrupt Flag Register* (TIFR). TCNT1 can also be used together with *Output Compare Register 1A* (OCR1A), *1B* (OCR1B), and *1C* (OCR1C) to generate an interrupt by setting the *Output Compare Flag 1A* (OCF1A) and in TIFR when TCNT0 and OCR0 match.{*(BL): this needs work*}

In addition to the usual Timer/Counter feature found in TCNT0, TCNT1 has a 16-bit *Input Capture Register 1* (ICR1). ICR1 is used to measure pulse widths of an incoming signal or simply capturing times.

The Input Capture Unit is used to capture external events by associating timestamps. The external signal indicating an event or multiple events can applied to the *Input Capture Pin 1*(ICP1), which is the pin 4 on PORT B and shares the pin with Output Compare pin 0 (OC0), or through the Analog Comparator Unit. The captured timestamps can be used to calculate frequency, duty cycle, and other characteristics of the applied signal.

When a event (logic change) occurs on the ICP1, a capture will be triggered. This causes the 16-bit content of TCNT1 to be written to the Input Capture Register 1 (ICR1) and the Input Capture Flag 1 (ICF1) is set. If input capture interrupt is enabled (TICIE1 = 1), an interrupt is generated.

TCNT1 also provides an *input noise canceller* feature, which prevents

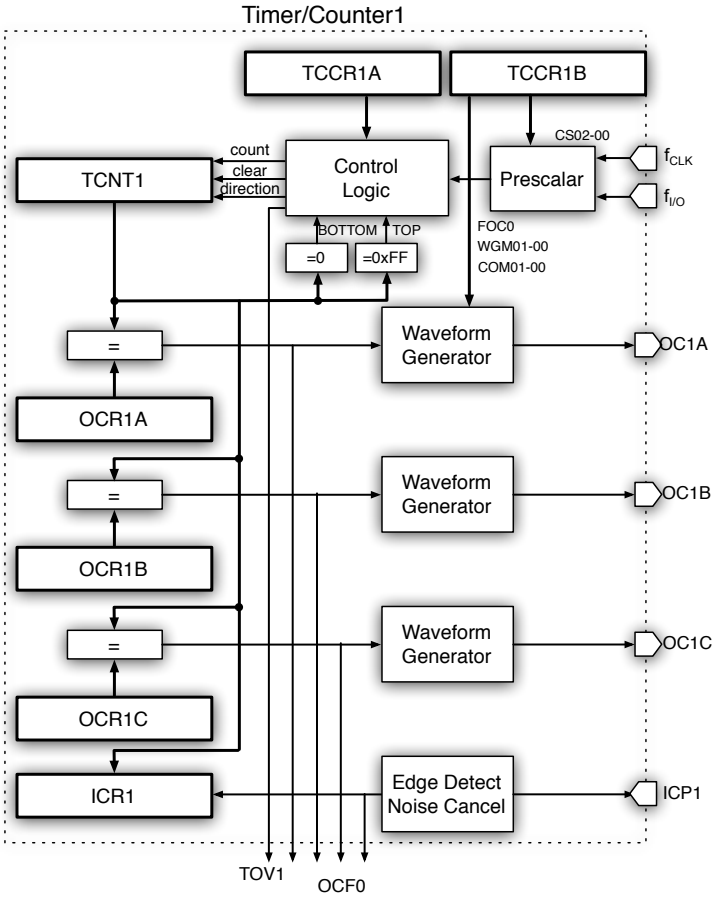


Figure 5.21: Block diagram of Timer/Clock2.

miscellaneous unwanted spikes in the signal applied to ICP. When this feature is active, ICP must remain at the active level for four successive samples before the microcontroller will consider the trigger as legitimate and capture the data. The noise cancelling feature is activated using the input capture noise canceller bit (ICNC1) in TCCR1B.

Reading in the 16-bit ICR0 is done by first reading the low byte of ICR0 (ICR0L) and then the high byte (ICR0H).

The behavior of TCNT1 controlled by Timer Counter Control Register 1 (TCCR1), which consists of a pair of registers TCCR1A and TCCR1B. These two registers are shown in Figure 5.22.

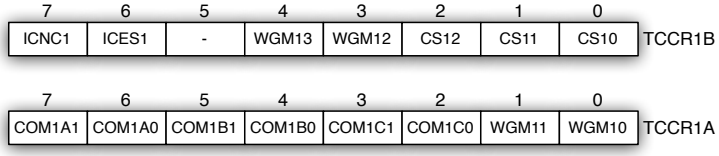


Figure 5.22: Time Counter Control Register 1.

TCCR1A has similar features as TCCR0 with additional capabilities. Bits COM1A1 and 0, COM1B1 and 0, and COM1B1 and 0 control the output compare pins OC1A, OC1B, and OC1C, respectively. Table 5.8 shows the meaning of these bits for non-PWM, Fast PWM, and Phase Correct and Phase and Frequency Correct PWM modes.

Table 5.7: Description of Wave Generation Mode Bits

Mode	WGM13-10	Mode of Operation	TOP	Update of OCR1A-C	TOV1 Flag set on
0	0000	Normal	0xFFFF	Immediate	MAX
1	0001	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0010	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0011	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0100	CTC	OCR1A	Immediate	MAX
5	0101	Fast PWM, 8-bit	0x00FF	TOP	TOP
6	0110	Fast PWM, 9-bit	0x01FF	TOP	TOP
7	0111	Fast PWM, 10-bit	0x03FF	TOP	TOP
8	1000	PWM, Phase & Freq. Correct	ICR1	BOTTOM	BOTTOM
9	1001	PWM, Phase & Freq. Correct	OCR1A	BOTTOM	BOTTOM
10	1010	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1011	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1100	CTC	ICR1	Immediate	TOP
13	1101	Reserved	-	-	MAX
14	1110	Fast PWM	ICRn	TOP	TOP
15	1111	Fast PWM	OCR1A	TOP	TOP

Bits CS12, CS11, and CS10 correspond to bits CS02, CS01, and CS00, respectively, in TCCR0. ICR1 is a 16-bit register that captures the content of TCNT1 when the microcontroller receives either a rising or falling edge signal from the *input capture pin* (ICP). The choice between rising and falling edge is controlled by input capture edge select bit (ICES1) in TCCR1B.

Table 5.8: Description of Compare Output Mode Bits, Fast PWM mode

COM1A1:0 COM1B1:0 COM1C1:0	Non-PWM	Fast PWM	Phase Correct and Phase & Frequency Correct PWM
00	Normal port operation, OC1A-C disconnected	Normal port operation, OC1A-C disconnected	Normal port operation, OC1A-C disconnected
01	Toggle OC1A-C on compare match	Toggle OC1A-C on compare match, OC1A-C disconnected	Toggle OC1A-C on compare match, OC1A-C disconnected
10	Clear OC1A-C on compare match on up counting. Set OC1A-C to low level	Clear OC1A-C on compare match, set OC1A-C at TOP	Clear OC1A-C on compare match on up counting. Set OC1A-C on compare match on down counting
11	Set OC1A-C on compare match, set OC1A/B/C to high level	Set OC1A-C on compare match, clear OC1A-C at TOP	Set OC1A-C on compare match when up counting. Clear OC1A-C on compare match when down-counting

5.4.3 TCNT Interrupt Mask and Interrupt Flag Registers

Timer/Counter Overflow flags (TOV0 and TOV1) and Output Compare Flags (OCF0, OCF1A, OCF1B, and OCF1C) are enabled using the Timer/-Counter Interrupt Mask Register (TIMSK). Figure 5.23 shows how TCNT0 and TCNT1 are connected to Timer/Counter Interrupt Flag (TIFR) via TIMSK.

5.5 Analog-to-Digital Converter

{*(BL): Work in progress...*} The features of the Analog-toDigital Converter are as follows:

- 8 or 10-bit Resolution
  - 0.5 LSB Integral Non-linearity
  - 2 LSB Absolute Accuracy
- 13 -260 μs Conversion Time
- 8 Multiplexed Single-Ended Input Channels
- 7 Differential Input Channels
- Optional Left Adjustment for ADC Result Readout

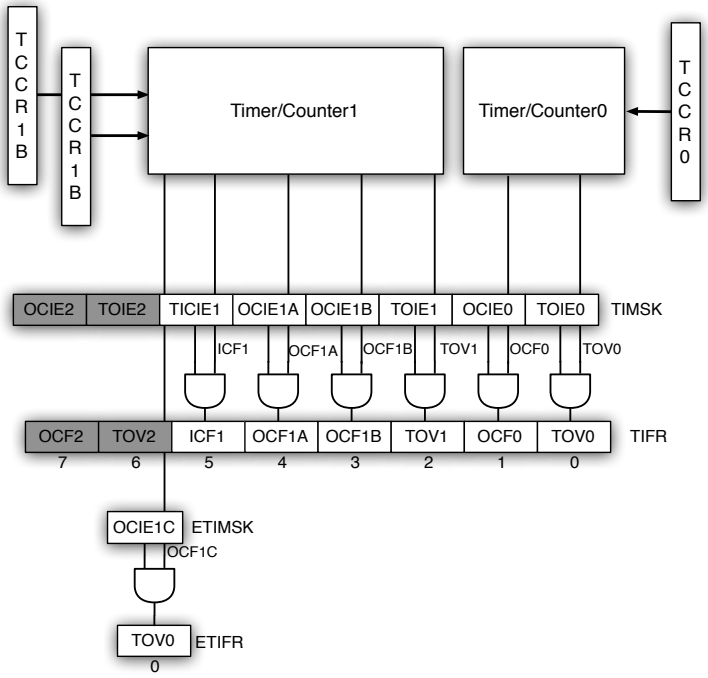


Figure 5.23: TIMSK and TIFR registers.

- 0-V_{CC} ADC Input Voltage Range
- Selectable 2.56 V ADC Reference Voltage
- Free Running or Single Conversion Mode
- Interrupt on ADC Conversion Complete

5.6 Universal Synchronous Asynchronous Receiver/-Transmitter

{(BL): Work in progress...}

Chapter 6

Embedded C

Most C compilers provide a convenient method of interacting with ports using `sfrb` and `sfrw` compiler commands to assign labels to each of the parallel ports and other I/O devices.

6.1 Basics of Embedded C

6.1.1 Comments

The character string "/" marks the beginning of a comment. The end of the comment is marked with "/".

Example:

```
/*  This is a comment  */
/*  This is a
            multiple line comment  */
```

One-line comments may be also defined by using the string "//" .
Example:

```
//  This is also a comment
```

Nested comments are not allowed.

6.1.2 Identifiers

An *identifier* is the name you give to a variable, function, label or other object. An identifier can contain letters (A....Z, a....z) and digits (0....9), as well as the underscore character(_). However an identifier can only start with a letter or an underscore. Identifiers are case sensitive, e.g., `variable1`

is not the same as `Variable1`. Identifiers can have up to 32 characters. An identifier cannot be a reserve word (see Section 6.1.3.

6.1.3 Reserve Words

Following is a list of keywords reserved by the compiler. These can not be used as identifier names.

```
break       bit         case        char        const
continue    default     defined     do          double
eeprom      else        enum        extern      flash
float       for         funcused    goto        if
inline      int         interrupt   long        register
return      short       signed      sizeof      sfrb
sfrw        static      struct      switch      typedef
union       unsigned    void        volatile    while
```

6.1.4 Variables

Variables and constants represent the stored data in embedded system. A variable is declared using a reserve word indicating its size and type followed by an identifier.

The syntax is:

$[< storage\ modifier >] < typedefinition > < identifier >;$

Variable Scope

Program variables can be *global* (accessible to all the functions in the program) or *local* (accessible only inside the function they are declared). If the global variables are not specifically initialized, they are automatically set to 0 at program start up. The local variables are not automatically initialized on function calls.

Example:

```
/* Global variables declaration */
char  a;
int b;
/* and initialization */
long c=1111111;

void main (void) {
      /* Local variables declaration */
      char d;
```

```
      int e;
      /* and initialization */
      long f=22222222;

}
```

Local variables that must conserve their values during different calls to a function must be declared as `static`.

Example:

```
int alfa (void) {
      /* declare and initialize the static variable */
      static int n=1;
      return n++;
}

void main (void) {
      int i;

      /* the function will return the value 1 */
      i=alfa();

      /* the function will return the value 2 */
      i=alfa();
}
```

Variables that are declared in other files must be preceded by the `extern` keyword.
Example:

```
extern int xyz;
/* now include the file which contains the variable xyz definition */
#include <file_xyz.h>
```

In order to instruct the complier to allocate a variable to a register, the `register` modifier must be used.
Example:

```
register int abc;
```

The complier may automatically allocate a variable to registers, even if this modifier is not used.

The `volatile` modifier must be used in order to prevent a variable form being allocated to a register and to warn the complier that it may be subject to outside change during evaluation.
Example:

```
volatile int abc;
```

All the global variables, not allocated to registers, are stored in the Global Variables area of SRAM. All the local variables, not allocated to registers, are stored in dynamically allocated space in the Data Stack area of SRAM.

Global variables can be stored at specific SRAM locations at design-time using the @ operator.

Example:

```
/* the integer variable a is stored in SRAM at address 80h */
int a @0x80;

/* the structure alfa is stored in SRAM at address 90h */
struct x {
       int a;
       char c;
} alfa @0x90;
```

The global variables located in the GPIOR register(s) and R2 to R14 memory space. These variables are declared using the `bit` keyword.

The syntax is:

$bit < identifier >;$

Example:

```
/* declaration and initialization for an ATtiny2313 chip which has GPIOR0, GPIOR1 and GP
bit alfa=1;       /* bit0 of GPIOR0 */
bit beta;         /* bit1 of GPIOR0 */

void main(void)
{
      if (alfa) beta =! Beta;

      /*  */
}
```

The allocation is realized in order of variable declaration until the R14 register is allocated. If the automatic register allocation is disabled, you can use the register keyword to specify which global variable to be allocated to registers.

Example:

```
/* disable automatic register allocation */
#pragma regalloc-
```

Table 6.1: Data Types.

Type	Size (Bits)	Range
`bit`	1	0 , 1
`char`	8	-128 to 127
`unsigned char`	8	0 to 255
`signed char`	8	-128 to 127
`int`	16	-32768 to 32767
`short int`	16	-32768 to 32767
`unsigned int`	16	0 to 65535
`signed int`	16	-32768 to 32767
`long int`	32	-2147483648 to 2147483647
`unsigned long int`	32	0 to 4294967295
`signed long int`	32	-2147483648 to 2147483647
`float`	32	±1.175e-38 to ±3.402e38
`double`	32	±1.175e-38 to ±3.402e38

```
/* allocated the variable alfa to a register */
register int alfa;
/* allocate the variable beta to the register pair R10, R11 */
register int beta @10;
```

The char, int and pointer local variables are automatically allocated, in order of declaration, to register R16 up to R21.

6.1.5 Data Types

The following table lists all the data types supported by the CodeVisionAVR C compiler, their range of possible values and their size :

The bit data type is supported only for global variables. If the Project — Configure — C Compiler — Code Generation — char is unsigned option is checked or ♯pragma uchar+ is used, then char has by default the range 0..255.

6.1.6 Constants

Constants are fixed values that cannot be modified during execution.

Numeric constant can be declared in may different ways:

- Integer or long integer constants may be written in decimal form (e.g. 1234), in binary form with the 0b prefix (e.g. 0b101001), in hexadecimal form with 0x prefix (e.g. 0xff) or in octal form with 0-prefix(e.g.0777).

- Unsigned integer constants may have the suffix U (e.g.10000U).
- Long integer constants may have the suffix L (e.g. 99L).
- unsigned long integer constants may have the suffix UL (e.g. 99UL)
- Floating point constants may have the suffix F(e.g.1.234F).
- Character constants must be enclosed in single quotation marks.(e.g. a).
- String constants must be enclosed in double quotation marks.(e.g. Hello world).
- If you place a string between quotes as a function parameter, this string will automatically be considered as constant and will be placed in FLASH memory .

Constant can be grouped on arrays, which can have up to 8 dimensions. Constant are stored in FLASH memory, to specify this you must use the flash or const keywords. Constant expressions are automatically evaluated during compilation.
Example;

```
flash int   integer_constant = 1234+5;
flash char  char_constant = a;
flash long  long_int_constant1 = 99L;
flash long  long_int_constant2 = 0x10000000;
flash int   integer_array1[ ] = {1, 2, 3};
/*  The first two elements will be 1 and 2, the rest will be 0  */
flash int   integer_array2[10] = {1, 2};
flash int   multidim_array[2][3] = {{1, 2, 3}, {4, 5, 6}};
flash char  string_constant1[ ] =  This is a string constant  ;
const char  string_constatn2[ ] =  This is also a string constant  ;
```

6.1.7 Preprocessors

The Preprocessor directives allows a programmer to (1) include text from other files, such as header files containing library and user function prototypes, to be included in a program; (2) Define macros that reduce programming effort and improve the legibility of the source code; (3) Set up conditional compilation for debugging purposes and to improve program portability; and (4)Issue compiler specific directives

Two parameters can be concatenated using the `##` operator.

Example:

```
#define ALFA(a,b)  a ## b
      /* the following code sequence will be replaced with char xy=1;  */
      char ALFA(x,y)=1;
```

A macro definition can be extended to a new line y using \.

Example:

```
#define MESSAGE This is a very \
      long text
```

A macro can be undefined using the `#undef` directive.

Example:

```
#undef ALFA
```

The `#ifdef`, `#ifndef`, `#else` and `#endif` directives can be used for conditional compilation. The syntax is given by

```
#ifdef macro_name
[set of statements 1]
#else
[set of statements 2]
#endif
```

The `#if`, `#ellif`, `#else` and `#endif` directives can also for used for conditional compilation. The syntax is given by

```
#if expression1
[set of statements 1]
#elif expression2
[set of statements 2]
#else
[set of statements 3]
#end if
```

If `expression1` evaluates to true, the `set of statements 1` will be compiled. If `expression2` evaluates to true, the `set of statements 2` will be compiled. Otherwise the `set of statements 3` will be compiled. The `#else` and `set of statements 3` are optional.

Macros

The following lists predefined macros :

- `__CODEVISIONAVR__` - the version and revision of the compiler represented as an integer, example for V1.24.0 this will be 1240

- `__LINE__` - the current line number of the compiled file
- `__FILE__` - the current compiled file
- `__TIME__` - the current time in hh:mm:ss format
- `__DATE__` - the current date in mmm dd yyyy format
- `__BUILD__` - the build number
- `_CHIP_ATXXXXX_` - where ATXXXXX is the chip type, in uppercase letters, specified in the Project Configure C Compiler Code Generation Chip option
- `_MCU_CLOCK_FREQUENCY_` - the AVR clock frequency specified in the Project Configure C Compiler Code Generation Clock option, expressed as an integer in Hz
- `_MODEL_TINY_` - if the program is compiled using the TINY memory model
- `_MODEL_SMALL_` - if the program is compiled using the SMALL memory model
- `_OPTIMIZE_SIZE_` - if the program is compiled with optimization for size
- `_OPTIMIZE_SPEED_` - if the program is compiled with optimization for speed
- `_ENHANCED_CORE_` - if the program is compiled using the enhanced core instructions available in the new Atmega chips
- `_HEAP_START_` - the heap starting address
- `_HEAP_SIZE_` - the heap size specified in the Project—Configure—C Compiler—Code Generation—Heap size option
- `_UNSIGNED_CHAR_` - if the Project Configure C Compiler Code Generation char is unsigned compiler option is enabled or ♯pragma uchar+ is used
- `_8BIT_ENUMS_` - if the Project Configure C Compiler Code Generation 8 bit enums compiler option is enabled or ♯pragma 8bit_enums+ is used.

The `#line` directive can be used to modify the predefined `__LINE__` and `__FILE__` macros. The syntax is given by

```
#line integer_constant ``file_name'']
```

Example:

```
/* This will set __LINE__ to 50 and __FILE__ to file2.c  */
#line 50 file2.c

/* This will set __LINE__ to 100  */
#line 100
```

The `#error` directive can be used to stop compilation and display an error message.
Syntax:

```
#error error_message
```

Example:

```
#error this is an error!
```

The `#warning` directive can be used to display a warning message.

Syntax:

```
#warning warning_message
```

Example:

```
#warning this is a warning!
```

Pragmas

The `#pragma` directive allows compiler specific directives, and is often used to surpress speciic error messages, manage heap and stack debugging, etc.,
Example:

```
/*  warnings are disabled  */
#pragma warn-
/*  write some code here  */

/*  warnings are enabled  */
#pragma warn+
```

The compilers code optimizer can be turned on or off using the `#pragma opt` directive. This directive must be placed at the start of the source file. The default is optimization turned on. Example /* Turn optimization off, for testing purposes */ ♯pragma opt- or /* Turn optimization on */ ♯pragma opt+

If the code optimization is enabled, you can optimize some portions or all the program for size or speed using the `#pragma optsize` directive. The default state is determined by the Project Configure C Compiler —Code Generation —Optimization menu setting. Example :

```
/*  The program will be optimized for minimum size  */
#pragma optsize+
/*  Place your program functions here  */

/* Now the program will be optimized for maximum execution speed*/
#pragma optsize-

/*  Place your program functions here  */
```

The automatic saving and restoring of registers affected by the interrupt handler, can be turned on or off using the `#pragma savereg` directive. Example :

```
/*  Turn registers saving off       */
#pragma savereg-

/*  interrupt handler         */
interrupt [1] void my_irq(void) {
/*  now save only the tegisters that are affected by the routines in the interrupt handl
#asm
          push r30
          push r31
          in   r30,SREG
          push r31
#endasm
/*  place the C code here           */
/*  ....                */
/*  now restore SREG, R31 and R30   */
#asm
          pop r30
          out  SREG,r30
          pop r31
          pop r30
#end asm
}
```

```
/*  re-enable register saving for the other interrupts  */
#pragma savereg+
```

The default state is automatic saving of registers during interrupts. The `#pragma savereg` directive is maintained only for compatibility with versions of the compiler prior to V1.24.1. This directive is not recommended for new projects.

The automatic allocation of global variables to registers can be turned on or off using the `#pragma regalloc` directive. The default state is determined by the Project — Configure — Code Generation — Automatic Register Allocation check box.

Example:

```
/*  the following global variable will be automatically allocated to a register  */
#pragma regalloc+
unsigned char alfa;

/*  the following global variable will not be automatically
allocated to a register and will be placed in normal SRAM  */
#pragma regalloc-
unsigned char beta;
```

The ANSI char to int operands promotion can be turned on or off using the `#pragma promotechar` directive.

Example:

```
/*  turn on the ANSI char to int promotion  */
#pragma promotechar+

/*  turn off the ANSI char to int promotion  */
#pragma promotechar-
```

Treating char by default as an unsigned 8 bit can be turned on or off using the `#pragma uchar` directive.

Example:

```
/*  char will be unsigned by default  */
#pragma uchar+

/*  char will be signed by default  */
#pragma uchar-
```

This option can also be specified in the Project — Configure — C Compiler — Code Generation — char is unsigned menu.

The `#pragma library` directive is used for specifying the necessity to compile/link a specific library file.

Example:

```
#pragma library mylib.lib
```

The `#pragma glbdef+` directive is used for compatibility with projects, created with versions of CodeVisionAVR prior to V1.0.2.2, where the Project ConfigureC Compiler Global ♯define option was enabled.

It signals the compiler that macros are globally visible in all the program modules of a project. This directive must be placed in beginning of the first source file of the project. By default this directive is not active, so macros are visible only in the program module where they are defined.

6.1.8 Enumeration

Enumerations can be stored in SRAM or EEPROM. To specify the storage in EEPROM, the eeprom keyword must be used.

Example:

```
eeprom enum days { sunday, monday, tuesday, wednesday,
                        thursday, friday, saturday} days_of_week;
```

It is recommended to treat enumerations as having 8 bit char data type, by checking the 8 bit enums check box in Project—Configure—ComplierCode Generation. This will improve the size and execution speed of the complied program.

6.1.9 Pointers

Pointers are variables that contain the address of a variable, constant, function, or data object. Pointer data type allocates enough memory to hold the address. Usually a 16 or 24 bit address (depending on the amount of memory)

Declared with the indirection operator `*`

Assigned with the address operator `&`

Example:

addresses	Data
0x3248	0x3F
0x3249	0x5A
0x324A	0xF1
0x324B	0x32
0x324C	0x9B
0x324D	0x38
0x324E	0x4E

Variable `c` may be assigned to address `0x324A`, but the value of `c` is `0xF1`

27cAddress of c

```
char *p;          //p is a pointer to a character
char c;           //c is a character (unsigned 8-bit integer)
c = 0xF1;         //c is assigned the value (hex) 0xF1
p = &c;           //p is assigned the address of c (p is pointing to c)
```

c is an 8-bit char, but p is a 16-bit address (a pointer) to a char

Example: What is the value of a and b when this segment of code is finished?

```
char *p;          //p is a pointer to a character
char a, b;        //a and b are characters (unsigned 8-bit integer)
a =10;
b =45;
p = &a;           //p is assigned the address of a (p is pointing to a)
*p = b;           //the memory location pointed to by p is assigned the value of b
```

Pointers

Example: What is the address of Port_Band what data is stored there?

```
unsigned char *Port_B;
Port_B= 0x1010;
*Port_B= 0x38;
```

Pointers Example: 31What is the address of Port_Band what data is stored there?

```
Port\_BAddress = 0x1010
Port\_BData = 0x38
unsigned char *Port\_B;
Port\_B= 0x1010;
*Port\_B= 0x38;
```

Variables placed in SRAM are accessed using normal pointers. For accessing constants placed in FLASH memory, the flash type storage modifier is used. For accessing variables placed in EEPROM, the eeprom type storage modifier is used. Although the pointers may point to different memory areas, they are by default stored in SRAM.

Example:

```
/* Pointer to a char string placed in SRAM */
char *ptr_to_ram=This string is placed in SRAM;
/* Pointer to a char string placed in FLASH */
flash char *ptr_to_flash1=This string is placed inFLASH;
char flash *ptr_to_flash2 =This string is also placed in FLASH;
```

```
/* Pointer to a char string placed in EEPROM */
eeprom char *ptr_to eeprom1=This string is placed in EEPROM;
char eeprom *ptr_to_eeprom2=This string is also placed in EEPROM;
```

In order to store the pointer itself in other memory areas, like FLASH or EEPROM, the flash or eeprom pointer storage modifiers must be used as in the examples below:

```
        /* Pointer stored in FLASH to a char string placed in SRAM */
char *flash flash_ptr_to_ram = This string is placed in SRAM;

/* Pointer stored in FLASH to a char string placed in FLASH */
flash char *flash flash_ptr_to_flash =This string is placed in FLASH;

/* Pointer stored in FLASH to a char string placed in EEPROM */
eeprom char *flash eeprom_ptr_to eeprom=This string is placed in EEPROM;

/* Pointer stored in EEPROM to a char string placed in SRAM */
char *eeprom eeprom_ptr_to_ram= This string is placed in SRAM;

/* Pointer stored in EEPROM to a char string placed in FLASH */
flash char *eeprom eeprom_ptr_to_flash =This string is placed in FlASH;

/* Pointer stored in EEPROM to a char string placed in EEPROM */
eeprom char *eeprom eeprom_ptr_to_eeprom =This string is placed in EEPROM;
```

In order to improve the code efficiency several memory models are implemented.

The TINY memory model uses 8bits for storing pointers to the variables placed in SRAM. In this memory model you can only have access to the first 256 bytes of SRAM

The SMALL memory model uses 16 bits for storing pointers the variables placed in SRAM. In this memory model you can have access to 65536 bytes of SRAM

In both TINY and SMALL memory models pointers to the FLASH memory area use 16 bits. Because in these memory models pointers to the FLASH memory are 16 bits wide, the total size of the constant arrays and literal char strings is limited to 64K. However the total size of the program can be the full amount of FLASH.

In order to remove the above mentioned limitation, there are available two additional memory models: MEDIUM and LARGE.

The MEDIUM memory model is similar to the SMALL memory model, except it uses pointers to constants in FLASH that are 32bits wide. The

pointers to functions are however 16 bit wide because they hold the word address of the function, so 16 bits are enough to address a function located in all 128kbytes of FLASH.

The MEDIUM memory model can be used only for chips with 128kbytes of FLASH.

The LARGE memory is similar to the SMALL memory model, except it uses pointers to the FLASH memory area that are 32 bits wide. The LARGE memory model can be used for chips with 256kbytes or more of FLASH.

In all memory models pointers to the EEPROM memory area are 16 bit wide.

Pointers can be grouped in arrays, which can have up to 8 dimensions.

6.2 Operators

C provides the following set of operators:

- I/O operations
- Arithmetic
- Bitwise
- Logical
- Relational
- Pre-and post-increment and decrement

I/O Operations

Embedded microcontrollers must interact directly with outside hardware through parallel ports on the controller. Microcontroller header files use `sfrb` and `sfrw` to assign labels to parallel ports.

Special Function Register (Byte or Word)

Defines labels such as DDRB, PINA, PORTB, etc

I/O Operations

To define inputs/outputs write to DDRx

Data direction register (could be A, B, C, etc)

```
DDRx= 0xff;             //make all lines outputs
DDRx= 0;          // make all lines inputs
```

To write to an output port

```
PORTC = value;
```

To read an input port

```
Variable_name= PINA;
```

```
#include <MEGA8535.h>         //register definition file
unsigned char z;              //z is an 8-bit number
void main (void) {
      DDRA = 0;               //make Port A all inputs
      DDRB = 0xff;            //make Port B all outputs
      while (1) {
            z = PINA;         //read Port A
            PORTB = z + 1;    //write the value (+1) to Port B
      }
}
```

Arithmetic Operators
The arithmetic operators in C are:

- Multiply: `*`
- Divide: `/`
- Addition: `+`
- Subtraction: `-`
- Negate: `-`
- Modulo: `%`

The bitwise operators in C are:

- Ones compliment:
- Left shift: `<<`
- Right shift: `>>`
- AND: `&`
- OR: `|`

- Exclusive OR
- The logical operators in C are:
- AND&&
- OR—: `|`

The results of these operations are either TRUE (equal to any non-zero result) or FALSE (equal to 0)

The relational operators in C are:

- Is equal to: `==`
- Is not equal to: `!=`
- Less than: `<`
- Less than or equal to¡=
- Greater than: `>`
- Greater than or equal to: `>=`

The increment/decrement operators in C are:

- `++x` pre-increment
- `x++` post-increment
- `--x` pre-decrement
- `x--`post-decrement

A short hand version of other C statements:
A += 3; same as A = A + 3;
B -= 2; same as B = B-2;
C —= 3;same as C = C — 3;

6.3 Control Statements

6.3.1 `if` Statements

The if statement has the following format:

if (expression) statement;

If the expression evaluates to TRUE, then the statement is executed. If the expression evaluates to FALSE then the statement is skipped and execution continues at the next statement.

6.3.2 `if/else` Statements

The `if/else` statement is nearly the same as the `if` statement and has the following format:

if (expression) true_statement;
else else_statement;

If the expression evaluates to TRUE, then the *true_statement* is executed. If the expression evaluates to FALSE then the *else_statement* is executed.

6.3.3 `if/else/else if` Statements

You may also need to have an if/else/else if structure

if (expression1) true_statement1;
else if (expression2) true_statement2;
else if (expression3) true_statement3;
else else_statement;

If *expression1* evaluates to TRUE, then the *true_statement1* is executed. If *expression1* evaluates to FALSE then the next expression (*expression2*) is evaluated. If *expression2* evaluates to TRUE, then the *true_statement2* is executed. If none of the expressions evaluate to TRUE then the *else_statement* is executed.

6.3.4 `while` Loop

The `while` loop has the following format:

while (expression) statement;

When program execution enters the while loop, the *expression* is evaluated. If the expression evaluates to TRUE the *statement* is executed. When all statements are executed, the program will be to this process again (evaluate the,expression, etc). If the expression evaluates to FALSE the *statement* is skipped and execution continues at the *statement* immediately following the while loop structure.

6.3.5 `do/while` Loop

The do/while loop has the following format:

do statement;
while (expression);

6.3.6 `do/while` Loop

The `do/while` loop has the following format:

do statement;
while (expression);

The `do/while` loop is very similar to the `while` loop except that the *expression* is evaluated at the end of the loop. When program execution enters the `do/while` loop, the *statement* is executed. If the *expression* evaluates to TRUE the program continues at the beginning of the `do/while` loop. This process continues as long as the *expression* evaluate to TRUE. The important difference between `while` and `do/while` is that the `do/while` statement is ALWAYS executed at least once.

6.3.7 `for` Loop

The `for` loop has the following format:

for (exp1; exp2; exp3) statement;

A `for` loop is typically used to execute a statement or statement block a specific number of times. *exp1* initializes the control variable. *exp2* is a conditional control statement that is used to decide if the loop should be

continued. *exp3* is an assignment statement that updates the value of the control variable. Execution continues as long as *exp2* evaluates to TRUE.

6.3.8 Conditional Statement

The conditional statement is nearly the same as the *if/else* statement, and has the following format:

expression? true_statement:false_statement;

If the *expression* evaluates to TRUE, then the *true_statement* is executed. If the *expression* evaluates to FALSE then the *false_statement* is executed.

6.3.9 `switch/case` statement

The `switch/case` statement is nearly the same as the `if/else/else-if` structure, and has the following format:

switch(expression)
case constant1:
statement(s)1; break;
case constant2:
statement(s)2; break;
case constant3:
statement(s)3; break;
default:
statement(s)4;

If the expression evaluates to one of the constants in the case statements, those statement(s) is(are) executed. The `break` statement causes execution to go to the end of the switch structure. If expression does not equal any of the constants, execution continues at the "default" label.

6.4 Functions

"A function is an encapsulation of a block of statements that can be used more than once in a program." Some languages call functions procedures, some call them subroutines. Functions can be used to break up the operational parts of a program into its fundamental parts, which then allows the programmer to debug each element of a program.

6.4.1 Library Functions

Functions may be developed so that they perform a certain function with no input parameters or accept input(s) as parameters to the function.

Example function calls

```
sleep( );         // no input parameters or returned values
printf("help me");// input parameter is: "help me..."
c = getchar( );   // getchar( ) will return a value & assign it to c
```

6.4.2 Function Structure

```
type function_name (type parameter 1, type parameter 2, ){
      statement 1;
      statement 2;
      ...
      statement X;}
```

This is the data type of the return variable. So, if the function was to return an int, this would be int.

```
type function_name(type parameter1, type parameter2, ...)
{
statement1;
statement2;
...
statementX;
}function_name.
```

Remember that C is case sensitive and any alpha-numeric key or the underscore can be used

`type` preceding the function name indicates the data type of the returned variable. `parameter1` and `parameter2` are the names of the variables in the function. `type` preceding each input parameter indicates it data type. For example, a function that adds two integers would have both of these parameters as type `int`.

The following example function `add_int(int A, int B)` would return the result of A+B:

```
int add_int(int A, int B){
      int x;
      x = A + B;
      return x;
}
```

6.4.3 Function Ordering

Functions, just like variables, must be declared before it is called. There are one of two ways to do this. The first method is to place the declarations in the order they will be called, which is often this is not possible. The second method is to *prototype* the function.

Example:

```
int var1, var2, var3;   //global variables
int function1(int x, int y) { statements..}
void function2 (void) { statements}
void main (void) {
var3 = function1(var1, var2);
function2 ( );
}
```

This is fine because `function1( )` and `function2( )` are declared in the order they are used and declared before they are used in main.This is not always possible.

Function Prototype

Example:

```
int var1, var2, var3; //global variables
void main (void) {
var3 = function1(var1, var2);
function2( );
}
int function1 (int x, int y) {
      ...statements...
}
void function2 (void) {
      ...statements...
}
```

In this case, errors would be generated because the function calls in main ( ) have not been declared the compiler doesnt have enough information about them yet. So, we must prototype the functions. Function Prototype

In the following example, the first two statements are prototype statements, which tells the compiler everything it needs to know to compile the program even though `function1( )` and `function2( )` are declared "out of order"

```
int function1(int, int);
void function2(void);
int var1, var2, var3;   //global variables
```

```
void main (void) {
      var3 = function1(var1, var2);
      function2( );
}

int function1(int x, int y) {
      ...statements...
}

void function2 (void) {
      ...statements...
}
```

6.4.4 #include

The `#include <filename.h>` compiler command tells the compiler that it can find the prototypes of functions used in this program in the file called filename.h. This is why we need to include Mega128.h (or whatever microcontroller we are using)

6.4.5 Recursive Functions

Recursive functions are functions that call themselves. Each time it is called, its variables are put on the stack with the address of the calling function. If the function returns a value, which is common, it will also place these values on the stack

The following code illustrates a recursive function call.

```
int factorial(int x){
      if (x==0)
            return 1;
      else
            return (x*factorial(x-1));
}
```

The integers (passed parameters), the returned value, and the return address are each usually 2 bytes. Therefore, each call of a recursive function will typically save 6 bytes on the stack. Running out of stack space is a common run-time error.

In the example below, 4 is passed to factorial the first time the function is called. Since `x` $\neq$ `0`, the `else` clause is executed and calls itself with 3 as

its input. This process repeats until the input value reaches 0, which resturs 1. Thus, the recursion performs 4 * 3 * 2 * 1.

```
int factorial (int x)
{
      if (x == 0)
            return 1;
      else
            return (x*factorial(x-1));
}

void main ( ) {
      int number = 4;
      number = factorial(number);
}
```

6.5 Arrays and Structures

6.5.1 Arrays

An *array* consists of a data set of a particular data type. All elements of an array must be the same type. Arrays are declared like any other variable except that the number of required array elements is included. The following example declares two arrays.

```
int digits[10];
char name[20];
```

Array initialization is performed as it is declared. This is done by placing the values within braces separate by commas. The following example illustrates array initialization.

```
int digits[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
char name[10] = {B, e, a, u b, a, d, i, n, e };
```

Array indexing references a specific element of an array through its index (or subscript). For example, the first element of array `digits` is *digits[0]*. The ninth element of the array `name` is `n`.

6.5.2 Multidimensional Arrays

C supports multidimensional arrays. Adjacent memory locations are referenced by the right-most index. For example, in the array declaration below

`matrix[1][2]` is 6. Moreover, the values adjacent to 6 are 5 and 7. This is referred to as row-major ordering. (as opposed to the 2 and the 10).

```
int matrix [3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

```
int matrix [3][4] = {0, 1, 2, 3,
                     4, 5, 6, 7,
                     8, 9, 10, 11};
```

6.5.3 Structures

A *structure* is a group of related variables under one name and is used to create a single data object from one or more variables. Variables within the structure are called *members*. Structures are the fundamental element in OOP. Structures can be thought of as an object. Members are the properties of the object. Members of a structure do not need to be of the same data type.

Structure declaration format

struct structure_tag_nametype variable1;type variable2;:type variableX;;

Example:

```
struct structure_name {
      type variable1;
      type variable2;
      ...
      type variableX;
};
```

```
struct employee_info {
      char name[20];
      char id[9];
      char month;
      char day;
      int year;
} employee[20];
```

An array of 20 `employee_info` data types referred to as employee. In the structure is their name (20 characters), their employee id number (9 characters), their birth dates (month, day, year).

To access one of the members of the structure, the member operator "." is used. For example, `employee[x] .month` would be used to access the month of birth for each employee.To access the first letter of an employee's name you would write `employee[x] .id[0]`

6.5.4 Nested Structures

```
struct BIRTH {
      char month;
      char day;
      int year;
};

struct employee_info{
      char name[20];
      char id[9];
      struct BIRTH birthdate;
} employee[20];
```

Example:

Construct a structure that holds the dimensions and the volume of a box. Declare a variable `Box_1` that is of type `box`.

```
struct box {
      float height;
      float width;
      float depth;
      float volume;
};
struct box Box_1;
```

Construct a structure that holds the dimensions and the volume of a box. Declare a variable `Box_1` that is of type box.

Structures Example

Example:

Initialize `Box_1` so that its height is 2.3 in, width is 8.5 in and its depth is 4 in.

Example:

```
Box_1.height = 2.3;
Box_1.width = 8.5;
Box_1.depth = 4;
```

Initialize `Box_1` so that its height is 2.3 in, width is 8.5 in and its depth is 4 in.

6.6 Accessing I/O Registers

I/O registers are located in the 64 I/O register address space and 160 Extended I/O register address space. Even though these registers are mapped to the Data Memory, they are registers. Therefore, special compiler commands are used to access the AVR microcontroller's I/O registers. These commands are referred to as *Special Function Registers* (SFR), and the `sfrb` and `sfrw` keywords indicate to the compiler that `in` and `out` assembly instructions, respectively, are to be used to access I/O registers. The following shows an example code where an 8-bit value is read from PORTD.

```
/* Define Special Function Registers */
sfrb PIND=0x19;               // 8-bit access to PINA
sfrb DDRD=0x11;               // 8-bit access to DDRD

void main(void)
{
      unsigned short x;

      DDRD=0x00               // Set all bits of PORT D to input
      x=PIND                  // Read PORTD input pins

      while(1)
      ...
}
```

{*(BL): Need to check, may be different for Atmega128*}

```
sfrb OSCCAL=4;    // Oscillator Calibration Value
sfrb PINE=5;            // Port E Input Pins
sfrb DDRE=6;            // Port E Data Direction Register
sfrb PORTE=7;           // Port E Data Direction Register
sfrb ACSR=8;            // Analog Comparator Control And Status Register
sfrb UBRRL=9;
sfrb UCSRB=0xa;
sfrb UCSRA=0xb;
sfrb UDR=0xc;           // USART I/O Data Register
sfrb SPCR=0xd;
sfrb SPSR=0xe;
sfrb SPDR=0xf;
sfrb PIND=0x10;   // Port D Input Pins
sfrb DDRD=0x11;   // Port D Data Direction Register
sfrb PORTD=0x12;  // Port D Data Direction Register
sfrb PINC=0x13;   // Port C Input Pins
```

```
sfrb DDRC=0x14;   // Port C Data Direction Register
sfrb PORTC=0x15;  // Port C Data Direction Register
sfrb PINB=0x16;   // Port B Input Pins
sfrb DDRB=0x17;   // Port B Data Direction Register
sfrb PORTB=0x18;  // Port B Data Direction Register
sfrb PINA=0x19;   // Port A Input Pins
sfrb DDRA=0x1a;   // Port A Data Direction Register
sfrb PORTA=0x1b;  // Port A Data Register
sfrb EECR=0x1c;   // EEPROM Control Register
sfrb EEDR=0x1d;   // EEPROM Data Register
sfrb EEARL=0x1e;
sfrb EEARH=0x1f;
sfrw EEAR=0x1e;   // 16 bit access
sfrb UBRRH=0x20;
sfrb UCSRC=0x20;
sfrb WDTCR=0x21;
sfrb ICR1L=0x24;
sfrb ICR1H=0x25;
sfrw ICR1=0x24;   // Input Capture Register 1   // 16 bit access
sfrb OCR1BL=0x28;
sfrb OCR1BH=0x29;
sfrw OCR1B=0x28; // 16 bit access
sfrb OCR1AL=0x2a; Output Compare Register 1A
sfrb OCR1AH=0x2b;
sfrw OCR1A=0x2a;  // Output Compare Register 1A // 16 bit access
sfrb TCNT1L=0x2c; // Timer/Counter1 Low Byte
sfrb TCNT1H=0x2d; // Timer/Counter1 High Byte
sfrw TCNT1=0x2c;  // Timer/Counter1 // 16 bit access
sfrb TCCR1B=0x2e; // Timer/Counter1 Control Register B
sfrb TCCR1A=0x2f;       // Timer/Counter1 Control Register A
sfrb SFIOR=0x30;  // Special Function IO Register
sfrb OCR0=0x31;   // Output Compare Register 0
sfrb TCNT0=0x32;  // Timer/Counter0
sfrb TCCR0=0x33;  // Timer/Counter0 Control Register
sfrb MCUCSR=0x34;
sfrb MCUCR=0x35;
sfrb EMCUCR=0x36;
sfrb SPMCR=0x37;
sfrb TIFR=0x38;   // Timer/Counter Interrupt Flag register
sfrb TIMSK=0x39;  // Timer/Counter Interrupt Mask Register
sfrb GIFR=0x3a;
sfrb GICR=0x3b;
sfrb SPL=0x3d;          // Stack Pointer Low
sfrb SPH=0x3e;    // Stack Pointer High
```

6.6.1 Bit-level access to I/O Registers

The bit level-access to the I/O registers is accomplished using *bit selectors* appended after the name of the I/O registers.

Because bit level access to I/O registers is done using the `CBI`, `SBI`, `SBIC` and `SBIS` instructions, the register address must be in the 0 to 1Fh range for `sfrb` and in the 0 to 1Eh range for `sfrw`.

```
sfrb PORTA=0x1b;
sfrb DDRA=0x18;
strb PINA=0x19;

void main(void) {
      /* set bit 0 of Port A as output */
      DDRA.0=1;

      /* set bit 1 of Port A as input */
      DDRA.1=0;

      /* set bit 0 of Port A  output */
      PORTA.0=1;

      /* test bit 1 input of Port A */
      if (PINA.1) {
            /* place some code here */
      };
      /*  */
}
```

To improve the readability of the program symbolic names can be defined for the bits in I/O registers:

```
      sfrb PINA=0x19;
      #define alarm_input PINA.2

      void main(void)
      {
            /* test bit 2 input of PORT a*/
            if (alarm_input) {
                  /* place some code here */
            };
            /*  */
      }
```

Note that bit access to I/O registers located in internal SRAM above address 5Fh (such as PORTF for the ATmega128 for example) will not

work, because the CBI, SBI, SBIC and SBIS instructions cant be used for SRAM access.

6.7 Accessing EEPROM

Accessing AVR Program Memory is accomplished using the `eeprom` keyword.

```
/* The value 1 is stored in the EEPROM during chip programming */
eeprom int alfa = 1;
eeprom char beta1;
eeprom long array1[5];

/* The string is stored in the EEPROM during chip programming */
eeprom char string[] = "Hell";
void main(void){
      int i;

      /* Pointer to EEPROM */
      int eeprom *ptr_to_eeprom;
      /* Write the value 0x55 directly to the EEPROM */
      alfa = 0x55;
      /* or indirectly by using a pointer */
      ptr_to_eeprom = &alfa;
      *ptr_to_eeprom =0x55;

      /* Read directly the value from the EEPROM */
      i = alfa;
      /* or indirectly by using a pointer */
      i = *ptr_to_eeprom;
}
```

Note that pointers to the EEPROM are always 16 bits.

6.8 Using Interrupts

Access to the AVR interrupt system is implemented with the `interrupt` keyword.

```
/* Vector numbers are for the AT90S8515 */
/* Called automatically on external interrupt */
interrupt [2] void external_int0(void)
{
```

```
      /* Place your code here */
}

/* Called automatically on TIMER0 overflow */
interrupt [8] void timer0_overflow(void)
{
      /* Place your code here */
}
```

Interrupt vector numbers start with 1.

The compiler will automatically save the affected registers when calling the interrupt functions and reset them back on exit.

A `RETI` assembly instruction is placed at the end of the interrupt function.

Interrupt functions cannot return a value or have parameters.

You must also set the corresponding bits in the peripheral control registers to configure the interrupt system and enable the interrupts.

The automatic saving and restoring of registers affected by the interrupt handler, can be turned on or off using the `♯pragma savereg` directive.

