Computer Organization and Assembly Language Programming: Embedded Systems Perspective

Ben Lee

Oregon State University

School of Electrical Engineering and Computer Science

Last updated January 2, 2019

ii

Preface

Under construction!!!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

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 only vaguely aware of – *embedded systems* or *computers*. Unlike general-purpose desktop and laptop computers, embedded systems are designed to perform one or a few dedicated functions, and more importantly, are *embedded* as a part of a complete device that often includes hardware and other mechanical parts. For example, remote keyless entry systems for automobiles have embedded computers that can send and receive special code to lock/unlock doors as well as other functionalities.

The meaning of an embedded system 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 embedded systems that perform *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/tablet devices with mobile operating systems (e.g., Google Android, Microsoft Windows Mobile, Apple iOS, RIM Blackberry, etc.) and the 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 of an embedded system is *any system with a computer that is not a desktop or laptop*!

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% of 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 systems to control Heating, Ventilating, and Air Conditioning (HVAC). In fact, embedded systems account for the most of the world's production of microprocessors! Therefore, understanding how they are programmed and how their internal structure is organized are essential for future engineers and computer scientists.

1.2   Spectrum of Computers and Their Processors

The spectrum of computers varies as much as their intended applications. *General-Purpose systems*, such as desktops, laptops, and servers, are designed to handle a multitude of applications from document processing to video conferencing. As such, these systems contain an array of peripheral devices controlled by a powerful processor. Figure 1.1(a) illustrates the organization of a typical desktop system (server and laptop systems are also fundamentally similar). At the time of the writing of this book, such a 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 Gigabytes (GB) and even have a separate chip, called Graphics Processing Unit (GPU), to process graphics for displays. Besides the processor,
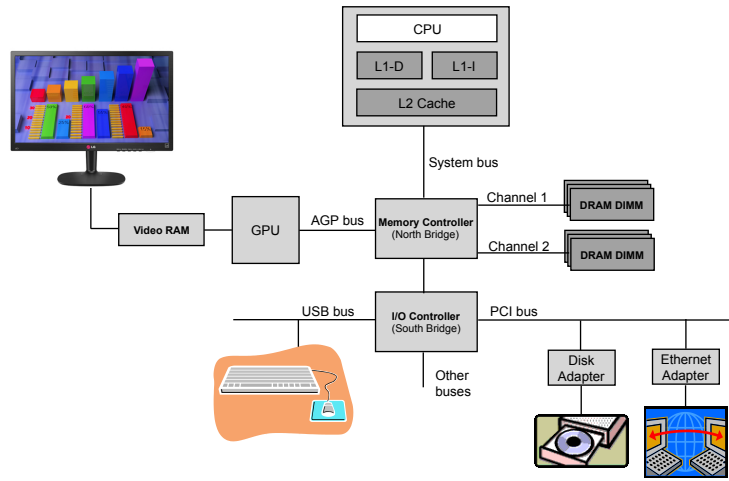
(a) Desktop system.
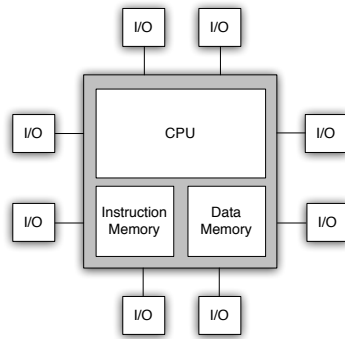
(b) Embedded system.

Figure 1.1: Desktop versus Embedded systems.

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 shown in Figure 1.1(b) is a self-contained *System-on-Chip* (SoC) that contains all the basic components needed to control I/O devices. The complexity, performance, power, and cost require-

ments of these embedded systems depend on their intended purpose. For example, a low-end embedded system includes an 8-bit or a 16-bit processor running at few to tens of MHz with Instruction and Data Memories of tens to hundreds of Kilobytes (KB). Unlike desktop systems, the I/O devices they control are usually simple devices, such as sensors, motors, and other mechanical devices, and are low power due to their size and speed. On the other hand, high-end embedded systems contain 32-bit processors clocked at several hundred MHz range with memories in the order of Megabytes (MB). Examples of such systems include PMPs, GPSs, car infotainment systems, feature phones, etc.

In recent years, a new class of mobile devices have emerged that straddle between embedded systems and general-purpose computers. These devices include smartphones and pad/tablet devices, which run at GHz range clock with large memories and resemble their desktop/laptop counterparts by having operating systems, graphics processors, and even multiple cores. From a software perspective, these ultra high-end embedded systems with vast arrays of Applications Programming Interfaces (APIs) can be viewed as programming resource constrained general-purpose computers. In fact, some of these mobile devices are based on processors from low-end general-purpose computers, e.g., Intel Atom processor. Therefore, the line between embedded systems and general-purpose computers is becoming more and more blurred.

1.3   Objectives of the book

One of the major objectives of this book is to understand the interrelationship between hardware and software. Most students new to modern computing concepts believe these are two distinct topics. However, this is not the case at all and the topics covered in this book will show that computer organization and assembly language is the interface where both electrical engineering and computer science disciplines merge.

In order to understand why this is the case, consider the hierarchical layers of problem solving in computing systems shown in Figure 1.2. From the perspective of a software designer, a problem statement is defined using a language-independent algorithm or a psuedocode. Then, the algorithm is coded using a high-level language, such as C/C++. On the other hand, a hardware designer implements basic digital components using existing circuit technology, and then these components are organized into a microarchitecture to implement a processor. The point at which both of these
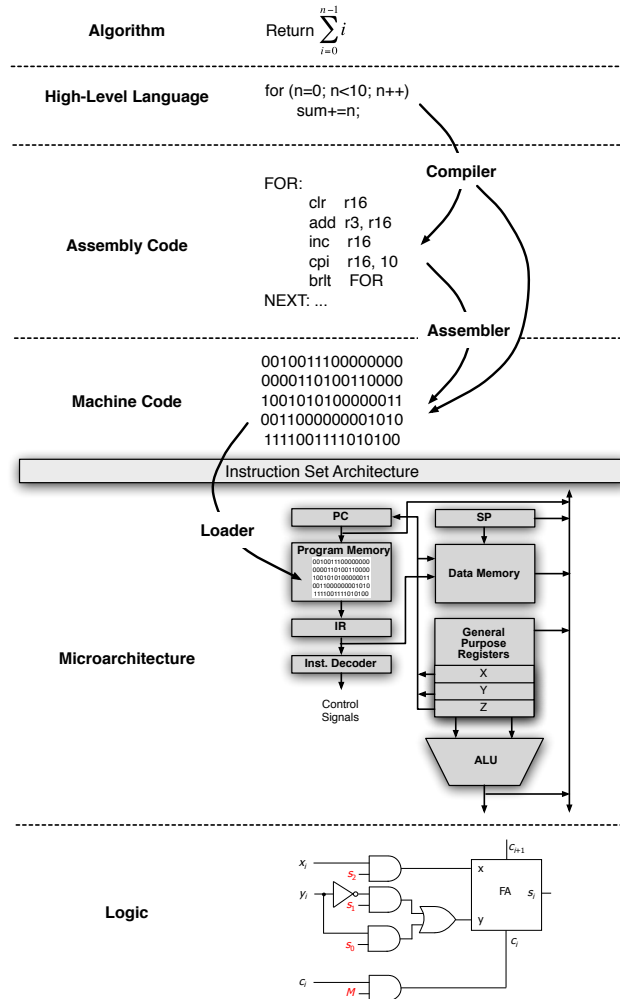
Figure 1.2: Computing system hierarchy.

perspectives merge is the *Instruction Set Architecture* (ISA), which is the basic set of operations or *assembly instructions* supported by the processor.

How efficiently a program runs on the processor depends on how well the compiler translates the high-level language program to a sequence of assembly instructions, i.e., an assembly program, as well as how well the microarchitecture that executes the assembly instructions is designed. This requires good understanding of both software and hardware. For example,

compiler writers have to be very familiar with the ISA of the processor to perform machine-dependent optimizations to minimize program execution time. On the other hand, a hardware designer has to understand the operations required by the ISA to implement a microarchitecture to execute the assembly program as fast as possible within given physical constraints such as complexity, memory size, and in some cases power.

Therefore, the focus of this book is the interfaces between language and ISA, and between ISA and microarchitecture. However, the topics covered by these layers have a much broader implication on both software and hardware designers. Understanding the essence of these concepts makes both software and hardware designers better at what they do. For example, programmers can write better programs by understanding how processor execute their programs, while hardware designers can design better processors by understanding the operational requirements of programs.

1.4   Roadmap for the Rest of the Book

The chapters in this book are organized to cover the various topics of the hierarchical layers of a computing system.

Chapter 2 discusses the fundamental concepts of Assembly Language programming. We start off by discussing what assembly language is and how it is used to directly interact with processors. This introduction is followed by a discussion of the core of assembly language and processor design — *Instruction Set Architecture* (ISA). We discuss the variety of factors that influence the design of an ISA and their effect on programming as well as processor performance. Afterwards, we discuss how instructions in an ISA are represented as an Instruction format consisting of different fields. Finally, the chapter concludes with an example ISA design to help students understand how the various issues presented in this chapter are considered.

Chapter 3 presents the basic computer organization concepts. This is done by first discussing the major components required by a processor, i.e., datapath, control unit, memory, and Input/Output (I/O). This is followed by a discussion of data transfer operations that dictate how these components interact. The final section of this chapter presents a simple processor design based on the example ISA presented in Chapter 2.

Chapter 4 presents assembly language programming using AVR microcontrollers. The chapter starts with important concepts that distinguish between high-level language programming and assembly language programming, and issues programmers have to be aware of when they are program-

ming in assembly. This is followed by a discussion on the characteristics of the AVR architecture, which includes registers and memories, addressing modes, and instructions. Throughout this discussion, numerous example pieces of assembly code are used to illustrate not only assembly programming techniques but also how they are related to equivalent high-level language code. This chapter also includes a discussion of how AVR assembly instructions are mapped to machine instructions. The purpose of this discussion is to show how symbolic representation of assembly instructions are mapped to 0's and 1's in the instruction format. This is important for understanding how a processor decodes and executes instructions, which will be covered in Chapter 8. The chapter also covers assembly coding techniques, which will help you to write well-structured assembly codes that are easy to understand and debug. Then, more advanced examples of assembly code are presented to illustrate how any high-level language programming constructs can be implemented in assembly. Finally, the chapter concludes by illustrating how an assembly program is mapped to binaries revealing what a processor has to do to fetch and decode instructions of a program.

Chapter 5 discusses one of the most important functionalities of microcontrollers – I/O! We first discuss the basic I/O capabilities of the AVR architecture, including ports and control registers for setting up these ports. Then, the concept of interrupt is introduced with an elaborate example of controlling Tekbots. The rest of the chapter presents various peripheral features available in the AVR architecture, which includes Timers/Counters, Universal Synchronous Asynchronous Receiver/Transmitter (USART), Analog-to-Digital Converter (ADC), Serial Peripheral Interface (SPI), Two-Wire Interface (TWI), and Analog Comparator.

Chapter 6 presents embedded C programming. This chapter reviews not only the basics of C programming language for completeness, but also presents a set of extensions to address common issues among different embedded systems. These issues include I/O operations, multiple distinct memories, embedding assembly, and fixed-point arithmetic.

Chapter 7 discusses the various digital components in a processor. These include multiplexers, decoders, memory elements, registers, and memory. This chapter is meant to be not only a review of basics of digital design, but also to emphasize the design of the critical components in a processor architecture.

Chapter 8 presents the organization of the AVR architecture. This chapter is unique because, to the best of my knowledge, it is the only published detailed description of the inner workings of the AVR architecture. The chapter starts with explanation of the various AVR instruction formats and

their fields. This is followed by a description of all the digital components in the microarchitecture. We then illustrate how AVR assembly instructions are executed and their timing requirements. The design of a Control Unit that orchestrates instruction execution is then discussed. Finally, the concept of pipelining is explained and how it is used in the AVR microarchitecture.

Finally, *Chapter 9* discusses the workhorse of a processor — Arithmetic and Logic Unit (ALU). We first review number systems and basic shift operations. This is followed by a discussion of adders, which is the core of an ALU, and surrounding logic that allows for implementation of a basic ALU. The ALU design is then extended to support fixed-point multiplication, including fast multiplication, and division. The chapter concludes with algorithms for Floating-point operations.

Chapter 2

Assembly Language Fundamentals

Contents

2.1 Introduction

Most of us write programs using *high-level languages*, such as C/C++, Java, or Fortran, and rely on sophisticated software libraries that implement complex functions to simplify our programming tasks. These programs are then compiled using another sophisticated program, i.e., compiler, to generate binary executables that processors understand. During execution, programs may also rely on run-time support provided by an operating system (OS), 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, designing a processor or writing a compiler 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 0's and 1's 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 code 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 unlike high-level languages, it 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 the 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 *device drivers* and interrupt facilities, requires a clear understanding of Input/Output (I/O) operations provided by an assembly language. Fourth, writing real-time applications that require 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.
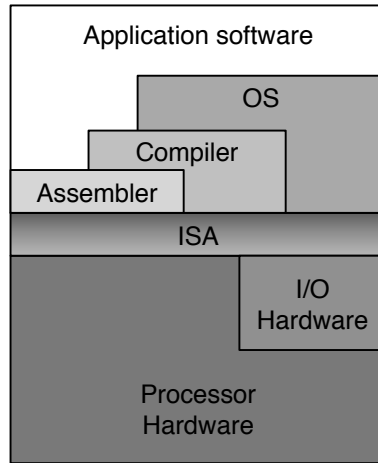
Figure 2.1: Simplified hierarchical view of software and hardware, and the role of ISA.

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.

The *Instruction Set Architecture* (ISA) is the portion of the processor visible to a programmer or a compiler. Therefore, ISA basically acts as the *interface* for either programmers to directly write or compiler to transform processor-independent programs into processor-dependent programs, or *machine language programs*. 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 written using instructions from the ISA.

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 the factors that influence the design of an ISA and how they affect 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 is an ISA 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 the processor hardware. These factors include

- Operations provided in the ISA;
- Number of explicit operands named per instruction;
- Operand locations and how they 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 instructions provided by 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
- Programming vs. hardware complexity.

The following discusses these issues in more detail.

Functional Completeness

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 (i.e., floating-point unit). This will in turn improve speed but increase complexity and cost.

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 allow information to be copied from one location to another either in the processor's internal memory (registers) or in the external 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 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 act as 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 (PCs) that run Windows, Mac OS X, or Linux 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 complexity of the processor and may not be worth it if these operations are rarely used.

Instruction Efficiency

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 again on how the processor will be used. 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 less powerful instructions. On the other hand, processors designed for PCs require powerful instructions running at high speeds.

Programming vs. Hardware Complexity

Simplicity of hardware design and/or programming is directly related to the two previous mentioned factors. From a programmer's point-of-view, powerful instructions will make programming easier while the complexity of the hardware increases. On the other hand, simple instructions leads to more complex programs, but results in simpler hardware.

Based on the aforementioned discussions, the proper 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.

2.3.2 Number of Operands per Instruction

The second factor in ISA design is the number of operands explicitly specified in the instruction. An *operand* refers to the data to be manipulated or operated on by an instruction, and can be in a register, in a memory location, or a literal constant. The number of operands to be specified in an instruction format is at most three (two source operands and a destination) due to the nature of binary operations performed by the Arithmetic and Logic Unit (ALU). 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

```
add   z, x, y, goto q
```

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 z ← x+y. In addition to the add operation, 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. Why is this an issue? Suppose

x, y, and z are memory addresses and the size of the memory is 4 KB.
Then, the instruction format requires 12 bits $\times$ 4 = 48 bits to specify just
the operands and additional bits to specify that this is an add operation.
Clearly, this cannot be supported even with a 32-bit instruction format.

The way ISA designers get around this problem is to get rid of q and use
registers rather than memory locations. This leads to the following *3-address
instruction* format:

- 3-address instruction

```
add    z, x, y;    q is implied
```

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. In addition, using a
small number of General Purpose Registers (GPRs), say 16 of them, reduces
the number of bits required to specify an operand from 12 bits for 4 KB of
memory to 4 bits for 16 registers. Now the three operands can be specified
in a 32- or even 16-bit instruction format. The 3-address instruction format
is typically used in high-end microprocessors for PCs and mobile computing
platforms, such as cell-phones, tablet computers (e.g., iPad), and Portable
Media Players (e.g., iPod).

Although a 32-bit instruction format can easily support three addresses,
supporting these with a 16-bit instruction formats is difficult unless the
number GPRs is significantly reduced. However, decreasing the number
of GPRs reduces the flexibility of having registers in the first place. The
following *2-address instruction* format reduces the bit requirement and yet
retains the power of a 3-address instruction format:

- 2-address instruction

```
add    x, y
```

This format is similar to the 3-address instruction format discussed above,
but both destination and one of the source operands is defined by x and is
equivalent to the operation $x \leftarrow x + y$. This format can easily be supported

with 16 bits. The 2-address instruction format is typically used in mid-range microcontrollers, e.g., AVR.

Some processors use the *1-address instruction* format shown below:

- 1-address instruction: Accumulator-based architecture

```
add     x
```

This format defines only one explicit operand, and thus the length of the instruction format is very short. The second operand as well as destination are *implicitly* defined by a special register called the *Accumulator* (AC). Therefore, this instruction performs the operation $AC \leftarrow AC + x$. The disadvantage of this instruction format is that AC is involved in every operation, and thus additional instructions are needed to move the data between AC and a register or memory location. This leads to more instructions to accomplish the same task compared to using either 3-address or 2-address instruction formats. This instruction format is very common in low-end microcontrollers, e.g., 8051.

The following instruction format does not define any operands:

- 0-address instruction: Stack-based architecture

```
add
```

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 based on Reverse Polish Notation (RPN).

2.3.3 Operand Locations and How They are Specified

An operand can reside in an instruction, a register, or a memory location. Operand locations can be specified in a number of different ways, and the way this is done is called *addressing modes*. All ISAs support the following set of addressing modes:
- Immediate addressing
- Direct addressing
    - Memory direct addressing
    - Register direct addressing
- Indirect addressing
    - Memory indirect addressing
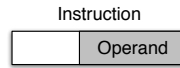
Figure 2.2: Immediate addressing mode - the operand is contained within the instruction.
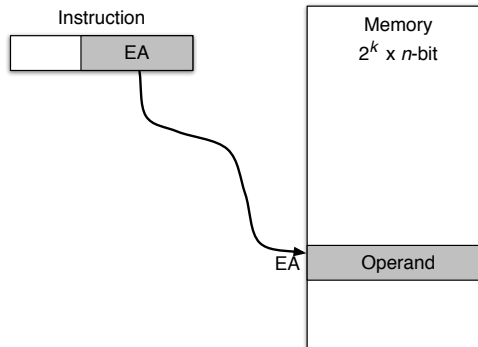
(a) Memory direct addressing.
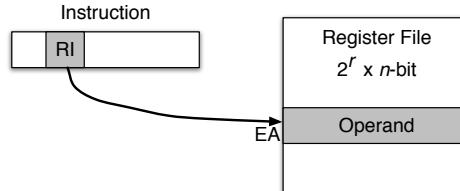
(b) Register direct addressing.

Figure 2.3: Direct addressing - the operand is in a memory location or a register.

      – Register indirect addressing

When an operand is located within the instruction, as shown Figure 2.2, it is called *immediate addressing*. This addressing mode allows for fast access to operands and is useful for coding constants or literal values that do not vary during program execution.

Figure 2.3 shows *direct* or *absolute addressing* that refers to operands in a memory location or a register. This addressing mode is used to access variables in memory. Figure 2.3(a) shows *memory direct addressing*, where a
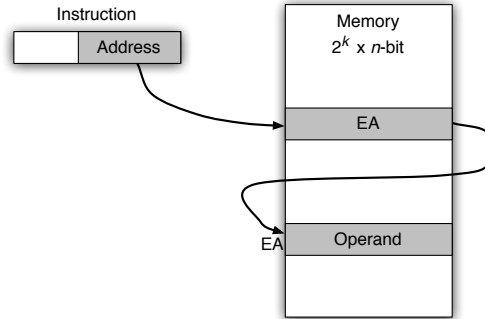
(a) Memory indirect addressing.
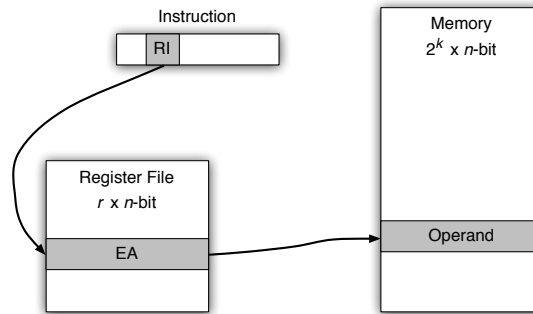
(b) Register indirect addressing.

Figure 2.4: Indirect addressing.

k-bit address, known as an *effective address* (EA), encoded in an instruction directly refers to an operand in memory. Figure 2.3(b) shows *register direct addressing* or simply *register addressing*, where an r-bit register address, known as a *register identifier* (RI), encoded in an instruction refers to an operand in a *register file*, which is an array of registers. Therefore, either EA or RI has to be encoded within the fixed size instruction, which is typically 16 or 32 bits.

Figure 2.4 shows *indirect addressing*, which uses a level of indirection to refer to operands in a memory location or a register. This addressing mode is used to implement *pointers* to variables in memory. Figure 2.4(a) shows an example of *memory indirect addressing*. Unlike memory direct addressing shown in Figure 2.3(a), the effective address is in a memory location instead of being encoded in the instruction. Figure 2.4(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; 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 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 present 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 a program language and a processor hardware,

opcode	field$_1$	...	field$_k$
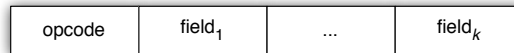
Figure 2.5: Instruction format.

and its layout is composed of fields of binary numbers. Figure 2.5 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 *operation code (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 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 same is true for address fields. For example, if the memory size is 4 K (i.e., $2^{12} = 4,096$) words, then the number of bits required to specify an address is 12 bits.

Obviously, there is a limit on the number of opcode bits and addresses an instruction format of either 16 or 32 bits can support. For example, with an 1-address instruction format of 16 bits, if the 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 2 K (or 2^{11}) words. Thus, there is a tradeoff between the number of operations supported and the size of memory for a given instruction format.

Instruction formats that support either a 2-address or a 3-address format get around this limitation by employing a register file. 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 two 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 three addresses with room to spare for other encoding possibilities.

A typical ISA supports well over one hundred different instructions. This
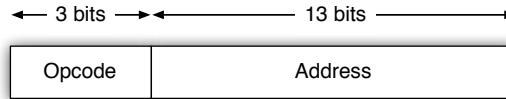
Figure 2.6: Instruction format of the pseudo-ISA.

will require 6 or more bits for the opcode field, which cannot be supported by static encoding schemes discussed above. Thus, a technique called *opcode extension* is used 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's decoder that other bits in the instruction format are used to encode additional instructions. Therefore, the size the opcode field and the location of its bits within the instruction format 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.

2.5   A pseudo-ISA

In order to put together the concepts discussed in this Chapter, this section discusses a design of a simple, pseudo-ISA. The instruction format for the pseudo-ISA is shown in Figure 2.6, 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 pseudo-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 $2^{13} = 8,192$ or 8 K 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 pseudo-ISA supports the following set of instructions:

- Data Transfer Instructions
    - LDA (*Load Accumulator*): Loads a memory word to the AC.
        * Usage: LDA   x ; x is a memory location
    - STA (*Store Accumulator*): Stores the content of the AC to memory.
        * Usage: STA   x ; x is a memory location
- Arithmetic and Logical Instructions
    - ADD (*Add to Accumulator*): Adds the content of the memory word

specified by the effective address to the content in the AC.
  * Usage: `ADD   x` ; x points to a memory location
  - `SUB` (*Subtract from accumulator*): Subtracts the content of the memory word specified by the effective address from the content in the AC.
    * Usage: `SUB   x` ; x points to a memory location
  - `NAND` (*logical NAND with accumulator*): Performs logical NAND between the contents of the memory word specified by the effective address and the AC.
    * Usage: `NAND   x` ; x points to a memory location
  - `SHFT` (*Shift*): The content of AC is shifted left by one bit. The bit shifted in is 0.
    * Usage: `SHFT` ; The content of AC is shifted left by one bit
- Control Transfer
  - `J` (*Jump*): Transfers the program control to the instruction specified by the target address.
    * Usage: `J   x` ; jump to instruction in memory location x
  - B*cc* (*Branch Conditionally*): Transfers the program control to the instruction specified by the target address based on condition *cc*.
    * Usage: `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 pseudo-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`, `SUB`, `NAND`, and `SHFT` allows for coding of any arbitrary arithmetic and logic functions. For example, a multiply operation can be performed by successive add and shift operations (see Chapter 9.5). 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 `IN` (Input) and `OUT` (Output) 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 `IN` and `OUT`, are used to perform I/O operations, e.g.,

```
      ;
      ; Equivalent assembly program
      ;
              .ORG  0       ; Program starts at location 0
              LDA   A       ; Load operand A to AC
              SHFT          ; Multiply A by 2
              SHFT          ; Multiply 2*A by 2
              ADD   B       ; Add operand 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 C = 4*A + B;.

AVR processors.

Now that we have discussed the operations of the eight instructions, let us write a small assembly program using our pseudo-ISA. The following example C program multiplies the variable `A` by 4 and adds it to the variable `B` and assigns the result to the variable `C`.

```
      /*    A simple C program    */

      main()
      {     int A = 83, B = -23, C = 0;

            C = 4*A + B;
      }
```

The equivalent assembly program for the above C code is shown in Figure 2.7, which consists of mnemonics, a data section, and assembly directives. *Mnemonics* represent the symbolic code for the assembly program and consist of `LDA`, `STA`, `ADD`, `SUB`, `NAND`, `SHFT`, `J`, and `BNZ` instructions. The *data section* defines data values 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 program at run-time. There are several types of assembly directives in the assembly program. For example, the assembly program starts with a `.ORG` directive and ends with a
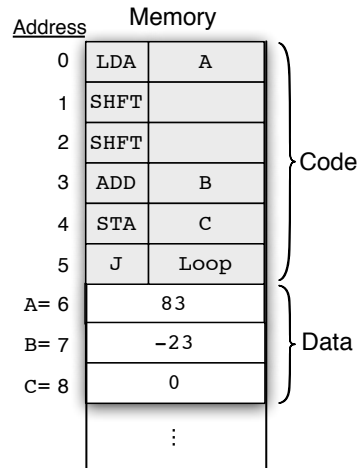
Figure 2.8: Assembly code and data in memory.

.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 content 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?'. Figure 2.8 shows what the assembly code and data look like 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 memory locations and assign decimal values. This is achieved in assembly language by using a special directive. In the example assembly program, the .DEC directive allocates a 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 (i.e., `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 do not 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

Contents

3.1   Introduction

Computer organization or *computer architecture*[1] 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 four aspects: instruction set architecture, microarchitecture, system design, and hardware design. The concept of *Instruction Set Architecture* (ISA) has already been discussed in Chapter 2. *Microarchitecture* is a lower level, more concrete and detailed, description of how the constituent parts of the processor are interconnected and how they interoperate in order to implement an ISA. *System design* involves how the processor and other peripheral components, such as memory, display, storage devices, etc., within a computer system come together. *Hardware design* represents the low-level implementation involving logic- and circuit-level implementation

---

[1]These terms will be used interchangeably throughout the book.
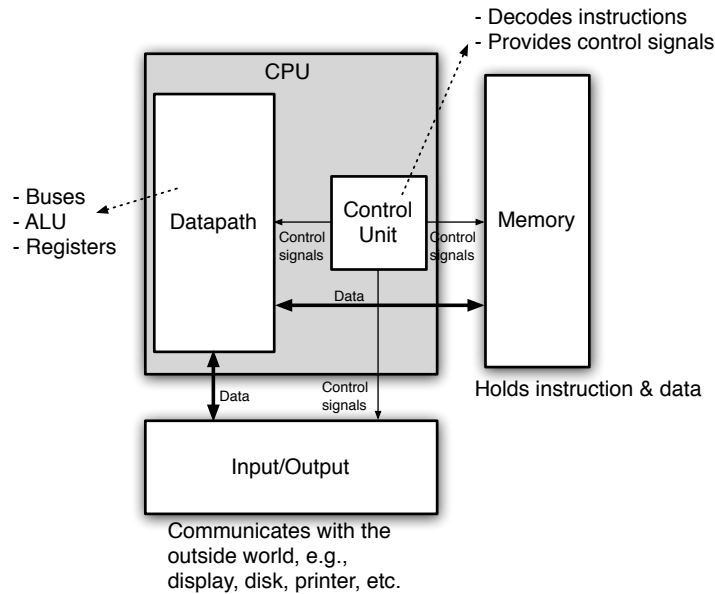
Figure 3.1: Computer Organization.

of the major components in a computer system. This is done through Hardware Description Languages (HDLs), integrated circuit design, and Printed Circuit Board (PCB) design.

The purpose of this chapter is to introduce fundamental concepts in computer organization. We will first discuss the organization of a typical computer system and its components. We will then discuss microarchitecture design by implementing a simple processor capable of executing the instructions in the pseudo-ISA discussed in Chapter 2. The concepts discussed here are meant to provide a basis for more detailed discussions of assembly language programming and microarchitecture design for the AVR processor in Chapters 4 and 8, respectively. Along the way, we will also include some discussions on digital design to gain a better understanding of how various components are implemented and interfaced. A more detailed discussion on the design of major components in a computer system will be provided in Chapter 7. I/O operations will be treated separately in Chapter 5.

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, motor controllers, sensors, etc., are basically designed this way. The memory holds instructions and data. I/O is used to communicate with the outside world (e.g., sensors, motors, network interfaces, printer, disk, display, mouse, etc.). The CPU contains the datapath and the Control Unit. The *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 by providing *control signals* to these various components.

3.2 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 in memory. 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 either *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 long. 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 4, 5, and 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 (PCs) have memory hierarchy consisting of all of the levels discussed thus

far. In contrast, simple microcontrollers for embedded systems may only have one or two levels of memory hierarchy (e.g., register file and memory).

3.3   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*. Each micro-operation consists of one or more *register transfer operations*. As the name suggest, a register transfer operation involves moving or copying the content of one register to another. The source and destination registers can be directly connected, share the same bus, or have some combinational logic, such as ALU or multiplexers, in between them. Therefore, specifying what needs to be done by the CPU in a micro-operation basically involves defining register transfer operation(s) that needs to be performed in one clock cycle. For this reason, the sequence of micro-operations required to implement an assembly instruction greatly depends on the microarchitecture.

Since there must be no ambiguity in defining the sequence of micro-operations, *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 for clear and concise specifications of part or all of a complex digital system, such as a processor.

The most fundamental register transfer operation is moving the content of one register to another register. This is 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 basic register transfer operation. There are several important assumptions being implied with the RTL description. First, all the registers involved in the register transfer are *clocked*, which allows multiple register transfers to occur simultaneously and synchronously. Second, the number of bits in both registers are equal and thus the number of
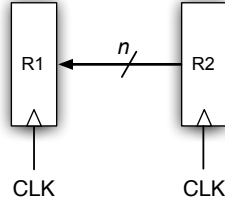
Figure 3.2: Simple register transfer between two n-bit registers.
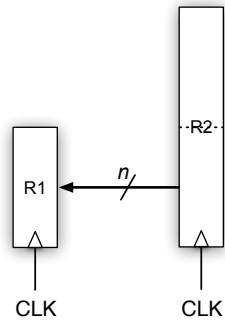
Figure 3.3: n-bit register transfer between two different sized registers.

bits being transferred does not have to be explicitly specified. Third, all n bits are transferred in parallel. Lastly, the order of the bits being transferred is preserved, 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. This type of operation is required when the size of $R2$ is different from $R1$. For example, the following statement indicates the transfer of bits $n - 1$ through 0 of $R2$ to $R1$, which is illustrated in Figure 3.3:

$$R1 \leftarrow R2(n - 1...0)$$

Note that the size of $R1$ is equal to the size of data being transferred.

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 the upper-half of $R2$ to the lower-half of $R1$, which is illustrated in Figure 3.4:

$$R1(L) \leftarrow R2(H)$$
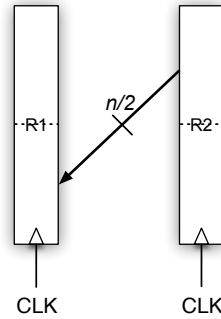
Figure 3.4: Register transfer of upper-half of R2 to lower-half of R1.

The content of a register can be subdivided into fields, each containing a specified number of bits. The following two unrelated statements specify transfers of the *field1* portion or the *field2* portion of $R2$ to $R1$.

$$R1 \leftarrow R2(field1)$$

or

$$R1 \leftarrow R2(field2)$$

Note that the number of bits for *field1* or *field2* is implied based on the size of its respective field, and it has to be equal to the size of $R1$.

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

$$R1 \leftarrow R3(field1), R2 \leftarrow R3(field2)$$
$$R1 \leftarrow R2, R2 \leftarrow R3$$
$$R1 \leftarrow R2, R2 \leftarrow R1$$

Figure 3.5 illustrates a simultaneous register transfer of m bits from $R3$ to $R1$ and $n - m$ bits from $R3$ to $R2$. Again, the sizes of $R1$ and $R2$ have to be equal to m and $n - m$.

Figure 3.6 illustrates a simultaneous register transfer 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 3.7 illustrates another example, where the output of $R2$ is available as the input for $R1$, and at the same time, the output of $R1$ is available
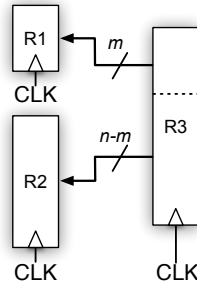
Figure 3.5: Simultaneous register transfer of $R3$ to $R2$ and $R3$ to $R1$.
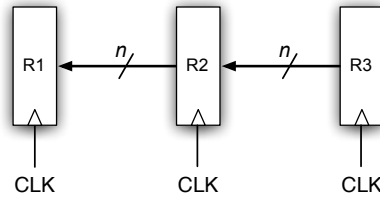
Figure 3.6: Simultaneous register transfer of $R3$ to $R2$ and $R2$ to $R1$.
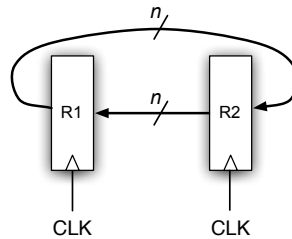
Figure 3.7: Simultaneous register transfer of $R2$ to $R1$ and $R1$ to $R2$.

as the input for $R2$. Thus, these contents are latched on to the respective registers at the edge of a clock.

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

$$R1 \leftarrow M[R2]$$
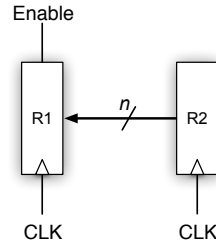
Figure 3.8: Conditional register transfer from $R2$ to $R1$.

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

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

The first four operations represent addition $(+)$, subtraction $(-)$, logical OR $(\vee)$, 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 register transfer operations that occur only when a certain condition is satisfied, not just every clock cycle. These cases can be represented by *conditional statements*. The following statement transfers the content of *R2* to *R1* when a condition is satisfied:

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

where *cond* represents the condition to be satisfied, such as equal, not equal, greater than, less than, etc. Figure 3.8 shows an example of a conditional register transfer, where the content of $R2$ is transferred to $R1$ only when the *Enable* signal is '1'.
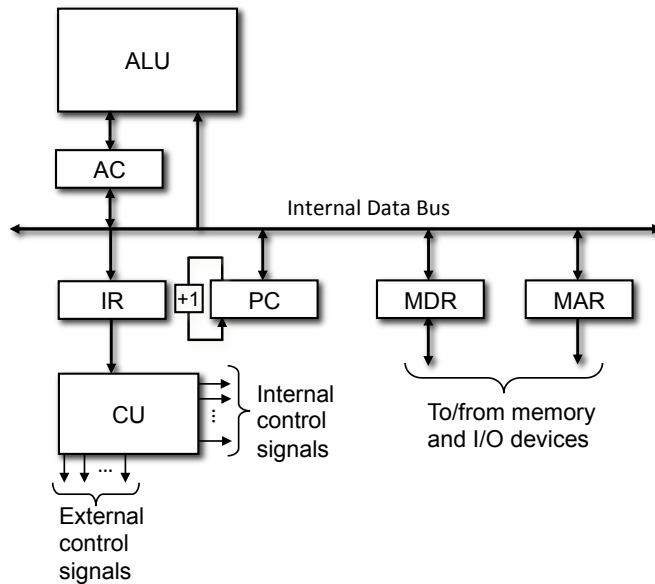
Figure 3.9: Organization of Pseudo-CPU.

3.4 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 unit) are "put together". Fig. 3.9 shows the organization of the *pseudo-CPU*, which is a very simple way to connect these components using a shared *Internal Data Bus*. Obviously, microarchitectures of commercial processors are much more complex, but nevertheless the pseudo-CPU shown in Fig. 3.9 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 good foundation for understanding how more complicated ISAs and microarchitectures work.

3.4.1 Major components of the pseudo-CPU

The pseudo-CPU consists of a set registers, an ALU, and a Control Unit. The functionalities of these components are explained in the following:

Registers

The pseudo-CPU has several important registers to hold information necessary for instruction fetch and execute, which are as follows:

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

As can be seen from the figure, the AC register serves as one of the inputs as well as the output for the ALU, which is consistent with the accumulator-based architecture of our pseudo-CPU (see Section 2.3.2). The other input to the ALU can be any of the registers connected to the Internal Data Bus. The IR register is used to latch the operation code (opcode) of an instruction or possibly an entire instruction so that the Control Unit (CU) can decode the opcode and provide an appropriate set of control signals to execute the instruction. Note that the PC register has a dedicated adder, which eliminates the need to use the ALU and extra transfers between registers to increment PC. Finally, the set of MDR and MAR registers allow access to the memory.

ALU

As the names suggests, ALU is a digital circuit that performs *arithmetic* and *logic* operations. The ALU is one of the most important components in a CPU since it is involved in practically every instruction execution. For example, in addition to arithmetic and logic operations, the ALU is also responsible for calculating effective addresses, branch target addresses, and any other operations required by instruction execution. We will discuss in detail the design of an ALU in Chapter 9. For now we will assume the ALU for the pseudo-CPU can perform any basic arithmetic and logic operations, and instead discuss how the ALU is connected to the AC and the Internal Data Bus.

Figure 3.10 shows how the ALU is connected to the AC and the Internal
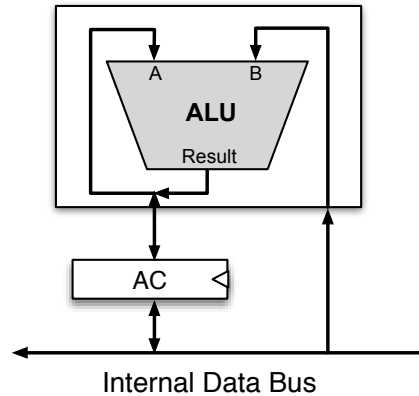
Figure 3.10: ALU connection to AC and Internal Data Bus.

Data Bus. The ALU accepts its left operand (A) from the AC and its right operand (B) from any one of the other registers connected to the Internal Data Bus. However, as you will see in Section 3.4.4, the right operand is typically in the MDR. The ALU generates a result and this is available as an input to the AC, which is then latched at the end of the clock cycle. This design is consistent with 1-address assembly instructions that are accumulator-based.

Memory

A memory consists of an array of n-bit words, and is used to hold instructions and data. A detailed discussion of how memories operate is provided in Section 7.6. This subsection provides a conceptual view of how memories operate.

Figure 3.11 shows the memory and its connections for the pseudo-CPU. A memory has a data bus, an address bus, and a Read/Write control signal. A *data bus* is n bits wide and is used to read a memory word from memory or write a memory word to the memory. The data bus is connected to the MDR register. An *address bus* is m bits wide and is used to specify a particular memory word to be read/written from/to memory. Thus, the maximum number of n-bit memory words a memory can have is limited to 2^m words. The address bus is connected to the MAR register. Finally, the Read/Write control signal is a single bit line, where '0' indicates a read operation and '1' indicates a write operation.
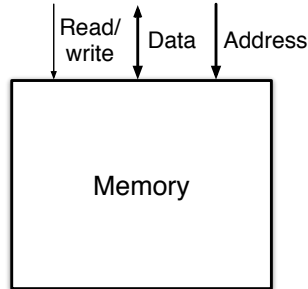
Figure 3.11: Memory subsystem.

In order to read a memory word from the memory, the address of the location in memory to be read needs to be latched onto MAR and the Read/Write control line is set to '0'. Then, the memory word pointed to by MAR is available on the data bus, which is then latched onto MDR. The write operation involves latching the address of the location to be written to MAR, latching the memory word to be written onto MDR, and then setting the Read/Write control signal to '1'.

Control Unit

The Control Unit (CU) is responsible for sequencing micro-operations and providing control signals to the various components to execute assembly instructions. The *sequencing* of micro-operations is implemented using a Finite State Machine (FSM) defined by a set of states and all the possible transitions between states as well as triggering conditions for each transition. In addition, a unique set of *controls signals* are generated for each state to enable the components necessary to perform a micro-operation. For example, control signals for the memory and ALU are generated by the CU. In addition, the control signals that allow the registers to perform register transfer operations are generated by the CU (see Section 3.4.2).

Control signals can be generated either internally or externally. Internal control signals are generated within the CPU chip, while external control signals are generated external to the CPU chip. Examples of internal control signals are ALU control signals and enable signals for registers (see Section 3.4.2). An example of an external control signal is the Read/Write control signal.
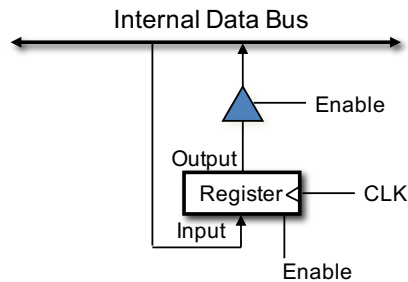
Figure 3.12: Single-input, single-output register connection to the Internal Data Bus.

3.4.2 Bus-Register Connections

Before discussing the basic 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. 3.12 shows a more detailed implementation of a *single bit* of a single-input, single-output register. For example, such a register would be used by the PC without the incrementer. An n-bit version would simply have n of these connected in parallel and controlled 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 system clock and synchronizes all of the operations in the processor. On the other hand, Enable signals are generated by the CU, which depend on the particular instruction being executed. In order to isolate the output of a register from the Internal Data Bus, a *tri-state buffer* is used. Table 3.1 shows the functional requirement for the tri-state buffer. A tri-state buffer acts as an open-circuit, i.e, high-impedance (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.

Table 3.1: Functional Requirement of Tri-State buffer.

Input	Enable	Output
x	0	Hi-Z
0	1	0
1	1	1
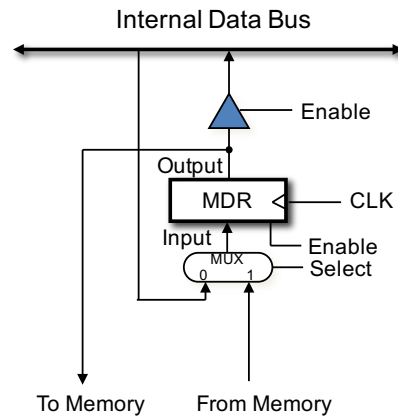
(a) MDR register connection.
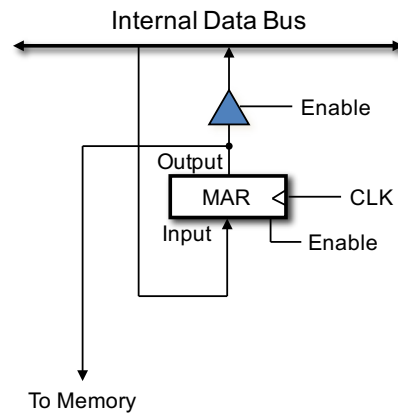
(b) MAR register connection.

Figure 3.13: Multi-input, multi-output register connections.

Fig. 3.13 shows examples of multi-input, multi-output register connections, which are used by AC, PC with the incrementer, MDR, and MAR. Figure 3.13(a) shows how the MDR is connected to the Internal Data Bus. In addition to tri-state buffers controlling the output connection to the Internal Data Bus, there is another tri-state buffer that controls the output connection to the memory. There is also a multiplexer (MUX) that controls whether the data latched onto MDR is from either Memory or the Internal Data Bus. Note that both AC and PC connections to the Internal Data Bus are the same as MDR, except the second set of input/output connections is

Instruction Format

opcode	address
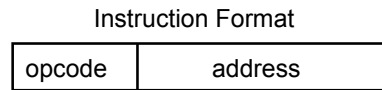
Figure 3.14: Instruction format for the pseudo-CPU.

to the ALU and the incrementer (+1), respectively, instead of memory.

3.4.3 Instruction Format

The instruction format for the pseudo-CPU is shown in Fig. 3.14, which represents a 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, for instructions that require two operands, e.g., $x + y$, one operand is in memory pointed to by EA in the address field, the other operand is in the AC, and the 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[2], and the number of bits in the address field dictates the size of the memory address space. For example, a 16-bit instruction format with a 4-bit opcode field and a 12-bit address field supports up to $2^4 = 16$ different instructions and a memory with $2^{12} = 4096$ memory words. These issues will be elaborated in the latter part of the chapter.

3.4.4 Instruction Cycle

An *instruction cycle* consists of a series of micro-operations required to *fetch* and *execute* one assembly instruction. As discussed in Section 3.3, a micro-operation represents what a microarchitecture can accomplish in one clock cycle. The sequence of micro-operations required to fetch an instruction, referred to as the *fetch cycle*, is typically identical for all instructions. In contrast, each instruction execution, referred to the *execute cycle*, has a unique sequence of micro-operations. This is because the sequence and the number of micro-operations required for an instruction depends on its

---

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

complexity (e.g., add vs. multiply) and hardware availability (e.g., multiplier vs. no multiplier hardware).

Fetch Cycle

The Fetch Cycle is defined by the following sequence of micro-operations using RTL description (see Section 3.3):

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

Figure 3.15 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 CU provide (1) PC_OUT_{enable} signal to the tri-state buffer, which causes the content of 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. When we say that the CU generates these signals, we are implying that that these signals are asserted or enabled, i.e., their values are set to '1'. All other control signals not specified are implied to be disabled, i.e., set to '0'. For reading from and writing to memory, the Read/Write signal is set to '0' for reads and '1' for writes.

In Cycle 2, M[MAR] refers to 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 CU provides the *Read* control signal to memory and MDR_{enable} signal to MDR. At the same time, PC is incremented and relatched (via PC_{enable}), which then points to the 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 do not require the use of the Internal data Bus, and thus do not interfere with each other.

Although the instruction has been fetched into the processor by the end of Cycle 2, the processor does not yet 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 MAR to point to the operand needed by the instruction in the Execute Cycle. This is achieved by allowing the content of MDR to appear on the Internal Data Bus (MDR_OUT_{enable}) and enabling the latching of IR (IR_{enable}) and MAR (MAR_{enable}). At this point, you may wonder how
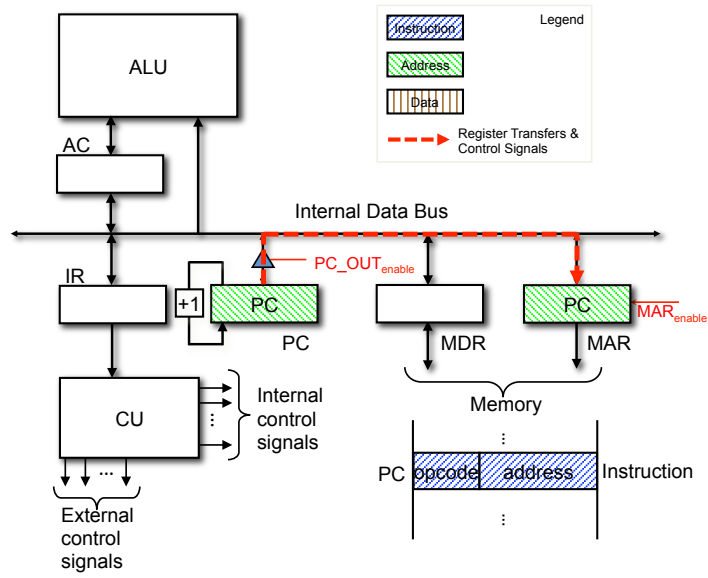
(a) Cycle 1: MAR←PC.
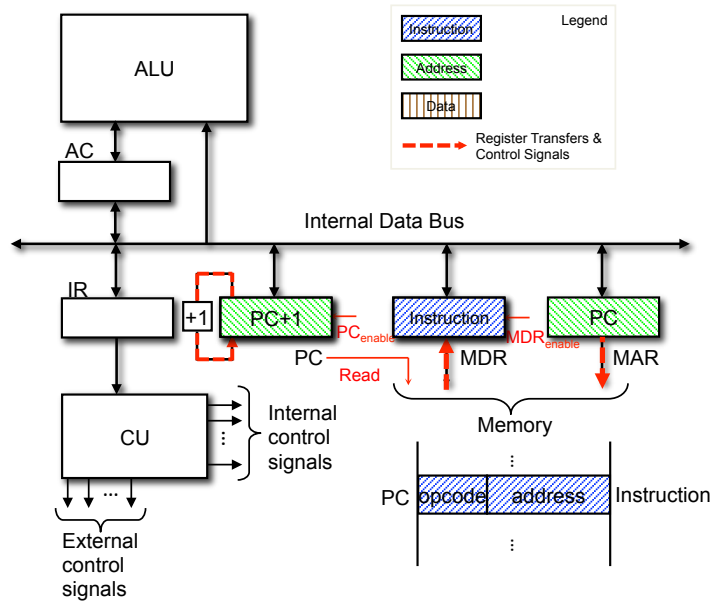
(b) Cycle 2: MDR←M[MAR], PC←PC+1

Figure 3.15: Fetch Cycle.
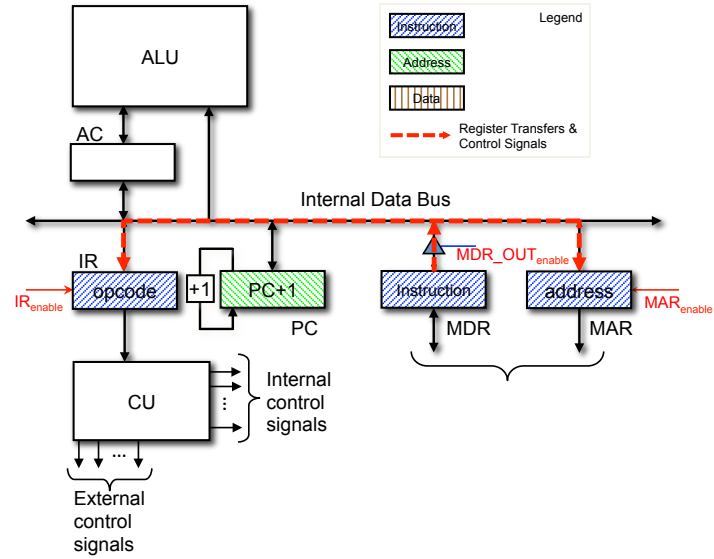
(c) Cycle 3: IR←MDR(opcode), MAR←MDR(address).

Figure 3.15: Fetch Cycle (cont.).
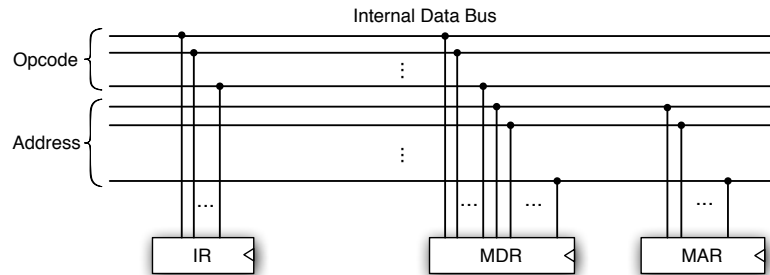
Figure 3.16: Simultaneous latching of IR and MAR.

the different parts of MDR, i.e., opcode and address, are latch onto two different registers at the same time. The answer is in the way IR and MAR are connected to the Internal Data Bus. This is illustrated in Figure 3.16, where the IR and MAR registers are connected to the upper and lower bits of the Internal data bus, respectively.

Note that not all instructions require an operand (e.g., branch instruc-
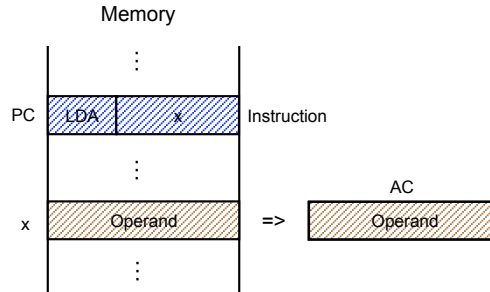
Figure 3.17: Operation of `LDA x`.

tions). However, performing this operation in the Fetch Cycle saves cycles in the Execute Cycle.

Execute Cycle

Unlike the Fetch Cycle, the Execute Cycle depends on the fetched instruction. Thus, the discussion of instruction execution is based on 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
	`BNZ x`	Branch conditionally

The following shows the sequence of micro-operations required for each instruction in Table 3.2.

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

Figure 3.17 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 original content of the AC will be overwritten. The `LDA x` instruction can be implemented by the following sequence of micro-operations:

Execute Cycle:

Cycle 1: MDR ← M[MAR]
Cycle 2: AC ← MDR

Figure 3.18 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, MDR_{enable} is asserted and the *Read/Write* signal is set to '0'. This latches the operand in the memory location pointed to by MAR, i.e., M[MAR], to MDR at the end of the cycle. In Cycle 2, both MDR_OUT_{enable} and AC_{enable} are asserted, which allows the operand in MDR to appear at the input of AC via the Internal Data Bus. Thus, the operand is latched onto AC at the end of the clock cycle.

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

Figure 3.19 illustrates the operation of `STA x`. This instruction is basically the inverse operation of `LDA x`, where the operand in the AC is stored to the memory location pointed to by `x`. The `STA x` instruction can be implemented by the following sequence of micro-operations:

Execute Cycle:

Cycle 1: MDR ← AC
Cycle 2: M[MAR] ← MDR

Fig. 3.20 illustrates the execute cycle for `STA x`. In Cycle 1, the content of AC is transferred to MDR. This is achieved by asserting the control signals AC_OUT_{enable} and MDR_{enable}. In Cycle 2, the content of MDR is written to the memory location pointed to by MAR, which is the effective address `x`. In Cycle 2, the operand in MDR is written to memory by setting the *Read/Write* signal to '1'.

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

Figure 3.21 illustrates the operation of `ADD x`. The effective address `x` points to one of the operands (i.e., Operand2), while the other operand is in AC. Then, 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:
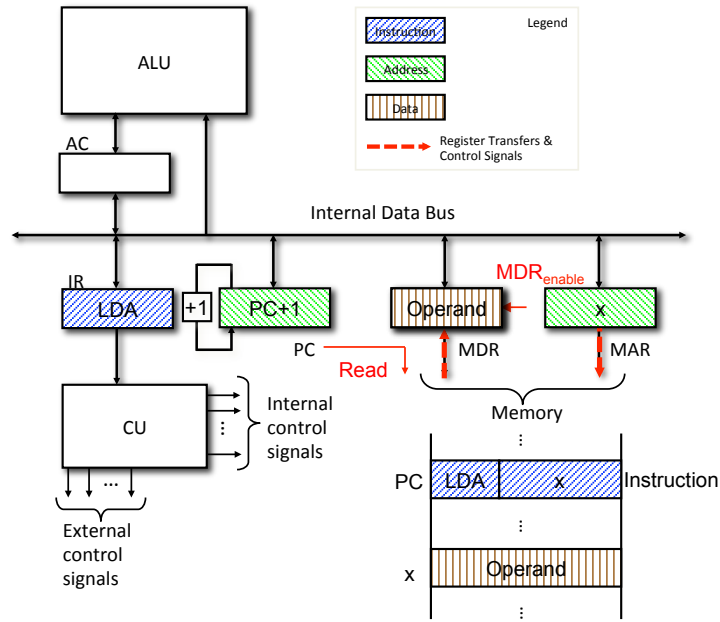
(a) Cycle 1: MDR←M[MAR].
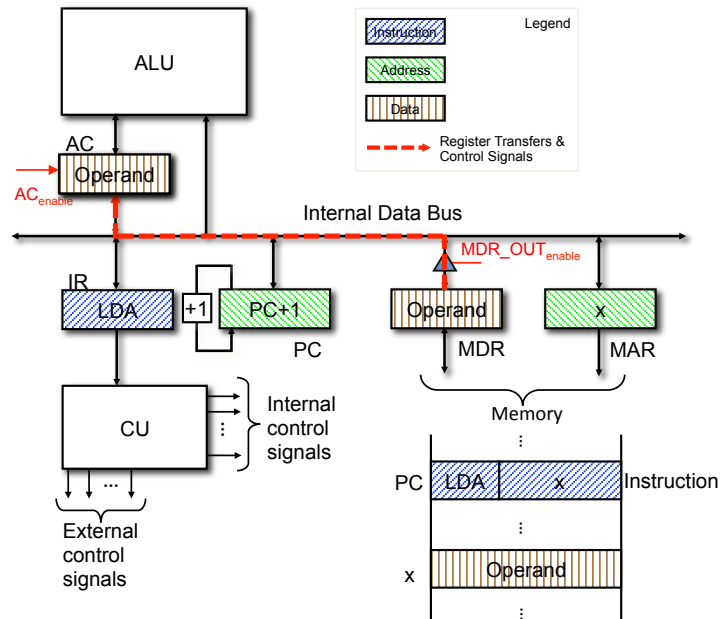
(b) Cycle 2: AC←MDR.

Figure 3.18: Microoperations for LDA x.
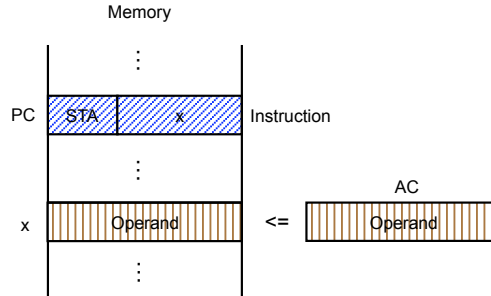
Figure 3.19: Operation of `STA x`.

Execute Cycle:

Cycle 1: MDR ← M[MAR]
Cycle 2: AC ← AC + MDR

Fig. 3.22 illustrates the execute cycle for `ADD x`. In Cycle 1, the operand in the 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., Operand1), and the result is latched to the AC.

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

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

Execute Cycle:

Cycle 1: MDR ← M[MAR]
Cycle 2: AC ← AC - MDR

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

`NAND x` is also similar to `ADD x` and `SUB x`, except that the operation performed is bit-wise logical NAND. The `NAND x` instruction can be implemented by the following sequence of micro-operations:
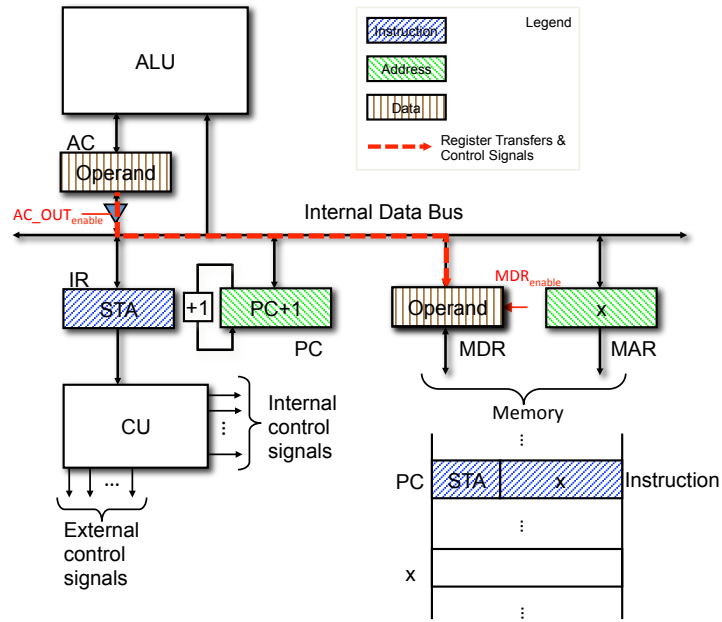
(a) Cycle 1: MDR ← AC.
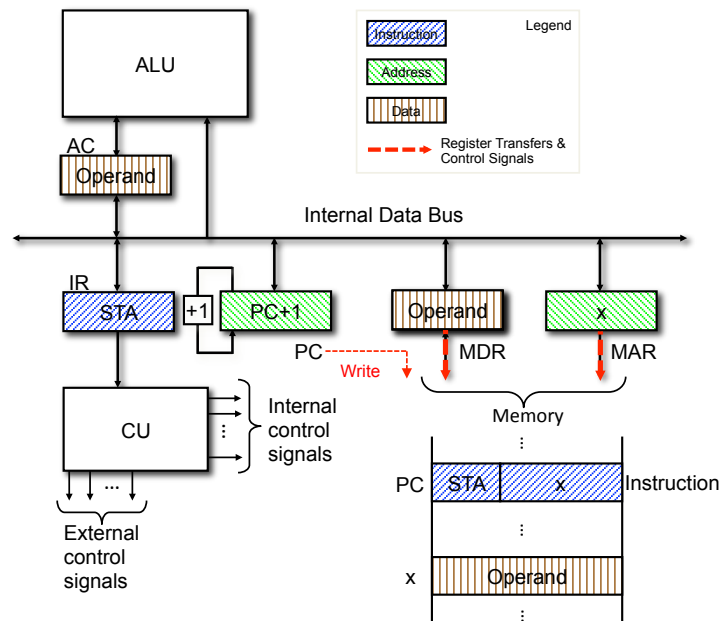
(b) Cycle 2: M[MAR] ← MDR.

Figure 3.20: Microoperations for STA x.
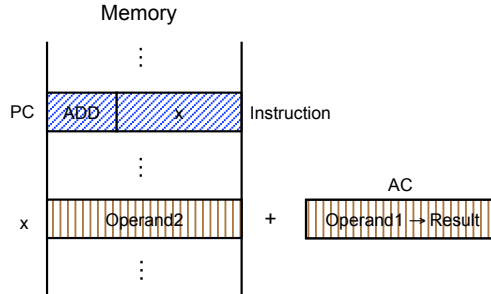
Figure 3.21: Operation of `ADD x`.

Execute Cycle:

Cycle 1: MDR ← M[MAR]
Cycle 2: AC ← $\overline{AC \wedge MDR}$

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:

Execute Cycle:

Cycle 1: AC ← *sl* AC

Note that `SHFT` can also be described as

Cycle 1: AC(n...1) ← AC(n-1...0), AC(0) ← 0

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

Figure 3.23 illustrates the operation of `J x`. The address `x` represents 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:

Execute Cycle:

Cycle 1: PC ← MDR(address)

or
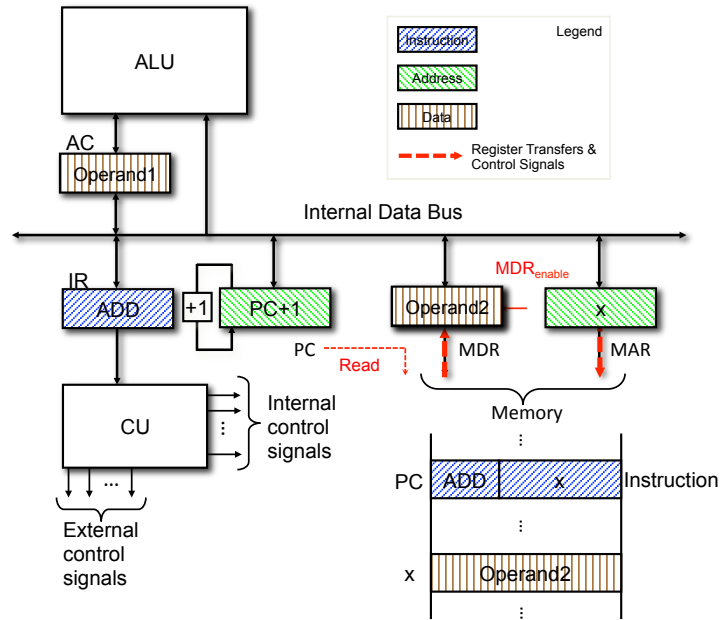
(a) Cycle 1: MDR ← M[MAR].
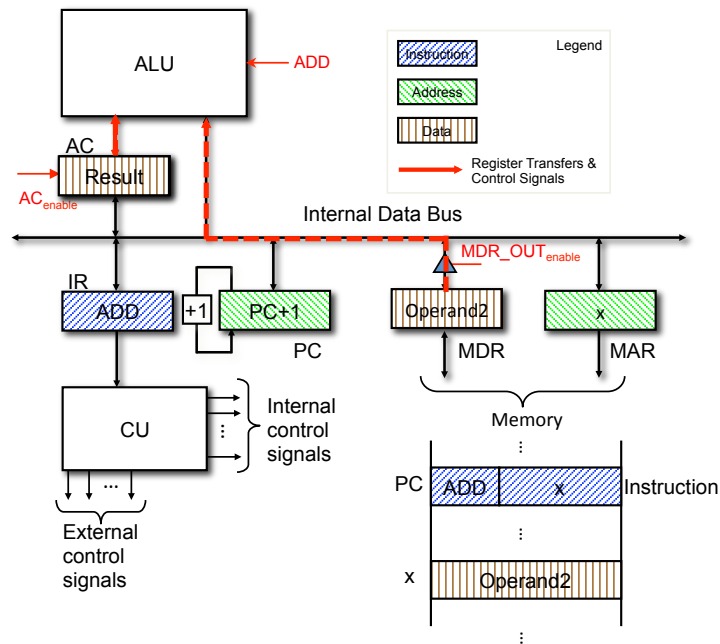
(b) Cycle 2: AC ← AC + MDR.

Figure 3.22: Microoperations for ADD x.
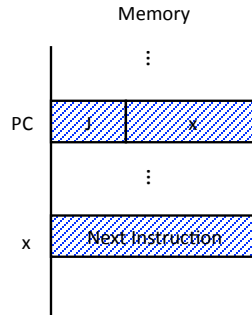
Figure 3.23: Operation of `J x`.

Cycle 1: PC ← MAR

Fig. 3.24 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. An alternative is to transfer `x` from MAR. Either way, the new instruction cycle starts from the *next instruction*.

Example 3.8. *Execute cycle for BNZ x.*

In contrast to `J`, which is an unconditional branch, `BNZ x` is a *conditional branch*. The 'NZ' part of `BNZ` represents whether the previous arithmetic instruction (i.e., the instruction just before `BNZ`) generated a result 'Not Equal to Zero'. The `BNZ x` instruction can be implemented by the following microoperation:

Execute Cycle:

Cycle 1: If (Z != 1) then PC ← MDR(address)

or

Cycle 1: If (Z != 1) then PC ← MAR

Fig. 3.25 illustrates the execute cycle for `BNZ x`, which also shows some additional logic required to test the 'NE' condition. Similar to the unconditional branch instruction `J x`, `x` serves as the target address for the branch. `BNZ` tests the Z-flag (or Zero-flag), which is one of a number of *condition codes* generated by the ALU after every arithmetic operation. The other condition codes include negative (N), overflow (V), and carry (C) flags (see Chapter 9). If the Z-flag is not set (i.e., the result is not zero) and the current
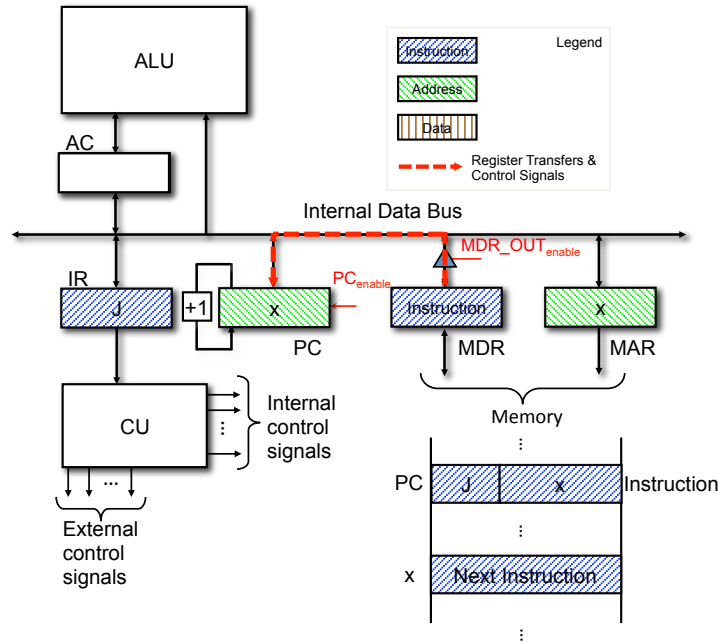
Figure 3.24: Microoperation for `J x`.

instruction is a branch (i.e., `BNZ`), which is indicated by the *Branch* signal from the CU, the address portion of the instruction `x` in MDR is transferred to PC. Alternatively, the target address `x` can also be transferred from MAR.

3.4.5 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. This subsection will discuss how the pseudo-ISA can be extended to support additional instructions as well as different addressing modes.

In order to understand the concept of *ISA extension*, consider the following example where a new instruction is added to the pseudo-ISA to ease programming and improve performance.

Example 3.9. *Suppose we want to implement the instruction "load accumulator indirect",* `LDA (x)`*.*
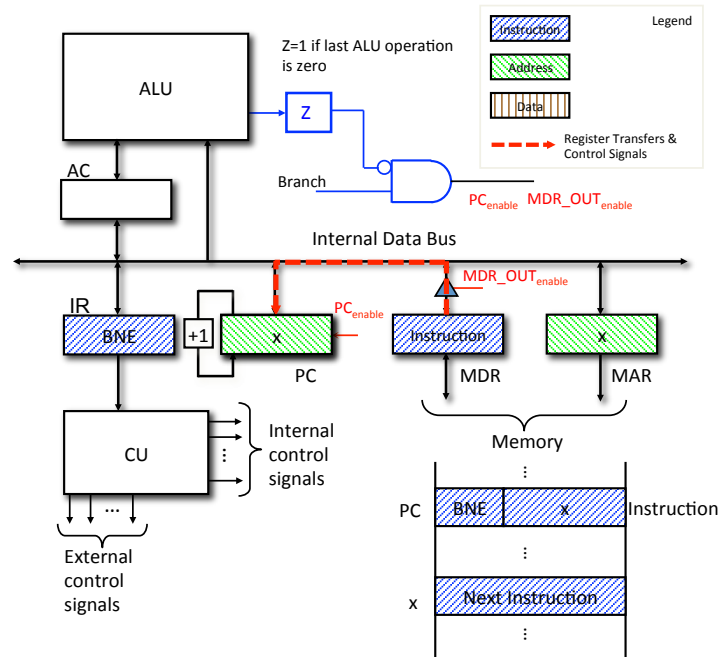
Figure 3.25: Microoperation for `BNZ x`.

`LDA (x)` is an example of an instruction that uses *indirect addressing mode*. Note that parentheses '(' and ')' distinguish indirect addressing from direct addressing. As first discussed in Section 2.3.3, addressing 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, which is also referred to as *indirection*, and thus allows for implementation of *pointers*.

Pointers in high-level languages 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;
}
```
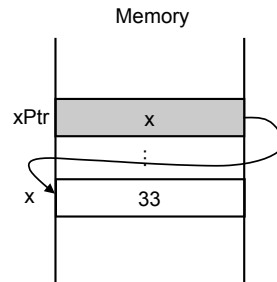
Figure 3.26: The concept of a pointer.

The variable `x` is of type integer, and `*xPtr` represents a pointer to type `int`. The first statement assigns '33' to `x`. In the second statement, the address of variable `x` is assigned to the pointer variable `xPtr`, where `&` represents the address operator. Figure 3.26 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 are stored in memory, not embedded within instructions, and thus 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 any element of array `A` by indexing off of the first element of `A`, i.e., `A[0]`. This is also the case for accessing members of a structure. For example, consider an array of elements stored in memory pointed to by `A`. This is illustrated in Figure 3.27. Since the pointer to the first element in array `A`, i.e., `A[0]`, is in the memory location pointed to by `APtr`, we can access any element of the array by adding an index to `A`. Moreover, elements of the array can be accessed one by one by simply incrementing `A` in a loop. Can you imagine what would happen if we only had `LDA x`, or more precisely `LDA A`? Since `A` is in encoded within the instruction, it cannot be manipulated within a loop. Thus, we would basically have to have separate instructions to access the different elements of the array as shown below:

```
LDA    A
LDA    A+1
...
LDA    A+n-1
```
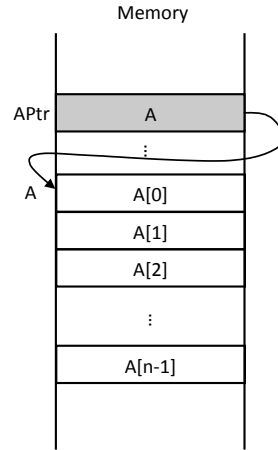
Figure 3.27: An array of elements in memory.
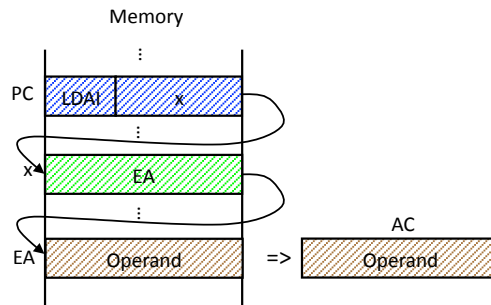
Figure 3.28: The concept of indirection.

Now that we have discussed pointers and indirection, let us resume the discussion of `LDA (x)`. Figure 3.28 illustrates the operation of `LDA (x)`. Note that `LDAI` in the opcode field indicates indirection and distinguishes it from its direct addressing counterpart. As can be seen from the figure, the *Operand* is pointed to by the *EA* in 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.

Execute Cycle:

Cycle 1: MDR ← M[MAR]   ; Read effective address (EA)
Cycle 2: MAR ← MDR   ; Move EA to MAR
Cycle 3: MDR ← M[MAR]   ; Read operand
Cycle 4: AC ← MDR   ; Move operand to AC

Note that the registers in the micro-operations are marked (in red) to indicate their contents. This makes it easier to keep track of what each micro-operation accomplishes.

Figure 3.29 illustrates the sequence of micro-operations for LDA (x). Cycle 1 involves accessing the memory location x (i.e., M[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* from memory in Cycle 3. Finally, *Operand* in MDR is transferred to AC in Cycle 4.

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

The operation of LDA -(x) is illustrated in Figure 3.30. This instruction is similar to LDA (x), except the content of the memory location pointed to by x, i.e., EA+1, is decremented and stored back *before* (referred to as *pre-decrement*) *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.

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. 3.9. 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 3.31 shows such a modification. Based on this design, the sequence of micro-operations for the Execute Cycle is shown below.

Execute Cycle:

Cycle 1: MDR ← M[MAR]   ; Read EA+1
Cycle 2: MDR ← MDR - 1   ; Decrement EA+1
Cycle 3: M[MAR] ← MDR   ; Store it back in memory location x (i.e., M[x])
Cycle 4: MAR ← MDR   ; Move EA to MAR
Cycle 5: MDR ← M[MAR]   ; Read operand
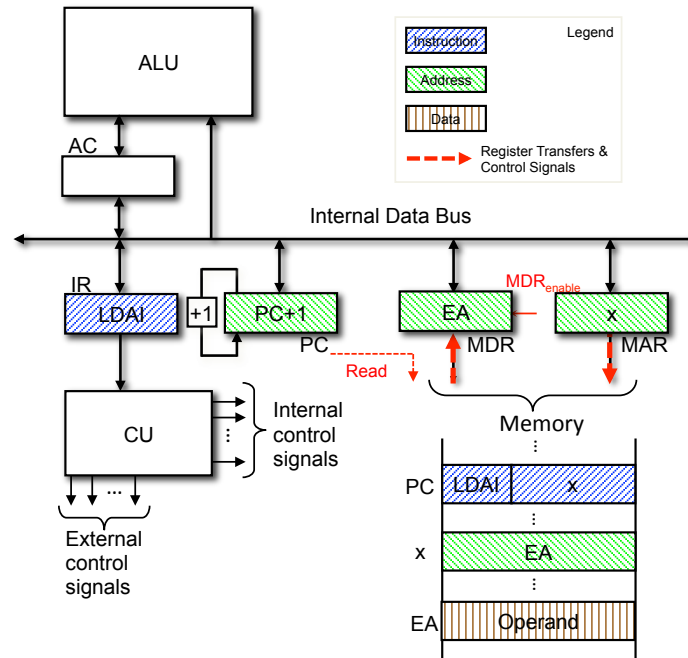
(a) Cycle 1: MDR ← M[MAR].
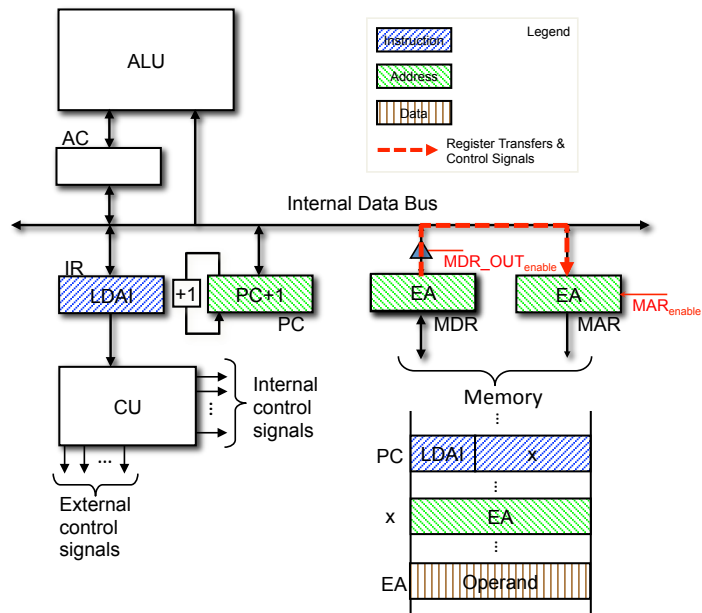
(b) Cycle 2: MAR ← MDR.

Figure 3.29: LDA indirect.
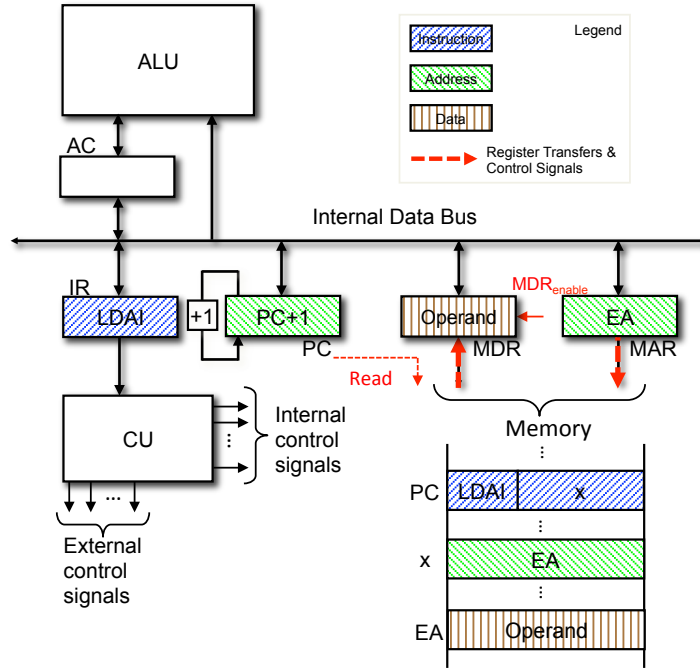
(c) Cycle 3: MDR ← M[MAR].
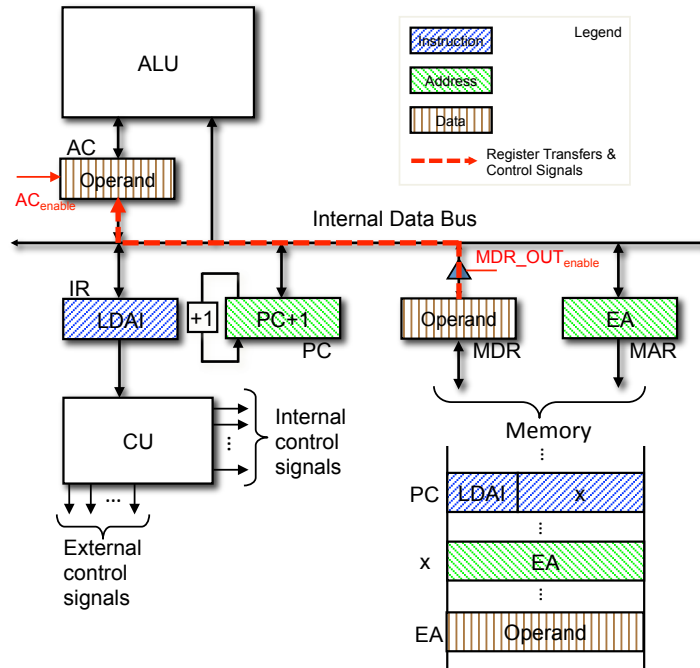
(d) Cycle 4: AC ← AC + MDR.

Figure 3.29: LDA indirect (cont.).
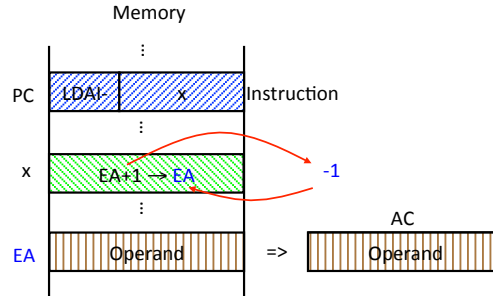
Figure 3.30: LDA indirect with pre-decrement.
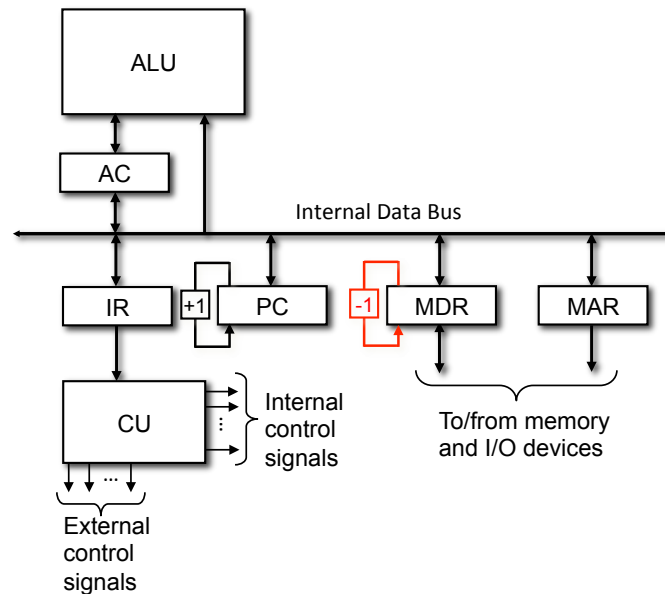
Figure 3.31: Pseudo-CPU with decrement capability in MDR.

Cycle 6:   AC   ← MDR   ; Move operand to AC

As can be seen, Cycles 1 and 4-5 are identical to the LDA (x) instruction. The only additional micro-operations required are decrementing $EA+1$ in Cycle 2 and storing it back into the memory location pointed to by x in Cycle 3. Note that Cycles 3 and 4 can be done in the same cycle. This is because moving the content of MDR into MAR occurs at the end of the

clock cycle and does not affect the content of MAR during writing of MDR into memory in the same cycle. Based on this, the optimized sequence of micro-operations for `LDA -(x)` is shown below:

Execute Cycle (optimized):

```
                EA+1           x
Cycle 1: MDR ← M[MAR]          ; Read EA+1
                EA    EA+1
Cycle 2: MDR ← MDR - 1         ; Decrement EA+1
              x         EA   EA     EA
Cycle 3: M[MAR] ← MDR, MAR ← MDR    ; Store it back in memory location x
         (i.e., M[x]) and move EA to MAR)
              Operand       EA
Cycle 4:  MDR ← M[MAR]         ; Read operand
              Operand   Operand
Cycle 5:  AC  ← MDR           ; Move operand to AC
```

One of the problems with using a dedicated decrementer as shown in Figure 3.31 is that it requires another n-bit adder hardware. 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 the indirect with pre-decrement addressing mode is used. Therefore, in the second method, we will use AC and the ALU to perform pre-decrement since ALU already has this capability.

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

Execute Cycle (optimized, no dedicated decrementer):

```
                EA+1           x
Cycle 1: MDR ← M[MAR]          ; Read EA+1
              EA+1      EA+1
Cycle 2:  AC  ← MDR           ; Move EA+1 to AC
            EA     EA+1
Cycle 3: AC ←  AC  - 1         ; Decrement EA+1
              EA    EA
Cycle 4: MDR ← AC            ; Move EA to MDR
              x         EA   EA     EA
Cycle 5: M[MAR] ← MDR, MAR ← MDR    ; Store it back in memory location x
         (i.e., M[x]) and move EA to MAR
              Operand       EA
Cycle 6:  MDR ← M[MAR]         ; Read operand
              Operand   Operand
Cycle 7:  AC  ← MDR           ; Move operand to AC
```

The $EA+1$ is decremented during Cycles 2-3, and the EA is stored back into the memory location pointed to by x during Cycles 4-5. As can be seen, this implementation requires 7 versus 5 cycles for the design with the dedicated decrementer but less amount of hardware.
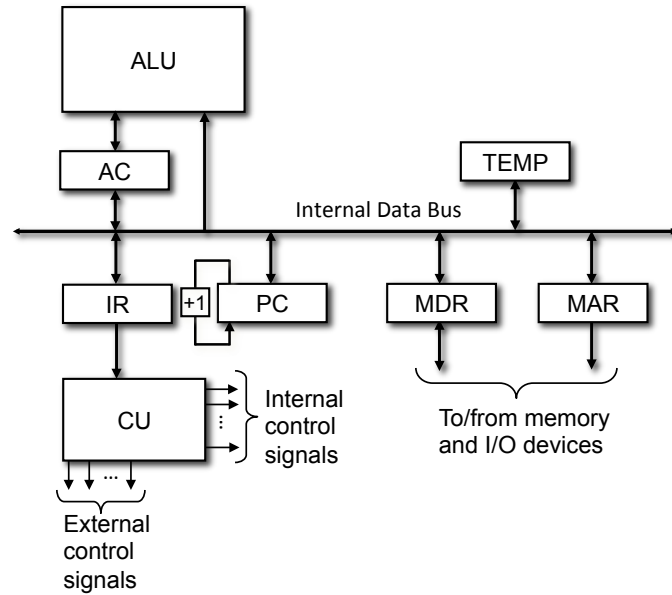
Figure 3.32: Pseudo-CPU with a temporary register.

It is important to note that the original content of AC was destroyed during Cycle 2, which is not a problem since the AC will be loaded with the operand. However, in some cases, the original content of AC needs to be preserved because it contains a valid data for a subsequent instruction, e.g., STA. This can be done by having an extra register to temporarily store the original content of the AC. This is shown in Figure 3.32. The addition of a *TEMP* register increases the flexibility for implementing more complicated instructions, e.g., STA -(x), LDA (x)+, and STA (x)- (see Section **??**).

Chapter 4

Atmel's AVR 8-bit Microcontroller: Part 1 - Assembly Programming

Contents

4.1 Introduction

This chapter presents assembly programming for one of the most widely used embedded processors, Atmel *AVR 8-bit microcontrollers*. As discussed
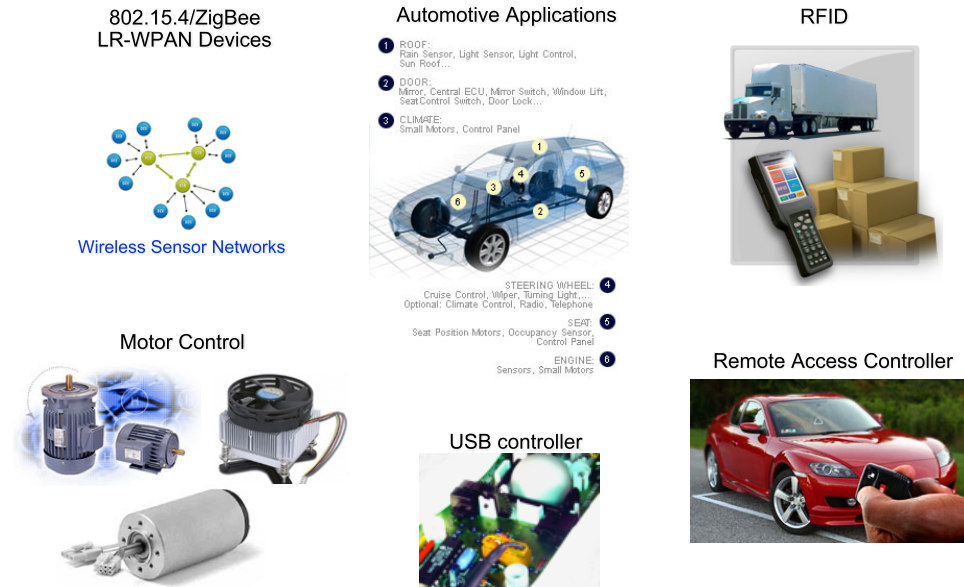
Figure 4.1: Some AVR-based products.

in Chapter 1.2, a microcontroller is a small computer system optimized for hardware control that encapsulates the processor core, memory, clock, timers, and Input/Output (I/O) ports on a single piece of silicon. AVR provides a family of microcontrollers with different Program and Data Memory sizes, number of I/O ports, clock speed, and data sizes for a variety of embedded solutions. The majority of microcontrollers in use today are embedded in other machinery, such as mobile devices, automobiles, telephones, appliances, and peripherals for computer systems. Such systems are called *embedded systems*. Figure 4.1 show some examples of AVR-based products.

Although there are numerous microcontrollers and microprocessors, several reasons exist for choosing, in particular, the AVR architecture to introduce assembly language programming in this book. First, unlike microprocessors in personal computers (PCs), e.g., Intel® Core™ processors, microcontrollers are designed for small, dedicated applications with low-power requirements. Microprocessors for PCs tend to receive most of the spotlight; however, microcontrollers are more prevalent and ubiquitous in our lives. While some embedded systems are very sophisticated, e.g., smartphones, many have small memory without operating systems 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. Moreover, embedded systems usually have no keyboard, screen, disk, printer, or other recognizable I/O devices of a PC, and may lack human interaction devices of any kind. Microcontrollers are an efficient and economical way to digitally control hardware devices by reducing size, power, and cost compared to PCs that use a design consisting of separate high-speed microprocessor, memory, and I/O devices.

Second, one of the most important aspects of assembly programming is I/O and handling interrupts (see Chapter 5). AVR microcontrollers are optimized for hardware control in the sense that their Instruction Set Architecture (ISA) provides a set of instructions and peripheral features to easily control I/O devices.

Finally, learning assembly language programming is much like learning to ride a bicycle – once you learn how to program in assembly for one processor, it is not hard to pick up assembly programming for other processors. The AVR architecture has a relatively generic ISA, so once you learn how to program in AVR assembly it is easy to transition to other processors.

4.2   General Characteristics

The AVR family of microcontrollers basically has the same ISA, but the processors differ in memory size, number of I/O ports, peripheral features, and clock speed. Our discussion of assembly language programming in this chapter will be based on a specific AVR processor, the *ATmega128*. Figure 4.2 shows the block diagram of the ATmega128 microcontroller. 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 (USART), etc. The discussion of I/O and peripherals will be presented in Chapter 5. For now, let us concentrate on the AVR architecture and assembly programming.

There are several important concepts in assembly programming that high-level language programmers are not used to. First, you have to know where the operands or data to be operated on are located in memories and registers. This is because all instructions involve either registers or a register and a memory location. Therefore, knowing what registers are available and how memory locations are referenced are crucial for learning how to program in assembly. Second, each assembly instruction can perform a simple operation limited by the architecture, whether it be moving data between memory and registers, an arithmetic operation, or a control transfer. Therefore, multiple assembly instructions will be required to accomplish the same
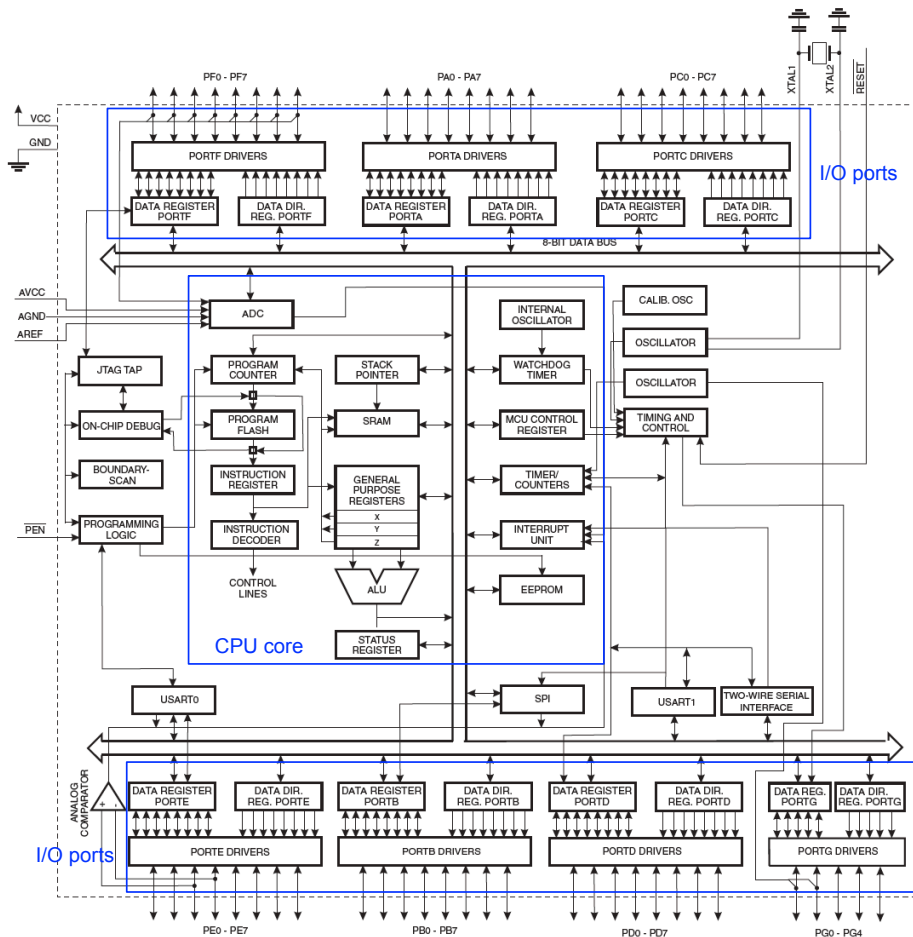
Figure 4.2: The block diagram of Atmega128.

functionality of a single expression in a high-level language. This will require not only writing a correct sequence of assembly instructions to implement the desired functionality but also knowing where to store intermediate results in registers or even in memory. Third, flow of control in an assembly program is determined by the state of the processor in the form of status or condition flags dictated by the last instruction execution *and* conditional branch instructions that may or may not alter the control flow. Therefore, managing the control flow in an assembly program requires you to be aware of how instructions change certain status flags and how these flags are tested.