```
/* Turn registers saving off */
#pragma savereg-

/* interrupt handler */
interrupt [1]  void my_irq(void){

/* Save only the registers that are affected by the interrupt handler, e.g., R30, R31 and SREG */
#asm
      push  r30
      push  r31
      in    r30, SREG
      push  r30
#endasm

/* place the C code here            */
/* ....                             */
/* now restore SREG, R31 and R 30   */

#asm
      pop   r30
      out   SREG, r30
      pop   r31
      pop   r30
```

```
#endasm
}

/* re-enable register saving for the other interrupts */
#pragma savereg+
```

The default is to automatically save registers during interrupts.

For every program, the CodeVisionAVR C complier automatically generates a code sequence to make the following initializations immediately after the AVR chip reset:

- interrupt vector jump table
- global interrupt disable
- EEPROM access disable
- Watchdog Timer disable
- external SRAM access and wait state enable if necessary
- clear register R2R14
- clear the SRAM
- initialize the global variables located SRAM
- initialize the Data Stack Pointer register Y
- initialize the Stack Pointer register SP
- initialize the UBPR register if necessary

A basic STARTUP.ASM file is supplied with the complier distribution and is located in the ..\BIN directory

```
; CodeVisionAVR C Compiler
; (C) 1998-2004 Pavel Haiduc, HP InfoTech s.r.l.

; EXAMPLE STARTUP FILE FOR CodeVisionAVR V1.24.1 OR LATER

      .EQU __CLEAR_START = 0X60     ; START ADDRESS OF SRAM AREA TO CLEAR
                                    ; SET THIS ADDRESS TO 0X100 FOR THE
                                    ; ATmega128 OR ATmega64 CHIPS
      .EQU __CLEAR_SIZE = 256       ; SIZE OF SRAM AREA TO CLEAR IN BYTES
```

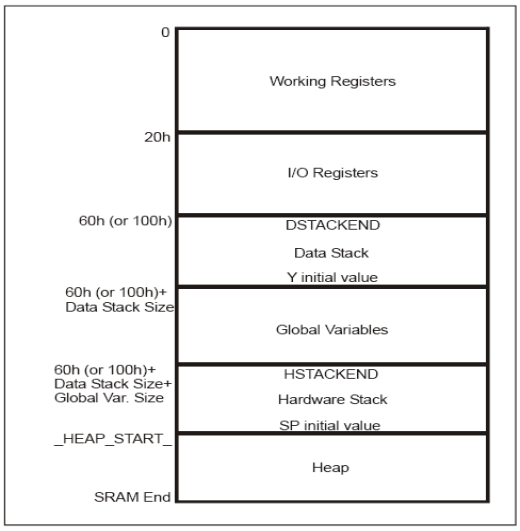


Figure 6.1: Program in Data Memory.

```
      CLI                           ; DISABLE INTERRUPTS
      CLR   R30
      OUT   EECR,R30                ; DISABLE EEPROM ACCESS

; DISABLE THE WATCHDOG
      LDI   R31,0x18
      OUT   WDTCR,R31
      OUT   WDTCR,R30

      OUT   MCUCR,R30               ; MCUCR=0, NO EXTERNAL SRAM ACCESS

; CLEAR R2-R14
      LDI   R24,13
      LDI   R26,2
      CLR   R27
_CLEAR_REG:
      ST    X+,R30
      DEC   R24
      BRNE  _CLEAR_REG
```

You can include assembly code anywhere in your program using the ♯`asm` and ♯`endasm` directives

Example:

```
void delay(unsigned char i)  {
while (i--)  {
            /* Assembly language code sequence */
            #asm
                  nop
                  nop
            #endasm
            };
}
```

Inline assembly may also be used. Example:

```
#asm(sei) /* enable interrupts */
```

The following example shows how to access functions written in assembly language from a C program:

```
// function in assembler declaration
// this function will return a+b+c
#pragma warn-                 // this will prevent warnings
int sum_abc(int a, int b, unsigned char c) {
#asm
      ldd   r30, y+3    ; R30=LSB a
      ldd   r31, y+4    ; R31=MSB a
      ldd   r26, y+1    ; R26=LSB b
      ldd   r27, y+2    ; R27=LSB b
      add   r30, r26    ; (R31, R30)=a+b
      adc   r31, r27
      ld    r26, y      ; R26=c
      clr   r27         ; Promote unsigned char c to int
      add   r30, r26    ; (R321, R30)=(R31, R30)+c
      adc   r31, r27
#endasm
}
#pragma warn+                 // enable warnings