Program Memory

0x0000	
	Program Flash (64K x 16)
0xFFFF	

16 bits

Data Memory

0x0000	32 8-bit GPRs
0x001F	
0x0020	64 8-bit I/O regs.
0x005F	
0x0060	160 Ext. 8-bit I/O regs.
0x00FF	
0x0100	Internal SRAM (4096x8)
0x10FF	

8 bits
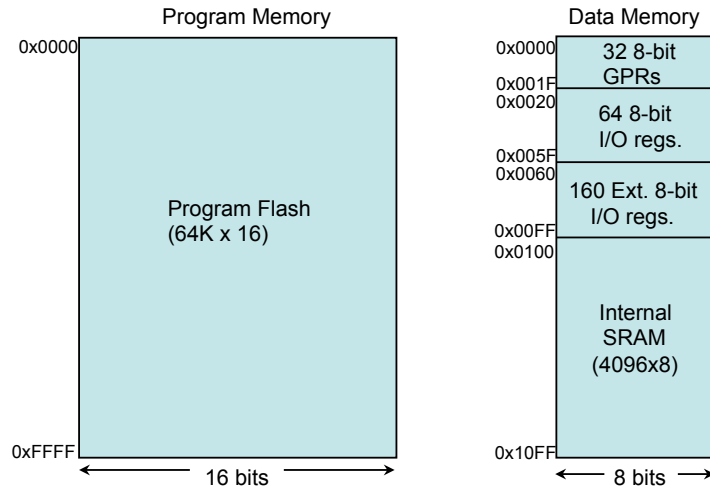
Figure 4.3: AVR memory organization.

At this point, you may be totally confused and overwhelmed by the prospect of learning assembly programming. Well, just keep in mind that assembly programming requires you to do a bit more work and forces you to be aware of the intricate details of program execution. In the process of learning assembly programming, you will be exposed to the requirements of the processor architecture, which is another major focus of this book. So, let us get started on AVR assembly programming!

4.2.1 Program and Data Memories

The AVR architecture has separate Program and Data Memories, which are indicated as Program Flash and SRAM, respectively, in Figure 4.2. A more detailed diagram of the two memories is shown in Figure 4.3.

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, and thus the reason for '128' after the name 'ATmega'. Note that most of AVR instructions are 16 bits; however, there are a few instructions that are 32 bits and thus require two 16-bit memory words. The Program Memory is implemented using non-volatile flash memory, which retains its contents even after the power is turned off. This is crucial since, unlike PCs, most embedded systems do not have hard disks and instead the code is programmed

directly onto the Program Memory once and it is expected to be retained even after the system is turned off.

The size of *Data Memory* is 4,352 bytes, which consists of 256 bytes of *General Purpose Registers* (GPRs) (see Section 4.2.2) and I/O registers plus 4 Kbytes of internal *Static RAM* (SRAM). The Data Memory is also expandable up to 64 Kbytes using external memory.

4.2.2   Registers

The AVR architecture has the following set of registers:
- 32 8-bit GPRs
- 16-bit X-, Y-, and Z-register
- 16-bit Program Counter (PC)
- 16-bit Stack Pointer (SP)
- 8-bit Status Register (SREG)
- 6 8-bit and one 5-bit I/O registers (ports)

Chapter 5 will provide a detailed discussion of the I/O ports in the AVR architecture. The discussion that follows covers the purpose of GPRs, PC, SP, and SREG.

GPRs

Figure 4.4 shows the 32 *GPRs*, which are located in the first 32 locations of the Data Memory and serve as a small storage space used by the processor to quickly access and perform operations on both data and addresses. The GPRs are referenced in assembly programs as `R0 - R31` (or `r0 - r31`).

As can be seen in Figure 4.2, GPRs are important because all the data manipulated by the *Arithmetic and Logic Unit* (ALU) and data transfer operations between memories and I/O ports are done through these registers. For example, arithmetic instructions, such as `ADD` (*Add two registers*) and `SUB` (*Subtract two registers*), require two source registers and one destination register to be GPRs. As another example, data from an I/O port cannot be stored directly into the Data Memory. Instead, the I/O instruction `IN` (*In port*) has to first read the I/O data into a GPR and then stored into the Data Memory using the `ST` (*Store indirect*) instruction. This is also the case in the reverse direction, i.e., from Data Memory to an I/O port.
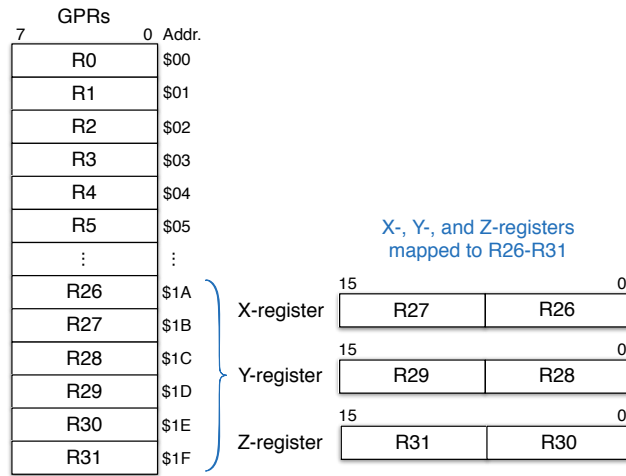
Figure 4.4: AVR GPRs.

X-, Y-, and Z-registers

X-, *Y-*, and *Z-registers* are *address registers* used as pointers to the Data Memory address space. One of these registers, Z-register, can also be used to access the Program Memory address space. These address registers are also mapped to the last six GPRs (see Figure 4.4). 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 access to up to 64 Kbytes of the Data Memory space.

The use of address registers is an unfamiliar concept for people new to assembly programming. This is because in high-level languages, such as C/C++, a data to be operated on is declared as a variable using an identifier of our choosing. The variable is then *virtually* referenced using the identifier without worrying about where it is physically stored in memory because the compiler hides all the details. Here lies the major difference between high-level language programming and assembly language programming. In assembly programming, the programmer has to know where variables and data structures are stored in memory and thus how to access them. This is achieved by storing addresses of variables and data structures in X-, Y-, and Z-registers and using them as pointers.

Program Counter

Program Counter (PC) is a special register that points to either the current instruction being executed or the next instruction to be executed depending on whether or not it has been incremented. PC is 16-bit wide so that all 64K ($=2^{16}$) memory words in the Program Memory can be accessed.

PC indicates where the processor is in the instruction execution sequence and imposes a strict sequential ordering on the fetch and execution of instructions from memory. During the execution of most instructions, PC is incremented by one, i.e., PC+1, to point to the next instruction in the control flow. However, the control flow can change due to execution of the following three types of instructions: Jumps or unconditional branches, conditional branches, and subroutine calls and returns. Jumps and subroutine calls and returns unconditionally change the PC with target addresses. On the other hand, conditional branches update the PC only when a specified condition is met.

Stack Pointer

Stack Pointer (SP) is used to point to the top of the stack. A *stack* is a data structure that implements a *last-in, first-out* (LIFO) behavior. A stack is used, for instance, to store information about the active subroutines of a program, i.e., return addresses of subroutine calls and input and output parameters.

Status Register

Status Register (SREG) contains a collection of *condition codes*, or flags, to indicate the current status of the processor. The contents of SREG are shown in Fig 4.5, where R/W indicates that the bit can be both read and written and the number in parenthesis indicates the initial value when the processor is powered on.

I-bit is used to turn on the interrupt facility (see Section 5.3 for a detailed discussion on interrupts). *T*-bit can be used as either a source or a destination for a single bit of a register to be operated on, and is useful for bit manipulation. For example, a bit from a register can be copied to T-bit using the BST (*Bit store from register to T*) instruction, and T-bit can be copied to a bit in a register by using the BLD (*Bit load from T to Register*) instruction. *H*-bit indicates a carry for Binary Coded Decimal (BCD) arithmetic. *S*-bit is an Exclusive-OR between N-bit and V-bit, and is used for two's-complement arithmetic. *N*-bit indicates a negative result

Status Register (SREG)

7	6	5	4	3	2	1	0
I	T	H	S	V	N	Z	C
R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)

Bit 7 - Global Interrupt Enable
Bit 6 - Bit- Copy Storage
Bit 5 - Half Carry Flag
Bit 4 - Sign Bit
Bit 3 - Two's Complement Overflow Flag
Bit 2 - Negative Flag
Bit 1 - Zero Flag
Bit 0 - Carry Flag
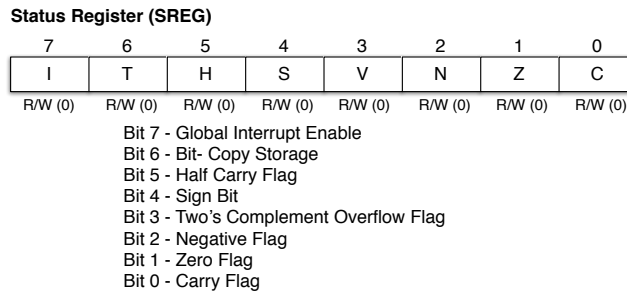
Figure 4.5: Status Register.

from an arithmetic or logic operation. V-bit indicates an overflow from an arithmetic operation. Z-bit indicates a zero result from either an arithmetic or a logic operation. Finally, C-bit indicates a carry from an arithmetic operation.

These flags are set/reset according to the outcome of an instruction execution, which can then be followed by a conditional branch instruction. For example, suppose an instruction, e.g., CP (*Compare*), causes Z-bit to be set indicating the result was zero. A subsequent execution of a conditional branch instruction that tests if Z-bit is set, i.e., BREQ (*Branch if equal*), will cause the control flow to be changed to the target address of the branch. Section 4.4.3 will provide a more detailed discussion on how these condition codes are used by Control Transfer instructions.

4.3 Addressing Modes

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

There are several addressing modes provided in the AVR architecture:

- Register (with one and two registers)
- Direct
- Indirect
    - with Displacement
    - with Pre-decrement
    - with Post-increment

The following subsections discuss these addressing modes.
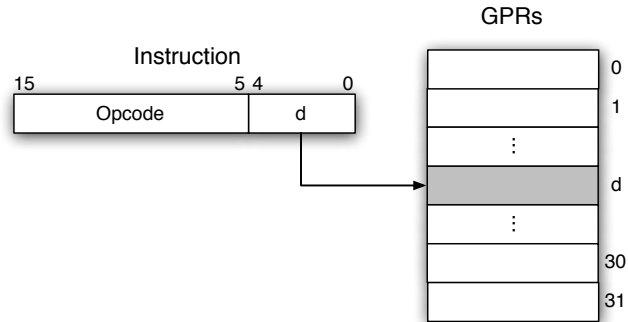
(a) Register addressing with one register.
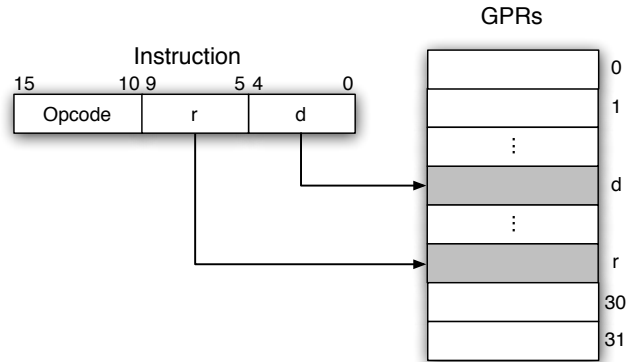
(b) Register addressing with two registers.

Figure 4.6: Register addressing mode.

4.3.1 Register Addressing Mode

Register Addressing mode is used to access data in GPRs. Fig. 4.6 shows two versions of the Register Addressing mode, where an instruction can specify either a single register using `Rd` or two registers using `Rd` and `Rr`, where `R` means 'register' and `d` and `r` mean 'number'. When both `Rd` and `Rr` are used, `Rd` stands for *d*estination register and `Rr` stands for sou*r*ce register.

The following shows a couple of AVR assembly instructions that use the register addressing mode with a single register:

```
INC Rd
CLR Rd
```

The INC (*Increment*) instruction increments the content of register Rd by one. The CLR (*Clear register*) instruction clears the content of register Rd. Figure 4.6(a) illustrates how a single operand is referenced by the 5-bit register identifier field d in the instruction format, which allows access to all 32 GPRs.

The following shows an instruction that uses the register addressing mode with two registers:

```
ADD Rd, Rr
```

The ADD instruction adds the contents of Rd and Rr, and stores the result in Rd, which is consistent with assignment statements in high-level languages. This example also illustrates the *2-address instruction format* used by the AVR ISA, where both left (d) and right (r) registers serve as input operands and the result of an operation is stored in the left (d) register. Figure 4.6(b) illustrates how the two operands are referenced by r and d fields. Again, both fields are 5 bits allowing all 32 GPRs to be used.

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

4.3.2 Direct Addressing Mode

Figure 4.7 shows examples of the *Direct Addressing* mode. Figure 4.7(a) illustrates an instruction that accesses an 8-bit word in Data Memory. Note that this is a 32-bit instruction format, which is stored in two consecutive 16-bit words in Program Memory and the second 16-bit word represents the *effective address*, or the address of the memory location that contains the operand. This allows direct access to all 2^{16}=64K words in Data Memory. The following shows example instructions that use the direct addressing mode:

```
STS $1000, Rr
LDS Rd, $1000
```

The STS (*Store Direct SRAM*) instruction stores the content of the source register Rr to the memory location $1000. LDS (*Load direct from SRAM*), which is the inverse of STS, loads the content of memory location pointed to by $1000 to the destination register Rd.

I/O operations shown in Figure 4.7(b) also use direct addressing. There are 64 I/O registers mapped to the Data Memory space, and thus 6-bit
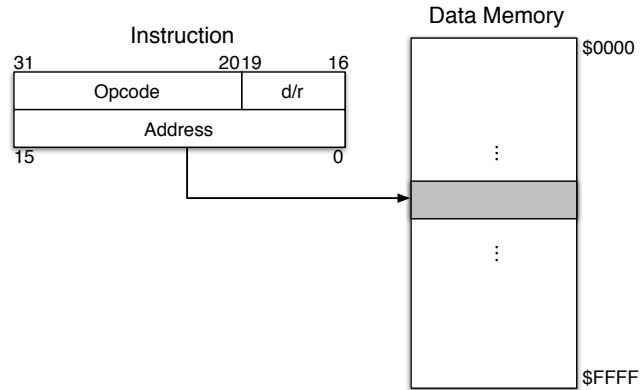
(a) Direct addressing.
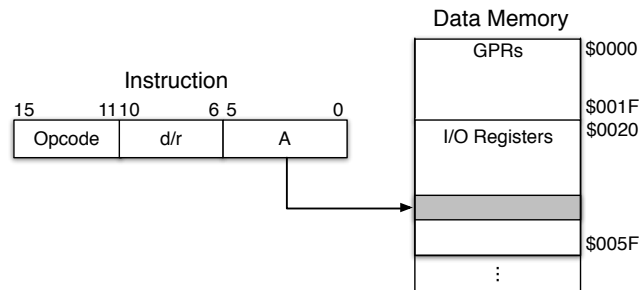
(b) I/O direct addressing.

Figure 4.7: Direct addressing modes.

address (A) field is included in the instruction format. The two instructions that use the *I/O Direct Addressing* mode are shown below:

```
IN Rd, $13
OUT $12, Rr
```

The IN (*In port*) instruction moves the content of the port register $13, which is PINC, to the destination register Rd. On the other hand, OUT (*Out port*) moves the content of the source register Rr to the port register $12, which happens to be PORTD. The 64 I/O register address space can also be directly accessed using LDS and STS. This is done by adding an offset of 32 to the 64 I/O register address. For example, IN Rd, $13 and OUT $12, Rr are equivalent to LDS Rd, $0033 and STS $0032, Rr, respectively. For
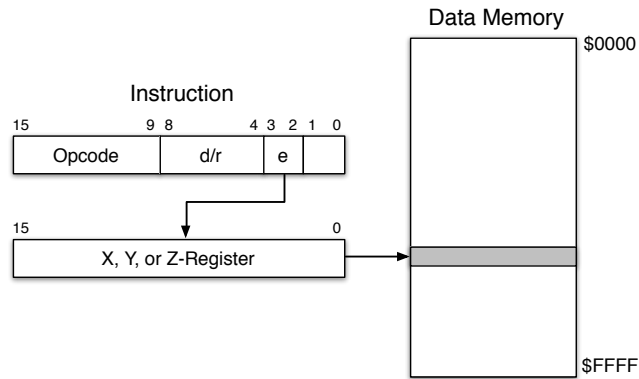
(a) Indirect addressing.
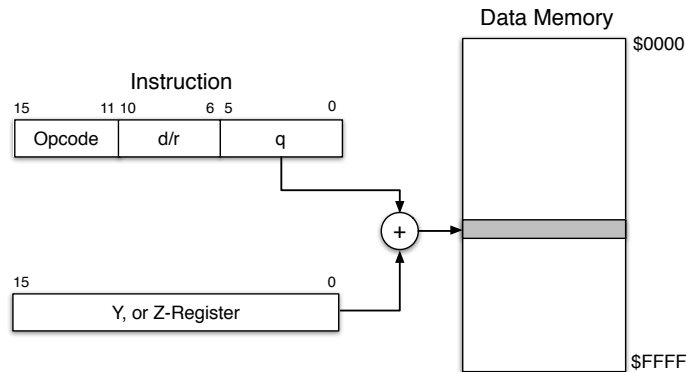
(b) Indirect addressing with displacement.

Figure 4.8: Indirect addressing modes.

the registers in the 160 extended I/O register address space (see Table C.2), instructions with direct addressing capability, i.e., `LDS` and `STS`, will have to be used.

4.3.3 Indirect Addressing Mode

As illustrated in Figure 4.7(a), direct addressing requires 32-bit instructions and thus their frequent use increases code size. *Indirect Addressing* allows an effective address to be put into an address register X, Y, or Z, and yet still implement these instructions with the 16-bit instruction format. In

addition, since the effective address is now in a pair of GPRs, it can easily be manipulated to provided added flexibility. Figure 4.8 shows examples of the indirect addressing mode.

The most common way to use indirect addressing is with the following two instructions illustrated in Fig. 4.8(a):

```
LD Rd, Y
ST X, Rr
```

For the `LD` (*Load indirect*) instruction, the effective address of the operand to be loaded into `Rd` is in one of the X, Y, or, Z-register specified by the *e* field of the instruction format. Similarly, `ST` (*Store indirect*) stores the contents of register `Rr` to the memory location pointed to by the effective address in one of the X, Y, or Z-register.

A variation of indirect addressing is to allow for *displacements*. The following two instructions use *indirect addressing with displacement*.

```
LDD Rd, Y+$10
STD Z+$20, Rr
```

Fig. 4.8(b) illustrates `LDD` (*Load indirect with displacement*) and `STD` (*Store indirect with displacement*) instructions. Both of these instructions calculate the effective address by adding the address in one of the X, Y, or Z-register with a 6-bit displacement in the *q* field of the instruction format. These instructions are useful for accessing data structures. For example, an address register would act as a base pointer for an array, and then a displacement would represent an offset to an element of the array.

Another variation of indirect addressing is to have *pre-decrement* and *post-increment* capabilities. This is shown in Figure 4.9. The following two instructions use *Indirect Addressing with Pre-Decrement*:

```
LD Rd, -Y
ST -Y, Rr
```
The following two instructions use *Indirect Addressing with Post-Increment*:

```
LD Rd, Y+
ST Y+, Rr
```

As the name suggests, pre-decrement decrements an address register and then it is used to access an operand as shown in Figure 4.9(a). On the other hand, post-increment uses an address register to access an operand and then the address register is incremented as shown in Figure 4.9(b).
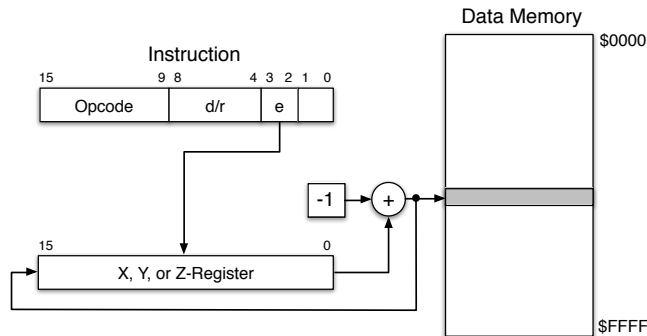
(a) Indirect addressing with pre-decrement.
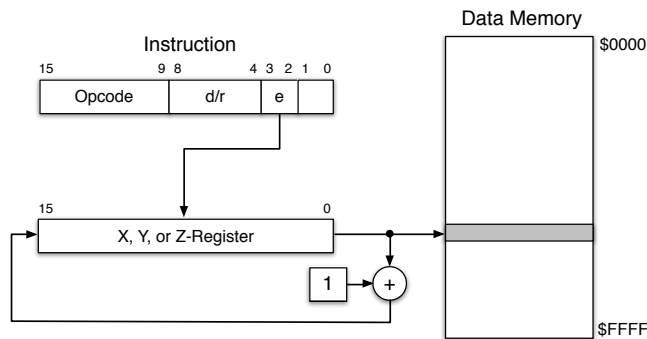
(b) Indirect addressing with post-increment.

Figure 4.9: Indirect addressing with pre-decrement and post-increment.

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/decrement address registers.

4.3.4 Program Memory Addressing Mode

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

Fig. 4.10 illustrates *Program Memory Constant Addressing*. The following variations of the LPM (*Load program memory*) instruction use Program
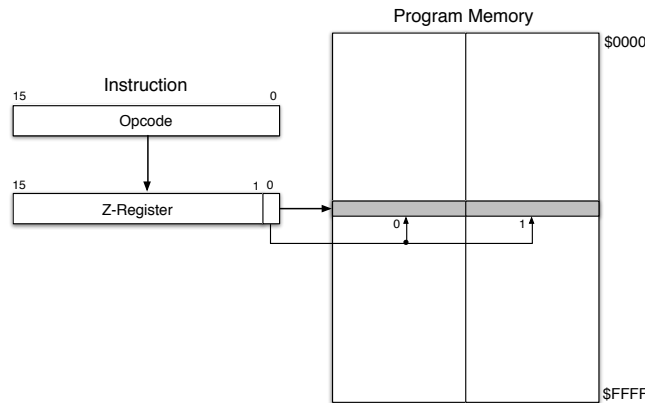
Figure 4.10: Program memory constant addressing.

Memory Constant Addressing:

```
LPM
LPM Rd, Z
LPM Rd, Z+
```

LPM can be used to access either 8-bit or 16-bit constants stored in the Program Memory. 8-bit constants are stored in consecutive 8-bit locations of the Program Memory and the least significant bit of the Z-register distinguishes between the first (left) and second (right) 8-bit constants stored in one Program Memory word. There are also three variations of LPM. The first option is simply LPM, where destination and source are *implicitly* defined as R0 and Z, respectively. The second option is to explicitly define the destination as well as Z. The final option is to use it with post-increment capability.

Fig. 4.11 shows variations of *Program Memory Addressing*, which affect how *target address* is generated for *jump* (JMP and IJMP) and *subroutine call* (CALL and ICALL) instructions. Figure 4.11(a) illustrates *Direct Program Memory Addressing*, where the second 16 bits of the 32-bit instruction represents the target address. In contrast, Figure 4.11(b) shows *Indirect Program Memory Addressing* that uses the Z-register to hold the target address.

Fig. 4.12 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 a signed 12-bit *displacement* to generate
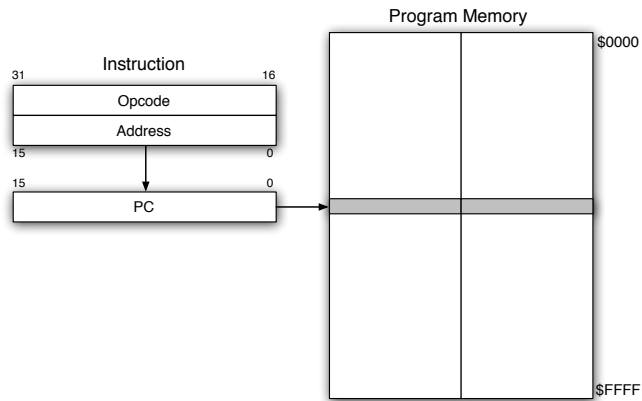
(a) Direct Program Memory addressing.
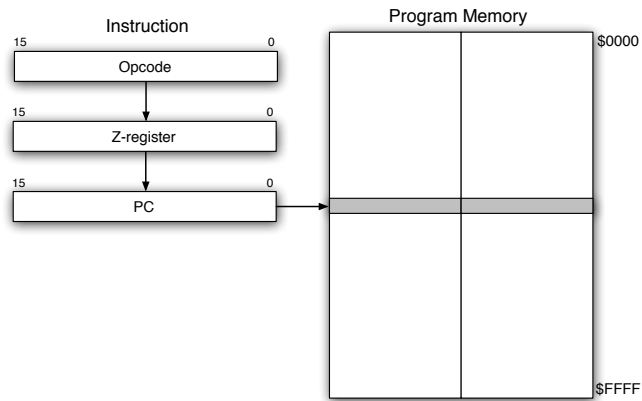
(b) Indirect Program Memory addressing.

Figure 4.11: Program memory addressing modes.

the target address. These types of instructions are also referred to as *PC-relative* jumps. The two AVR assembly instructions that use this addressing mode are RJMP (*Relative jump*) and RCALL (*Relative subroutine call*). The signed 12-bit displacement is represented as a two's-complement number, which allows the displacement to be between $-2^{11} = -2,048$ and $2^{11} - 1 = 2,047$. There is also a variation of this where a 7-bit displacement is used instead of 12 bits, which is used by all the conditional branch instructions.
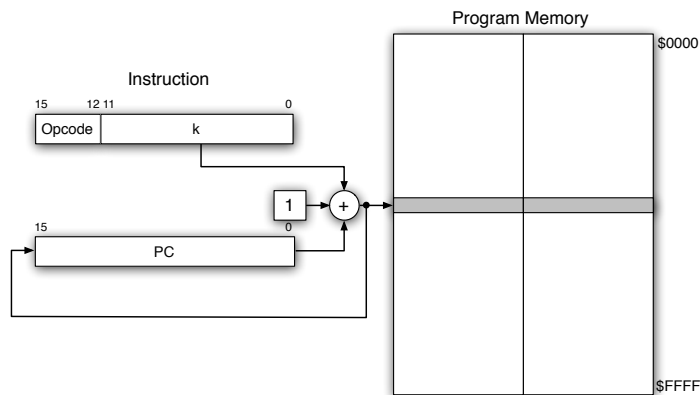
Figure 4.12: Relative Program Memory constant addressing.

4.4   Instructions

AVR has 134 different instructions. These instructions fall into four categories; (1) Data Transfer, (2) Arithmetic and Logic, (3) Control Transfer (branch/jump),(4) Bit and Bit-test, and (5) MCU Control. Appendix A provides a complete listing of all the AVR instructions. This section discusses the first four categories of instructions. Note that the coverage of these instructions is by no means complete, but it should be sufficient for you to be proficient in writing AVR assembly programs.

Before these instructions are presented, we first discuss the format or syntax of AVR instructions. Each AVR assembly instruction in a line has the following syntax:

Syntax:
label: *mnemonic   operand(s)*   ; *Comment*

An instruction consists of *mnemonic* and *operand(s)*. A *mnemonic* represents an operation, and it typically requires one or two *operands* to form an instruction (e.g., `ADD R0, R1`). Also, the AVR assembler accepts both upper- and lower-case letters (e.g., `add r0, r1`). Each instruction can also be assigned an optional *label* delimited by ":" to indicate to the assembler that the location of this instruction has a symbolic name. Any text between ";" and End-Of-Line (EOL) is considered as a comment and is ignored by the assembler.

Now that we have discussed the syntax, we are ready to explore the AVR instructions.

4.4.1   Data Transfer Instructions

The majority of instructions in any assembly program are *data transfer* instructions. These instructions essentially move data between GPRs and memory. Moreover, the location of data to be moved is dictated by the addressing modes discussed in Section 4.3.

Table 4.1 shows the two instructions that move data between registers: `MOV` and `MOVW`. The `MOV` instruction transfers data from `Rr` to `Rd`, where both `Rr` and `Rd` can be any one of the 32 GPRs. On the other hand, `MOVW` moves 16-bit data by concatenating `Rr+1` and `Rr`, i.e., `Rr+1:Rr`, and moving it to `Rd+1` and `Rd`, i.e., `Rd+1:Rd`. This instruction is particularly useful for moving an address pointer from one address register (i.e., X-, Y-, or Z-register) to another. The following example assembly code moves the content of Y-register to Z-register

```
; AVR assembly code - Move Y to Z
     MOVW  R30, R28    ; Move R29:R28 (Y) to R31:R30 (Z)
```

Note that these move instructions represent copy operations, thus the contents of `Rr` and `Rr+1` are not destroyed. Also, these instructions are examples of Register Addressing.

Table 4.1: Move Instructions.

Move Instructions		
Instruction	Operation	Description
MOV Rd, Rr	Rd ← Rr	Copy register
d = 0, 1, ..., 31 r = 0, 1, ..., 31		
MOVW Rd, Rr	Rd+1:Rd ← Rr+1:Rr	Copy register word
d = 0, 2, ..., 30 r = 0, 2, ..., 30		

`LD` and `ST` loads and stores 8-bit data from and to Data Memory, respectively. Table 4.2 defines the `LD` and `ST` instructions and their variations.

`LD` and `ST` instructions use Indirect Addressing mode and the address registers X, Y, and Z hold the effective addresses. Both the destination register `Rd` for `LD` and the source register `Rr` for `ST` can be any one of the 32

Table 4.2: Load and Store Instructions

Load and Store Instructions		
Instruction	Operation	Description
`LD Rd,` *src*	$Rd \leftarrow M(src)$	Load indirect (& with pre-decrement/post-increment)
`d = 0, 1, ..., 31` *src* = `X, X+, -X, Y, Y+, -Y, Z, Z+,` or `-Z`		
`ST` *dst*`, Rr`	$M(dst) \leftarrow Rr$	Store indirect
dst = `X, X+, -X, Y, Y+, -Y, Z, Z+,` or `-Z` `r = 0, 1, ..., 31`		
`LDD Rd,` *src*	$Rd \leftarrow M(src)$	Load indirect with displacement
`d = 0, 1, ..., 31` *src* = `Y+q` or `Z+q` `q` = 6-bit displacement (0≤`q`≤63) represented in decimal (no prefix), binary (prefix `0b`), octal (prefix `0`), or hex (prefix `0x` or `$`)		
`STD` *dst*`, r`	$M(dst) \leftarrow Rr$	Store indirect with displacement
dst = `Y+q` or `Z+q` `r = 0, 1, ..., 31` `q` = 6-bit displacement (0≤`q`≤63) represented in decimal (no prefix), binary (prefix `0b`), octal (prefix `0`), or hex (prefix `0x` or `$`)		
`LDI Rd, K`	$Rd \leftarrow K$	Load immediate
`d = 16, 17, ..., 31` `K` = 8-bit value (0≤`K`≤255) represented in decimal (no prefix), binary (prefix `0b`), octal (prefix `0`), or hex (prefix `0x` or `$`)		
`LDS Rd, k,`	$Rd \leftarrow M(k)$	Load direct from SRAM
`d = 0, 1, ..., 31` `k` = 16-bit address		
`STS k, Rr`	$M(k) \leftarrow Rr$	Store direct to SRAM
`k` = 16-bit address `r = 0, 1, ..., 31`		

GPRs. Similarly, *src* for `LD` and *dst* for `ST` can be any one of the address registers with pre-decrement/post-increment options. As shown in Figure 4.9, '−' sign in front of and '+' signs after the address registers represent pre-decrement and post-increment operations, respectively. As discussed in Section 4.3.3, these two features are very useful for stepping through an array of elements from beginning to end, and vice versa. This is illustrated in Figure 4.13. These features are typically used in a loop, and they eliminate the need to separately increment/decrement pointers.

`LDD` and `STD` allow for a displacement off of a base address in an address register. This type of addressing mode is useful for accessing an element of a data structure. For example, when a structure is declared in a high-level language, its members are allocated to consecutive memory locations. Thus,

(a) Post-increment.　　　(b) Pre-decrement.
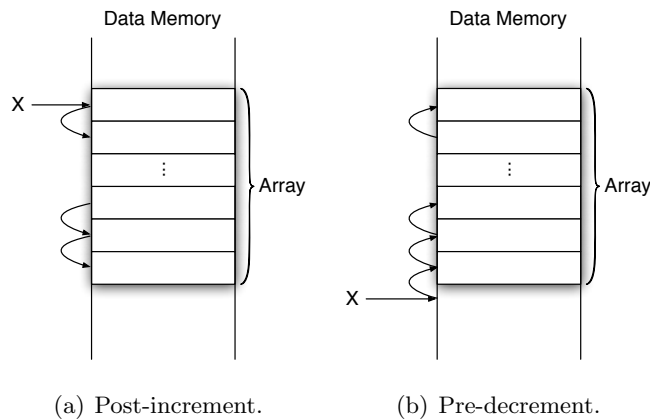
Figure 4.13: Post-increment and pre-decrement operations.

the base (i.e., Y- or Z-register) points to the beginning of the structure and each member can be accessed using a fixed displacement or index off of the base. The following assembly code example shows how STD can be used to store a value to an element of a structure.

```
; AVR assembly code - Store a value to element of structure
; Assume Y points to beginning of a structure
      ADD   R1, R2      ; Add two values
      STD   Y+4, R1     ; Store result to location offset by 4 bytes
```

As another example, Figure 4.14 illustrates how an element of an array can be accessed using indirect addressing with displacement. The Y-register serves as the base pointer for the array A, and the ith element of the array can be accessed by adding i to Y. Note that only Y- or Z-register can be used as an address register.

LDI allows an immediate (or constant) value to be moved into a register. Note that the destination register (Rd) for this instruction must be in the upper 16 GPRs (R16 - R31). The following example assembly code shows how LDI can be used to add a constant to a register.

```
; AVR assembly code - Add a constant to a register
      LDI   R16, 24     ; Load immediate value 24 into R16
      Add   R1, R16     ; Add it to R1
```

Note that 24 can be replaced with 0b00011000, $18, or 0x18.
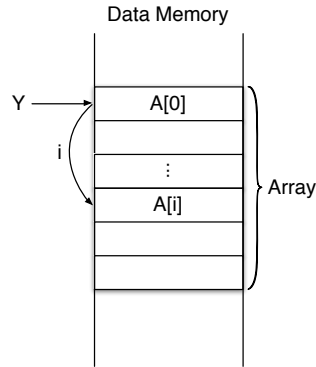
Figure 4.14: Indirect with displacement.

The LDI instruction is important for another reason. The address registers X, Y, and Z used by load and store instructions need to be initialized with pointers to operands. In order to understand the process of loading pointers to these registers, suppose an operand is in Data Memory location $0F10 (i.e., the effective address of the operand is $0F10). Then, the following assembly code initializes the Y-register to point to the operand.

```
; AVR assembly code - A code that initializes Y with address $0F10
     LDI   R29, $0F    ; Load $0F to upper byte of Y
     LDI   R28, $10    ; Load $10 to lower byte of Y
```

LDI limits the immediate value to be loaded to 8 bits. Therefore, the 16-bit address is moved in parts to the upper (R29) and the lower (R28) bytes of Y-register. Afterwards, the Y-register points to the operand, which can then be loaded from the Data Memory to a GPR using the LD instruction. Note that the store equivalent of LDI does not exist.

LDS and STS use Direct Addressing to move an 8-bit data between Data Memory and GPRs. Direct Addressing requires a 16-bit address to be encoded in the second half of the instruction format. LDS and STS basically have the same functionality as LD and ST, respectively, except that LDS and STS encode the address of the operand, i.e., effective address, directly into the instruction format, while LD and ST use an address register. This allows LDS and STS to access data from the Data Memory without having to separately load high and low bytes of the effective address to upper and lower bytes of an address register. However, these instructions require two

Program Memory words and the effective address encoded in the instruction cannot be modified.

Table 4.3 shows the `LPM` instruction, which is used to access Program Memory (indicated by M_P) rather than Data Memory. `LPM` relies on `Z`-register as a pointer to the Program Memory and can be combined with the post-increment capability. This instruction also has a special format where `LPM` can be used without *dst* and *src* fields.

Table 4.3: Load Program Memory instruction.

Load Program Memory instructions		
Instruction	Operation	Description
`LPM Rd`, *src*	`Rd` $\leftarrow M_P(src)$	Load program memory
`d` = 0, 1, ..., 31		
src =`Z` or `Z+`		
`LPM`	`R0` $\leftarrow M_P($`Z`$)$	Load program memory
`R0` (implied)		
`Z` (implied)		

Table 4.4 shows `PUSH` and `POP` instructions, which pushes and pops data on and off the stack, respectively. The discussion on stack operations requires special treatment. A stack is implemented as *last-in, first-out* (LIFO), and is one of the most important data structures in computer science and engineering. For example, many compilers use a stack for parsing the syntax of expressions, program blocks, etc. before translating it 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.

Table 4.4: Stack manipulate instructions.

Push and Pop Instructions		
Instruction	Operation	Description
`PUSH Rr`	STACK $\leftarrow$ `Rr`	Push register on stack
`r` = 0, 1, ..., 31		
`POP Rd`	`Rd` $\leftarrow$ STACK	Pop register from stack
`d` = 0, 1, ..., 31		

The following assembly code demonstrates how values 0x32 and 0x24 are pushed onto the stack:

(a) Stack initialized.

(b) After executing the second PUSH R22.

(c) After executing POP R22.
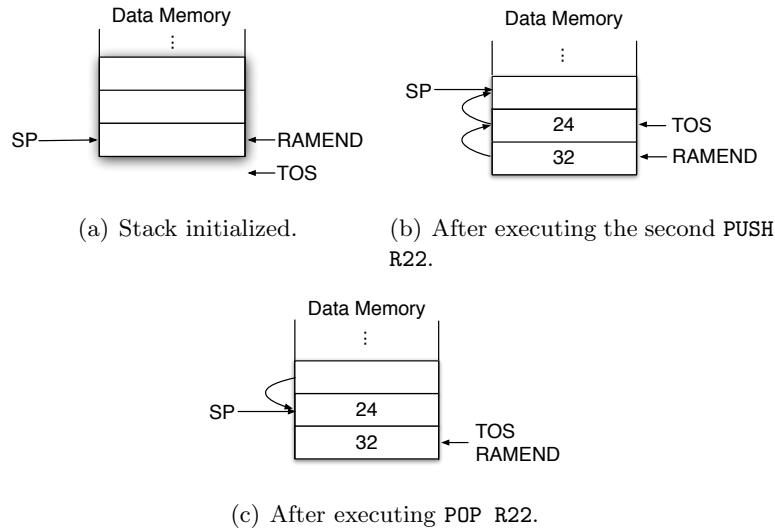
Figure 4.15: Push and pop operations.

```
; AVR assembly code - Push $32 and $24 onto the stack
     LDI   R22, $32    ; Load $32 into R22
     PUSH  R22         ; Push $32 on to the stack
     LDI   R22, $24    ; Load $24 into R22
     PUSH  R22         ; Push $24 to the stack
     ...
     POP   R22         ; Pop TOS (i.e., $24) to R22
```

Figure 4.15 illustrates the above code. There are a couple of things being implied with stack manipulations. First, the SP register is assumed to have been initialized, usually with the address of the end of the Data Memory indicated as RAMEND. This is shown in Figure 4.15(a). Second, SP is used to point to the Top Of the Stack (TOS). However, the actual content of TOS resides in the Data Memory location that is one address higher than SP. This way, TOS is beyond the range of the Data Memory and thus the stack is considered *empty*. Figure 4.15(b) shows the content of the stack after the second PUSH operation, which indicates $24 is the content of the TOS. Figure 4.15(c) shows the content of the stack after POP. Note that POP is a copy operation and thus $32 is not destroyed, but it is irrelevant within the context of the stack.

The final category of Data Transfer instructions are I/O instructions that allow data to be read from or written to I/O registers. Table 4.5 shows

the format of I/O instructions. The `IN` instruction moves data from one of the 64 I/O registers to one of the GPRs. The `OUT` instruction moves data from one of the GPRs to one of the 64 I/O registers. I/O registers are located between $0020 and $005F in Data Memory and consist of I/O port registers, I/O control registers, as well as a number of special registers. I/O operations will be discussed in detail in Chapter 5.

Table 4.5: I/O instructions.

I/O Instructions		
Instruction	Operation	Description
`IN Rd, A`	`Rd ← I/O(A)`	In port
`d = 0, 1, ..., 31`		
`A = 0, 1, ..., 63`		
`OUT A, Rr`	`I/O(A) ← Rd`	Out port
`A = 0, 1, ..., 63`		
`r = 0, 1, ..., 31`		

4.4.2   Arithmetic and Logic Instructions

Most of the *arithmetic and logic* instructions operate on two 8-bit operands and can modify S, V, N, Z, and C condition codes in the SREG (see Figure 4.5). Table 4.6 shows a breakdown of the most commonly used Arithmetic and Logic instructions.

Table 4.6: Commonly Used Arithmetic and Logic Instructions.

Description	Instructions
Addition	`ADD, ADC, ADIW`
Subtraction	`SUB, SUBI, SBC, SBCI, SBIW`
Logic	`AND, ANDI, OR, ORI, EOR`
Complement	`COM, NEG`
Register Bit Manipulation	`SBR, CBR`
Register Manipulation	`INC, DEC, TST, CLR, SER`
Multiplication	`MUL, MULS, MULSU`

The format for arithmetic and logic instructions is shown in Table 4.7. Instructions `ADD`, `ADC`, `SUB`, `SBC`, `AND`, `OR`, and their immediate versions `SUBI`, `ANDI`, `ORI`, are relatively straightforward.

The following example assembly code shows how a constant can be added to a register:

Table 4.7: Arithmetic and Logic Instructions.

Arithmetic and Logic Instructions		
Instruction	Operation	Description
Two registers		
ADD Rd, Rr	Rd ← Rd + Rr	Add two registers
ADDC Rd, Rr	Rd ← Rd + Rr + C	Add with carry two registers
SUB Rd, Rr	Rd ← Rd − Rr	Subtract two registers
SBC Rd, Rr	Rd ← Rd − Rr − C	Subtract with carry two registers
AND Rd, Rr	Rd ← Rd ∧ Rr	Logical AND registers
OR Rd, Rr	Rd ← Rd ∨ Rr	Logical OR registers
EOR Rd, Rr	Rd ← Rd ⊕ Rr	Exclusive OR registers
d = 0, 1, ..., 31		
r = 0, 1, ..., 31		
Register and Immediate Value		
SUBI Rd, K	Rd ← Rd − K	Subtract constant from register
ANDI Rd, K	Rd ← Rd ∧ K	Logical AND register and constant
ORI Rd, K	Rd ← Rd ∨ K	Logical OR register and constant
d = 16, 17, ..., 31		
K = 8-bit value (0≤K≤255) represented in decimal (no prefix), binary (prefix 0b), octal (prefix 0), or hex (prefix 0x or $)		

```
; AVR assembly code - Add a constant to a register
     LDI   R16, 24     ; Load immediate value 24 into R16
     ADD   R1, R16     ; Add it to R1
```

Here is another example assembly code that uses the `ADC` instruction to add an 8-bit value to the 16-bit X-register.

```
; AVR assembly code - Add a constant to X-register
     LDI   R16, 0      ; Zero R16
     LDI   R17, 0X18
     ADD   R26, R17    ; Add 24 to low byte of X-register
     ADC   R27, R16    ; Add carry (if set) to R27
```

Note that in this code, the decimal value 24 was represented as hexadecimal value $18.

The following assembly code shows an example of a logical AND operation:

```
; AVR assembly code - AND R1 and R2
     LDI   R16, 0b00001111   ; Load 15 into R16
     LDI   R17, 0b00001010   ; Load 10 into R17
     AND   R17, R16          ; AND 15 and 10, result => R17=10
```

In the above example, 15 and 10 are represented as binary numbers 00001111 and 00001010, respectively, using the prefix `0b`. Performing a logical AND operation on these two binary values results in 00001010, and this is stored in R17.

Unlike arithmetic and logic instructions that operate on 8-bit data, `ADIW` and `SBIW` instructions operate on 16-bit data. The format of these two instructions is shown in Table 4.8. These instructions add and subtract an 8-bit value to and from a 16-bit address register, i.e., X-, Y-, or Z-register. This allows an entire address register to be updated with an arbitrary value and simplifies manipulation of pointers. The following example assembly code increments X-register by 4.

```
; AVR assembly code - Increment X by 4
     ADIW  R26, 4     ; Add 4 to R27:R26 (X)
```

Without this instruction, the constant would first have to be added to the lower byte of X-register, and then the upper byte of X-register would have to be updated using `ADC`.

Table 4.8: Add/Subtract Immediate to/from word.

Add/Subtract Immediate to/from Word Instructions		
Instruction	Operation	Description
`ADIW Rd+1:Rd, K`	`Rd+1:Rd ← Rd+1:Rd + K`	Add immediate to word
`SBIW Rd+1:Rd, K`	`Rd+1:Rd ← Rd+1:Rd − K`	Subtract immediate from word
d = 26, 28, 30 K = 6-bit value (0≤K≤63) represented in decimal (no prefix), binary (prefix `0b`), octal (prefix `0`), or hex (prefix `0x` or `$`)		

`COM` and `NEG` instructions are used to perform one's-complement and two's-complement operations, respectively. The formats for these two instructions are shown in Table 4.9. The assembly code shown below negates the value 33 by taking the two's-complement of it:

Table 4.9: Complement and Negate Instructions.

Complement and Negate Instructions		
Instruction	Operation	Description
`COM Rd`	`Rd ← $FF − Rd`	One's complement
`NEG Rd`	`Rd ← $00 − Rd`	Two's complement
Rd = 0, 1, ..., 31		

```
; AVR assembly code - Negate 33
      LDI   R16, 33    ; 33 = 0b00100001
      COM   R16        ; 0b11011110
      INC              ; 0b11011111
```

This code takes 33, which is 0b00100001, and first performs one's-complement to generate 0b11011110. This is then incremented by 1 to generate 0b11011111. In order to verify that 0b1101111 is indeed -33, we take the two's-complement which results in 0b00100001. Of course, we could have also used the following code:

```
; AVR assembly code - Negate 33, simpler way
      LDI   R16, 33    ; 33 = 0b00100001
      NEG   R16        ; 0b11011111
```

SBR and CBR instructions set and clear bits in a register, respectively. The formats for these two instructions are shown in Table 4.10. SBR sets the bits in a register by performing a logical OR with an 8-bit constant K. On the other hand, CBR clears the bits in a register by taking K and inverting its bits (i.e., take the one's-complement) and then performing logical AND operations. For example, the following instruction can be used to set the bits 7-4 of a destination register R17:

Table 4.10: Set/Clear Bits in Register Instructions.

Set/Clear Bits in Register Instructions		
Instruction	Operation	Description
SBR Rd, K	Rd ← Rd ∨ K	Set bit(s) in register
CBR Rd, K	Rd ← Rd ∧($FF−K)	Clear bit(s) in register
d = 16, 17, ..., 31 K = 8-bit value (0≤K≤255) represented in decimal (no prefix), binary (prefix 0b), octal (prefix 0), or hex (prefix 0x or $)		

```
      SBR   R17, $F0          ; Set R17 with 0b11110000
```

Note that SBR is equivalent to LDI. On the other hand, the following CBR instruction clears the bits that was set by the above SBR:

```
      CBR   R17, $F0          ; Clear R17 by ANDing with 0b00001111
```

Table 4.11: Unary Instructions.

Unary Instructions		
Instruction	Operation	Description
INC Rd	Rd ← Rd + 1	Inrement
DEC Rd	Rd ← Rd − 1	Decrement
TST Rd	Rd ← Rd ∧ Rd	Test for zero or minus
CLR Rd	Rd ← Rd ⊕ Rd	Clear register
SER Rd	Rd ← $FF	Set register
d = 0, 1, ..., 31		

The reason this works is because one's-complement of 0b1111000 is 0b0000111, which is also equivalent to performing $FF−$F0=$0F. When 0b00001111 is ANDed with 0b11110000, all the bits are cleared.

INC, DEC, TST, CLR, and SER are unary instructions that operate on one register. Table 4.11 shows the format of these instructions. INC and DEC are typically used to update loop counters. Based on this update, TST can be used to determine whether or not a loop should exit. CLR and SER instructions are used to clear and set a register, respectively. The following example assembly code shows how some of these instructions can be used to implement a loop.

```
; AVR assembly code - A simple loop
      LDI   R16, 24     ; Load loop count 24 into R26
Loop:
      TST   R16         ; Test if zero
      BREQ  Exit        ; If zero, exit loop
      ...               ; Do something
      DEC   R16         ; Decrement loop count
      RJMP  Loop        ; Jump to Loop
Exit:
      ...               ; Continue with program
```

The above code first loads the loop count to R16. Then, at the beginning of the loop, a test is made to see if the loop count is zero. As long as the loop count is not equal to zero, it is decremented and the loop repeats. When the loop count becomes zero, BREQ is satisfied and the loop exits. Note that this loop executes 24 times.

The final group of arithmetic and logic instructions are the *multiply* instructions. Table 4.12 shows these instructions. The MUL instruction multiplies two 8-bit operands and generates a 16-bit result, where the upper and lower bytes of the result are stored in registers R1 and R0, respectively.

Table 4.12: Multiply Instructions.

Multiply Instructions		
Instruction	Operation	Description
`MUL Rd, Rd`	R1:R0 ← Rd × Rr	Multiply unsigned
`MULS Rd, Rr`	R1:R0 ← Rd × Rr	Multiply signed
`MULSU Rd, Rr`	R1:R0 ← Rd × Rr	Multiply signed with unsigned
`d` = 0, 1, ..., 31		
`r` = 0, 1, ..., 31		

The following assembly code shows an example of multiplying two numbers:

```
; AVR assembly code - A multiply operation
    LDI   R16, 32          ; Load 32 into R16
    LDI   R17, 8           ; Load 8 into R17
    MUL   R16, R17         ; 32 x 8 = 256
```

The above code multiplies 32 and 8, which results in 256 or $0100. Thus, R1 would contain $01 and R0 would contain $00.

MULS and MULSU are signed versions of MUL, where the former assumes both operands are signed numbers, while the latter assumes only the second operand (i.e., Rr) is a signed number.

4.4.3   Control Transfer Instructions

Control transfer instructions are used to change the flow of control within a program. Typically, around 20%∼ 25% of the instructions in a program are control transfer instructions. A control transfer instruction basically modifies the PC and redirects where the next instruction will be fetched. Without these instructions it would not be possible to implement if-else statements, case statements, and functions in high-level languages. There are two types of control transfer instructions: conditional and unconditional branches.

Conditional Branches

A *conditional branch* will modify the PC if the corresponding condition is met. In AVR, the condition is determined by condition codes or flags in SREG (see Figure 4.5). For example, the BREQ (*Branch if equal*) instruction will test the Zero (Z) flag of SREG. If Z = 1, 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 condition flags in SREG. All arithmetic and logic instructions can affect the SREG bits. However, the most commonly used instructions to set condition flags are the *compare* instructions CP and CPI. These instructions are then immediately followed by conditional branch instructions that test condition flags to determine whether or not branches should be taken.

Table 4.13 shows the format and meaning of these instructions. These compare instructions subtract the two signed values in the corresponding registers (one register and an immediate value in the case of CPI), and depending on their outcomes, modify the condition flags. For example, Z-flag is set to 1 if the compared values are equal; otherwise, Z-flag is set to 0.

Table 4.13: Compare Instructions

Compare Instructions		
Instruction	Operation	Description
CP Rd, Rr	Rd - Rr	Compare
CPC Rd, Rr	Rd - Rr -C	Compare with carry
CPI Rd, K	Rd - K	Compare register with immediate
CPSE Rd, Rr	If Rd = Rr then PC ← PC+2 (or 3) else PC ← PC+1	Compare skip if equal
d = 0, 1, ..., 31 or d = 16, 17, ..., 31 (immediate) r = 0, 1, ..., 31 K = 8-bit value ($0 \leq K \leq 255$) represented in decimal (no prefix), binary (prefix 0b), octal (prefix 0), or hex (prefix 0x or $)		
Flags Affected		
if Rd = Rr or K then Z = 1 if Rd ≠ Rr or K then Z = 0 if Rd ≥ Rr or K then N = 0, V = 0 or N = 1, V = 1 if Rd < Rr or K then N = 1, V = 0 or N = 0, V = 1		

In the cases of comparing whether one value is greater than equal to or less than the other value, both N and V flags are affected. For example, if Rd is greater than equal to Rr or K, the result will be positive and thus N-flag is set to 0. On the other hand, if Rd is less than Rr or K, the result will be negative and thus N-flag is set to 1. However, if one of the values is negative, it could generate a result that overflows. For example, suppose Rd = 126 and Rr = -35. Subtracting the two values results in 161, which is larger than the maximum value of 127, and causes V-flag to be set to 1

indicating overflow and N-flag will be set to 1. Similarly, if `Rd` = -126 and `Rr` = 35, subtracting them would generate -161, which is lower than the minimum value -128, and causes V-flag to be set to 1 and N-flag will be set to 0. Note that unlike Arithmetic and Logic instructions, these compare instructions do not update any registers.

The `CPC` instruction can be used together with `CP` to compare 16-bit values in two register pairs. The idea is similar to using `ADD` and `ADC` to add two 16-bit values. The `CP` instruction is first used to subtract the two low-byte values. Then, `CPC` is used to subtract the two high-byte values, which causes appropriate condition flags to be set.

Unlike the previous three compare instructions, `CPSE` does not affect any condition flags. Instead, it conditionally changes the control flow by incrementing the PC by one (if the next instruction is one word) or two (if the next instruction is two words).

Once condition flags are updated by a compare instruction, a conditional branch instruction can be used to test these flags. Table 4.14 shows commonly used conditional branch instructions and what condition flags are evaluated.

Table 4.14: Conditional Branch Instructions

Conditional Branch Instructions		
Instruction	Operation	Description
`BREQ` *label*	if (Z = 1) then PC ← PC+k+1	Branch if equal
`BRNE` *label*	f (Z = 0) then PC ← PC+k +1	Branch if not equal
`BRGE` *label*	if (N⊕V= 0) then PC ← PC+k+1	Branch if greater or equal, signed
`BRLT` *label*	if (N⊕V= 1) then PC ← PC+k +1	Branch if less than, signed
`BRCC` *label*	if (C = 0) then PC ← PC+k +1	Branch if carry cleared
`BRCS` *label*	if (C = 1) then PC ← PC+k +1	Branch if carry set
`BRPL` *label*	if (N = 0) then PC ← PC+k +1	Branch if plus
`BRMI` *label*	if (N = 1) then PC ← PC+k +1	Branch if negative
`BRVC` *label*	if (V = 0) then PC ← PC+k +1	Branch if overflow flag is cleared
`BRVS` *label*	if (V = 1) then PC ← PC+k +1	Branch if overflow flag is set
label= PC + k + 1		
k = 7-bit displacement (-64 ≤ K ≤ 63)		

The following example assembly code shows how compare and conditional branch instructions are used to implement control flow:

```
; AVR assembly code - Equivalent assembly for the IF statement
      CP    R0, R1      ; Compare R0 with R1
      BRGE  NEXT        ; Jump to NEXT if R0 >= R1
      ...               ; If R0 < R1, do something
NEXT: ...               ; Otherwise, continue on
```

Table 4.15: Skip Instructions

Skip Instructions		
Instruction	Operation	Description
SBRC Rd, *bit*	if (Rd(*bit*)=0) PC ← PC+2 or 3	Skip if bit in register is cleared
SBRS Rd, *bit*	if (Rd(*bit*)=1) PC ← PC+2 or 3	Skip if bit in register is set
SBIC A, *bit*	if (I/O(A, *bit*)=0) PC ← PC+2 or 3	Skip if bit in I/O register is cleared
SBIS A, *bit*	if (I/O(A, *bit*)=1) PC ← PC+2 or 3	Skip if bit in I/O register is set
d = 0, 1, ..., 31		
A = 0, 1, ..., 31 (only the lower 32 I/O registers)		
bit = 0, 1, ..., 7		
PC + 2 or 3 depending whether the next instruction is 16-bit or 32-bit.		

This code tests whether R0 is less than R1. If the condition is true, the instruction associated with the condition is executed; otherwise, the instruction is skipped.

Conditional branches also include the skip instructions SBRC, SBRS, SBIC, and SBIS. Table 4.15 shows the meaning of these instructions. These instructions will skip the next instruction if the condition is met, and is very useful in, for example, waiting for some status to be set. The following example code waits for bit 0 of PINA to be set.

```
; AVR assembly code - Test and loop on bit 0 on port A
LOOP:
     SBIS  PINA, 0    ; Skip next instruction if bit 0 on PINA is  set
     RJMP  LOOP       ; Jump back if not set
     ...              ; Do something if bit is set
```

This test is repeated as long as the bit is not set. When the bit is set, it exists the loop. Note that this code assumes the *m128def.inc* definition file is included, which maps PINA to the the location $19 in the I/O register address space.

Unconditional Branches

Unconditional branches modify the PC without any conditions. These instructions are known as *jumps* because they cause the program to "jump" to another location in Program Memory. Table 4.16 shows the jump instructions. Among these, RJMP is the most common because it is implemented as a PC-relative jump using a 16-bit instruction and does not require a separate address register or memory word to hold the target address. The JMP instruction is a 32-bit instruction, where the second 16-bit encodes the

Table 4.16: Jump Instructions

Jump Instructions		
Instruction	Operation	Description
RJMP *label*	PC ← PC+k+1	Relative jump
JMP *label*	PC ← k	Direct jump
IJMP *label*	PC ← Z	Indirect jump to (Z)
label = PC+k+1, k, or Z		
k = 12-bit (RJMP) or 16-bit (JMP)		

target address. The `IZMP` instruction is a 16-bit instruction, which requires a separate instruction to move the target address into the Z register.

There are also special unconditional branch instructions known as *subroutine calls*. These instructions are shown in Table 4.17. 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. The combination of subroutine calls and returns is used to implement functions in AVR assembly. The following shows an example skeleton code for subroutine call and return.

```
; AVR assembly code - Subroutine call and return
      ...
      RCALL SUBR              ; First call to SUBR
      ...
      RCALL SUBR              ; Second call to SUBR
      ...
SUBR:
      ...
      ...Do something...      ; Subroutine
      ...
      RET
```

In the above code, the subroutine begins with the instruction at label `SUBR` and ends with the `RET` instruction. Moreover, the first `RCALL` to subroutine `SUBR` returns to the instruction right after the first `RCALL` and the second `RCALL` returns to the instruction right after the second `RCALL`. Thus, a subroutine is written once and it can be called multiple times. See Section 4.9.2 for a more detailed discussion on subroutines.

Table 4.17: Subroutine Call and Return Instructions

Subroutine Call and Return Instructions		
Instruction	Operation	Description
RCALL *label*	PC ← PC+k+1, STACK ← PC+1	Relative subroutine call
CALL *label*	PC ← k, STACK ← PC+1	Direct subroutine call
ICALL *label*	PC ← Z, STACK ← PC+1	Indirect call to (Z)
RET	PC← STACK	Subroutine return
label = PC+k+1, k, or Z		
k = 12-bit (RCALL) or 16-bit (CALL)		

4.4.4 Bit and Bit-test Instructions

Bit and bit-test instructions manipulate or test individual bits within a register. There are two types of bit and bit-test instructions; *Shift and Rotate* and *Bit Manipulation*. The following discusses these two types of instructions.

Shift and Rotate

Shifting a register involves shifting all the bits one position either to the left or the right. The AVR ISA categorizes register shift operations as *shift* and *rotate*. Table 4.18 defines these instructions.

Table 4.18: Shift and Rotate Instructions

Shift and Rotate Instructions		
Instruction	Operation	Description
LSL Rd	Rd(n+1) ← Rd(n), Rd(0) ← 0	Logical shift left
LSR Rd	Rd(n-1) ← Rd(n), Rd(7) ← 0	Logical shift right
ROL Rd	Rd(0) ← C, Rd(n+1) ← Rd(n), C ← Rd(7)	Rotate left through carry
ROR Rd	Rd(7) ← C, Rd(n-1) ← Rd(n), C←Rd(0)	Rotate right through carry
ASR Rd	Rd(n-1) ← Rd(n), n=1, 2, ..., 7	Arithmetic shift right
SWAP Rd	Rd(3...0) ← Rd(7...4), Rd(7...4) ← Rd(3...0)	Swap nibbles
n = bit position		

The AVR instructions that perform shift left and right are LSL and LSR, respectively. Figure 4.16 illustrates these operations. LSL shifts in 0 to the 0th bit position and the 7th bit (b_7) is shifted out to the C-bit. On the other hand, LSR shifts in 0 to the 7th bit position and shift out the 0th bit (b_0) to the C-bit.

LSL and LSR instructions are useful for a couple of reasons. First, they can be used to test each bit in a GPR through the C-bit. Second, they

(a) Shift left.   (b) Shift right.
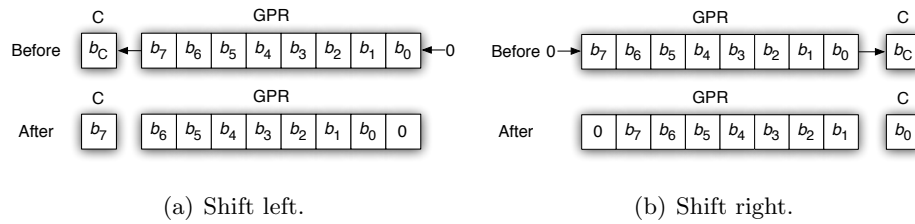
Figure 4.16: Logical shift left and right operations.

(a) Rotate left.   (b) Rotate right.
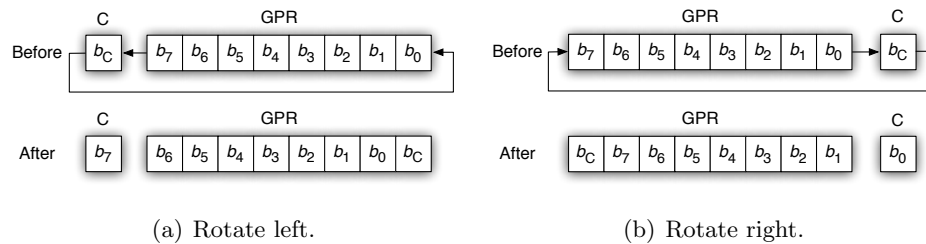
Figure 4.17: Rotate left and right operations.

can be used to perform multiplication and division by powers of two. For example, consider the binary representation of 34 shown below. Performing logical shift left results in $34 \times 2 = 68$, while logical shift right results in $34 \div 2 = 17$.

0 0 1 0 0 0 1 0 (34) – Initial value
0 1 0 0 0 1 0 0 (68) – After `LSL`
0 0 0 1 0 0 0 1 (17) – After `LSR`

Figure 4.17 illustrates rotate left and right operations. `ROL` will shift in b_C to the 0th bit and shift out the bit b_7 to the C-bit. *Rotate right* will shift in b_C to the 7th bit and shift out b_0 to the C-bit. Therefore, rotate operations will not lose any bits, while shift operations will loose the bits that are shifted out.

`ASR` behaves like `LSR` except it does not shift in a 0 to bit b_7. Instead, bit b_7 will not change to maintain the sign of the number being sifted. `ASR` can be used in conjunction with `LSR` to perform fast multiplication and division on *signed numbers*. For example, consider -34 represented in two's-complement shown below. As in the case of positive numbers, `LSL` performs $-34 \times 2 = -68$. On the other hand, `ASR` maintains the sign of the number during shifting and thus performs $-34 \div 2 = -17$.
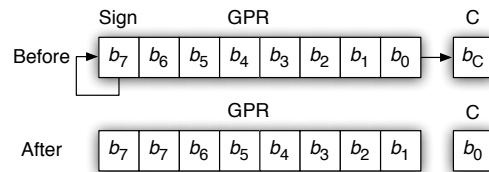
Figure 4.18: Arithmetic shift right.
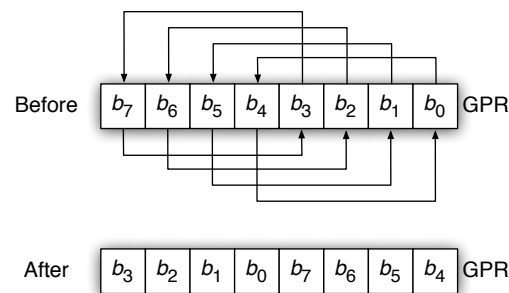
Figure 4.19: Swap nibbles.

1 1 0 1 1 1 1 0 (-34) – Initial value
1 0 1 1 1 1 0 0 (-68) – After `LSL`
1 1 1 0 1 1 1 1 (-17) – After `ASR`

The `SWAP` instruction swaps the upper and lower 4 bits with each other. This is illustrated in Figure 4.19. This instruction is useful for manipulation of Binary Coded Decimal (BCD) values.

Bit Manipulation

Bit Manipulation instructions allow individual bits within an I/O register or SREG to be set or cleared. Table 4.19 shows the most commonly used Bit Manipulation instructions.

`SBI` and `CBI` will set and clear, respectively, any bit in an I/O register. The following assembly code shows how `CBI` can be used to clear bit 0 of PINA after it has been set.

```
; AVR assembly code - Test and loop on bit 0 of PINA, and clear bit 0 when set
LOOP: SBIS  PINA, 0    ; Skip next instruction if bit 0 of PINA is set
      RJMP  LOOP       ; Loop if not set
```

Table 4.19: Bit Manipulation Instructions

Bit Manipulate Instructions		
Instruction	Operation	Description
`SBI A`, *bit*	$I/O(A,bit) \leftarrow 1$	Set bit in I/O register
`CBI A`, *bit*	$I/O(A,bit) \leftarrow 0$	Clear bit in I/O register
`SE`*f*	$f \leftarrow 1$	Set condition flag in SREG
`CL`*f*	$f \leftarrow 0$	Clear condition flag in SREG
`A` = 0, 1, ..., 31 (only the lower 32 I/O registers)		
bit = 0, 1, ..., 7		
f = I, T, H, S, V, N, Z, C		

```
    CBI   PINA, 0     ; Clear bit
    ...               ; Do something if bit was set
```

SE*f* and CL*f* instructions set and clear condition flag *f*, where *f* can be any one of the 8 bits in SREG. The following assembly instruction turns on the global interrupt facility, i.e., I-bit.

```
; AVR assembly code - Turn on global interrupt
    SEI               ; Set I-bit
```

4.5   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 instructions and registers, as well as memory locations using labels. These mnemonics are then assembled into *machine instructions* consisting of 0's 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 computers were first developed back in 1947, they were programmed using machine instructions (via mechanical switches) until assemblers and compilers came along! Second, the processor decodes and executes machine instructions; therefore, understanding how information from instructions is encoded is crucial for knowing how to design and develop processor hardware to decode and execute these instructions.

Figure 4.20 shows the machine instruction mapping for the following `ADD` instruction:

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

0000	11rd	dddd	rrrr

d dddd = 0 1111

r rrrr = 1 0000
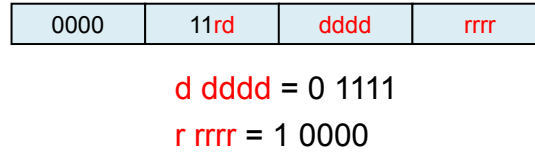
Figure 4.20: Machine instruction mapping for `ADD R15, R16`.

1001	000d	dddd	1000

d dddd = 1 0000
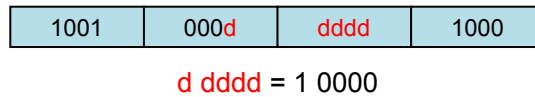
Figure 4.21: Machine instruction mapping for `LD R16, Y`.

The bit pattern '0000 11' represents the opcode for the `ADD` instruction, and the destination register identifier bits `d dddd` and the source register identifier bits `r rrrr` are located in the instruction format as shown in the figure. Since the destination and source register identifiers are `R15` and `R16`, `d dddd` and `r rrrr` will be '0 1111' and '1 0000', 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 4.21 shows the machine instruction mapping for the following `LD` instruction:

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

The location of the destination register identifier bits `d dddd` 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`, `d dddd` will be '1 0000'.

Figure 4.22 shows the machine instruction mapping for the following `LDI` instruction:

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

Again, the location of the destination register identifier `dddd` is the same as `ADD` and `LD`, but the most significant bit does not exist! The reason for this is that the instruction format has to also support the two-digit hexadecimal value `$F0`. Therefore, the instruction format forgoes the most significant bit

1110	KKKK	dddd	KKKK

1 dddd = 1 1110

KKKK KKKK = 1111 0000
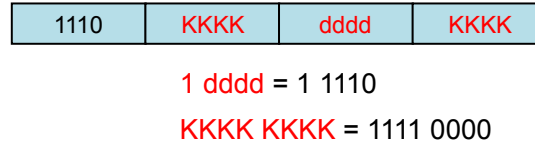
Figure 4.22: Machine instruction mapping for LDI R30, $F0.

10q0	qq0d	dddd	1qqq

d dddd = 0 0100

q qq qqq = 0 00 010
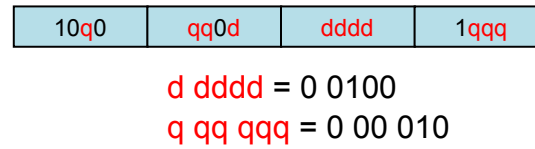
Figure 4.23: Machine instruction mapping for LDD R4, Y+2.

1011	0AAd	dddd	AAAA

d dddd = 1 1001

AA AAAA = 01 0110
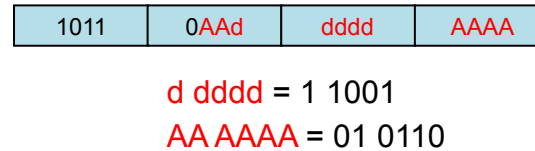
Figure 4.24: Machine instruction mapping for IN R25, $16.

of the destination register identifier, which is implied as 1 (and thus only
R16 - R31 can be accessed), to accommodate the 8-bit immediate value. The
immediate value is then represented by the constant field KKKK KKKK, which
is '1111 0000'.

Figure 4.23 shows the machine instruction mapping for the following LDD
instruction.

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

Again, the location of the destination register identifier d dddd is the same as
LD. The 6-bit displacement field q qq qqq is spread across the instruction
format and can represent any number between 0 and 63. Therefore, the
displacement 2 is encoded as '0 00 010'.

Figure 4.24 shows the machine instruction mapping for the following IN
instruction:

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

1111	00kk	kkkk	k001

kk kkkk k = 01 0010 1
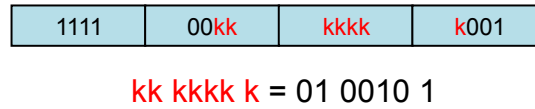
Figure 4.25: Machine instruction mapping for BREQ label.

Again, the location of the destination register identifier d dddd is the same as the other instructions. The bits AA AAAA flank the d dddd 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 the I/O register address is offset by 32. This is because the I/O register address space starts after the 32 GPRs (see Figure 4.3). 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 4.25 shows the machine instruction mapping for the following BREQ instruction:

```
BREQ  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 of label and the address of the next instruction (i.e., PC+1) to PC+1. This way, only the offset has to be stored in the instruction format. This is illustrated in the following code. The address of the instruction after BREQ is $0234, while the target address 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, R1
0233              BREQ  SKIP
0234              ...                 ; Next instruction
...               ...
0259      SKIP:
...               ...
```

1001	0100	0000	1110
kkkk	kkkk	kkkk	kkkk

kkkk kkkk kkkk kkkk = 0000 0011 1111 0000

Figure 4.26: Machine instruction mapping for `CALL label`.

Figure 4.26 shows the machine instruction mapping for the following `CALL` instruction:

```
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. `CALL` is one of the few 32-bit instructions in the AVR Instruction Set Architecture. Note that there are additional `k` bits in the first half of the instruction format (which are all 0's) to allow for future expansion of the Program Memory.

The `CALL` instruction has another important functionality; the *return address* of the subroutine call is pushed, or saved, on the stack. The following code illustrates this process:

Address   Code

```
...                  ...
0230                 CALL  SUBR       ; Subroutine call
0232                 ...              ; Next instruction
...                  ...
03F0         SUBR:
...                  ...
...                  ...Do something...; Subroutine
...                  ...
...                  RET
```

`CALL` is used together with `RET` to implement subroutine calls and returns. Therefore, after the subroutine (`SUBR`) is executed, `RET` jumps to the instruction after the `CALL`, which is the return address. This requires `CALL` to save the return address onto the stack before jumping to the subroutine. Figure 4.27(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, the higher and lower bytes of the return

(a) Return address pushed onto the stack.

(b) Return address is popped from the stack.
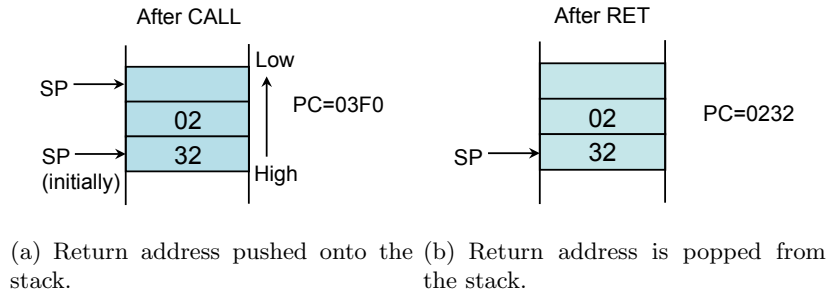
Figure 4.27: Stack manipulation for subroutine call and return.

address will be popped from the stack as shown in Figure 4.27(b) and moved to PC. Note that the return address still remains in Data Memory but it is irrelevant with respect to the stack.

4.6 Assembler Directives

Assembler directives are special instructions that are executed before the code is assembled, and they provide the assembler with information it needs to know in order to carry out the assembly process, e.g., code location, constant definitions, storage definitions, etc. These directives are denoted by a preceding dot '.', e.g., `.EQU`, and are not assembled into machine instructions. Instead, they are used to adjust the location of the program in memory, define macros, initialize memory, and so on. Table 4.20 provides an overview of the directives supported by the AVR assembler.

The following subsections discuss the most commonly used directives, which are `.ORG`, `.DB`, `.DW`, `.BYTE`, `.CSEG`, `.DSEG`, `.DEF`, `.EQU`, and `.INCLUDE`.

4.6.1 `.ORG` - Set program origin

The `.ORG` directive sets the location of a code or data to an absolute value given as a parameter. This allows a code or data to be placed anywhere in memory. The following shows the syntax of `.ORG`, where the location of the instruction following a `.ORG` directive is defined by its *expression*:

Syntax:
`.ORG`   *expression*

Table 4.20: AVR 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

The following example assembly code shows how `.ORG` can be used to define the starting address of a code:

```
; AVR assembly code - Example use of .ORG directive
.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
```

In the above code, `.ORG $0000` indicates that the `RJMP` instruction is in the Program Memory location $0000, and `.ORG $00042` indicates that the `MOV` instruction is in the Program Memory location $0042, and thus, the label `main` is equal to $0042. Note that if `.ORG $0000` is omitted, the address of `RJMP` defaults to zero.

4.6.2   `.DB` - Define constant byte(s)

The `.DB` directive defines 8-bit constants in Program Memory. In order to refer to the reserved locations, a label should precede the `.DB` directive. The syntax for `.DB` is shown below.

Syntax:
label: `.DB`   *expression_list*

The `.DB` directive takes a list of expressions, separated by commas, and must contain at least one expression. Each expression must evaluate to a number between -128 and 255 since it is represented by 8 bits.

In order to better understand the use of `.DB`, consider data structures in high level languages that are initialized, e.g., an array that is declared and initialized. The following example assembly code shows the use of `.DB` directives to place arrays of constants and characters in Program Memory.

```
; AVR assembly code - Example use of .DB directive
consts:
      .DB   0, 255, 0b01010101, -128, $AA
text: .DB   "Hello World!"
```

In the first array labeled as `consts`, each of the five constant values occupies 8 bits and can be represented as decimal, binary, octal, or hexadecimal. In the second array labeled as `text`, each character within the quotes is represented as an 8-bit ASCII code.

4.6.3   `.DW` - Define constant word(s)

The `.DW` directive is similar to `.DB` except that it defines 16-bit constants in Program Memory. The syntax for `.DW` is shown below.

Syntax:
label: `.DW`   *expression_list*

The `.DW` directive takes a list of expressions, separated by commas, and must contain at least one expression. Each expression must evaluate to a number between -32768 and 65535 since it is represented by 16 bits.

The following example assembly code shows the use of `.DW` directives to place arrays of 16-bit constants in Program Memory.

```
; AVR assembly code - Example use of .DW directive
varlist:
      .DW 0, 0xffff, 0b1001110001010101, -32768, 65535
```

4.6.4   .BYTE - Reserve byte(s) to a variable

In contrast to .DB, the .BYTE directive reserves memory resources in Data
Memory. In order to refer to the reserved locations, a label should precede
the .BYTE directive. The syntax for .BYTE is shown below.

Syntax:
label: .BYTE   *expression_list*

The .BYTE directive takes an expression, which is the number of bytes to
be reserved. The allocated bytes are not initialized and instead they will be
loaded with values and/or values will be read in using I/O operations. The
use of .BYTE is akin to declaring variables and arrays in high-level languages.
The following example assembly code shows the use of .BYTE directives to
allocate the variable Var and the array Array in Data Memory.

```
; AVR assembly code - Example use of .BYTE directive
Var:  .BYTE 1
Array:
      .BYTE array_size
```

The label Var represents the identifier for a variable of one byte and the
array Array consists of array_size bytes. Once allocated, they can be
referenced by their labels. We will see how this is done later using HIGH()
and LOW() functions discussed in Section 4.7.3.

4.6.5   .CSEG and .DSEG - Code and Data Segment

The .CSEG and .DSEG directives define the start of a *Code Segment* and
a *Data Segment*, respectively. An assembly file can contain multiple Code
Segments and Data Segments. When a program is assembled, the multiple
Code Segments and Data Segments are concatenated into one large Code
Segment and Data Segment and placed in the Program Memory and Data

Memory, respectively. Therefore, these directives can be used to intermix code sections and data sections within a program. The syntaxes for `.CSEG` and `.DSEG` are shown below:

Syntax:
`.CSEG`

Syntax:
`.DSEG`

The example assembly code below shows how `.CSEG` and `.CSEG` directives are used.

```
; AVR assembly code - Example use of .CSEG and .DSEG directives
      .DSEG                ; Start of Data Segment
vartab:
      .BYTE 4              ; Reserve 4 bytes in SRAM
      .CSEG
const:
      .DW   2              ; Write 0x0002 in Program Memory
      MOV R1, R0           ; Do something
```

The above example code uses `.DSEG` to reserve 4 bytes in the Data Memory and `.CSEG` to define a 16-bit constant in the Program Memory. The `.ORG` directive can also be used together with `.CSEG` and `.DSEG` to place segments in specific locations of Program Memory and Data Memory, respectively. Note that the default segment type is Code. Therefore, unless words need to be reserved in Data Memory `.CSEG` is not necessary. In the example above, `.DSEG` was used to allocate words in Data Memory, and thus, `.CSEG` was included to indicate that what follows is code.

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

The `.DEF` directive associates symbolic names with registers. This allows programmers to refer to registers with familiar symbolic names rather than register numbers. The following shows the syntax for `.DEF`.

Syntax:
`.DEF`   *symbol = register*

The following example assembly code shows how registers `R16` and `R0` can be assigned symbolic names `TEMP` and `IOR`, respectively, and used in the program.

```
; AVR assembly code - Example use of symbolic names
.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
```

`.DEF` is useful for a couple of other reasons. First, a register can be assigned several different symbolic names. Second, a symbolic name can be redefined by simply changing its `.DEF` definition at the beginning of the program rather than going through the entire program to change all the register assignments.

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

The `.EQU` directive assigns a value to a label. As we saw before, 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 a memory location in Data Memory, or any constant. This label can then be used anywhere in the program. A label assigned to a value by `.EQU` is a constant and cannot be changed or redefined. The following shows the syntax of `.EQU`.

Syntax:
`.EQU`   *label = expression*

The following example assembly code 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.

```
; AVR assembly code - Example use of symbolic names
.EQU  IO_OFFSET = $23
.EQU  PORTA = IO_OFFSET + 2
      CLR   R2                ; Clear R2
      OUT   PORTA, R2         ; Write to Port A
```

4.6.8 `.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 shows the syntax for `.INCLUDE`.

Syntax:
`.INCLUDE`   *filename*

The following example assembly code shows the use of the `iodefs.asm` file.

```
; AVR assembly code - Example use of .include directive
.INCLUDE   "iodefs.asm"; Include I/O definitions
     IN   R0, SREG         ; Read status register
```

In this code, the `IN` instruction uses `SREG`, which is defined in the `iodefs.asm` file as

```
.EQU  SREG = $3F       ; Status register
```

Thus, a programmer can use the more familiar symbolic name `SREG` rather than the cryptic address `$3F`.

4.7 Expressions

As we saw in the last few code examples, the AVR assembler supports *expressions*. Expressions can consist of operands, operators, and functions. These are discussed in the following subsections.

4.7.1 Operands

The following *operands* can be used in an expression:
- Labels that define instruction locations and reserved memory locations.
- Constants defined by the `.EQU` directive.
- Integer constants that can be given in different formats, including
  - Decimal (default): 10, 255
  - Hexadecimal (two notations): 0x0a, $0a, 0xff, $ff
  - Binary: 0b00001010, 0b11111111
- PC that defines the current instruction execution.

4.7.2   Operators

The AVR assembler supports a number of *operators* shown in Table 4.21. These operators can be commonly associated with C/C++ operators. Note that these operations are done only during assmbly and are not used in place of AVR Instructions.

Table 4.21: 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

The following example assembly code illustrates the use of a combination of Shift Left (`<<`) and Bitwise OR (`|`) operators:

```
; AVR assembly code - Example use of operators
.EQU  RXEN1 = 4
.EQU  TXEN1 = 3
.DEF  mpr = R16
      ...
      LDI   mpr, (1<<RXEN1|1<<TXEN1)
```

The `LDI` instruction takes 1 (i.e., 0b00000001) and shifts it left by 4 bits and 3 bits, which generate 0b00010000 and 0b00001000, respectively, and

then performs a logical OR on the two values resulting in 0b00011000. After preprocessing the expression, the `LDI` instruction becomes equivalent to

```
LDI   R16, 0b00011000
```

4.7.3   Functions

Functions can be used to return a particular portion of the result of an expression. Table 4.22 lists all the functions provided by the AVR assembler.

Table 4.22: Functions.

Function	Description
`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)

The two most commonly used functions are `HIGH()` and `LOW()`. As the names suggest, `HIGH()` and `LOW()` functions extract the high and low byte, respectively, of an expression. These functions used together are particularly useful for initializing an address register X, Y, or Z. The following example assembly code shows how X-register can be set to point to the first element of array `Array`.

```
; AVR assembly code - Pointer initialization
.DSEG
Array:
      .BYTE array_size
.CSEG
      LDI   R27, HIGH(Array)  ; Load high byte address of Array to R27
      LDI   R26, LOW(Array)   ; Load low byte address of Array to R26
      LD    R7, X
```

As we saw in the previous discussion with the `.BYTE` directive in Section 4.6.4, the label `Array` is a 16-bit address that points to the the first 8-bit element in the array. The function `HIGH(ARRAY)` extracts the upper byte

of the address and loads it into R27, which is the upper byte of X-register
(see Figure 4.4). This process is repeated with `LOW(ARRAY)`, where the lower
byte of the address is loaded into the lower byte of X-register. Afterwards,
X-register can be used to access any element in the array using indirect,
indirect with displacement, indirect with pre-decrement, and indirect with
post-increment addressing modes (see Table 4.2).

4.8   Assembly Coding Techniques

For many of you, this may be the first time you are writing an assembly
program. This section presents general guidelines to produce well-structured
codes that can be easily understood and debugged. Section 4.8.1 starts with
a discussion on good assembly code structure. This will be followed by
the usefulness of having include files to simplify assembly programming in
Section 4.8.2.

4.8.1   Code Structure

It is important to create and maintain a consistent *code structure* through-
out a program. Assembly language in general can be very confusing, and
spending several hours trying to find a specific problem area within a piece
of code can become quite frustrating. A well-structured program will reduce
this confusion and make the program much more 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, i.e., where certain parts of
the program are located, how an instruction looks within a line, etc. There
are several ways to write out a code on 'paper', but the most important
part 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 lead to confusion and make the code difficult
to understand.

   The most common style for writing assembly program is to use the *four-
column method*. This is illustrated in Figure 4.28. If this style is used
consistently throughout the program, the program should consist of four
columns. A column is usually separated with one or two tabs depending on
data string length. The first column contains assembler directives and la-
bels. If a label is longer than one tab length, then the instruction mnemonic
goes on the next line. Some directives, e.g., `.include`, can span beyond a
single tab. In this case, its parameter definition can be placed in the third

```
********************************************************************
;* AVR assembly code - XOR Block of Data
;* This code segment will XOR all the bytes of data between
;* the two address ranges.
;********************************************************************
Col 1 Col 2         Col 3                               Col 4
Directive  Parameter                               Comment
.include "m128def.inc"       ; Include definition file

         Parameter
.def tmp = r15               ; temp register
.def xor = r6                ; xor register
.equ addr1 = $1500           ; Beginning address of data
.equ addr2 = $2000           ; Ending address of data


.org $0000                   ; Set the program starting address
         Mnemonic
Label    Parameters
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:
     ld  tmp, X+             ; Load data into tmp
     eor xor, tmp            ; XOR tmp with xor register
     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



      →│ →│                   →│

     Tab   Tab               Tab
```

Figure 4.28: Illustration of line formatting rules.

column. The second column contains directive parameters and instruction mnemonics. The third column contains instruction parameters, and they start one tab from instruction mnemonics. Finally, the fourth column contains comments. Since it is common for instruction parameters to exceed one, two, or even three tabs, the comment column can start several tabs from the parameter column. Although there are no specific guidelines on how to place comments, aligning them results in more readable code.

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. By putting forth the effort and paying attention to code placement, you, and others that read

Table 4.23: 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 5.3.1 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 are any additional source code includes, they will go last.

your code, will appreciate the efficiency with which you are able to locate and debug problems in your program. Table 4.23 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.

4.8.2   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, but its location in memory may not be the same across different processors. Thus, a definition file can be used to define common names for I/O registers, such as `SREG` and `SPH`, so that programmers do not have to 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 Data Memory (`RAMEND`). See Appendix D for a complete listing of I/O register and bit definitions.

4.9 Mapping Between Assembly and High-Level Language

Now that we have discussed AVR assembly instructions and directives, and how to write well-structured codes, we are ready to look at more involved assembly codes. This section presents assembly codes required to implement typical high-level programming constructs, which include basic *expressions*, *control-flow*, *subroutines*, *functions*, and *data structures*. The following subsections discuss each of these topics, with the exception of basic expressions, which are incorporated into explanations of other elements of assembly programming. Moreover, in order to make it easier for you to transition into assembly language programming, we will start with C/C++ examples and then discuss their equivalent assembly codes. This will allow you to see that assembly programming is not much different than high-level language programming. The main exception is that you have to be aware of the processor architecture and its capabilities, e.g., actual memory locations and how they are referenced, arithmetic and logic instructions available, memory size, etc., and thus a lot more work has to be performed.

4.9.1 Control Flow

Conditional branches and jumps are essential for implementing control-flow in a program. Therefore, this subsection discusses examples of how C/C++ control-flow expressions are coded in assembly.

IF and IF-ELSE Statements

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

```
if (expression)
    statement;
```

If *expression* is true, then *statement* is executed; otherwise, *statement* is skipped. This can be implemented in assembly as well. For example, consider the following C code.

```
/* C code - Example IF statement */
if (n >= 3)
{
    expr++;
```

```
    n = expr;
}
```

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

```
; AVR assembly code - Equivalent implementation of IF statement
.def  n = R16
.def  expr = R1
.equ  cmp = 3


...
      CPI   n, cmp            ; Compare value
IF:   BRGE  EXEC              ; If n >= 3 then branch to EXEC
      RJMP  NEXT              ; otherwise jump to NEXT
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 `BREQ` 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 `BRCS` instead of `BRGE` (see Table 4.14).

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 `BRLT`. This is shown below:

```
; AVR assembly code  - Alternative implementation of IF statement
.def  n = R16
.def  expr = R1
.equ  cmp = 3
...
      CPI   n, cmp            ; Compare value
IF:   BRLT  NEXT             ; If n >= 3 is false then branch to EXEC
      INC   expr             ; Increment expr
      MOV   n, expr          ; Set n = expr
NEXT: ...                    ; Continue on with the code
```

The above assembly code uses one less line of code, and is also easier to read and understand. You may wonder how much of a speed improvement is achieved by removing one instruction. 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 `BRCC` instead of `BRLT` (see Table 4.14).

IF-ELSE statement is very similar to IF statement, except it has an additional ELSE part. Consider the following example C code.

```
/* C code - Example IF-ELSE statement */
if (n == 5)
     expr++;
else
     n = expr;
```

The equivalent AVR assembly code is shown below:

```
; AVR assembly code - Assembly equivalent of IF-ELSE statement
.def  n = r16
.def  expr = r1
.equ  cmp = 5
...
      CPI   n, cmp      ; Compare values
IF:   BRNE  ELSE        ; Goto ELSE 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 conditional `BRNE` instead of `BREQ`, which makes the code more compact and run faster.

Loops

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

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

```
/* C code - FOR loop */
for (n = 0; n < 10; n++)
     sum += n;
```

The following is the equivalent assembly code:

```
; AVR assembly code - Assembly equivalent of FOR loop
.def  n = r16
.def  sum = r3
.equ  limit = 10
...
      CLR   n           ; Initialize n to 0
FOR:  ADD   sum, n      ; sum += n
      INC   n           ; increment n
      CPI   n, limit    ; compare n and 10
      BRLT  FOR         ; repeat loop if n does not equal 10
NEXT: ...               ; rest of code
```

This code uses `CPI n, limit` to check if the end of the loop has been reached. As long as `n` is less than `10`, the loop continues to iterate.

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

```
while (expression)
     statement;
```

First *expression* is evaluated, if it is true, then *statement* is executed, and control-flow goes 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 the *next statement*. The following C code illustrates this example:

```
/* C code - WHILE loop */
while (n < 10) {
      sum += n;
      n++;
}
```

The equivalent AVR assembly code is shown below:

```
; AVR assembly code - Assembly equivalent of WHILE loop
.def  n = r16
.def  sum = r3
.equ  limit = 10
...
WHILE:
```

```
      CPI   n, limit     ; Compare n with limit
      BRGE  NEXT          ; If n >= limit, goto NEXT
      ADD   sum, n        ; sum += n
      INC   n             ; n++
      RJMP  WHILE         ; Go back to beginning of WHILE loop
NEXT: ...                 ; Continue on with code
```

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

```
/* C code - DO loop */
do {
     sum += n;
     n--;
} while (n > 0);
```

The assembly code for the DO statement is also very similar to the WHILE statement. This is shown below:

```
; AVR assembly code  - Assembly equivalent of DO loop
.def  n = r16
.def  sum = r3
...
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 executed before the `BRNE` instruction, and thus the `CPI` instruction is not needed.

4.9.2 Subroutine

In assembly programming, a *subroutine* is a piece of code within a large program that performs a specific task. A subroutine can generally be thought as a "reusable code", which is any segment of code that can be used over and over again throughout the program, and allows a programmer to drastically reduce the size and complexity of a code. In some ways, subroutines are similar to macros but they are much more flexible and powerful. There are three major differences between a macro and a subroutine. First, a macro is supported by the assembler and called by simply using its name, while
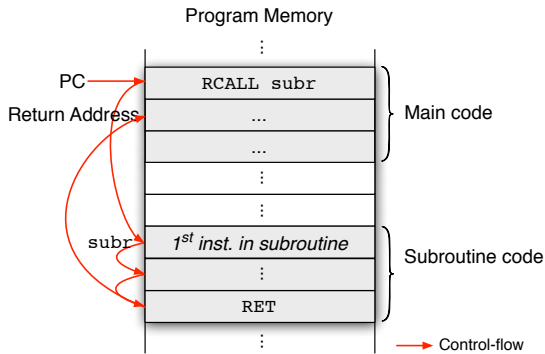
Figure 4.29: Control-flow for subroutine call and return.

a subroutine is called using a subroutine call instruction. Second, only a single copy of a subroutine code exists, while a macro requires its body to be substituted into the code each time it is referenced. Thus, very long macros that are used many times in a program will significantly increase the code size. Finally, subroutines can be used to pass parameters to implement function calls as in high-level languages, while macros cannot.