void main(void) {
      int r;
      // now we call the function and store the result  in r
      r = sum_abc(2, 4, 6);
}
```

Chapter 7

Digital Components

Digital components are fundamental building blocks for any digital systems. They include decoders/encoders, multiplexers, counters, registers, memories, and Arithmetic and Logic Units (ALUs), and are implemented using basic logic gates, such as NAND, NOR, NOT, etc., and memory elements. Understanding how digital components work is important because microarchitecture implementation, or implementation of any digital systems for that matter, involves modular design using these basic components. Therefore, this chapter reviews some basic concepts in digital system design and how they relate to design of microarchitectures.

Figure 7.1 shows the basic microarchitecture of the AVR processor, which will be discussed in detail in Chapter 8. It consists of memories for program and data, multiplexers (MUXA-MUXJ), registers (PC, IR, DMAR, and NPC), Register File, adders (+ and Address Adder), Arithmetic and Logic Unit (ALU), Concatenation (||) Unit, Zero Fill (zf) Unit, Sign Extension (se) Unit, and several logic components that are specialized for the AVR microarchitecture, such as Alignment Unit and Register Address Logic. There are also some hidden components, mainly decoders that are integrated into memories and Register File.

In the following sections, we discuss the functionality and implementation of decoders, multiplexers, registers, register file, and memory, and the roles they play in a microarchitecture. The specialized components will be discussed later in Chapter 8, and a detailed discussion of Arithmetic and Logic Units (ALUs) will be provided in Chapter 9.

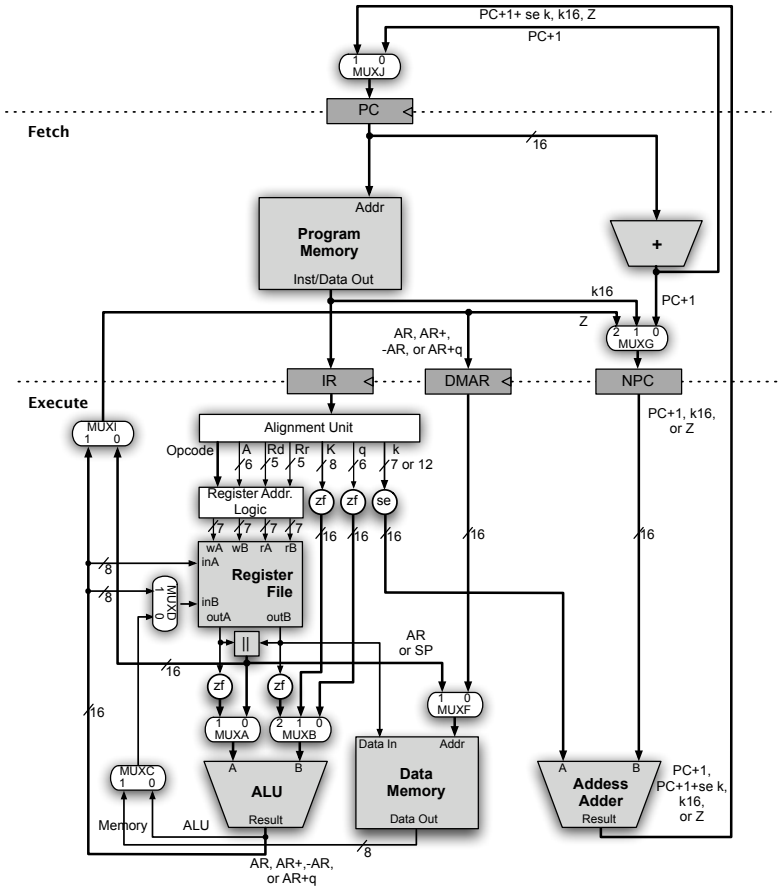


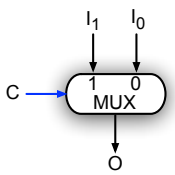
Figure 7.1: Digital components in the AVR microarchitecture.

7.1 Multiplexers

A *multiplexer* (MUX) is a digital switch that connects data from one of 2^k inputs to its output using k select input lines. In the AVR microarchitecture, MUXs are used to choose among multiple input sources. For example, MUXJ in Figure 7.1 allows the PC register to latch the address from either the Address Adder or the Address Incrementer (+).

Figure 7.2 shows the logic symbol, function table, K-map, and logic diagram for a 2-to-1 MUX. The input C is used to select from either input I_0 or I_1, which then appears on the output O.

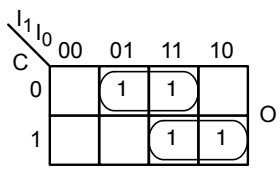
In microarchitecture design, MUXs are typically used to select a single



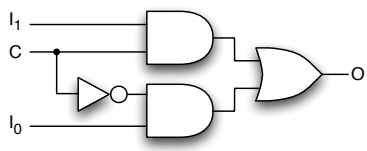
(a) Logic symbol.

Input			Output
I_1	I_0	C	O
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(b) Function table.



(c) K-map.



(d) Logic diagram.

Figure 7.2: 2-to-1 MUX.

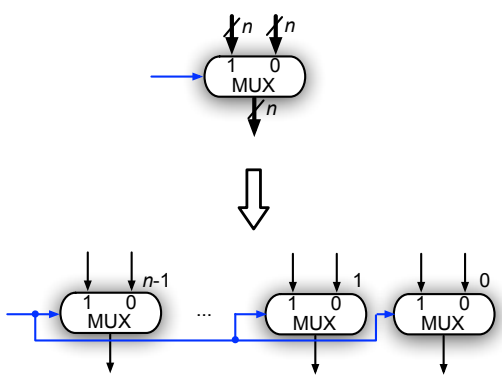


Figure 7.3: n-bit 2-to-1 Multiplexer.

input consisting not just a single bit but n bits. Figure 7.3 shows an n-bit 2-to-1 MUX, which simply consists of n 2-to-1 MUXs connected in parallel.

Figure 7.4 shows the logic equations for 4-to-1 and 8-to-1 MUXs. As can be seen, the number of inputs for AND and OR gates grows as n grows.

- 4:1 mux: $Z = A'\,B'\,I_0 + A'\,B\,I_1 + A\,B'\,I_2 + A\,B\,I_3$
- 8:1 mux: $Z = A'B'C'I_0 + A'B'CI_1 + A'BC'I_2 + A'BCI_3 + AB'C'I_4 + AB'CI_5 + ABC'I_6 + ABCI_7$

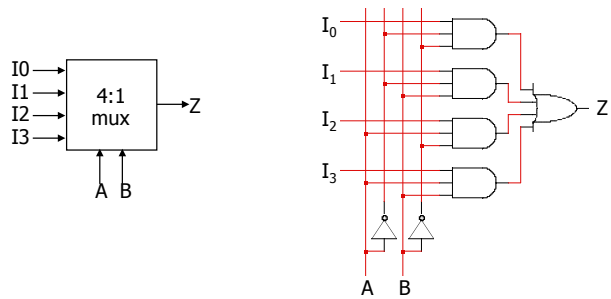


Figure 7.4: Examples of 4-to-1 and 8-to-1 Muxes.

For example, the number of inputs to the OR gate for 8-to-1 MUX is eight. This causes a problem where the switching delay increases with n, referred to as the *fan-in problem*. The limit on fan-in is typically 4.

In order to mitigate the fan-in problem, larger MUXs can built by cascading smaller MUXs. Figure 7.5 shows two different examples of implementing an 8-to-1 MUX using a combination of 2-to-1 and 4-to-1 MUXs. As can be seen, the difference between the two implementations is the way the input select lines ABC are used to control the MUXs. For the implementation on the left, the most significant input select bit (i.e., A) selects either the first or second half of the inputs, and the input select bits BC select one of the four inputs for each of the two 4-to-1 MUXs. In contrast, the implementation on the right uses the two most significant input select bits (i.e., AB) to select one of the four two-bit inputs, and the input select bit C is used to select one of the two inputs for each of the four 2-to-1 MUXs.

A *demultiplexer* (DEMUX) takes a single input and forwards it to one of its n outputs based on the bit pattern on the select lines. DEMUX is complementary to MUX, and both are often used to merge and separate information.

7.2 Decoders

In general, a *decoder* is a multiple-input, multiple-output logic circuit that converts coded inputs to coded outputs, where inputs and outputs are dif-

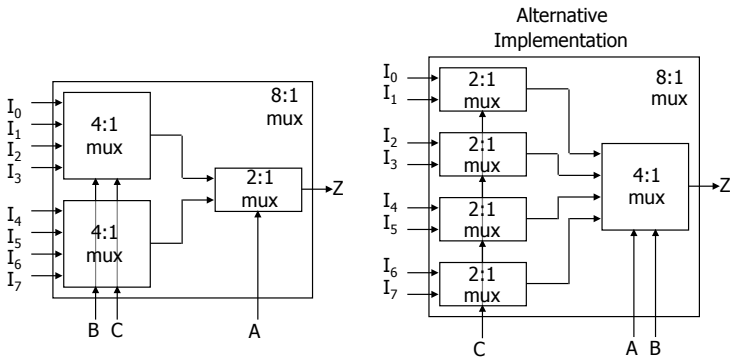


Figure 7.5: Cascaded MUXs.

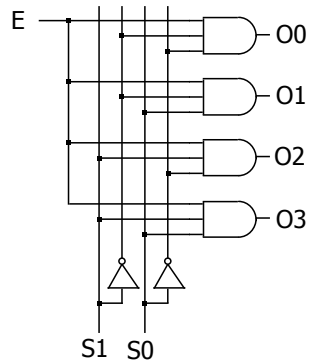
ferent. There are many types of decoders. In Chapter 8, we will see an example decoder called *instruction decoder* that decodes the opcode part of instructions and generates control signals to control the operations of the datapath. In this chapter, we discuss a more common decoder circuit called a *binary decoder*, which is an n-input, 2^n-output logic circuit that provides exactly one output to be 1 (or 0) and all the rest of the outputs are 0s (or 1s). The one output that is logically 1 is the output corresponding to the input pattern that it is expected to detect. Binary decoders are commonly used to select one of many components (i.e., words/registers) in memories and Register files.

Figure 7.6 shows the function table and logic diagram for a 2-to-4 decoder. As can be seen from the truth table, the output Oi is 1 only when the input $S1S0$ corresponds to the binary representation of i. This decoder also has an *enable* (E) input to enable or disable the operation of the decoder. Thus, when E is 0, outputs are 0s.

As mentioned before, binary decoders are used as an front-end to memories and register files. Figure 7.7 illustrates how a 4-to-16 decoder is used as an address select logic for a memory containing 16 1-bit words. The 4-bit address input $A_3A_2A_1A_0$ is used to enable the corresponding memory or RAM cell. We will discuss more on reading or writing from/to a selected memory cell in Section 7.5.1.

Input			Output			
E	S1	S0	O3	O2	O1	O0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

(a) Function table.



(b) Logic diagram.

Figure 7.6: 2-to-4 decoder.

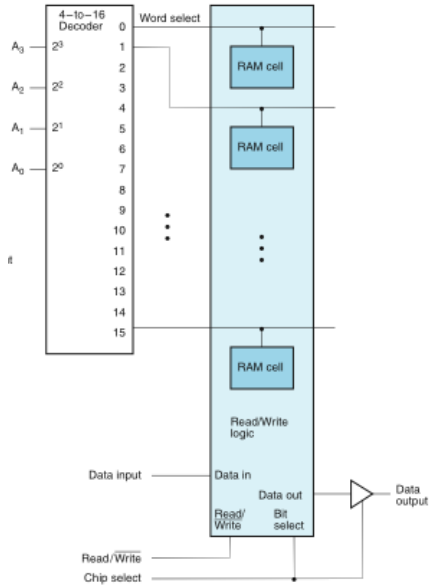
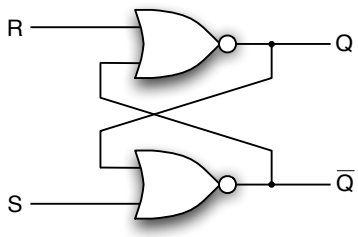


Figure 7.7: Decoder front-end for memories.

7.3 Memory Elements

A storage or *memory element* can maintain a binary state indefinitely until directed by an input signal to switch states. The most basic storage elements are *latches*, from which *flip-flops* can be constructed. Although the terms



(a) NOR implementation.

S	R	Q
0	0	No change
0	1	0
1	0	1
0	0	Not allowed

(b) Function table.

Figure 7.8: S-R latch.

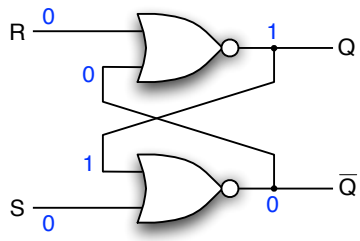
latch and flip-flop are used interchangeably, a *latch* refers to a sequential device that continuously watches its input and can change its output at any time, while a *flip-flop* refers to a sequential device that samples its inputs and changes its output only when a clocking signal is asserted.

7.3.1 Latches

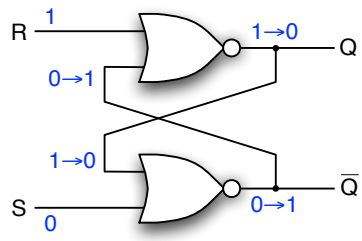
Figure 7.8 shows the *Set-Reset* (S-R) latch and its function table. The S-R latch is the most fundamental latch and is constructed from two cross-coupled NOR gates. The circuit has two inputs, S and R, and two outputs, Q and $\overline{Q}$. Figure 7.9 illustrates the operations of the S-R latch. If S and R are both 0s, the circuit's feedback loop retains one of the two logic states, i.e., $Q = 0$ or $Q = 1$. A logic 1 can be written to S-R latch by setting $S = 1$ and $R = 0$. Conversely, a logic 0 can be written to S-R latch by setting $S = 0$ and $R = 1$. If both S and R are set to 1s, both outputs will be 0s, which violates the requirement that the outputs be the complement of each other. Moreover, when inputs are simultaneously returned to 0s, the circuit may go into the metastable state where the output oscillates between 0 and 1. In normal operations, these problems are avoided by guaranteeing that that bot inputs are not 1s.

Alternatively, a S-R latch can also be constructed using NAND gates. Figure 7.10 shows S-R latch based on NAND gates. As can be seen, the main difference between the two implementations is that their inputs are flipped. Thus, the NAND implementation is also referred to as $\overline{S}$-$\overline{R}$ latch, where set and reset inputs are active low.

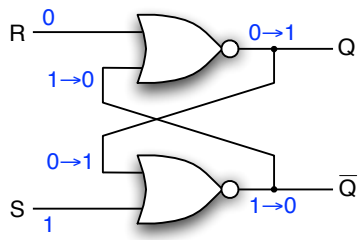
The operation of the basic S-R latch can be modified by providing an additional control input that determines when the state of the latch can



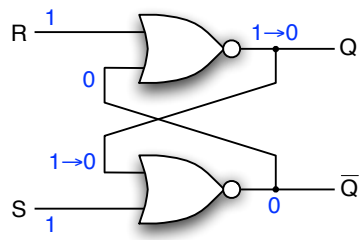
(a) S=0, R=0 (no change); initially Q=1.



(b) S=0, R=1 (reset); initially Q=1.

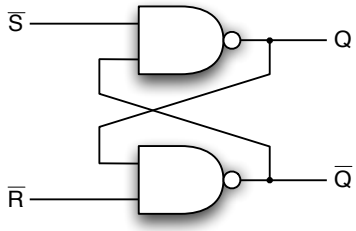


(c) S=1, R=0 (set); initially Q=0.



(d) S=1, R=1 (not allowed); initially Q=1.

Figure 7.9: Operations of S-R latch.



(a) NAND implementation.

$\overline{S}$	$\overline{R}$	Q
0	0	Not allowed
0	1	1
1	0	0
1	1	No change

(b) Function table.

Figure 7.10: $\overline{S}$-$\overline{R}$ latch.

be changed. Figure 7.11 shows a S-R latch with *enable*. The two NAND gates together with control signal (C) determine whether or not set-reset operations are activated. This allows a S-R latch to be written to only when

certain condition is true.

As discussed before, inputs to a S-R latch cannot be both 1s at the same time, otherwise unpredictable behavior occurs. This can be avoided using a *D latch with enable* shown in Figure 7.12, where an inverter is added to generate complements of S and R inputs. The control input (labeled C in the Figure) is active high and serves as either an enable or clock signal.

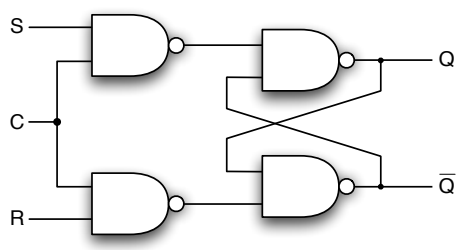
7.3.2 Flip-Flops

A flip-flop contains a latch and its state is triggered by a change in the control input.

Edge-Triggered D Flip-Flop

Unlike level-triggered flip-flops we saw in Figures 7.11 and 7.12, *edge-triggered flip-flops* change their states only at the falling or rising edge of a controlling clock signal. This feature is critical in synchronous circuits, where output transitions are synchronized with a clock edge.

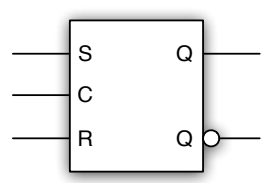
Figure 7.13 shows a *negative* edge-triggered D flip-flop. An edge-triggered flip-flop in general contains two latches referred to as *master* and *slave*. The



(a) Circuit using $\overline{S}$-$\overline{R}$ latch.

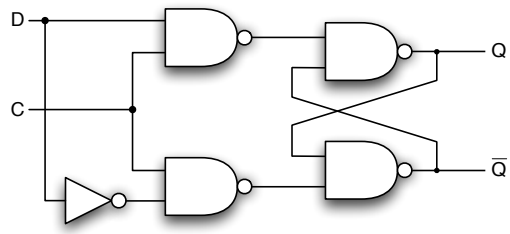
S	R	C	Q
0	0	1	No change
0	1	1	0
1	0	1	1
1	1	1	Not allowed
×	×	0	No change

(b) Truth table.



(c) Logic symbol.

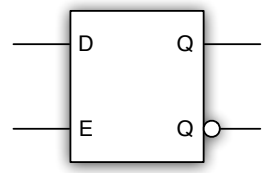
Figure 7.11: S-R latch with enable.



(a) Circuit using $\overline{S}$-$\overline{R}$ latch.

D	E	Q
0	1	0
1	1	1
×	0	No change

(b) Truth table.



(c) Logic symbol.

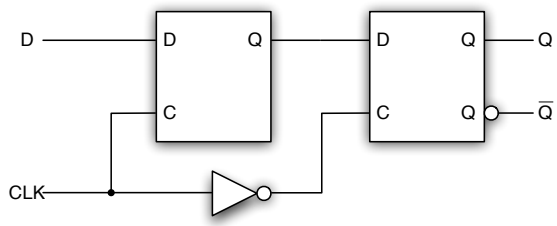
Figure 7.12: D latch with enable.

slave section is basically the same as the master section except that it is clocked on the inverted clock pulse and is controlled by the outputs of the master section rather than by the external inputs.

The master latch is open or enabled and follows the input D when CLK is 1. When CLK transitions to 0, the master latch is disabled and its output appears on the input of the slave latch. The slave latch is enabled all the while CLK is 0, but changes only at the beginning of this interval since the master latch is disabled and does not change during the rest of the interval.

The logic symbol contains a triangle and a circle on the CLK input to indicate that this is negative and edge-trigger, respectively.

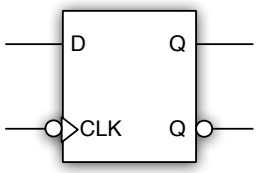
Figure 7.14 illustrates the functional behavior of a negative-edge-triggered D flip-flop. The clock periods t and $t+1$ indicate when the signal X (which can be 0 or 1) is applied to the input D and when X appears at the output Q, respectively. Consider the timing just before the clock signal transitions from 1 to 0 (indicated by '↓'). Since the clock signal is 1, the Master D flip-flop latches the signal X, while the Slave D Flip-flop is disabled and thus holds the previous output, i.e., Q(t-1). When the clock signal transitions from 1 to 0, the Master D flip-flop becomes disabled, while the Slave D flip-flop becomes enabled. This causes the signal X at the output of the



(a) Circuit using D latches.

D	CLK	Q
0	↓	0
1	↓	1
×	0	No change
×	1	No change

(b) Function table.



(c) Logic symbol.

Figure 7.13: Negative-edge triggered D-FF.

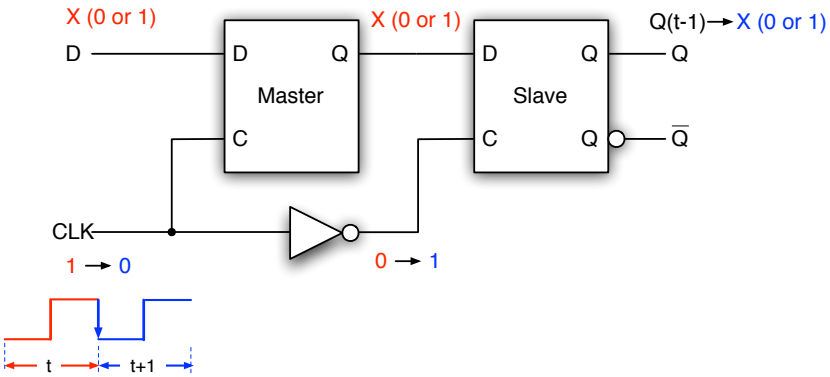
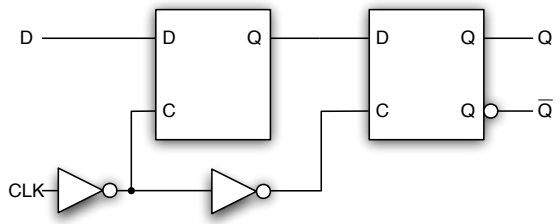


Figure 7.14: Functional behavior of a negative-edge-triggered D flip-flop.

Master D flip-flop to be latched onto the Slave D flip-flop, and at the same time prevents any new signal from being latched onto the Master flip-flop.

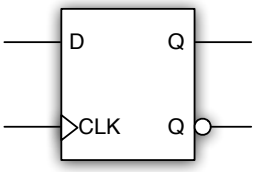
Figure 7.15 shows a positive-edge triggered D flip-flop. Its operations are similar to a negative edge-triggered flip-flop except the CLK input is inverted so that all the signals transition during the rising edge of CLK.



(a) Circuit using D latches.

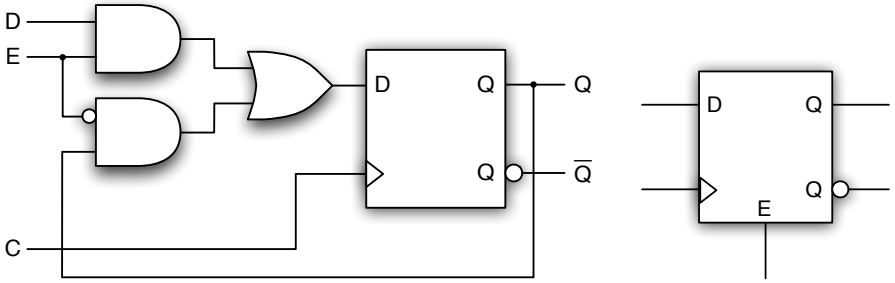
D	CLK	Q
0	↑	0
1	↑	1
×	0	No change
×	1	No change

(b) Function table.



(c) Logic symbol.

Figure 7.15: Positive-edge triggered D-FF.



(a) Logic diagram. (b) Logic symbol.

Figure 7.16: Positive-edge triggered D flip-flop with high enable.

7.3.3 Edge-Triggered D Flip-Flop with Enable

One problem with D flip-flops discussed previously is that the system clock, which supplies a continuous train of clock pulses, will cause the output Q will change if input D changes. This is not necessarily a desirable characteristics, and instead a designer may need to control when data should be latched onto a register. This is accomplished by adding an *enable* (E) input. Figure 7.16

shows the circuit design of a positive-edge-triggered D flip-flop with high enable. This circuit contains a 2-to-1 MUX in front of the input to the D flip-flop to select either the external input D or the flip-flop's current output Q. Thus, unless the enable signal E is asserted, the flip-flip will maintain the last value stored.

7.4 Registers

A *register* consists of a set of commonly clocked D flip-flops together with additional logic to determine the new data to be latched onto the flip-flops. Registers in a microarchitecture are used to temporarily hold and separate information among various parts of the datapath. The role of registers can be simple as holding a set of bits or complicated as performing a variety of functions.

7.4.1 n-bit Register

Figure 7.17 shows a clocked n-bit register with parallel load capability. Similar to Figure 7.16, each input to the D flip-flop has a 2-to-1 MUX to select either the external input D_i or the flip-flop's current output Q_i.

7.4.2 Shift Registers

A *shift register* is an n-bit register with the capability to shifts its data one bit left (down) or right (up) with each tick of the clock. Figure 7.18 shows a *serial-in, serial-out 4-bit shift register*. Shift registers are useful in many applications. One of its common use is to convert between serial and parallel interfaces. Shift registers can also be used as simple delay circuits. Several bi-directional shift registers could also be connected in parallel for a hardware implementation of a stack. In microprocessors, shift registers are used to handle data processing. Most assembly languages provide "shift left" and "shift right" instructions that perform multiply by two and divide by two, respectively, with each shift. These instructions can also be used together with the *carry bit* (C-bit) to test bits in a register.

Sometimes it is also necessary to have parallel load as well as shift capabilities. Such a circuit can be implemented by combining the shift registers in Figure 7.18 with n-bit register with parallel load capability in Figure 7.17. Figure 7.19 shows the logic diagram and function table for a 4-bit *bidirectional shift register with parallel load*. Input to each flip-flop is a 4-to-1 MUX

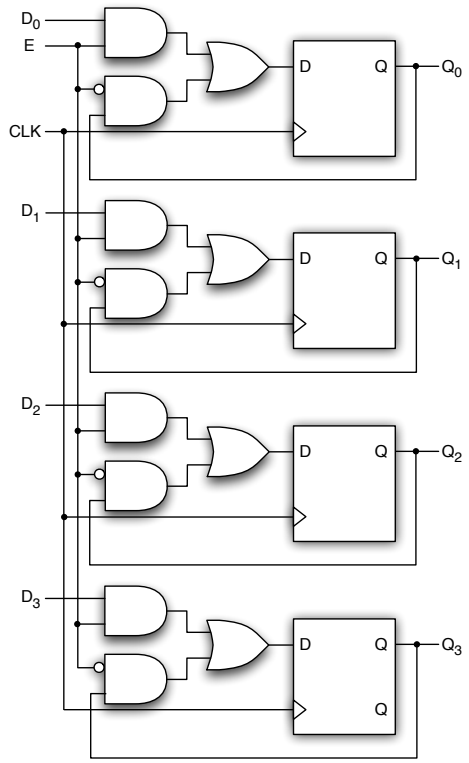
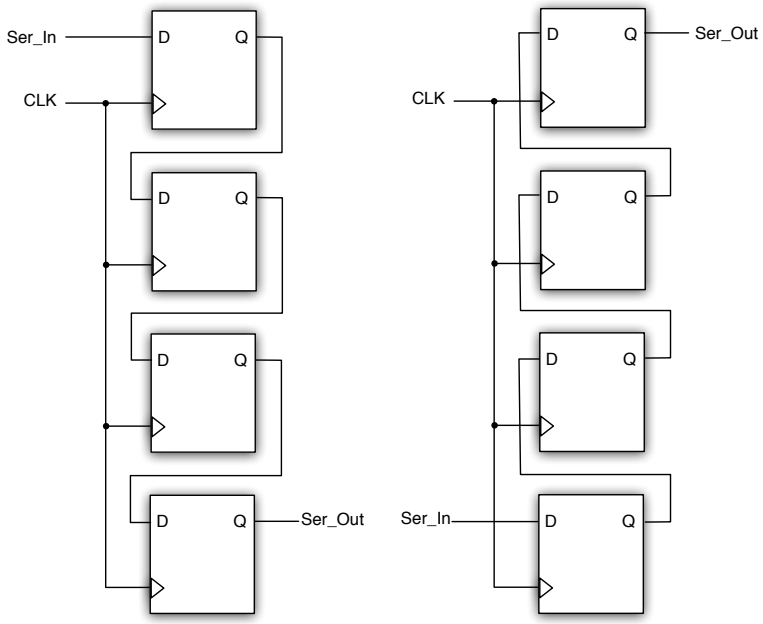


Figure 7.17: Register with load enable.

that chooses among (1) serial-in right, (2) the current stored value (i.e., Q), (3) input data (i.e., D), or (4) serial-in left.

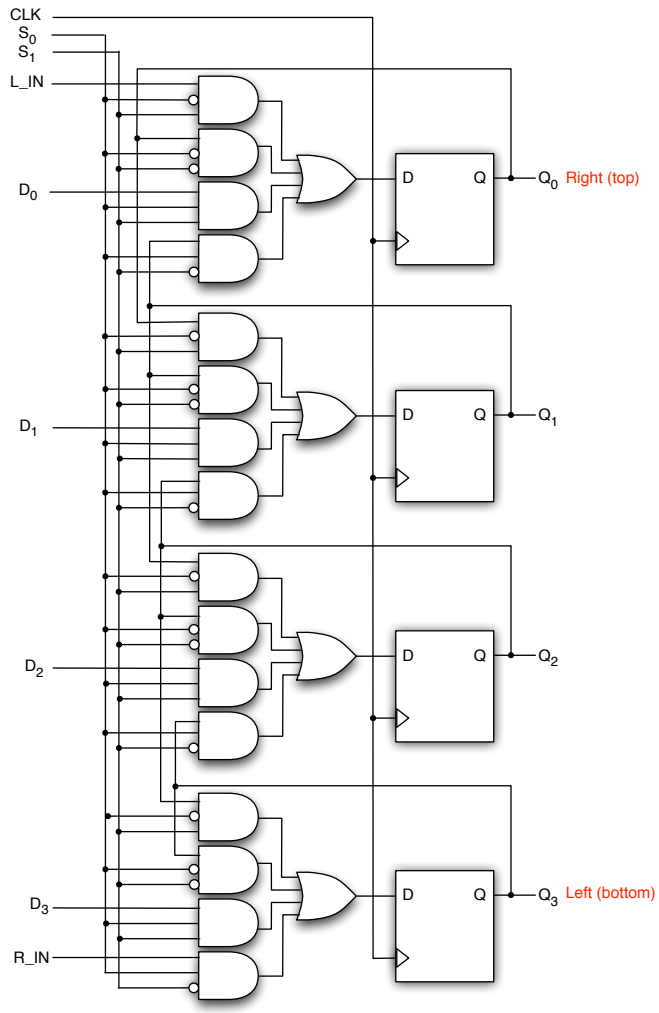
7.5 Memory

The AVR microarchitecture has two memories: Program Memory and Data Memory. Although the technologies used in the two memories are different (Program Memory is based on flash memory, while Data Memory is based on static RAM), their fundamental operations are similar. Thus, we will not distinguish the two and instead focus on the structure of memories using basic memory elements and registers discussed in Sections 7.3 and 7.4.



(a) Left (down). (b) Right (up).

Figure 7.18: 4-bit shift register.



(a) Logic diagram.

Function	Control Input S_1	S_0	Next State Q_3	Q_2	Q_1	Q_0
Hold	0	0	Q_3	Q_2	Q_1	Q_0
Shift right	0	1	R_IN	Q_3	Q_2	Q_2
Shift left	1	0	Q_2	Q_1	Q_0	L_IN
Load	1	1	D_3	D_2	D_1	D_0

(b) Function table.

Figure 7.19: Bidirectional shift register with parallel load.

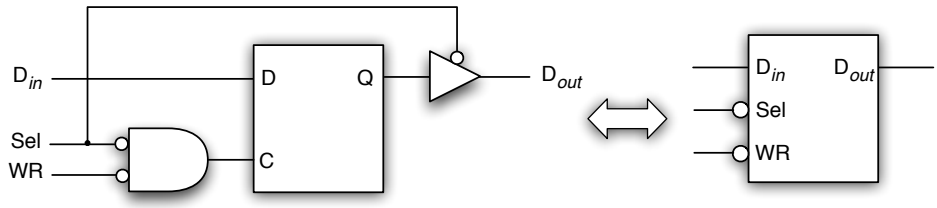


Figure 7.20: Logic diagram of a SRAM cell.

7.5.1 Static RAM (SRAM)

Static RAM (SRAM) is basically made up of D flip-flops and is used in registers and Data Memory of the AVR processor.

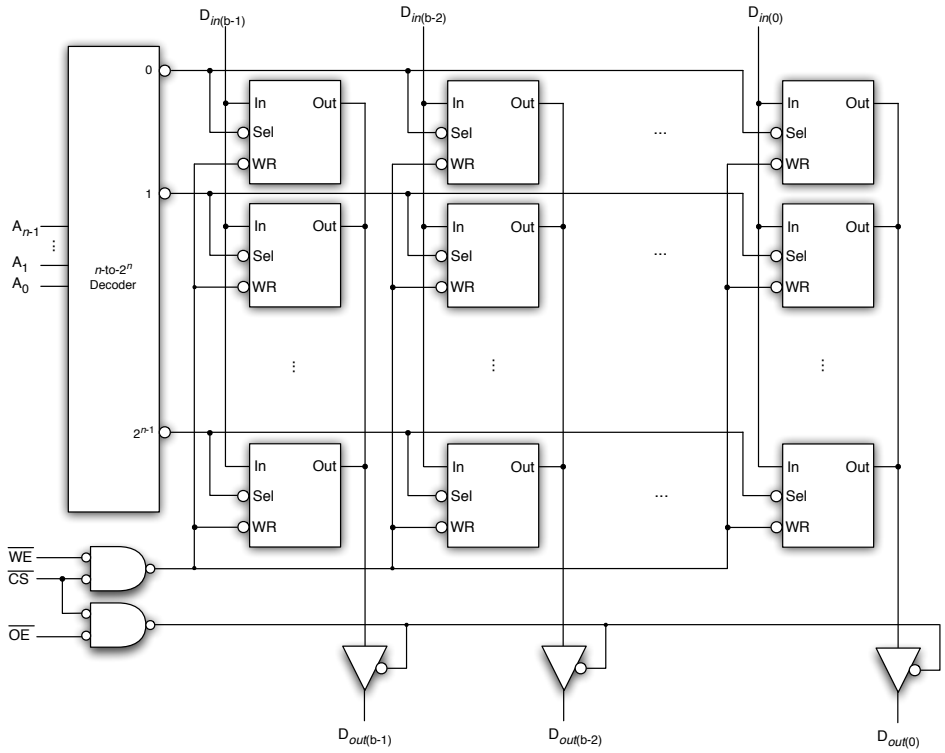
SRAM Cell

Figure 7.20 shows the functional behavior of a SRAM cell. The storage element in each cell is a D flip-flop controlled by an equivalent NOR gate with Select (*Sel*) and Write (*WR*) inputs and a tri-state buffer controlled by *Sel*. Note that *Sel* and *WR* are low-enabled meaning when both signals are low the control is enabled. In order to read the bit stored in the cell, *Sel* is set 0 to enable the tri-state buffer. In order to write a bit into the cell, both *Sel* and *WR* signals are set to 1s. It is also important to note that when *Sel* is high D_{out} is in high-impedance state, which causes the cell to be isolated from the rest of the SRAM structure.

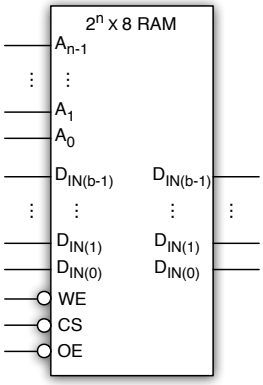
SRAM Structure

A complete static RAM structure is constructed using an array of SRAM cells with additional logic. Figure 7.21 shows the structure of a 2^n by b-bit SRAM. As discussed in Section 7.2, the front-end of the SRAM structure contains a n-to-2^n decoder that accepts an n-bit address and selects one of the 2^n outputs. This enables one of the rows or words to be accessed.

The control signals for the SRAM structure consists of *Write Enable* ($\overline{WE}$), *Chip Select* ($\overline{CS}$), and *Output Enable* ($\overline{OE}$). When $\overline{WE}$ is asserted, the input data D_{in} is written to the selected word. $\overline{OE}$ enables the tri-state buffer allowing data to be read from the SRAM. Finally, the $\overline{CS}$ input can be thought of as the main switch for the SRAM and provides flexibility in controlling multiple SRAM structures. Thus, either $\overline{WE}$ or $\overline{OE}$ together



(a) Logic Diagram.



(b) Logic Symbol.

Figure 7.21: 2^n by b-bit SRAM structure.

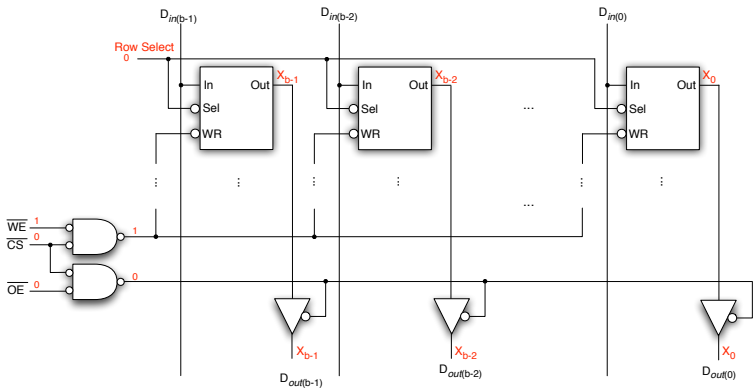


Figure 7.22: Read operation.

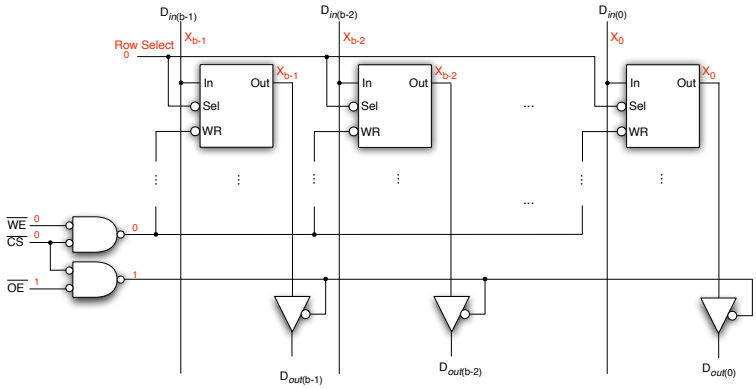


Figure 7.23: Write operation.

with $\overline{CS}$ are asserted to make the SRAM operational.

Figure 7.22 illustrates a read operation. A read is performed by providing an address of the word to be read to the decoder front-end, which then enables the selected word or row (via Row Select signal), and setting $\overline{OE}$ and $\overline{CS}$ to 0s. This enables the tri-state buffers allowing the word to appear on the output D_{out}.

Figure 7.23 illustrates a write operation. A write operation is also performed by providing both the word and the address of the word to be written and setting $\overline{CS}$) to 0. This selects the corresponding word (via Row Select signal) and asserts the $\overline{WR}$ inputs, which causes input data X to be written to the selected row of SRAM cells.

SRAM versus Dynamic RAM

In contrast to SRAMs, *dynamic RAMs* (DRAMs) have a much higher capacity. This is because each DRAM cell consists of a capacitor and a transistor compared to six transistors for a SRAM cell, called *6T-cell*. However, DRAMs are much slower than SRAMs for several reasons. First, because a DRAM has more memory words, its decoder front end is much larger resulting in more delay. Second, the charge stored in the capacitor, which represents logic 1, leaks over time, and therefore, all the cell have to be periodically refreshed. Finally, each read operation discharges the charge stored in the cell, and thus, has to be followed by a write to restore the original value in the cell.

In general, microcontrollers, such as AVR, do not use DRAMs due to its slow speed and minimal memory requirements. However, high-speed processors in high-end mobile devices, e.g., smartphones, and PCs use a combination of SRAMs and DRAMs to implement a hierarchy of memory. For example, these processors have registers and caches (Level-1 and Level-2) implemented as SRAMs, and then have main memory based on DRAMs.

The discussion of the memory hierarchy is beyond the scope of this book, and interested readers are referred to a number of excellent book on computer architectures, such as 'Computer Organization and Design: The Hardware/Software Interface' by Patterson and Hennessy (Elsevier Morgan Kaufmann, 2009).

7.5.2 Building Bigger and Wider Memories

Memory often needs to be expanded to accommodate larger applications and data. For example, ATmega128 AVR processor has 4,966×8-bit internal SRAM, and is expandable to 64K×8-bits using external memory. Obviously, a simple solution is to provide a single SRAM chip that matches the required size. However, a large memory can also be built using smaller memories.

Figure 7.24 shows an example of a 256K×8-bit memory using four 64K×8-bit memories. The basic idea is each of four memories represents a sub-address space of the entire memory address space of 64K by 8-bit words. The key then is the use of *Chip Select* ($\overline{CS}$) available on each memory chip to activate only a certain part of the memory of interest. This is done by having feeding the least significant 16 bits of the address lines (i.e., $A_{15} - A_0$) in to all the memories, and then have two most significant bits (i.e., A_{17} and A_{16}) become the input for a 2-to-4 decoder that selects one of the four memory chips. This way, the lower 16 bits of the address lines select the same

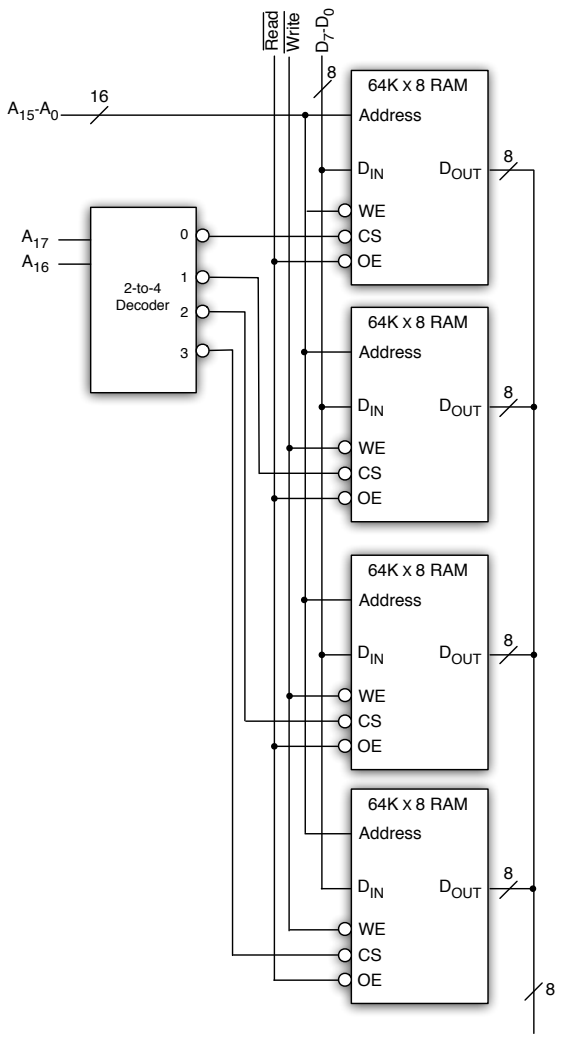


Figure 7.24: Implementing 256K×8-bit RAM using 64K×8-bit RAMs.

word for each of the four memories, but only one memory chip is enabled.

Figure 7.25 shows an example of a 256K×16-bit memory using two 256K×8-bit memories. All the address lines (i.e., $A_{17}, A_{16}, \cdots, A_0$) is fed to both of the memories. The upper byte of the data ($D_{15} - D_8$) is fed into the data input of the left memory, while the lower byte of the data ($D_7 - D_0$) is fed into the data input of the right memory.

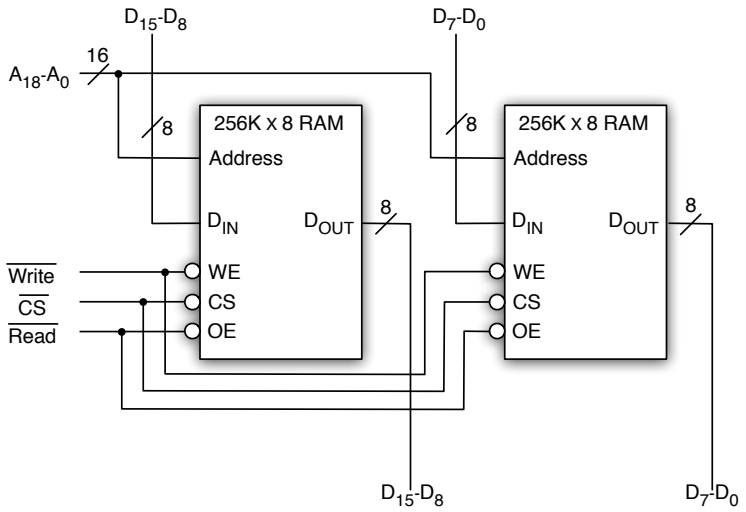


Figure 7.25: Implementing 256K×16-bit RAM using 256K×8-bit RAMs.

7.6 Register File

A *register file* consists of set of registers that are used to stage data between memory and Arithmetic Logic Unit (ALU) on the microarchitecture. An instruction set architecture (ISA) of a processor defines a set of registers, referred to as *General Purpose Registers* (GPRs), which can be specified directly within the instruction to define operands to be involved in the operation. A register file can simply be thought of as a small, fast memory. However, a register file in a microarchitecture are designed to be *multiple-read port, multiple-write port*, which allows one or more words to be read and one or more words to be written. For example, in the AVR processor, address registers `X`, `Y`, and `Z` each consists of a pair GPRs that need to be either read or written at the same time.

Figure 7.26 shows the internal structure of the 128-entry, two read-port, two write-port Register File. The read operation is performed by providing 7-bit *register identifiers* rA and rB, which controls MUX A and MUX B, to read the two registers at the same time. The write operation is bit more involved and requires three inputs: 7-bit register identifiers wA and wB, data inA and inB, and write signals RF_wA and RF_wB. Decoders A and B assert only one of their outputs based on wA and wB. These signals control whether inA or inB is applied to the input of each register entry

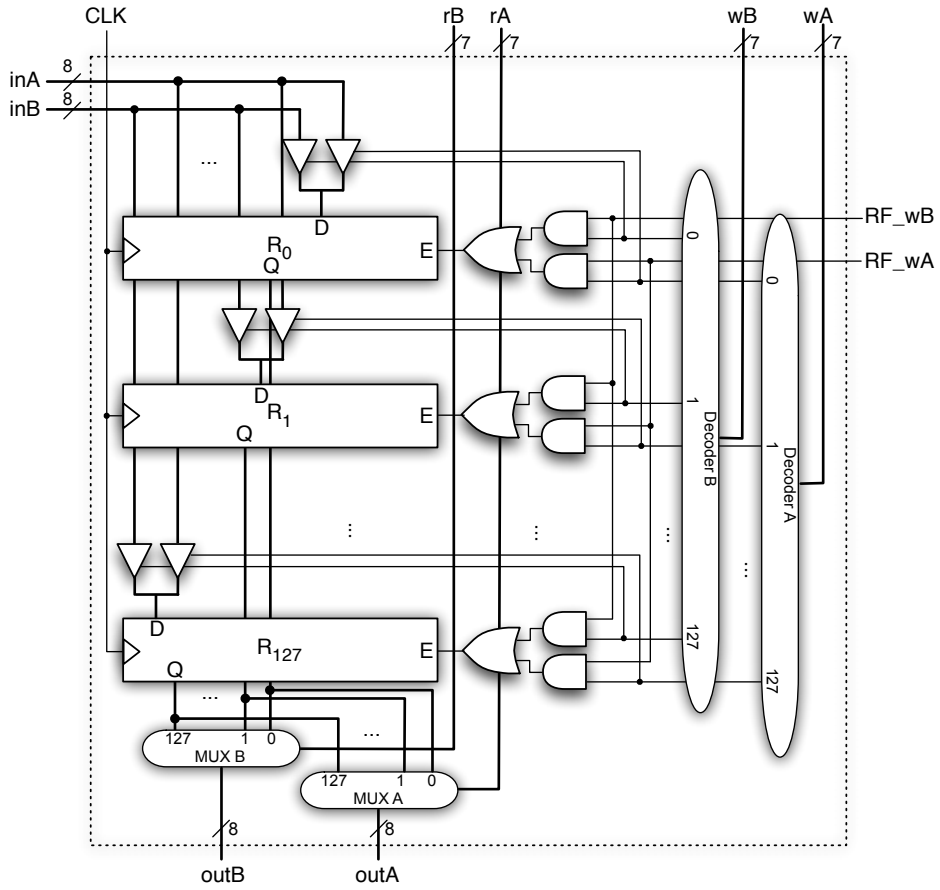


Figure 7.26: Internal structure of the 2 read-port, 2 write-port register file.

using tri-state buffers. Note that both tri-state buffers cannot be enabled at the same time. This situation does not occur in AVR because the pair of registers being accessed will never the same. Finally, the signals asserted from the decoders together with RF_wA and RF_wB determine whether or not registers latch their inputs.

7.7 Arithmetic and Logic Unit and Address Adder

Address Adder is a dedicated adder for generating the target address for PC-relative branches, direct jumps, and indirect jumps. ALU is the workhorse

of the microarchitecture. The design of an ALU, which is based on an adder, is a huge topic and require a entire chapter on its own. A detailed discussion on the ALU design is provided in Chapter 9.

Chapter 8

Microarchitecture Design

8.1 Microarchitecture

The term *Microarchitecture* (sometimes abbreviated to μarch) refers to the way a given instruction set architecture (ISA) is implemented in a processor. Another related terminology is *Computer architecture*, which is the combination of microarchitecture and instruction set design. A microarchitecture consists of a datapath and a control unit. A *datapath* is a collection of basic digital components (see Chapter 7), such as registers, memories, Arithmetic and Logic Unit (ALU), multiplexers, etc., that are interconnected by buses to perform data processing operations. A *control unit* (CU), on the other hand, is a logic that determines the sequence of data processing or data transfer operations to be performed by the datapath. The instruction set architecture defines many aspects of how a microarchitecture is implemented. However, a given instruction set architecture may be implemented with different microarchitectures due to different goals of a given design or shifts in technology.

In this chapter, we will discuss a microarchitecture design based on the AVR instruction set architecture. This implementation, referred to as *pseudo-AVR microarchitecture*, will give us an opportunity to understand the basic components needed to implement a datapath to perform data processing operations and design a control unit to provide appropriate signals to the various components to control the operations within the datapath.

You may be wondering why you should learn about microarchitecture design when most of you will never design your own processors. There are several reasons why the concepts presented in this chapter are important. First, there is intellectual merit in learning about a piece of technology that

revolutionized the way we live, work, and play, and yet most of us know very little about. Second, understanding the inner workings of a processor allows us to better understand how the overall computer system works. For example, few people may design processors but many people design hardware systems that contain processors. Third, knowing how processors work makes you a better programmer. Finally, some of you may design processors. Typically, the development of a new processor requires a large number of people working on different aspects of a processor design, including validation and verification, and compiler design. And there is a good chance that you will be involved in some part of this intricate process.

8.2 Instruction Format

Before discussing the details of the datapath design, we need to first understand the different AVR instruction formats and the information that needs to be extracted or decoded. Figure 8.1 shows the seven most common AVR instruction formats. The symbol '-' represents an opcode bit, which will be different for different instructions. We will consider these opcode bits during the design of a control unit in Sec. 8.6. There are also several other special instruction formats, which will not be discussed here.

Figure 8.1(a) shows the *two-operand format*, where the 5-bit register identifiers d dddd (Rd) and r rrrr (Rr) specify the left (or first) and right (or second) source operands, respectively. Since this is a 2-address instruction format (see Chapter 2), the register identifier d dddd also specifies the destination register. This format is used to encode any binary operations involving two registers, e.g., `ADD`, `CP`, `MOV`, `AND`, etc.

Instead of having both source operands in registers, one of the operand can be an immediate or a constant value encoded in the instruction format. This type of format, referred to as the *immediate format*, is shown in Figure 8.1(b). The 8-bit KKKK KKKK (K) field specifies an 8-bit unsigned value as the second operand. The dddd field has an implied '1' appended to the left of the most significant bit, and thus is equivalent to 1dddd. This means that only the upper 16 registers (R31-R16) can be specified as the destination. Example of instructions that follow this format are `LDI`, `ANDI`, `ORI`, etc.

Figure 8.1(c) shows the *I/O format*, where the 6-bit I/O register identifier AA AAAA (A) field specifies one of 64 I/O registers. There are two instructions that used this format, `IN` and `OUT`. The 5-bit d dddd field specifies the source register for the `OUT` instruction and the destination register for the `IN` instruction.

----	--rd	dddd	rrrr

(a) Two-operand format.

----	KKKK	dddd	KKKK

(b) Immediate format.

----	-AAd	dddd	AAAA

(c) I/O format.

----	---d	dddd	----

(d) One operand format.

--q-	qq-d	dddd	-qqq

(e) Displacement format.

----	kkkk	kkkk	kkkk
----	--kk	kkkk	k---

(f) PC relative format.

----	---k	kkkk	---k
kkkk	kkkk	kkkk	kkkk

(g) Direct format.

Figure 8.1: AVR instruction formats.

Unary operations require only one source register. These instructions include `INC`, `DEC`, `COM`, `LSR`, `ROR`, etc. The 5-bit `d dddd` (`Rd`) field specifies any one of the 32 general purpose registers (GPRs).

Figure 8.1(e) shows the *displacement format*, which is used in conjunction with load and store instructions. The two instructions that use the displacement format are `LDD` and `STD`. The 6-bit `q qq qqq` (`q`) field represents an unsigned offset ($0 \leq$ `q` ≤ 63), which is added to one of the *address registers* (ARs) `X`, `Y`, or `Z`. The 5-bit `d dddd` (`Rd`) field specifies any one of the 32 general purpose registers and can serve as either the source or destination register depending on whether the instruction is `STD` or `LDD`, respectively.

There are two *PC-relative formats*, 12-bit and 7-bit. This is shown in Figure 8.1(f). The signed 12-bit or 7-bit displacement field `kkkk kkkk kkkk` or `kk kkkk k` (`k`) is added to `PC`+1 to generate the branch target address with a range of -1024 $\leq$ `k` $\leq$ 1023 with the 12-bit displacement or -64 $\leq$ `k` $\leq$ 63 with the 7-bit displacement. The 12-bit PC-relative format is used by `RJMP` and `RCALL` instructions. On the other hand, the 7-bit PC-relative

format is used exclusively by conditional branch instructions, such as `BREQ`, `BRLT`, `BRGE`, etc.

Figure 8.1(g) shows the *direct format*, which is used only by the 32-bit instructions `CALL` and `JMP`. The 16-bit target address field `kkkk kkkk kkkk kkkk` (`k16`), which is the second 16-bit of the 32-bit instruction, allows the PC to point anywhere within the 64K Program Memory address space. The additional 6-bit `k kkkk k` field in the first 16 bits of the instruction format allows the Program Memory address space to be expanded by a factor of 2^6. For AVR processors with only 64K of Program Memory, the upper 6 bits are all zeros.

8.3 Basic Components in a Datapath

Figure 8.2 shows the basic 2-stage datapth for the pseudo-AVR microarchitecture consisting of *Fetch* and *Execute* stages. The Fetch stage is responsible for reading the instruction to be executed from the Program Memory. The Execute stage decodes the fetched instruction and performs operations defined by the instruction. The basic datapath shown in Figure 8.2 executes instructions that require either one or two cycles in the Execute stage depending on their complexity. Section 8.5 will discuss an enhanced datapath that can execute more complex instructions.

8.3.1 Registers

There are several *special-purpose registers* that serve as buffers for the Fetch and Execute stages. They are

- Fetch stage
  - Program Counter (PC)
- Execute stage
  - Instruction Register (IR)
  - Next PC (NPC)
  - Data Memory Address Register (DMAR)

The *Program Counter* (PC) register, also referred to as the *Instruction Pointer*, contains the address of the instruction in Program Memory to be fetched and executed. The PC is incremented (`PC+1`) during the Fetch stage and re-latched at the end of the clock cycle so that instructions are

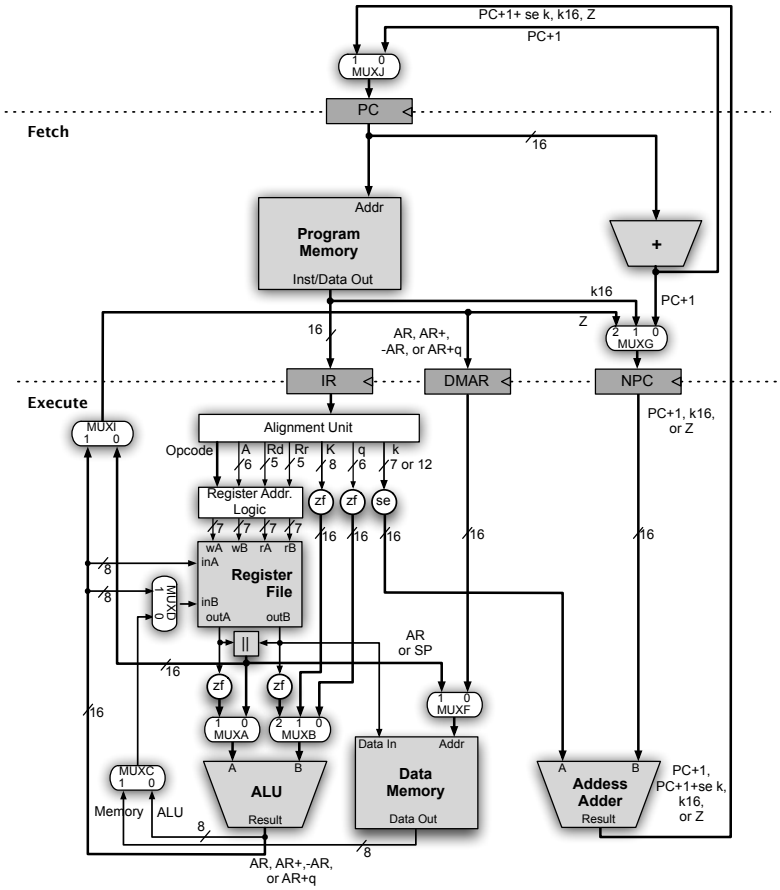
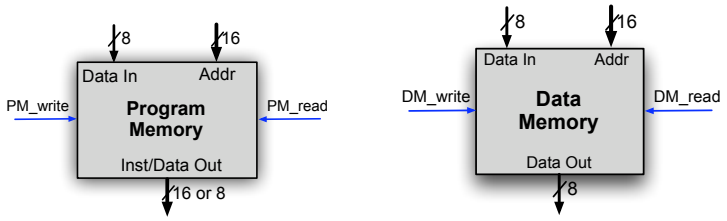


Figure 8.2: Basic 2-stage microarchitecture

fetched sequentially from the Program Memory. Certain instructions, such as branches, jumps, and subroutine calls and returns, interrupt the sequencing by updating the PC with a new address.

The *Instruction Register* (IR) holds an instruction fetched during the Fetch stage. Since some instructions require multiple Execute cycles, the IR latches an instruction at the end of the Fetch stage and holds its information during multiple Execute cycles.

The *Next PC* (NPC) register holds either PC+1 to be added with a displacement k to implement PC-relative branches or k16 to implement direct jumps.



(a) Program Memory. (b) Data Memory.

Figure 8.3: Program and Data Memories.

The *Data Memory Address Register* (DMAR) holds addresses to be used to access the Data Memory, which can come from `X`-, `Y`-, and `Z`-register as well as the *Stack Pointer* (SP) register.

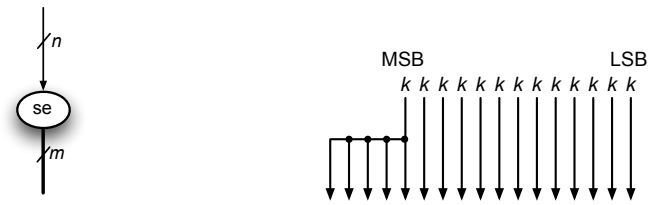
8.3.2 Program and Data Memories

Figure 8.3 shows the Program Memory and Data Memory. The operations of the two memories are fundamentally similar. The main difference is that the Program Memory holds 16-bit instructions, whereas the Data Memory holds 8-bit data.

8.3.3 Sign-Extension and Zero-Fill Units

Sign-extension is needed whenever a number of bits used to represent a signed data, e.g., PC-relative displacement `kk kkkk k`, needs to match the number of bits required by a component, e.g., 16-bit Adder shown in Figure 8.2. Figure 8.4 shows a sign-extension (`se`) unit. As shown in Figure 8.4(a), Sign-extension unit takes an n-bit number as input and simply duplicates the sign (i.e., MSB) and generates a m-bit number. This allows an n-bit signed number to become a m-bit signed number. For example, 0010111, which represents 23 in decimal, sign-extended to 16 bits becomes 0000000000010111. The number 1101001 represents -23 in decimal, which when sign extended to 16 bits becomes 1111111111101001. Figure 8.4(b) shows the sign-extension requirement for 12-bit PC-relative displacement `k` (see Section 8.3.5 for an explanation of 12-bit PC-relative displacement `k`).

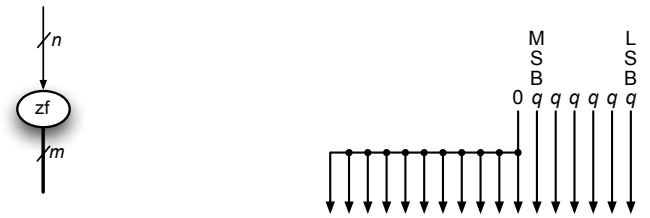
Similar to sign-extension, *zero-fill* is needed whenever a number of bits used to represent an unsigned data, e.g., load indirect with displacement `q qq qqq`, needs to match with the number of bits required by a component,



(a) n-bit to m-bit sign extension.

(b) 12-bit k sign-extended to 16-bit.

Figure 8.4: Sign-extension unit.



(a) n-bit to m-bit zero fill.

(b) 6-bit q zero-filled to16-bit.

Figure 8.5: Zero-fill unit.

e.g., 16-bit Address Adder shown in Figure 7.1. Figure 8.5 shows the zero-fill (zf) unit. As shown in Figure 8.5(a), zero-fill unit takes an n-bit number as input and simply pads zeros and generates a m-bit number. This allows an n-bit unsigned number to become a m-bit unsigned number. For example, 101001, which represents 41 in decimal, when zero-filled to 16 bits becomes 0000000000101001. Figure 8.5(b) shows the zero-fill requirement for 6-bit displacement q (see Section 8.3.5 for an explanation of 6-bit displacement q).

8.3.4 ALU

The 16-bit *Arithmetic and Logic Unit* (ALU) shown in Figure 8.6 is the workhorse of the microarchitecture. It takes two 16-bit inputs and performs an operation defined by the control signals ALU_f. Table 8.1 shows all the operations provided by the ALU.

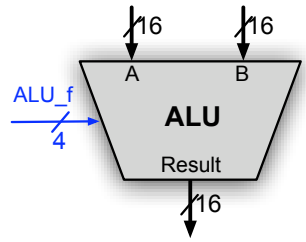


Figure 8.6: Arithmetic and logic Unit (ALU).

As will be seen in Section 8.4.2, the ALU handles arithmetic and logic operations on 8-bit registers as well as addition operations on 16-bit address registers concatenated from two 8-bit registers. In order to handle 8-bit arithmetic and logic operations, operands read from the Register File are first zero-filled to generate 16-bit operands, and then after the operation the results are truncated to 8 bits. Note that unary operations, such as negate, increment, and decrement, require only one operand.

A detailed discussion on the design of an ALU is provided in Chapter 9.

Table 8.1: Arithmetic and Logic Operations for the 8-bit ALU

Operation	Description	ALU_f
Arithmetic Operations		
$Result = A + B$	Add	0000
$Result = A + B + C$	Add with carry (C)	0001
$Result = A + \bar{B} + 1$	Subtract	0010
$Result = \bar{B} + 1$	Negate	0011
$Result = A + 1$	Increment	0100
$Result = A - 1$	Decrement	0101
$Result = A$	Move A	0110
$Result = B$	Move B	0111
Logic Operations		
$result = A \wedge B$	AND	1000
$result = A \vee B$	OR	1001
$result = A \oplus B$	Exclusive-OR (EOR)	1010
$result = \bar{B}$	Complement	1011
$result = B - 1$	Decrement	1100

8.3.5 Alignment Unit

Figure 8.7 shows the *Alignment Unit*, which extracts the various fields needed by the Execute stage from different instruction formats. These fields consist of

- Rr - 5-bit source register identifier indicated by bits r rrrr in the instruction format.
- Rd - 5-bit destination register identifier indicated by either bits d dddd or dddd in the instruction format. In the case of dddd, MSB is assumed to be 1 and thus equivalent to 1 dddd.
- K - 8-bit constant value indicated by bits KKKK KKKK in the instruction format.
- q - 6-bit displacement indicated by bits q qq qqq in the instruction format.
- k - Either 7-bit or 12-bit PC-relative displacement indicated by bits kk kkkk k or kkkk kkkk kkkk, respectively, in the instruction format.
- k16 - 16-bit target address indicated by bits kkkk kkkk kkkk kkkk in the instruction format.
- A - 6-bit I/O register address indicated by bits AA AAAA in the instruction format.

The fields A, Rd, and Rr are fed to the *Register Address Logic* to appropriately read/write from/to the register file. The K field provides an immediate value as the second operand to the ALU. The q and k fields are fed to the Address Adder either as a displacement to one of the address registers or PC+1.

8.3.6 Register File

In AVR, 32 GPRs and 64 I/O registers are mapped to the first 96 memory locations in the Data Memory, and can be accessed using load and store instructions. These registers are also implemented as a 96-entry *two read-port, two write-port Register File* (RF) shown in Figure 8.8.

As the name suggests, this register file allows two registers to be either read or written at the same time (see Chapter 7.6). This way, both source operands specified by Rd and Rr can be read at the same time and fed to the

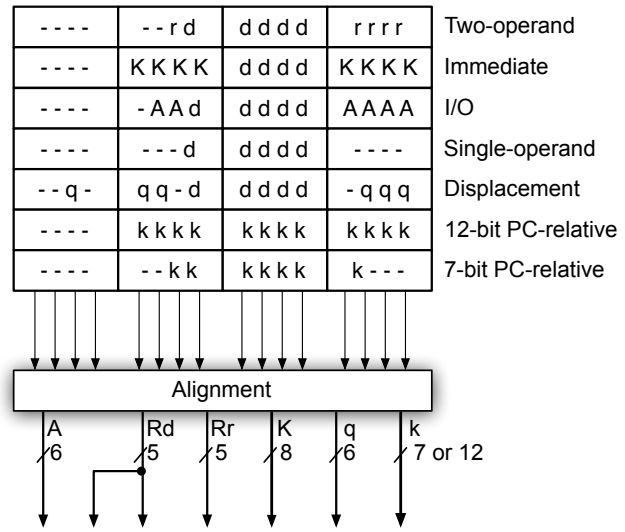


Figure 8.7: Alignment Unit.

ALU, and then the result can be written back to the lower write port (inB) of the Register File based on the destination register identifier Rd provided on wB. Furthermore, both upper and lower bytes of X-, Y-, and Z-register as well as the Stack Pointer (SP) can be read simultaneously and concatenated to generate 16-bit addresses. Each pair of registers can also be operated on by the ALU and simultaneously written back to the Register File.

Reading from the Register File is achieved by providing 7-bit source register identifiers to rA and rB and the corresponding register contents become available on outA and outB. Writing an 8-bit result to the Register File occurs by providing 7-bit destination register identifier to wB and the data to be written to inB, and asserting the write control signal (RF_write). Writing a 16-bit result to the Register File occurs by providing a pair of 7-bit destination register identifiers Rd+1 and Rd to wA and wB, respectively, and the upper and lower bytes of the data to inA and inB, respectively, and asserting RF_write.

8.3.7 Address Adder

Figure 8.9 shows the 16-bit *Address Adder*, which is a dedicated adder for generating the target addresses for PC-relative branches, direct jumps, and

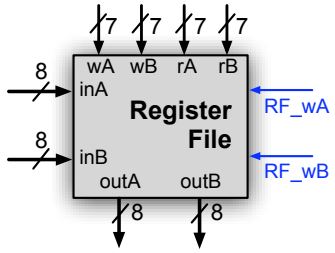


Figure 8.8: 2 read-port, 2 write-port Register File.

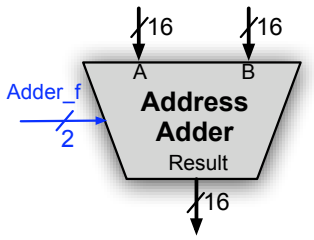


Figure 8.9: 16-bit Address Adder.

indirect jumps. Table 8.2 shows all the operations performed by the Address Adder.

Table 8.2: Operations for the 16-bit Address Adder.

Operation	Description	Adder_f
$Result = A + B$	Add	00
$Result = B + 1$	Increment	01
$Result = B - 1$	Decrement	10
$Result = B$	Move B	11

There are several possible address calculations. A move operation (i.e., Move B) simply passes the address on input B to the output. This operation does not require a second (left) input A to the Address Adder (see Section 9). Binary operations on addresses consist of PC+1 added with 7-bit or 12-bit displacement (`k`) (i.e., PC-relative branches and jumps) and an address register added with 6-bit displacement (`q`) (e.g., `LDD` and `STD`).

8.3.8 Multiplexers

As discussed in Section 7.1, a *Multiplexer* allows one of its multiple inputs to be selected onto its output. There are eight multiplexers in the basic datapath, and together they control how data transfer opertions are performed within the datapath.

MUXA is a 16-bit 2-to-1 multiplexer that selects either an 8-bit register content from the outA port of the register file that is zero-filled to 16 bits or a 16-bit address generated from concatenating two registers, and directs it to input A of the ALU. *MUXB* is a 16-bit 3-to-1 multiplexer that chooses among a register content from the outB port of the Register File, 8-bit constant K, and 6-bit displacement q, all zero-filled to 16 bits and directs it to input B of the ALU. These two multiplexers allow arithmetic and logic operations to be performed with Rd and either Rr (e.g., `ADD Rd, Rr`) or an immediate value (e.g., `ORI Rd, 0b00001111`). They also allow address registers `X`, `Y` and `Z` to be either incremented/decremented or added with a displacement q (e.g., `LDD Rd, Y+3`).

MUXC together with *MUXD* allow either the lower byte of the ALU output or 8-bit data from the Data Memory to appear on the lower write port (inB) of the Register File. The value can then be written to the Register File by providing the register identifier Rd on rB and asserting RF_wB. In addition, the upper byte of the ALU output for 16-bit operations can be written at the same time based on the register identifier Rd+1 on rA and asserting the RF_wB signal.

MUXF provides addresses for Data Memory accesses. MUXF is a 16-bit 2-to-1 multiplexer that chooses either a 16-bit address generated from concatenating two registers, i.e., Address Registers (AR), or an address stored in DMAR.

MUXG is a 16-bit 3-to-1 multiplexer that selects from different addresses (i.e., PC+1, k16, or Address Registers), which will be either added with a displacement or simply passed through the Address Adder to generate a target address.

MUXI is a 16-bit 2-to-1 multiplexer that selects either AR concatenated from a pair of 8-bit registers or AR concatenated from a pair of 8-bit registers and then modified by the ALU (-AR, AR+, or AR+q). These addresses are latched to both DMAR and NPC, and are either used to access the Data Memory or as target addresses for indirect jumps.

MUXJ chooses between PC+1 and 16-bit target address generated from the Address Adder.

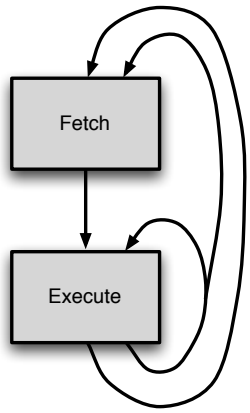


Figure 8.10: Fetch and Execute stages.

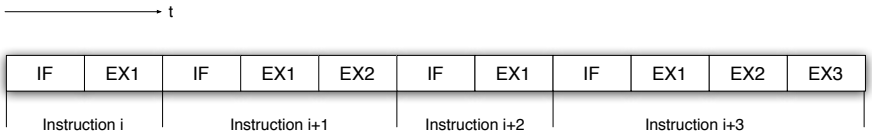


Figure 8.11: Multi-cycle implementation.

8.4 Multi-cycle Implementation

Figure 8.10 shows the general operations of the Fetch and Execute stages. The Fetch stage fetches an instruction from the Program Memory and forwards it to the Execute stage. The Execute stage decodes the instruction and spends one or more cycles in this stage to execute the instruction. Depending on the type of instructions, the number cycles required in the Execute stage can be anywhere between one to three cycles. Some instructions even require part of their execute cycles to be performed in the Fetch stage.

Our discussion of the Fetch and Execute stages in this section is based on a *multi-cycle implementation* as shown in Figure 8.11, where the instruction cycle, each consisting of a Fetch and one or more Execute cycles, are executed sequentially. The discussion of a *pipeline implementation*, where the instruction cycles are partially overlapped, will be discussed in Section 8.7.