As discussed in Subsection 4.4.3, a subroutine is implemented using the `CALL`, `RCALL`, or `ICALL` instruction and is paired with the `RET` instruction to return the control flow to the address of the instruction after the subroutine call, referred to as the *return address*. A subroutine is preceded by a label that signifies its name. Figure 4.29 illustrates the flow of control for subroutine call and return.

When `RCALL` is executed, the processor first pushes the return address, which is PC+1 (since `RCALL` is a 16-bit instruction) onto the stack. This is an important concept since it means that *the stack must be initialized* before subroutines can be used. The `RCALL` instruction will then jump to the address specified by the label `subr`. The next instruction to be executed will be the first instruction in the subroutine. Upon completion of the subroutine, the `RET` instruction pops the return address from the stack and loads it into the PC. Thus, the next instruction to be executed will be the instruction after the `RCALL` instruction.

It is important to keep track of what is pushed to or popped from the stack. If data is not popped correctly within a subroutine, `RET` can pop the wrong value as the return address and thus the program will not function correctly. In addition, a subroutine must not be exited via another jump instruction other than `RET`. Doing so will cause the data in the stack to never

be popped and thus the state of the stack will be incorrect.

As discussed before, the first task before using subroutines is to initialize the stack. This can be done using the following four lines of code during the initialization phase of the program:

```
; AVR assembly code - Stack initialization
.include "m128def.inc"
...
INIT: ldi   r16, low(RAMEND)  ; Load low byte of RAMEND addr
      out   SPL, r16          ; Set SP Low register
      ldi   r16, high(RAMEND) ; Load high byte of RAMEND addr
      out   SPH, r16          ; Set SP High register
```

The SP register is set to the last location in the Data Memory indicated by RAMEND, which is defined in the `m128def.inc` include file and its value depends on the Data Memory size of the processor you are working with. For the ATmega128 processor, RAMEND is $10FF (see Figure 4.3). This is done by extracting the low and high bytes of the RAMEND address using the functions `low()` and `high()` and moving them to the low and high bytes of the SP indicated by `SPL` and `SPH`, respectively. Note that moving the lower and higher bytes of the RAMEND address is done using the `OUT` instruction because `SPL` and `SPH` registers are located in the 64 I/O register space.

4.9.3   Function

Under Construction!!!

4.10   Anatomy of an Assembly Program

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

Consider an example program shown below, which adds eight numbers stored in Program Memory.

```
; AVR assembly code - Adding 8 numbers
.include    "m128def.inc"
.org  $0000
      RJMP  Init_ptr
.org  $000B
```

```
Init_ptr:
      LDI    ZL, low(Nums<<1)   ; Load loop count
      LDI    ZH, high(Nums<<1)  ; Z points to 12
Main: LDI    R16, 8             ;
      CLR    R1                 ; Clear accumulator R1:R0
      CLR    R0                 ;
Loop: LPM    R2,Z+              ; Load data to R2 and post-inc ptr
      ADD    R0, R2             ; Add R2 to R0(L)
      BRCC   Skip               ; No carry, skip next step
      INC    R1                 ; Add carry R1(H)
Skip: DEC    R16                ; Decrement loop count
      BRNE   Loop               ; If not done loop
Done: JMP    Done               ; Done.  Loop forever.
Nums: .DB    12, 24, 0x3F, 255, 0b00001111, 2, 21, 6
```

Figure 4.30 shows the layout of the machine code represented in hexadecimal for the above assembly code in Program Memory. One of the first things to note is that upper and lower bytes of the 16-bit instruction format are flipped. That is, unlike how we view the instruction format from MSB to LSB, or from most significant hexadecimal digit to least significant hexadecimal digit, the actual order in Program Memory is reversed. This is because AVR uses *little-endian*, where the lower byte of instruction comes first.

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

The very first instruction of the machine code is RJMP, which jumps to label Init_ptr upon reset. RJMP is located at address $0000 defined by the directive .org $0000. The target address of RJMP is Init_ptr, which is located at address $000B. Since RJMP is a PC-relative jump, its 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 included in the instruction.

The second section of the code involves pointer initialization (which starts at address location $000B) to access data contained in the last section of the machine code. The eight numbers to be accumulated are stored starting at the label Nums, which is at Program Memory addresses $0017. In order to access these locations, Z-register is initialized to point to Nums. This is achieved using the following two instructions:

```
      LDI    ZL, low(Nums<<1)   ;
      LDI    ZH, high(Nums<<1)  ; Z points to 12
```
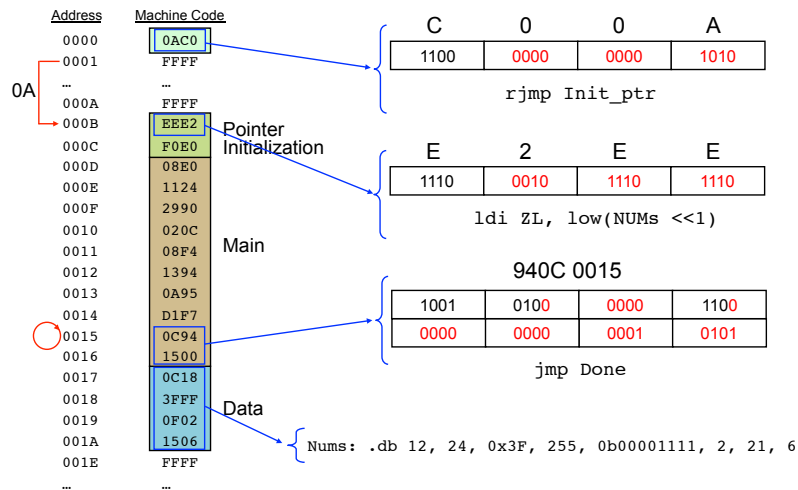
Figure 4.30: Contents of the Program Memory for the program that adds 8 numbers.

As discussed in Section 4.7.3, `low()` returns the low-byte of the address generated by the expression `Nums<<1`. The operator `<<` means logical shift left by one bit. Thus, `Nums<<1` takes the address of `Nums`, which is $0017 ($0000\ 0000\ 0001\ 0111_2$), and shifts it left by one bit resulting in $002E ($0000\ 0000\ 0010\ 1110_2$). The first `LDI` instruction moves the low-byte $2E to the low-byte of the Z-register. The same applies to the second `LDI` instruction except the high-byte of the address $00 is loaded into the high-byte of the Z-register. Thus, after the execution of these two instructions, Z-register is initialized to point to the first two values, and LSB is used to distinguish the first (left) byte from the second (right) byte. This is shown in Figure 4.31.

The third section of the code represents the main program, which starts at location $000D. The `LDI` instruction initializes the loop count and the two `CLR` instructions clear `R1` and `R0`, which serve as the accumulator. The `lpm R2, Z+` instruction is used to load each data element and then post-increment Z-register to point to the next 8-bit data. The data is then added to the lower byte of the accumulator (i.e., `R0`). If there is no carry out, i.e., no overflow, `BRCC` skips the next instruction; otherwise, the upper byte of the accumulator (i.e., `R1`) is incremented. Afterwards, the loop count is decremented, and if it is not zero, the program loops back to the label `Loop`.
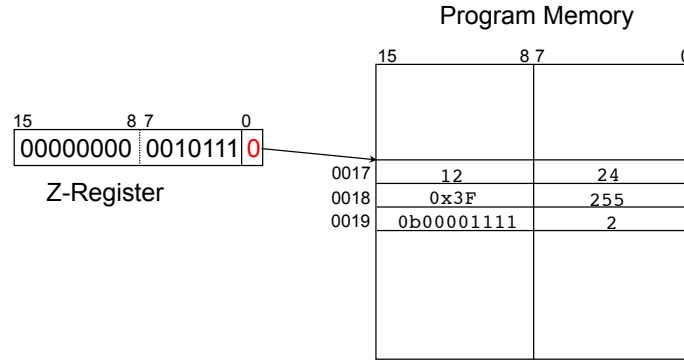
Program Memory

Figure 4.31: `LPM` instruction.

This repeats until the loop count reaches zero. Finally, the program ends with the 32-bit `JMP` instruction that loops back to itself. The `JMP` instruction uses Direct Program Memory Addressing, and thus, the second 16-bit of the instruction holds the address of the label `Done` (i.e., $0015). Note that `JMP` has six additional bits for address in the first 16-bit of the instruction. These bits are zeros, and leave room for possible expansion to the 64K Program Memory address space.

Chapter 5

Atmel's AVR 8-bit Microcontroller: Part 2 - Input/Output

Contents

5.1 Introduction

Input/Output (I/O) refers to the communication or interface between a processor and external devices. High-end microprocessors for laptops, desktops, and servers support complex but familiar I/O devices, such as keyboard, mouse, display, printer, network interface, hard disk drive, etc., which in turn have microcontrollers in them. In contrast, microcontrollers for embedded systems typically have simple I/O interfaces to communicate with relatively simple devices.

Figure 5.1: Block Diagram of ATmega128.

Microcontrollers handle different types of I/O depending on the application. The simplest form of I/O is to send or receive a signal through one of the port pins, for example, to turn on an LED (output) or to detect when a switch (input) has been depressed (discussed in Section 5.2). Other typical applications involve *sensors* that monitor some physical conditions, such as temperature, motion, pressure, light, sound, etc., as input signals, which can then be received and processed by a microcontroller.

This chapter discusses I/O capabilities of microcontrollers, in particular, AVR microcontrollers. Figure 5.1 shows a block diagram of the AVR ATmega128 microcontroller, which has a set of features to handle a variety of I/O functionalities. These included ports, timers/counter, analog-to-digital converter (ADC), Universal Synchronous Asynchronous Receiver/Transmitter (USART), Serial Peripheral Interface (SPI), Two-Wire Interface (TWI),
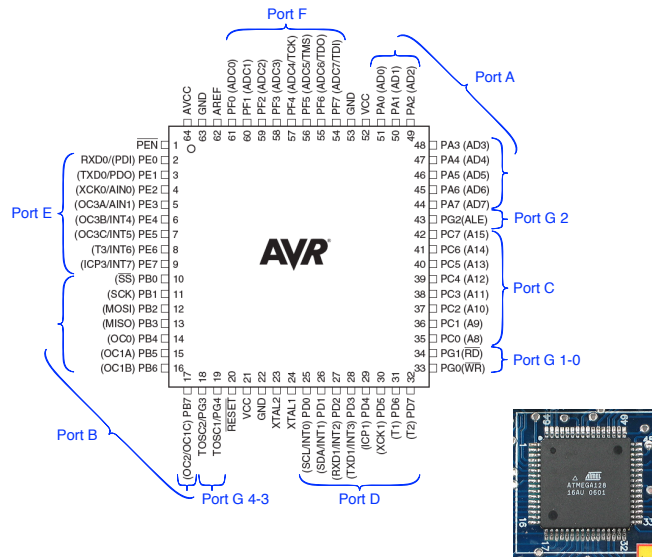
Figure 5.2: I/O port pins

and Analog Comparator. In addition, an Interrupt Unit allows these I/O features to notify the processor that services are needed. The following Sections discuss each of these I/O features.

5.2 I/O Ports

I/O ports are general-purpose pins on microcontrollers that can be used to communicate with and control external devices. These pins can be either input or output and configured by the user at run time.

5.2.1 AVR I/O Ports

I/O ports for the ATmega128 chip are shown in Figure 5.2, which has six 8-bit ports and one 5-bit port for a total of 53 I/O lines. These ports are referred to as Port A-G. Each port can be used to send/receive 8-bit data or each I/O line or *pin* can be configured to send/receive a single bit at a time.

There are three I/O registers associated with each port: *Port x Data register* (PORTx), *Port x Input Pins* (PINx), and *Port x Data Direction Register* (DDRx), where the suffix 'x' represents the port name A-G. PORTx

Port x Data Register (PORTx)

7	6	5	4	3	2	1	0	
PORTx7	PORTx6	PORTx5	PORTx4	PORTx3	PORTx2	PORTx1	PORTx0	PORTx
R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	

PORTxn = 1 and DDxn = 0 Activates Pull-Up Register

Port x Input Pins (PINx)

7	6	5	4	3	2	1	0	
PINx7	PINx6	PINx5	PINx4	PINx3	PINx2	PINx1	PINx0	PINx
R (N/A)	R (N/A)	R (N/A)	R (N/A)	R (N/A)	R (N/A)	R (N/A)	R (N/A)	

Port x Data Direction Register (DDRx)

7	6	5	4	3	2	1	0	
DDx7	DDx6	DDx5	DDx4	DDx3	DDx2	DDx1	DDx0	DDRx
R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	

DDxn = 0   Input
DDxn = 1   Output

Figure 5.3: PORTx, PINx, and DDRx I/O registers.

is used to output data onto the port pins, while PINx is used to input data from the port pins. Since each I/O pin can be read from or written to, DDRx is used to control whether these lines function as input or output. PORTx, PINx, 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 (see Appendix C).

Figure 5.3 shows the PORTx, PINx, and DDRx I/O registers. For each bit, 'R/W' indicates that the bit can be read as well as written, 'R' indicates read only, and the number in parentheses indicates the initial value after reset. Since PINx is read-only, writing a value to it has no effect. In addition, the initial value of PINx is 'N/A' because its value depends on the value provided by the external device connected to the port at reset.

Figure 5.4 shows a simplified diagram of how each bit of PORTx, PINx, and DDRx are interfaced to their respective I/O pin. Note that this diagram does not include the Schmitt trigger, bidirectional tri-state buffers, and other control signals required for its proper operation. Each I/O pin, Pxn, is connected to a pair of latches PORTxn and PINxn, and a control latch DDxn, where the suffix 'n' represents the bit position (7-0) within a port 'x'. PORTxn is used to send data to the pin and PINxn is used to receive data from the pin. The PORTxn latch has an additional purpose when the pin is configured as input: Writing a 1 to it activates a *pull-up resistor*. The purpose of a pull-up resistor will be discussed in Section 5.2.2. DDRxn
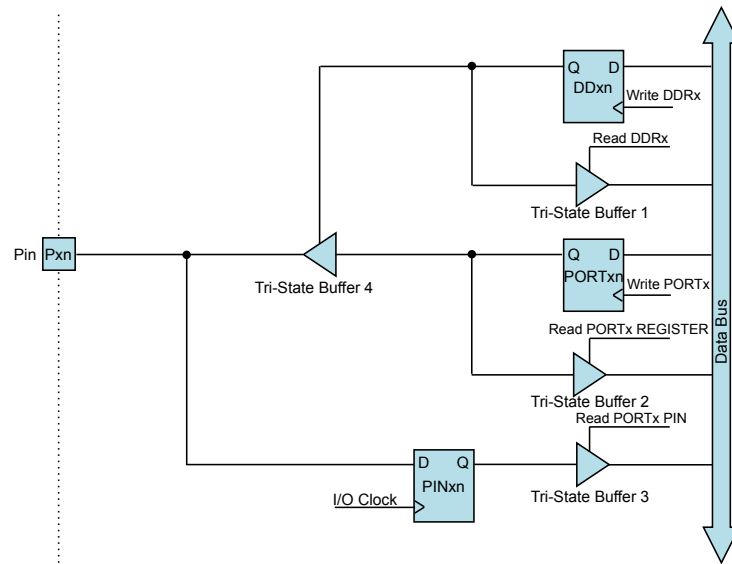
Figure 5.4: A simplified diagram of a single pin of a port.

controls the direction of the pin, which can be either input or output. Internal control signals for the three Tri-State Buffers 1~3 control when the data in PORTxn, PINxn, and DDRxn appear on the Data Bus.

Fig. 5.5 illustrates the configuration of an I/O pin for either input or output. Fig. 5.5(a) shows how the pin can be configured for input. This is done by writing a 0 on DDRxn, which causes the output of Tri-state Buffer 4 between Pxn and PORTxn to be in high-impedance state, i.e., open-circuit, allowing the signal on Pxn to be latched onto PINxn. Fig. 5.5(b) shows how the pin can be configured for output. Writing a 1 into DDRxn enables Tri-state Buffer 4, which provides a direct connection between PORTxn and Pxn.

The code below shows how Port A can be configured for input and output.

```
; AVR Assembly code - Configuring Port A
.include "m128def.inc"
; Setting Port A 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
```
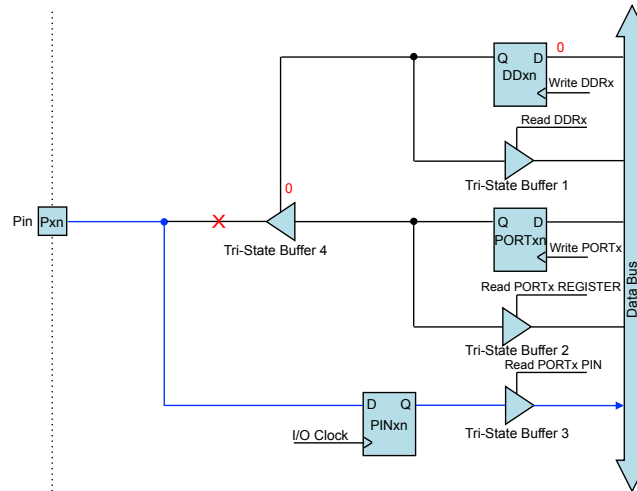
(a) Configuring for input.
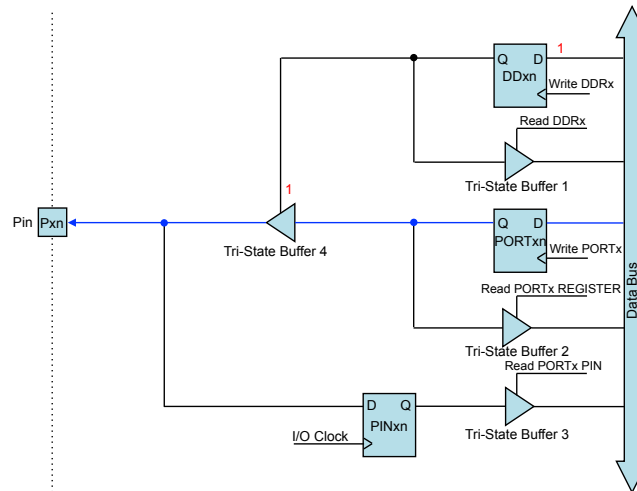
(b) Configuring for output.

Figure 5.5: Reading and writing to a port.

```
; Setting Port A for output
      ldi    r16, $ff
      out    DDRA, r16    ; Write 1s to DDRA setup for output
      ...
```
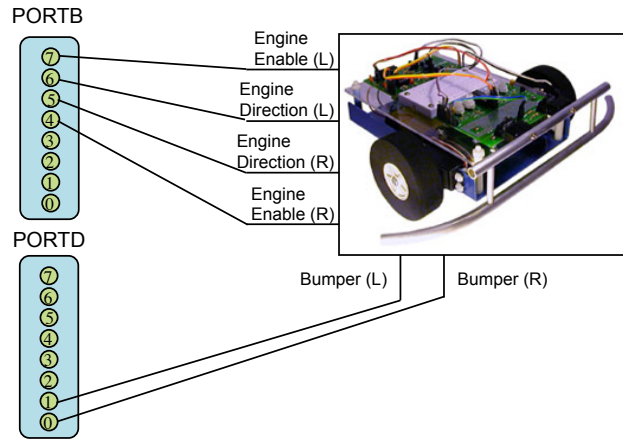
Figure 5.6: Connection of motor control and bumper switches to PORTB and PORTD in TekBot.

```
out   PORTA, r16  ; Then use this instruction to write to PORTA
```

The combination of `ldi` (*Load immediate*) and `out` (*Out port*) instructions can be used to write either 0's or 1's to DDRA to configure Port A to input or output. For input, the AVR I/O instruction `in` (*In port*) instruction can be used to move the content of `PINA` to `R16`. For output, the AVR I/O instruction `out` (*Out port*) can be used to move the content of `r16` to `DDRA`. Note that both `PINA` and `PORTA` are I/O registers in the 64 I/O register address space and their addresses are defined in the `.m128def.inc` include file.

5.2.2   I/O Operations for TekBots

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 using a *TekBot*, which is an AVR-microcontroller-based programmable robot developed at the 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 in the electrical and computer engineering field.

Figure 5.6 shows a picture of a TekBot, which is driven by a pair of motorized wheels and is controlled by a board with an AVR ATmega128 microcontroller. It also has two switches (left and right) on the bumper to

detect bumps. The right switch is connected to PORTD pin 0, while the left switch is connected to PORTD pin 1. Detection of a bump initiates a routine to turn the TekBot around. Right/Left engine enable is connected to PORTB pins 4/7, where 0 turns on the motor and 1 turns off the motor. Right/Left engine direction is connected to PORTB pins 5/6, where 1 moves TekBot forward and 0 moves TekBot backward.

The code shown in Figure 5.7 controls the basic functionalities of a Tek-Bot, which is to move forward until it bumps an object and then move backward for one second then turn left or right for one second depending on which switch is hit, and then move forward again. 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 the `HitLeft` subroutine, which is similar to the `HitRight` subroutine, has been omitted to save space.

The code first starts with a series of `.def` and `.equ` directives that assign symbolic names to registers (e.g., `mpr`, `waitcnt`, `ilcnt`, and `olcnt`) and labels to values (e.g., `WTime`, `WskrR`, `WskrL`, etc.) to make it easier to keep track of the code. The last few `.equ` directives also assign labels to values, but these values are evaluated using expressions. For example, consider the following definition:

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

Interrupt Vectors

Following the register definitions and constants, the very first part of the actual program code contains two `.org` directives that indicate that (1) `rjmp INIT` will be the first instruction executed when the TekBot is turned on and (2) the rest of the code will be placed 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 jumps to label `INIT` that contains the code for

```
; AVR Assembly code - Basic operations of Tekbot
.include "m128def.inc"
;*************************************************************
;*    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 = 0               ; Right Whisker Input Bit
.equ  WskrL = 1               ; 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 Forward Command
.equ  MovBck = $00            ; Move Backward 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            ; 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    mpr, (1<<EngEnL)|(1<<EngEnR)|(1<<EngDirR)|(1<<EngDirL)
      out    DDRB, mpr        ; Set Port B Directional Register for output
      ; Initialize Port D for inputs
      ldi    mpr, (0<<WskrL)|(0<<WskrR)
      out    DDRD, mpr        ; Set Port D Directional Register for input
      ldi    mpr, (1<<WskrL)|(1<<WskrR)
      out    PORTD, mpr       ; Activate pull-up resistors
      ; Initialize TekBot Forward Movement
      ldi    mpr, MovFwd      ; Load Move Forward Command
      out    PORTB, mpr       ; Send command to motors
```

Figure 5.7: AVR Assembly Code for TekBot Movement.

```
;----------------------------------------------------------------
; Main Program
;----------------------------------------------------------------
MAIN:
      in    mpr, PIND        ; Get bumper switch input from Port D
      com   mpr              ; Complement since bumpers are active low
      andi  mpr, (1<<WskrL)|(1<<WskrR) ; Mask out other bits
      cpi   mpr, (1<<WskrR)  ; Check for Right Whisker input
      brne  NEXT             ; Continue with next check
      rcall HitRight         ; Call the subroutine HitRight
      rjmp  MAIN             ; Continue with program
NEXT:
      cpi   mpr, (1<<WskrL)  ; Check for Left Whisker input
      brne  MAIN             ; No Whisker input, continue
      rcall HitLeft          ; Call subroutine HitLeft
      rjmp  MAIN             ; Continue through main
;----------------------------------------------------------------
; Sub:     HitRight
; Desc:    Functionality for TekBot when the right switch is triggered.
;----------------------------------------------------------------
HitRight:
      ; 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
      ; Move Forward again
      ldi   mpr, MovFwd      ; Load Move Forward command
      out   PORTB, mpr       ; Send command to port
      ret                    ; Return from subroutine
;----------------------------------------------------------------
; Sub:     Wait
; Desc:    A wait loop that waits approx. waitcnt*10ms.
;----------------------------------------------------------------
Wait:
OLoop:
      ldi   olcnt, 224       ; (1) Load middle-loop count
MLoop:
      ldi   ilcnt, 237       ; (1) Load inner-loop count
ILoop:
      dec   ilcnt            ; (1) Decrement inner-loop count
      brne  Iloop            ; (2/1) Continue inner-loop
      dec   olcnt            ; (1) Decrement middle-loop count
      brne  Mloop            ; (2/1) Continue middle-loop
      dec   waitcnt          ; (1) Decrement outer-loop count
      brne  OLoop            ; (2/1) Continue outer-loop
      ret                    ; Return from subroutine
```

Figure 5.7: AVR Assembly Code for TekBot Movement (cont.).

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*, and thus placing any code in this address space may cause unwanted behavior. Therefore, `.org $0046` causes the rest of the program to be placed after the interrupt vectors.

Program Initialization

When the TekBot, and thus the processor, is turned on, the port pins that are connected to the wheels and the left and right switches for the bumper have to be appropriately configured. As shown in Figure 5.6, 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 configured as output to control the motors. 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 as well as store and restore register values. This part of the code has been omitted to simplify the discussion at hand, but will be discussed in more detailed in Section 5.3. For now, let us assume that the stack has been set up.

The first two instructions set the bits 7-4 of the `DDRB` register to 1's ensuring that the corresponding bits in `PORTB` will function as outputs. The same is true for initializing `PORTD`, except it will be set for input. However, detecting when the bumper switches are triggered requires the use of pull-up resistors. This is because a bumper switch is nothing more than a *passive switch*. That is, when the bumper is hit, it turns on the switch; otherwise, the switch is off. However, turning the switch on and off need to be translated into 0 and 1 so that the processor can detect this on its I/O pins.

Figure 5.8 shows how a pull-up resistor is used to latch 0 or 1 onto PINxn when the switch is turned on or off, respectively. Figure 5.8(a) shows that when the switch 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 switch is triggered and it becomes on as shown in Figure 5.8(b), the voltage is pulled down to ground, which is considered 'low'. Therefore, when the switch is triggered, 0 is latched onto PINxn; otherwise, 1 is latched onto PINxn. The pull-up resistor is activated by writing a 1 onto PORTxn together with setting DDRxn to 1 for input. This causes the pass transistor located between the supply voltage and the pull-up resistor to be turned on.

Once the ports are configured, writing $60 or 0b01100000 to `PORTB` causes the TekBot to move forward. This is because the bits 6 and 5 that control
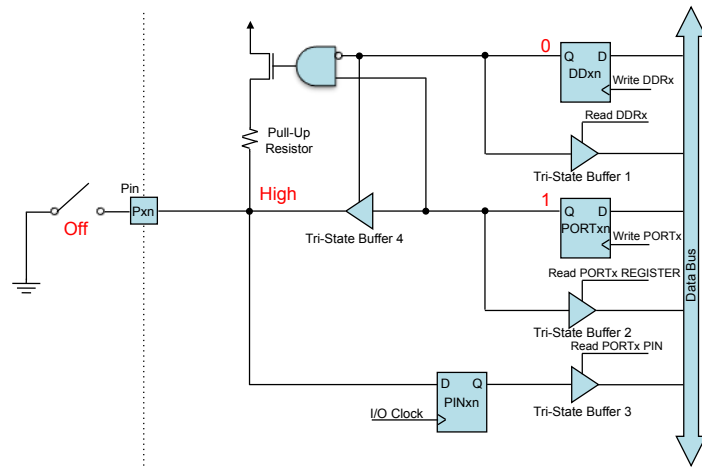
(a) High input.
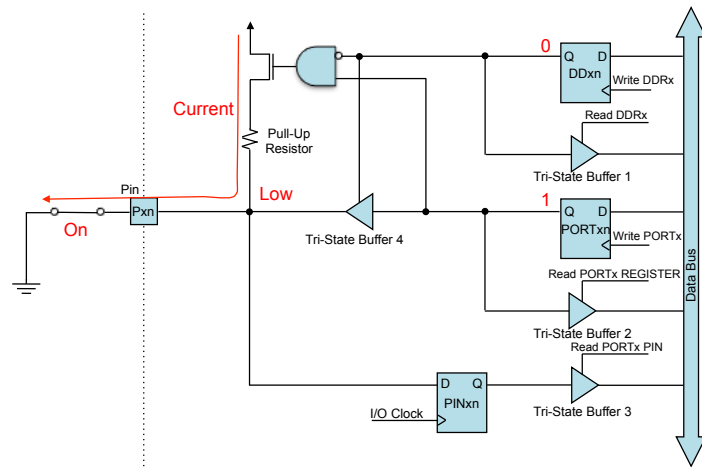
(b) Low input (triggered!).

Figure 5.8: Triggering of the bumper switches.

engine direction are set to 1's and 7th and 4th bits that enable the engines are set to 0's.

Main Program

The Main Program is simply a loop that checks whether the right or left bumper switch has been triggered. If a switch is hit, the corresponding PIND

latches, i.e., bit 0 and bit 1 for the right and left switches, respectively, will be cleared. Therefore, the Main Program checks the two switches in succession by (1) loading the content of PIND and complementing it, (2) masking out unrelated bits, and (3) comparing to see if the corresponding bit is zero, and if so, execute either the `HitRight` or `HitLeft` subroutine.

HitRight Subroutine

When the right switch 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., both bits 5 and 6 of PORTB are set to 0), and then setting the left wheel to move backward and the right wheel to move forward (i.e., bits 5 and 6 of PORTB are set to 1 and 0, respectively), and finally setting both wheels to move forward (i.e., both bits 5 and 6 of PORTB are set to 1). In addition, moving backwards and turning left each requires a duration of one second. This is achieved by calling the `Wait` subroutine.

Wait Subroutine

The `Wait` subroutine is coded to execute for approximately one second (0.99962 sec to be exact!). The code structure is a triple-nested loop where each iteration of the outer-most loop takes around 10 ms to execute. Thus, the value passed to the `r17` register is 100, which allows the triple-nested loop to execute for 100×10 ms = 1 sec. Now, how do we know each iteration of the outer-loop will take 10 ms to execute? The beauty of assembly programming is that the programmer knows (or can find out) the number of clock cycles each instruction takes to execute (see Appendix A for a complete listing of cycles required for AVR instructions). Moreover, the *clock cycle time* or clock period is known based on the clock frequency used by the processor. For example, the ATmega128 version we are using has a 16 MHz clock rate, which results in a clock cycle time of 62.5 ns.

   Now going back to our `Wait` subroutine, the number of cycles each instruction takes in the triple-nested loop is known (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 (the reason why this is the case will be discussed in Chapter 8). Thus, the time required to execute one iteration of the outer-loop, t_{oloop}, is given by the following equation:

$$t_{oloop} = ((((3 \times ilcnt) - 1 + 4) \times mlcnt) - 1 + 4) \times 62.5 ns \qquad (5.1)$$

The $(3 \times ilcnt) - 1$ portion of the equation represents the number of cycles required 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* iterations 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` instructions 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. Thus, with *mlcnt* equal 224 and *ilcnt* equal to 237, the number of cycles required for each iteration of the outer-loop, Cyc_{oloop}, is 159,935 clock cycles leading to a delay of 9.996 ms.

Therefore, the number of cycles required for the `Wait` subroutine, Cyc_{WAIT}, is given by Equation 5.2.

$$Cyc_{WAIT} = Cyc_{oloop} \times waitcnt - 1 + 7 \qquad (5.2)$$

where the -1 terms comes from the fact `brne OLoop` is not taken in the last iteration, 7 cycles are required to execute `rcall Wait` (3 cycles) and `ret` (4 cycles), and *waitcnt* is the value in `r7`. Finally, the time required to call the `Wait` subroutine, execute the subroutine, and then return is 15,993,906 cycles $\times 62.5 \ ns =$ 0.99962 sec.

You may wonder if there is a simpler way to wait for a second, and fortunately there is. In Section 5.4, we will discuss about *Timer/Counters* that can be used to implement the `Wait` subroutine as well a variety of other timing related functions. However, the main purpose of this discussion was to (1) show that assembly programmers have complete control of their code's timing and (2) illustrate nested loops.

5.3   Interrupts

In the code for TekBot movement shown in Figure 5.7, the loop in the Main Program constantly checks whether or not a switch is triggered. This method of checking for an external event is referred to as *busy waiting*. Unfortunately, a processor busy-waiting expends all its processing power waiting for events to occur. A better method is to use interrupts. An

interrupt is signaling of an unexpected event, in terms of timing, that causes a change in the normal program flow. 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. This is an important concept because it allows an I/O device to informed the processor when it needs to be serviced rather than the other way around and frees the processor to perform other tasks.

An external event can come in many forms. Some examples are
- I/O device is ready to send or receive data
- Universal Synchronous Asynchronous Receiver/Transmitter (USART) transmit buffer is empty.
- I/O device has input ready
- Key pressed on a keyboard
- A TekBot bumper switch is triggered
- An internal timer generates an interrupt periodically

Note that interrupts are different from *traps*, which are caused by either events that occur within the processor or by software. For this reason, traps are also called *software interrupts*. Some examples of events that generate 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.

This section discusses the concept of interrupt. Section 5.3.1 presents the interrupt facility provided by AVR microcontrollers. This is followed by an interrupt-based TekBot example.

5.3.1 AVR Interrupt Facility

AVR ATmega128 microcontrollers can handle 35 interrupt sources, which consists of
- Reset
- 8 External Interrupt Requests
- 4 Timer/Counters with Compare Match and Overflow
- ADC Conversion Complete
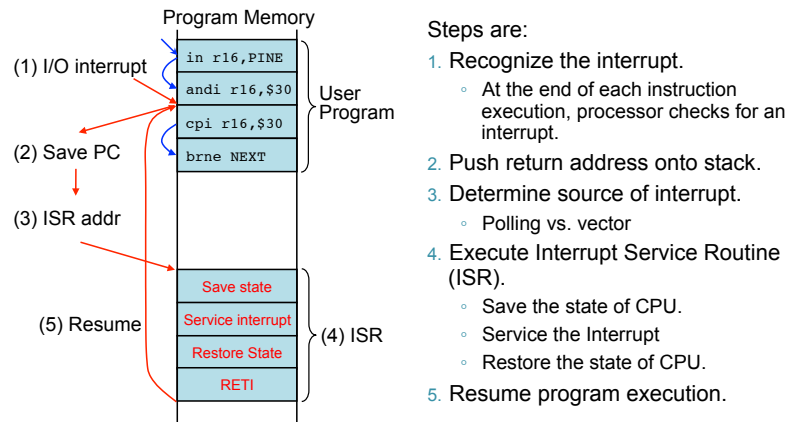
Figure 5.9: The process of handling interrupts.

- USART Tx/Rx complete
- SPI Serial Transfer Complete
- $\cdots$ and many more

Reset is the most fundamental form of interrupt that gets triggered when the processor is powered on. *External interrupts* are triggered on pins INT7-INT0 (see Figure 5.11), and thus up to eight interrupt sources can 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 10-bit binary values. *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). *Serial Peripheral Interface* (SPI) is another serial link that operates in a synchronous fashion.

Now that we have discussed the interrupt sources for the AVR microcontroller, we are ready to discuss how interrupts from these sources are handled. Figure 5.9 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 (i.e., PC+1), onto the stack. Next, the source of the interrupt is identified, which allows the appropriate *Interrupt Service Routine* (ISR) to be executed. The state

of the processor is also saved and restored before and after executing the ISR, respectively. After the ISR completes, the control flow 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

The AVR microcontroller's interrupt facility needs to be configured before it can be used. The first thing that needs to be done is to turn on the interrupts. This is done by setting the MSB of the SREG, called the *Global Interrupt Enable* (I-bit), shown in Figure 4.5. As the name suggests, the I-bit needs to be set to allow interrupts to be detected. As shown in Table 4.19, the I-bit can be directly set using SEI (*Set global interrupt flag*) or cleared using CLI (*Global interrupt disable*). In addition, the I-bit is automatically cleared after an interrupt is detected and set by the RETI (*Return from interrupt*) instruction. Therefore, the default behavior is to not allow other interrupts from occurring while the current interrupt is being serviced. This behavior can easily be changed by manually setting the I-bit and allowing other devices to interrupt the processor. Furthermore, different interrupts can be prioritized by allowing a higher priority device interrupt a lower priority device.

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. In other words, the PC is updated with a vector that corresponds to the interrupt source, which allows the control flow to be redirected to the corresponding ISR. Table 5.1 shows the *interrupt vector table* that defines how various interrupt sources are mapped to the different locations in the Program Memory. For example, RESET is mapped to the Program Memory location $0000 and has the highest priority. This is followed by the eight external interrupts (INT0-INT7). The rest of the vectors provide mapping for the other interrupt sources, and will be discussed in the latter part of this chapter.

Each vector is allocated with two Program Memory words (i.e., 32 bits), which allows for either a jump or subroutine call (either 16-bit or 32-bit version) to transfer the control flow to the corresponding ISR. Figure 5.10 shows an example code of how interrupt vectors may be set up. In this example, when the processor is reset, this interrupt is detected and the program counter is loaded with the address of its vector (i.e., $0000). This causes

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

the processor to jump to the ISR located at label `RESET`, which is usually an initialization routine to set up the microcontroller. Similarly, other interrupt sources can be set up by placing control flow transfer instructions in their respective interrupt vectors and providing dedicated ISRs. Note that `.ORG` directives in the code not only indicate the beginning addresses of interrupt vectors but are also necessary because `RJMP` instructions are 16-bit instructions. If we had used `JMP` instructions, which are 32-bit instructions, then `.ORG` directives can be omitted. However, its good practice to include them to clearly indicate the locations of the interrupt vectors.

```
; AVR Assembly Code - Setting up interrupt vectors
.ORG  $0000
      RJMP  RESET
.ORG  $0002
      RJMP  EXT_INT0     ; External Interrupt Request 0
.ORG  $0004
      RJMP  EXT_INT1     ; External Interrupt Request 1
      ...                ...
.ORG  $0010
      RJMP  EXT_INT7     ; External Interrupt Request 7
.ORG  $0012
      RJMP  TIME2_COMP   ; Timer/Counter2 Compare Match
.ORG  $0014
      RJMP  TIME2_OVF    ; Timer/Counter2 Overflow
      ...
; ISR for RESET
RESET:
      ...
      ...
      RETI


      ...
; ISR for External Interrupt 0
EXT_INT0:
      ...
      ...
      RETI
      ...
```

Figure 5.10: An example code for setting up interrupt vectors.

Saving and Restoring the Processor State

As mentioned before, an interrupt 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*, which is defined by PC, SREG, and GPRs, must not be changed. This way, when the control flow returns to the user program after executing the ISR, the state of the processor would be identical to the way it was before the interrupt was serviced. Thus, the state of the processor must be stored and restored before and after servicing the interrupt.

Although the processor automatically pushes the PC onto the stack, the programmer will have to save and restore GPRs and SREG if the ISR uses them. These registers can be saved on the stack using the PUSH (*Push register on stack*) instruction and restored from the stack using the POP (*Pop*
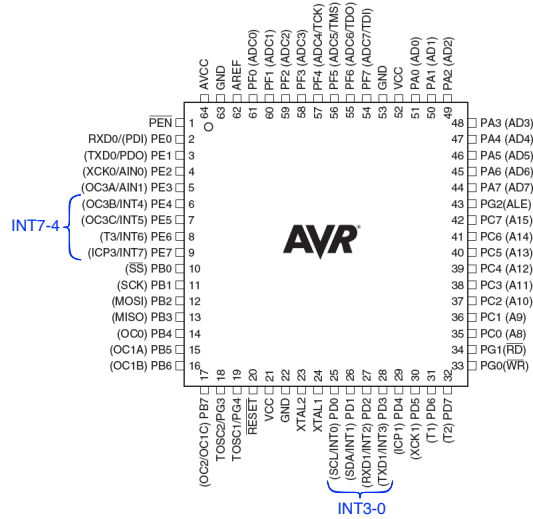
Figure 5.11: External interrupt pins.

register from stack) instruction.

5.3.2   AVR External Interrupts

Now that we have discussed the general concept of interrupts, this subsection presents the details on how to handle AVR external interrupts. AVR can handle up to eight external interrupts generated by I/O devices connected to pins INT7 - INT0. These pins are shown in Figure 5.11, where INT7-4 are connected to PORTE pins 7-4, while INT3-0 are connected to PORTD pins 3-0.

Figure 5.12 shows the set of I/O registers that control how external interrupts will be detected. 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 interrupt to be either detected or ignored. An interrupt can also be triggered by a falling edge, rising edge, or low level input signal by setting up *External Interrupt Control Registers* (EICR), which consists of a pair of 8-bit registers EICRA and EICRB.

Figure 5.13 illustrates how these registers are conceptually organized. EIFR latches interrupts from INT7-INT0, and when an interrupt is detected,

External Interrupt Flag Register (EIFR)

7	6	5	4	3	2	1	0	
INTF7	INTF6	INTF5	INTF4	INTF3	INTF2	INTF1	INTF0	EIFR
R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	

INTFn = 1   Triggers interrupt request

External Interrupt Mask Register (EIMSK)

7	6	5	4	3	2	1	0	
INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0	EIMSK
R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	

INTn = 1   Enables interrupt

External Interrupt Control Register A (EICRA)

7	6	5	4	3	2	1	0	
ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00	EICRA
R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	

External Interrupt Control Register B (EICRB)

7	6	5	4	3	2	1	0	
ISC71	ISC70	ISC61	ISC60	ISC51	ISC50	ISC41	ISC40	EICRB
R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	

ISCn1:0 External Interrupt n Sense Control Bits
00 - Low level generates an interrupt request.
01 - Reserved (for ISC3-0)
    Any logical change on generates an interrupt request (ISC7-4)
10 - Falling edge generates an interrupt request.
11 - Rising edge generates an interrupt request.

Figure 5.12: Control registers for interrupts.

it transfers the control to the corresponding ISR and clears the interrupt flag. These interrupt flags can also be manually cleared by writing 1's (yes, 1's not 0's!) to these bits. How the transfer of control is done will be discussed shortly. The EIMSK register is used to mask out unwanted interrupts. This can be thought of as a safety measure to prevent an undesired event (e.g., static electricity) from triggering an interrupt.

Finally, a pair of bits controls how the corresponding sense amplifier detects each external interrupt, 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 controls how interrupts are sensed for INT3-INT0, while EICRB controls how interrupts are sensed 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 other hand, EICRB provides one additional option for INT7-INT4, i.e., any level change (01).
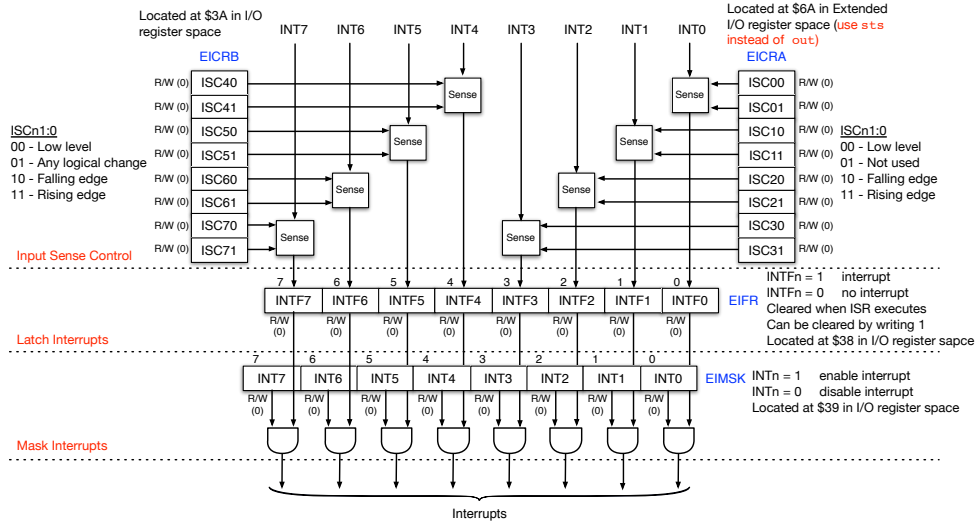
Figure 5.13: Controlling interrupts.

5.3.3   Interrupt-based TekBot Example

We conclude this section by developing an interrupt version of the TekBot Movement code discussed in Section 5.2.2.

Figure 5.14 shows the interrupt version of the TekBot Movement code. As seen previously in the basic Tekbot Movement code example in Figure 5.7, the code begins with a series of `.def` and `.equ` directives that assign symbolic names to registers and labels to values. Next, the interrupt vectors for INT0 and INT1, which are connected to the right and left bumper switches, respectively, are set up to transfer the control to appropriate ISRs. This is done by using `.org $0002` and `.org $0004` to place subroutine calls (i.e., `rcall`) to ISRs `HitRight` and `HitLeft`. Note that each of these subroutine calls 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 section starts by setting up the 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 functions `high(RAMEND)` and `low(RAMEND)` return 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 SRAM*), which is defined in

```
; AVR assembly code - Tekbot Movement (interrupt version)
.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 = 0                   ; Right Whisker Input Bit
.equ  WskrL = 1                   ; 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 Forward Command
.equ  MovBck = $00                ; Move Backward 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 Interrupt Vectors
      rjmp  INIT               ; Reset interrupt
      .org  $0002               ; INT0 => pin0, PORTD
      rcall HitRight           ; Call HitRight subroutine
      reti                     ; Return from interrupt
      .org  $0004               ; INT1 => pin1, PORTD
      rcall HitLeft            ; Call HitLeft subroutine
      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 5.14: Interrupt-based code for Tekbot movement.

```
     ; 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
     ; Initialize Port D for input
     ldi    mpr, (0<<WskrL)|(0<<WskrR)
     out    DDRD, mpr          ; Set the DDR register for Port D
     ldi    mpr, (1<<WskrL)|(1<<WskrR)
     out    PORTD, mpr         ; Set the Port D to Input with Hi-Z
     ; Initialize external interrupts (to trigger on falling edge)
     ldi    mpr, (1<<ISC01)|(0<<ISC00)|(1<<ISC11)|(0<<ISC10)
     sts    EICRA, mpr         ; Use sts, EICRA is in extended I/O space
     ; Set the External Interrupt Mask
     ldi    mpr, (1<<INT0)|(1<<INT1)
     out    EIMSK, mpr
     ; Turn on interrupts
     sei
;-----------------------------------------------------------
; Main Program
;-----------------------------------------------------------
MAIN:
     ; Move Robot Forward
     ldi    mpr, MovFwd        ; Load FWD command
     out    PORTB, mpr         ; Send to motors
     rjmp   MAIN               ; Infinite loop. End of program.
;----------------------------------------------------------------------
; Sub:     HitRight
; Desc:    Functionality for when the right bumper switch 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: Interrupt-based code for Tekbot movement (cont.)
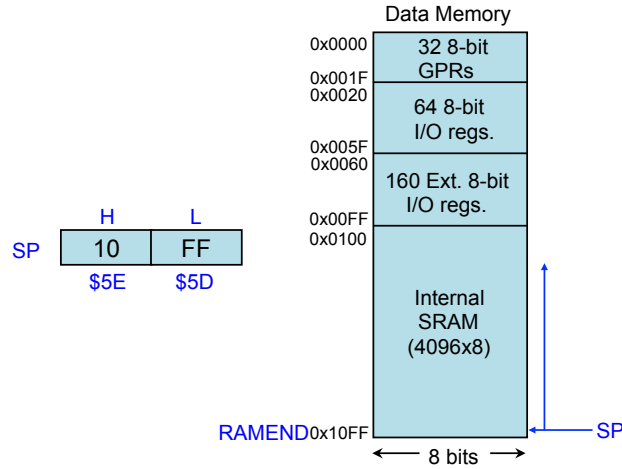
Figure 5.15: Initializing the stack.
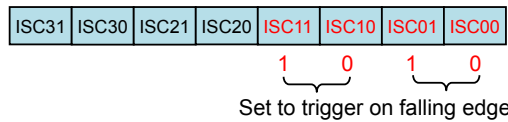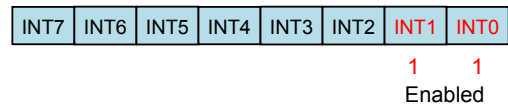
Figure 5.16: Initializing interrupts.

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 `$10FF` into the SP register. The rest of the Program Initialization involves setting up the ports and interrupt handling. Initialization of ports are similar to the busy-waiting version, with expressions used to define bit patterns for Data Direction Registers `DDRB` and `DDRD`.

Initialization of the external interrupts is done by appropriately setting

bits for `EICRA` and `EIMSK` registers. If you recall, the left and right bumper switches are connected to pins 1 and 0 of Port D, respectively (see Figure 5.6). There was a reason for this; these pins also happen to serve as INT1 and INT0 as shown in Figure 5.11.

Figure 5.16 illustrates the initialization of INT1 and INT0 to trigger on a falling edge, which is done because hitting a switch results in a voltage transition from high to low (when pull-up resistors are enabled). The instructions shown below uses labels `ISC11`, `ISC10`, `ISC01`, and `ISC00` to define the bit positions 3-0 within the `EICRA` register.

```
ldi   mpr, (1<<ISC01)|(0<<ISC00)|(1<<ISC11)|(0<<ISC10)
sts   EICRA, mpr  ; Set INT0 & 1 to trigger on falling edge
```

Note that these labels are predefined in the `m128def.inc` include file. This combined with shift operations generate the bit pattern 00001010, which is then written to the `EICRA` register. As shown in Figure 5.13, this causes interrupt signals on INT1 and INT0 to be triggered on a falling edge.

Similarly, the instructions shown below uses labels `INT1`,and `INT0` to define the bit positions 1 and 0 within the `EIMSK` register to generate the bit pattern 00000011.

```
ldi   mpr, (1<<INT0)|(1<<INT1)
out   EIMSK, mpr
```

Again, these labels are defined in the `m128def.inc` include file. When this value is written into the `EIMSK` register, all other interrupts are masked except for INT1 and INT0. The last part of the Program Initialization is to enable the interrupt using `sei` (*Set global interrupt flag*).

The `HitRight` routine is basically the 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 by 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 `push` instruction only works with GPRs and `SREG` is located in the I/O register address space (locations $5E and $5D for `SPH` and `SPL`, respectively). Thus, the I/O instruction `in` has to be used to first move it to a GPR.

Finally, with the interrupt facility setup, the Main Program is much simpler than the busy-waiting version. It simply involves repeatedly writing the bit pattern 0110000, which is generated based on the shift operations and directives defined for `EngDirL`, `EngDirR`, and `MovFwd`, into the PORTB register to move the TekBot forward. Although the Main Program of this code performs a very simple operation, the interrupt facility allows you to implement more complex operations.

5.4 Timers/Counters

Timer/Counters are one of the most commonly used features in a microcontroller. A Timer/Counter can be used to measure some elapsed time (clock cycles or ticks) or external events, e.g., the time between the leading edge of two input pulses. They can also be used to generate periodic outputs (i.e., a pulse train) to provide a baud rate clock to a USART (see Section 5.5) or to drive external devices, such as a DC motor.

AVR ATmega128 has two 8-bit Timer/Counters (Timer/Counter0 and 2) and two 16-bit Timer/Counters (Timer/Counter1 and 3). Each tick of the internal clock either increments or decrements the contents of these Timer/Counters.

5.4.1 Timer/Counter0 and 2

Figure 5.17 shows a block diagram of *Timer/Counter0*, which is an 8-bit Timer/Counter and consists of three user accessible registers: *Timer/-Counter 0 register* (TCNT0), *Output Compare Register 0* (OCR0), and *Timer/Counter Control Register 0* (TCCR0). TCNT0 is controlled by the *Control Logic* using three signals: The *count* signal enables counting, the *direction* signal selects between counting up or down, and the *clear* signal resets the count to zero.

The most basic way to use Timer/Counter0 is to write a value to TCNT0 and then let it count up to its maximum value – 255 for TCNT0 and 2 and 65,535 for TCNT1 and 3. When the count rolls over, the *Timer/-Counter Overflow 0* (TOV0) flag is set, which can be either manually detected (see Section 5.4.4) or used to automatically generate an interrupt (see Section 5.4.3). Thus, the elapsed time will be the difference between the maximum value and the value written to TCNT0 multiplied by the clock period. TCNT0 can also be used together with OCR0 to compare the two values, and if they are equal, a match is signaled by setting the *Output*
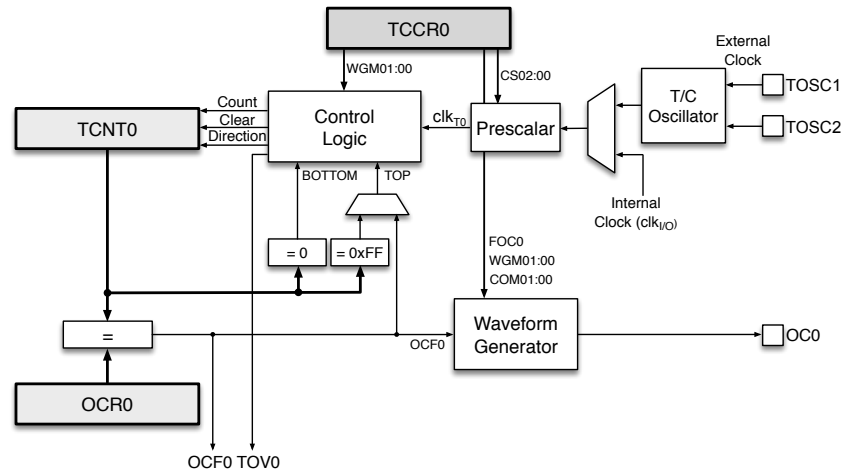
Figure 5.17: Block diagram of Timer/Counter0.

Compare Flag 0 (OCF0) at the next timer clock cycle. Similar to TOV0, OCF0 can be either manually detected or used to generate an interrupt. Both TOV0 and OCF0 can also be used to generate a waveform output on the *Output Compare pin 0* (OC0) using the *Waveform Generator*.

TCNT0 counts up (or down) with each tick of the *Timer/Counter0 clock* (clk_{T0}), which is by default connected to the system clock. This clock can also be scaled by setting the *Prescaler* module that controls the rate at which TCNT0 is incremented (or decremented). Timer/Counter0 can also be asynchronously clocked from external clocks connected to *Timer Oscillator pin 1* and *2* (TOSC1 and 2), which are bits 4 and 3 on Port G, respectively. The exact behavior of Timer/Counter0 depends on its mode of operation (see Section 5.4.4), which is controlled by setting TCCR0 (see Section 5.4.5).

Timer/Counter2 is also an 8-bit Timer/Counter with features almost identical to Timer/Counter0. The only difference is that Timer/Counter2 can be clocked by a single external clock source on the *Timer/Counter2 Clock Input* (T2) pin, which is bit 7 on Port D.

5.4.2   Timer/Counter1 and 3

16-bit Timer/Counter1 and *3* have similar functionalities as Timer/Counter0 and 2, but provide much higher range. Since both Timer/Counter1 and 3 are the same, we will only discuss the operations of Timer/Counter1.

The block diagram of the Timer/Counter1 is shown in Figure 5.18, which
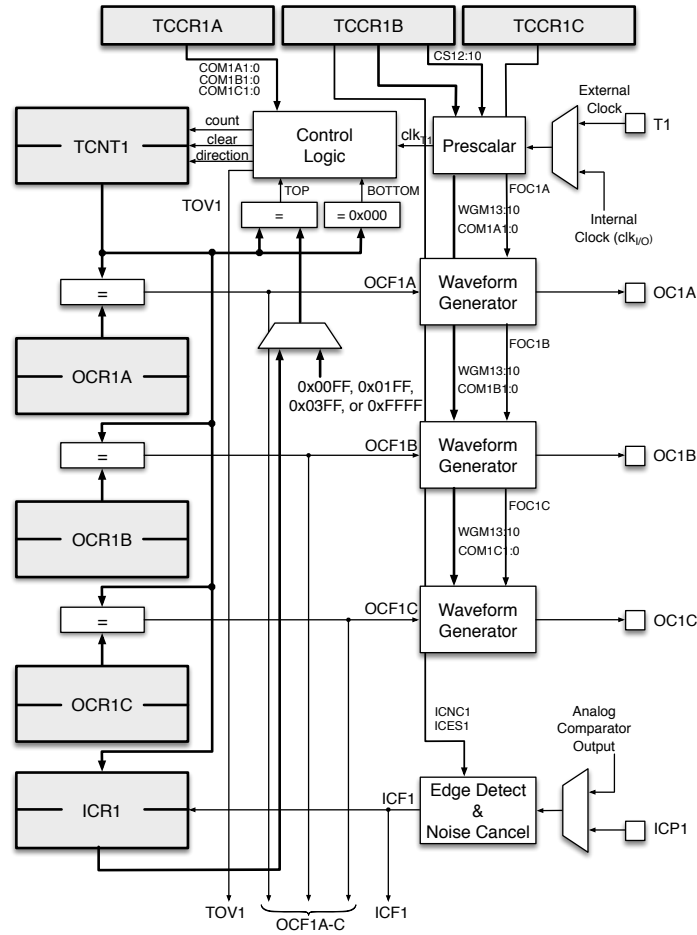
Figure 5.18: Block diagram of Timer/Counter1.

consists of 16-bit *Timer/Counter 1 register* (TCNT1) and three 16-bit *Output Compare Registers* (OCR1A, OCR1B, and OCR1C). Since all of these registers are 16 bits, each consists of a high byte and a low byte, e.g., TCNT1H and TCNT1L.

Similar to TCNT0, a value can be loaded onto TCNT1, which counts up to 65,535 (i.e., 0xFFFF) and when it rolls over (i.e., transition from 65,535 to 0) the *Timer/Counter Overflow 1* (TOV1) flag is set and can be used to generate an interrupt (see Section 5.4.4). The value of TCNT1 can also be continuously compared with OCR1A-C, and when a match occurs the corresponding Output Compare Flag (OCF1A, OCF1B, or OCF1C)

is set in the next clock cycle. These flags can then be used to generate output compare interrupts. These OCF1A-C signals can also be used by their respective Waveform Generators to generate waveform outputs based on the operating mode discussed in Section 5.4.4.

Timer/Counter1 also has a 16-bit *Input Capture Register 1* (ICR1), which is used to measure pulse widths of an incoming signal. This is done by the *Edge Detect and Noise Cancel* module that captures external events by associating timestamps. The captured timestamps can be used to calculate frequency, duty cycle, and other characteristics of the applied signal. The external signal indicating an event or multiple events can applied to the *Input Capture Pin 1* (ICP1), which is pin 4 on Port D. When an event (logic change) occurs on the ICP1 pin, a capture will be triggered. This causes the content of TCNT1 to be written to ICR1 and *Input Capture Flag 1* (ICF1) is set, which can then be used to generate an interrupt. The output of the Analog Comparator can also be used to trigger a capture (see Section 5.9).

5.4.3   Timer/Counter Interrupt Mask and Interrupt Flag Registers

In order to use Timer/Counter Overflow flags (TOV0 and TOV1) and Output Compare Flags (OCF0, OCF1A, OCF1B, and OCF1C) as interrupts, they first need to be enabled. This is done using the *Timer/Counter Interrupt Mask Register* (TIMSK). Figure 5.19 shows how Timer/Counter0 and 1 are connected to *Timer/Counter Interrupt Flag Register* (TIFR) and TIMSK.

For Timer/Counter0, the OCF0 and TOV0 flags are masked by *Timer/-Counter0 Output Compare Match Interrupt Enable* (OCIE0) and *Timer/-Counter0 Overflow Interrupt Enable* (TOIE0) bits, respectively, in the TIMSK register. Therefore, OCIE0 and TOIE bits must be set to 1 to enable these interrupts.

For Timer/Counter1, ICF1, OCF1A, OCF1B, OCF1C, and TOV1 flags are masked by *Timer/Counter Input Capture Interrupt Enable 1* (TICIE1), *Timer/Counter1 Output Compare A Match Interrupt Enable* (OCIE1A), *Timer/Counter1 Output Compare B Match Interrupt Enable* (OCIE1B), and *Timer/Counter1 Overflow Interrupt Enable* (TOIE1) bits, respectively, in the TIFR register. Note that OCF1C is detected by the OCF1C bit in the *Extended Timer/Counter Interrupt Flag Register* (ETIFR) and is masked by OCIE1C bit in the *Extended Timer Interrupt Mask Register* (ETIMSK), which are not shown in their entirety.
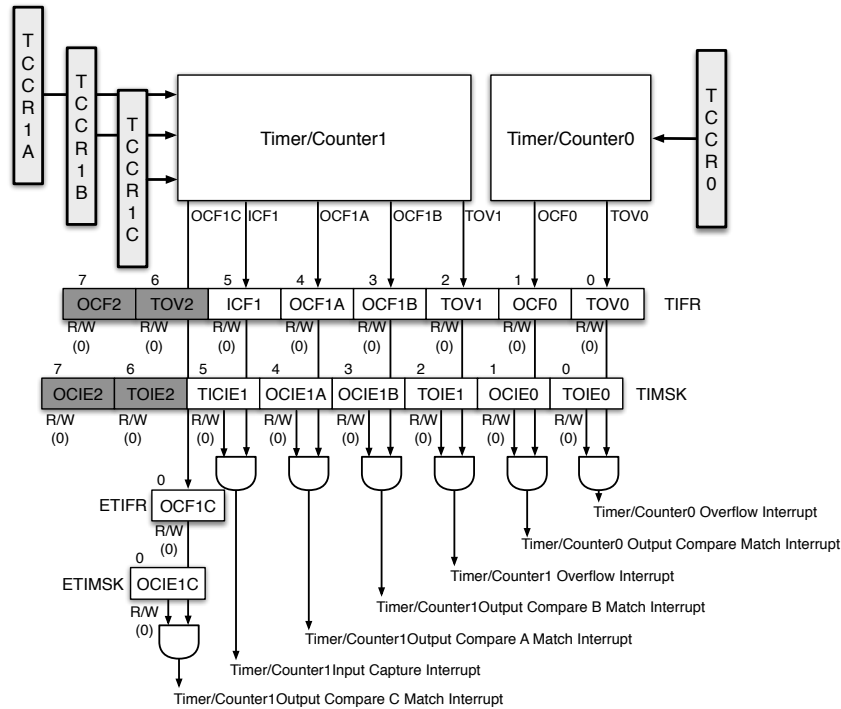
Figure 5.19: TIMSK and TIFR registers.

5.4.4 Modes of Operation

There are several modes of operation with Timer/Counters. This subsection discusses the three most commonly used modes, which are *Normal*, *Clear Timer on Compare Match* (CTC), and *Fast Pulse Width Modulation* (Fast PWM). There are two other less commonly used modes called *Phase Correct PWM* and *Phase and Frequency Correct PWM* that provide additional features, but these two modes will not be discussed. Instead, interested readers are encourage to look at the ATmega128 Datasheet. In addition, Table 5.2 defines some important Timer/Counter values used throughout this section.

Normal and CTC Modes

Figure 5.20 illustrates the basic operations of the Normal and CTC modes. In the Normal mode, TCNT0 is loaded with a value and the TOV0 flag in TIFR is set (see Figure 5.19) when the counter rolls over (i.e., transitions

Table 5.2: Definitions for BOTTOM, MAX, and TOP

BOTTOM	The Timer/Counter reaches the BOTTOM when it becomes zero (i.e., 0x00 or 0x0000).
MAX	The Timer/Counter reaches its MAX when it becomes 0xFF or 0xFFFF.
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 one of the OCRs, i.e., OCR0, OCR1A, OCR1B, and OCR1C, or assigned to 0xFF or 0xFFFFF (i.e., MAX). The assignment is dependent on the mode of operation.

(a) Normal mode   (b) CTC mode
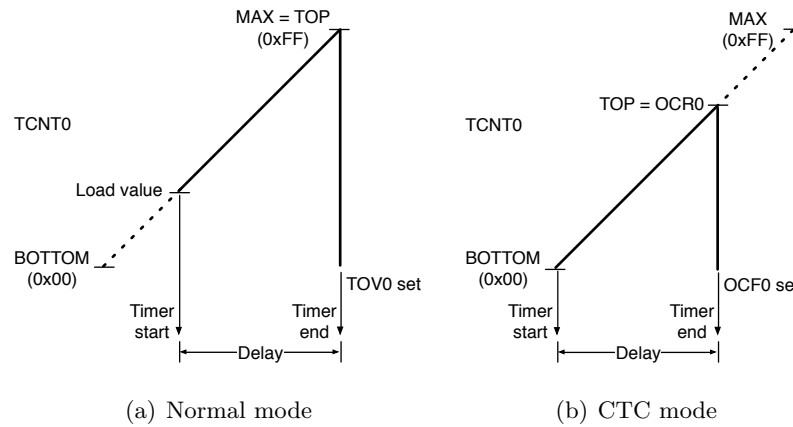
Figure 5.20: Timing diagrams of Normal and CTC modes for Timer/-Counter0.

from 0xFF to 0x00 for TCNT0 and 2 or 0xFFFF to 0x0000 for TCNT1 and 3). On the other hand, in the CTC mode, the count starts at 0 and OCR0 is used to control when the counting ends. The content of OCR0 defines the TOP value and thus its resolution. Moreover, TCNT0 is reset to zero when TCNT0 and OCR0 match.

Note that both TOV0 and OCF0 need to be cleared, which is achieved by writing 1 to these flags, before they can be used again. Moreover, in order to generate interrupts on TOV0 and OCF0, these flags need to be enabled by setting the TOIE0 and OCIE0 bits in the TIMSK register (see Figure 5.19). In addition, the loaded value for the Normal mode and the OCR0 value for the CTC mode can be adjusted using interrupts.

For the Normal mode, the time or delay period between when a Timer/-

Counter is loaded with a value and its corresponding TOV0 flag is set, $Delay_{Normal}$, is given by the following equations:

$$Delay_{Normal} = \frac{(MAX + 1 - value) \cdot prescale}{clk_{I/O}}, \qquad (5.3)$$

where *value* is the initial count, MAX represents 255 for TCNT0 and 2 and 65,535 for TCNT1 and 3, $clk_{I/O}$ represents the I/O clock frequency, and *prescale* can be 1, 8, 32, 64, 128, 256 or 1024 (see Table 5.3). The prescale can be adjusted by configuring TCCR0, which will be discussed later in Section 5.4.5.

For the CTC mode, the time or delay period between when a Timer/-Counter start at 0 and its corresponding OCF0 flag is set, $Delay_{CTC}$, is given by the following equations:

$$Delay_{CTC} = \frac{(TOP + 1) \cdot prescale}{clk_{I/O}}, \qquad (5.4)$$

where TOP represents the value in OCR0.

The following two examples show how TCNT0 can be used in Normal mode.

Example 5.1. *Suppose we want a delay of 10 ms using Timer/Counter0 with the system clock frequency of 16 MHz using the Normal mode.*

16 MHz system clock leads to a clock period of 62.5 ns. Solving for *value* in Equation 5.3 leads to the following equation:

$$value = 256 - \frac{10ms}{prescale \times 62.5ns}$$

In general, any combination of *value* and *prescale* that satisfies the above equation will work. However, we would like to use a *prescale* value that would lead to the highest resolution (i.e., lowest prescale value) and yet the period can be covered by the timer count. Prescale values of 1, 8, 32, 64, 128, and 256 will all not work since they result in the second component of the above equation to be larger than 256. Therefore, a prescale of 1024 will be used resulting in each tick to be 62.5 ns × 1024 = 64 μs. Thus, the value to be loaded onto TCNT0 is $\lceil$256 - (10 ms/64 μs)$\rceil$ = 100, which leads to a delay period of 9.98 ms. We can get this delay to be much closer to 10 ms by using a higher resolution 16-bit Timer/Counters, but we will limit our discussion to 8-bit Timer/Counters. Instead, using a 16-bit Timer/Counter is left as an exercise.

Example 5.2. *Write a subroutine called* `WAIT_1sec` *that waits for 1 sec before returning using the delay derived in Example 5.2. Assume the microcontroller has already been configured to operate in Normal mode with a prescale of 1024.*

The code shown below implements a 1 sec delay, where the basic idea is to execute 10 ms delay 100 times. This is done by setting up an outer-most loop that executes 100 times, and for each iteration, the value 100 is loaded onto the Timer/Counter0 register. Afterwards, the TOV0 flag is repeatedly tested until it is set indicating 10 ms has elapsed. TOV0 is reset by writing a 1 to the flag, and the outermost loop count is decremented. This process repeats until `count` reaches 0.

```
; AVR assembly code - Wait one second
WAIT_1sec:
      LDI   R17, 100           ; Load count = 100
WAIT_10msec:
      LDI   R16, 100           ; Value for delay = 100
      OUT   TCNT0, R16         ; (Re)load a value for delay
LOOP:
      IN    R18,TIFR           ; Read in TOV0
      ANDI  R18, 0b00000001    ; Check if its set
      BREQ  LOOP               ; Loop if TOV0 not set
      LDI   R18, 0b00000001    ; Reset TOV0
      OUT   TIFR, R18          ; Note - write 1 to reset
      DEC   R17                ; Decrement count
      BRNE  WAIT_10msec        ; Loop if count not equal to 0
      RET
```

In addition to measuring some elapsed time, the CTC mode can be used to generate a waveform on the OC0 pin, which is the pin 4 on Port B, by toggling, setting, or clearing it on each match. For example, the waveform shown in Figure 5.21 is generated by configuring the CTC mode to toggle the OC0 pin each time TCNT0 and OCR0 match. Thus, two iterations of count up and a match generate a signal period with 50% *duty cycle*, which is the proportion of the time the output is high within a period. In addition, the period of the waveform can be adjusted by changing the value of OCR0. Thus, both Normal and CTC modes can be used to generate a waveform with a fixed duty cycle with varying frequency. It is important to note that
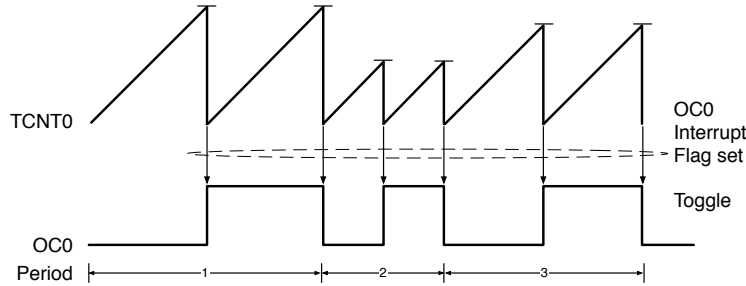
Figure 5.21: Example timing of PWM in CTC mode.

the OC0 pin is part of a port. Therefore, the data direction for the pin must be set to output to make signals be 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 the CTC mode, f_{CTC}, is controlled using the following equation:

$$f_{CTC} = \frac{clk_{I/O}}{2 \cdot prescale \cdot (1 + OCR0)},\qquad(5.5)$$

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

Note that the Normal mode can also be used to generate a waveform on the OC0 pin. However, this is not recommended because the loaded value is not retained as it counts up, and thus it has to be reloaded each pulse. This is in contrast to OCR0 for CTC mode, which is loaded just once.

Fast Pulse Width Modulation (Fast PWM) Mode

The *Fast PWM* mode allows for generating high frequency pulse or square waves. Before discussing the details of this mode, some explanation is in order to understand the usefulness of this feature. PWM uses a rectangular pulse wave whose pulse width can be modulated to vary the average value of the waveform. This method is commonly used to drive motors, heaters, or lights in varying speeds or intensities.

Figure 5.22 shows an example Fast PWN mode timing diagram. In this mode, TCNT0 counts up from BOTTOM (0x00) to MAX (0xFF), then restarts at BOTTOM. For a *non-inverted PWM output*, the OC0 signal is set to 0 when TCNT0 and OCR0 match, and is set to 1 when the counter transitions from 0xFF to 0x00. The inverse occurs using an *inverted PWM*
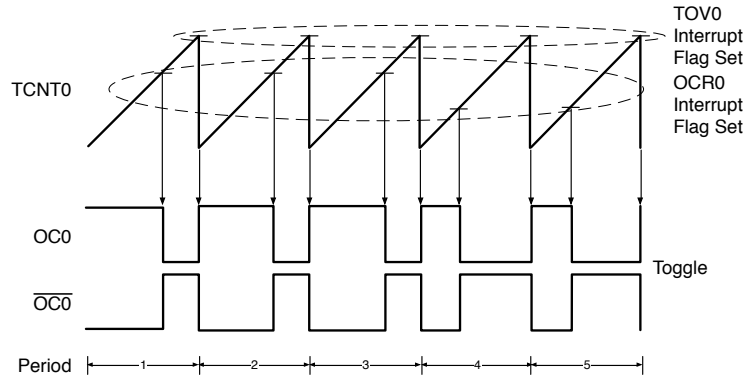
Figure 5.22: Example timing diagram in Fast PWM mode.

output. The result is a waveform where its duty cycle can be varied by
adjusting the value of OCR0. Thus, the Fast PWM mode can be used to
generate a waveform with a fixed frequency, but whose duty cycle is variable.
Note that both Fast PWM and CTC modes can be used to generate a pulse
train. However, the CTC mode can only vary the frequency of the waveform,
not its duty cycle, and thus it cannot be used to perform PWM.

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

$$f_{PWM} = \frac{clk_{I/O}}{prescale \cdot 256},\tag{5.6}$$

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

5.4.5   Timer/Counter Control Register

Controlling the behavior of Timers/Counters is done with the *Timer/-
Counter Control Register 0-3* (TCCR0-3). Again, since TCCR0 and 2 have
identical formats, we will only discuss how TCCR0 can be used to control
the operation of TCNT0. The same will be true for TCCR1 and 3.

TCCR0 and 2

Figure 5.23 shows the format of TCCR0. The *Clock Select* bits CS02, CS01,
and CS00 allow the internal frequency to be derived or scaled from the I/O
clock ($clk_{I/O}$) using the prescale factor. This allows the rate at which a

Timer/Counter Control Register 0 (TCCR0)

7	6	5	4	3	2	1	0	
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
W (0)	R/W (0)	R/W (0)	R/W (0)	R/0 (0)	R/W (0)	R/W (0)	R/W (0)	

Bit 7 - Force Output Compare
Bit 6, 3 - Waveform Generation Mode
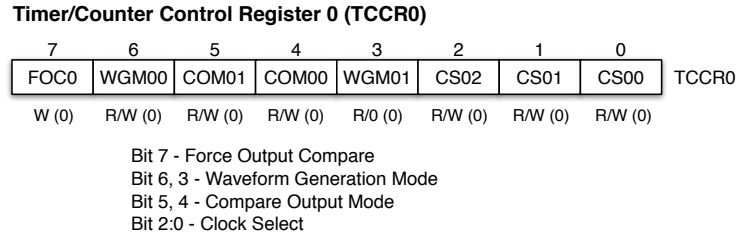Bit 5, 4 - Compare Output Mode
Bit 2:0 - Clock Select

Figure 5.23: Timer/Counter Control Register 0.

Timer/Counter increments/decrements to be controlled. Table 5.3 shows the eight possible choices.

Table 5.3: Clock Select bits in TCCR0

Clock Select bits in TCCR0.			
CS02	CS01	CS00	Description
0	0	0	No clock source
0	0	1	$clk_{I/O}$
0	1	0	$clk_{I/O}/8$
0	1	1	$clk_{I/O}/32$
1	0	0	$clk_{I/O}/64$
1	0	1	$clk_{I/O}/128$
1	1	0	$clk_{I/O}/256$
1	1	1	$clk_{I/O}/1024$

The different modes of operation are defined by *Waveform Generation Mode* bits WGM01 and WGM00. This is shown in Table 5.4.

Table 5.4: Description of Waveform Generation Mode bits in TCCR0.

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

The behavior of the OC0 pin is determined by *Compare Output Mode* bits COM01 and COM00 shown in Table 5.5, as well as the mode of operation. When the COM01 and COM00 bits are both set to zeros, the OC0 pin (which is shared with bit 4 of Port B) is disconnected from Timer/Counter0

and operates as an ordinary port pin. When COM01 and COM00 are set to 0 and 1, respectively, the OC0 output toggles whenever a compare match occurs in Normal and CTC modes. When COM01 and COM00 are set to 1 and 0, respectively, the OC0 output is cleared whenever a compare match occurs for all three modes. However, the Fast PWN mode has an addition behavior, which is to set the OC0 output to TOP. Finally, when COM01 and COM00 are both set to ones, the OC0 output is set whenever a compare match occurs for all three modes. In addition, the OC0 output is set to TOP in the Fast PWM mode.

Table 5.5: Description of Compare Output Mode (COM) bits in TCCR0

COM01	COM00	Normal	CTC	Fast PWM
0	0	Normal port operation, OC0 disconnected		
0	1	Toggle OC0 on compare match		Reserved
1	0	Clear OC0 on compare match		Clear OC0 on compare match, set OC0 at TOP (non-inverting)
1	1	Set OC0 on compare match		Set OC0 on compare match, clear OC0 at TOP (inverting)

Finally, setting the *Force Output Compare* bit FOC0 forces an immediate compare match, which affects the behavior of the OC0 pin according to COM01 and COM00 bit settings, but does not set the OCF0 bit. This feature is useful when the OC0 pin needs to be set to either 1 or 0. Note that the FOC0 bit is only active when WGM01:00 bits are set to a non-PWM mode (i.e., Normal and CTC modes).

TCCR1 and 3

The behavior of Timer/Counter1 is controlled by *Timer Counter Control Register 1A, 1B, and 1C* (TCCR1A, TCCR1B, and TCCR1C.) These registers are shown in Figure 5.24.

TCCR1A provides similar features as TCCR0 with additional capabilities. The bits COM1A1-0, COM1B1-0, and COM1C1-0 control the output compare pins OC1A, OC1B, and OC1C, respectively. Table 5.6 shows the meaning of these bits for Normal, CTC, and Fast PWM modes.

Table 5.7 shows the description of the Wave Generation Mode bits WGM13, WGM12, WGM11, and WGM10, which are spread across TCCR1A and

7	6	5	4	3	2	1	0	
COM1A1	COM1A0	COM1B1	COM1B0	COM1C1	COM1C0	WGM11	WGM10	TCCR1A
R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/0 (0)	R/W (0)	R/W (0)	R/W (0)	

Bit 7:6 - Compare Output Mode for Channel A
Bit 5:4 - Compare Output Mode for Channel B
Bit 3:2 - Compare Output Mode for Channel C
Bit 1:0 - Waveform Generation Mode bits 1:0

7	6	5	4	3	2	1	0	
ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
R/W (0)	R/W (0)	R (0)	R/W (0)	R/0 (0)	R/W (0)	R/W (0)	R/W (0)	

Bit 7 - Input Capture Noise Canceler
Bit 6 - Input Capture Edge Select
Bit 5 - Reserved
Bit 4:3 - Waveform Generation Mode bits 3:2
Bit 2:0 - Clock Select

7	6	5	4	3	2	1	0	
FOC1A	FOC1B	FOC1C	-	-	-	-	-	TCCR1C
W (0)	W (0)	W (0)	R (0)	R (0)	R (0)	R (0)	R (0)	

Bit 7 - Force Output Compare for Channel A
Bit 6 - Force Output Compare for Channel B
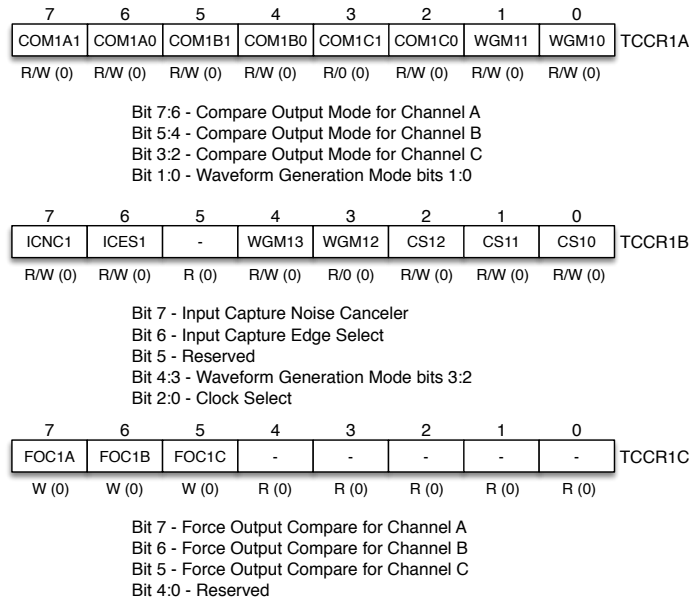Bit 5 - Force Output Compare for Channel C
Bit 4:0 - Reserved

Figure 5.24: Timer/Counter Control Register 1.

TCCR1B. The three basic Wave Generation Modes are 0 (Normal), 4 (CTC), and 15 (Fast PWM). In addition, there are variations to CTC and Fast PWM operations. For instance, Fast PWM also provides 8-bit, 9-bit, and 10-bit resolutions, which limit the count to 0x00FF, 0x01FF, and 0x03FF, respectively. Furthermore, both CTC and Fast PWM provide ICR1 as the TOP value.

The bits CS12, CS11, and CS10 in TCCR1B are Clock Select bits for TCNT1 and are defined in Table 5.8. The Input Capture Edge Select (ICES1) bit chooses between rising and falling edge from the *Input Capture Pin 1* (ICP1) to capture the content of TCNT1 onto ICR1. The *Input Capture Noise Canceler 1* (ICNC1) bit is used to activate the filtering of the signal from ICP1. The filter function requires four successive equal valued samples of the ICP1 pin for changing its output.

Finally, setting the *Force Output Compare* bits FOC1A, FOC1B, and FOC1C in TCCR1C force immediate compare matches that affects the behavior of the OC1A, OC1B, and OC1C pins according to COM1A1:0, COM1B1:0, and COM1C1:0 bit settings, respectively. As was the case for the FOC0 bit in TCCR0, these bits do not set the OCF1A:C bits and are

Table 5.6: Description of Compare Output Mode (COM) bits in TCCR1

COM1A1:0 COM1B1:0 COM1C1:0	Normal	CTC	Fast PWM
00	Normal port operation, OC1A-C disconnected		
01	Toggle OC1A-C on compare match		Reserved
10	Clear OC1A-C on compare match		Clear OC1A-C on compare match, set OC1A-C at BOTTOM (non-inverting)
11	Set OC1A-C on compare match		Set OC1A-C on compare match, clear OC1A-C at BOTTOM (inverting)

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
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
12	1100	CTC	ICR1	Immediate	TOP
14	1110	Fast PWM	ICR1	TOP	TOP
15	1111	Fast PWM	OCR1A	TOP	TOP

only active when WGM13:10 bits are set to a non-PWM mode (i.e., Normal and CTC modes).

5.4.6   Assembly Program Examples Using Timers/Counters

This subsection concludes the discussion on Timers/Counters by presenting several example programs to demonstrate how TCNT0 together with the various modes can be used to turn on and off an LED connected to the OC0 pin.

Our first example code causes an LED to turn on for approximately half a second. This is done by first turning on OC0 then having TCNT0 generate a delay of 10 ms as discussed in Example 5.2, and then encapsulating this delay in a loop that iterates 50 times. For this version, TCNT0 operates in Normal mode, and the TOV0 flag is continually checked to see if TCNT0

Table 5.8: Clock Select bits in TCCR1B

Clock Select bits in TCCR1B.			
CS02	CS01	CS00	Description
0	0	0	No clock source
0	0	1	$clk_{I/O}$
0	1	0	$clk_{I/O}/8$
0	1	1	$clk_{I/O}/64$
1	0	0	$clk_{I/O}/256$
1	0	1	$clk_{I/O}/1024$
1	1	0	External clock source on T1 pin. Clocked on falling edge
1	1	1	External clock source on T1 pin. Clocked on rising edge

rolls over. When TCNT0 rolls over 50 times, OC0 is turned off. The code is shown below.

```
; AVR Assembly Code - Turn on OC0 for 500 ms
; (Normal mode, OC0 disconnected)

.INCLUDE "m128def.inc"
.DEF A = R16                    ; General purpose register A
.DEF B = R17                    ; General purpose register B

.ORG $0000                      ; Reset and Power On interrupt
      RJMP  INITIALIZE          ; Jump to initialization
.ORG $0046                      ; End of interrupt vectors

INITIALIZE:
      ; Initialize stack
      LDI   A, high(RAMEND)
      OUT   SPH, A
      LDI   A, low(RAMEND)
      OUT   SPL, A
      ; Initialize TCNT0
      SBI   DDRB, PB4           ; Set bit 4 of port B (OC0) for output
      LDI   A, 0b00000111       ; Activate Normal mode, OC0 disconnected,
      OUT   TCCR0, A            ; and set prescaler to 1024

MAIN:
      SBI   PORTB, PB4          ; Turn on OC0
      RCALL WAIT_0.5sec         ; Call WAIT_0.5sec subroutine
      CBI   PORTB, PB4          ; Turn off OC0
LOOP:
```

```
        RJMP  LOOP                ; Loop forever

; Subroutine to wait for 500 ms
WAIT_0.5sec:
        LDI   B, 50               ; Load loop count = 50
WAIT_10msec:
        LDI   A, 100              ; (Re)load value for delay
        OUT   TCNT0, A
; Wait for TCNT0 to roll over
CHECK:
        IN    A, TIFR             ; Read in TIFR
        ANDI  A, 0b00000001       ; Check if TOV0 set
        BREQ  CKECK               ; Loop if TOV0 not set
        LDI   A, 0b00000001       ; Otherwise, Reset TOV0
        OUT   TIFR, A             ; Note - write 1 to reset
        DEC   B                   ; Decrement count
        BRNE  WAIT_10msec         ; Loop if count not equal to 0
        RET
```

The above code executes the `INITIALIZE` code upon reset, which sets up the stack and the OC0 pin for output. In addition, TCNT0 is configured to operate in Normal mode (i.e., WGM01=0 and WGM01=0) with the OC0 pin disconnected from the Waveform Generator (i.e., COM01=0 and COM00=0), and the prescaler value is set to 1024 (i.e., CS02=1, CS01=1, and CS00=1). In the `MAIN_LOOP`, the OC0 pin is first turned on. Afterwards, a subroutine call is made to `WAIT_0.5sec`, which implements `WAIT_10msec` loop. For each iteration of the loop, TCNT0 is loaded with 100 and the TOV0 bit is continuously checked to see if it is set. When it is set, TOV0 is reset (by writing a 1), and the loop count is decremented. After executing the loop for 50 times, the `WAIT_0.5sec` subroutine returns. Finally, the OC0 pin is turned off.

In the following example code, we use the CTC mode to cause the LED to blink by toggling the OC0 pin on and off approximately every half a second.

```
; AVR Assembly Code - Manually toggle OC0 every 500 ms
; (CTC mode, OC0 disconnected)
.INCLUDE "m128def.inc"
.DEF A = R16                ; General purpose register A
.DEF B = R17                ; General purpose register B
```

```
.ORG $0000                      ; Reset and Power On interrupt
      RJMP   INITIALIZE         ; Jump to initialization
.ORG $0046                      ; End of interrupt vectors

INITIALIZE:
      ; Initialize stack
      ...
      ; Initialize TCNT0
      SBI    DDRB, PB4          ; Set pin 4 of Port B (OC0) for output (LED)
      LDI    A, 0b00001111      ; Activate CTC mode, OC0 disconnected,
      OUT    TCCR0, A           ; and set prescaler to 1024
      LDI    A, 156             ; Set output compare value
      OUT    OCR0, A            ;

MAIN_LOOP:
      RCALL TOGGLE              ; Call TOGGLE subroutine
      RCALL WAIT_0.5sec         ; Call WAIT_0.5sec subroutine
      RJMP  MAIN_LOOP           ; Loop forever

; Subroutine to toggle OC0
TOGGLE:
      IN    A, PORTB            ; Get current OC0 value
      LDI   B, (1 << PB4)       ; Set bit to toggle
      EOR   A,B                 ; Toggle OC0
      OUT   PORTB, A            ; Write it back
      RET


; Subroutine to wait for 500 ms
WAIT_0.5sec:
      LDI   B, 50               ; Load loop count = 50
Wait_10msec:
      LDI   A, 0                ; Initialize TCNT0 to 0
      OUT   TCNT0, A
; Wait for TCNT0 to match with OCR0
LOOP:
      IN    A,TIFR              ; Read in OCF0 in TIFR
      ANDI  A, 0b00000010       ; Check if OCF0 set
      BREQ  LOOP                ; Loop if OCF0 not set
      LDI   A, 0b00000010       ; Otherwise, reset OCF0
      OUT   TIFR, A             ; Note - write 1 to reset
      DEC   B                   ; Decrement count
      BRNE  WAIT_10msec         ; Loop if count not equal to 0
      RET
```

First, TCCR0 is set to the CTC mode (i.e., WGM01=1 and WGM01=0),

which sets the OCF0 flag on compare match. In addition, OC0 is discon-
nected from the Waveform Generator so that it can be manually toggled
when a compare match occurs (i.e., COM01=0 and COM00=0). Second,
the OCR0 register is loaded with the value 156 (see Equation 5.4), which
results in delay time of 10 ms. In the MAIN_LOOP, the subroutine TOGGLE is
called to read the current value of OC0, toggle it by performing an exclusive-
OR operation with a 1, and then write it back. Afterwards, a subroutine
call is made to WAIT_0.5sec, which implements the WAIT_10msec loop. In
the WAIT_0.5sec subroutine, TCNT0 is initialized to 0 for each occurrence
of 10 ms delay. Moreover, the OCF0 flag tested to see if it is set.

In the following two example codes, we use Fast PWM mode to generate
a pulse train to drive the LED. This has the effect of varying the LED's
intensity, depending on the frequency of PWM.

The following example uses the Fast PWM mode to automatically toggle
OC0 every 8.16 ms.

```
; AVR Assembly code - Fast PWM mode
      .INCLUDE    "m128def.inc"
      .DEF  A = R16              ; General purpose register

      .ORG $0000                 ; Reset and Power On interrupt
      RJMP  INITIALIZE           ; Jump to initialization
      .ORG $0046                 ; End of interrupt vectors

INITIALIZE:
      ; Initialize stack
      ...
      SBI   DDRB, PB4            ; Set bit 4 of port B (OC0) for output
      LDI   A, 0b01101111        ; Activate Fast PWM mode with toggle
      OUT   TCCR0, A             ; (non-inverting), and set prescaler to 1024
      LDI   A, 128               ; Set compare value
      OUT   OCR0, A              ;

MAIN_LOOP:
      RJMP MAIN_LOOP             ; Do nothing loop
```

During initialization, TCCR0 is set to the Fast PWM mode (i.e., WGM01=1
and WGM01=1), and clear the OC0 pin on compare match and set the
OC0 pin when TCNT0 reaches TOP value (i.e., COM01=1 and COM00=0).
Moreover, OCR0 is set to 128, which is the mid-point of TCNT0. This will

generate a pulse train with a frequency of 122.5 Hz with a duty cycle of 50%.

In our final example, we take the Fast PWM mode version discussed above and add the capability to adjust the duty cycle using interrupts.

```
; AVR Assembly code - Fast PWM mode with adjustable duty cycle

.INCLUDE "m128def.inc"
.DEF A = R16                 ; General purpose register

.ORG $0000                   ; Reset on Power On interrupt
     RJMP   INITIALIZE       ; Jump to initialization
.ORG $001E                   ; Compare Match vector
     RJMP   TIMO_COMPA
.ORG  $0046                  ; End of interrupt vectors

INITIALIZE:
     SBI    DDRB, PB4        ; Set bit 4 of port B (OC0) for output
     LDI    A, 0b01101111    ; Activate Fast PWN mode with toggle
     OUT    TCCR0, A         ; (non-inverting) & set prescaler to 1024
     LDI    A, 0b00000010    ; Enable output compare interrupt
     OUT    TIMSK, A
     SEI                     ; Enable global interrupt

MAIN_LOOP:
     RJMP MAIN_LOOP          ; Loop and wait for for interrupts

TIMO_COMPA:
     IN    A, OCR0           ; Read OCR0
     INC   A                 ; Increment OCR0
     OUT   OCR0, A           ; Write it back
     RETI
```

In this version, an interrupt is set up to execute the TIMO_COMPA interrupt service routine when a compare match interrupt occurs. This is done by using the interrupt vector located at $001E, which is for the *Timer/Counter0 Compare Match* interrupt (see Table 5.1). The TIMO_COMPA interrupt service routine basically increments the value in OCR0. Thus, OCR0 initially starts at 0 and increments up to 255 causing the duty cycle to start at 0% and increase to 100%. This causes the intensity of the LED to slowly increase within a span of 16.38 ms $\times$ 256= 4.19 sec.

5.5 USART

Universal Synchronous/Asynchronous Receiver/Transmitter (USART) is a highly flexible serial communications system. The USART hardware allows a microcontroller to transmit and receive data serially to and from other devices, such as a computer or another microcontroller. USART is supported by many embedded I/O devices and sensors, including Bluetooth, Infrared (IR), RFID reader, Global Positioning System (GPS), Global System for Mobile communication (GSM), etc. In older computers, devices such as mice, printers, and dial-up modems used USART to communicate via a *serial port* using the *RS-232 protocol*. Serial ports have since been displaced by Universal Serial Bus (USB); however, they are still used in many test and measurement equipment, industrial machines, and networking equipment.

5.5.1 Serial Communications Basics

Before getting into the details of AVR's USART capabilities, we begin with a discussion on the basic concepts of serial communication.

A USART *transmitter* (Tx) takes an n-bit data and transmits the individual bits in a serial fashion. A USART *receiver* (Rx) re-assembles the received bits into the original data. Each USART contains a shift register, which is used to convert data between serial and parallel forms.

There are some issues to consider when two independent devices have to communicate serially. First, how does the receiver know when the data being transmitted starts and ends? Second, how fast are the bits transmitted/received, and how do the sender and receiver agree on a transmission rate? The answer to the first question is to encapsulate serial data into *frames*. The answer to the second question depends on whether the serial communication is performed *synchronously* or *asynchronously*. The following two subsections discuss these issues in more detail.