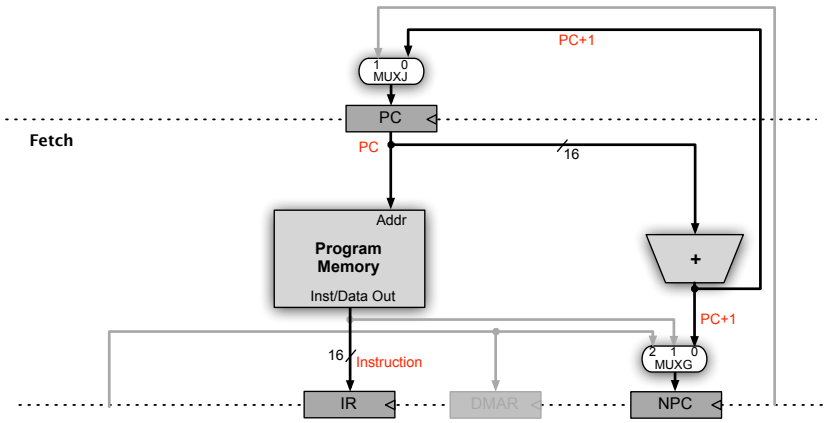


Figure 8.12: Fetch cycle.

8.4.1 Fetch Stage

In the Fetch cycle, the current instruction pointed to by PC is fetched from the Program Memory and latched on the Instruction Register (IR). In addition, PC is incremented (i.e., PC+1) and re-latched onto PC as well as latched onto the next PC (NPC) register. PC+1 points to either the next instruction or a target address for 32-bit direct jump (`JMP k`) and subroutine call (`CALL k`) instructions. On the other hand, NPC is used by the Execute stage to determine target addresses for PC-relative jump, branch, and subroutine call instructions.

Table 8.3 formally defines the micro-operation performed by the Fetch (IF) stage. Figure 8.12 illustrates the data transfer operations in the Fetch stage. The data transfer operation IR←M[PC] indicates that the content of the memory location pointed to by PC, i.e., M[PC], is latched onto the IR at the end of the cycle. At this point, we will not distinguish between accessing Program Memory versus Data Memory. Using PC to access memory automatically implies Program Memory. In addition to fetching the current instruction, the data transfer operation PC←PC+1 indicates that the PC is incremented to point to either the next instruction to be fetched and executed or a target address for 32-bit instructions, and re-latched onto the PC register at the end of the cycle, which is achieved by selecting input 0 of MUXJ. At the same time, PC+1 is latched onto NPC by selecting input 0 of MUXG.

Table 8.3: Micro-operations for the Fetch Stage

Stage	Micro-operations
IF	IR ← M[PC], PC ← PC +1, NPC ← PC +1

8.4.2 Execute Stage

The micro-operations performed by the Execute stage depend on the instructions, which can be subdivided into the following three categories:

- Arithmetic and Logic Instructions
- Data Transfer Instructions
- Branch and Jump Instructions

Instructions in each category utilize similar parts of the datapath. The instructions discussed in this section require either one or two execute cycles. Section 8.5 will discuss more complex instructions requiring 2 or 3 execute cycles.

Arithmetic and Logic Instructions

There are three different formats for arithmetic and logic instructions:

- op Rd, Rr
- op Rd, K
- op Rd

where op represents an arithmetic or logic operation (see Table 8.1).

Figure 8.13 shows the portion of the datapath that performs arithmetic and logic operations. Table 8.4 shows the micro-operations for the Execute (EX) stage for arithmetic and logic instructions.

Table 8.4: Micro-operations for Arithmetic and Logic Instructions

Arithmetic and Logic Instructions			
Stage	Micro-operations		
	Binary	Immediate	Unary
EX	Rd ← Rd *op* Rr	Rd ← Rd *op* K	Rd ← *op* Rd

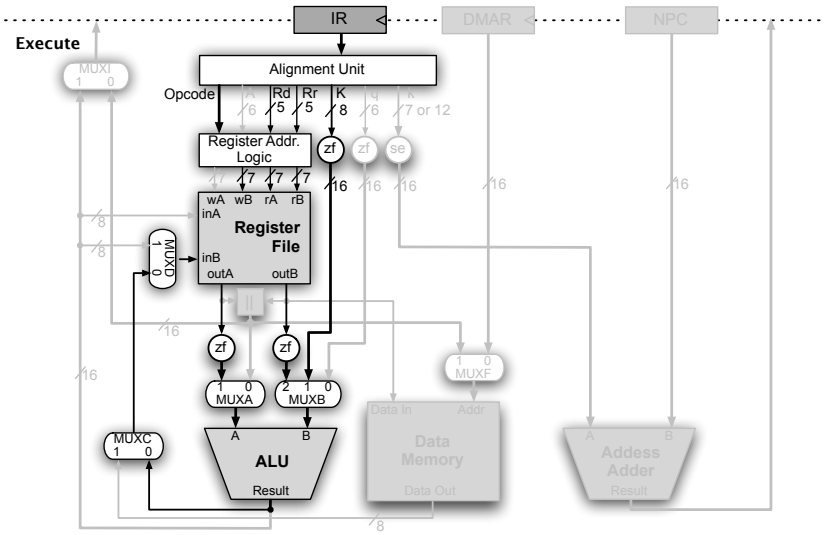


Figure 8.13: The portion of the basic datapath utilized by Arithmetic and Logic instructions.

For *binary operations* involving two registers, such as `ADD` and `SUB`, both source operands are obtained from the Register File (RF) based on register identifiers Rd and Rr. Operations involving immediate (or constant) values, such as `ORI` and `SUBI`, are provided from the 8-bit K-field in the instruction format. As can be seen from Figure 8.13, MUXB selects either a register or immediate value as the second operand. *Unary operations*, such as `INC`, `NEG`, `CLR`, etc., do not require a second source operand since they are taken care of within the ALU (see Chapter 9).

For all three formats, the ALU performs an operation based on op and the result becomes available on the lower write-port (i.e., inB) of the Register File by appropriately selecting the inputs for MUXC and MUXD, which then becomes latched onto the destination register based on Rd at the end of the clock cycle. Note that only the lower byte of the 16-bit result is written back to the Register File. With the exception of few instructions that involve 16-bit operands (i.e., `ADIW` and `SBIW`) and operations that generate 16-bit results (e.g., `MUL`), most of the arithmetic and logic instructions complete in a single Execute cycle (see Table A.1).

Data Transfer Instructions

Most data transfer instructions require either one or two execute cycles depending on their type and whether data is transfer between registers or between a register and the Data Memory. Table 8.5 summarizes the micro-operations for move and I/O instructions that require one execute cycle.

Table 8.5: Micro-operations for Move and I/O Instructions

Move and I/O Instructions				
Stage	Micro-operations			
	Move 8-bit	In	Out	Move 16-bit
EX	Rd ← Rr	Rd ← A	A ← Rr	Rd+1:Rd ← Rr+1:Rr

Instructions that require a single Execute cycle consist of

- `MOV Rd, Rr`
- `IN Rd, A`
- `OUT A, Rr`
- `MOVW Rd, Rr`

where A represents an I/O register in the 64 I/O register address space (see Figure 4.4).

Figure 8.14 shows the portion of the datapath that performs single-cycle 8-bit data transfers, i.e, `MOV`. Although the path is similar to those used by arithmetic and logic instructions (see Figure 8.13), only the content of one source register defined by Rr is passed through the ALU without modifying its content. This unaltered value becomes available on the lower write-port inB, which is then latched onto the destination register based on Rd at the end of the clock cycle.

In contrast to `MOV`, `MOVW` moves 16-bit data, which is generated by concatenating a pair of registers defined by Rr+1 and Rr. The part of the datapath utilized by `MOVW` is shown in Figure 8.15. The `MOVW` instruction only specifies Rd and Rr, from which the Register Address Logic automatically generates Rd+1 and Rr+1. Note that Rd+1 or Rr+1 indicates a register identifier, not the content of a register, is incremented by one. For example, when Rr is specified as R26, concatenated register pair R27:R26 is moved.

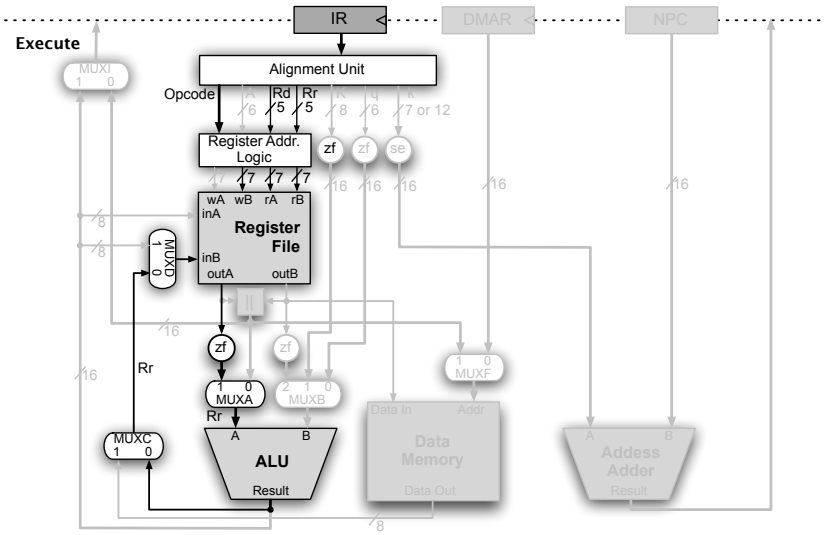


Figure 8.14: The portion of the datapath for 1-cycle 8-bit Data transfer.

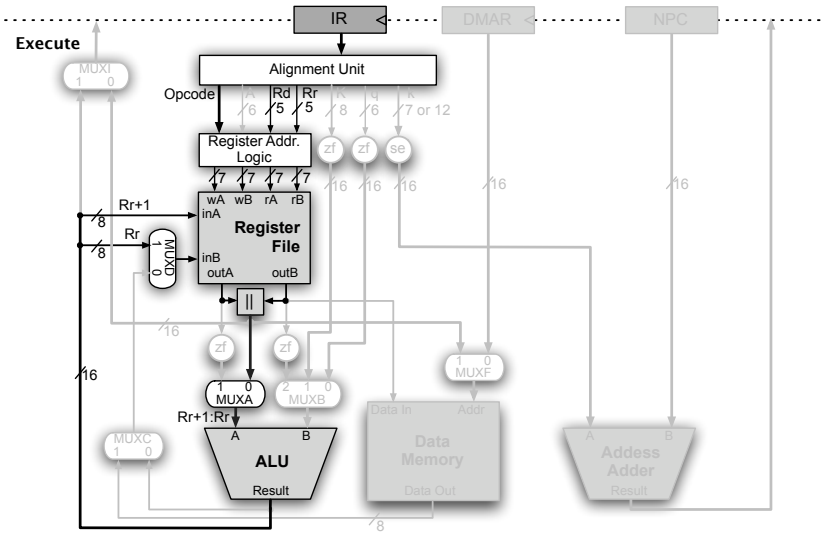
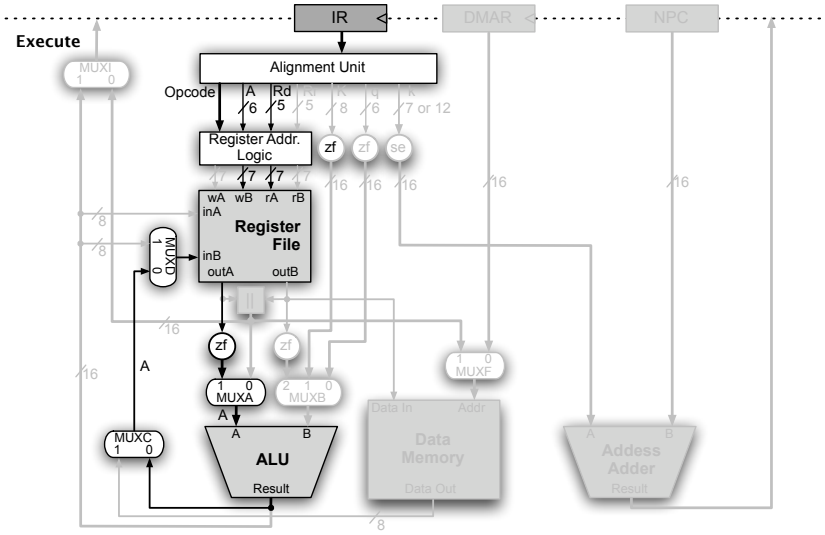
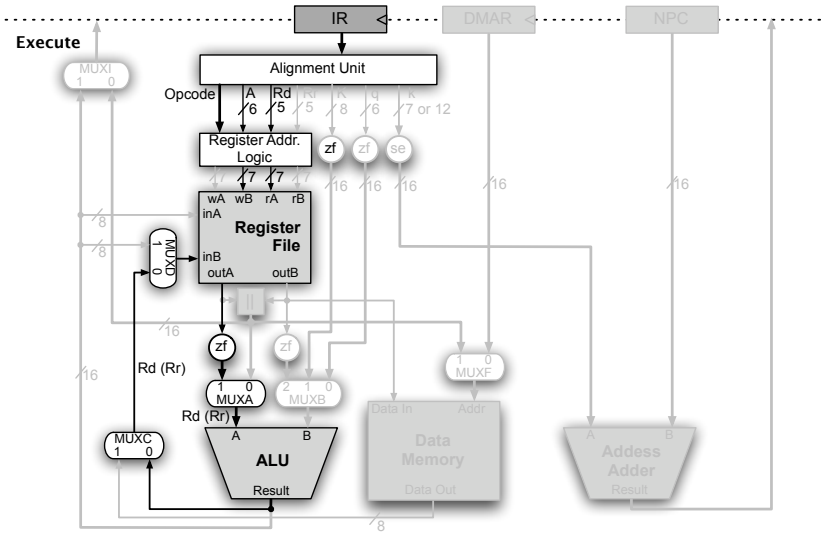


Figure 8.15: The portion of the datapath for 1-cycle 16-bit Data transfer.

The register pair Rr+1 and Rr are then written back to the Register File based on the destination register identifiers Rd+1 and Rd, respectively.



(a) IN.



(b) OUT.

Figure 8.16: Part of the datapath utilized by IN and OUT instructions.

IN and OUT instructions are similar to a single-cycle 8-bit transfer except that data transfers are performed between a GPR and an I/O register, which are both contained in the 96-entry Register File. Figures 8.16(a) and 8.16(b) show the part of the datapath utilized by IN and OUT instructions, respectively. For the IN instruction, 32 is added to 6-bit I/O register identifier A and then zero filled to generate a 7-bit address, which is then used to read the Register File. The value read from an I/O register is then routed and written to the Register File based on Rd. For the OUT instruction, the content of Rr is read from the Register File and written back to the Register File defined by A. Note that there is no separate register identifiers for Rd and Rr in the instruction format. Therefore, the Rd field defines the destination register for IN and the source register for OUT

Data transfer instructions that require two execute cycles are load and store instructions. Tables 8.6 shows the sequence of micro-operations for these instructions, where AR represents an address register X, Y, or Z.

Table 8.6: Micro-operations for Load and Store Instructions

Load and Store Instructions				
Stage	Micro-operations			
	Normal	Displacement	Pre-Decrement	Post-Increment
EX1	DMAR ← AR	DMAR ← AR+q	DMAR ← AR-1, AR ← AR -1	DMAR ← AR, AR ← AR +1
EX2	Loads		Stores	
	Rd ← M[DMAR]		M[DMAR] ← Rr	

In the first Execute cycle (EX1), which micro-operation is performed depends on the addressing mode. Figure 8.17 shows the operation for normal *register-indirect* loads and stores, e.g., LD Rd, X and ST X, Rr, where the high and low bytes of an address register (ARh and ARl), which could be X-, Y-, or Z-register, are read from the Register File and concatenated to generate a 16-bit address (i.e., ARh:ARl). This address moved through the ALU (without modifying the content) and is then latched onto DMAR, which will be used to access the Data Memory in the subsequent cycle(s). Note that the path from the output of the concatenate unit (||) to the 0-input of MUXI could also be used. However, the specific purpose of this path is to implement post-increment, which will be discussed shortly (see Figure 8.19(b)), and thus is not used for the normal register-indirect addressing mode.

Figure 8.18 shows EX1 of *register indirect with displacement*. The high (ARh) and low (ARl) bytes of an address register read from the Register File

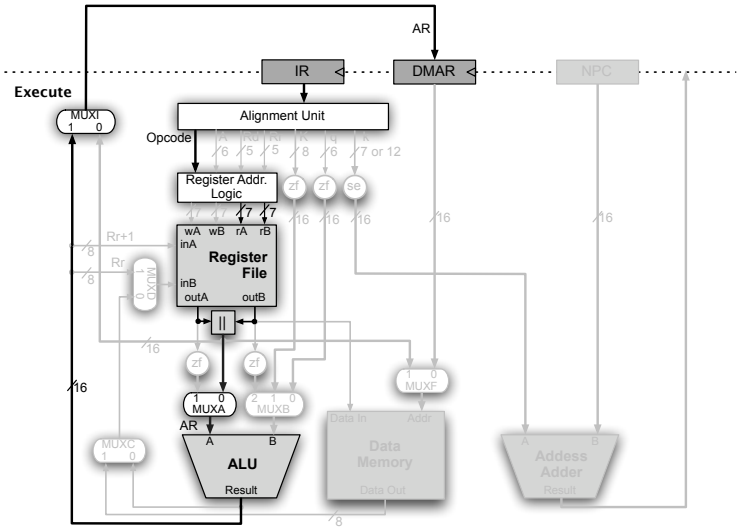


Figure 8.17: EX1 of register indirect for loads and stores.

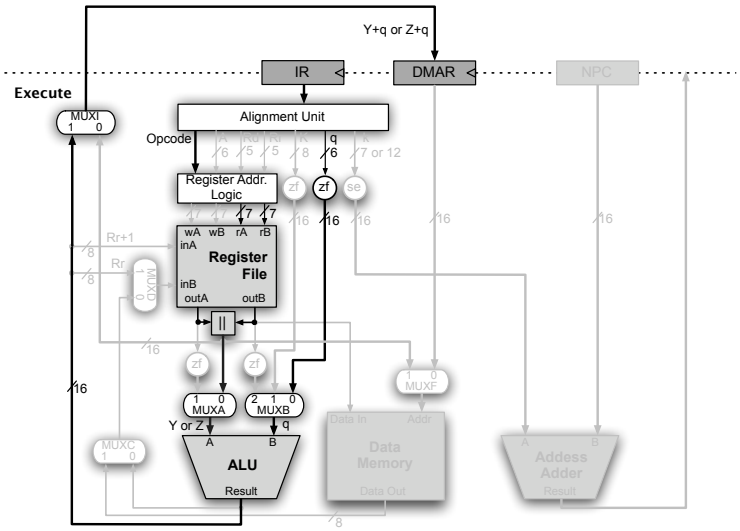


Figure 8.18: EX1 of register indirect with displacement for loads and stores.

are concatenated and added with zero-filled 6-bit displacement q. Then, AR+q is latched onto DMAR.

Figure 8.19 shows EX1 of register indirect with pre-decrement and post-increment. Figure 8.19(a) shows the data transfer operations for *register indirect with pre-decrement*. The high (ARh) and low (ARl) bytes of an address register read from the Register File are concatenated and then decremented using the ALU. Then, ARh:ARl-1 is latched onto DMAR, and at the same time written back to the Register File. Figure 8.19(b) shows the data transfer operations for *register indirect with post-decrement*. ARh and ARl read from the register file are latched on to DMAR using the path from the output of the concatenate unit (||) to the 0-input of MUXI . At the same time, AR is incremented using the ALU and written back to the register file.

In EX2, the operand is read from the Data Memory and latched onto Rd in the Register File for loads, and the operand specified by Rr is written to the Data Memory for stores. The data transfer operations for EX2 for load and store are shown in Figures 8.20(a) and 8.20(b), respectively.

Branch and Jump Instructions

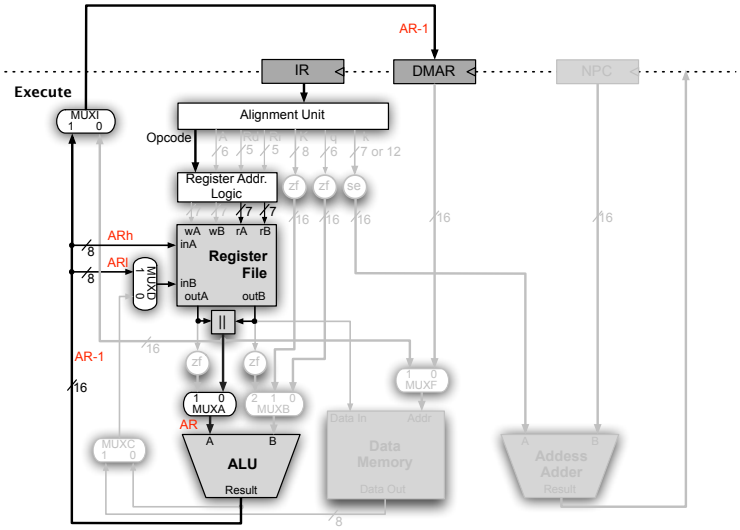
There are two types of control transfer instructions: conditional branches conditional branches and jumps jumps. A *conditional branch* evaluates a condition code or flag, e.g., Z, N, V, or S flag, and if the condition is true transfers the control flow to the target address. A *jump* instruction unconditionally transfers control flow to the target address, thus is also referred to as *unconditional branches*.

Table 8.7 shows the micro-operations for branch instructions.

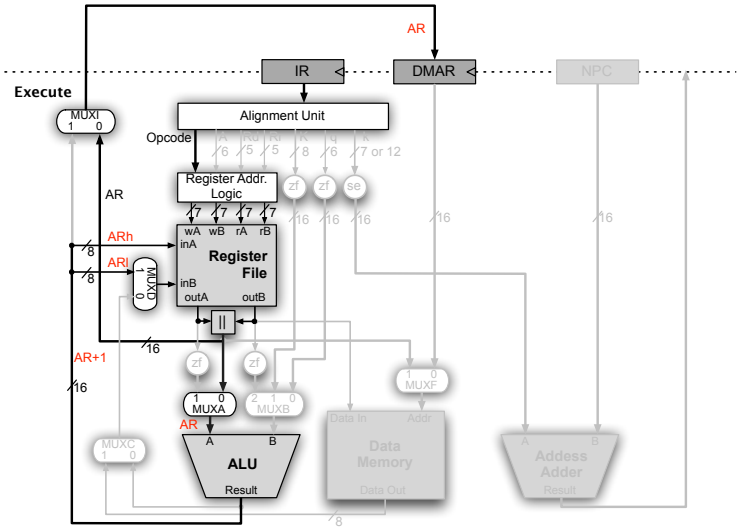
Table 8.7: Branch and Jump Instructions

Branch and Jump Instructions		
Stage	Micro-operations	
	PC-relative	Direct
EX1	If (*flag*) then PC←NPC + se k	NPC ← M[PC]
EX2		PC ← NPC

Figure 8.21 shows the portion of the datapath affected by PC-relative branch instructions. In EX1, either the 12-bit or 7-bit displacement k is sign-extended to 16-bits and added to the content of NPC, which is PC+1. The resulting address, known as *branch target address*, becomes available at the input of the PC register to be latched at the end of the clock cycle.

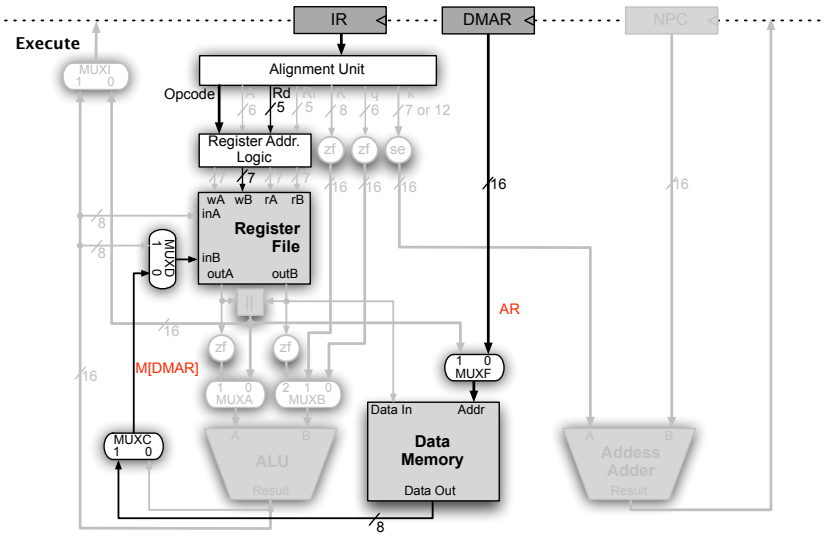


(a) EX1 for load/store with pre-decrement.

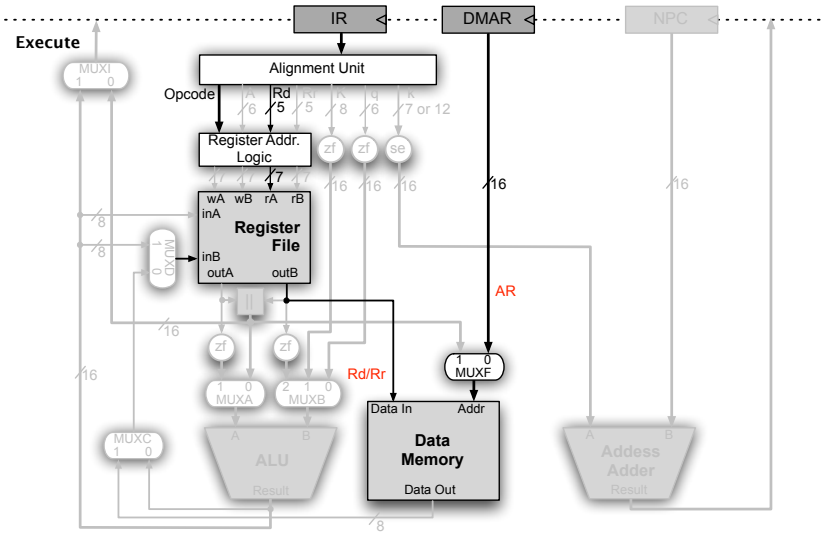


(b) EX1 for load/store with post-increment.

Figure 8.19: EX1 for register indirect with pre-decrement and post-increment for loads and stores.



(a) EX2 for loads.



(b) EX2 for stores.

Figure 8.20: EX2 for loads and stores.

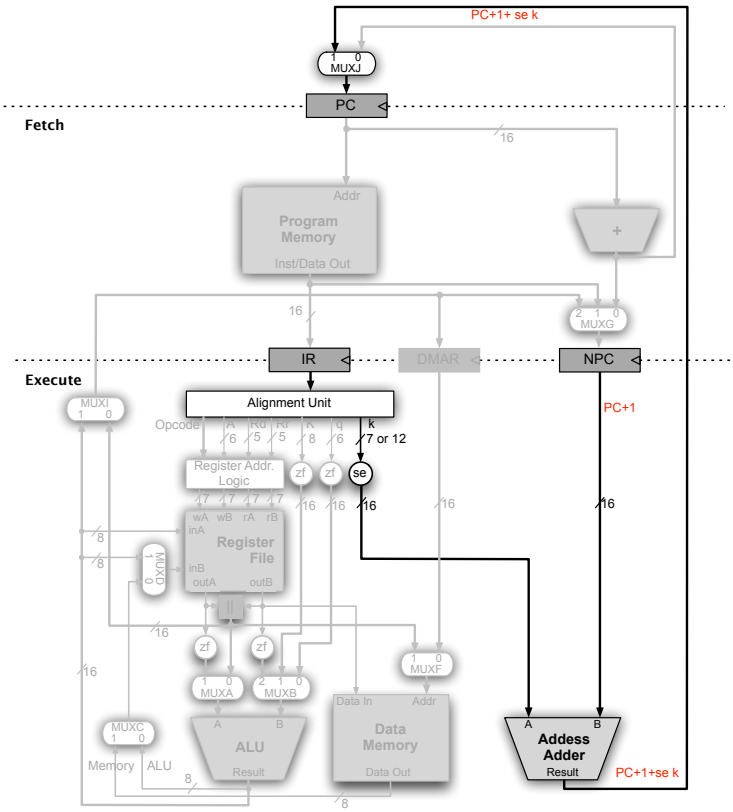
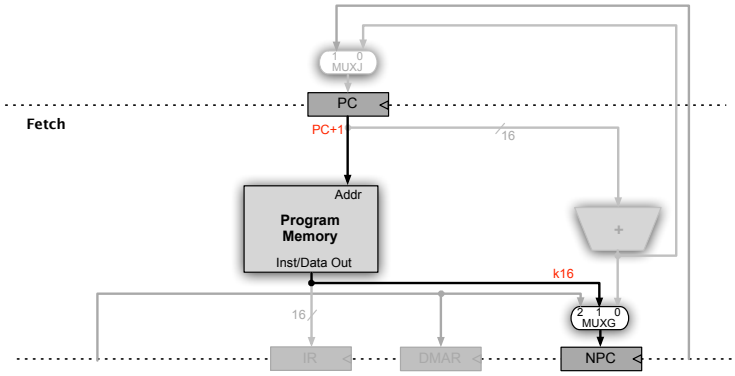


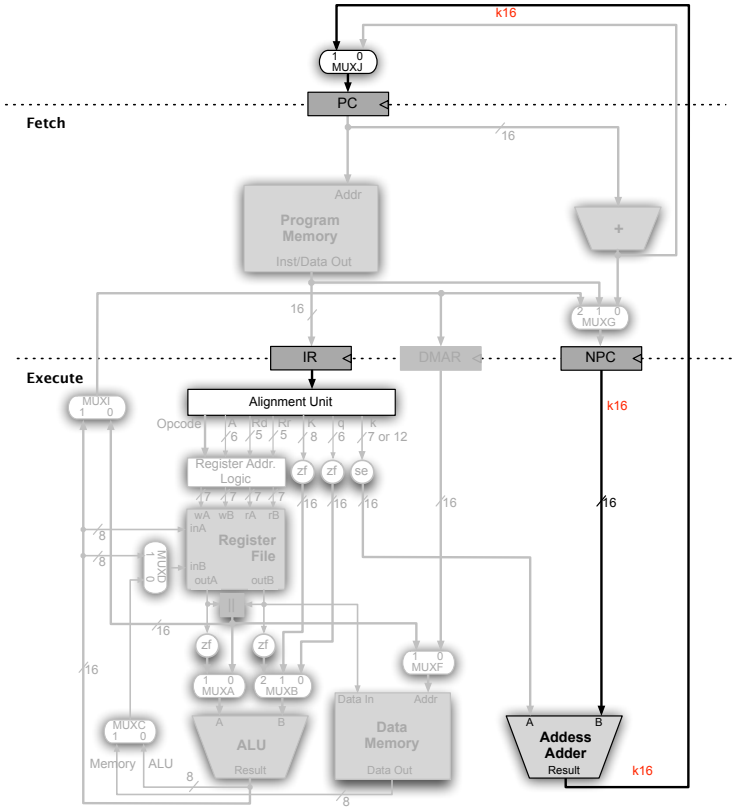
Figure 8.21: EX1 of PC-relative branch instruction.

Figure 8.22 shows the portion of the datapath affected by direct jump instructions, which consist of `JMP` and `CALL`. For now, we will concentrate on `JMP` since `CALL` involves more complex operations.

`JMP` is a 32-bit instruction where the second 16-bit of the instruction, `k16`, represents the *target address* of the jump. Therefore, unlike other 16-bit instructions, the second half of the instruction has to be fetched again from the Program Memory in the Fetch stage and latched onto the NPC register. This can be thought of as fetching the target address instead of fetching the next instruction. In EX2, the target address in NPC is made available to the input of the PC register via the Address Adder to be latched at the end of the clock cycle.



(a) EX1 for `JMP k`.



(b) EX2 for `JMP k`.

Figure 8.22: Micro-operations for direct jump instructions.

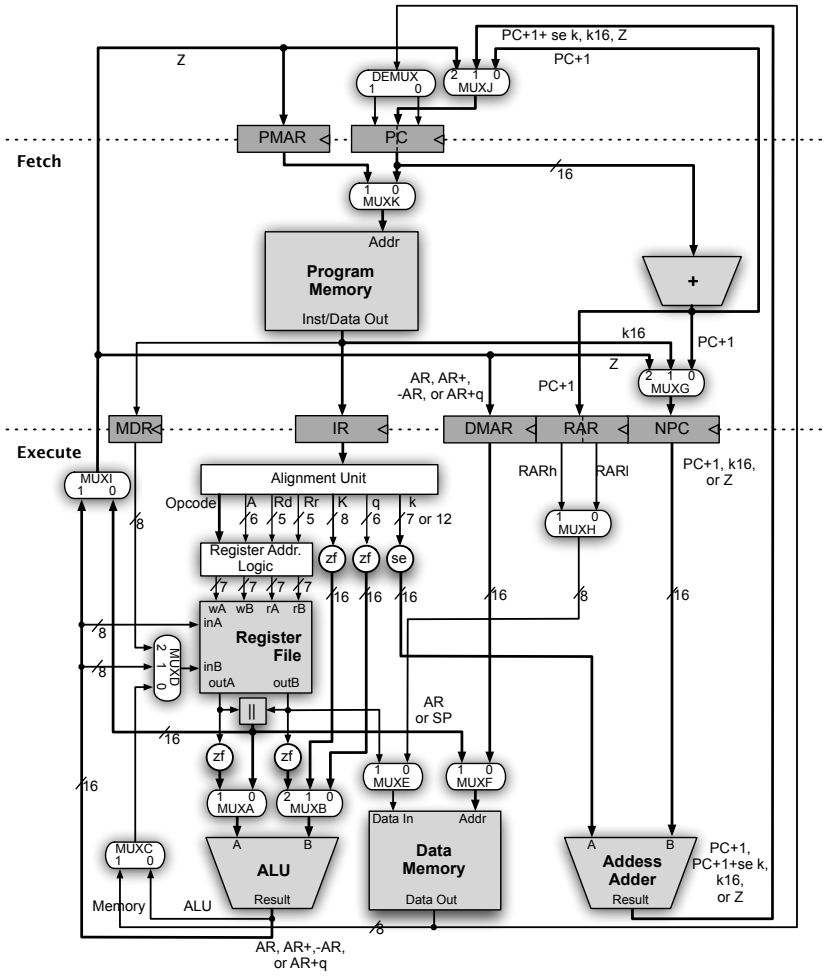


Figure 8.23: Enhanced 2-stage microarchitecture

8.5 Execution of More Complex Instructions

The basic datapath discussed thus far is capable of implementing most of the AVR instructions. However, some of the more complex AVR instructions require additional functionalities not available in the basic datapath. Figure 8.23 shows the *enhanced datapath* to handle more complex instructions.

In the Fetch stage, the main difference between the basic datapath shown in Figure 8.10 and the enhanced datapath is that the latter has three addi-

tional special-purpose registers.

The *Return Address Register* (RAR) latches the return address for a subroutine call, which is either PC+1 for relative and indirect subroutine calls and PC+2 for direct subroutine calls. You may wonder why we need both NPC and RAR when it appears both registers latch PC+1. The reason is that NPC can be either PC+1 (or PC+2 incase of direct jumps) or k16. If k16 is latched on to NPC for 32-bit direct subroutine call (i.e., `CALL k`), the return address of the subroutine call, i.e., PC+2, will be lost. Thus, RAR allows a return address of a direct subroutine call to be pushed onto the stack in the Execute stage. RAR consists of RARh and RARl representing the high and low bytes of the return address, respectively, which can be selected separately using MUXH. The write-port of the Data Memory (i.e., Data In) is preceded by MUXE to select between RARh/RARl and an operand from the Register File.

The *Program Memory Address Register* (PMAR) provides addresses for constants stored in the Program Memory. These addresses are stored in the `Z`-register, which is used exclusively by `LPM` (*Load Program Memory*) instructions. The *Memory Data Register* (MDR) latches constants accessed from the Program Memory, which together with MUXD expanded to a 3-to-1 multiplexer allow them to be written to GPRs in the Execute stage. Also, the PC is separated into PCh and PCl representing the high and low bytes of the PC, respectively. These registers can be separately updated with upper and lower bytes of a return address popped from the stack during a subroutine return.

MUXJ and MUXG are also expanded to accept `Z`-register as the target address for *indirect jumps* (e.g., `IJMP` and `ICALL`). The inclusion of DEMUX allows return addresses of subroutine calls to be popped from the stack and latched onto PC one byte at a time.

These enhancements allow complex instructions, such as stack operations (`PUSH` and `POP`), subroutine calls and return (`CALL`, `RCALL`, `ICALL`, and `RET`), and load program memory (`LPM`), to be implemented

The micro-operation for the Fetch stage for the enhanced datapath is shown in Table 8.8, which requires one additional data transfer operation of latching PC+1 onto RAR.

Table 8.8: Micro-operations for the Fetch Stage

Stage	Micro-operations
IF	IR ← M[PC], PC ← PC +1, NPC ← PC +1, RAR ← PC +1

In Section 8.6, we will discuss how one of these complex instructions, `CALL` (*Direct Subroutine Call*), can be implemented on the enhanced datapath.

8.6 Control Unit Design

As we saw in Section 8.1, a microarchitecture consist of two major components: a datapth and a control unit. The discussion up now now has been on the datapth, which allows manipulation of data based on micro-operations using data transfer operations. On the other hand, the *Control Unit* (CU) provides signals that activate various components within the datapath to perform the specified micro-operations. The Control Unit also generates signals for sequencing the set of micro-operations required to implement an instruction.

This section discusses the implementation of a control unit for a multicycle implementation of the pseudo AVR Microarchitecture. In order to illustrate the design of the control unit, we will based our discussion on a small sub-set of AVR instructions. Table 8.9 shows descriptions of the instructions covered by the Control Unit.

Table 8.9: AVR Instructions for Control Unit Design

Representative AVR Instructions for Control Unit Design				
Category	Mnemonics	Description	Operation	Flags
ALU	ADD Rd, Rr	Add two Registers	Rd← Rd + Rr	Z,C,N,V,H
	ORI Rd, K	Logical OR Register and Constant	Rd← Rd ∨ K	Z,C,N,V,H
Data Transfer	LD Rd, Y	Load Indirect	Rd ← M[Y]	None
	ST Y, Rr	Store Indirect	M[Y] ← Rr	None
Branch	BREQ k	Branch if Equal	if (Z=1) then PC←PC+1+k	None
	CALL k	Direct Subroutine call	PC←k, STACK←PC+1	None

8.6.1 Opcode Encoding

Figure 8.24 shows the encoding of opcodes for instructions in Table 8.9 (see Figure A.4 for a complete encoding of all the AVR instructions). There are four groups of instructions: Group A, Group B, Group C, and Group D. Group A consists of binary ALU operations involving two source registers and some of the data transfer instructions. Group B consists of ALU instructions with a source register and an immediate value. Group C encodes the largest number of instructions and consists of data transfer, control trans-

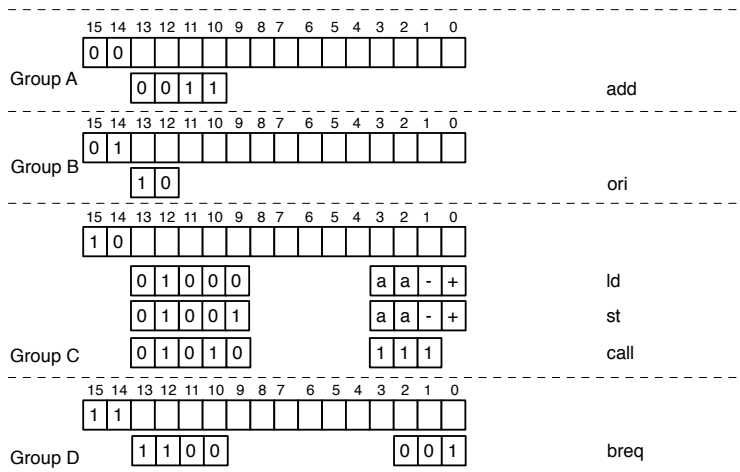


Figure 8.24: Opcode encoding for instructions in Table 8.9.

fer, bit test and set, as well as other miscellaneous instructions. Group D consists of conditional branch and relative jump and call instructions.

As can be seen from Figure 8.24, these four groups of instructions are identified by the two most significant bits (bits 15-14), i.e., 00 (Group A), 01 (Group B), 10 (Group C), and 11 (Group D). For Group A, the next 4 bits (bits 13-10) determine the different instructions within this group. For Group B, the next two bits (bits 13-12) determine the encoded instructions. Similarly, bits 13-12 for Group C determine the four different types of instructions within this group. Each type of instructions defined by 01, 10, and 11, and the next three bits (bits 11-9) further subdivides into different instructions. Note that for the instructions covered in Table 8.9, only 01 for bits 13-12 is listed. Furthermore, for some instructions, additional bits define different operations for similar instructions. For example, the `LDLD` (*Load indirect from data space to register* instruction has indirect addressing mode as well as indirect with pre-decrement and post-increment capabilities. These are defined by bit 1 and bit 0 indicated by "-" and "+", respectively, where 1 in these bit positions enable the pre-decrement and post-increment capability. Note that both bits 1 and 0 cannot be set at the same time. The same applies to Group D. Bits 13-12 define different types of instructions within this group, and then additional bits define each instruction. For example, all the conditional branch instructions are defined by 1100 and 1101 in bits 13-10, and bits 2-0 define one of the conditions.

8.6.2 Control and Alignment Unit

Figure 8.25 shows the *Control and Alignment Unit* (CAU), which is a combination of the Control Unit and the Alignment Unit discussed in Section 8.3.5. There are two parts to the design of CAU. First, a set of control signals need to be generated for each cycle or stage of an instructions execution. Second, the sequence control needs to be defined for the series of micro-operations required for the execution of instructions. These two requirements are met by implementing the CAU as a *finite state machine* (FSM).

Fetch Stage

The Fetch stage is the same for all instructions and is controlled by the following signals: MJ, PC_en, MK, PM_read, and PM_write. Since the discussion of the Fetch stage does not include writing to the Program Memory, PM_write signal will be ignored. In addition, the PM_read signal will also be ignored since the Program Memory will be read every cycle but its content will be latched onto the IR and NPC registers only when the IR_en and NPC_en signals, respectively, are asserted. For example, an instruction fetched from the Program Memory is latched onto IR only when IR_en is asserted (i.e., set to 1) together with the clock. All other registers, i.e., DMAR, RAR, and MDR, are simply latched with the clock. Figure 8.26 shows the control signals needed to fetch an instruction.

MUXK selects PC as its input and allows the current instruction pointed to by the PC to be read from the Program Memory. Asserting IR_en latches the instruction (i.e., M[PC]) onto IR at the end of the clock cycle. At the same time, PC is incremented by one to point to either the next instruction or the second 16-bit of a 32-bit instruction, which is then latched onto PC by selecting the 0-input of MUXJ and asserting PC_en. Note that there are three way to latch an address onto PC. The first method is to latch the entire 16-bit address using PC_en. The second method is to latch the upper byte of the PC using PCh_en. The last method is to latch the lower byte of the PC using PCl_en. The latter two methods are used together to latch on the return address of a subroutine call a byte at a time. Therefore, unless the instruction is `RET` (*Return from Subroutine*), both PCh_en and PCl_en are not asserted (i.e., set to 0s). Moreover, only one of the three control signals (PC_en, PCh_en, or PCl_en) can be asserted at a time.

The IR is enabled only during the Fetch cycle since the fetched instruction dictates the operations to be performed during one or more Execute cycles. The PC is enabled during every Fetch cycle, and during some Ex-

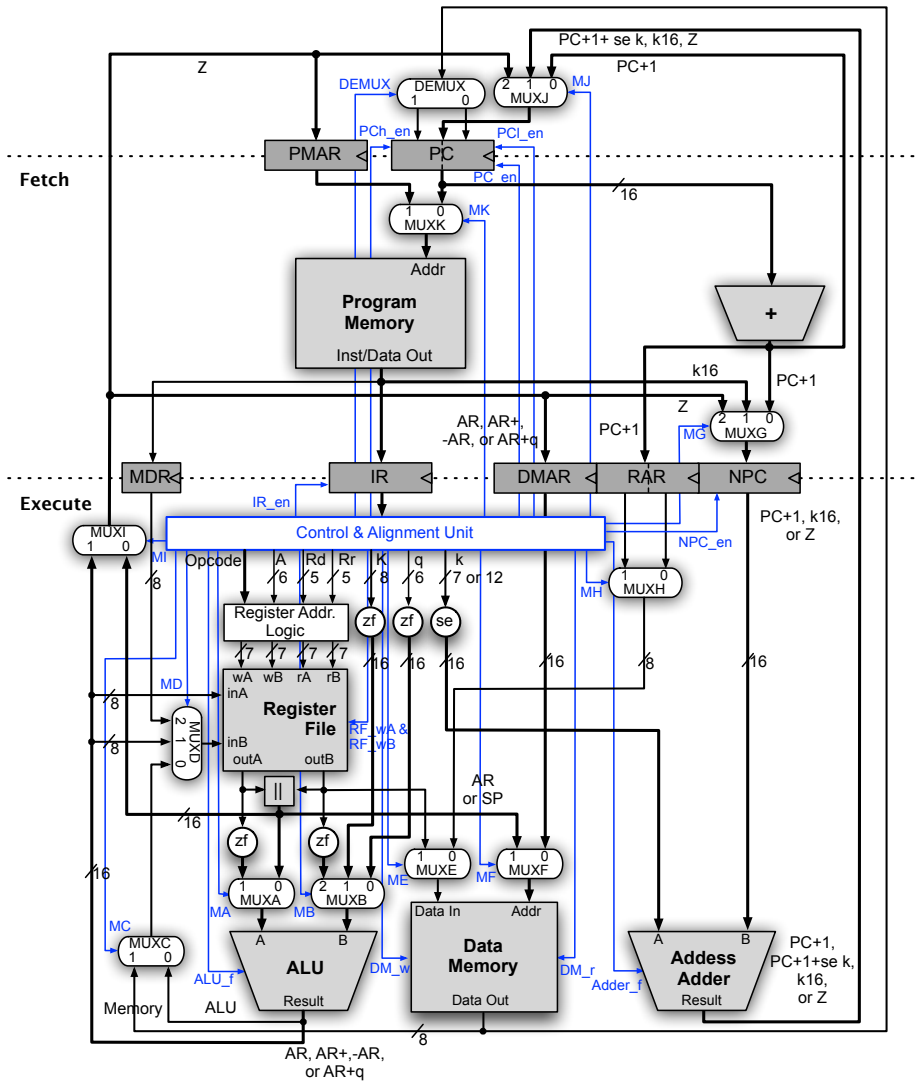


Figure 8.25: Control signals for the enhanced AVR datapath.

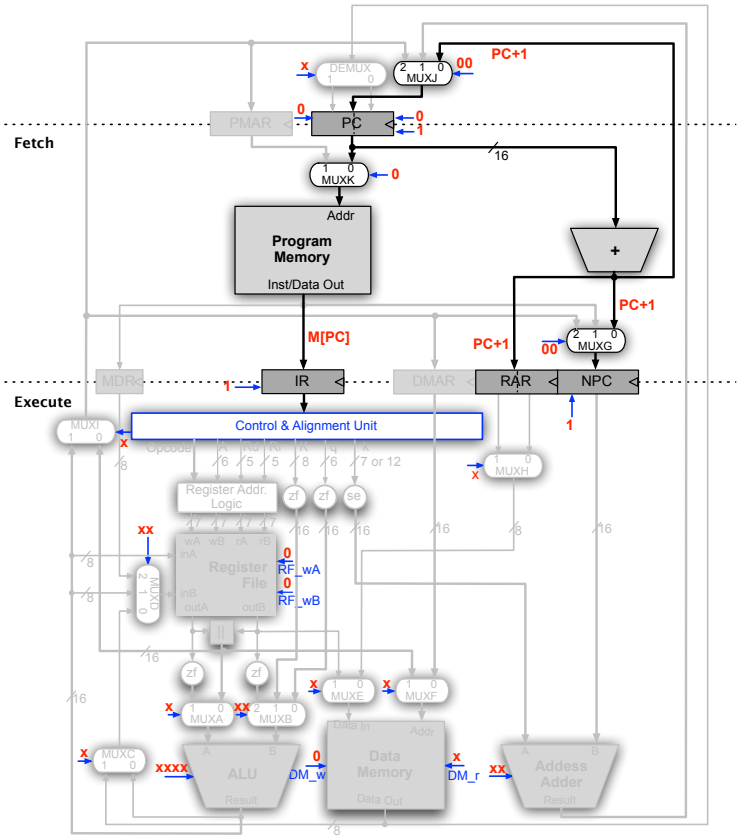


Figure 8.26: Control signals for the Fetch stage.

ecute cycles involving 32-bit instructions. Similarly, PC+1 is latched onto NPC by selecting the 0-input of MUXG and asserting NPC_en. Again, NPC is enabled only during the Fetch cycle, and some Execute cycles involving 32-bit instructions. In contrast, RAR does not have an enable signal, which will latch PC+1 regardless of whether or not it is used in the Execute stage. As you will see, this is not a problem as long as the correct return address of a subroutine call is pushed onto the stack.

Meanwhile, all the control signals for the Execute stage can be "don't cares" as long as the Register File and Data Memory are not modified. This is achieved by setting DM_w=0 for the Data Memory and RF_wA=0 and RF_wA=0 for the Register File.

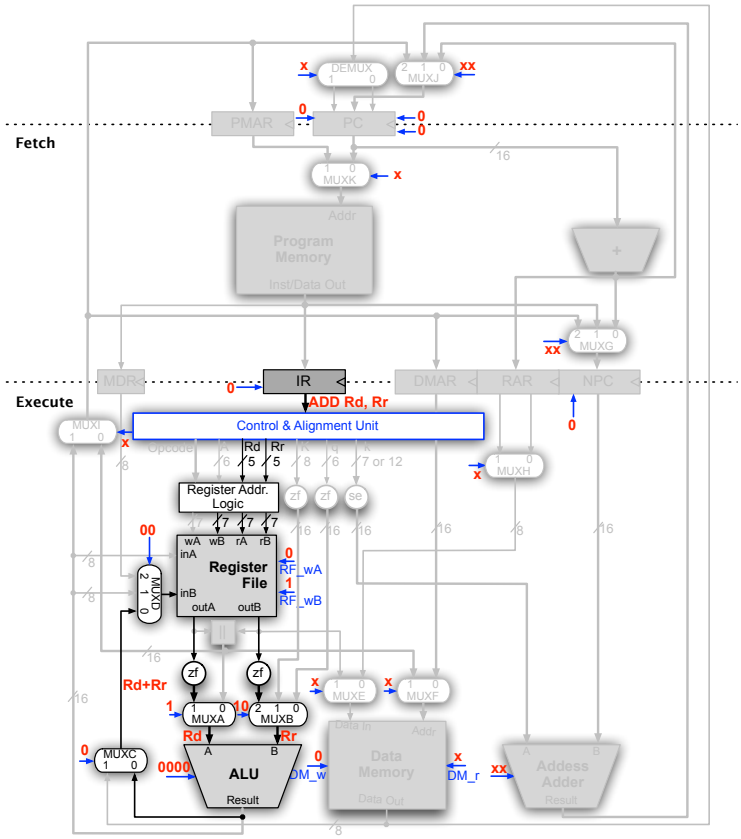


Figure 8.27: Control signals required in EX1 for `ADD Rd,Rr` instruction.

Execute Cycle

As mentioned before, controls signals required in the Execute (EX) stage are different for different instructions.

Figure 8.27 shows the control signals required for the `ADD` instruction. The register identifiers Rd and Rr from the instruction are decoded by the Register Address Logic to read the two source operands from the Register File. MUXA and MUXB are set to 1 and 10, respectively, to accept the zero-filled register contents specified by Rd and Rr from the Register File. The ALU receives the two source operands and performs an add operation defined by the control signal ALU_f = 0000 (see Table 8.1). The result of the operation is then routed to the lower input (i.e., inB) of the Register File via

MUXC (via input-0) and MUXD (via input-0). Finally, the control signal RF_wB=1 allows the result to be written to the Register File at that end of the clock cycle. All the other control signals for the Execute stage can be "don't cares", except DM_w, which has to be set to 0 to prevent the Data Memory from being updated with unrelated or incorrect data. In addition, the control signals PC_en, IR_en, and NPC_en in the Fetch stage are all set to 0 to prevent these registers from being updated with invalid information. This is crucial because PC points to the next instruction (i.e., PC+1) and IR has the information of the current instruction being executed in the Execute stage. Therefore, modifying these contents will be detrimental. MUXs and DEMUX in the Fetch stage can also be "don't cares" since both PC and IR are not updated.

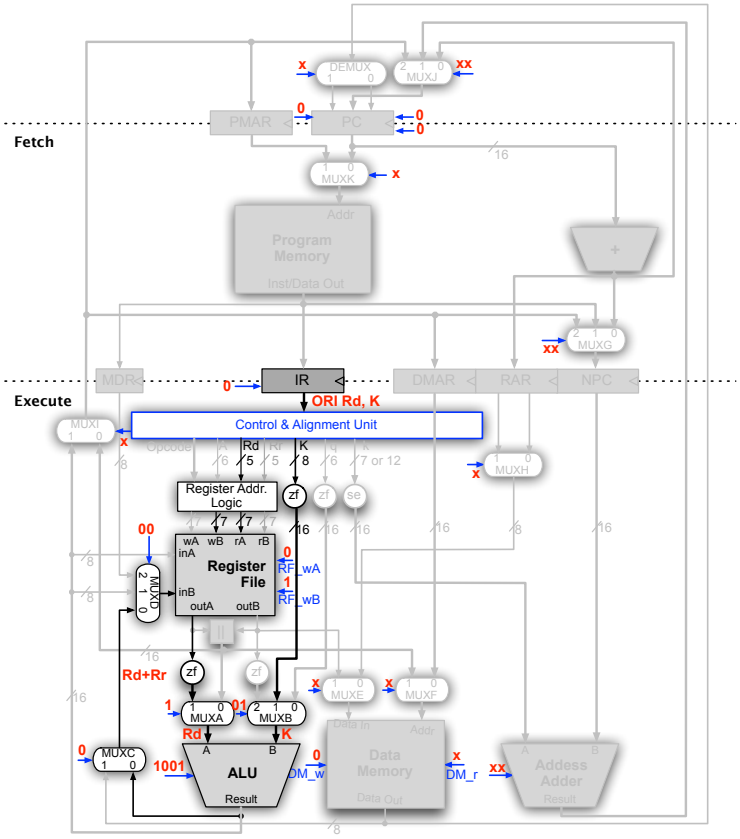


Figure 8.28: Control signals required in EX1 for `ORI Rd,K` instruction.

Figure 8.28 shows the control signals required in EX1 for the `ORI` instruction. As can be seen, the required control signals are almost identical to the `ADD` instruction because `ORI` utilizes similar parts of the datapath. The only difference is that MUXB is set to `01` to select the zero-filled 8-bit constant `K` (i.e., input-1), and the ALU performs logical OR operation by setting the control signal `ALU_f` to `1001` (see Table 8.1).

Figure 8.29 shows the control signals required in EX1 for the `BREQ` instruction. The Address Adder adds the content of NPC (i.e., `PC`+1) latched during the Fetch stage and the sign-extended 7-bit `k` value to generate a PC-relative target address for the branch instruction. If the Z-bit is set, then MUXJ and `PC_en` are set to latch the branch target address onto PC. Otherwise, PC is not latched, i.e., `PC_en`=`0`.

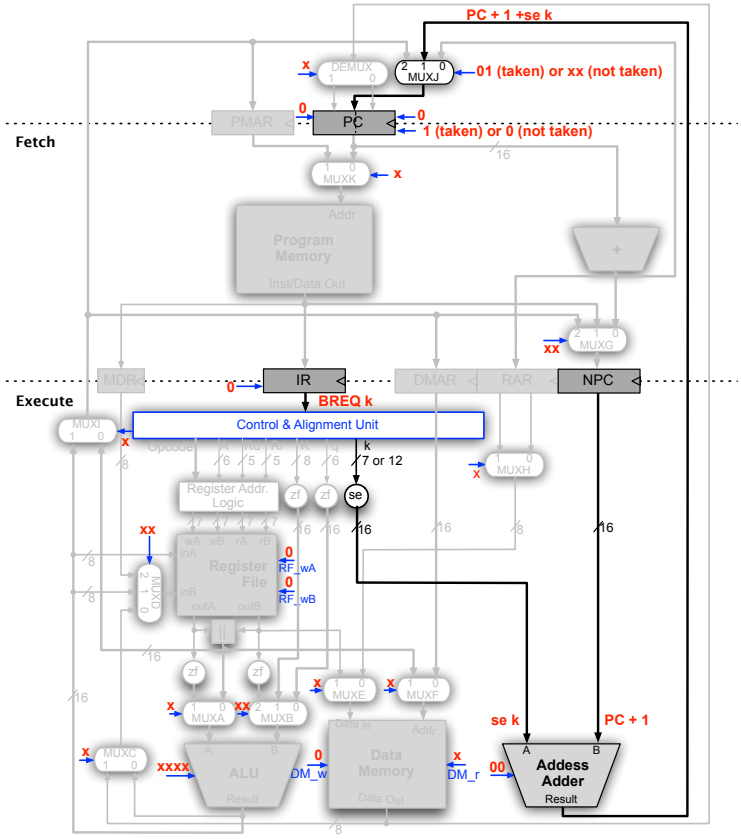


Figure 8.29: Control signals required in EX1 for `BREQ k` instruction.

Figure 8.30 shows the control signals required for LD and ST instructions in EX1, which are common for both instructions. In this example, the upper and lower parts of the Address Register (which in this case is the Y-register) are simultaneously fetched from the Register File, concatenated, and fed to the ALU by selecting the 0-input of MUXA. The control signal ALU_f = 0110 (see Table 8.1) causes the ALU to simply pass the content of the Y-register to the output, which is then available to the input of DMAR by setting MUXI to 1. Meanwhile, control signals DM_w, RF_wA, and RF_wB are all set to zeros to prevent the Data Memory and the Register File from being updated. Similarly, PC_en, IR_en, and NPC_en are also zeros to prevent the contents of PC and IR from being overwritten. Control signals to all the other MUXs (as well as DEMUX) are "don't cares".

Figure 8.31 shows the datapath operations in EX2 for LD and ST instructions. For both of these instructions, MUXF is set to 0 so that the effective address latched onto DMAR in cycle EX1 can be used to access the Data Memory. For loads, the Data Memory is read by setting control signals DM_r = 1 and DM_w = 0, and the operand that is read is routed back to the lower write port (i.e., inB) of the Register File via MUXC and MUXD. The operand is written back to the Register File at the end of the clock cycle by setting RF_wB to 1. For stores, the operand Rr to be written to the Data Memory is read from the Register File and provided as input to the Data Memory by setting the control signal for MUXE to 1, and then written by setting DM_w to 1 and DM_r to 0. All other control signals can be "don't cares", except for RF_wA, RF_wB, IR_en, and PC_en, which are all zeros.

Table 8.10 shows the sequence of micro-operations for CALL k. Figure 8.32 shows the control signals required for CALL. Since CALL is a 32-bit instruction, only the first 16-bit of the instruction has been latched onto IR during the Fetch cycle. Thus, the second 16-bit of the instruction, which represents the target address of CALL, needs to be fetched from the Program Memory using MUXK and latched onto NPC. This is achieved by setting MUXG to accept input-1 and asserting NPC_en. At the same time, PC is incremented again (i.e., PC+2) so that it points to instruction following the CALL instruction, or the return address of the subroutine call, and latched onto PC using MUXJ and PC_en, as well as RAR. Again, except for IR_en, RF_wA, RF_wB, and DM_w, all other control signals can be "don't cares".

In EX2, there are two major operations. First, the low-byte of the return address (RARl) is pushed onto the stack. Second, the Stack Pointer (SP) is decremented so that the high byte of the return address (RARh) can be pushed onto the stack in EX3. The first operation is performed by using the

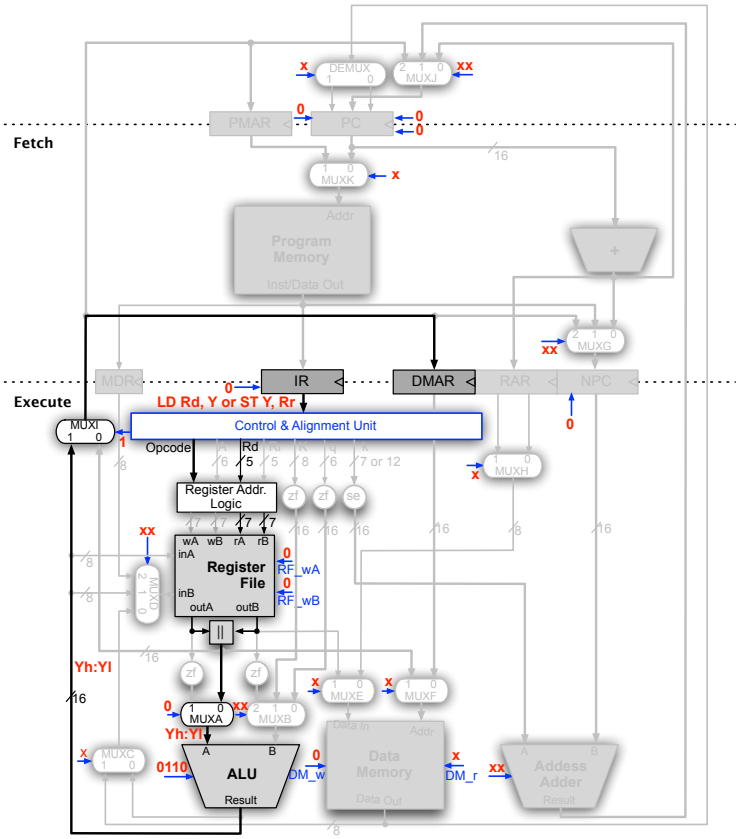
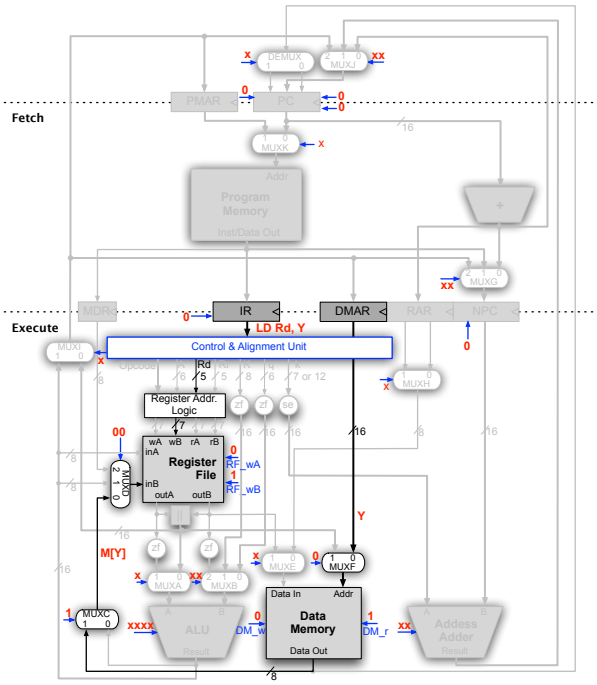


Figure 8.30: Control signals required in EX1 for `LD Rd,Y` and `ST Y,Rr` instructions.

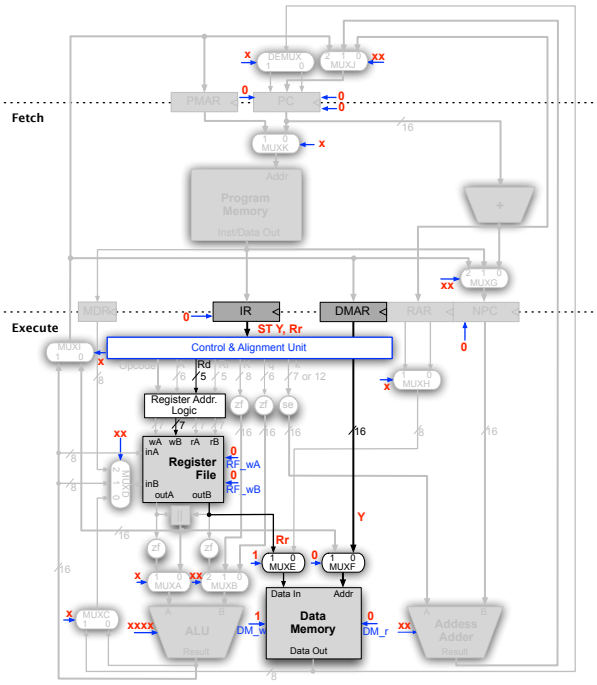
Table 8.10: Direct Subroutine Call.

Direct Subroutine Call	
Stage	Micro-operations
EX1	NPC ← M[PC], PC ← PC + 1, RAR ← PC + 1
EX2	M[SP] ← RARl, SP ← SP -1
EX3	M[SP] ← RARh, SP ← SP -1, PC ← NPC

current address in SP to write RARl into the Data Memory. This is achieved by selecting input-0 for MUXE and input-1 for MUXF, and setting DM_w to 1. The second operation is accomplished by feeding SP into the ALU and decrementing it by setting MUXA to 0 and ALU_f=1010. The decremented



(a) EX2 for loads.



(b) EX2 for stores.

Figure 8.31: Control signals required in EX2 for `LD Rd,X` and `LD X,Rr` instructions.

SP (i.e., SP-1) is then routed back to inA and inB of the Register File, and updated by setting both RF_wA and RF_wB to ones. All other control signals can be "don't cares", except for PC_en, IR_en, NPC_en, and DM_w.

Operations in EX3 are similar to EX2, except this time RARh is pushed onto the stack. In addition, PC is updated with the target address, which is achieved by routing k16 in NPC through the Address Adder and latching it to the PC by setting MUXJ to 01 and PC_en to 1. Otherwise all other control signals are identical to EX2.

Table 8.11 summarizes the required control signals for the six instructions discussed in this section.

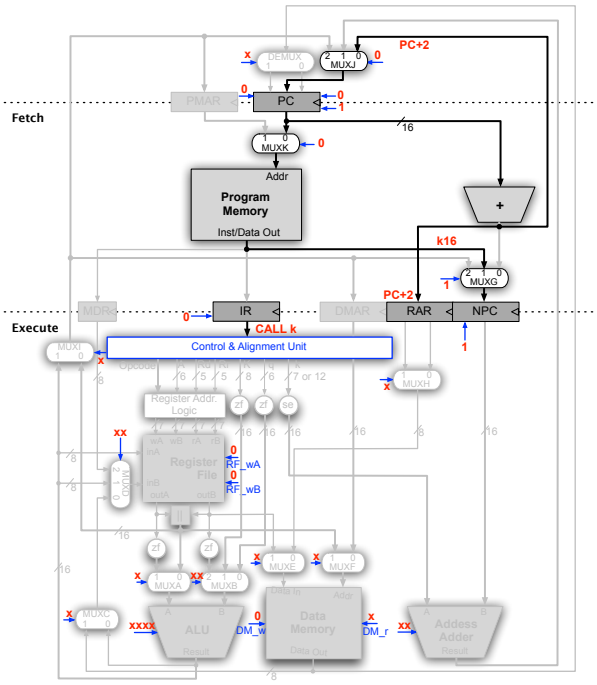
8.6.3 Sequence Control

Now that all of the control signals for Fetch and Execute cycles have been defined, we need a *sequence control* that governs the transitions from one cycle to another, where the cycles are IF, EX1, EX2, and EX3. In sequential control terms, each cycle is referred to as a *state*. Thus, the sequential control of our multi-cycle implementation involves defining a finite state machine. Figure 8.33 shows the *finite state diagram* for the multi-cycle implementation of the six instructions in Table 8.9.

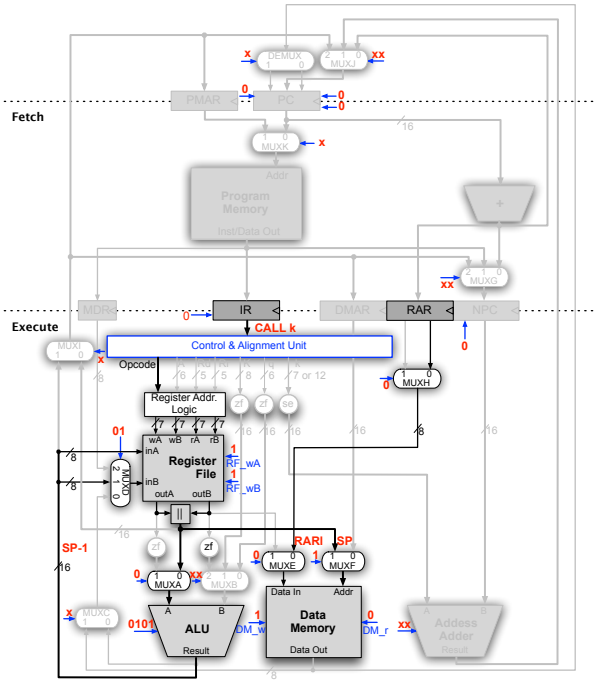
A circle represents a state containing register transfer operations or control signals that are activated while the CAU is in this state. An arrow between states indicates transition from one state to another. There are eleven states in the finite state digram for the multi-cycle implementation, each represented by a 4-bit binary number (0000∼1010).

Table 8.12 shows the *state table*, which was derived from the finite state diagram in Figure 8.33 and control signals from Table 8.11. The sequence control starts at state 0000 (i.e., IF) and then appropriately transitions to other states based on the opcode of the *fetched* instruction indicated by '–' in the Next State field. The opcode field and the Z-bit are "don't cares" indicating that instruction fetch occurs regardless of these inputs.

For instructions that require a single Execute cycle (e.g., `ADD`, `ORI`, `BREQ` instructions), the current state, i.e., 0000 (IF), together with the opcode of the fetched instruction uniquely define the next state. The bit pattern 00 in the two most significant bits of the fetched instruction uniquely defines this instruction as `ADD` and its next state as 0001. Similarly, the bit pattern 01 in the two most significant bits of the fetched instruction uniquely defines this instruction as `ORI` and its next state as 0010. The `BREQ` instruction requires the Z-bit as well as the opcode as inputs. Therefore, if the bits 15-14 are 11, and if the Z-bit is set by the predecessor instruction, the sequence



(a) EX1 of `CALL`.

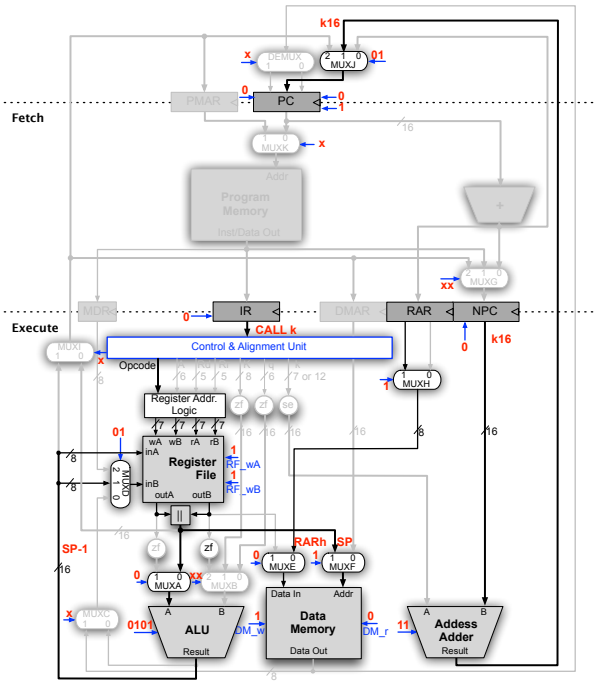


(b) EX2 of `CALL`.

Figure 8.32: Control signals required for `CALL k` instruction.

Table 8.11: Summary of control signals for instructions in Table 8.9

Control Signals	IF	Instructions										
		ADD	ORI	BREQ		LD		ST		CALL		
		EX1	EX1	EX1		EX1	EX2	EX1	EX2	EX1	EX2	EX3
				Z=0	Z=1							
MK	0	x	x	x	x	x	x	x	x	0	x	x
IR_en	1	0	0	0	0	0	0	0	0	0	0	0
PC_en	1	0	0	0	1	0	0	0	0	1	0	1
NPC_en	1	0	0	0	1	0	0	0	0	1	0	0
MG	00	xx	xx	xx	xx	xx	xx	xx	xx	01	xx	xx
MA	x	1	1	x	x	0	x	0	x	x	0	0
MB	xx	10	01	xx	xx	xx	xx	xx	xx	xx	xx	xx
ALU_f	xxxx	0000	1001	xxxx	xxxx	0110	xxxx	0110	xxxx	xxxx	0101	0101
MC	x	0	0	x	x	x	1	x	x	x	x	x
MD	xx	00	00	xx	xx	xx	00	xx	xx	xx	01	01
RF_wA	0	0	0	0	0	0	0	0	0	0	1	1
RF_wB	0	1	1	0	0	0	1	0	0	0	1	1
ME	x	x	x	x	x	x	x	x	1	x	0	0
MF	x	x	x	x	x	xx	0	x	0	x	1	1
DM_r	x	x	x	x	x	0	1	0	0	0	0	0
DM_w	0	0	0	0	0	0	0	0	1	0	1	1
Adder_f	xx	xx	xx	xx	00	xx	xx	xx	xx	xx	xx	11
MH	x	x	x	x	x	x	x	x	x	x	0	1
MI	x	x	x	x	x	1	x	1	x	x	x	x
MJ	00	xx	xx	xx	01	xx	xx	xx	xx	00	xx	01
DEMUX	x	x	x	x	x	x	x	x	x	x	x	x
PCh_en	0	0	0	0	0	0	0	0	0	0	0	0
PCl_en	0	0	0	0	0	0	0	0	0	0	0	0



(c) EX3 of `CALL`.

Figure 8.32: Control signals required for `CALL k` instruction.

control transitions to state 0100. If the opcode bits indicate `BREQ` but the Z-bit is not set, then the sequence control transitions to state 0011, where '–' indicates the state of the processor is not modified in this state.

For instructions that require multiple execute cycles, again the current state 0000 (IF) together with opcode uniquely define the next state (e.g., EX1). The `LD` and `ST` instruction share the common EX1 state (i.e., 0101), and have identical opcodes except for bit 9. If bit 9 is 0 then the instruction is `LD` and thus the next state is 0110, otherwise, it is `ST` and thus the next state is 0111. Therefore, if the bits 15-10 are 100100, then the instruction is either `LD` or `ST` and thus the next state is 0101. Once in state 0101, bit 9 distinguishes between the two instructions. The `CALL` instruction is uniquely defined by bit pattern 1001010 in bits 15-9, and thus state transitions occurs from 0000 to 1000. However, once in state 1000, the transition to state 1001 and the from state 1001 to state 1010 are independent of the input and defined only by the Current State.

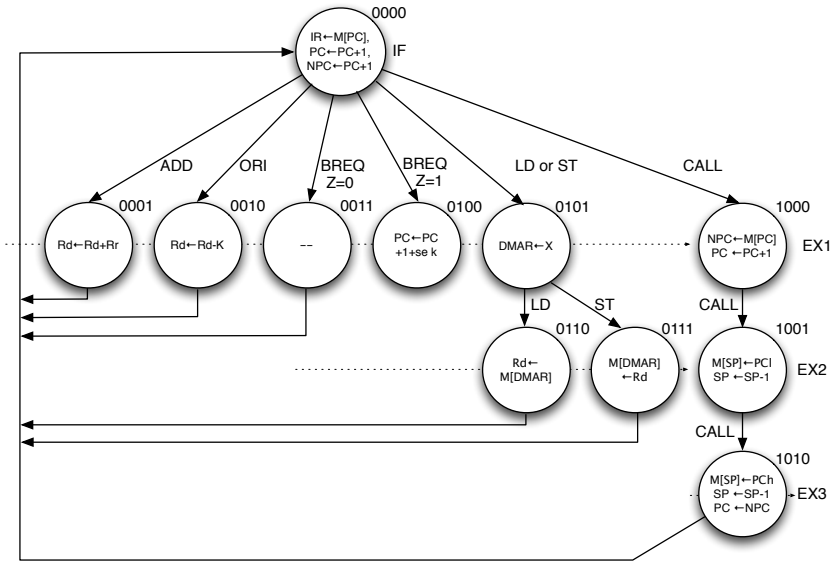


Figure 8.33: The Finite State Machine control for the multi-cycle datapath.

After each instruction executes its last Execute cycle, the control sequence transitions back to state 0000, and the instruction cycle starts over.

Based on finite state table requirements above, the Control Unit can be designed using a finite state machine implementation shown in Figure 8.34. The inputs to the Control Logic are bits of the instruction and Z-bit, while its outputs are the control signals (MK, IR_en, PC_en, NPC_en, etc.). The current state is defined by the output of the four D flip-flops $D_3D_2D_1D_0$. The next state is generated by the Control Logic based on the current state and inputs as defined in Table 8.12.

8.6.4 Register Address Logic

The *Register Address Logic* takes 5-bit register identifiers Rd and Rr, 6-bit I/O register identifier `A`, and opcode, and appropriately generates 7-bit register addresses for the two read ports (rA and rB) and two write ports (wA and wB).

Figure 8.35 shows the mapping for `ADD` and `ORI`, where Rd and Rr are *explicitly* defined by the instructions (op and X represent opcode bits and "don't care", respectively). These two instructions directly map Rd and Rr to inputs rA, rB, and wB of the Register File, i.e., rA = Rd, rB=Rr, and

Table 8.12: Finite state table for the multi-cycle implementation.

Description	Current State	Inputs		Next State	Outputs																						
		Bits 15-9 Opcode	Z		MK	IR_en	PC_en	NPC_en	MG	MA	MB	ALU_f	MC	MD	RF_wA	RF_wB	ME	MF	DM_r	DM_w	Adder_f	MH	MI	MJ	DEMUX	PCh_en	PCl_en
IF	0000	xxxxxxx	x	–	0	1	1	1	00	x	xx	xxxx	x	xx	0	0	x	x	x	0	xx	x	x	00	x	0	0
ADD	0001	00xxxxx	x	0000	x	0	0	0	xx	1	10	0000	0	00	0	1	x	x	x	0	xx	x	x	xx	x	0	0
ORI	0010	01xxxx	x	0000	x	0	0	0	xx	1	01	1001	0	00	0	1	x	x	x	0	xx	x	x	xx	x	0	0
BREQ (Z=0)	0011	11xxxxx	0	0000	x	0	0	0	xx	x	xx	xxxx	x	xx	0	0	x	x	x	0	00	x	x	01	x	0	0
BREQ (Z=1)	0100	11xxxxx	1	0000	x	0	1	1	xx	x	xx	xxxx	x	xx	0	0	x	x	x	0	00	x	x	01	x	0	0
LD/ST (EX1)	0101	100100x	x	–	x	0	0	0	xx	0	xx	0110	x	xx	0	0	x	x	0	0	xx	x	1	xx	x	0	0
LD (EX2)	0110	xxxxxx0	x	0000	x	0	0	0	xx	x	xx	xxxx	1	00	0	1	x	0	1	0	xx	x	x	x	x	0	0
ST (EX2)	0111	xxxxxx1	x	0000	x	0	0	0	xx	x	xx	xxxx	x	xx	0	0	1	0	0	1	xx	x	x	xx	x	0	0
CALL (EX1)	1000	1001010	x	1000	0	0	1	1	01	x	xx	xxxx	x	xx	0	0	x	x	0	0	xx	x	x	00	x	0	0
CALL (EX2)	1001	xxxxxxx	x	1010	x	0	0	0	xx	0	xx	0101	x	01	1	1	0	1	0	1	xx	0	x	xx	x	0	0
CALL (EX3)	1010	xxxxxxx	x	0000	x	0	1	0	xx	0	xx	0101	x	01	1	1	0	1	0	1	11	1	x	01	x	0	0

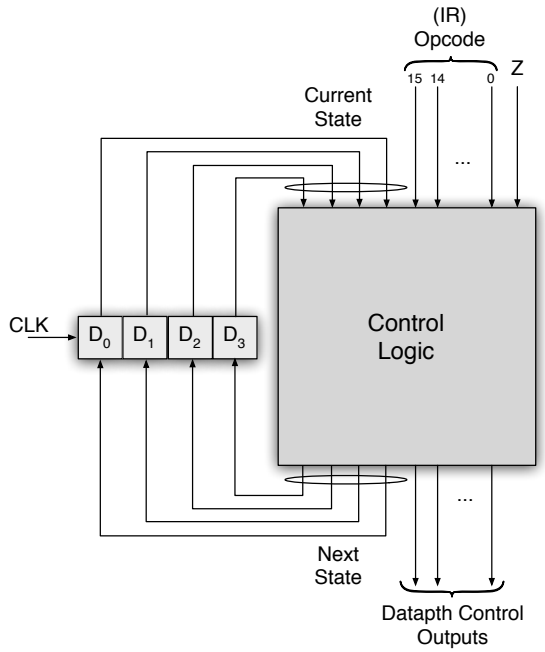


Figure 8.34: The Finite State Machine control for the datapath.

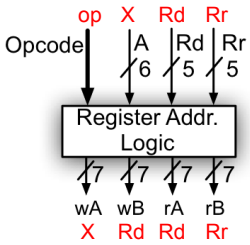
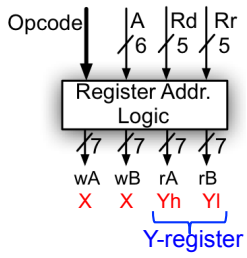


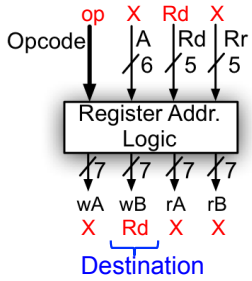
Figure 8.35: RAL Mapping for `ADD` and `ORI`.

wB=Rd. Note that even though both instructions have the same mapping, `ORI` uses the constant K rather than the source operand Rr.

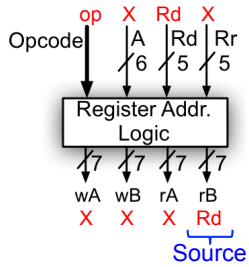
The I/O register identifier A needs to be offset by 32 since 64 I/O registers reside after the 32 GPRs. This is achieved by padding a zero left of the most significant bit to convert the 6-bit I/O register identifier `A` into a 7-bit number and then adding 32 to do it. Since I/O instructions are not included



(a) Mapping for `LD` and `ST` in EX1.



(b) Mapping for `LD` in EX2.



(c) Mapping for `ST` in EX2.

Figure 8.36: RAL Mapping for `LD` and `ST`.

in the six instructions in Table 8.9, their mapping will not be discussed.

The logic implementations for instructions that *implicitly* define registers (e.g., `X`, `Y`, and `Z`-registers for `LD` and `ST`, and SP for `CALL`) are more tricky and require the understanding of how opcodes are encoded. For example, consider `LD` and `ST` instructions. The address register for these instructions can be either `X`, `Y`, or `Z`, which is specified by bits 3 and 2 of the instruction format (indicated as "aa" in Figure 8.24). For the `Y`-register, bits 3-2 are 10 (for your information, bits 3-2 are 11 for `X`-register and 00 for `Z`-register). Thus, these two bits would have to be decoded from the IR and appropriately mapped to registers R27 and R26. Similarly, `CALL` requires the use of SP to push the return address onto the stack. Thus, the opcode bits must be uniquely encoded and mapped to the low and high bytes of the SP (i.e., SPl and SPh) located in 95th ($5D) and 96th ($5E) locations in the Register File. In the context of the six instructions covered in this chapter, instructions that implicitly define registers are `LD`, `ST`, and `CALL`.

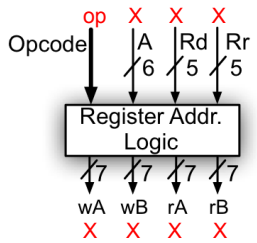
Figure 8.36 shows the mapping for the LD and ST instructions. These two instructions require the information from the opcode as well its current state to determine the mapping. These two instructions shared the same EX1, i.e., state 0101, and refer to Y-register, and thus Yh and Yl have to be mapped to rA and rB, respectively. For LD in EX2 (i.e., state 0110), Rd in the instruction format serves as the destination register, and thus is mapped to wB. In contrast, for ST in EX2 (i.e., state 0111), Rd in the instruction format serves as the source register, and is mapped to rB.

Figure 8.37 shows the mapping for the CALL instruction. CALL also requires the information from the opcode as well its current state to determine the mapping. In EX1 (i.e., state 1000), no mapping is necessary since only the Fetch stage is active in fetching the target address k16 from the Program Memory. In EX2 (i.e., state 1001), SP needs to be read from the Register File, decremented, and written back to the Register File. Therefore, rA and rB as well as wA and wB are mapped to SPh and SPl. The same occurs in EX3 (i.e., state 1010).

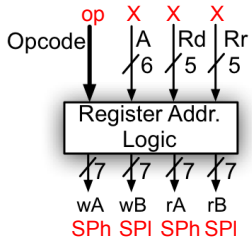
Figure 8.38 shows the implementation of the Register Address Logic for the instructions in Table 8.9. The lower three MUXs choose between registers defined either explicitly or implicitly. For explicitly defined register identifiers, such as Group A and B instructions, bits 15-14 in the instruction format (which equal to either 00 or 01) causes the GPR signal choose rA=Rd, rB=Rr, and wB=Rd. For implicitly defined register identifiers, such as Group C instructions, the Decoder Logic uses GPR as well as SP_AR to choose between Y-register or SP on the two upper MUXs based on bits 15-9 of the instruction. For example, the CALL instruction is uniquely defined by the bit pattern 1001010 for bits 15-9. Thus, when this pattern appears on the input of the Decoder Logic, the SP_AR and GPR signals choose rA=SPh, rB=SPl, wA=SPh, and wB=SPl so that the SP can be read from the Register file, decremented, and store back into the Register File.

8.7 Pipeline Implementation

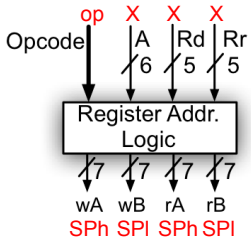
{*(BL): Work-in-progress...*} Figure 8.39 shows the pipline datapath for the AVR microarchitecture.



(a) Mapping for `CALL` in EX1.



(b) Mapping for `CALL` in EX2.



(c) Mapping for `CALL` in EX3.

Figure 8.37: RAL Mapping for `CALL`.

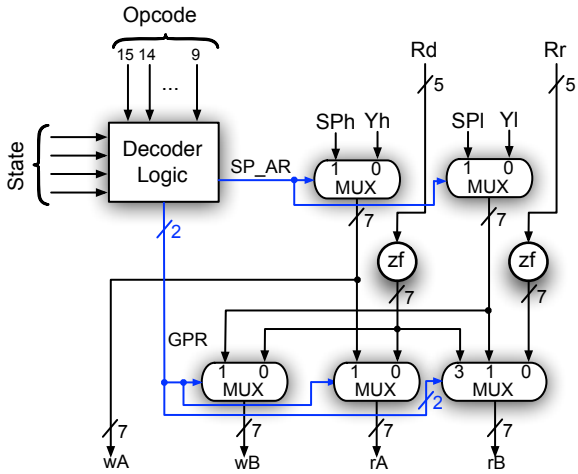


Figure 8.38: Register Address Logic.

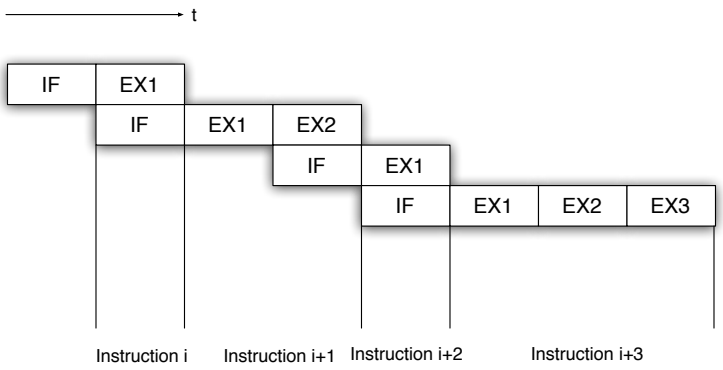


Figure 8.39: Pipeline implementation of the datapath.

Chapter 9

Arithmetic and Logic Unit

9.1 Number Systems

Computers deal with information, but information has many different meanings. Figure 9.1 shows the taxonomy of information handle by computers. At the highest level, information in a computer is either *instructions*, more specifically assembly instructions, or *data*. In Chapters 4 and 8, we discussed the information contained in an instruction format and how it is decoded and executed by the underlying microarchitecture. This chapter discusses operations on data, which can be either numeric or non-numeric. *Non-numeric* data refers to *American Standard Code for Information Interchange* (ASCII) codes that represent the character-encoding scheme for the English Alphabet. *Numeric* data can be further divided into fixed-point and floating-point. *Floating-point* format is significantly different than fixed-point and requires special treatment. *Fixed-point* or integer data can be not only of different sizes but also unisgned as well signed. Therefore, it is important to know the type of data we are dealing with since an n-bit data will have different meaning depending on its type. This is the reason why high-level languages require that programmers declare type information with each variable declaration, such as `int`, `short`, `long`, `float`, `double`, `signed`, `unsigned`, or `char`. This section discusses arithmetic and logic operations on fixed-point binary numbers.

An unsigned n-bit binary number N is given as

$$N = (b_{n-1}b_{n-2}\cdots b_1b_0), \tag{9.1}$$

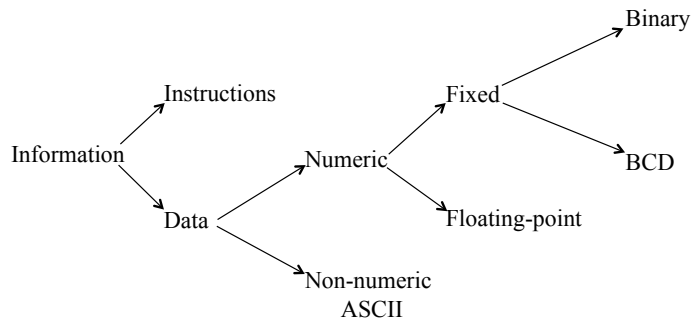


Figure 9.1: Information in a computer.

which has a range of

$$0 \leq N \leq 2^n - 1. \tag{9.2}$$

In contrast, there are three ways to represent signed numbers: *sign-magnitude*, *1's-complement*, and *2's-complement*. The following subsections discuss each one of them.

9.1.1 Sign-Magnitude Representation

Signed-Magnitude representation of an n-bit binary number N is given as

$$\begin{aligned} N &= (0b_{n-2}b_{n-3}\cdots b_1b_0) \text{ and} \\ -N &= (1b_{n-2}b_{n-3}\cdots b_1b_0), \end{aligned} \tag{9.3}$$

which has a range of

$$\begin{aligned} &+0 \leq N \leq 2^{n-1} - 1 \text{ and} \\ &-(2^{n-1} - 1) \leq N \leq -0. \end{aligned} \tag{9.4}$$

The following example shows sign-magnitude representation:

Example 9.1. *Sign-magnitude representation with n=4.*

$0111 = 7$	$1111 = -7$
$0110 = 6$	$1110 = -6$
$0101 = 5$	$1101 = -5$
$0100 = 4$	$1100 = -4$
$0011 = 3$	$1011 = -3$
$0010 = 2$	$1010 = -2$
$0001 = 1$	$1001 = -1$
$0000 = +0$	$1000 = -0$

As can be seen from this example, sign-magnitude representation is pretty straightforward and consistent with the way we represent numbers in decimal. That is, the most significant bit represents the sign and the rest of the $n - 1$ bits represent the number. However, there are some challenges with using signed-magnitude to perform arithmetic. First, there are two zeroes (i.e., $+0$ and -0), which require additional logic to distinguish these two cases. Second, adding two numbers of opposite signs is difficult. Case in point, consider the following example of adding two sign-magnitude numbers with opposite signs.

Example 9.2. *Addition of two numbers with opposite signs in Sign-magnitude representation.*

$$\begin{array}{cccccc} & 0 & 1 & 0 & 0 & (+4) \\ + & 1 & 1 & 0 & 1 & (-5) \\ \hline & 1 & 0 & 0 & 1 & (-1) \end{array}$$

The result of this operation is pretty straightforward when it is performed in decimal, i.e., 4 - 5 = -1. However, in order to perform this operation in binary, we applied some know-how gained from having performed arithmetic since grade school. That is, we compared the magnitudes of the two numbers to determine that 5 is larger than 4 and thus we perform 5 - 4 = 1, and then the result was negated to generate -1. Thus, performing this operation in hardware requires a magnitude comparator and a subtractor. We will see in subsequent subsections that 1's- and 2's-complement representations do not have this shortcoming.

9.1.2 1's-complement Representation