Serial Data Frame Format

The serial data frame format is shown in Figure 5.25. A *frame* contains n-bit of data (e.g., 8-bit) with *synchronization bits* (start and stop bits), and optionally a *parity bit* for error checking. USART accepts the following combinations as valid frame formats:
- 1 start bit
- 5, 6, 7, 8, or 9 data bits
- none, even or odd parity bit

St   Start bit, alway low
(n)   Data bit (0 to 8)
P   Parity bit (odd or even)
Sp   Stop bit, always high
IDLE   No transfers on the communication line (RxD or TxD). An IDLE line must be high
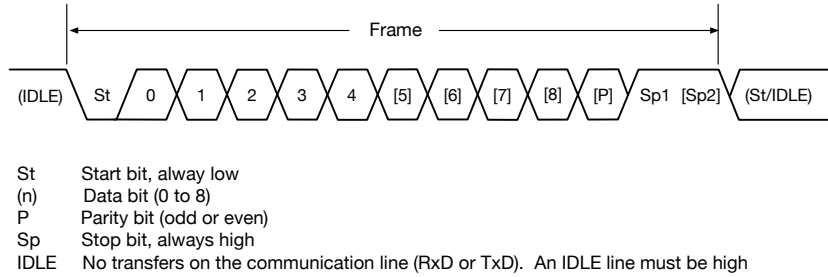
Figure 5.25: Data frame format.

- 1 or 2 stop bits

Initially, the signal on the serial port is high indicating that it is *idle*. A frame starts with a falling edge, which indicates the beginning of the *start* (St) bit. This is followed by the least significant data bit (i.e., bit 0). Then, the data bits, up to a total of nine, follow and end with the most significant bit (i.e., either bit 7 or 8). If enabled, the *parity* (P) bit is inserted after the data bits, and before the *stop* (Sp) bit(s), which can be either one (Sp1) or two bits (Sp1 and Sp2). When the transmission of a frame completes, it can be directly followed by a new frame, or the serial port can be set to the idle (i.e., high) state.

The P-bit is an additional bit transmitted with the data to enhance data integrity during transmission. With a single parity bit, a single bit error can be detected. The parity for the data can be either even or odd. For *even parity*, the parity is set to make the total number of 1's even. For *odd parity*, the parity is set to make the total number of 1's odd. The P-bit is calculated by performing an exclusive-OR of all the data bits. If odd parity is used, the result of the exclusive-OR is inverted. The relationship between the parity bit and data bits is given as follows:

$$P_{even} = d_{n-1} \oplus \cdots \oplus d_2 \oplus d_1 \oplus d_0 \oplus 0 \qquad (5.7)$$
$$P_{odd} = d_{n-1} \oplus \cdots \oplus d_2 \oplus d_1 \oplus d_0 \oplus 1$$

For example, if we have a data byte 0b00101101 and we want odd parity, the parity bit is set to 1 to make the total number 1's five, which results is an odd number of 1's.

(a) Synchronous.   (b) Asynchronous.
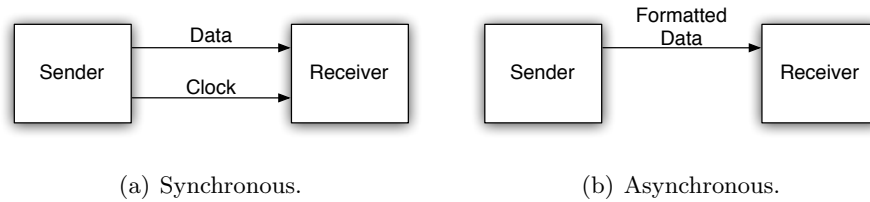
Figure 5.26: Synchronous vs. asynchronous serial communication.

Synchronous vs. Asynchronous Serial Communication

Figure 5.26 shows the basic difference between synchronous and asynchronous modes of transmission. In *synchronous* mode, the sender (i.e., master) provides a clock to the receiver (i.e., slave), and this clock synchronizes the two devices. In *asynchronous* mode, both sender and receiver agree on a *Baud rate* (see Section 5.5.3), and the receiver automatically recovers the transmitted clock rate by detecting the incoming signal at this agreed rate.

5.5.2 AVR's USART

There are two identical USARTs in AVR ATmega128: *USART0* and *US-ART1*. USART pins for the ATmega128 is shown in Figure 5.27. Each USARTn has a pair of *Receive Data* (RxDn) and *Transmit Data* (TxDn) pins, and *External Clock* (XCKn) pin, where n is 0 and 1 for USART0 and USART1, respectively.

The block diagram of USARTn is shown in Fig. 5.28, which consists of Clock Generator, Transmitter, and Receiver. The *Transmitter* block is responsible for transmitting data bits serially on the TxDn pin. This is done by writing the data to be transmitted to *USARTn I/O Data Register* (UDRn), which then gets moved to a special buffer, called *Transmit Shift Register*. This also frees up UDRn for the subsequent transmission. The Transmit Shift Register shifts the data a bit at a time and adds parity bit(s) and transmits them on the TxDn pin. The *Receiver* block receives the data on the RxDn pin and checks and recovers the data onto *Receive Shift Register*, and when all the data bits of a frame are properly received they are moved to UDRn (Receive). The *Clock Generator* block consists of *Baud Rate Generator*, which is controlled by *USARTn Baud Rate Registers high and low* (UBRRnH and UBRRnL).
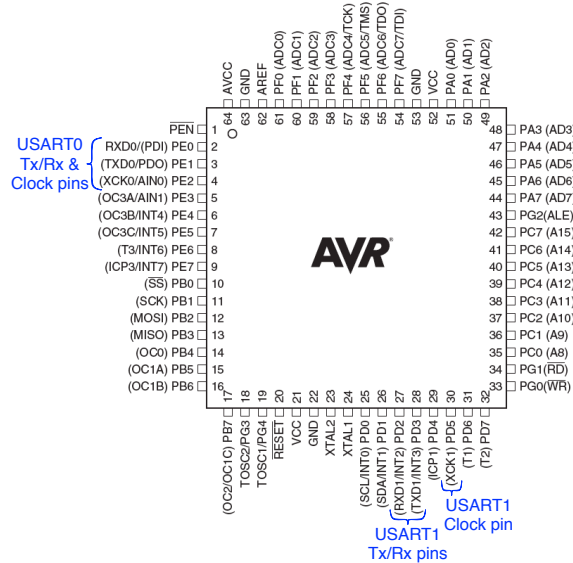
Figure 5.27: USART0 and USART1 pins.

5.5.3   Control and Status Registers

The functionality of each USART is defined by configuring the following:
- Synchronous vs. Asynchronous Mode
- Data Frame Format
- Baud Rate
- Transmitter and Receiver Enable
- Data Transmitted or Received Status
- Interrupts

These operations are controlled using *USARTn Control and Status Register A-C* (UCSRnA-C) shown in Figure 5.29. The following discusses these operations.

Synchronous vs. Asynchronous Mode

Table 5.9 shows the control bits for transmission mode, which is selected using *USARTn Mode Select* (UMSELn) bit in UCSRnC. When UMSELn=0, USARTn operates in asynchronous mode. If UMSELn=1, it operates in synchronous mode.
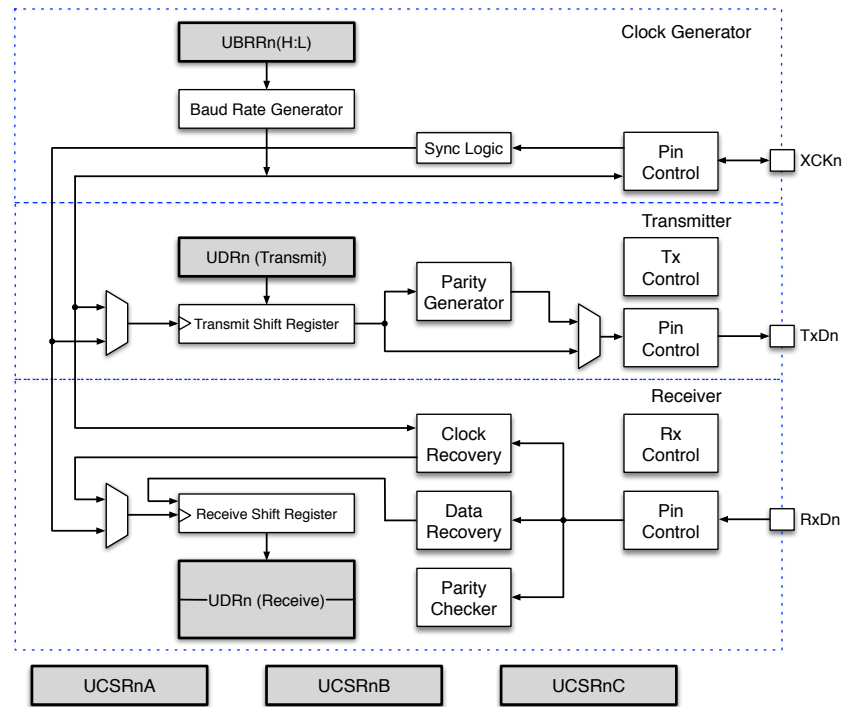
Figure 5.28: USART Block Diagram.

In synchronous mode, the XCKn pin shown in Figure 5.28 is used as either clock input for the Slave device or clock output for the Master device. The *USARTn Clock Polarity* (UCPOLn) bit in UCRSnC selects which XCKn clock edge is used for data sampling and which is used for data change. When UCPOLn=0, the data will be changed at rising XCKn edge and sampled at falling XCKn edge. If UCPOLn=1, the data will be changed at falling XCKn edge and sampled at rising XCKn edge.

In asynchronous mode, XCKn is not used. Instead, the Clock Recovery logic is employed to synchronize the internally generated Baud rate clock to the incoming asynchronous serial frames on the RxDn pin. Then, the Data Recovery logic samples the incoming bits.

Data Frame Format

The data frame format used by USARTn is set with the *USARTn Character SiZe* bits 2 through 0 (UCSZn2:0), the *USARTn Parity mode* bits 1 and 0

USART Control and Status Register A (UCSRnA)

7	6	5	4	3	2	1	0
RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn
R (0)	R/W (0)	R (1)	R (0)	R (0)	R (0)	R/W (0)	R/W (0)

Bit 7 - USART Receive Complete
Bit 6 - USART Transmit Complete
Bit 5 - USART Data Register Empty
Bit 4 - Frame error
Bit 3 - Data OverRun
Bit 2 - Parity Error
Bit 1 - Double USART Transmission Speed
Bit 0 - Multi-Processor Communication Mode

USART Control and Status Register B (UCSRnB)

7	6	5	4	3	2	1	0
RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n
R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R (0)	R/W (0)

Bit 7 - RX Complete Interrupt Enable
Bit 6 - TX Complete Interrupt Enable
Bit 5 - USART Data Register Empty Interrupt Enable
Bit 4 - Receiver Enable
Bit 3 - Transmitter Enable
Bit 2 - Character Size (combine with UCSZn1:0 in UCSRnC)
Bit 1 - Receive Data Bit 8
Bit 0 - Transmit Data Bit 8

USART Control and Status Register C (UCSRnC)

7	6	5	4	3	2	1	0
-	UMSELn	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn
R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (1)	R/W (1)	R/W (0)

Bit 7 - Reserved Bit
Bit 6 - USART Mode Select
Bit 5:4 - Parity mode
Bit 3 - Stop bit select
Bit 2:1 - Character size UCSZn2:0
Bit 0 - Clock Polarity

Figure 5.29: USARTn Control and Status Register A, B, and C.

(UPMn1:0), and the *USARTn Stop Bit Select* (USBSn) bit spread across
UCSRnB and UCSRnC as shown in Figure 5.29. Both the receiver and
transmitter use the same setting, and this should not be changed during
any ongoing communication.

Table 5.10 shows the control bit settings for the data frame format. The
UCSZn2:0 bits select the number of data bits in the frame. The UPMn1:0
bits enable and set the type of parity used. The selection between one or
two stop bits is done using the USBSn bit. The extra stop bit allows for
additional receive processing time, especially at high baud rates. Note that
the receiver ignores the second stop bit; therefore, a frame error will only
be detected in the cases where the first stop bit is zero.

Table 5.9: Control bits for transmission mode

Control bits	Bit	Meaning
UMSELn	0	Asynchronous mode
	1	Synchronous mode
UCPOLn	0	Data changes/sampled at rising/falling XCK edge
(synchronous)	1	Data changes/sampled at falling/rising XCK edge

Table 5.10: Control bits for Data Frame Format

Control bits	Bits			Meaning
UCSZn2:0	0	0	0	5-bit
	0	0	1	6-bit
	0	1	0	7-bit
	0	1	1	8-bit
	1	0	0	Reserved
	1	0	1	Reserved
	1	1	0	Reserved
	1	1	1	9-bit
UPMn1:0	0	0		No parity
	1	0		Even parity
	1	1		Odd parity
USBSn	0			1 stop bit
	1			2 stop bits

If a data frame contains 9 bits, *Transmit Data Bit 8* (TXB8n) and *Receive Data Bit 8* (RXB8n) hold the 8th bit. For example, during transmission, the last bit has to be first written to TXB8n before writing the lower 8 bits of data to UDRn. Similarly, during reception, the last bit has to be first read from RXB8n before reading the lower 8 bits of data from UDRn.

Baud Rate

The rate at which data is transmitted is called the *bit-rate*, measured in bits per second (bps). *Baud rate* refers to the rate at which symbols are transmitted, measured in symbols per second, and includes the synchronization bits, i.e., start bit and stop bit(s). For example, if 10-bit symbol is used per 8-bit character at a Baud rate of 9600, then this equates to 9600 symbols/second × 10 bits/symbol or a bit rate of 96000 bps.

The Baud rate is controlled using the *USARTn Baud Rate Register* (UBRRn) shown in Figure 5.28. The Baud Rate Generator loads the UBRR value and decrements it. When the count reaches zero, a clock is generated

and the UBRR value is reloaded. Moreover, the Baud rate is determined based on the transmission mode.

For the asynchronous normal mode, the Baud rate is given by the following equation:

$$\text{Baud Rate} = \frac{f_{CLK}}{16 \times (UBRR + 1)}, \tag{5.8}$$

where f_{CLK} is the system clock frequency and 16 is the Baud rate divider. Solving for UBRR leads to the following equation:

$$UBRR = \frac{f_{CLK}}{16 \times (\text{Baud Rate})} - 1 \tag{5.9}$$

For example, the required value for UBRR for a Baud rate of 2,400 baud and f_{CLK} of 16 MHz is given by

$$UBRR = \frac{16MHz}{16 \times 2400} - 1 = 416 = \text{0x01A0}. \tag{5.10}$$

The transfer rate can also be doubled by setting the *Asynchronous Double Speed Mode* (U2Xn) bit in UCSRnA, which reduces the Baud rate divider from 16 to 8 and results in UBRR value of 832 or 0x0340.

For the synchronous mode, the Baud rate is given by the following equation:

$$\text{Baud Rate} = \frac{f_{CLK}}{2 \times (UBRR + 1)} \tag{5.11}$$

Therefore, the UBRR value is given by

$$UBRR = \frac{f_{CLK}}{2 \times (\text{Baud Rate})} - 1 \tag{5.12}$$

Transmitter and Receiver Enable

Transmitter and Receiver of USARTn are enabled using *Receiver Enable* (RXENn) and *Transmitter Enable* (TXENn) bits in UCSRnB.

Data Transmitted or Received Status

In order to transmit a sequence of characters, the transmitter has to know whether one character has been successfully transmitted before the next character can be transmitted. This is indicated by the *USARTn Data Register Empty* (UDREn) flag in UCSRnA. If UDREn is one, the buffer is empty, and thus it is ready to be written. Writing a character to UDRn clears the UDREn flag. Another less common way is to monitor the *USARTn Transmit Complete* (TXCn) flag in UCSRnA, which is set when all the bits in the Transmit Shift Register shown in Figure 5.28 have been transmitted, and there is no new data in UDRn (Transmit).

Table 5.11: Status bits for transmission and reception

Status bits	Bit	Meaning
RXCn	0	Receive incomplete
	1	Receive complete
TXCn	0	Transmit incomplete
	1	Transmit complete
UDREn	0	UDRn (trasnmit buffer) full
	1	UDRn (transmit buffer) empty

Similarly, a reception of a new character in UDRn (Receive) sets the *USARTn Receive Complete* (RXCn) flag in UCSRnA. If the Receiver is disabled, i.e., RXENn=0, UDRn (Receive) will be flushed and consequently the RXCn bit will become zero.

Transmit and Receive Data Buffers

The UDRn register contains *USARTn Transmit Data Buffer* (TXBn) and *USARTn Receive Data Buffer* (RXBn). These two registers share the same I/O address as UDRn. This is shown in Figure 5.30. TXBn will be the destination for data written to UDRn, and data read from UDRn will be in RXBn. TXBn, and thus UDRn, can only be written when the UDREn flag in UCSRnA is set indicating it is empty. Data written to UDRn when the UDREn flag is not set will be ignored by the USARTn Transmitter. When data is written to UDRn, it will be written into Transmit Shift Register when it is empty. Then, the data will be serially transmitted on the TxDn pin. When data is received by the Receive Shift register, it will be written to UDRn. UDRn, and thus TXBn, can only be read when the RXC flag in UCSRnA is set indicating it is full.

USARTn I/O Data Register (UDRn)

7	6	5	4	3	2	1	0	
			RXBn[7:0]					UDRn (Read)
			TXBn[7:0]					UDRn (Write)
R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	
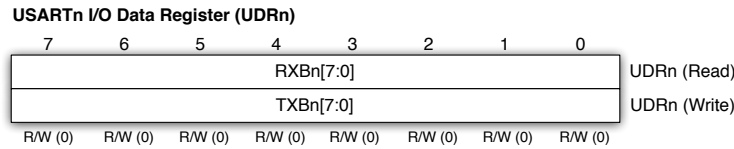
Figure 5.30: USARTn I/O Data Register (UDRn)

Interrupts

Both TXCn and RXCn can also be used to generate *Transmit Complete
interrupt* and *Receive Complete interrupt* by setting *TX Complete Inter-
rupt Enable* (TXCIE) and *RX Complete Interrupt Enable* (RXCIE) bits in
UCSRnB. For USART0, Transmit Complete interrupt and Receive Com-
plete interrupt are mapped to vector numbers 21 and 19 at addresses \$0028
and \$0024, respectively (see Table 5.1). The corresponding interrupts for
USART1 are mapped to vector numbers 33 and 31 at addresses \$0040 and
\$003C, respectively (see Table 5.1). Note that TXCn and RXCn flags are
cleared when their respective interrupt service routines are called.

The same is true for the UDREn flag, which can generate a *USARTn
Data Register Empty interrupt* by setting *USARTn Data Register Empty
Interrupt Enable* (UDRIEn) in UCSRnB. UDREn is automatically set after
a reset to indicate that the Transmitter is ready.

Error Reporting

There are three flags to indicate errors that can occur during transmis-
sion/reception. This is shown in Table 5.12.

Table 5.12: Status bits for Error Reporting

Status bits	Bit	Meaning
DORn	0	No error
	1	Data OverRun error
FEn	0	No error
	1	Framing error
UPEn	0	No error
	1	Parity error

The first is the *Data OverRun* (DORn) flag in UCSRnA, which is set
when UDRn (Receive) is full (i.e., contains two characters), there is a new

character waiting in the Receive Shift Register, and a new start bit is detected. This bit is valid until UDRn (Receive) is read.

The second is the *Framing Error* (FEn) bit in UCSRnA, which is set if the next character in UDRn (Receive) had a frame error when it was received. As mentioned before, a frame error occurs when the first stop bit of the next character in UDRn (Receive) is zero. This bit is valid until UDRn (Receive) is read. The FEn bit is zero when the stop bit of received data is one.

The third is the *Parity Error* (UPEn) bit in UCSRnA, which is set if the character in UDRn (Receive) had a parity error and the parity checking was enabled, i.e., UPM$n1 = 1$ (see Table 5.10). This bit is valid until UDRn (Receive) is read.

Note that all three of these bits are initialized to zeros on reset.

5.5.4   Programming Model

USART Initialization

USART has to be initialized before it can be used. The code shown below initializes USART0 to transmit on Port E, pin 0 (i.e., TxD0) and to receive on Port E, pin 1 (RxD0) with a Baud rate of 2,400 bps in asynchronous mode, double data rate, and a frame format with 8 data and 2 stop bits.

```
; AVR Assembly Code - USART0 Initialization
.include "m128def.inc"       ; Include definition file
.def  mpr = r16              ; Multi-purpose register

.ORG $0000                   ; Reset and Power On interrupt
     rjmp   INITIALIZE        ; Jump to initialization
.ORG $0024                   ; USART0, Rx complete interrupt
     rjmp   USART_Receive     ; Jump to ISR

INITIALIZE:
     ; Initialize stack
     ldi   mpr, high(RAMEND)
     out   SPH, mpr
     ldi   mpr, low(RAMEND)
     out   SPL, mpr

     ; Initialize I/O Ports
     ldi   mpr, (1<<PE1)      ; Set Port E pin 0 (RXD0) for input and
     out   DDRE, mpr          ; Port E pin 1 (TXD0) for output
```

```
    ; Initialize USART0
    ldi   mpr, (1<<U2X0)    ; Set double data rate
    out   UCSR0A, mpr       ;

    ; Set baudrate at 2400
    ldi   mpr, high(832)    ; Load high byte of 0x0340
    sts   UBRR0H, mpr       ; UBRR0H in extended I/O space
    ldi   mpr, low(832)     ; Load low byte of 0x0340
    out   UBRR0L, mpr       ;

    ; Set frame format: 8 data, 2 stop bits, asynchronous
    ldi   mpr, (0<<UMSEL0 | 1<<USBS0 | 1<<UCSZ01 | 1<<UCSZ00)
    sts   UCSR0C, mpr       ; UCSR0C in extended I/O space

    ; Enable both transmitter and receiver, and receive interrupt
    ldi   mpr, (1<<TXEN0 | 1<<RXEN0 | 1<<RXCIE0)
    out   UCSR0B, mpr       ;

    sei                     ; Enable global interrupt
```

Upon reset, the code jumps to the label INITIALIZE to perform initialization. Note that the code is set up to call the USART_RECEIVE interrupt service routine when a USART0 Receive Complete interrupt occurs. The first part of initialization is to set up the stack. Next, the TxD0 and RxD0 pins are a part of Port E (see Figure 5.27), and thus, their directions have to be set to output and input, respectively. This is done by setting bit 0 (PE0) and bit 1 (PE1) of DDRE to 1 and 0, respectively.

Initialization of USART0 starts by enabling the double data rate and setting the Baud rate. Turning on the double data rate is done by setting the U2X0 bit in UCSR0A. The Baud rate is set by loading 832 or 0x0340 on to UBRR. The data frame format is set by writing 011 into bits UCSZ02, UCSZ01, and UCSZ00, respectively, in UCSR0B and UCSR0C. At the same time, asynchronous mode and two stop bits are chosen by setting UMSEL0 to 0 and USBS0 to 1 in UCSR0C. The final piece of code enables both the transmitter and the receiver by setting bits RXEN0 and TXEN0 in UCSR0B, and the Receive Complete interrupt is turned by setting RXCIE0 to 1 in UCSR0B. Note that UBRR0H and UCSR0C are in the extended I/O space (see Table C.2), and thus the sts (*Store direct to SRAM*) instruction must be used. Finally, the global interrupt is enabled by setting the I-bit in SREG using the sei instruction.

Sending Data

The following subroutine can be used transmit one character on USART0.

```
; AVR Assembly Code - USART Transmit
USART_Transmit:
     sbis  UCSR0A, UDRE0     ; Loop until UDR0 is empty
     rjmp  USART_Transmit
     out   UDR0, r17         ; Move data to Transmit Data Buffer
     ret
```

The `sbis` instruction checks bit 5 of UCSR0A (URDE0 is define as the 5th bit in the `m128def.inc` include definition file) to see if UDR0 is empty. If the URDE0 bit is set, which means TXB0 or UDR0 (Transmit) is empty, the `rjmp` instruction is skipped and the data can be written to UDR0 for transmission.

Receiving Data

The following subroutine shows the receive operation, which is called when a Receive Complete interrupt occurs.

```
; AVR Assembly Code - USART Receive (interrupt driven)
USART_Receive:
     push  mpr               ; Save mpr
     in    r17, UDR0         ; Get data from Receive Data Buffer
     ...                     ; Do something with data
     pop   mpr               ; Restore mpr
     reti
```

Note that the `reti` instruction is needed to return from the interrupt service routine and reenable the global interrupt. Moreover, the above code assumes the `mpr` register will be used within the interrupt service routine, and thus it is saved to and restored from the stack.

5.6 Analog-to-Digital Converter

Under Construction!!!

5.7 SPI Bus Protocol

Under Construction!!!

5.8 TWI

Under Construction!!!

5.9 Analog Comparator

Under Construction!!!

Chapter 6

Embedded C

Contents

6.1 Introduction

As the title of this book suggests, *assembly language programming* is one of the major topics of this book. However, despite the many advantages of assembly programming, writing programs with any level of sophistication requires a *high-level programming language*. Since an assembly language is tied to a particular processor and the way it works, assembly language programming can be difficult to master and requires you to learn another assembly language when you change to a different microcontroller family. For this reason, there are C compilers for microcontrollers that can, unlike assembly, abstract away the lower-level details of what a processor does to execute your programs. Using C, you can write software much faster, and create programs that are much easier to understand and maintain than assembly language programs. In addition, C works reasonably close to the

processor, which allows programmers to generate code that requires less memory space and runs faster than other high-level languages, such as Java.

This chapter discusses *Embedded C*, which adds additional functionality to the C programming language providing portability across different embedded systems. These additions include direct access to hardware (e.g., I/O ports), interrupt handling, and even embedding assembly code within C. Therefore, Embedded C offers both machine-independent and machine-dependent programming extensions, providing both the power of general-purpose programming and a detailed interface to hardware. This chapter assumes the audience has experience with general C programming, and is familiar with AVR I/O features discussed in Chapter 5. Thus, the emphasis will be on the additional features available for programming embedded systems.

There are several C compilers for AVR, which include CodeVisionAVR by HP InfoTech, AVR IAR by IAR Systems, and AVR-GCC, which is the open source software development tool for AVR microcontrollers. These compilers basically differ on how they handle machine-dependent details, such as I/O operations and embedding assembly into C. This chapter discusses Embedded C in the context of AVR-GCC, which is the C compiler used by the Atmel Studio Integrated Development Environment (IDE) (see Appendix F).

Section 6.2 provides a brief tutorial on C programming. This is followed by a discussion of AVR I/O operations in Section 6.3. Section 6.4 discusses how to access the Program Memory, Data Memory, and EEPROM of AVR. Programming the AVR interrupt facility is discussed in Section 6.5. Section **??** explains the importance and correct usage of `volatile` in Embedded C programs. Finally, Section 6.6 discusses embedding assembly into C.

6.2   A Quick Primer on C Programming

Under Construction!!!

6.3   I/O Operations in AVR

Under Construction!!!

6.4 Accessing Program Memory, Data Memory, and EEPROM in AVR

Under Construction!!!

6.5 Using Interrupts in AVR

Under Construction!!!

6.6 Mixing C and Assembly

Under Construction!!!

6.7 TekBot Programming Using Embedded C

Under Construction!!!

Chapter 7

Digital Components

Contents

7.1 Introduction

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,

191

Figure 7.1: Digital components in the AVR microarchitecture.

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.
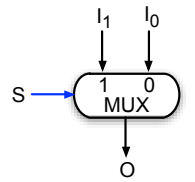
(a) Logic symbol.

Input			Output
I_1	I_0	S	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.
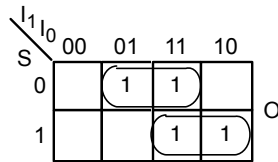
(c) K-map.
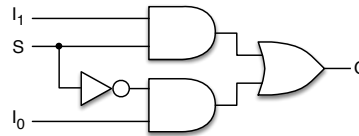
(d) Logic diagram.

Figure 7.2: 2-to-1 MUX.

7.2   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 S is used to select from either input I_0 or I_1, which then appears on the output O.

In microarchitecture design, a MUX is typically used to select one of the 2^k inputs 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.

The logic equations for 2-to-1, 4-to-1, and 8-to-1 MUXs are shown below and Figure 7.4 shows the logic symbol and the logic diagram for 4-to-1 MUX:

2-to-1 MUX: $O = S'I_0 + SI_1$

4-to-1 MUX: $O = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$

8-to-1 MUX: $O = S_2'S_1'S_0'I_0 + S_2'S_1'S_0I_1 + S_2'S_1S_0'I_2 + S_2'S_1S_0I_3$
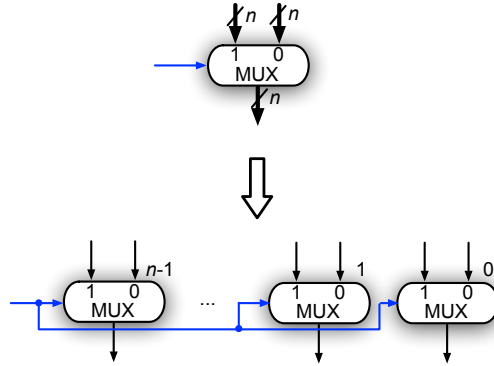
Figure 7.3: n-bit 2-to-1 Multiplexer.
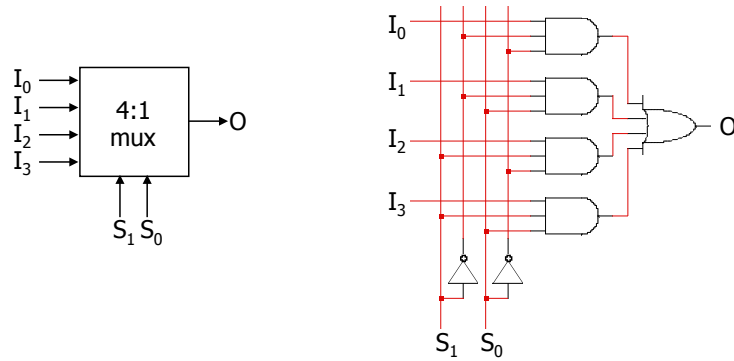
Figure 7.4: Example of 4-to-1 MUX.

$$+ \; S_2S_1{}'S_0{}'I_4 \; + \; S_2S_1{}'S_0I_5 \; + \; S_2S_1S_0{}'I_6 \; + \; S_2S_1S_0I_7$$

As can be seen from the above logic equations and Figure 7.4, the number of inputs for AND and OR gates grows as k grows. For example, the number of inputs to the AND gate and the OR gate for 8-to-1 MUX are four and eight, respectively. This causes a problem where the switching delay increases with k, referred to as the *fan-in problem*. The limit on fan-in (i.e., 2^k inputs) is typically four.

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
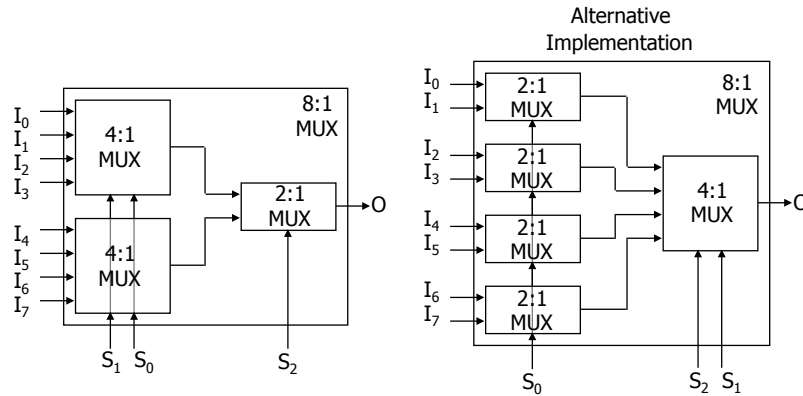
Figure 7.5: Cascaded MUXs.

select lines $S_2S_1S_0$ are used to control the MUXs. For the implementation on the left, the most significant input select bit (i.e., S_2) selects either the first or second half of the inputs, and the input select bits S_2S_0 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., S_2S_1) to select one of the four two-bit inputs, and the input select bit S_0 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 2^k outputs based on the k-bit pattern on the select lines. DEMUX is complementary to MUX, and both are often used to merge and separate information.

7.3   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 different. 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 a k-input, 2^k-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

Input			Output			
E	S_1	S_0	O_3	O_2	O_1	O_0
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.
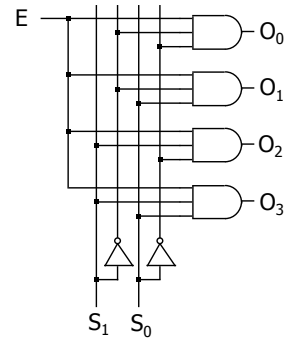
(b) Logic diagram.

Figure 7.6: 2-to-4 decoder.

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 O_i is 1 only when the input $S_1 S_0$ 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 a 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 $A_3 A_2 A_1 A_0$ 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 7.6.1.

7.4   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 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.
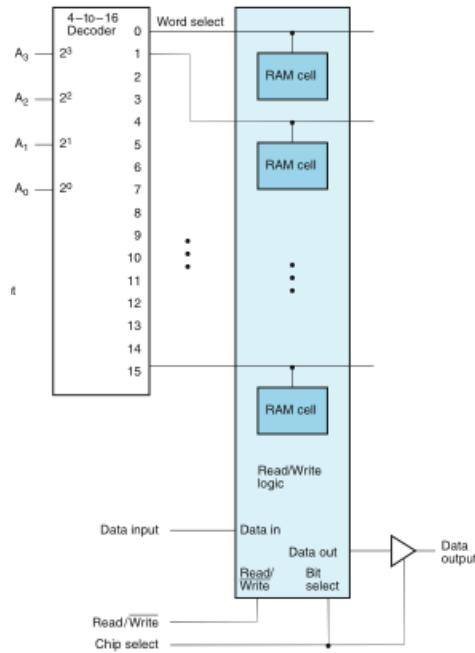
Figure 7.7: Decoder front-end for memories.

7.4.1 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 both inputs are not 1s.

Alternatively, an S-R latch can also be constructed using NAND gates. Figure 7.10 shows an S-R latch based on NAND gates. As can be seen, the main difference between the two implementations is that their inputs are

(a) NOR implementation.
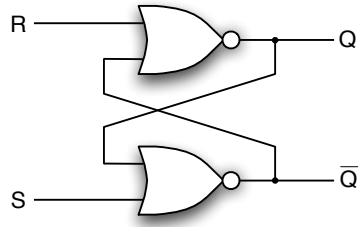
S	R	Q
0	0	No change
0	1	0
1	0	1
1	1	Not allowed

(b) Function table.
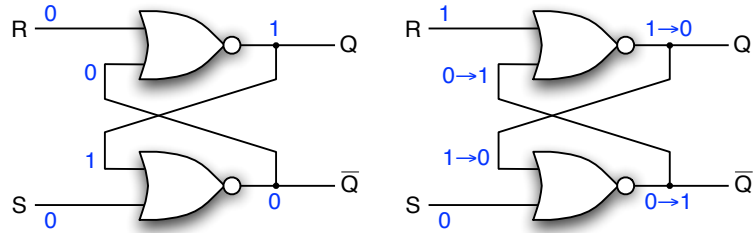
Figure 7.8: S-R latch.

(a) S=0, R=0 (no change); initially (b) S=0, R=1 (reset); initially Q=1.
Q=1.
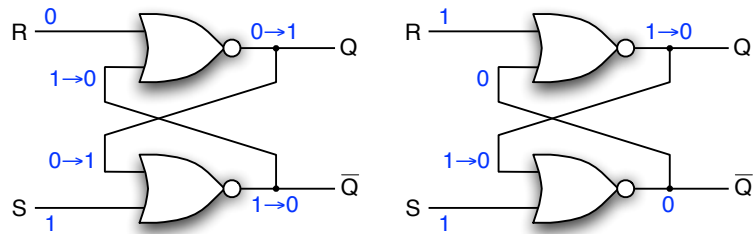
(c) S=1, R=0 (set); initially Q=0.   (d) S=1, R=1 (not allowed); ini-
tially Q=1.

Figure 7.9: Operations of S-R latch.

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
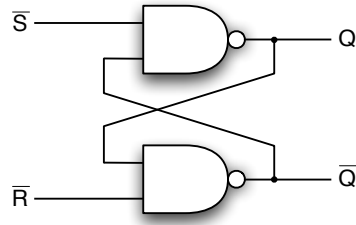
(a) NAND implementation.

$\overline{S}$	$\overline{R}$	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.

additional control input that determines when the state of the latch can be changed. Figure 7.11 shows an S-R latch with *enable*. The two NAND gates together with the control signal (C) determine whether or not set-reset operations are activated. This allows an S-R latch to be written to only when a certain condition is true.
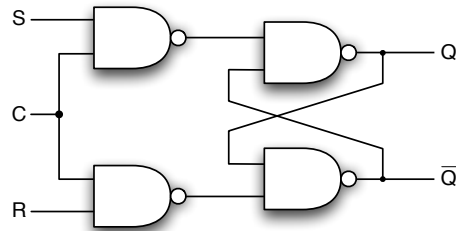
(a) Circuit using $\overline{S}$-$\overline{R}$ latch (i.e., NAND implementation).

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.
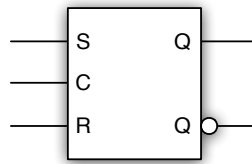
(c) Logic symbol.

Figure 7.11: S-R latch with enable.

(a) Circuit using $\overline{S}$-$\overline{R}$ latch.

D	C	Q
0	1	0
1	1	1
×	0	No change

(b) Truth table.

(c) Logic symbol.

Figure 7.12: D latch with enable.

As discussed before, inputs to an 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 a clock signal.

7.4.2   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, which 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 slave section is basically the same as the master section except that it is

(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.

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 Master D flip-flop to be latched onto the Slave D flip-flop, and at the same
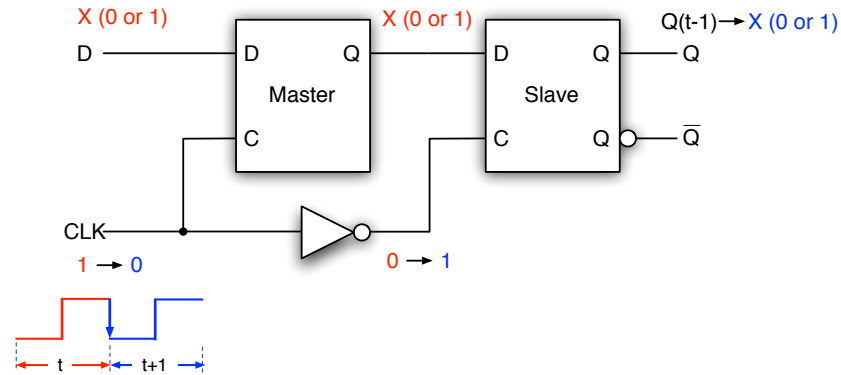
Figure 7.14: Functional behavior of a negative-edge-triggered D flip-flop.

time prevents any new signal from being latched onto the Master flip-flop.
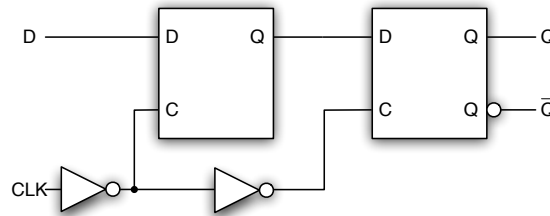
(a) Circuit using D latches.

D	CLK	Q
0	↑	0
1	↑	1
×	0	No change
×	1	No change

(b) Function table.
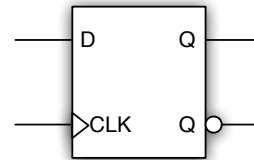
(c) Logic symbol.

Figure 7.15: Positive-edge triggered D-FF.

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) Logic diagram.   (b) Logic symbol.
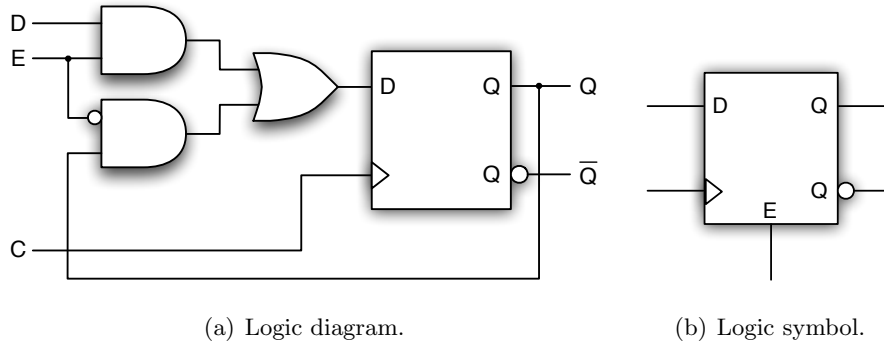
Figure 7.16: Positive-edge triggered D flip-flop with high enable.

7.4.3 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 characteristic, 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-flop will maintain the last value stored.

7.5 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 as simple as holding a set of bits or as complicated as performing a variety of functions.

7.5.1 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
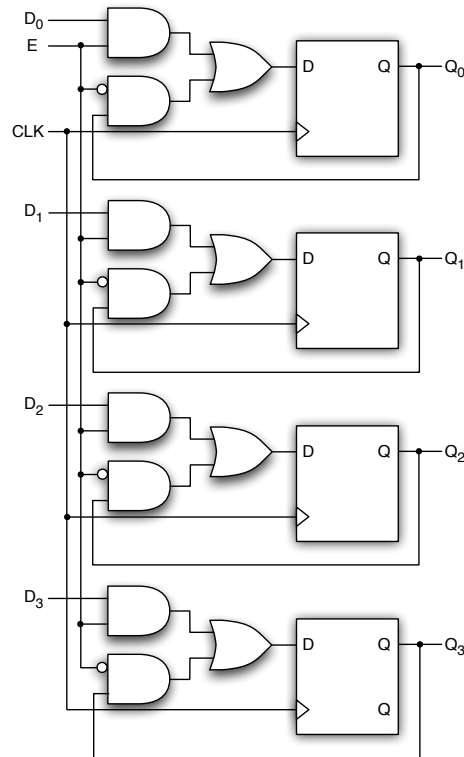
Figure 7.17: Register with load enable.

either the external input D_i or the flip-flop's current output Q_i.

7.5.2   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.

(a) Left (down).

(b) Right (up).
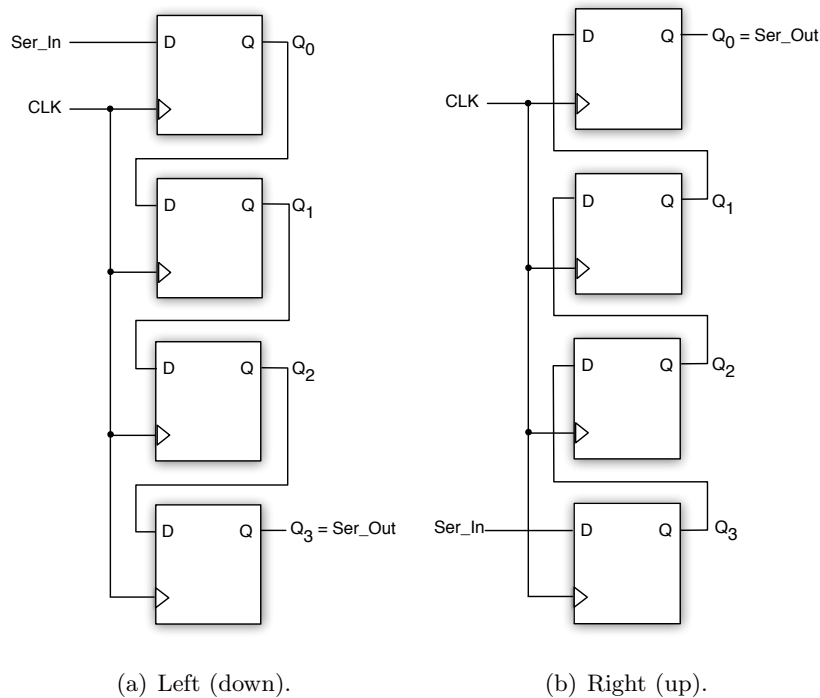
Figure 7.18: 4-bit shift register.

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 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.

7.6 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 7.4 and 7.5.
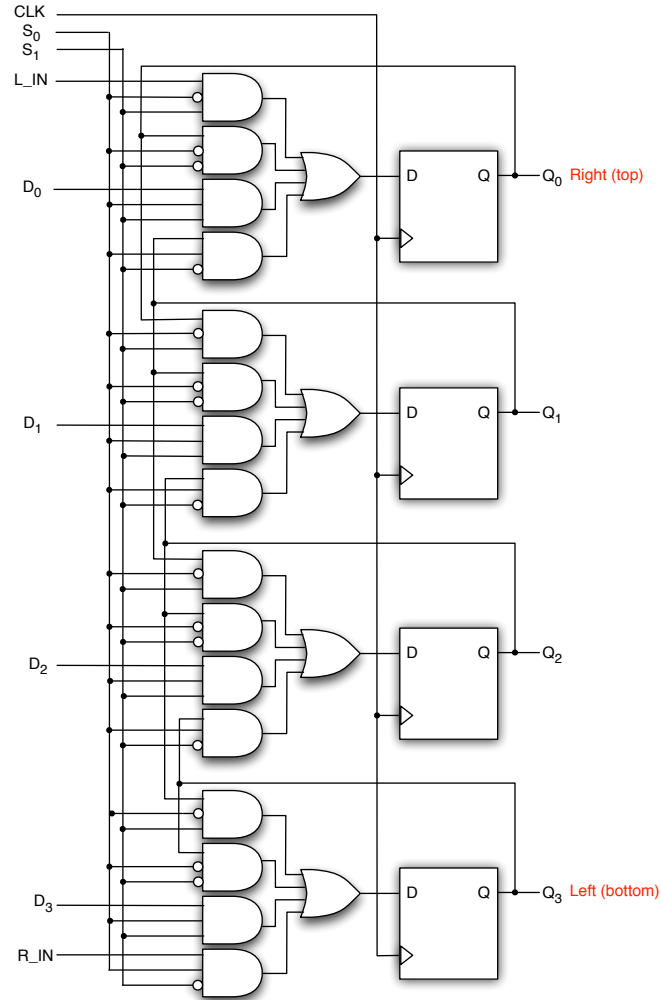
(a) Logic diagram.

Function	Control Input		Next State			
	S_1	S_0	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_1
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.
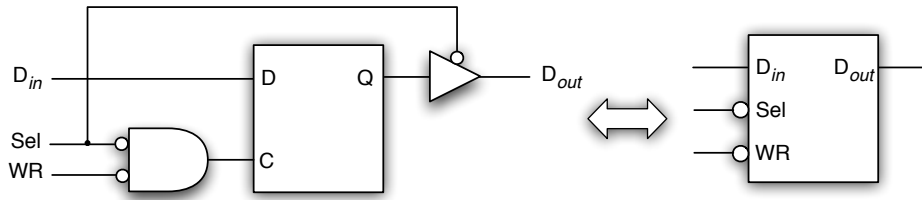
Figure 7.20: Logic diagram of a SRAM cell.