1's-complement representation of an n-bit binary number N is given as

$$\begin{aligned} N &= (0b_{n-2}b_{n-3}\cdots b_1b_0) \text{ and} \\ -N &= (1\bar{b}_{n-2}\bar{b}_{n-3}\cdots \bar{b}_1\bar{b}_0), \end{aligned} \tag{9.5}$$

with a range given by

$$\begin{aligned} +0 \leq N \leq 2^{n-1} - 1 \text{ and} \\ -(2^{n-1} - 1) \leq N \leq -0 \end{aligned} \tag{9.6}$$

The following example shows 1's-complement representation:

Example 9.3. *1's-complement representation with n=4.*

$0111 = 7$	$1111 = -0$
$0110 = 6$	$1110 = -1$
$0101 = 5$	$1101 = -2$
$0100 = 4$	$1100 = -3$
$0011 = 3$	$1011 = -4$
$0010 = 2$	$1010 = -5$
$0001 = 1$	$1001 = -6$
$0000 = 0$	$1000 = -7$

We learned from introductory course in digital logic that 1's-complement of a number N, $OC(N)$, is obtained by complementing each bit. However, there is mathematical meaning of $OC(N)$ with n bits, which is given by

$$OC(N) = 2^n - N - 1 \tag{9.7}$$

For example, 1's-complement of 0101 (5) is $2^4 - 5 - 1 = 10$, which is equal to 1010, which is equivalent to complementing each bit of 0101. Similarly, 1's-complement of 1010 (-5) is $2^4 - 10 - 1 = 5$, which is equal to 0101.

The following examples illustrate addition/subtraction operations in 1's-complement, which do not require a comparator or a subtractor:

Example 9.4. *Addition of two positive numbers in 1's-complement.*

$$\begin{array}{cccccc} & 0 & 0 & 1 & 1 & (3) \\ + & 0 & 0 & 1 & 0 & (2) \\ \hline & 0 & 1 & 0 & 1 & (5) \end{array}$$

This example adds two positive numbers, and thus performed using straightforward binary addition.

Example 9.5. *Addition of two numbers with opposite signs in 1's-complement:*

$$\begin{array}{cccccc} & 0 & 1 & 0 & 0 & (4) \\ + & 1 & 0 & 1 & 0 & (-5) \\ \hline & 1 & 1 & 1 & 0 & (-1) \end{array} \qquad \begin{array}{cccccc} & 0 & 1 & 0 & 1 & (5) \\ + & 1 & 0 & 1 & 1 & (-4) \\ \hline \mathbf{1} & 0 & 0 & 0 & 0 & \\ + & & & & 1 & \text{EAC} \\ \hline & 0 & 0 & 0 & 1 & (1) \end{array}$$

In the first example, the second operand is a negative number. Therefore, the net effect is to subtract the second operand from the first operand, which is performed by simply adding the two numbers. The result is a negative number, which is indicated by the most significant bit, and thus taking 1's-complement of 1110 gives 0001 indicating that the result 1110 is -1.

The second example generated a carry-out, which needs to be added to the partial result to generate the final result. This process of adding the carry-out is referred to as *end-around carry* (EAC). So why do we have to perform EAC when we add/subtract in 1's-complement? The answer lies in Eq. 9.7.

Consider two positive numbers N_1 and N_2. Performing $N_1 - N_2$ is equivalent to adding 1's-complement of N_2 to N_1, which is represented by the following equation:

$$\begin{aligned} N_1 + OC(N_2) &= N_1 + 2^n - N_2 - 1 \\ &= 2^n + (N_1 - N_2) - 1. \end{aligned} \tag{9.8}$$

If $N_1 > N_2$, then the result should be $(N_1 - N_2)$, but there are extra terms (mainly 2^n and -1) that should not be part of the result. 2^n term represents the *carry-out*, thus adding 1 to the partial result eliminates the -1 term, thus EAC! If $N_1 < N_2$, then the result should be $OC(N_1 - N_2)$, which is $2^n + (N_1 - N_2) - 1$. Thus, there will be no carry-out and thus no need to perform EAC! In the case $N_1 = N_2$, the result should be $2^n - 1$, which means there is no carry-out and the result should be all ones (try it for yourself!).

These results show that 1's-complement can be used as a magnitude comparator. That is, after performing $N_1 - N_2$, if there is carry-out then

$N_1 > N_2$. If there is no carry-out, then $N_1 < N_2$. If there is no carry-out and the result is zero, then $N_1 = N_2$.

The problem with 1's-complement representation is that, in addition to having to detect two zeros, an addition/subtraction takes at most two add delays, where an *add delay* represents the delay to perform n-bit addition/-subtraction, due EAC.

Example 9.6. *Addition of two numbers that cause an overflow.*

$$\begin{array}{cccccc} & 0 & 1 & 0 & 1 & (5) \\ + & 0 & 0 & 1 & 1 & (3) \\ \hline & 1 & 0 & 0 & 0 & (\text{-}7) \end{array}$$

This example adds two positive numbers but the result is negative. This is an example of an *overflow*, which is caused when the result is larger than the maximum range of $2^{n-1} - 1$.

9.1.3 2's-complement Representation

2's-complement representation of an n-bit binary number N is given as

$$\begin{aligned} N &= (0b_{n-2}b_{n-3}\cdots b_1b_0) \text{ and} \\ -N &= (1\bar{b}_{n-2}\bar{b}_{n-3}\cdots \bar{b}_1\bar{b}_0 + 1), \end{aligned} \tag{9.9}$$

with a range given by

$$-2^{n-1} \leq N \leq 2^{n-1} - 1. \tag{9.10}$$

The following example shows 2's-complement representation:

Example 9.7. *2's-complement representation with n=4.*

$0111 = 7$	$1111 = -1$
$0110 = 6$	$1110 = -2$
$0101 = 5$	$1101 = -3$
$0100 = 4$	$1100 = -4$
$0011 = 3$	$1011 = -5$
$0010 = 2$	$1010 = -6$
$0001 = 1$	$1001 = -7$
$0000 = 0$	$1000 = -8$

As can be seen, 2's-complement representation has only one zero and extends the range of negative numbers by one more number.

Again, we learned that 2's-complement of a number N, $TC(N)$, is obtained by first performing 1's-complement and then adding one. Similar to $OC(N)$, the mathematical meaning of $TC(N)$ with n bits is given by

$$TC(N) = 2^n - N. \tag{9.11}$$

For example, 2's-complement of 0101 (5) is $2^4 - 5 = 11$, which is 1011 equivalent to complementing each bit of 0101 and then adding one to it. Similarly, 2's-complement of 1011 (-5) is $2^4 - 11 = 5$, which is 0101.

Example 9.8. *Addition of two numbers with opposite signs:*

	0	1	0	0	(4)
+	1	0	1	0	(-6)
	1	1	1	0	(-2)

	0	1	1	0	(6)
+	1	1	0	1	(-3)
1	0	0	1	1	Discard carry
	0	0	1	1	(3)

In the first example, no carry-out was generated. In the second example, carry-out was generated and discarded to yield the final result. The reason why the carry-out is discarded in 2's-complement can be explained using Eq. 9.11.

Consider two positive numbers N_1 and N_2. Performing $N_1 - N_2$ is equivalent to

$$N_1 + TC(N_2) = N_1 + 2^n - N_2 = 2^n + (N_1 - N_2). \tag{9.12}$$

Suppose $N_1 > N_2$, then the result should be $(N_1 - N_2)$, but there are an extra term (i.e., 2^n) that should not be part of the result. 2^n term represents the carry-out, and thus discarding it results in the correct answer! When $N_1 < N_2$, then the result should be $OC(N_1 - N_2)$, which is $2^n + (N_1 - N_2)$. Thus, there should not be a carry-out! For $N_1 = N_2$, the result should be 2^n, which means all zeros and a carry-out (try it for yourself!).

Similar to 1's-complement, 2's-complement can be used to perform magnitude comparison. That is, after performing $N_1 - N_2$, if there is carry-out, then $N_1 > N_2$. If there is no carry-out, then $N_1 < N_2$. If there is a carry-out and the result is zero, then $N_1 = N_2$.

Based on the aforementioned discussion, 2's-complement is the best number system for binary arithmetic because it only has one zero and requires at most one add delay. Therefore, signed numbers are represented in 2's-complement.

9.2 Shift Operations

There are three types of shifts: logical, arithmetic, and circular. Since signed numbers are represented as 2's-complement, we will discuss shift operations in the context of 2's-complement numbers.

9.2.1 Logical Shifts

A logical shift operation involves simply shifting in a zero left or right. The following examples illustrate logical shift left and right:

Example 9.9. *Logical shift left and right:*

0	0	1	0			1	0	1	0	
0	1	0	**0**	<= Shift left		**0**	1	0	1	<= Shift right

Logical shift left and right on a positive number N are equivalent to $N \times 2$ and $\lfloor N/2 \rfloor$, respectively. However, these operations have no numerical meaning when the numbers are negative.

9.2.2 Arithmetic Shifts

Arithmetic shifts are identical to logical shifts when the numbers are positive, but when the numbers are negative retain their numerical meaning. The following examples illustrate arithmetic shift left and right of 2's-complement numbers:

Example 9.10. *Arithmetic shift left and right:*

0	0	1	0	(2)		0	0	1	0	(2)		
0	1	0	**0**	(4)	<= Shift left	**0**	0	0	1	(1)	<= Shift right
1	1	0	1	(-3)		1	1	0	1	(-3)		
1	0	1	**0**	(-6)	<= Shift left	**1**	1	1	0	(-2)	<= Shift right

As can be seen, when a number is positive, arithmetic shifts are equivalent to logical shifts. For a negative number $-N$, a 0 is shifted in on a shift left operation ($-N \times 2$), while the sign is maintained by shifting in a 1 for a shift right operation ($\lceil N/2 \rceil$).

9.2.3 Circular Shifts

A circular shift moves either the most significant bit to the last significant bit, while shifting all other entries to the next position, or vice versa. The following examples illustrate circular shift left and right operations:

Example 9.11. *Circular shift left and right:*

1	0	1	0		1	0	1	**0**		
0	1	0	**1**	<= Shift left	**0**	1	0	1	<= Shift right	

Unlike logical and arithmetic shifts, circular shifts have no mathematical meaning. Circular shifts implement bit permutations and are used in many applications, e.g., cryptography.

9.3 Basic ALU Design

9.3.1 Full Adder

The fundamental component in an ALU is the *full adder*. Fig. 9.2 shows a full adder with its truth or function table and Karnaugh map (K-map). The full adder takes x_i and y_i and carry-in c_i as inputs and generates sum s_i and carry-out c_{i+1}. The realization of s_i and c_{i+1} based on the K-maps is given by the following equations:

$$s_i = x_i \oplus y_i \oplus c_i \text{ and} \tag{9.13}$$

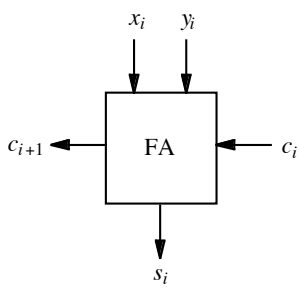
$$c_{i+1} = x_i y_i + (x_i \oplus y_i) c_i \text{ or} \tag{9.14}$$

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i. \tag{9.15}$$

As can be seen by Fig. 9.2(c) and Eqs. 9.14 and 9.15, there are couple of ways to implement c_{i+1}. Fig. 9.3 shows a full adder implementation using Eq. 9.15.

9.3.2 n-bit Ripple Carry Adder

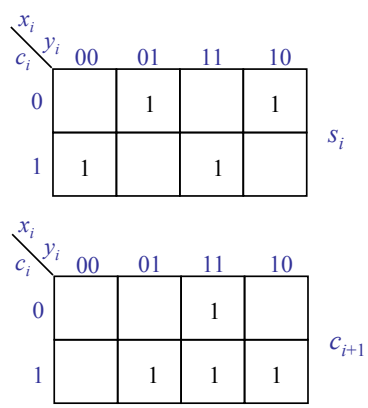
The simplest way to implement an n-bit adder is by cascading n full adders. This type of adders is called *Ripple Carry Adder* because a carry has to ripple across all the full adders from right to left before a valid result can be generted. Fig. 9.4 shows a diagram of an n-bit Ripple Carry Adder, where



(a) Full adder.

x_i	y_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(b) Truth table.



(c) K-map.

Figure 9.2: Full adder and its truth table and K-map.

x_i and y_i represent the inputs, s_i represents the sum, c_0 is the carry-in, and c_n is the carry-out.

In the following subsection, we will discuss how an n-bit adder can be added with additional logic to implement functionalities provided in a typical ALU. Moreover, an adder serves as the basis for performing more complicated operations, such as multiplication and division. Therefore, the delay involved in performing addition/subtraction is crucial for determining how fast an ALU will be.

Suppose, all logic gates have a delay of 1 *gd*, where *gd* stands for *gate*

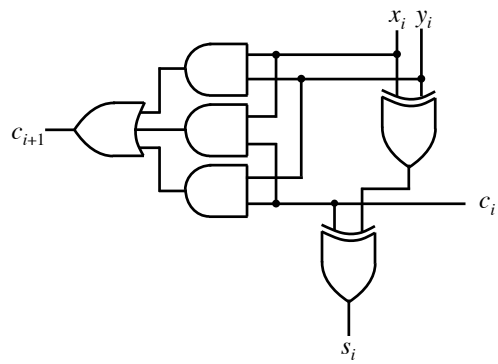


Figure 9.3: Full adder implementation.

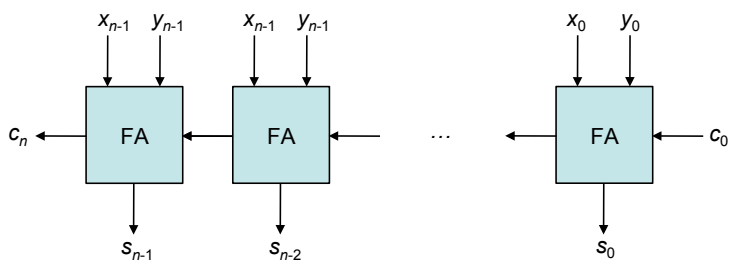


Figure 9.4: Ripple Carry Adder.

delay, except for the EOR gate, which has a delay of 2 *gds*. For an n-bit Ripple Carry Adder, the delay to perform an add, which is defined as the time between when a valid input ($x_i's$, $y_i's$, and c_0) is provided and when a valid output ($s_i's$ and c_n) is available, is given as $2n$ *gds*. Fig. 9.5 illustrates the propagation delay of an n-bit Ripple Carry Adder. Let us focus on the last, i.e., $(n-1)^{th}$, stage of the adder. The delay for carry-in (i.e., c_i) to propagate through one stage is 2 *gds* due to the AND-OR gates. In the last stage, the result of the first EOR gate (i.e., $x_{n-1} \oplus y_{n-1}$) was already generated by the time carry-in c_{n-1} becomes available after $2(n-1)$ *gds*. Therefore, both c_n and s_{n-1} become available after 2 more *gds* resulting in total delay of $2n$ *gds*.

The significance of $2n$ *gds* is that as n increases the delay becomes significant. For example, a 64-bit operation would require 128 *gds*. This means that an add/subtract operation may not complete in one cycle or the clock

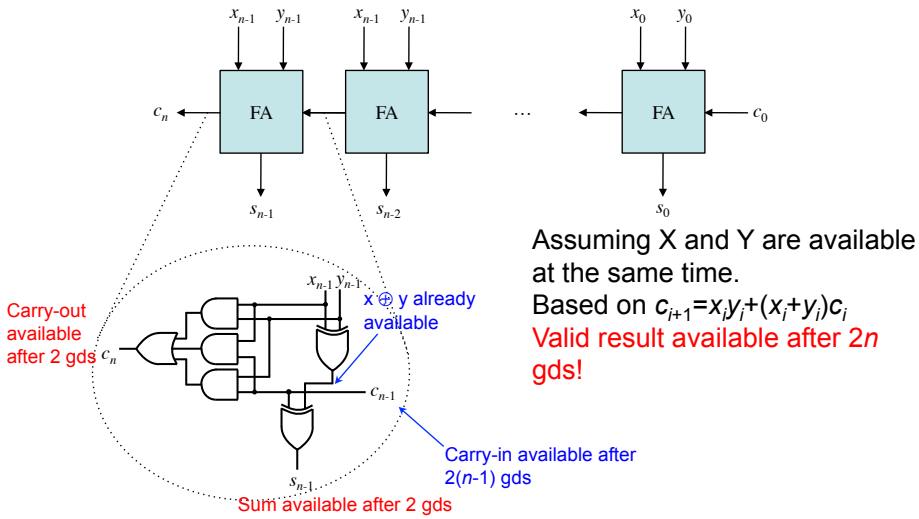


Figure 9.5: Ripple Carry Adder delay.

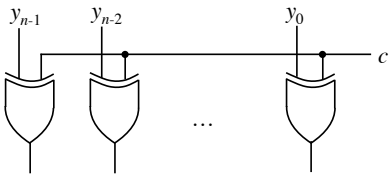
cycle time may not be arbitrary decreased by making the clock faster. In Sec. 9.3.5, we will discuss some techniques to speedup additions/subtractions.

9.3.3 n-bit ALU

An n-bit ALU can be implemented using an n-bit Ripple Carry Adder and some additional logic. As we discussed in Sec. 9.1, one of the major advantages of using 2's-complement number system is that a separate subtractor is not required. Instead, a subtraction can be performed by negating one of the operands by performing 2's-complement and adding it to the other operand. Figure 9.6 shows a *complementer* that can be used to perform complement operation, and together with an adder implement a variety of operations.

As can be seen from the function table in Fig. 9.6(b), the set of EORs act as a complementer when the control signal c is set to 1. Using the combination of a simple complementer and a Ripple Carry Adder as shown in Fig. 9.7, the following set of operations can perform be performed:

- $x + y$ - add x and y
- $x + \bar{y}$ - add x with 1's-complement of y (1's-complement subtract)



(a) A complementer.

y_i	c	Result
0	0	0
0	1	1
1	0	1
1	1	0

(b) Function Table.

Figure 9.6: A complementer with its Function Table.

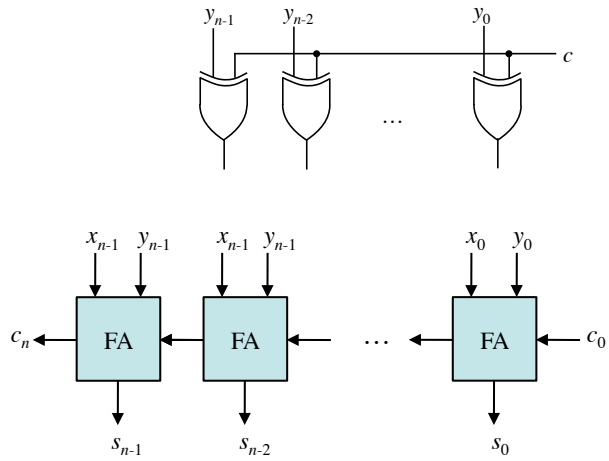
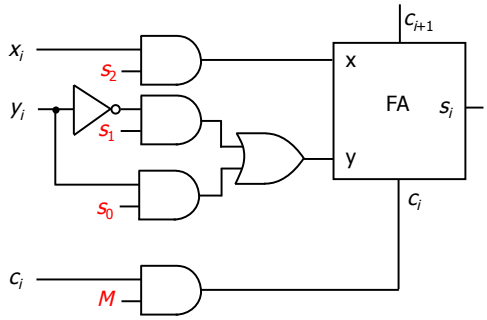


Figure 9.7: Ripple Carry Adder with a complementer.

- $x + y + 1$ - add x and y with carry
- $0 + y$ - move y
- $\bar{y}$ - 1's-complement of complement y
- $\bar{y} + 1$ - 2's-complement of complement y
- $x + \bar{y} + 1$ - add x with 2's-complement of y (2's-complement subtract)

Fig. 9.8 shows a single *bit-slice* or *stage* of an n-bit ALU with all the arithmetic and logic operations that can be performed. As can be seen from the figure, each stage of the ALU consists of a full-adder with additional logic to control its operations. The control signals s_2, s_1, s_0, and M determine



(a) Bit-slice ALU.

ARITHMETIC

M	$S_2S_1S_0$	C_0	Micro Operation	Description
1	0 0 1	1	s = y + 1	Increment y
1	0 1 0	1	s = y' + 1	2's complement y
1	1 0 0	1	s = x + 1	Increment x
1	1 0 1	0	s = x + y	Add x and y
1	1 1 0	0	s = x + y'	x plus 1's complement of y
1	1 1 0	1	s = x + y' + 1	x plus 2's complement of y
1	1 1 1	0	s = x - 1	Decrement x

(b) Arithmetic operations.

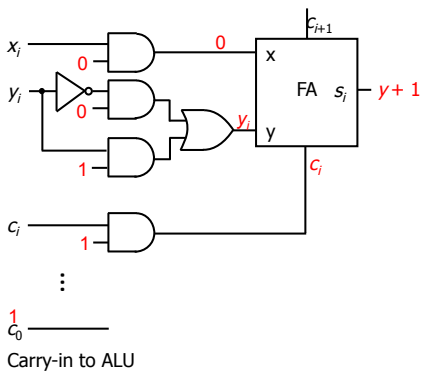
LOGIC

M	$S_2S_1S_0$	Micro Operation	Description
0	0 0 0	s = 0	Clear
0	0 0 1	s = y	Transfer y
0	0 1 0	s = y'	Comp. y
0	0 1 1	s = 1	Set
0	1 0 0	s = x	Transfer x
0	1 0 1	s = x⊕y	EOR
0	1 1 0	s = (x⊕y)'	ENOR
0	1 1 1	s = x'	Comp. x

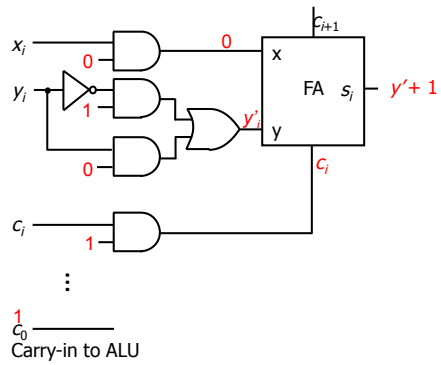
(c) Logic operations.

Figure 9.8: Bit-slice ALU.

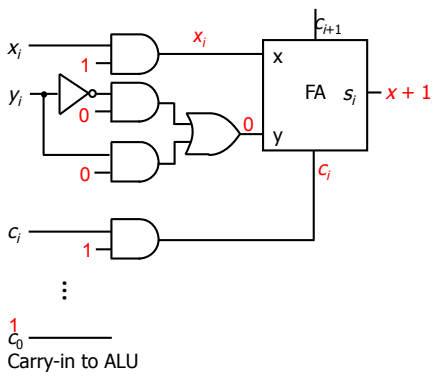
which operation will be performed. When M is 1, all the bit slices are connected, which allows carries to propagate through the stages for arithmetic operations. On the other hand, when M is 0, each stage is independent and implements logic operations. The control signals s_2, s_1, and s_0 determine arithmetic (when M=1) and logic (when M=0) operations. Figs. 9.9 and 9.10 illustrate the six arithmetic and eight logic operations of the bit-slice ALU.



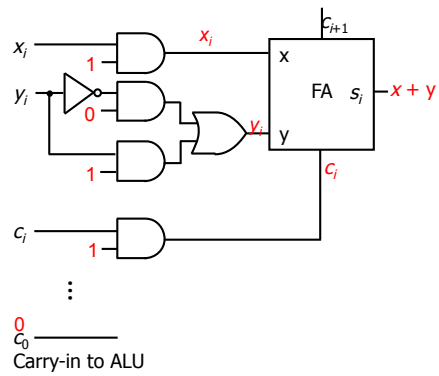
(a) $y + 1$



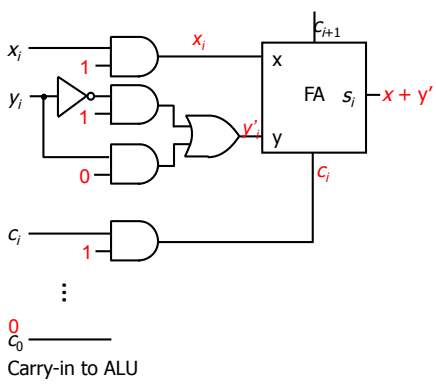
(b) $\bar{y} + 1$.



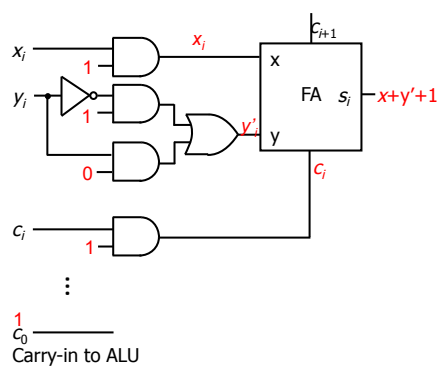
(c) $x + 1$



(d) $x + y$

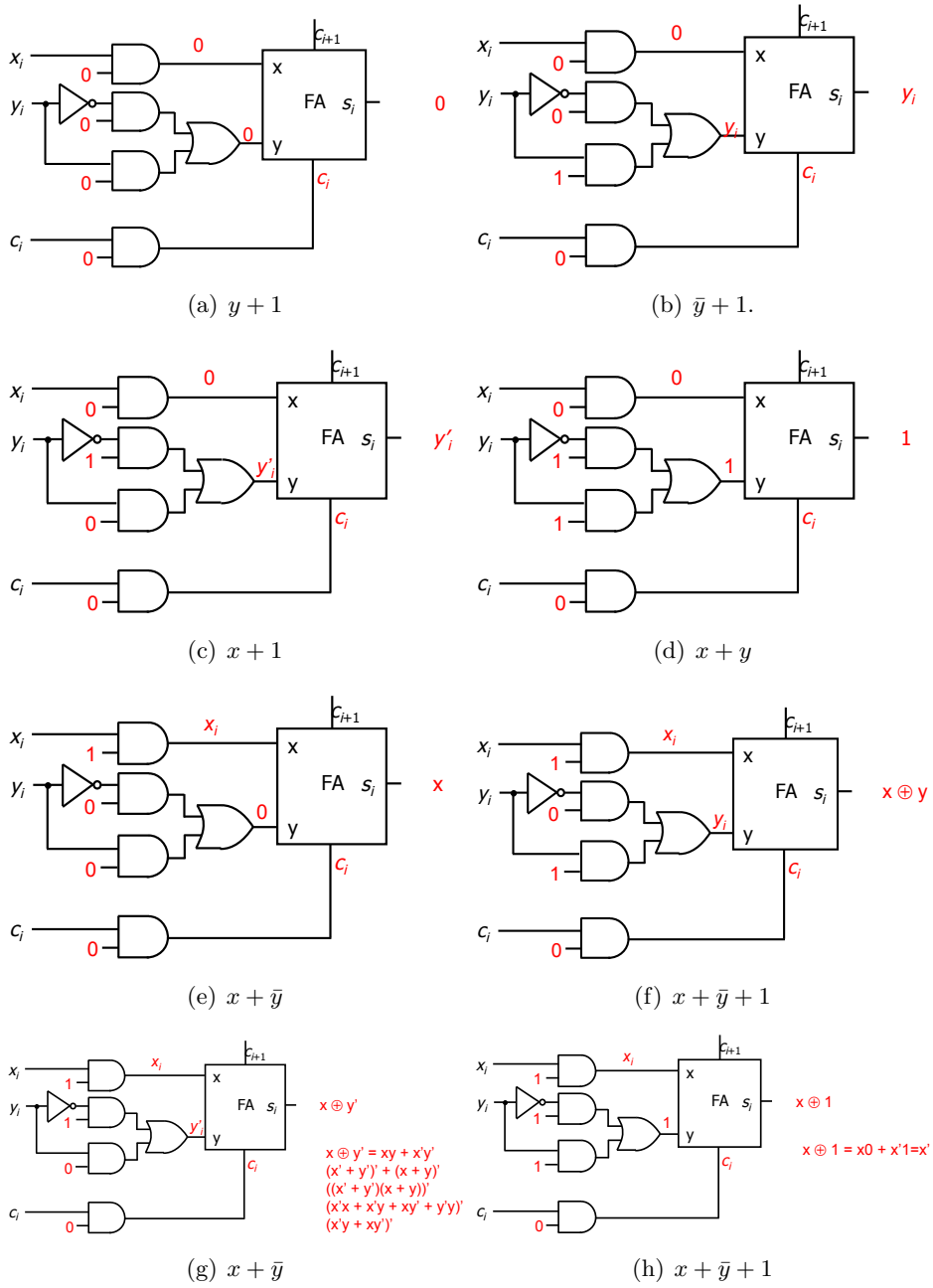


(e) $x + \bar{y}$



(f) $x + \bar{y} + 1$

Figure 9.9: Bit-slice arithmetic operations.



(a) $y + 1$ (b) $\bar{y} + 1$.

(c) $x + 1$ (d) $x + y$

(e) $x + \bar{y}$ (f) $x + \bar{y} + 1$

(g) $x + \bar{y}$ (h) $x + \bar{y} + 1$

Figure 9.10: Bit-slice logic operations.

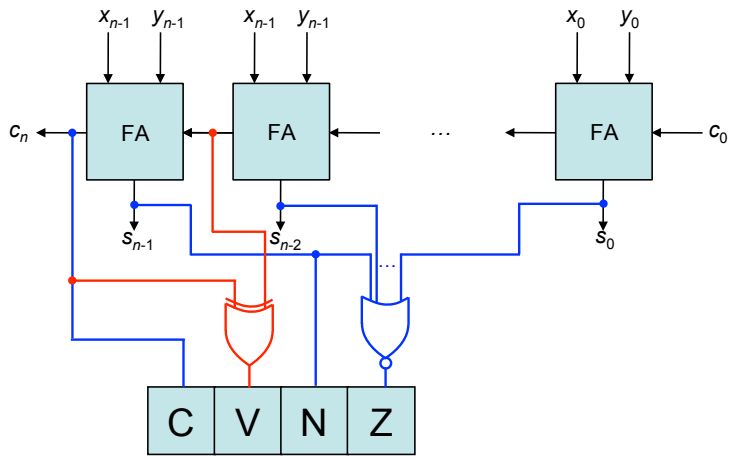


Figure 9.11: Setting condition codes

9.3.4 Condition Codes

Condition codes C (carry), V (overflow), N (negative), and Z (zero) indicate important status information after each arithmetic or shift operation. These flags dictate the outcome of a conditional branch instruction and thus transfer of control. For example, AVR Assembly instruction `bne` (*Branch If Not Equal*) tests the Z-flag and branches to a target address if it is cleared. In other words, if the last arithmetic instruction (e.g., subtract or compare) resulted in a non-zero result, the branch will be taken.

As can be seen from Fig. 9.11, the conditions for setting C, N, and Z flags are rather straightforward. The C-flag is simply set by the carry-out c_n from the $n-1^{th}$ stage. The N-flag indicates the result is negative and thus set by the s_{n-1} bit. The Z-flag indicates the result is zero and is implemented by logical NOR of all the bits of the result.

The V-flag indicates that the addition resulted in an overflow. Addition-s/subtractions using 2's-complement number systems have to follow a simple rule in the $(n-1)^{th}$-stage for the results to be valid: (1) if there is a carry-in, there must be a carry-out; and if there is no carry-in, there must be no carry-out. If this rule is violated then it is an overflow. Fig. 9.12 illustrates the two cases that cause overflow, where the first two bits and the third bit in the $(n-1)^{th}$-stage represent x_{n-1} and y_{n-1} and s_{n-1}, respectively. In Fig. 9.12(a), adding two positives numbers resulted in a negative number. This was cause by the fact that there was a carry-in but there was no carry-out. In Fig. 9.12(b), adding two negative numbers resulted in a positive

Carry-out	$(n-1)^{th}$ Stage	Carry-in
0	0	1
	0	
	1	

(a) Carry-in, no carry-out.

Carry-out	$(n-1)^{th}$ Stage	Carry-in
1	1	0
	1	
	0	

(b) No carry-in, Carry-out.

Figure 9.12: Two conditions that cause overflow.

number, which was cause by the fact that there was no carry-in but there was a carry-out. These two conditions can be detected by Exclusive-ORing the carry-in and carry-out of the $(n-1)^{th}$-stage.

9.3.5 Fast Adders

Since the implementation of an ALU is based on an adder/subtractor, it is important to perform additions fast. For example, 64-bit Ripple Carry Adder requires *128 gds*, which may not be completed in one clock cycle if a very fast clock is employed. Therefore, this section examines *Carry Lookahead Adders* (CLAs) that speed up additions.

In order to understand the concept behind Carry Lookahead Adders, let us re-examine the equations for a full adder:

$$s_i = x_i \oplus y_i \oplus c_i \text{ and}$$
$$c_{i+1} = x_i y_i + (x_i + y_i)c_i,$$

where $g_i = x_i y_i$ represents the *generation term* and $p_i = x_i + y_i$ represents the *propagation term*. g_i is called the generation term because when both x_i and y_i are 1s, a carry-out is generated. On the other hand, p_i is called the propagation term because a carry-in is propagated to the next stage when either x_i or y_i is 1. These two terms are known independently and in parallel for all n stages, and thus the carry-out term c_{i+1} can be rewritten as

$$c_{i+1} = g_i + p_i c_i. \tag{9.16}$$

Therefore, the following requirements can be obtained for the sum and carry-out equations for n=4 bits, where c_0 is the carry into the adder:

0th-stage:

$$\begin{aligned} s_0 &= x_0 \oplus y_0 \oplus c_0 \\ c_1 &= g_0 + p_0 c_0 \end{aligned}$$

1st-stage:

$$\begin{aligned} s_1 &= x_1 \oplus y_1 \oplus c_1 \\ c_2 &= g_1 + p_1 c_1 \\ c_2 &= g_1 + p_1(g_0 + p_0 c_0) \\ &= g_1 + p_1 g_0 + p_1 p_0 c_0 \end{aligned}$$

2nd-stage

$$\begin{aligned} s_2 &= x_2 \oplus y_2 \oplus c_2 \\ c_3 &= g_2 + p_2 c_2 \\ &= g_2 + p_2(g_1 + p_1 g_0 + p_1 p_0 c_0) \\ &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \end{aligned}$$

3rd-stage

$$\begin{aligned} s_3 &= x_3 \oplus y_3 \oplus c_3 \\ c_4 &= g_3 + p_3 c_3 \\ &= g_3 + p_3(g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0) \\ &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 \end{aligned}$$

These equations reveal something important about generation of carry-outs. That is, c_{i+1} is dependent only on g_i and p_i for $i = n-1, n-2, \cdots 0$. Since g_i's and p_i's can be determined independently and in parallel, the sum can also be determined independently and in parallel! Therefore, the above equations represent the implementation of a CLA.

Fig. 9.13 shows an n-bit CLA consisting of *Modified Full Adders* (MFAs) connected to a *Carry Lookahead Network*. Each MFA contains extra AND and OR gates to implement g_i and p_i, and the Carry Lookahead Network implements the carry-out equations (c_1, c_2, c_3, and c_4) shown above. The delay through the CLA is given as follows:

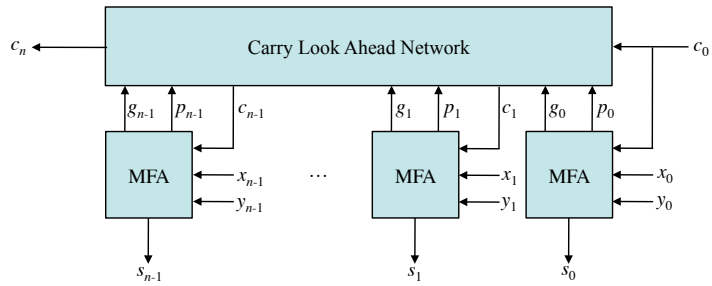


Figure 9.13: n-bit Carry Look-ahead Adder

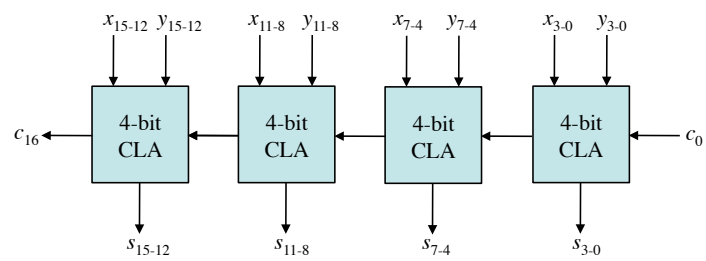


Figure 9.14: A cascaded CLA

1. 1 *gd* to generates g_i's and p_i's for $i = 0, 1, \cdots, n-1$;
2. 2 *gds* to generates c_i's for $i = 1, 2, \cdots, n$; and
3. 2 *gds* for s_i's for $i = 0, 1, \cdots, n-1$.

Therefore, the total delay for the CLA is of 5 *gds* and is, in theory, independent of n. However, in practice the number of inputs to the AND and OR gates is limited by fan-in of 4, and thus the largest CLA is 4 bits.

There are a couple of ways to implement a larger adders using 4-bit CLAs. The first method is to implement a *Cascaded CLAs*, which combines the concept of a Ripple Carry Adder with CLAs. Fig. 9.14 shows a 16-bit adder by cascading four 4-bit CLAs.

The delay through the cascaded CLAs based on m stages of 4-bit CLAs can be analyzed as follows:

1. 1 *gd* to generate g_is and p_is for $i = 0, 1, \cdots, n-1$;
2. 2 *gds* for each c_{4j} for $j = 1, 2, \cdots, m-1$;

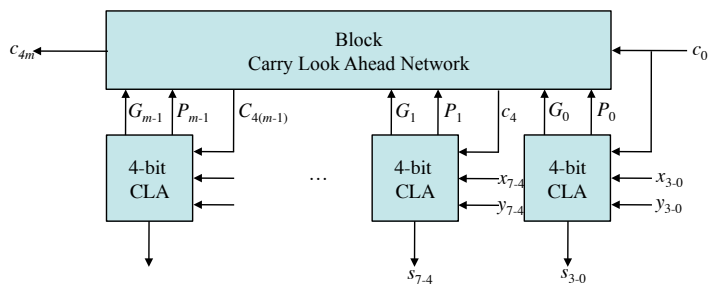


Figure 9.15: A block CLA

3. 2 *gds* to generate c_{4m}, c_{4m-1}, c_{4m-2}, and; c_{4m-3} and

4. 2 *gds* for s_{4m-1}, s_{4m-2}, s_{4m-3}, and s_{4m-4}.

In general, an adder with m CLA stages has a total delay of $3 + 2m$. For example, $m = 4$ results 11 *gds* compared to 32 *gds* for a Ripple Carry Adder.

Cascaded CLAs suffer from the same problem faced by Ripple Carry Adders, i.e., carry propagation. The alternative method is to implement *Block CLAs*, which involves looking at CLAs in a hierarchical fashion. Fig. 9.15 shows a block CLA, which consist of 4-bit CLAs connected by a *Block Carry Lookahead Network*.

The carry equations for a Block CLA is similar to a CLA, except they are evaluated in groups of fours. For example, consider the following equation for the carry-out of a 4-bit CLA, c_4:

$$c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0, \tag{9.17}$$

where

$$G_0 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 \tag{9.18}$$

is the *block generation* term and

$$P_0 = p_3p_2p_1p_0 \tag{9.19}$$

is the *block propagation* term. So Eq. 9.17 can be rewritten as

$$c_4 = G_0 + P_0c_0 \tag{9.20}$$

For the next stage, the equation can be written as

$$c_8 = G_1 + P_1c_4 = G_1 + P_1G_0 + P_1P_0c_0 \tag{9.21}$$

And for the next stage

$$c_{12} = G_2 + P_2 c_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0 \tag{9.22}$$

and so on. The c_4, c_8, c_{12}, and c_{16} terms are generated by a *Block Carry Lookahead Network*, which has the exact same logic requirement as the Carry Lookahead Network in Fig. 9.15 except its input are G_i's and P_i's rather than g_i's and p_i's.

The delay through the Block CLA is given as follows:

1. 1 *gd* to generate g_i's and p_i's for $i = 0, 1, \cdots, n-1$
2. 2 *gds* for G_i's and P_i's for $j = 0, 1, \cdots, m-1$
3. 2 *gds* for c_4, c_8, c_{12}, c_{16} , $\cdots$
4. 2 *gds* for c_{3-1}, c_{7-5}, c_{11-9}, c_{15-13}
5. 2 *gds* for s_i's for $i = 0, 1, \cdots, n-1$

Therefore, the total delay for the Block CLA is 9 *gds*. But again, due to the fan-in problem, m is limited to 4.

Based on the aforementioned discussions, a large adder can be built using Block CLAs and a 'Super-block' Carry Lookahead Network. For example, a 64-bit adder can be implemented using four 16-bit Block CLAs. We leave the delay analysis for such an adder as an exercise.

9.4 Multiplication

9.4.1 Add-and-Shift

Multiplication involves a series of add and shift operations. The following example illustrates this process using the traditional paper-and-pencil method.

Example 9.12. *Illustration of multiplication using the paper-and-pencil method.*

						0	1	0	1	1	1	(23) Multiplicand
					×	0	1	0	0	1	1	(19) Multiplier
						0	1	0	1	1	1	
					0	1	0	1	1	1		
				0	0	0	0	0	0			Partial
			0	0	0	0	0	0				Products
		0	1	0	1	1	1					
+	0	0	0	0	0	0						
	0	0	1	1	0	1	1	0	1	0	1	(437) Result

The top operand is referred to as the *multiplicand*, while the bottom operand is called the *multipliplier*. It is usually not necessary to distinguish between multiplicand and multiplier due to the commutative property of multiplication. However, we make this distinction to clarify the different parts of the data handled a multiplier hardware.

The multiplication starts by examining the least significant (0th) bit of the multiplier, and generates a *partial product*, which can be either the multiplier or zero depending on wether the bit is 1 or 0, respectively. In the next step, the 1st bit is examined and the partial product shifted left by one bit is generated. This process repeats for all n bits. The partial products are then added to generated the final result.

There are many ways to implement a multiplier. The simplest method is to implement it as an Add-and-Shift multiplier as shown in Fig. 9.16, which consists of an n-bit adder (see Sec. 9.3), n-bit registers MULT, MPY, and ACC, and a Carry (C) bit. Note that our discussion is based on hardware implementation; however, the concept of an Add-and-Shift multiplier can also be implemented in software, which will be required for Instruction Set Architectures (ISAs) that do not provide multiply instructions.

The multiplication is initialized by latching the multiplicand onto MULT, the multiplier onto MPY, and clearing ACC and the C-bit. The concatenation of the ACC and MPY registers represent the accumulation of the partial products. Moreover, the combination of C-bit, ACC, and MPY operates as a shift-right register.

As the name suggest, an $n \times n$ multiplication consists of n steps, and each step consists of (1) an add operation and (2) a shift operation. The add operation involves ANDing the least significant bit of MPY with the multiplicand in MULT, which is then fed to the n-bit adder. The result and the carry-out, which represents a partial product, is then latched onto ACC and the C-bit. Then, the shift operation involves taking the contents of C, ACC, and MPY registers and shift to the right by one bit. This step

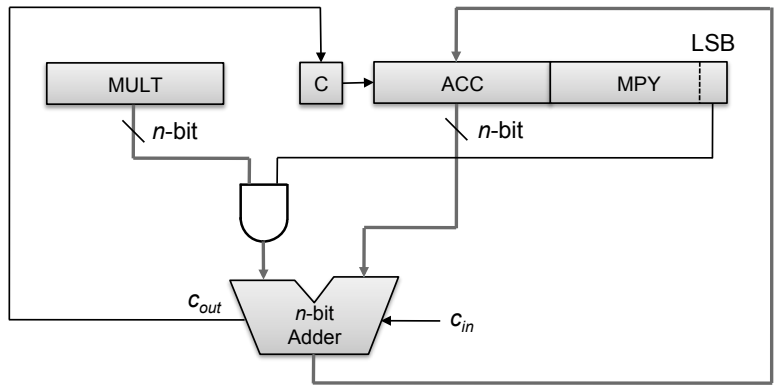


Figure 9.16: Hardware implementation of Add-and-Shift multiplier.

is repeated for n bits until the $2n$-bit result appears on the ACC and MPY registers.

The following example illustrates the Add-and-Shift multiplication:

Example 9.13. *Add-and-Shift multiplication of 10111×10011 (23×19) using hardware implementation. Note that the numbers are unsigned.*

MULT		C	ACC	MPY	
1 0 1 1 1		0	0 0 0 0 0	1 0 0 1 1	
	+	0	1 0 1 1 1		add+shift
		0	1 0 1 1 1		
		0	0 1 0 1 1	1 1 0 0 1	
	+	0	1 0 1 1 1		add+shift
		1	0 0 0 1 0		
		0	1 0 0 0 1	0 1 1 0 0	
					shift
		0	0 1 0 0 0	1 0 1 1 0	
					shift
		0	0 0 1 0 0	0 1 0 1 1	
	+	0	1 0 1 1 1		add+shift
		0	1 1 0 1 1		
			0 1 1 0 1	**1 0 1 0 1**	
			Final result		

In the first step, the least significant bit of MPY is one and thus represents add followed by a shift. This is achieved by ANDing the least significant bit with the multiplicand in MULT and then feeding it to the n-bit adder. The result is latched to ACC and the C-bit. Then, the content of C, ACC, and MPY are all shifted right by one bit. Note that this is a logical shift right (see Sec. 9.2.1) operation, meaning a '0' is shifted into the C-bit and

the least significant bit of MPY is shifted out. The second step also involves adding the multiplicand to the ACC since the least significant bit is one. At this point, the purpose of the C-bit becomes clear. That is, the C-bit latches the carry-out from the addition since an n-bit addition can generate at most $n+1$-bit result. Thus, the carry-out becomes a part of the partial product when the shift-right operation is performed. In the third step, the least significant bit is one, thus zero is added to ACC followed by a shift-right operation. As can be seen, this process continues for n steps until the $2n$-bit result appears on the ACC and MPY registers.

Our discussion of multiplication thus far has been based on unsigned numbers. Signed numbers are little more tricky, especially when the multiplier is negative. The following example illustrates multiplication of a negative multiplicand and a positive multiplier.

Example 9.14. *Illustration of multiplication of a negative multiplicand and a positive multiplier.*

				1	0	0	1	(-7) Multiplicand
			×	0	1	0	1	(5) Multiplier
1	1	1	1	1	0	0	1	
0	0	0	0	0	0	0		
1	1	1	0	0	1			
0	0	0	0	0				
1	1	0	1	1	1	0	1	(-35) Result

In order to handle negative multiplicands, each partial product is sign-extended. For example, the negative multiplicand 1001 (-7) is sign extended to 1..11001, which is also -7, and added as partial products. This can be implemented in the Add-and-Shift multiplier shown in Fig. 9.16 by performing *arithmetic shift right*,operations rather than logical shift right, of the C-bit, ACC, and MPY combination to maintain the signs of the partial products.

When the multiplier is negative, we have no choice but to covert it to a positive number, perform the multiplication, and then convert the sign of the result. This is because 0s and 1s in 2's-complement representation of a multiplier do not have the same meaning as in say 1's-complement, where 0 means multiply by 1 and 1 means multiply by 0. In the next section, we discuss how we can handle negative multipliers by using a method called *Booth Recoding*.

9.4.2 Booth Algorithm

The basic idea behind Booth recoding is *redundant codes*. Binary redundant codes contain the following digit set $\{\bar{1}, 0, 1\}$, where $\bar{1}$ represents -1. As the name suggests, redundant codes allow a number to be represented with different numbers. For example, a binary number 0111 (7) can also be represented by $100\bar{1}$ (7), i.e., $2^3 + (-1)2^0 = 7$. The following example illustrates the transformation from 0111 to $100\bar{1}$.

Example 9.15. *A transformation of a positive binary number to its redundant code using the digit set* $\{\bar{1}, 0, 1\}$.

0	1	1	1	(7)	Binary number
		+1	-2		
1	0	0	$\bar{1}$	(7)	Redundant code

The basic idea behind Booth recoding is to transform k consecutive 1s into $\bar{1}$ in its 0th bit position and 1 in the kth bit position by adding -2 to the 0th bit position and +1 to the 1st bit position. The net result is -2 + 2, which does change the original value, but in the process generates a redundant code. In the above example, there are three consecutive 1's starting at the 0th bit position and ending at the 2nd bit position. By adding -2 to the 0th bit position and +1 to the 1st bit position, we generate $\bar{1}$ in the 0th bit position and 1 in the 3rd bit position.

Similarly, Booth recoding can also be used to transform k consecutive $\bar{1}$s into 1 in its 0th bit position and $\bar{1}$ in the kth bit position by adding 2 to the 0th bit position and -1 to the 1st bit position.

The major advantage of Booth recoding is that it also works on negative numbers. The following example illustrates the transformation from 101110 (-18) to $0\bar{1}00\bar{1}0$ (-18):

Example 9.16. *A transformation of a negative binary number to its redundant code using the digit set* $\{\bar{1}, 0, 1\}$.

1	0	1	1	1	0	(-18)	Binary number
			+1	-2			
1	1	0	0	$\bar{1}$	0		
+1	-2						
0	$\bar{1}$	0	0	$\bar{1}$	0	(-18)	Redundant code
-1	+2						
$\bar{1}$	1	0	0	$\bar{1}$	0	(-18)	Redundant code

In this example, the first partial transformation occurs on bits 3-1, which results in $\bar{1}$ in the 1st bit position and 1 in the 4th bit position. The second partial transformation is performed on bits 5 and 4 resulting in $\bar{1}$ in the 4th bit position and the carry-out from the 5th bit position is ignored since the numbers are represented in 2's-complement. This generates the equal to $(-1)2^4 + (-1)2^1 = -18$. The third partial transformation generates another redundant code, which is also equal to the original value $(-1)2^5 + 2^4 + (-1)2^1 = -18$.

Another way to look at the the Booth recoding process is to examine two bits at a time, i.e., bit i and bit $i-1$, starting from the least significant bit to the most significant bit. For example, let us re-examine the original number 101110 and the recoded number $\bar{1}100\bar{1}0$ in Example 9.16. As the bits in the original number are scanned from right to left, each pair of bits have a meaning. The scanning start by looking at bit 0 together with bit -1, which is referred to as the *pad bit* initialized to 0. Since the pair of bits is 00, it means we are in the middle of 0s and thus is recoded as 0. The 1st and 0th bits are 10, which means we started to run into a string of 1's. Thus, they are recoded as $\bar{1}$. The next pair of bits, i.e., bit 2 and bit 1, is 11 indicating that we are in the middle of 1's. This is recoded as 0. The same is true for bits 3 and 2. Bits 4 and 3 are 01, which means we stop running into 1's, and thus recoded as 1. Finally,10 in bits 5 and 4 indicate that we started to run into a string of 1's and is recoded as $\bar{1}$.

Based on this discussion, Table 9.1 shows the Booth recoding table. When the pair of bits scanned is either 00 or 11, addition/subtraction of the multiplicand is not needed, and the partial product is simply shifted. When the scanned bits are 01, they represent the recoded bit 1, and thus the multiplicand is added and then the partial product is shifted. Finally, the pair of bits 10 is recoded as $\bar{1}$, and thus the multiplicand is subtracted and then the partial product is shifted.

Table 9.1: Booth recoding table.

Bit i	Bit $i-1$	Operation	Description
0	0	$0 \times M$	Shift
0	1	$+1 \times M$	Add and shift
1	0	$-1 \times M$	Subtract and shift
1	1	$0 \times M$	Shift

The following example illustrates how the Booth recoding table is applied to perform multiplication with a negative multiplier.

Example 9.17. *An example binary multiplication using Booth recoding.*

$$
\begin{array}{cccccc}
 & 1 & 1 & 0 & 1 & \\
\times & 1 & 0 & 1 & 1 & \mathbf{0} \\
\hline
\end{array}
\quad \Rightarrow \quad
\begin{array}{ccccccccc}
 & & & 1 & 1 & 0 & 1 & & (-3) \\
 & & \times & \bar{1} & 1 & 0 & \bar{1} & & (-5) \\
\hline
 & & & 0 & 0 & 1 & 1 & & \\
 & & 0 & 0 & 0 & 0 & & & \\
1 & 1 & 1 & 0 & 1 & & & & \\
0 & 0 & 1 & 1 & & & & & \\
\hline
0 & 0 & 0 & 1 & 1 & 1 & 1 & & (15)
\end{array}
$$

The two numbers on the left are the original values to be multiplied. Since the Booth recoding requires examining two bits at a time, a pad bit of zero (indicated in bold) is appended to the right of the multiplier. The first pair of bits scanned is 10, which is recoded as $\bar{1}$ and a subtract and a shift are performed. The second pair of bits examined is 11, which is recoded as 0 resulting in a simple shift. The third pair of bits is 01, which is recoded as 1 and an add and a shift are performed. Finally, the fourth pair of bits is 10, which means $\bar{1}$ and a subtract and a shift are performed. The recoded multiplier and its result is shown on the right. Note that the third partial product, which is negative, is sign extended to maintain the meaning of the number.

The hardware implementation of the Booth recoding only requires a slight modification to the Add-and-Shift multiplier shown in Fig. 9.16, which are a pad bit and additional logic to implement Table 9.1. The following example illustrates the hardware implementation of 4-bit×4-bit Booth recoding multiplication.

Example 9.18. *Hardware implementation of 1101×1011 (-3×-5) using the Booth algorithm.*

MULT (-3)					C		ACC					MPY (-5)				Pad
1	1	0	1		0		0	0	0	0		1	0	1	$\underline{1}$	$\underline{0}$
				+	0		0	0	1	1					subtract+shift	
					0		0	0	1	1						
					0		0	0	0	1		1	1	0	$\underline{1}$	$\underline{1}$
															shift	
					0		0	0	0	0		1	1	1	$\underline{0}$	$\underline{1}$
				+	1		1	1	0	1					add+shift	
					1		1	1	0	1						
					1		1	1	1	0		1	1	1	$\underline{1}$	$\underline{0}$
				+	0		0	0	1	1					sub+shift	
					0		0	0	0	1						
					0		**0**	**0**	**0**	**0**		**1**	**1**	**1**	**1**	
							Final result									

The least significant bit of MPY and the pad bit represent the pair of bits scanned at each step. Moreover, when the partial product is shifted right, the least significant bit of MPY is shifted into the pad bit. Note that in the third step, the multiplicand is signed extended, i.e., the C-bit is equal to 1, and added to ACC. This generates a negative partial product, which is then shifted using arithmetic shift right to maintain the sign.

9.4.3 Modified Booth Algorithm (Bit-pair recoding)

The Booth algorithm handles both positive and negative numbers, and in general, reduces a long string of 1's to a pair 1 and $\bar{1}$. However, when the length of a string of 1's is less than or equal to two, the number of 1's or $\bar{1}$'s generated will be larger than the original number of 1's. Consider the following example of applying Booth recoding on a number with alternating 1's and 0's.

$$\begin{array}{cccccccl}
0 & 1 & 0 & 1 & 0 & 1 & (42) & \text{Binary number} \\
 & & & \Downarrow & & & & \\
1 & \bar{1} & 1 & \bar{1} & 1 & \bar{1} & (42) & \text{Redundant code}
\end{array}$$

The recoded number has more 1's than the original number leading to slower multiplication. The *Modified Booth algorithm*, also known as *Bit-pair Recoding*, eliminates this problem by looking three bits at a time (two bits $i+1$ and i, and the previous bit $i-1$) and removing redundancies in the original Booth recoding. Moreover, since the algorithm looks at two bits (plus the previous bit) at a time, an n-bit multiplication generates at most $n/2$ partial products.

The basic idea behind bit-pair recoding is to eliminate the redundancies in the Booth algorithm as illustrated below:

$$\begin{array}{cccccccl}
0 & 1 & 0 & 1 & 0 & 1 & & \\
1 & \bar{1} & & \bar{1} & 1 & & & \Rightarrow \text{Booth recoding} \\
2M & -M & & -2M & M & & & \\
 & M & & & -M & & & \text{Net effect}
\end{array}$$

In the above example, alternating 0's and 1's (either 010 or 101) causes both recoded bits to be non-zero, and thus requires add/subtract operation for both steps. However, when these two operations (either $1\bar{1}$ or $\bar{1}1$) are considered together, the net effect is a much simpler operation. For example, the recoded bits $1\bar{1}$ require adding the multiplicand M followed by adding $2M$. However, when these two operations are combined, they result in

$2M - M = M$. Similarly, the recoded bits $\bar{1}1$ is equivalent to $-2M + M = -M$. Therefore, the bit patterns 010 and 101 result in add and subtract, respectively. Similar analysis can be done on all the bit patterns to generate bit-pair recoding shown in Table 9.2.

Table 9.2: Bit-pair recoding table.

Bit $i+1$	Bit i	Bit $i-1$	Operation	Description
0	0	0	$0 \times M$	2×Shift
0	0	1	$+1 \times M$	Add & 2×Shift
0	1	0	$+1 \times M$	Add & 2×Shift
0	1	1	$+2 \times M$	2×Add & 2×Shift
1	0	0	$-2 \times M$	2×Subtract & 2×Shift
1	0	1	$-1 \times M$	Subtract & 2×Shift
1	1	0	$-1 \times M$	Subtract & 2×Shift
1	1	1	$0 \times M$	2×Shift

The following example illustrates the bit-pair recoding for multiplication and its relationship to the Booth algorithm.

Example 9.19. *Relationship between Booth algorithm and bit-pair recoding*

							1	1	1	1	0	1		(-3)
					×		0	1	1	1	0	1	0	(+29)
							1	0	0	1	1	$\bar{1}$		Booth Algorithm
								+2		-1		+1		Bit-pair recoding
1	1	1	1	1	1	1	1	1	1	1	0	1		
0	0	0	0	0	0	0	0	0	1	1				
1	1	1	1	1	1	0	1	0						
1	1	1	1	1	1	0	1	0	1	0	0	1		(-87)

Multiplication based on bit-pair recoding looks at two bits plus the previous bit to determine the operation. Since two bits are examined at a time, the shifting of the partial product is a two-bit shift rather than an one bit shift. In addition, $\pm 2 \times M$ operation involves shifting M by one bit to the left. Most importantly, at most $n/2$ partial products are generated by eliminating the redundancies in the Booth algorithm.

Example 9.20. *Hardware implementation of 111101×011101 (-3×29) using bit-pair recoding.*

MULT (-3)							C	ACC						Mpy (+29)						Pad
1	1	1	1	0	1		0	0	0	0	0	0	0	0	1	1	1	$\underline{0}$	$\underline{1}$	$\underline{0}$
						+	1	1	1	1	1	1	1					add+2shift		
							1	1	1	1	1	0	1							
							1	1	1	1	1	1	1	0	1	0	1	$\underline{1}$	$\underline{1}$	$\underline{0}$
						+	0	0	0	0	0	1	1					sub+2shift		
							0	0	0	0	0	1	0							
							0	0	0	0	0	0	0	1	0	0	1	$\underline{0}$	$\underline{1}$	$\underline{1}$
						+	1	1	1	1	0	1	0					2add+2shift		
							1	1	1	1	0	1	0							
							1	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	
								Final result (-87)												

The two least significant bits of MPY and the pad bit represent the three bits scanned at each step. Note that the partial product is shifted two bits to the right.

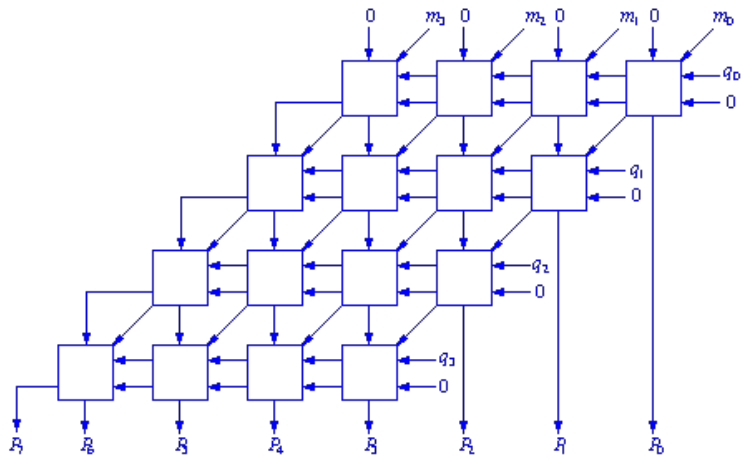
9.4.4 Array Multipliers

Array multipliers consist of identical cells, each forming a new partial product and adding it to previously accumulated partial product. Array multipliers provide a number of advantages over add-and-shift multipliers. First, it is conceptually simple and easily expandable due to its array structure and thus suitable for VLI implementation and pipelining. Second, it is faster since partial product generation and summation are performed together.

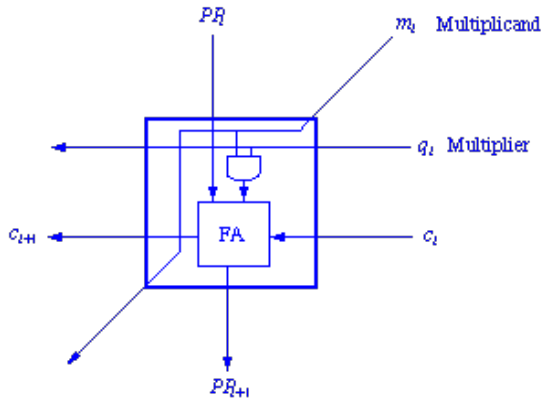
Fig. 9.17(a) shows a 4×4 multiplier, where its input consist of multiplicand $m_3m_2m_1m_0$, multiplier $m_3m_2m_1m_0$, and carry-ins all initialized to 0's. Each cell shown in Fig. 9.17(b) consists of a full-adder and a AND gate. The inputs to the full adder are the partial product bit pp_i from the previous row of cells, the multiplicand bit m_i ANDed together with the multiplier bit q_j, and carry-in c_i from the from the previous cell. In addition, the partial product bits are provided to all the rows, while the multiplier bits are provided to the columns.

9.4.5 Implementing Large Multiplication using Smaller Multiplications

A large multiplier can be implemented using a number of smaller multipliers. The advantages of this method are that each smaller multiplier can operate faster than one large multiplier and all the multipliers can proceed in parallel, and thus overall delay required to generate the partial products is reduced. Moreover, this method can be combined with other fast multiplication techniques discussed in this section to further reduce delay.



(a) 4×4 Array multiplier



(b) A cell.

Figure 9.17: Array multiplier.

Fig. 9.18 illustrates a 4×4 multiplication consisting of multiplicand a, multiplier b, and final product p. For each partial product bit $a_i b_j$, the sum of its subscripts i and j represents the bit position in the final product, i.e., p_{i+j}. Fig. 9.18(b) shows the multiplication in Fig. 9.18(a) subdivided into four separate 2×2 multipliers. Each of these multiplications generates a partial product $\hat{p}_k$, where the subscript k of its least significant bit is $i + j$, where i and j represent subscripts of the least significant bits of the subdivided multiplicand and multiplier. Finally, the four partial products

$$
\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & \times & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & a_3b_0 & a_2b_0 & a_2b_0 & a_0b_0 \\
 & & & a_3b_1 & a_2b_1 & a_2b_1 & a_0b_1 & \\
 & & a_3b_2 & a_2b_2 & a_2b_2 & a_0b_2 & & \\
 & a_3b_3 & a_2b_3 & a_2b_3 & a_0b_3 & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

(a) 4×4 multiplication.

$$
\begin{array}{cccc}
 & & a_1 & a_0 \\
 & \times & b_1 & b_0 \\
\hline
 & & a_1b_0 & a_0b_0 \\
 & a_1b_1 & a_0b_1 & \\
\hline
\hat{p}_3 & \hat{p}_2 & \hat{p}_1 & \hat{p}_0
\end{array}
\qquad
\begin{array}{cccc}
 & & a_3 & a_2 \\
 & \times & b_1 & b_0 \\
\hline
 & & a_3b_0 & a_2b_0 \\
 & a_3b_1 & a_2b_1 & \\
\hline
\hat{p}_5 & \hat{p}_4 & \hat{p}_3 & \hat{p}_2
\end{array}
$$

$$
\begin{array}{cccc}
 & & a_1 & a_0 \\
 & \times & b_3 & b_2 \\
\hline
 & & a_1b_2 & a_0b_2 \\
 & a_1b_3 & a_0b_3 & \\
\hline
\hat{p}_5 & \hat{p}_4 & \hat{p}_3 & \hat{p}_2
\end{array}
\qquad
\begin{array}{cccc}
 & & a_3 & a_2 \\
 & \times & b_3 & b_2 \\
\hline
 & \times & a_3b_2 & a_2b_2 \\
 & a_3b_3 & a_2b_3 & \\
\hline
\hat{p}_7 & \hat{p}_6 & \hat{p}_5 & \hat{p}_4
\end{array}
$$

(b) 4×4 multiplication subdivided into four parallel 2×2 multiplications.

$$
\begin{array}{cccccccc}
 & & & & \hat{p}_3 & \hat{p}_2 & \hat{p}_1 & \hat{p}_0 \\
 & & \hat{p}_5 & \hat{p}_4 & \hat{p}_3 & \hat{p}_2 & & \\
 & & \hat{p}_5 & \hat{p}_4 & \hat{p}_3 & \hat{p}_2 & & \\
\hat{p}_7 & \hat{p}_6 & \hat{p}_5 & \hat{p}_4 & & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

(c) Alignment of partial products.

Figure 9.18: A large multiplier implemented with small multipliers

are aligned and added to generate the final result as shown in Fig. 9.18(c).

9.4.6 Partial Product Reduction (Reduction of Summands)

{(BL): Work in progress...}

9.5 Division

Division involves successive compare, shift, and subtract operations. In order understand binary division, let us revisit decimal division based on the following example using the pencil-and-paper method:

Example 9.21. $118 \div 8$.

```
        1  4    Quotient
8 | 1  1  8
    -  8
       3  8
    -  3  2
          6    Remainder
```

As simple as this example may be, it illustrates the process of shift, compare, and subtract operations for division. For example, the division process starts by examining the dividend one digit at a time from right to left. In the first step, the first digit cannot be divided by the divisor; therefore; no quotient digit is generated and the next digit is examined together with the current digit. In the second step, 11 divided by 8 generates 1 as the quotient and 3 as the remainder. In the third step, the remainder 3 together with the next digit 8 is divided by the divisor to generate 4 as the quotient and 6 as the remainder. Thus, the final answer is 14 as the quotient and 6 as the remainder.

Performing binary division is identical to that of the decimal division except that binary subtraction is used. The following example illustrates the binary division of $01110110 \div 000010000$, which is equivalent to $118 \div 8$.

Example 9.22. *Binary division of* $01110110 \div 000010000$ *using the paper-and-pencil method.*

```
                                  1  1  1  0    Quotient
0 0 0 0 1 0 0 0 | 0  1  1  1  0  1  1  0
                - 0  1  0  0  0
                     0  1  1  0  1
                  - 0  1  0  0  0
                       0  1  0  1  1
                     - 0  1  0  0  0
                          0  0  1  1  0
                        - 0  0  0  0  0
                             0  1  1  0    Remainder
```

The above example shows that binary division is basically identical to decimal division. However, a hardware or a software implementation of division has to perform certain operations that we as human beings are smart enough to avoid. For example, we can easily see that the last step in the above example does not require a subtraction because 01000 is larger than 00110.

However, a hardware or a software implementation requires the divisor to be subtracted from the partial remainder and if the result is positive then the quotient bit is 1; otherwise the quotient bit is 0.

There are two basic approaches for binary division: *restoring* and *non-restoring*. We will examine both methods in the following two subsections. Moreover, a division can be perform using either n-by-n or n-by-$n/2$. An n-by-n division generates an n-bit quotient and an n-bit remainder and requires n steps. On the other hand, an n-by-$n/2$ division generates an $n/2$-bit quotient and an $n/2$-bit remainder and requires only $n/2$ steps. This distinction is important when divisions have to implemented in software. For example, the AVR processor does not provide a divide instruction, and thus being able to perform n-by-$n/2$ division when possible significantly speeds up its operation.

In order to decide whether to use n-by-n or n-by-$n/2$, we simply subtract the divisor from the higher order bits of the dividend. If the result is negative we know the quotient will not overflow using $n/2$ bits. The following example illustrates this process:

Example 9.23. *Checking whether n-by-$n/2$ division can be used.*

					0	x	x	x	x	x	Quotient will fit in $n/2$ bits
	0	0	0	1	1	1	0	1	1	0	(118)
-	0	1	0	0	0						(8)
					1	x	x	x	x	x	Will need n bits
	0	0	0	1	1	1	0	1	1	0	(118)
-	0	0	0	1	1						(3)

9.5.1 Restoring

As the name suggests, restoring division restores the partial remainder to a positive number after a subtraction. The following example illustrates $118 \div 8$ using 12-bit by 6-bit division.

Example 9.24. *Dividend = 0001110110 (118) and Divisor = 01000 (8) using n-by-$n/2$ restoring division.*

		C	ACC					DIVIDEND					
		0	0	0	0	1	1	1	0	1	1	0	
Shift		0	0	0	1	1	1	0	1	1	0	X	
Subtract	+	1	1	1	0	0	0						
		1	1	1	1	1	1						
Restore		0	0	0	1	1	1	0	1	1	0	0	Set $q_0 = 0$
Shift		0	0	1	1	1	0	1	1	0	0	X	
Subtract	+	1	1	1	0	0	0						
		0	0	0	1	1	0						
		0	0	0	1	1	0	1	1	0	0	1	Set $q_0 = 1$
Shift		0	0	1	1	0	1	1	0	0	1	X	
Subtract	+	1	1	1	0	0	0						
		0	0	0	1	0	1						
		0	0	0	1	0	1	1	0	0	1	1	Set $q_0 = 1$
shift		0	0	1	0	1	1	0	0	1	1	X	
Subtract	+	1	1	1	0	0	0						
		0	0	0	0	1	1						
		0	0	0	0	1	1	0	0	1	1	1	Set $q_0 = 1$
Shift		0	0	0	1	1	0	0	1	1	1	X	
Subtract	+	1	1	1	0	0	0						
		1	1	1	1	1	0						
Restore		0	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	Set $q_0 = 0$
			Remainder					Quotient					

For an n-by-$n/2$ division, the upper and lower bits of the dividend are stored in ACC and DIVIDEND, respectively. At the beginning of each step, C-bit, ACC, and DIVIDEND together are shift left by one bit. After the shift, the least significant bit indicated by 'X' is a don't care and will be replaced by a quotient bit at the end of the step. The divisor is subtracted from the upper bits of the dividend in ACC by taking the 2's-complement and adding it to ACC. If the subtraction results in a positive partial remainder, which is indicate by the C-bit, then the subtraction was successful and a quotient bit is set to 1; otherwise, the quotient bit is set to 0. In addition to generating a quotient bit, if the partial remainder is negative, it has to be restored to its positive form. This is done by adding the divisor back to the negative partial remainder. This process repeats for $n/2$ steps, and the final quotient and remainder will be contained in DIVIDEND and ACC, respectively.

9.5.2 Non-Restoring

The key to non-restoring division is to allow both negative as well as positive partial remainders. This way the absolute value of the partial remainder is reduced every cycle by adding or subtracting the divisor from it. To

understand why this works, suppose A represents a partial remainder and B represents a divisor. During the division steps, if the partial remainder is positive, then the partial remainder is shifted left by one bit and then the divisor is subtracted. This can be represented by the following equation:

$$2A - B \tag{9.23}$$

On the other hand, if the partial remainder is negative, it is restored, shifted left, and then the divisor is subtracted. This is represented by

$$2(A + B) - B = 2A + B \tag{9.24}$$

The above two equations show that when the partial remainder is positive then, we perform shift the partial remainder and subtract the divisor. On the other hand, if the partial remainder is negative, we simply shift the partial remainder and add the divisor.

The following example illustrates a 5-bit by 5-bit non-restoring division.

Example 9.25. *Dividend = 10011 (19) and Divisor = 00011 (3) using n-by-n non-restoring division.*

		C	ACC					DIVIDEND					
		0	0	0	0	0	0	1	0	0	1	1	
Shift		0	0	0	0	0	1	0	0	1	1	X	
Subtract	+	1	1	1	1	0	1						
		1	1	1	1	1	0	0	0	1	1	0	Set $q_0 = 0$
Shift		1	1	1	1	0	0	0	1	1	0	X	
Add	+	0	0	0	0	1	1						
		1	1	1	1	1	1	0	1	1	0	0	Set $q_0 = 0$
Shift		1	1	1	1	1	0	1	1	0	0	X	
Add	+	0	0	0	0	1	1						
		0	0	0	0	0	1	1	1	0	0	1	Set $q_0 = 1$
shift		0	0	0	0	1	1	1	0	0	1	X	
Subtract	+	1	1	1	1	0	1						
		0	0	0	0	0	0	1	0	0	1	1	Set $q_0 = 1$
Shift		0	0	0	0	0	1	0	0	1	1	X	
Subtract	+	1	1	1	1	0	1						
		1	1	1	1	1	0	0	0	1	1	0	Set $q_0 = 0$
Restore		0	0	0	0	0	1	0	0	1	1	0	Result
			Remainder					Quotient					

For an n-by-n division, the dividend is stored in DIVIDEND and ACC is set to 0s. When the partial remainder after a subtraction is negative, this number is not restored, and instead the partial remainder is maintained and

in the following step is added with the divisor. Note that because non-restoring division can have both positive and negative partial remainders, the very last step may result in a negative partial remainder, which will have to be restored.

9.6 Floating-Point Numbers

The discussion up to now has been exclusively on signed, fixed-point numbers. This section presents floating-point numbers and their operations.

High-level languages support both fixed-point (i.e., data types `int`, `long int`, `unsigned int`, and `unsigned long int`) and floating-pointer numbers (i.e., data types `float` and `double`). Unlike fixed-point numbers that can only represent unsigned and signed integers, floating-point numbers represent *real* numbers and can support a much wider range of values. Floating-point numbers are in general represented as

$$m \times r^e, \tag{9.25}$$

where m, r, and e represent the mantissa, radix, and exponent.

The *mantissa* is a typically fraction and, as with integers, can be represented in 1's-complement, 2's-complement, or sign-magnitude. However, based on the discussion presented in Sec. 9.1, 2's-complement representation is used. The *radix* represents the base for the number and can be base 2, 4, 8, 16, or 10. The range increases as the radix increases, while the precision increases as radix decreases. The *exponent* is an integer that raises the mantissa to base-r power. Floating-point representation trades off between the size of the mantissa and the size of the exponent. That is, increasing the number of bits for the mantissa improves the *precision* of the fraction, while increasing the number of bits for the exponent increases the *range* of the numbers that can be represented. In the follow subsection. we discuss the IEEE 754 Floating-Point format, which is the de facto standard for representing real numbers in computer systems.

9.6.1 IEEE 754 Floating-Point Standard

Figure 9.19 shows the format of a single-precision IEEE 754 Floating-Point Format.

The numbers represented in this format is equivalent to

$$(-1)^S \times (1 + Fraction) \times 2^{(Exponent - Bias)}. \tag{9.26}$$

Figure 9.19: Single-precision IEEE 754 floating-point format.

The S-bit represents the sign of the number. The *exponent* has a bias or excess of 127 (referred to as excess-127), and thus the 127 or 01111111_2 in the Exponent field represents 0, and its range is represented by

$$-127 \leq Exponent \leq 128 \tag{9.27}$$

Using the excess notation, rather than say 2's-complement, is useful because it allows to a simple comparator to determine the size of the exponents.

The *fraction* consists of 23 bits and assumes a radix point immediately left of the most significant bit. The mantissa is represented by 23-bit fraction plus a hidden 1-bit for a total of 24 bits. The hidden bit comes from the fact that floating-point numbers have to be *normalized*, i.e., the most significant bit of the mantissa has to be 1. Without normalization, a number can have many different representations and make it difficult to perform comparisons, e.g., sorting. Thus, the hidden bit is implicitly defined and extends the mantissa field by an extra bit.

The range of numbers that can be represented is given by

$$(1 + Fraction) \times 2^{-126} \leq |x| \leq (1 + Fraction) \times 2^{127} \tag{9.28}$$

Note that exponent values -127 and 128 are not included. These two values are used to represent zero and infinity, which will be discussed below.

Example 9.26. *Show the IEEE 754 single-precision binary Floating Point representation of the number -5.*

-5_{10} is equivalent to $-101_2 \times 2^0$, which when normalized is equal to $-1.010_2 \times 2^2$. Therefore, the binary representation is given by

S	Exponent	Fraction
1	10000001	01000000000000000000000

When the numbers are normalized, the smallest and the largest possible mantissas are $1.00...0_2$ and $1.11...1_2$, which lead to the following range:

$$\begin{aligned} 1.0 \times 2^{-126} \leq |x| \leq 1.11...1 \times 2^{127} \\ 1.0 \times 2^{-126} \leq |x| < 2 \times 2^{127} \\ 1.175 \times 10^{-38} \leq |x| < 3.4 \times 10^{38} \end{aligned} \tag{9.29}$$

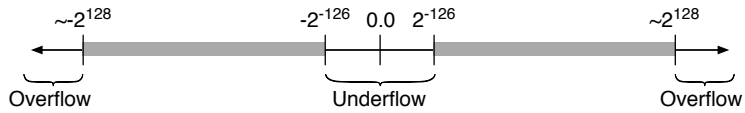


Figure 9.20: Underflow and overflow ranges in single-precision IEEE 754 floating-point format.

Based on this, Figure 9.20 shows the ranges of numbers that IEEE 754 floating-point format *cannot* represent. If a number lies between -2^{-126} and $+2^{-126}$, we say that *underflow* has occurred. On the other hand, if a number is grater than 2^{128} or less than -2^{128}, then we say that *overflow* has occurred. These two cases are referred to as *arithmetic exceptions* and handled by interrupt service routines, for example, to set the value to 0 when underflow occurs and proceed. A more common way to handle underflow and overflow is to represent these numbers using a unique value, which will be discussed shortly.

The representations discussed thus far assumed that the numbers are normalized. That is, the most significant bit of the mantissa is 1. The IEEE floating-point format also has unique formats to represent special values, which are *zero*, *infinity* (∞), *Not a Number* (NaN), and *denormalized*.

Zero is not directly representable due to the implied 1-bit. Instead, it is represented by all 0s in the exponent and the fraction fields. Infinity (∞) is represented with all 1s (255) in the exponent and all 0s in the fraction. Both + and − infinity can be represented using the sign-bit. Infinity is useful because it allows operations to continue beyond overflow situations. *NaN* is represented with all 1s in the exponent and a non-zero fraction in the mantissa. *NaN* is used to represent error values, such as

- The result of an invalid operation, e.g., divide by 0;
- The result returned by a library function when it would be incorrect to return a numeric value; and
- Undetermined value.

There are two types of *NaN*s: *Quiet NaN*(*QNaN*) and *Signaling NaN*(*SNaN*). A *QNaN* is a *NaN* with the most significant bit of the Mantissa set. *QNaNs* propagate freely through most arithmetic operations. These values occur when an operation generates a result that is not mathematically defined. Floating-point operations produce only *QNaNs*. A *SNaN* is a *NaN* with the

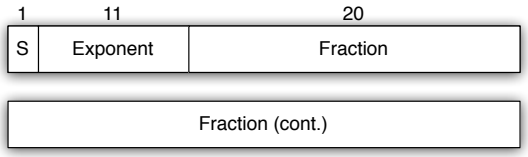


Figure 9.21: Double-precision IEEE 754 floating-point format.

most significant bit in the Mantissa cleared. It is used to signal an exception during operations. *SNaNs* is useful for assigning to uninitialized variables to trap premature usage. Table 9.3 summarizes the results generated from operations on special numbers.

Table 9.3: Results of operations on special numbers.

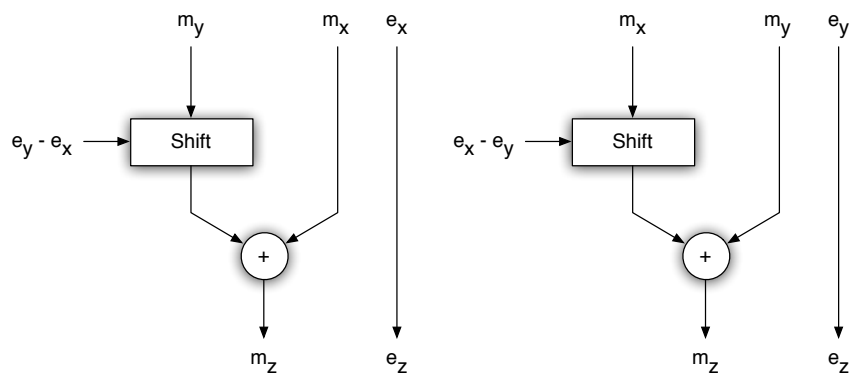
Operation	Result
$n \div \pm\infty$	0
$\pm\infty \times \pm\infty$	$\pm\infty$
$\pm nonzero \div 0$	$\pm\infty$
$\infty + \infty$	∞
$\pm 0 \div \pm 0$	NaN
$\infty - \infty$	NaN
$\pm\infty \div \pm\infty$	NaN
$\pm\infty \times 0$	NaN

Figure 9.21 shows the Double Precision IEEE 754 floating format, which consists of two 32-bit words. The exponent is expanded to 11 bits and uses excess-1023, and the fraction consists of 52 bits. This allows the range to be

$$\begin{aligned} 1.0 \times 2^{-1022} \leq |x| \leq 1.11...1 \times 2^{1023} \\ 1.0 \times 2^{-1022} \leq |x| < 2 \times 2^{1023} \\ 10^{-323.3} < |x| < 10^{308.3} \end{aligned} \tag{9.30}$$

9.6.2 Four Basic Operations

Suppose we have the following two floating-pointer numbers: $x = m_x r^{e_x}$ and $y = m_y r^{e_y}$



(a) $e_x \geq e_y$. (b) $e_x < e_y$.

Figure 9.22: Floating-point add operations.

The basic operations that can be performed on x and y are addition/-subtraction, multiplication, and division. The following discusses these operations.

Addition/Subtraction
Addition/subtraction is represented by the following equation:

$$\begin{aligned} z = x \pm y = m_x r^{e_x} \pm m_y r^{e_y} &= (m_x \pm m_y r^{e_y - e_x}) r^{e_x} \text{ if } e_x \geq e_y \\ &= (m_x r^{e_x - e_y} \pm m_y) r^{e_y} \text{ if } e_x < e_y \end{aligned}$$

Figure 9.22 illustrates the add operation. If the exponents are different, the mantissa with the smaller exponent has to be aligned to the mantissa with the larger exponent. This alignment process is referred to as *pre-normalization*, and involves the following: If $e_x \geq e_y$, then m_y is shifted by $r^{e_y - e_x}$, which is equivalent to shifting m_y to the right by $\mid e_y - e_x \mid$ bits, or shifting its radix point to the left by $\mid e_y - e_x \mid$ bits. Otherwise, m_y is shifted by $r^{e_x - e_y}$, which is equivalent to shifting m_x to the right by $\mid e_x - e_y \mid$ bits, or shifting its radix point to the left by $\mid e_y - e_x \mid$ bits. This leads to the following equations for mantissa m_z and exponent e_z:

$$\begin{aligned} m_z &= m_x \pm m_y r^{e_y - e_x} \text{ and } e_z = e_x \text{ if } e_x \geq e_y \\ m_z &= m_x r^{e_x - e_y} \pm m_y \text{ and } e_z = e_y \text{ if } e_x < e_y \end{aligned}$$

The following example illustrates the addition operation:

Example 9.27. *Perform the operation* $1.001_2 \times 2^3 + 1.001_2 \times 2^2$

First, we consider the exponents. Since $e_x = 3$ and $e_y = 2$, $e_y - e_x = -1$ indicates that m_y is shifted right by one bit or its radix point is shift left by one bit (i.e., r^{-1}). Thus, m_y becomes 0.100. Second, the mantissas are added as shown below

	Mantissa				Exponent
	1.	0	0	1	3
+	0.	1	0	0	3
	1.	1	0	1	3

which results in 1.101×2^3.

In this example, no post-normalization was required since the most significant bit was '1'. However, if this is not the case then *post normalization* will be required. Note that the least significant bit of m_y was lost during pre-normalization. In Section 9.6.3, we will discuss how the lost in precision in minimized by keeping extra bits beyond the number of bits in the mantissa and performing rounding.

Multiplication

Multiplication can be defined by the following set of operations:

$$m_z = m_x + m_y$$
$$e_z = e_x + e_y$$

There are several interesting characteristics about multiplication. First, the smallest normalized mantissa is 1 (i.e., 1.0...), and thus multiplying two smallest mantissas leads to

$$1 \times 1 = 1.$$

Second, the largest normalized mantissa is approximately equal to 2 (i.e., 1.11...). Thus, multiplying two largest mantissas results in

$$2 \times 2 \approx 4.$$

However, the normalized result has satisfy the relationship $1 \leq m_z < 2$. Therefore, the result has to be shifted right at most by one bit, or the radix point has to be shifted left at most by one bit.

Division

Division can be defined by the following set of operations:

$$m_z = m_x / m_y$$
$$e_z = e_x - e_y$$

There are also interesting characteristics about division. The smallest and the largest normalized mantissas are 1 and ≈ 2, respectively. Therefore, the smallest possible result is $\approx 1/2$ and the largest possible results is ≈ 2. Since the results has to be $1 \leq m_z < 2$, we only have to shift the result left at most by one bit, or the radix point has to be shifted right at most by one bit.

9.6.3 Rounding

Rounding is necessary to covert results of floating-point operations into fixed number of bits for the mantissa. In order to illustrates the need for rounding, consider the following example of converting a 32-bit integer to floating-point format.

Example 9.28. *Conversion from integer to floating-point:*

11110000 11001100 10101010 00001111	$\rightarrow$ Integer number
$+1.1110000\ 11001100\ 10101010 \times 2^{31}$	$\rightarrow$ single precision floating-point number
11110000 11001100 10101010 00000000	$\rightarrow$ rounded number

The original fixed 32-bit number is equal to 4,039,944,719. However, when the number is converted to the floating-point representation, the 8 least significant bits are lost due to the precision limitation of the 24-bit mantissa, which results in 4,039,944,704.

What do we need in the hardware to implement rounding? Figure 9.23 shows the additional bits necessary to perform rounding, which are guard (G) bit, rounding (R) bit, and sticky (S) bit. These three extra bits together with the carry (C) bit, i.e., the carry out from an arithmetic operation, provide extra level of precision not possible with just the 23-bit fraction.

The primary purpose of G, R, and S bits is to hold the aligned operand that is shifted right during exponent comparison and provides added precision to the final fraction. In addition, each of these bits has a specific purpose. The *guard bit* catches the result bit immediately right of the least significant bit of the mantissa from either a subtraction or division operation. As discussed previously, results of division require at most shift left by one bit (or the radix point is shifted right one bit) to be normalized. Therefore, the guard bit catches this bit, which can then be shifted into the mantissa. The *sticky bit* represents the lesser significant bits that are not kept due to extreme shifts, which may occur during during mantissa alignment for addition/subtraction. If a value of 1 is ever shifted into the sticky bit position, that sticky bit remains a 1 ("sticks" at 1), despite further

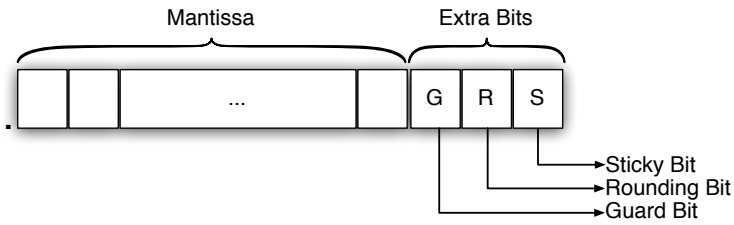


Figure 9.23: Sticky, Rounding, and guard bits for rounding.

shifts. Finally, the *rounding bit* is used to determine how rounding will be performed.

Table 9.4 shows the four rounding modes in the IEEE 754 Standard:

Table 9.4: Rounding modes in the IEEE 754 standard.

Rounding Mode	Negative Result	Positive Result
Always round up (∞)	+1 if R $\vee$ S	
Always round down (-∞)		+1 if R $\vee$ S
Truncate		
Round to nearest even	+1 if (R $\wedge$ LSB) $\vee$ (R $\wedge$ S)	+1 if (R $\wedge$ LSB) $\vee$ (R $\wedge$S)

The following example illustrates the rounding process:

Example 9.29. *Suppose we have* $-1.01000_2 \times 2^2 = -5$ *multiplied by* $1.01000_2 \times 2^3 = 10$ *represented with 5 bits of fraction.*

The mantissas are multiplied as shown below.

$$\begin{array}{cccccc} & 1. & 0 & 1 & 0 & 0 & 0 \\ \times & 1. & 0 & 1 & 0 & 0 & 0 \\ \hline \end{array}$$

This results in

$$\begin{array}{ccccccccccc} 0 & 1. & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ & & & & & & & R & S & & & \end{array}$$

Since the result is already normalized, no post-normalization is required. Moreover, the two bits right of the 5-bit mantissa represent R and S bits. The guard bit does not come into effect because no post-normalization was required.

The results based on the four rounding methods are shown below.

Round to -∞	.1	0	1	0	0
Round to +∞	.1	0	0	1	1
Truncate	.1	0	0	1	1
Round to to Nearest Even	.1	0	0	1	1

Since the exponent is 2+3 = 5, the final result is $-1.10100_2 \times 2^5 = -52$ for round to $-\infty$, and $-1.10011_2 \times 2^5 = -51$ for all other rounding methods.

In general, Round to the nearest even is the most accurate. This can be seen in Table 9.5, which results in accumulated error of 0..

Table 9.5: Errors in the Round to Nearest Even mode.

Number	Round(X)	Error
X0.00	X0.	0
X0.01	X0.	-1/4
X0.10	X0.	-1/2
X0.11	X1.	+1/4
X1.00	X1.	0
X1.01	X1.	-1/4
X1.10	X1.+1	+1/2
X1.11	X1.+1	+1/4

9.6.4 Example: Fractional Division Mantissas

{(BL): needs more work...}

For a divide, there are several key points to keep in mind:

- For a fractional divide, the radix point is assumed to be in between ACC and DIVIDEND registers (instead of the right hand side of the DIVIDEND register as in integer divide).
- For an n-bit by n-bit division, the DIVIDEND register is initialized to zeros. The reasons for this is that division by paper and pencil method requires normalizing fractions to integers. For example,

$$
\begin{array}{cccccc|cccccc}
0. & 1 & 0 & 1 & 0 & 1 & 0. & 0 & 1 & 1 & 0 & 1 \\
 & & & & & & & & & & & \\
 & & & & & & & & & & & \\
 & & & & & & & & & & &

- Dividend A has to be less than divisor B. If $A > B$, A/B will generate integer terms as well (we only want fractional terms). Therefore, a check is made to see if $A < B$. If this condition is not true, A is adjusted (i.e., the mantissa is shifted right and exponent increased) so that $A < B$.
- If the MSB of the dividend is 1, the very first shift-left will cause an overflow. Therefore, we need to have an extra bit called an *extended* (E) bit.

Example 9.30. *Let dividend* $= (1.00)_2$ *and divisor* $= (1.11)_2$*:*

		C	E	ACC			DIVIDEND			
		0	0	1	0	0	0	0	0	
Shift		0	1	0	0	0	0	0	X	
Sub	+	1	1	0	0	1				2s-comp of divisor
		0	0	0	0	1	0	0	**1**	=> positive
Shift		0	0	0	1	0	0	1	X	
Sub	+	1	1	0	0	1				
		1	1	0	1	1	0	1	**0**	=> negative
Shift		1	0	1	1	0	1	0	X	
Add	+	0	0	1	1	1				
		1	1	1	0	1	1	0	**0**	=>negative
				Remainder			Quotient			

At the point, the quotient is 0.100. Since we need to perform rounding, we need additional bits. For division, we need to shift right at most once. Testing whether A is less than B already took care of this possible condition; therefore, we only need a rounding bit and a sticky bit. To generate the rounding and sticky bit, we continue on from the last sequence, except zeros are fed into the ACC register (the final quotient is not touched).

		C	E	ACC			DIVIDEND			
		1	1	1	0	1	**0**	**0**	**0**	=> DIVIDEND=0
Shift		1	1	0	1	0	0	0	X	
Add	+	0	0	1	1	1				
		0	0	0	0	1	0	0	**1**	=> positive
Shift		0	0	0	1	0	0	1	X	
Sub	+	1	1	0	0	1				
		1	1	0	1	1	0	1	**0**	=> negative
								R	S	

Therefore, the resulting mantissa before rounding is 1.00**10**. Applying Round to Nearest Even leads to 1.00.

Appendix A

AVR Instruction Set Summary

A.1 Arithmetic and Logic Instructions

Table A.1: AVR Arithmetic and Logic Instructions

ARITHMETIC AND LOGIC INSTRUCTIONS					
Mnemonics	Operands	Description	Operation	Flags	#Clks
ADD	Rd, Rr	Add two Registers	Rd←Rd+Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry two Registers	Rd←Rd+Rr+C	Z,C,N,V,H	1
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl←Rdh:Rdl+K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract two Registers	Rd←Rd-Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Constant from Register	Rd←Rd-K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry two Registers	Rd←Rd-Rr-C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd←Rd-K-C	Z,C,N,V,H	1
SBIW	Rdl,K	Subtract Immediate from Word	Rdh:Rdl←Rdh:Rdl-K	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND Registers	Rd← Rd∧Rr	Z,N,V	1
ANDI	Rd, K	Logical AND Register and Constant	Rd←Rd∧K	Z,N,V	1
OR	Rd, Rr	Logical OR Registers	Rd← Rd∨Rr	Z,N,V	1
ORI	Rd, K	Logical OR Register and Constant	Rd← Rd∨K	Z,N,V	1
EOR	Rd, Rr	Exclusive OR Registers	Rd←Rd⊕Rr	Z,N,V	1
COM	Rd	One's Complement	Rd←$FF-Rd	Z,C,N,V	1
NEG	Rd	Two's Complement	Rd←$00-Rd	Z,C,N,V,H	1
SBR	Rd,K	Set Bit(s) in Register	Rd←Rd∨K	Z,N,V	1
CBR	Rd,K	Clear Bit(s) in Register	Rd←Rd×($FF-K)	Z,N,V	1
INC	Rd	Increment	Rd←Rd+1	Z,N,V	1
DEC	Rd	Decrement	Rd←Rd-1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd←Rd×Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd←Rd⊕Rd	Z,N,V	1
SER	Rd	Set Register	Rd←$FF	None	1
MUL	Rd, Rr	Multiply Unsigned	R1:R0←Rd×Rr	Z,C	2
MULS	Rd, Rr	Multiply Signed	R1:R0←Rd×Rr	Z,C	2
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0←Rd×Rr	Z,C	2
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0←(Rd×Rr)<<1	Z,C	2
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0←(Rd× Rr)<<1	Z,C	2
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0←(Rd× Rr)<<1	Z,C	2

A.2 Data Transfer Instructions

Table A.2: Data Transfer Instructions

DATA TRANSFER INSTRUCTIONS					
Mnemonics	Operands	Description	Operation	Flags	#Clocks
MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None	1
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd ← Rr+1:Rr	None	1
LDI	Rd, K	Load Immediate	Rd ← K	None	1
LD	Rd, X	Load Indirect	Rd ← (X)	None	2
LD	Rd, X+	Load Indirect and Post-Inc.	Rd ← (X), X ← X + 1	None	2
LD	Rd, -X	Load Indirect and Pre-Dec.	X ← X - 1, Rd ← (X)	None	2
LD	Rd, Y	Load Indirect	Rd ← (Y)	None	2
LD	Rd, Y+	Load Indirect and Post-Inc.	Rd ← (Y), Y ← Y + 1	None	2
LD	Rd, -Y	Load Indirect and Pre-Dec.	Y ← Y - 1, Rd ← (Y)	None	2
LDD	Rd,Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None	2
LD	Rd, Z	Load Indirect	Rd ← (Z)	None	2
LD	Rd, Z+	Load Indirect and Post-Inc.	Rd ← (Z), Z ← Z+1	None	2
LD	Rd, -Z	Load Indirect and Pre-Dec.	Z ← Z - 1, Rd ← (Z)	None	2
LDD	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)	None	2
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None	2
ST	X, Rr	Store Indirect	(X) ← Rr	None	2
ST	X+, Rr	Store Indirect and Post-Inc.	(X) ← Rr, X ← X + 1	None	2
ST	-X, Rr	Store Indirect and Pre-Dec.	X ← X - 1, (X) ← Rr	None	2
ST	Y, Rr	Store Indirect	(Y) ← Rr	None	2
ST	Y+, Rr	Store Indirect and Post-Inc.	(Y) ← Rr, Y ← Y + 1	None	2
ST	-Y, Rr	Store Indirect and Pre-Dec.	Y ← Y - 1, (Y) ← Rr	None	2
STD	Y+q,Rr	Store Indirect with Displacement	(Y + q) ← Rr	None	2
ST	Z, Rr	Store Indirect	(Z) ← Rr	None	2
ST	Z+, Rr	Store Indirect and Post-Inc.	(Z) ← Rr, Z ← Z + 1	None	2
ST	-Z, Rr	Store Indirect and Pre-Dec.	Z ← Z - 1, (Z) ← Rr	None	2
STD	Z+q,Rr	Store Indirect with Displacement	(Z + q) ← Rr	None	2
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None	2
LPM		Load Program Memory	R0 ← (Z)	None	3
LPM	Rd, Z	Load Program Memory	Rd ← (Z)	None	3
LPM	Rd, Z+	Load Program Memory and Post-Inc	Rd ← (Z), Z ← Z+1	None	3
ELPM		Extended Load Program Memory	R0 ← (RAMPZ:Z)	None	3
ELPM	Rd, Z	Extended Load Program Memory	Rd ← (RAMPZ:Z)	None	3
ELPM	Rd, Z+	Extended Load Program Memory and Post-Inc	Rd ← (RAMPZ:Z), RAMPZ:Z ← RAMPZ:Z+1	None	3
SPM		Store Program Memory	(Z) ← R1:R0	None	-
IN	Rd, P	In Port	Rd ← P	None	1
OUT	P, Rr	Out Port	P ← Rr	None	1
PUSH	Rr	Push Register on Stack	STACK ← Rr	None	2
POP	Rd	Pop Register from Stack	Rd ← STACK	None	2

A.3 Branch Instructions

Table A.3: AVR Arithmetic and Logic Instructions

BRANCH INSTRUCTIONS					
Mnemonics	Operands	Description	Operation	Flags	#Clocks
RJMP	k	Relative Jump	PC←PC+k+1	None	2
IJMP		Indirect Jump to (Z)	PC←Z	None	2
JMP	k	Direct Jump	PC←k	None	3
RCALL	k	Relative Subroutine Call	PC←PC+k+1	None	3
ICALL		Indirect Call to (Z)	PC←Z	None	3
CALL	k	Direct Subroutine Call	PC←k	None	4
RET		Subroutine Return	PC←STACK	None	4
RETI		Interrupt Return	PC←STACK	I	4
CPSE	Rd,Rr	Compare, Skip if Equal	if (Rd = Rr) PC←PC+2 or 3	None	1 / 2 / 3
CP	Rd,Rr	Compare	Rd - Rr	Z, N,V,C,H	1
CPC	Rd,Rr	Compare with Carry	Rd - Rr - C	Z, N,V,C,H	1
CPI	Rd,K	Compare Register with Immediate	Rd - K	Z, N,V,C,H	1
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b)=0) PC←PC+2 or 3	None	1 / 2 / 3
SBRS	Rr, b	Skip if Bit in Register is Set	if (Rr(b)=1) PC←PC+2 or 3	None	1 / 2 / 3
SBIC	P, b	Skip if Bit in I/O Register Cleared	if (P(b)=0) PC←PC+2 or 3	None	1 / 2 / 3
SBIS	P, b	Skip if Bit in I/O Register is Set	if (P(b)=1) PC←PC+2 or 3	None	1 / 2 / 3
BRBS	s, k	Branch if Status Flag Set	if (SREG(s)=1) then PC←PC+k+1	None	1 / 2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s)=0) then PC←PC+k+1	None	1 / 2
BREQ	k	Branch if Equal	if (Z=1) then PC←PC+k+1	None	1 / 2
BRNE	k	Branch if Not Equal	if (Z=0) then PC←PC+k+1	None	1 / 2
BRCS	k	Branch if Carry Set	if (C=1) then PC←PC+k+1	None	1 / 2
BRCC	k	Branch if Carry Cleared	if (C=0) then PC←PC+k+1	None	1 / 2
BRSH	k	Branch if Same or Higher	if (C=0) then PC←PC+k+1	None	1 / 2
BRLO	k	Branch if Lower	if (C=1) then PC←PC+k+1	None	1 / 2
BRMI	k	Branch if Minus	if (N=1) then PC←PC+k+1	None	1 / 2
BRPL	k	Branch if Plus	if (N=0) then PC←PC+k+1	None	1 / 2
BRGE	k	Branch if Greater or Equal, Signed	if (N⊕V= 0) then PC←PC+k+1	None	1 / 2
BRLT	k	Branch if Less Than Zero, Signed	if (N⊕V= 1) then PC←PC+k+1	None	1 / 2
BRHS	k	Branch if Half Carry Flag Set	if (H=1) then PC←PC+k+1	None	1 / 2
BRHC	k	Branch if Half Carry Flag Cleared	if (H=0) then PC←PC+k+1	None	1 / 2
BRTS	k	Branch if T Flag Set	if (T=1) then PC←PC+k+1	None	1 / 2
BRTC	k	Branch if T Flag Cleared	if (T=0) then PC←PC+k+1	None	1 / 2
BRVS	k	Branch if Overflow Flag is Set	if (V=1) then PC←PC+k+1	None	1 / 2
BRVC	k	Branch if Overflow Flag is Cleared	if (V=0) then PC←PC+k+1	None	1 / 2
BRIE	k	Branch if Interrupt Enabled	if (I=1) then PC←PC+k+1	None	1 / 2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC←PC+k+1	None	1 / 2

A.4 Bit and Bit-test Instructions

BIT AND BIT-TEST INSTRUCTIONS					
Mnemonics	Operands	Description	Operation	Flags	#C
SBI	P,b	Set Bit in I/O Register I/O	(P,b) ← 1	None	2
CBI	P,b	Clear Bit in I/O Register I/O	(P,b) ← 0	None	2
LSL	Rd	Logical Shift Left	Rd(n+1)←Rd(n), Rd(0)←0	Z,C,N,V	1
LSR	Rd	Logical Shift Right	Rd(n)←Rd(n+1), Rd(7)←0	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	Rd(0)←C,Rd(n+1)←Rd(n),C←Rd(7)	Z,C,N,V	1
ROR	Rd	Rotate Right Through Carry	Rd(7)←C,Rd(n)←Rd(n+1),C←Rd(0)	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0..6	Z,C,N,V	1
SWAP	Rd	Swap Nibbles	Rd(3..0)←Rd(7..4),Rd(7..4)←Rd(3..0)	None	1
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)	1
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)	1
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T	1
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None	1
SEC		Set Carry	C ← 1	C	1
CLC		Clear Carry	C ← 0	C	1
SEN		Set Negative Flag	N ← 1	N	1
CLN		Clear Negative Flag	N ← 0	N	1
SEZ		Set Zero Flag	Z ← 1	Z	1
CLZ		Clear Zero Flag	Z ← 0	Z	1
SEI		Global Interrupt Enable	I ← 1	I	1
CLI		Global Interrupt Disable	I ← 0	I	1
SES		Set Signed Test Flag	S ← 1	S	1
CLS		Clear Signed Test Flag	S ← 0	S	1
SEV		Set Twos Complement Overflow.	V ← 1	V	1
CLV		Clear Twos Complement Overflow	V ← 0	V	1
SET		Set T in SREG	T ← 1	T	1
CLT		Clear T in SREG	T ← 0	T	1
SEH		Set Half Carry Flag in SREG	H ← 1	H	1
CLH		Clear Half Carry Flag in SREG	H ← 0	H	1

A.5 MCU Control Instructions

MCU CONTROL INSTRUCTIONS			
Mnemonics	Description	Flags	#Clocks
NOP	No Operation	None	1
SLEEP	Sleep (see specific descr. for Sleep function)	None	1
WDR	Watchdog Reset (see specific descr. for WDR/timer)	None	1
BREAK	Break For On-chip Debug Only	None	N/A

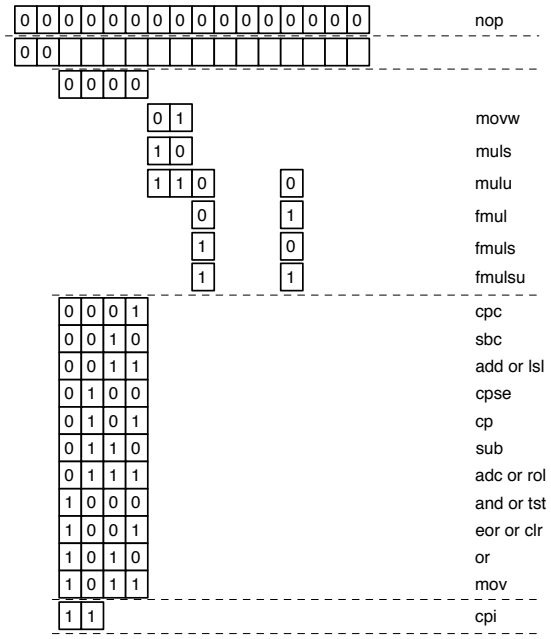


Figure A.1: Category 1 opcode encoding.

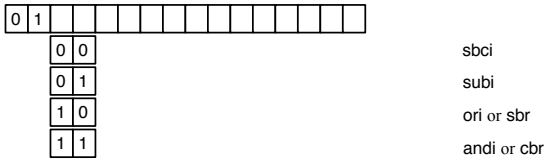


Figure A.2: Category 2 opcode encoding.

A.6 AVR Opcode Encoding

A.7 AVR Opcodes

```
Legend:
        r   any register
        d   'ldi' register (r16-r31)
        v   'movw' even register (r0, r2, ..., r28, r30)
        a   'fmul' register (r16-r23)
        w   'adiw' register (r24,r26,r28,r30)
        e   pointer registers (X,Y,Z)
        b   base pointer register and displacement ([YZ]+disp)
        z   Z pointer register (for [e]lpm Rd,Z[+])
        M   immediate value from 0 to 255
        n   immediate value from 0 to 255 ( n = ~M ). Relocation impossible
        s   immediate value from 0 to 7
        P   Port address value from 0 to 63. (in, out)
        p   Port address value from 0 to 31. (cbi, sbi, sbic, sbis)
        K   immediate value from 0 to 63 (used in 'adiw', 'sbiw')
        i   immediate value
        l   signed pc relative offset from -64 to 63
        L   signed pc relative offset from -2048 to 2047
        h   absolute code address (call, jmp)
        S   immediate value from 0 to 7 (S = s << 4)
        ?   use this opcode entry if no parameters, else use next opcode entry

     1001010010001000   clc
     1001010011011000   clh
     1001010011111000   cli
     1001010010101000   cln
     1001010011001000   cls
     1001010011101000   clt
     1001010010111000   clv
     1001010010011000   clz
     1001010000001000   sec
     1001010001011000   seh
     1001010001111000   sei
     1001010000101000   sen
     1001010001001000   ses
     1001010001101000   set
     1001010000111000   sev
     1001010000011000   sez
     100101001SSS1000   bclr    S
     100101000SSS1000   bset    S
     1001010100001001   icall
     1001010000001001   ijmp
     1001010111001000   lpm     ?
     1001000ddddd010+   lpm     r,z
     1001010111011000   elpm    ?
     1001000ddddd011+   elpm    r,z
     0000000000000000   nop
```

```
1001010100001000   ret
1001010100011000   reti
1001010110001000   sleep
1001010110011000   break
1001010110101000   wdr
1001010111101000   spm
000111rdddddrrrr   adc     r,r
000011rdddddrrrr   add     r,r
001000rdddddrrrr   and     r,r
000101rdddddrrrr   cp      r,r
000001rdddddrrrr   cpc     r,r
000100rdddddrrrr   cpse    r,r
001001rdddddrrrr   eor     r,r
001011rdddddrrrr   mov     r,r
100111rdddddrrrr   mul     r,r
001010rdddddrrrr   or      r,r
000010rdddddrrrr   sbc     r,r
000110rdddddrrrr   sub     r,r
001001rdddddrrrr   clr     r
000011rdddddrrrr   lsl     r
000111rdddddrrrr   rol     r
001000rdddddrrrr   tst     r
0111KKKKddddKKKK   andi    d,M
0111KKKKddddKKKK   cbr     d,n
1110KKKKddddKKKK   ldi     d,M
11101111dddd1111   ser     d
0110KKKKddddKKKK   ori     d,M
0110KKKKddddKKKK   sbr     d,M
0011KKKKddddKKKK   cpi     d,M
0100KKKKddddKKKK   sbci    d,M
0101KKKKddddKKKK   subi    d,M
1111110rrrrr0sss   sbrc    r,s
1111111rrrrr0sss   sbrs    r,s
1111100ddddd0sss   bld     r,s
1111101ddddd0sss   bst     r,s
10110PPdddddPPPP   in      r,P
10111PPrrrrrPPPP   out     P,r
10010110KKddKKKK   adiw    w,K
10010111KKddKKKK   sbiw    w,K
10011000ppppppsss  cbi     p,s
10011010ppppppsss  sbi     p,s
10011001ppppppsss  sbic    p,s
10011011ppppppsss  sbis    p,s
1111011lllllll000  brcc    l
1111001lllllll000  brcs    l
1111001lllllll001  breq    l
1111011lllllll100  brge    l
1111011lllllll101  brhc    l
1111001lllllll101  brhs    l
```

```
1111011lllllll111   brid    l
1111001lllllll111   brie    l
1111001lllllll000   brlo    l
1111001lllllll100   brlt    l
1111001lllllll010   brmi    l
1111011lllllll001   brne    l
1111011lllllll010   brpl    l
1111011lllllll000   brsh    l
1111011lllllll110   brtc    l
1111001lllllll110   brts    l
1111011lllllll011   brvc    l
1111001lllllll011   brvs    l
1111011lllllllsss   brbc    s,l
1111001lllllllsss   brbs    s,l
1101LLLLLLLLLLLL    rcall   L
1100LLLLLLLLLLLL    rjmp    L
1001010hhhhh111h    call    h
1001010hhhhh110h    jmp     h
1001010rrrrr0101    asr     r
1001010rrrrr0000    com     r
1001010rrrrr1010    dec     r
1001010rrrrr0011    inc     r
1001010rrrrr0110    lsr     r
1001010rrrrr0001    neg     r
1001000rrrrr1111    pop     r
1001001rrrrr1111    push    r
1001010rrrrr0111    ror     r
1001010rrrrr0010    swap    r
00000001ddddrrrr    movw    v,v
00000010ddddrrrr    muls    d,d
000000110ddd0rrr    mulsu   a,a
000000110ddd1rrr    fmul    a,a
000000111ddd0rrr    fmuls   a,a
000000111ddd1rrr    fmulsu  a,a
1001001ddddd0000    sts     i,r
1001000ddddd0000    lds     r,i
10o0oo0dddddbooo    ldd     r,b
100!000dddddee-+    ld      r,e
10o0oo1rrrrrbooo    std     b,r
100!001rrrrree-+    st      e,r
1001010100011001    eicall
1001010000011001    eijmp



0000000000000000    nop

00 11 KKKKddddKKKK   cpi     d,M
```

```
00 0000 01 ddddrrrr   movw    v,v
00 0000 10 ddddrrrr   muls    d,d
00 0000 11 0ddd0rrr   mulsu   a,a
00 0000 11 0ddd1rrr   fmul    a,a
00 0000 11 1ddd0rrr   fmuls   a,a
00 0000 11 1ddd1rrr   fmulsu  a,a

00 0001 rdddddrrrr   cpc     r,r
00 0010 rdddddrrrr   sbc     r,r
00 0011 rdddddrrrr   add     r,r
00 0011 rdddddrrrr   lsl     r**
00 0100 rdddddrrrr   cpse    r,r
00 0101 rdddddrrrr   cp      r,r
00 0110 rdddddrrrr   sub     r,r
00 0111 rdddddrrrr   adc     r,r
00 0111 rdddddrrrr   rol     r**
00 1000 rdddddrrrr   and     r,r
00 1000 rdddddrrrr   tst     r**
00 1001 rdddddrrrr   eor     r,r
00 1001 rdddddrrrr   clr     r**
00 1010 rdddddrrrr   or      r,r
00 1011 rdddddrrrr   mov     r,r

01 00 KKKKddddKKKK   sbci    d,M
01 01 KKKKddddKKKK   subi    d,M
01 10 KKKKddddKKKK   ori     d,M
01 10 KKKKddddKKKK   sbr     d,M**
01 11 KKKKddddKKKK   andi    d,M
01 11 KKKKddddKKKK   cbr     d,n**



10 q0 qq 0d dddd 1qqq    ldd     Rd,Y+q
10 q0 qq 1r rrrr 1qqq    std     Y+q,Rr
10 01 00 0d dddd ee-+    ld      r,e
10 01 00 1r rrrr ee-+    st      e,r



10 01 000 ddddd 010+   lpm     r,z
10 01 000 ddddd 011+   elpm    r,z

10 01 000 rrrrr 1111   pop     r
10 01 001 rrrrr 1111   push    r


\bf{10 01010 xxxx 0xxx}

10 01 010 rrrrr 0000   com     r
```

```
  10 01 010 rrrrr 0001   neg     r
  10 01 010 rrrrr 0010   swap    r
  10 01 010 rrrrr 0011   inc     r
  10 01 010 rrrrr 0101   asr     r
  10 01 010 rrrrr 0110   lsr     r
  10 01 010 rrrrr 0111   ror     r



  10 01 010 rrrrr 1010   dec     r
  10 01 010 hhhhh 110h   jmp     h
  10 01 010 hhhhh 111h   call    h

\bf{10 0101 0 xxxx 1000}
  10 01 010 0 0000 1000   sec
  10 01 010 0 0001 1000   sez
  10 01 010 0 0010 1000   sen
  10 01 010 0 0011 1000   sev
  10 01 010 0 0100 1000   ses
  10 01 010 0 0101 1000   seh
  10 01 010 0 0110 1000   set
  10 01 010 0 0111 1000   sei
  10 01 010 0 1000 1000   clc
  10 01 010 0 1001 1000   clz
  10 01 010 0 1010 1000   cln
  10 01 010 0 1011 1000   clv
  10 01 010 0 1100 1000   cls
  10 01 010 0 1101 1000   clh
  10 01 010 0 1110 1000   clt
  10 01 010 0 1111 1000   cli

  10 01 010 0 1SSS 1000   bclr    S
  10 01 010 0 0SSS 1000   bset    S


  10 01 010 1 0000 1000   ret
  10 01 010 1 0001 1000   reti
  10 01 010 1 1000 1000   sleep
  10 01 010 1 1001 1000   break
  10 01 010 1 1010 1000   wdr
  10 01 010 1 1100 1000   lpm     ?
  10 01 010 1 1101 1000   elpm    ?
  10 01 010 1 1110 1000   spm

  \bf{10 01010 x xxxx 1001}

  10 01 010 0 0000 1001   ijmp
  10 01 010 1 0000 1001   icall
```

```
 \bf{10 01 xxxx xxxx xxxx}
10 01 00 1 ddddd0000   sts     i,r
10 01 00 0 ddddd0000   lds     r,i

10 01 01 00 00011001   eijmp
10 01 01 01 00011001   eicall
10 01 01 10 KKddKKKK   adiw    w,K
10 01 01 11 KKddKKKK   sbiw    w,K

10 01 10 00 pppppsss   cbi     p,s
10 01 10 10 pppppsss   sbi     p,s
10 01 10 01 pppppsss   sbic    p,s
10 01 10 11 pppppsss   sbis    p,s

10 01 11 rdddddrrrr   mul     r,r

10 11 0PPdddddPPPP   in      r,P
10 11 1PPrrrrrPPPP   out     P,r



11 00 LLLLLLLLLLLL   rjmp    L
11 01 LLLLLLLLLLLL   rcall   L

11 10 KKKKddddKKKK   ldi     d,M
11 10 1111dddd1111   ser     d

11 11 00 lllllll101   brhs    l
11 11 00 lllllll000   brcs    l
11 11 00 lllllll000   brlo    l**
11 11 00 lllllll001   breq    l
11 11 00 lllllll010   brmi    l
11 11 00 lllllll011   brvs    l
11 11 00 lllllll100   brlt    l
11 11 00 lllllll110   brts    l
11 11 00 lllllll111   brie    l

11 11 00 lllllllsss   brbs    s,l

11 11 01 lllllll000   brcc    l
11 11 01 lllllll000   brsh    l
11 11 01 lllllll001   brne    l
11 11 01 lllllll010   brpl    l
```

```
11 11 01 lllllll011   brvc    l
11 11 01 lllllll100   brge    l
11 11 01 lllllll101   brhc    l
11 11 01 lllllll110   brtc    l
11 11 01 lllllll111   brid    l

11 11 01 llllllsss   brbc    s,l

11 11 10 0 ddddd0sss   bld     r,s
11 11 10 1 ddddd0sss   bst     r,s
11 11 11 0 rrrrr0sss   sbrc    r,s
11 11 11 1 rrrrr0sss   sbrs    r,s
```

1	0																
		0	X	0	0	0											ld
		0	X	0	0	1											st
		X	0			0						0					ldd
		X	0			1						1					std
		0	1	0	0	0							0	1	0	X	lpm
													0	1	1	X	elpm
													1	1	1	1	pop
		0	1	0	0	1							1	1	1	1	push
		0	1	0	1	0							0	0	0	0	com
													0	0	0	1	neg
													0	0	1	0	swap
													0	0	1	1	inc
													0	1	0	1	asr
													0	1	1	0	lsr
													0	1	1	1	ror
													1	0	1	0	dec
													1	1	0	X	jmp
													1	1	1	X	call
		0	1	0	1	0	0	0	0	0	0	1	0	0	0		sec
							0	0	0	1							sez
							0	0	1	0							sen
							0	0	1	1							sev
							0	1	0	0							ses
							0	1	0	1							seh
							0	1	1	0							set
							0	1	1	1							sei
							1	0	0	0							clc
							1	0	0	1							clz
							1	0	1	0							cln
							1	0	1	1							clv
							1	1	0	0							cls
							1	1	0	1							clh
							1	1	1	0							clt
							1	1	1	1							cli
		0	1	0	1	0	1	0	0	0	0	1	0	0	0		ret
								0	0	0	1						reti
								1	0	0	0						sleep
								1	0	0	1						break
								1	0	1	0						wdr
								1	1	0	0						lpm
								1	1	0	1						elpm
								1	1	1	0						spm
		0	1	0	1	0	0	0	0	0	0	1	0	0	1		ijmp
							1	0	0	0	0						icall
		0	1	0	0	0							0	0	0	0	lds
						1											sts
		0	1	0	1	0	0	0	0	0	1	1	0	0	1		eijmp
						0	1										eicall
		0	1	0	1	1	0										adiw
						1	1										sbiw
		0	1	1	0	0	0										cbi
						0	1										sbic
						1	0										sbi
						1	1										sbis
		1	1	0													in
				1													out

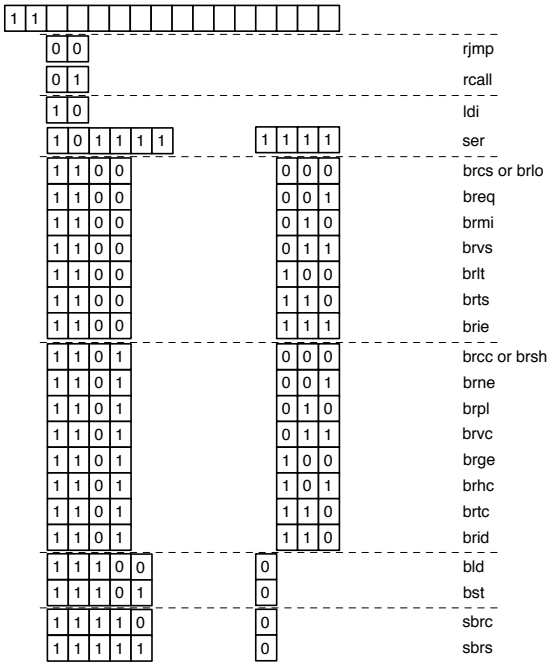


Figure A.4: Category 4 opcode encoding.

Appendix B

AVR Studio 4

AVR Studio 4 is the new professional Integrated Development Environment (IDE) for writing and debugging AVR applications in Windows 9x/NT/2000/XP environments. AVR Studio 4 supports the following development tools: ICE50, JTAGICE, ICE200, STK500, and AVRISP. AVR Studio 4 was created by the Atmel Corporation and can be downloaded for free from http://www.atmel.com/atmel/products/prod203.htm.

This section provides general information on to successfully use AVR Studio 4 to complete most, if not all, of the projects. Not every aspect of AVR Studio 4 will be covered here, but for those who choose to learn the program in more detail, additional information can be attained from Atmels website at http://www.atmel.com.

B.1 Startup Tutorial

This tutorial will give a step-by-step guide on how to install AVR Studio 4, create a project, and simulate the project. For the purposes of this tutorial, the first lab for ECE 375 will be used. The code can be downloaded from http://engr.oregonstate.edu/ zier/ECE375/Lab1.asm.

B.1.1 Installation

The installation of AVR Studio 4 is a pretty straightforward process, but nevertheless, here is how to do it.

1. Download astudio4.zip from http://www.atmel.com/atmel/products/prod203.htm. As of this writing, it should be the first link under Development Software on said website.
2. Unzip astudio4.zip into an easily accessible directory such as My Desktop or My Documents.
3. Run the setup program by double left clicking (DLC) on setup.exe.
4. Follow the instructions in the setup program. Most of the default installation directories will work just fine.
5. When finished, click on the Finish button. Thats it; AVR Studio 4 is successfully installed.

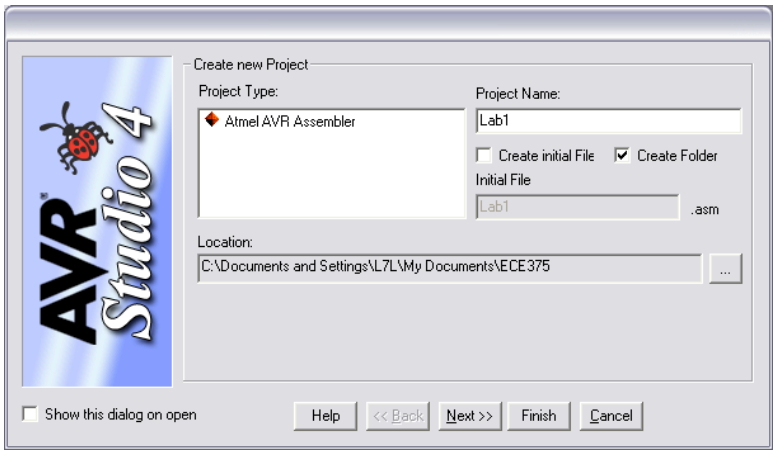


Figure B.1: AVR Studio Project Creation.

B.1.2 Project Creation

AVR Studio 4 is an Integrated Development Environment (IDE). Just like any other IDE, AVR Studio 4 is project based. A project is like an environment for a particular program that is being written. It keeps track of what files are open, compilation instructions, as well as the current Graphical User Interface (GUI) selections. It would be wise to create a separate project and project folder for each and every lab. To demonstrate creating a project, the first lab will be used.

1. Start AVR Studio 4 by clicking through the menus Start-¿Programs-¿Atmel AVR Tools-¿AVR Studio 4. The path could be different if changed during installation.
2. AVR Studio 4 should start up and place a Welcome to AVR Studio 4 dialogue box up. Click on the New Project button in the upper top left. If box does not appear, click on Project-¿New Project.
3. In the Project Type box, select Atmel AVR Assembler. Future versions will have different project types, but for now, this is the only option available.
4. In the Project Name text box, type the name of the project, such as Lab 1.
5. Normally, both check boxes would be checked to create initial file and create folder, but since we already have the initial file, leave that box unchecked.
6. The Location of the project can be changed by clicking on the button next to path name. It is recommended to place the project file in a networked directory (such as ENGR of FLOP) if using a computer on campus or in My Documents folder on a personal machine.
7. When finished, the popup dialogue should look like Figure 1: AVR Studio Project Creation.
8. Click Next to continue.
9. The next dialogue asks to Select Debug Platform and Device.

   (a) Select AVR Simulator from the Debug Platform window.
   (b) Select ATmega128 from the Device window.

10. Click Finish to complete the project creation.
11. Normally, if Create initial File was selected, a blank page would appear to start writing the program on. But, the program is already written, so we must add it to the project.
12. Download the lab1.asm file into the newly created project folder.
13. In the Workspace window in AVR Studio 4, select the Project tab on the bottom left.
14. Open the project tree by clicking on the plus sign next to Lab1. You should now see a folder labeled Assembler.
15. Right Click on the Assembler folder to bring up an option box.
16. Select Add Existing File from the option box and an Open dialogue should appear.
17. If lab1.asm was downloaded correctly, it should be the only file available to choose from. Double left click on lab1.asm to open the file.
18. lab1.asm should now appear under the Assembler folder in the Workspace window and the actual program should now appear in the editor area on right.
19. That is it. The project file has now been created.

B.1.3 Project Simulation

Now that the project has been created, it needs to be tested. Running the program on a simulated chip built into AVR Studio 4 does this. AVR Studio 4 has the capability to simulate almost every AVR microcontroller offered by Atmel. For the purposes of this tutorial and ECE 375 class, the ATmega128 will be the chip that will be simulated. This chip was selected during the project creation phase. The chip can also be reselected by clicking on Debug-¿Select Platform and Device.

1. Before the program can be simulated, the program must first be compiled. There are three ways to do this:
   (a) Follow the menu Project-¿Build.
   (b) Click on the build icon, .
   (c) Or use the F7 key.
2. If the code was successfully compiled, a message in the message box at the bottom should read Assembly complete with no errors. If it does not say this, then there were some errors in the code. Clicking on the errors in the message box will highlight the error in the code in the editor window.
3. Once the code has been successfully compiled, simulation can begin. There are two ways to simulate the chip, debugging mode that allows a line-by-line simulation and run mode that continuously runs the program.
   (a) There are a couple of ways to run in debug mode:
      i. Follow the menu Debug-¿Start Debugging.

ii. Or click on the Start Debugging Icon, .

(b) To start the run mode:

i. Follow the menu Debug-¿Run.

ii. Click on the Run icon, .

iii. Or use the F5 key.

4. To stop the simulation at any point:

(a) Follow the menu Debug-¿Stop Debugging.

(b) Or click on the Stop Debugging Icon, .

5. That is how to simulate a program. For more detailed simulation tips and strategies, see Simulation Tips.

B.2 Simulation Tips

Just simulating a program is not enough. Knowing how to use the simulator and debugger is essential to get results from simulation. This section will provide the necessary information needed to get the most out of a simulation.

B.2.1 Line-By-Line Debugging

Line-by-line debugging is the best way to take control of the simulation. It allows the programmer to verify data in registers and memory. There are several ways to get into line-by-line debugging mode. The first would be to start the simulation in line-by-line debug mode by clicking on the debug icon, , or through the menu Debug-¿Start Debugging. When the program is in run mode, hitting the pause key, , will halt the simulation and put it into line-by-line mode. Also, if a break point was set in the code, the simulation will automatically pause at the break point and put the simulation into line-by-line mode.

When running in line-by-line mode, several new buttons will be activated. These allow you to navigate through the program.

- Step Into (F11) Steps into the code. Normal operation will run program line-by-line, but will step into routine calls such as the RCALL command.
- Step Over (F10) Steps over routine calls. Normal operation will run program line-by-line, but will treat routine calls as a single instruction and not jump to the routine instructions.
- Step Out (Shift+F11) Steps out of routine calls. This will temporarily put the simulation into run mode for the remainder of the routine and will pause at the next instruction after the routine call.
- Run to Cursor (Ctrl+F10) Runs simulation until cursor is reached. The cursor is the blinking line indicating where to type. Place the cursor by putting the mouse over the instruction you want to stop at and hit the Run to Cursor icon.

After experimenting around with these four commands, you should be able to navigate through the code with ease.

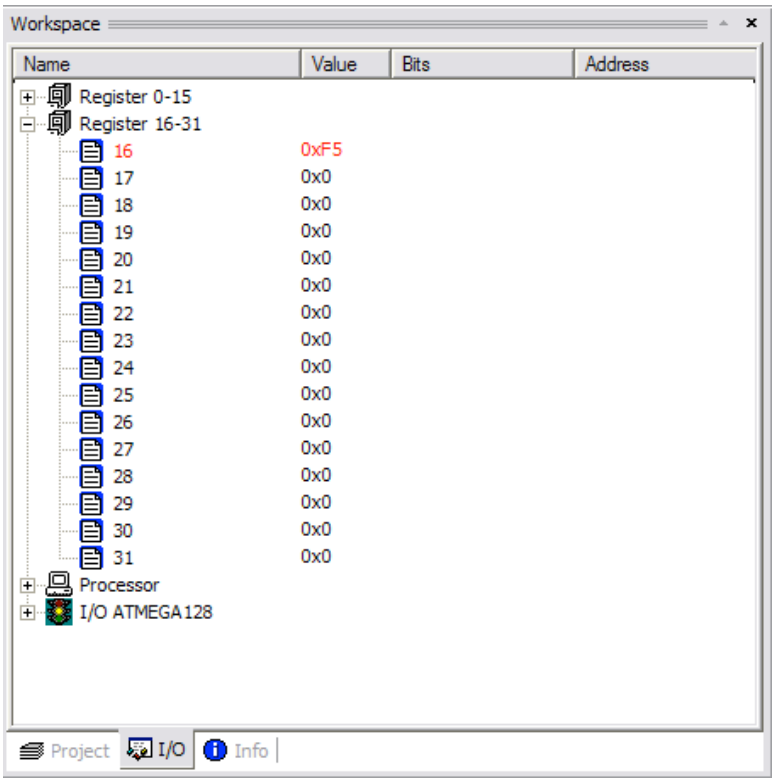


Figure B.2: I/O Window in Workspace.

B.2.2 Workspace Window

The workspace window holds valuable information about the project and the simulation. There are three areas of the workspace windows signified by tabs at the bottom; Project, I/O, and Info.

The Project window contains all the information related to the project such as files and file structuring. From here, files can be created, destroyed, rearranged, or organized. You can also use this window to select which file appears in the editor window.

The I/O window contains all the registers associated with the simulated chip. By default, this window should automatically be displayed when simulation is run in line-by-line mode. Figure 2: I/O Window in Workspace shows an example of what the I/O window looks like during the Lab 1 simulation. By expanding this window, additional information is available such as Bits and Address of the registers. It is in this window where you can simulate input on the ports. The node labeled I/O ATMEGA128 contains all the special registers associated with the chip.

Note: By expanding the I/O ATMEGA128 node, you have access to the ports and can simulate input or interrupts through them.

The Info window displays information about the current chip that is being simulated.

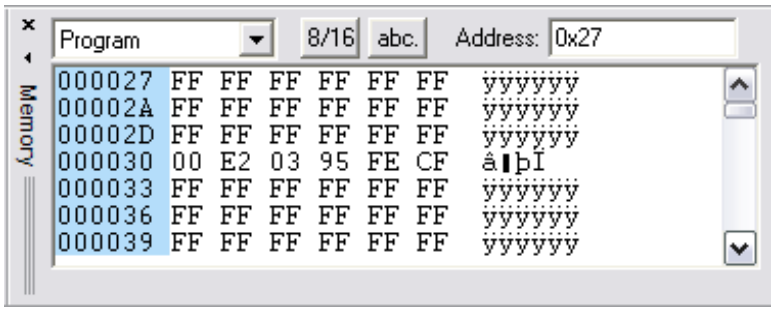


Figure B.3: Memory Window.

This information contains names and addresses of registers and external pins. It is a good point of reference when programming for the chip.

B.2.3 Memory Windows

In actuality, all of the registers are actually parts of memory within the ATmega128. In addition to the register memory, the ATmega128 has several other memory banks, including the program memory, data memory, and EEPROM memory. Of course, no good simulator is complete without being able to view and/or modify this memory, and AVR Studio 4 is no exception.

To view the memory window, follow the menu command View-¿Memory Window or hit Alt+4. The window will pop up on top and can sometimes get in the way. A good way to organize it is to drag the window down next to the message window at the bottom and it will slide in right next to it. Expanding the window will also show additional information. Figure 3: Memory Window shows an example of what it looks like when it is docked next to the message window and expanded.

The main area of the Memory Window contains three sets of information; the address of the line in the blue, the data of the memory in hexadecimal format, and the ASCII equivalent of that data. The pull down menu on the top left allows you to select the various memory banks available for the ATmega128. To edit the memory, just place the cursor in the hexadecimal data area and type in the new data.

B.3 Debugging Strategies

Debugging code can be the most time consuming process in programming. Here are some tips and strategies that can help with this process:

- Comment, Comment, Comment. Unless it is absolutely blatantly obvious of what the code is doing, comment EVERY line of code. Even if the code is obvious, at least comment what the group of instruction is doing, for example, Initializing Stack Pointer.
- Pick a format and stick with it. The format is how you lay out your code. A single programming format will make reading the code a lot easier.

- Before writing any actual code, write it out in pseudo-code and convince yourself that it works.
- Break the code down into small routines and function calls. Small sections of code are much easier to debug than one huge section of code.
- Wait loops should be commented out during debugging. The simulator is much slower than the actual chip and extensive wait loops take up a lot of time.
- Use breakpoints to halt the simulation at the area known to be buggy. Proper use of breakpoints can save a lot of time and frustration.
- Carefully monitor the I/O Window and Memory Windows throughout the simulation. These windows will indicate any problem.
- Make sure the AVR instruction is actually supported by the ATmega128.
- The ATmega128 has certain memory ranges; so make sure that when manipulating data, the addresses are within range.

Appendix C

Microcontroller Programming with PonyProg2000

The primary goal of most AVR assemblers is to use their code on an actual microcontroller. Unfortunately, AVRStudio4 does not provide a way to physically upload the data file onto a microcontroller. Future versions of AVRStudio4 may contain such enhancements, but they will probably rely on Atmel proprietary interface. For now, there is an excellent freeware serial device programmer called PonyProg2000. There are several different flavors of PonyProg, but since this document focuses on the ATmega128, we will use PonyProg2000. This section will contain several step-by-step tutorials on how to use PonyProg. The main site that supports PonyProg2000 can be found at http://www.lancos.com/prog.

C.1 Installing PonyProg2000

This section explains where and how a user would download and install PonyProg2000. This is an ideal section for those who do not already have PonyProg2000 on their system. This program is freeware and will cost you nothing to use, but freeware has it limitations in documentation and support. So far, the author has found no problems with the current version of PonyProg2000.

The following is a step-by-step tutorial on how to download and install the program. This program is designed for Windows machines only.

1. Go to the site http://www.lancos.com/ppwin95.html and click on the link that reads v2.05a BETA. This will start the download process.
2. Save the downloaded zip file labeled ponyprogV205a.zip to an easily accessible folder.
3. Once the zip file has finished downloading, unzip ponyprogV205a.zip and extract the setup.exe program from the zip file.
4. Run the setup.exe program to begin the installation process.

5. Click yes on the dialog box that appears.
6. Follow the steps in the Installation Wizard.
7. The default directories it assigns for the program location and group icons are OK.
8. Once the Installation Wizard is complete, the program is completely installed.

C.1.1 PonyProg2000 Setup

There are a few things that need to be done with the program before a user can upload data into a microcontroller. This section gives a step-by-step guide on how to setup both PonyProg2000 and the microcontroller board.

C.1.2 Setup the AVR Microcontroller Board

1. Connect the ISP dongal, (A small device that converts parallel data to serial ISP data; for students using the TekBots V3.75 kit, it is the small board with a single chip on it that came with the kit), to the parallel port on the PC. This should be LPT1 on most systems.
2. Connect one end of the ISP cable, a ribbon cable with a notched 2x5 female connector at either end, to the ISP dongal.
3. Connect the other end of the ISP cable to the ISP port on the AVR micro-controller board; it should be the J22 that is right next to the serial port.
4. Now that the ISP connection is made, apply power to the microcontroller board. NOTE: it is advisable to remove power from the AVR Microcontroller Board any time you plug or unplug the ISP cable.
5. The AVR Microcontroller Board is now connected to the PC and is ready to be programmed.

Setup PonyProg2000

1. Make sure the Microcontroller Board is properly connected as stated in Section 3.2.1 and then start up the PonyProg2000 program.
2. Click OK to remove the splash screen and get access to the program. Also note, if you are like the author, you might want to check the box labeled Disable Sound first.
3. It will then bring up a prompt telling you to calibrate before any read/write operations. Click OK to close this dialog.
4. Another prompt asking to run setup will appear, click OK to close this as well.
5. The first thing is to select the correct device. There are two pull down menus on the right that do just that. Select AVR micro from the first pull down menu and ATmega128 from the second.
6. Now run the calibration by navigating the menu options, Setup-¿Calibration.
7. Click YES on the dialog to begin calibration. It should go very quick. A dialog will then pop and state that the calibration is complete. If you do not get this, then close all the other running applications and try again.

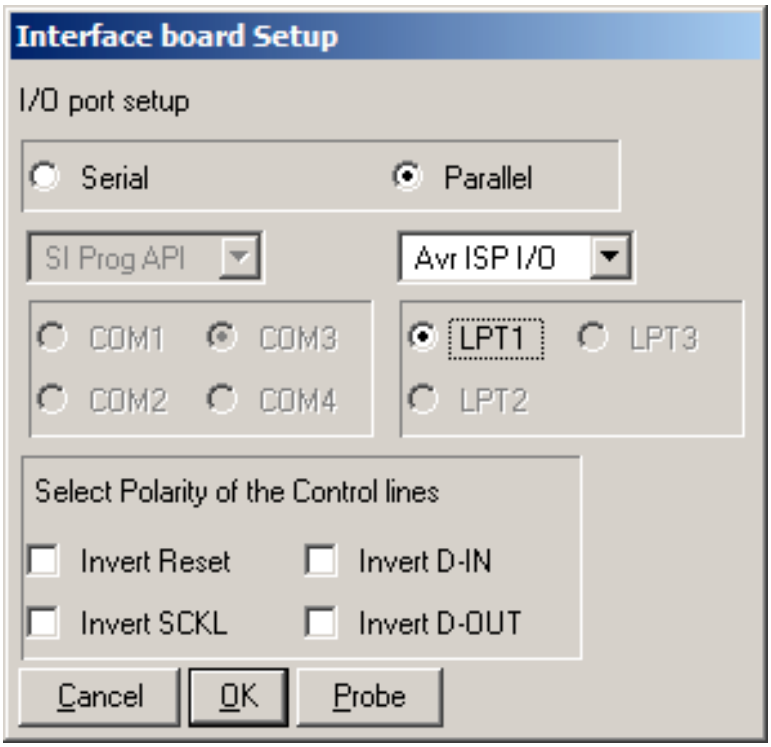


Figure C.1: PonyProg2000 Interface Board Setup.

8. Now run the interface setup by navigating the menu options, Setup-¿Interface Setup.
9. A new dialog appears, fill in the information to match the figure below.
10. Click Probe to make sure the connection is ok and then click OK.
11. With everything setup correctly, you should have a screen that looks similar to the following figure. You have now successfully setup PonyProg2000 for the ATmega128.

Figure 5: Fully Initialized PonyProg2000

C.2 Uploading a Program

After all the initialization is complete, uploading a program onto the AVR Micro-controller Board is easy. You will, of course, need a program to upload. Here is the step-by-step tutorial on how to do this process.

1. Compile your program in AVRStudio4. This generates a usable AVR program in the form of an HEX file. This is the actual binary data that gets uploaded into the ATmega128.

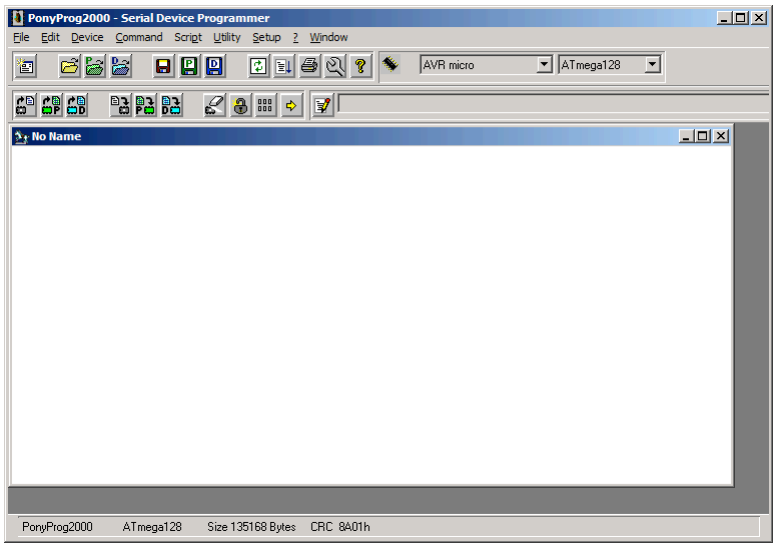


Figure C.2: Fully Initialized PonyProg2000.

2. Click on the icon or File-¿Open Device File to load a program into PonyProg2000.
3. Navigate to the directory where your AVRStudio4 project is located.
4. Select *.hex from the pull down menu labeled Files of type:.
5. A file should appear that has the same name of your project in the AVRStudio4. Click on this name and then click Open to load the program.
6. A successfully loaded program should look similar to the following figure.
7. Finally, upload the program onto the ATmega128 by clicking on the icon or by the menu, Command-¿Write All. Also, Ctrl-W will work. This should bring up a dialog that confirms your request to program the device. Click Yes to begin upload.
8. A progress bar will appear. It will write to the device, and verify it twice. This takes about a minute or two to complete. When it is finish, a dialog appears telling that it was successful or not. Click on OK to go back to PonyProg2000.
9. Once the program has been written to the ATmega128, the ATmega128 will automatically reset and start running your program. Congratulations, you have now successfully uploaded your program

During any stage of development, the program is not generally perfect from the start. Several revisions must be made and a lot of debugging. This generally means that a program will have to be downloaded several times into the chip. Well, you do not have to reload the program each time you make changes; there is a nice little function that does this for us. First, keep both AVRStudio4 and PonyProg2000 open. Make your change in AVRStudio4 and then recompile your program (F7). Now, switch over to PonyProg2000, it should still have the last version of the program loaded. Click the icon to refresh the code. Doing so will automatically update the loaded program file in the window to reflect

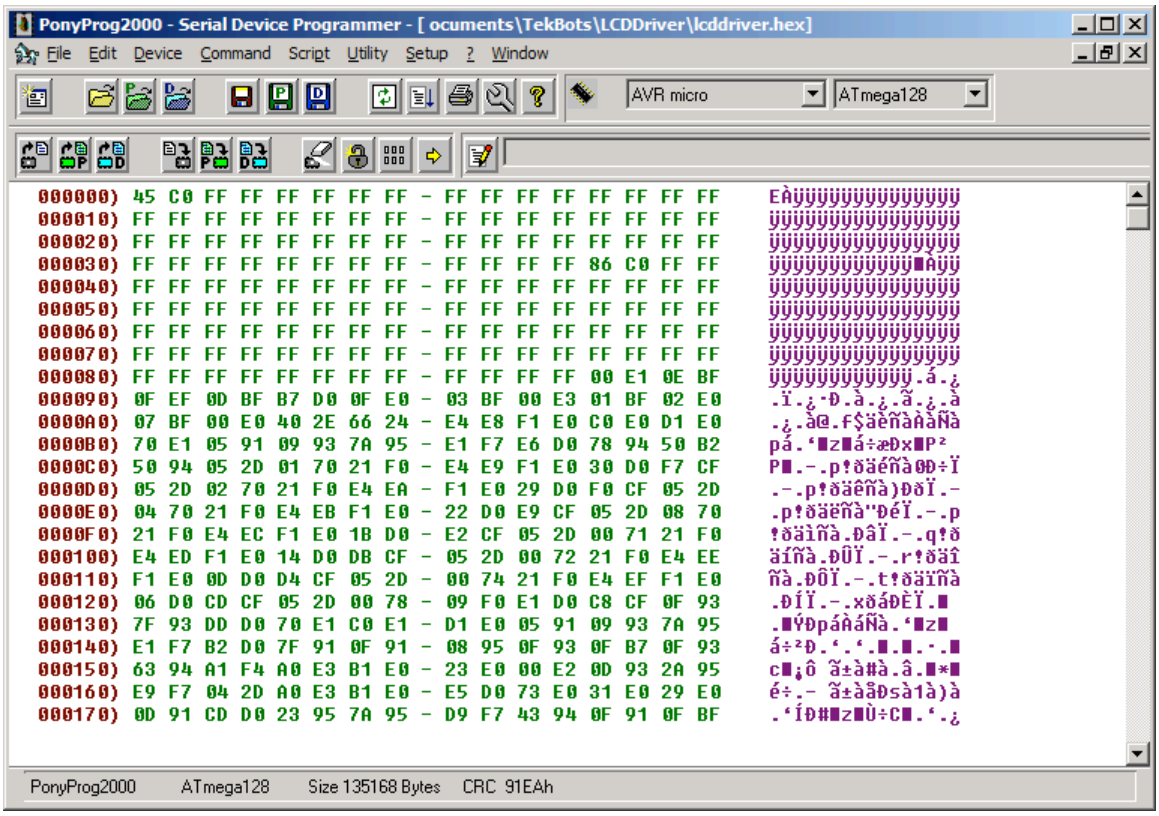


Figure C.3: Successfully Loaded AVR Hex Program File in PonyProg2000.

any changes made. Finally, just upload the program again. This is really a time saver and you will use it a lot during the development of projects.

Feel free to explore the rest that PonyProg2000 has to offer. This document only covers the basics and does not go into any more detail about PonyProg2000. There is one more mention though, be careful of any modification to the fuse bits as they can several alter the way the ATmega128 functions.

Index

∞ (infinity), 298
n-by-n division, 293
n-by-$n/2$ division, 293
(, 238
(`ANDI` *Logical AND with immediate*), 210
(`COM` *One's complement*), 211
(`DEC` *Decrement*), 211
(`INC` *Increment*), 211
(`IN` *Load an I/O location to register*), 210
(`IN` *Store register to I/O location*), 210
(`LDI` *Load immediate*), 210
(`LSR` *Logical shift right*), 211
(`ORI` *Logical OR with immediate*), 210
(`ROR` *Rotate right through carry*), 211
`ADD x`, 51
`ADD` (*Add without carry*), 210, 224
`ADD` (*Add without cary*), 252
`ADIW` (*Add immediate to word*), 224
`AND` (*Logical AND*), 210
`BNE x`, 57
`BREQ` (*Branch if equal*), 212, 244
`BRGE` (*Branch if greater or equal*), 212
`BRLT` (*Branch if less than*), 212
`CALL k`, 222
`CALL` (*Log call to a subroutine*), 233
`CALL` (*Long call to a subroutine*), 212, 236, 256
`CLR` (`Clear register`), 224
`CP` (*Compare*), 210
`ICALL` (*Indirect call to subroutine*), 236
`INC` (*Increment*), 224
`IN` (*Load an I/O location to register*), 228
`IN` (*Store register an I/O location*), 228
`J x`, 55
`JMP` (*Jump*), 233
`JMP k`, 222
`JMP` (*Jump*), 212
`LDA (x)`, 58
`LDA -(x)`, 60
`LDA x`, 50
`LDD`, 219
`LDD` (*Load indirect with displacement from data space to register*), 211
`LPM` (*Load program memory*), 236
`MOV` (*Copy register word*), 225
`MOV` (*Copy register*), 210, 225
`MUL` (*Multiply unsigned*), 224
`NEG` (*Two's complement*), 224
`ORI` (*Logical OR with immediate*), 224, 252
`POP` (*Pop register from stack*), 236
`PUSH` (*Push register on stack*), 236
`RCALL`, 77
`RCALL` (*Relative call to subroutine*), 236
`RCALL` *Relative call to subroutine*, 211
`RET` (*Return from Subroutine*), 239
`RET` (*Return from subroutine*), 236
`RJMP`, 77
`RJMP` *Relative jump*, 211
`SBIW` (*Subtract immediate from word*), 224
`STA x`, 51
`STD` (*Store indirect with displacement from register to data space*), 211
`SUBI` (*Subtract immediate*), 224
`SUB` (*Subtract without carry*), 224
`bne` *Branch If Not Equal*), 275
'X' - don't care, 294
16-bit Timer/Counter, 138
2-address instruction format, 210
2-stage datapath, 212
64 I/O register address space, 225
8-bit Timer/Counter, 138
8-bit data transfers, 225
96-entry Register File, 228

Output Compare (OC), 139
personal computer (PC), 68

A, 217
ACC register, 281, 294
accumulated error, 304
Accumulator (AC), 45
add delay, 264
Add-and-Shift multiplier, 281
Address Adder, 218
address register (AR), 228
address registers, 71
address registers (ARs), 211
addressing mode, 29, 228
Alignment Unit, 217
American Standard Code for Information Interchange ASCII), 259

Analog-to-Digital Converter (ADC), 127
analog-to-digital converter (ADC), 69
Arithmetic and Logic instructions, 223
arithmetic and logic instructions, 83
Arithmetic and Logic Unit (ALU), 215
arithmetic exceptions, 298
arithmetic instructions, 26
Arithmetic Logic Unit (ALU), 206
arithmetic logic units (ALUs), 37
arithmetic shift, 266
array, 174
array initialization, 174
array multipliers, 289
assembled, 22
assembly language programming, 22
AVR 8-bit microcontroller, 68
AVR instruction formats, 210

base, 296
bias, 297
bidirectional shift register with parallel load, 197
binary decoder, 189
binary division, 291
binary operation, 224
Bit-pair Recoding, 287
bit-slice, 271
Block Carry Lookahead Network, 279, 280
BlockCLAs, 279
Booth Recoding, 283
BOTTOM, 140
Branch and Jump instructions, 223
branch target address, 211, 230
buses, 37
busy-waiting, 126

C (carry, 275
cache memory, 39
carry (C) flag, 58
carry bit (C-bit), 197
Carry Lookahead Adders (CLAs), 276
Carry Lookahead Network, 277
carry-in, 267
carry-out, 263, 267
Cascaded CLAs, 278
Central Processing Unit (CPU), 18, 37
Chip Select, 204
chip set, 18
circular shift, 267
clock cycle time, 125
complementer, 270
computer architecture, 37, 209
computer organization, 37
condition code, 275
condition codes, 57, 72
conditional branches, 84, 230
constants, 77, 155
Control and Alignment Unit (CAU), 239
control signals, 248
control transfer instructions, 26, 83, 230
control unit, 37
Control Unit (CU), 237
control unit (CU), 209

Data Memory, 69, 225
Data Memory Address Register (DMAR), 212, 214
Data Transfer instructions, 223
data transfer instructions, 26, 78, 225
data transfer operations, 209
datapath, 37, 209
decimal instructions, 26
decoder, 188
demultiplexer (DEMUX), 188
denormalized, 298
destination register, 39, 210
direct addressing, 75
direct format, 212
direct jump, 233
direct or absolute addressing, 29
direct Program Memory addressing, 77
Direct Subroutine Call (`CALL k`), 237
displacement, 77
displacement format, 211
dividend, 294
DIVIDEND register, 294
division, 291
divisor, 294
dynamic RAMs (DRAMs), 204

effective address, 29, 75
EICRA, 130
EICRB, 130
embedded computers, 17
embedded systems, 17, 37, 67
enable, 189
End-Around Carry (EAC), 263
enhance datapath, 212
enhanced datapath, 235
Ethernet, 18
EX1, 248
EX2, 248
EX3, 248
excess, 297
excess-1023, 299
excess-127, 297
Exclusive-OR (EOR), 216
Execute (EX) stage, 242
execute cycle, 47
Execute stage, 212, 221, 223
exponent, 296
External Interrupt Flag Register (`EIFR`), 130

External Interrupt Mask Register (EIMSK), 130

fan-in, 278
fan-in Problem, 188
Fetch (IF) stage, 222
Fetch and Execute cycles, 248
fetch cycle, 47
Fetch stage, 212, 221
finite state diagram, 248
Finite State Machine (FSM), 239
fixed-point format, 259
flip-flop, 191
flip-flops, 190
floating-point format, 259
floating-point instructions, 26
floating-point numbers, 296
full adder, 267
function table, 267

gate delay (gd), 269
General Purpose Registers (GPRs), 69, 71, 254
general purpose registers (GPRs), 211
general-purpose computers, 17, 37
generation term, 276
Global Interrupt Enable (I-bit), 128
global varaible, 152
Graphics Processing Unit (GPU), 18
guard bit, 302

Hardware Description Language (HDL), 39
hidden 1-bit, 297
high impedance state, 117
high-level language, 21

I/O Controller, 18
I/O format, 210
I/O instructions, 26, 225
I/O pin, 116
I/O ports, 115
identifier, 151
IEEE 754 Floating-Point format, 296
IF, 248
immediate format, 210
immediate value, 210
immediate values, 77
indirect addressing mode, 58
indirect jumps (`IJMP`), 236
indirect Program Memory addressing, 77
indirection, 59
infinity, floating-point, 297
Input/Output (I/O), 37
Input/Output(I/O), 115
instruction cycle, 252
instruction cycles, 221
instruction decoder, 189
Instruction Pointer, 212
Instruction Register (IR), 45, 222
Instruction register (IR), 213
Instruction Register (PC), 212
Instruction Set Architecture (ISA), 23
Instruction Set Architectures (ISAs), 281
Internal Data Bus, 43
interrupt, 126
Interrupt Service Routine (ISR), 127, 130
interrupt service routine (ISR), 131
interrupt vector table, 131
interrupt vectors, 119

K, 217
k, 217
Karnaugh map (K-map), 267

L1 cache, 18
L3 cache, 18
last-in, first-out (LIFO, 72
latches, 190
least significant bit (LSB), 111
little endian, 111
local variable, 152
logical instructions, 26
logical shift, 266

machine instructions, 21, 87
mantissa, 296
master, 193
MAX, 140
members, 175
memory, 37
Memory Address Register (MAR), 45
Memory Controller, 18
Memory Data Register (MAR), 45
Memory Data Register (MDR), 236
memory element, 190
memory indirect addressing, 29
memory-mapped I/O, 34
micro-operation, 222
microarchitecture, 209
microcontrollers, 67
microopertions, 39
mnemonics, 87
Modified Booth algorithm, 287
Modified Full Adders (MFAs), 277
most significant bit (MSB), 111
MPY register, 281
MULT register, 281
multi-cycle implementation, 221, 248
multiplexer, 220
multiplexer (MUX), 186
multiplicand, 281
multiplier, 281
MUXA, 220
MUXB, 220

MUXC, 220
MUXD, 220
MUXF, 220
MUXG, 220
MUXI, 220
MUXJ, 220

N (negative), 275
NaN (Not a Number), 298
negative (N) flag, 58
negative edge-triggered D flip-flop, 193
negative partial remainder, 294
Next PC (NPC), 212, 213
next PC (NPC), 222
non-numeric data, 259
non-restoring, 293
non-restoring division, 294
normalized, 297, 298
numeric data, 259

opcode bits, 210
opcode extension, 32
operand, 27
output compare interrupt, 139
Output Compare Registers , 139
overflow, 264, 298
overflow (V) flag, 58

pad bit, 285, 286
partial product, 281
partial remainder, 293
PC-relative branch, 89, 230
PC-relative formats, 211
PC-relative target address, 244
PCh, 236
PCl, 236
pipeline implementation, 221
pointers, 59
Port A-H Data Direction Register (DDRA-H), 116
Port A-H Data Register (PORTA-H), 116
positive partial remainder, 294
post-increment, 230
post-normalization, 303
pre-decrement, 230
pre-normalization, 300
precision, 296
Prescalar, 140
Program Counter (PC), 45, 72, 212
Program Memory, 69, 221
Program Memory Address Register (PMAR), 236
propagation term, 276
prototype, 172
pseudo-AVR microarchitecture, 209
pull-up register, 123

q, 217
Quiet NaN (QNaN), 298
quotient, 292

radix, 296
radix point, 297
random access memory (RAM), 38
range, 296
RARh, 236
RARl, 236
Rd, 217
real numbers, 296
Recursive functions, 173
Reduced Instruction Set Computer (RISC), 25
register, 197
Register Address Logic, 217, 225, 242, 256
register addressing, 73
register file, 206
register file (RF), 217
register identifier, 29
register identifiers, 210, 218
register indirect addressing, 29
register indirect with displacement, 228
register indirect with post-decrement, 230
register indirect with pre-decrement, 230
Register Transfer Language (RTL), 39
register-indirect, 228
registers, 37
remainder, 292
restoring, 293
return address, 91, 236, 241
Return Address Register (RAR), 236
Ripple Carry Adder, 267
rounding, 302
rounding bit, 302
rounding modes, 303
Rr, 217

self modifying code, 30
sensor, 115
sequence control, 248
serial-in, serial-out 4-bit shift register, 197
Set-Reset (S-R) latch, 191
shift register, 197
sign-extended, 283
Sign-extension, 214
sign-extension, 214
Signaling NaN($SNaN$), 298
single-precision, 296
slave, 193
source operands, 210
source register, 39
Special Function Registers (SFR), 177
special-purpose registers, 212
Stack Pointer (SP), 72, 214
state, 248

state table, 248
Static RAM (SRAM), 69, 201
Status Register (SREG), 72
sticky bit, 302
string instructions, 26
structure, 175
sum, 267
system instructions, 26
System-on-Chip (SoC), 18

target address, 212, 217, 220, 222, 230, 233
TCNT0, 138
TCNT1, 138
TCNT2, 138
TCNT3, 138
Tekbots, 119
Time Counter Control Register 0 (TCCR0), 140
Timer Oscillator 1 (TOSC1), 140
Timer Oscillator 2 (TOSC2), 140
TOP, 140
truth table, 267
two read-port, two write-port Register File, 217
two-operand format, 210

unconditional branches, 230
underflow, 298
Universal Synchronous Asynchronous Receiver/-
Transmitter (USART), 127

V (overflow), 275
validation, 210
variables, 152
verification, 210

Waveform Generator, 140
WLAN, 18

Z (zero), 275
zero, floating-point, 297
zero-fill, 214, 215