7.6.1 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 7.3, 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 with $\overline{CS}$ are asserted to make the SRAM operational.
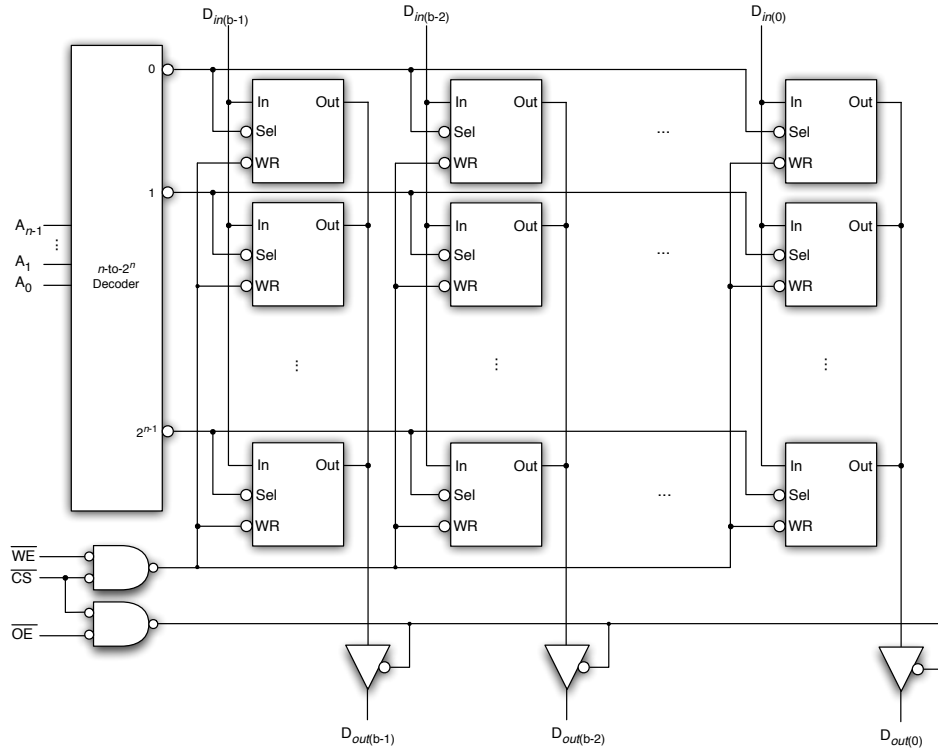
(a) Logic Diagram.
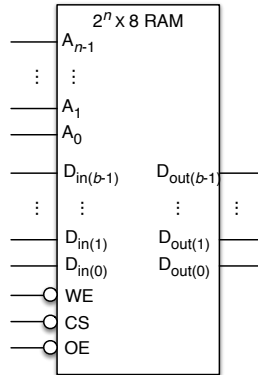
(b) Logic Symbol.

Figure 7.21: 2^n by b-bit SRAM structure.
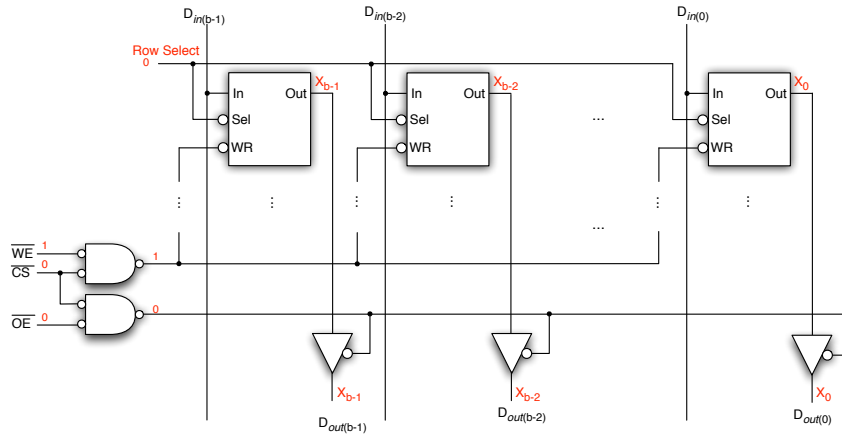
Figure 7.22: Read operation.

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
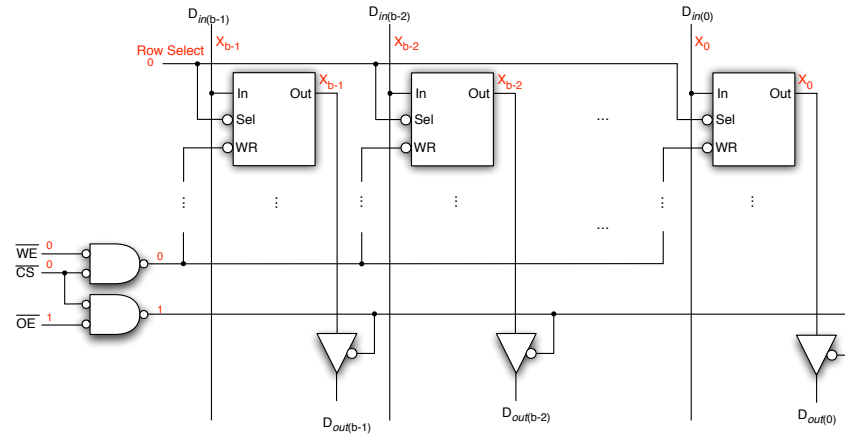
Figure 7.23: Write operation.

their 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 books on computer architectures, such as 'Computer Organization and Design: The Hardware/Software Interface' by Patterson and Hennessy (Elsevier Morgan Kaufmann, 2009).

7.6.2   Building Bigger and Wider Memories

Memory often needs to be expanded to accommodate larger applications and data. For example, ATmega128 AVR processor has $4,966 \times 8$-bit internal SRAM, and is expandable to $64K \times 8$-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 $256K \times 8$-bit memory using four $64K \times 8$-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., $A_{15} - A_0$)
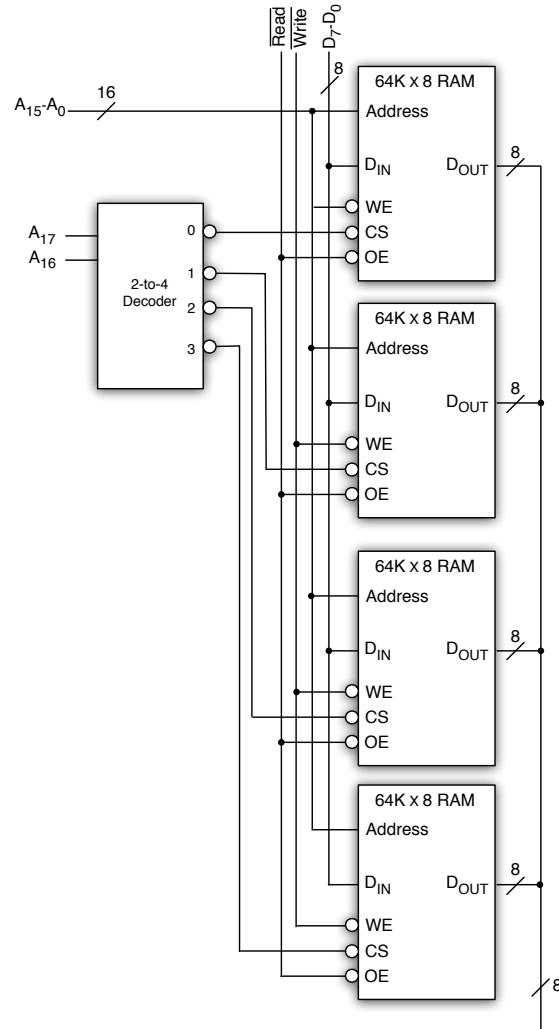
Figure 7.24: Implementing 256K×8-bit RAM using 64K×8-bit RAMs.

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 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
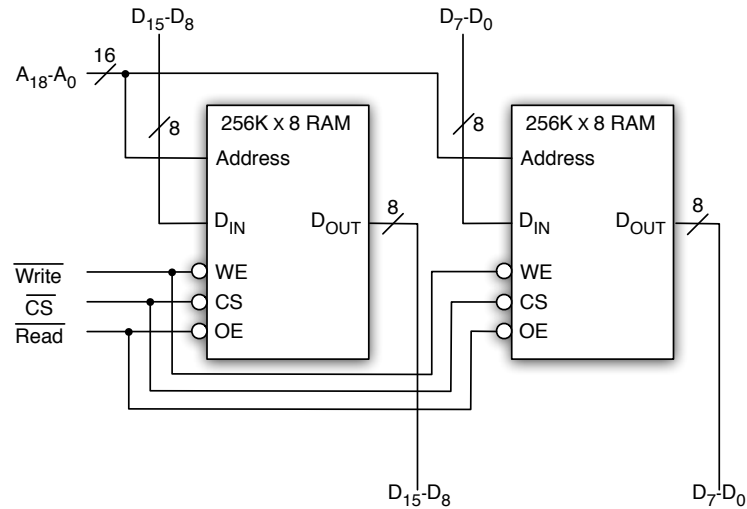
Figure 7.25: Implementing 256K×16-bit RAM using 256K×8-bit RAMs.

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.

7.7   Register File

A *register file* consists of a 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 is 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,
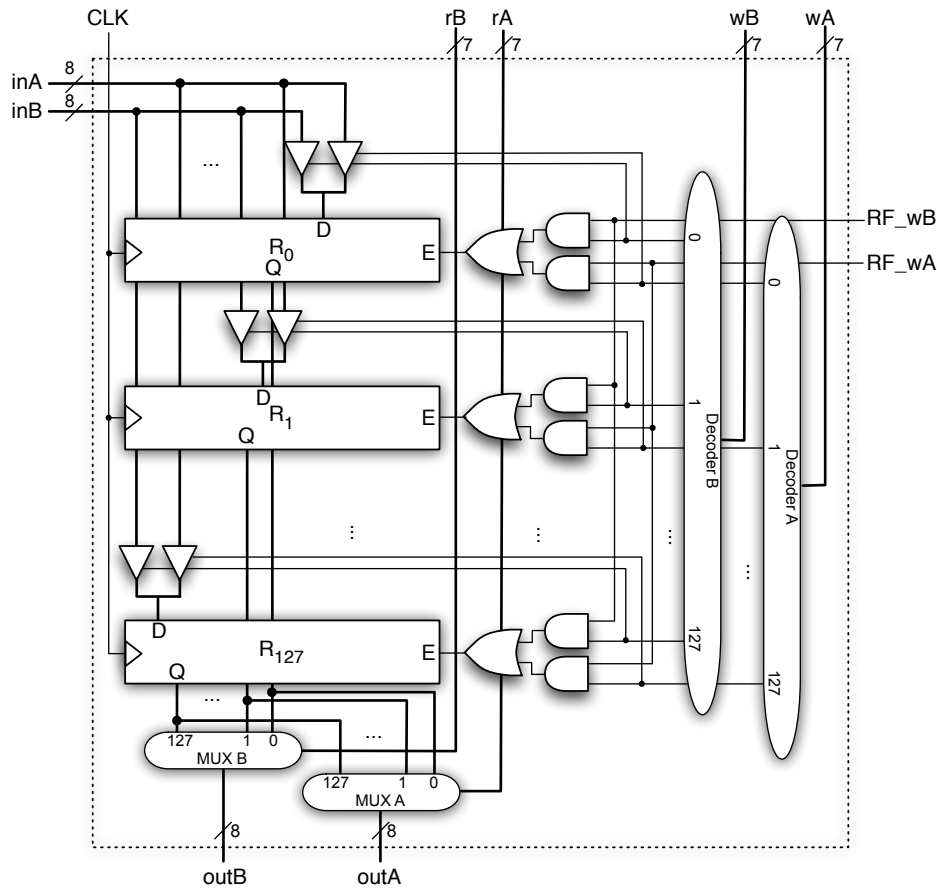
Figure 7.26: Internal structure of the 2 read-port, 2 write-port register file.

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 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 be the same. Finally, the signals asserted from the decoders together with RF_wA and RF_wB determine whether or not registers latch their inputs.

7.8   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 requires an entire chapter on its own. A detailed discussion on the ALU design is provided in Chapter 9.

AVR instruct

Chapter 8

Atmel's AVR 8-bit Microcontroller: Part 3 - Microarchitecture

Contents

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. 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 transfer and processing operations. A *Control Unit* (CU), on the other hand, is a logic block that determines the sequence of data transfer or processing operations to be performed by

the datapath. An ISA influences many aspects of how a microarchitecture is implemented. Moreover, a given ISA can be implemented with different microarchitectures based on the design requirements or shifts in technology.

This chapter presents a microarchitecture design based on the AVR ISA discussed in Chapter 4, referred to as *pseudo-AVR microarchitecture*. The objectives of this chapter are to (1) understand the basic components needed and how they are interconnected to implement a datapath to perform data transfer and processing operations, and (2) design a Control Unit to provide appropriate signals to the various components to control the operations within the datapath.

You may be wondering why we are discussing microarchitecture design when most of you will probably never design a processor. 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 it. Second, knowing the inner workings of a processor allows a better understanding of how the overall computer system works. For example, few people may design processors but many people design hardware systems that contain processors, e.g., embedded systems. Third, knowing how processors work makes you a better programmer. Finally, some of you may actually 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 a 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 that will vary depending on the instruction being encoded. 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. Both Rd and Rr fields can specify any one of the 32 General Purpose Registers (GPRs). Since this is a 2-address

- - - -	- - r d	d d d d	r r r r

(a) Two-operand format.

- - - -	K K K K	d d d d	K K K K

(b) Immediate format.

- - - -	- A A d	d d d d	A A A A

(c) I/O format.

- - - -	- - - d	d d d d	- - - -

(d) One-operand format.

- - - -	k k k k	k k k k	k k k k

- - q -	q q - d	d d d d	- q q q

(e) Displacement format.

- - - -	- - k k	k k k k	k - - -

(f) PC-relative format.

- - - -	- - - k	k k k k	- - - k
k k k k	k k k k	k k k k	k k k k

(g) Direct format.
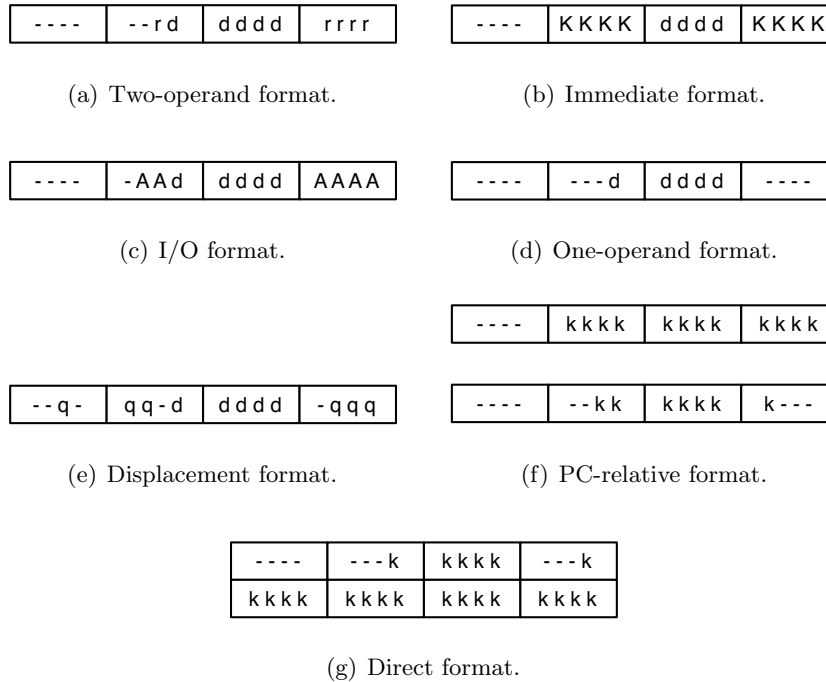
Figure 8.1: AVR instruction formats.

instruction format (see Chapter 2), the register identifier Rd also specifies the destination register. This format is used to encode any binary operations involving two registers, e.g., ADD (*Add two registers*), CP (*Compare*), MOV (*Copy register*), and AND (*Logical AND registers*).

Instead of having both source operands in registers, one of the operands 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 (Rd) field for this instruction format 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. Examples of instructions that follow this format are LDI (*Load immediate*), ANDI (*Logical AND register and constant*), and ORI (*Logical OR register and constant*).

Figure 8.1(c) shows the *I/O format*, where the 6-bit I/O register identifier AA AAAA (A) field specifies one of the 64 I/O registers. There are two instructions that use this format, IN (*In port*) and OUT (*Out port*). The 5-bit

d dddd (Rd) field specifies either the source register for the OUT instruction or the destination register for the IN instruction.

Unary operations that only require one source register are specified using the *one-operand format* shown in Figure 8.1(d). These instructions include INC (*Increment*), DEC (*Decrement*), COM (*One's complement*), LSR (*Logical shift right*), ROR (*Rotate right through carry*), etc. The 5-bit d dddd (Rd) field specifies any one of the 32 GPRs. The one-operand format is also used by LD (*Load indirect*) and ST (*Store indirect*) instructions as well as their auto-increment/decrement variations, where d dddd represents Rd for LD and Rr for ST.

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 (*Load indirect with displacement*) and STD (*Store indirect with displacement*). The 6-bit q qq qqq (q) field represents an unsigned offset ($0 \leq q \leq 63$), which is added to one of the *address registers* (ARs) Y or Z, but not X. The 5-bit d dddd (Rd) field specifies any one of the 32 GPRs 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. These are shown in Figure 8.1(f). The signed 12-bit or 7-bit displacement field kkkk kkkk kkkk (k12) or kk kkkk k (k7) is added to PC+1 to generate a branch target address with a range of $-2048 \leq k \leq 2047$ 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 (*Relative jump*) and RCALL (*Relative subroutine call*) instructions. On the other hand, the 7-bit PC-relative format is used exclusively by conditional branch instructions, such as BREQ (*Branch if equal*), BRLT (*Branch if less than*), and BRGE (*Branch if greater or equal*).

Figure 8.1(g) shows the *direct format*, which is used only by 32-bit instructions CALL (*Direct subroutine call*) and JMP (*Direct jump*). The 16-bit target address field kkkk kkkk kkkk kkkk (k16), which is the second 16-bit portion of a 32-bit instruction, is the same size as the PC. This means that the target address of this direct format instruction can be anywhere within the 64K-word 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 words of Program Memory, these bits are all zeros.
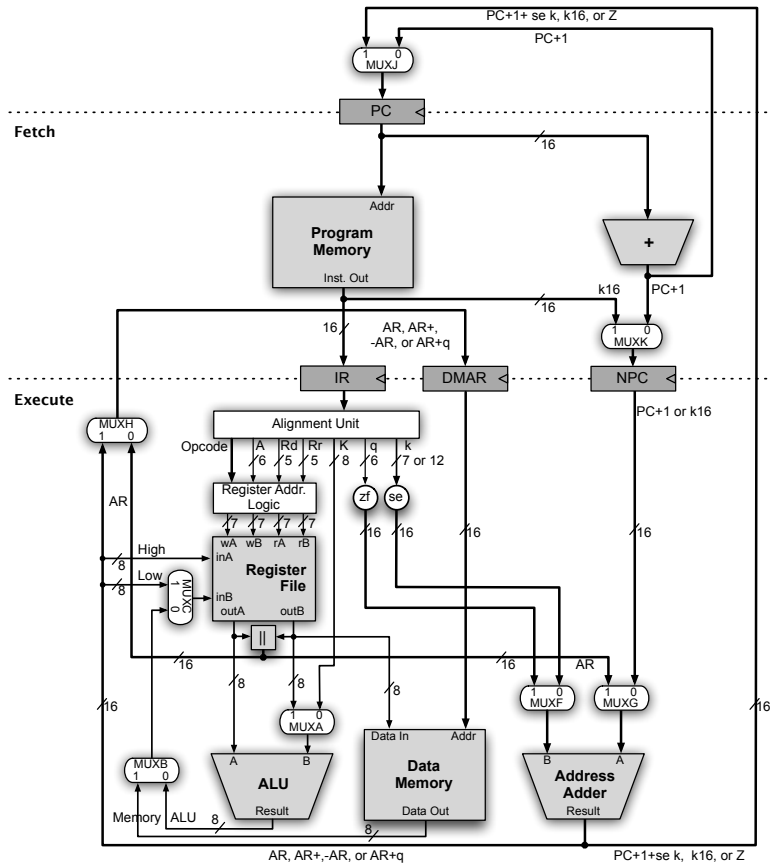
Figure 8.2: Basic 2-stage datapath

8.3   Components in the Basic Datapath

Figure 8.2 shows the *basic 2-stage datapath* 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 microoperations required for 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. In Section 8.5, we will discuss an enhanced datapath that can execute more complex AVR instructions.

8.3.1   Special-Purpose 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)
    - Data Memory Address Register (DMAR)
    - Next PC (NPC)

The *Program Counter* (PC), 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 latched at the end of the clock cycle so that instructions are fetched sequentially from the Program Memory. Certain instructions, such as branches, jumps, and subroutine calls/returns, interrupt the sequencing by updating the PC with a target address.

The *Instruction Register* (IR) holds an instruction fetched during the Fetch stage. The IR latches an instruction at the end of the Fetch cycle and holds its information during one or more Execute cycles.

The *Data Memory Address Register* (DMAR) holds an address based on the contents of the X-, Y-, or Z-register, and is used to access the Data Memory.

The *Next PC* (NPC) register holds either PC+1 to be added with a displacement k7 or k12 to implement PC-relative branches, or k16 to implement direct jumps.

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 (see Section 7.6). The main difference is that the Program Memory holds 16-bit instructions, whereas the Data Memory holds 8-bit data. Both memories are controlled by read enable signals: PM_read for the Program Memory and DM_read for the Data Memory. The Data Memory also has the write enable signal DM_write.

8.3.3   Sign-Extension and Zero-Fill Units

Sign-extension is required whenever the number of bits used to represent a *signed* value needs to match an input of a component. For example, PC-relative displacement kk kkkk k (k7) has to match the number of bits required

(a) Program Memory.   (b) Data Memory.
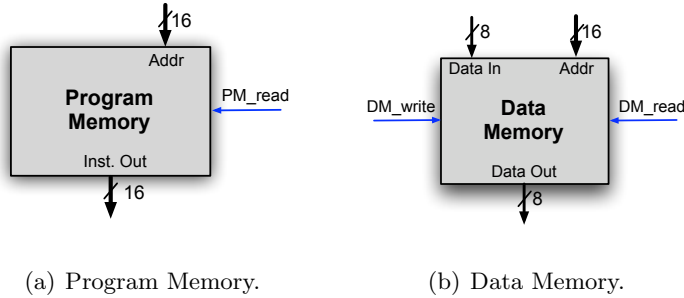
Figure 8.3: Program and Data Memories.

(a) n-bit to m-bit sign extension.

(b) 12-bit k sign-extended to 16-bit.
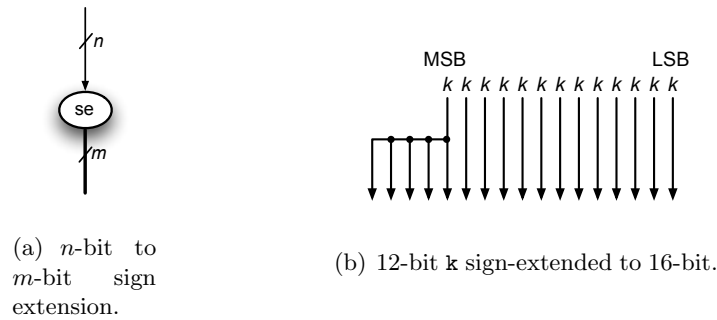
Figure 8.4: Sign-extension unit.

by the 16-bit Address Adder shown in Figure 8.2. Figure 8.4 shows a sign-extension (se) unit. As shown in Figure 8.4(a), the se unit takes an n-bit number as input and simply duplicates the sign (i.e., MSB) to generate an m-bit number. This allows an n-bit signed number to become an m-bit signed number. For example, 0010111_2, which represents 23 in decimal, sign-extended to 16 bits becomes 0000000000010111_2. The number 1101001_2 represents -23 in decimal, which when sign extended to 16 bits becomes 1111111111101001_2. Figure 8.4(b) shows the sign-extension requirement for 12-bit PC-relative displacement k12 (see Section 8.3.5 for an explanation of 12-bit PC-relative displacement k12).

Similar to sign-extension, *zero-fill* is needed whenever the number of bits used to represent an *unsigned* value needs to match with the number of bits required by a component. For example, the unsigned displacement q qq qqq (q) used in displacement format instructions has to match the number of bits required by the 16-bit Address Adder. Figure 8.5 shows the zero-fill

(a) n-bit to m-bit zero fill.

(b) 6-bit q zero-filled to16-bit.
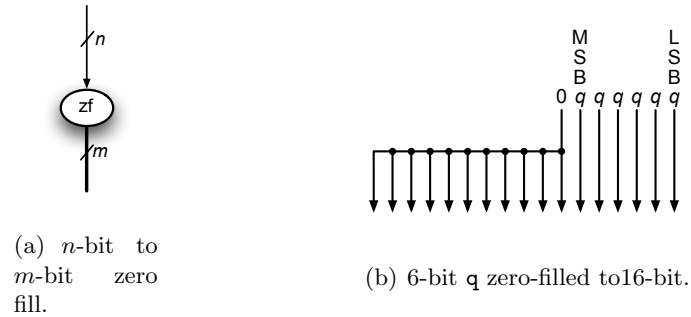
Figure 8.5: Zero-fill unit.
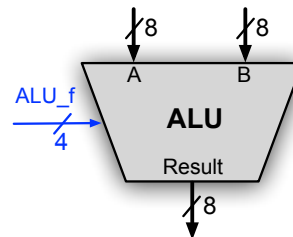
Figure 8.6: Arithmetic and Logic Unit (ALU).

(zf) unit. As shown in Figure 8.5(a), a zf unit takes an n-bit number as input and simply pads zeros and generates an m-bit number. This allows an n-bit unsigned number to become an m-bit unsigned number. For example, 101001_2, which represents 41 in decimal, when zero-filled to 16 bits becomes 0000000000101001_2. 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 8-bit *Arithmetic and Logic Unit* (ALU) shown in Figure 8.6 is the workhorse of the microarchitecture. It takes two 8-bit inputs and performs an operation defined by the 4-bit control signals ALU_f. The ALU can also handle instructions that require only one operand, such as NEG (*Two's complement*), INC (*Increment*), and DEC (*Decrement*). Table 8.1 shows all the operations provided by the ALU. Note that the operations in Table 8.1 are

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 A	0100
$Result = A - 1$	Decrement A	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 B	1100

sufficient to handle virtually all AVR instructions requiring arithmetic and logic operations. The only exception is the multiplication operation.

A detailed discussion on the design of an ALU is provided in Chapter 9.

8.3.5 Alignment Unit

Figure 8.7 shows the *Alignment Unit*, which extracts the various fields from different instruction formats needed by the Execute stage. These fields consist of the following:

- Rr - 5-bit source register identifier indicated by bits r rrrr.
- Rd - 5-bit destination register identifier indicated by either bits d dddd or dddd. 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.
- q - 6-bit displacement indicated by bits q qq qqq.
- k7 or k12 - Either 7-bit or 12-bit PC-relative displacement indicated by bits kk kkkk k or kkkk kkkk kkkk, respectively.
- A - 6-bit I/O register address indicated by bits AA AAAA.

The fields A, Rd, and Rr are fed to the *Register Address Logic* (RAL) so that 5-bit source/destination register identifiers and 6-bit I/O addresses can be converted to 7-bit addresses to read/write from/to the Register File. The functionality of RAL is more involved and a detailed discussion is deferred

----	--rd	dddd	rrrr	Two-operand
----	KKKK	dddd	KKKK	Immediate
----	-AAd	dddd	AAAA	I/O
----	---d	dddd	----	Single-operand
--q-	qq-d	dddd	-qqq	Displacement
----	kkkk	kkkk	kkkk	12-bit PC-relative
----	--kk	kkkk	k---	7-bit PC-relative

Alignment

A /6   Rd /5   Rr /5   K /8   q /6   k /7 or 12
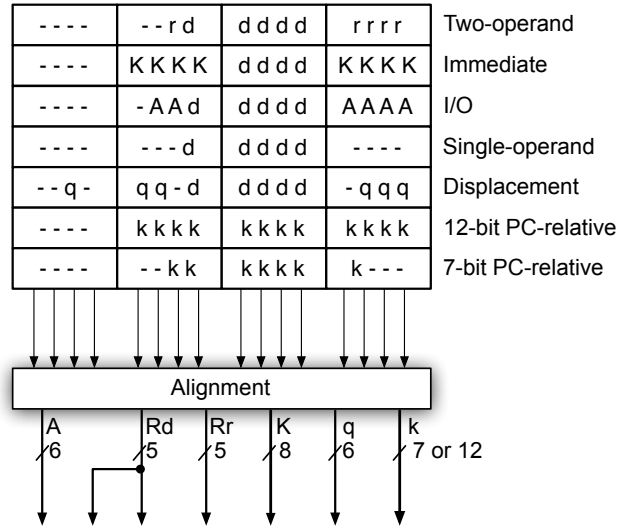
Figure 8.7: Alignment Unit.

until Section 8.6.3. The K field provides an immediate value as the second operand to the ALU. The q and k fields, after zero-filled and sign-extended, respectively, are fed to the 16-bit Address Adder as a displacement to an address register (either Y- or Z-register) and PC+1, respectively.

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 they can be accessed using load and store as well as IN and OUT 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 7.7). 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, respectively. 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_wB. Writing a 16-bit result 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
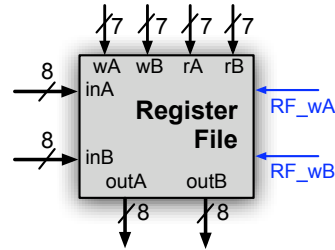
Figure 8.8: Two read-port, two write-port Register File.
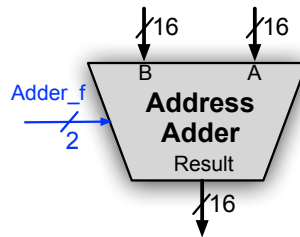
Figure 8.9: 16-bit Address Adder.

data to inA and inB, respectively, and asserting both RF_wA and RF_wB.

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 indirect jumps, and effective addresses for operands using the X-, Y-, or Z-register. 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
$Result = A + 1$	Increment	01
$Result = A - 1$	Decrement	10
$Result = A$	Move A	11

There are several possible address calculations. A move operation (i.e., Move A) simply passes the address on input A to the output. This operation does not require the second input (i.e., input B) to the Address Adder (see Section 9). Add operations on addresses consist of PC+1 added with either the 7-bit (k7) or 12-bit displacement (k12) (i.e., PC-relative branches and jumps) and either Y- or Z-register added with the 6-bit displacement (q) (e.g., LDD and STD. Increment and decrement operations can also be performed on any one of the address registers.

8.3.8   Multiplexers

As discussed in Section 7.2, 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 operations are performed within the datapath.

MUXA is an 8-bit 2-to-1 multiplexer that chooses between the 8-bit content of a register from the outB port of the Register File and the 8-bit constant K, and directs it to input B of the ALU. This allows 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, K).

MUXB together with *MUXC*, which are both 8-bit 2-to-1 multiplexers, allow the output from either the ALU or 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 wB and asserting RF_wB. In addition, both the upper and lower bytes of the Address Adder output can be written at the same time to the Register File (with MUXC used to select the lower byte), based on the register identifiers Rd+1 and Rd on wA and wB, respectively, and asserting both RF_wA and RF_wB signals.

MUXF is a 16-bit 2-to-1 multiplexer that selects either 6-bit q zero-filled to 16 bits (zf q) or 7-/12-bit displacement k sign-extended to 16 bits (se k). The 6-bit displacement will be added to either Y- or Z-register to generate an effective address for an operand in Data Memory, while the 7- or 12-bit displacement will be added to PC+1 to generate a branch target address. *MUXG* is a 16-bit 2-to-1 multiplexer that selects an address from either the NPC (i.e., PC+1 or k16) or an address register (AR), which will then be either added with a displacement (i.e., zf q or se k) or simply passed through the Address Adder to generate a target address (i.e., k16 or Z).

MUXH is a 16-bit 2-to-1 multiplexer that selects either an address register concatenated from a pair of 8-bit registers (AR) or an address register concatenated from a pair of 8-bit registers and then modified by the Ad-
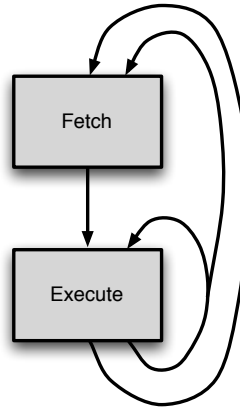
Figure 8.10: Fetch and Execute stages.

dress Adder (-AR, AR+, or AR+q). These addresses are latched to DMAR and then used to access the Data Memory in the subsequent cycle.

MUXJ is a 16-bit 2-to-1 multiplexer that chooses between PC+1 and a 16-bit target address generated from the Address Adder, which can be PC+1+se k, k16, or Z.

Finally, *MUXK* is a 16-bit 2-to-1 multiplexer that chooses between PC+1 and 16-bit target address k16 for direct jumps.

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 instruction, the number of 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. The number of Execute cycles required for each AVR instruction is defined in Appendix A.

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 cycles, 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 8.8.
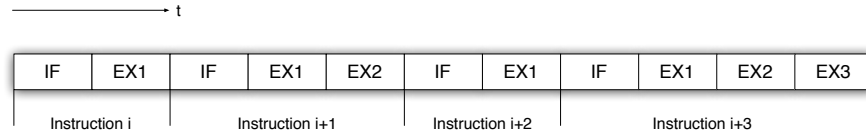
Figure 8.11: Multi-cycle implementation.
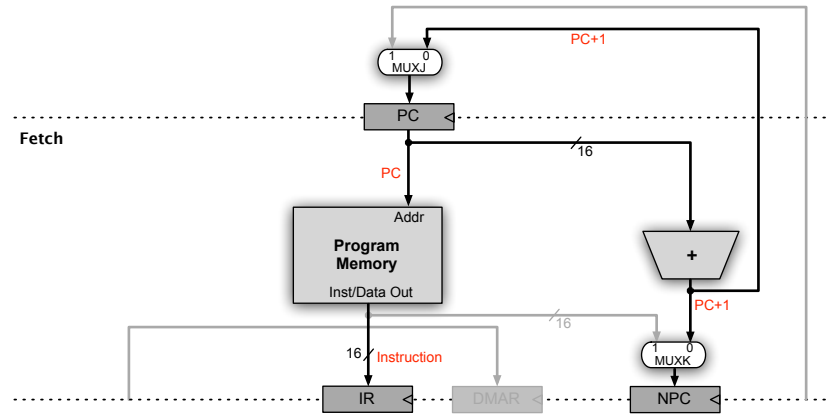
Figure 8.12: Fetch cycle.

8.4.1   Fetch Stage

In the Fetch stage, the current instruction pointed to by PC is fetched from the Program Memory and latched onto IR. In addition, PC is incremented (i.e., PC+1) and latched onto PC as well as the NPC register. PC+1 points to either the next instruction or a target address for 32-bit direct jump (JMP) and subroutine call (CALL) 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 in 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 IR at the end of the cycle. At this point, we will not distinguish between accessing Program Memory versus Data Memory, since 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

Table 8.3: Micro-operations for the Fetch Stage

Stage	Micro-operation
IF	IR ← M[PC], PC ← PC+1, NPC ← PC+1

Table 8.4: Micro-operations for Arithmetic and Logic Instructions

Arithmetic and Logic Instructions			
Stage	Micro-operation		
	Binary	Immediate	Unary
EX	Rd ← Rd *op* Rr	Rd ← Rd *op* K	Rd ← *op* Rd

is incremented to point to either the next instruction to be fetched and executed or a target address for 32-bit instructions. PC+1 is re-latched onto the PC 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, i.e., NPC←PC+1, by selecting input 0 of MUXK.

8.4.2 Execute Stage

The micro-operations performed by the Execute stage depends on the fetched instructions, which can be subdivided into the following three categories:

- Arithmetic and Logic,
- Data Transfer, and
- Branch and Jump.

Instructions in each category utilize similar parts of the datapath. The instructions discussed in this section require either 1 or 2 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:

- Rd *op* Rr
- Rd *op* K
- *op* Rd

where *op* represents an arithmetic or logic operation (see Table 8.1). Table 8.4 shows the micro-operations for the Execute (EX) stage for arithmetic and logic instructions.

Figure 8.13: The portion of the basic datapath utilized by Arithmetic and Logic instructions.

Figure 8.13 shows the portion of the datapath that performs arithmetic and logic operations. For instructions involving two registers, such as `ADD` and `SUB`, both source operands are obtained from the Register File based on register identifiers `Rd` and `Rr`. Instructions 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, MUXA selects either a register or an immediate value as the second operand. Instructions such as `INC`, `NEG`, `CLR`, etc., do not require a second source operand.

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 MUXB and MUXC, which then becomes latched onto the destination register based on `Rd` at the end of the clock cycle. With the exception of a few instructions that involve 16-bit operands, i.e., `ADIW` (*Add immediate to word*) and `SBIW` (*Subtract immediate from word*), and operations that generate 16-bit results, e.g., `MUL` (*Multiply unsigned*), most of the arithmetic and logic instructions complete in a single Execute cycle (see Table A.1).

Figure 8.14: The portion of the datapath for 1-cycle 8-bit Data transfer.

Data Transfer Instructions

Most *data transfer* instructions require either one or two Execute cycles depending on their type and whether data is transferred between registers or between a register and the Data Memory.

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 4.3). Table 8.5 summarizes the micro-operations for these four instructions.

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

Figure 8.15: The portion of the datapath for 1-cycle 16-bit Data transfer.

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 Register File based on the destination register identifier Rd available at wB at the end of the clock cycle.

In contrast to MOV, MOVW transfers 16-bit data, which is achieved 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 that a register identifier, not the content of a register, is incremented by one. For example, when Rr is specified as R26, the concatenated register pair R27:R26 is moved. 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. Note that these two paths for the register pair come from the 16-bit path from the Address Adder that are split into upper and lower 8 bits.

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
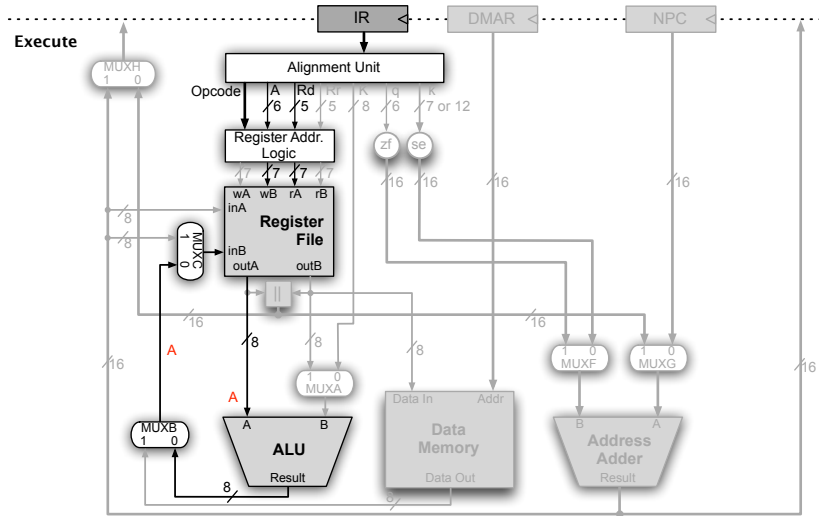
(a) IN.
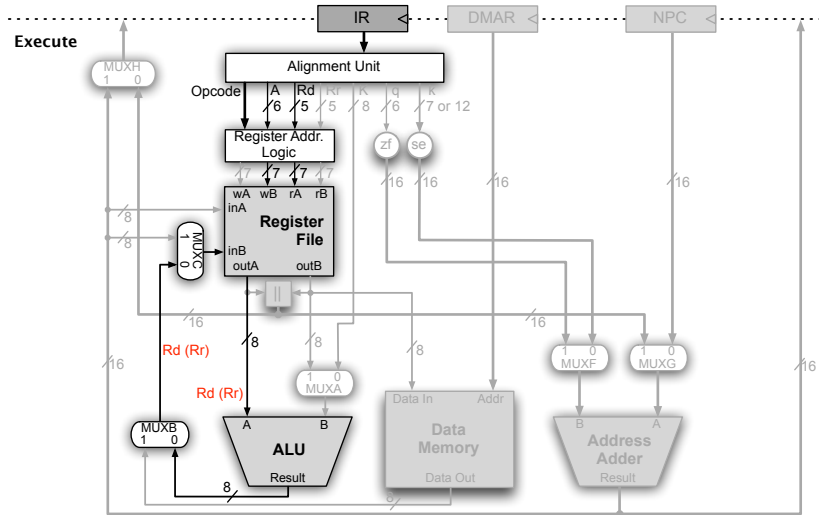
(b) OUT.

Figure 8.16: Part of the datapath utilized by IN and OUT instructions.

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 location defined by `A+32`. Note that there are no separate register identifiers for `Rd` and `Rr` in the instruction format (see Figure 8.1(c)). 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. Table 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` or simply `AR`). This address is moved through the Address Adder without modifying the content and then it is 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 MUXH 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 and low bytes of an `AR` read from the Register File are concatenated and

Figure 8.17: EX1 of register indirect for loads and stores.

Figure 8.18: EX1 of register indirect with displacement for loads and stores.
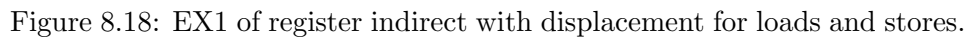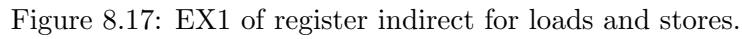
added with zero-filled 6-bit displacement q. Then, AR+q is latched onto DMAR.

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 and low bytes of an AR read from the Register File are concatenated and then decremented using the Address Adder. Then, AR-1 is latched onto DMAR, and at the same time it is written back to the Register File. This is done by directing the high and low bytes of the decremented address, i.e., AR-1(h) and AR-1(l), to the write ports inA and inB. Figure 8.19(b) shows the data transfer operations for *register indirect with post-increment*. The AR read from the register file is latched on to DMAR using the path from the output of the concatenate unit (||) to the 0-input of MUXH. At the same time, AR is incremented using the Address Adder and written back to the Register File.

In EX2, the operand is read from the Data Memory and latched onto the register specified by Rd 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: *branches* and *jumps*. The major difference between these two types of instructions is that branch instructions are *conditional*, while jump instructions are *unconditional*. A *conditional branch* evaluates a condition code or flag, e.g., Z, N, V, or S flag, and if the condition is true the control flow is transferred to the *target address*. Another difference is that branches uses *PC-relative addressing* while jumps can use either PC-relative addressing or *direct addressing*. PC-relative addressing uses 16-bit instruction format but the range is limited. On the other hand, absolute addressing allows for a much larger range but requires 32-bit instruction format. Table 8.7 shows the micro-operations for AVR's branch and jump instructions.relocatable.

Figure 8.21 shows the portion of the datapath used by conditional and unconditional 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 7-bit displacement k7 is used exclusively by conditional branch instructions (e.g., BREQ, BRLT, BRGE, etc.), while the 12-bit displacement k12 is used by relative jump and call instructions (i.e., RJMP and RCALL). The resulting address PC+1+se k7 or PC+1+se k12, known as the *branch target address*, becomes available at the input of PC to be either

(a) EX1 for load/store with pre-decrement.

(b) EX1 for load/store with post-increment.
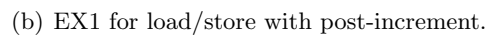
Figure 8.19: EX1 for register indirect with pre-decrement and post-increment for loads and stores.
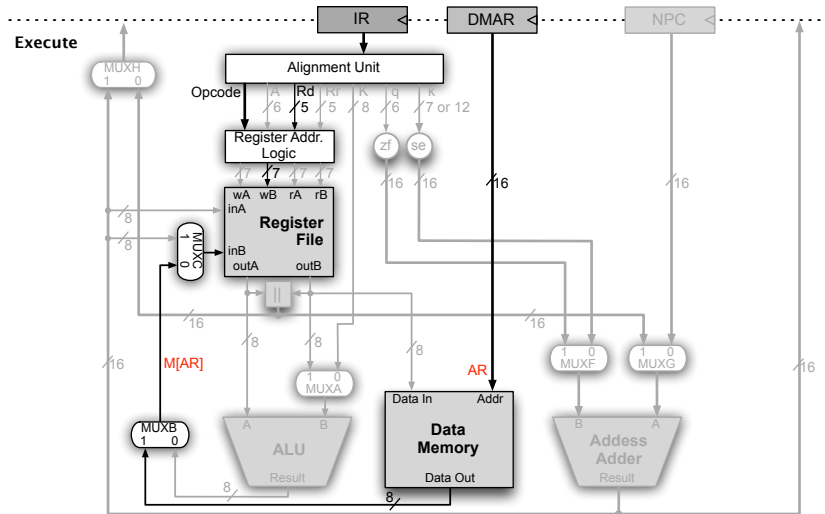
(a) EX2 for loads.
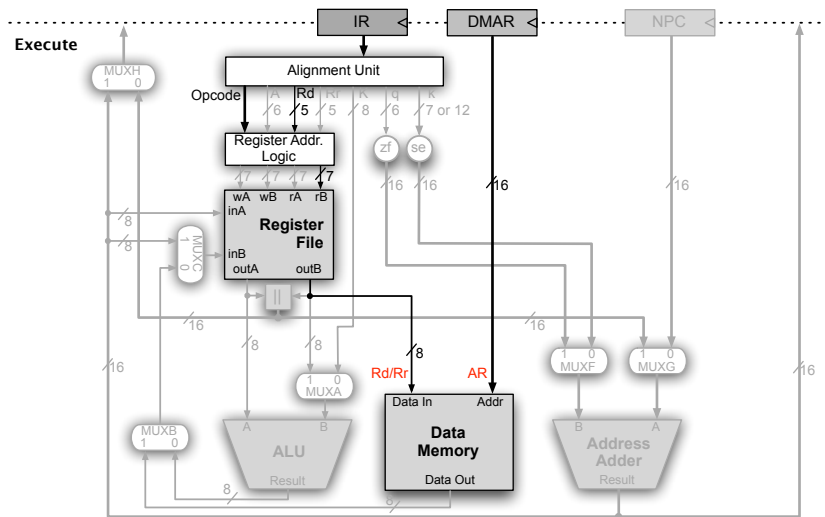
(b) EX2 for stores.

Figure 8.20: EX2 for loads and stores.

Table 8.7: Branch and Jump Instructions

Branch and Jump Instructions				
Stage	Micro-operations			
	Branches		Jumps	
	Conditional	Unconditional	Direct	Indirect
EX1	If (*flag*) then PC ← NPC+se k7	PC ← NPC+se k12	NPC ← M[PC]	PC ← Z
EX2			PC ← NPC	
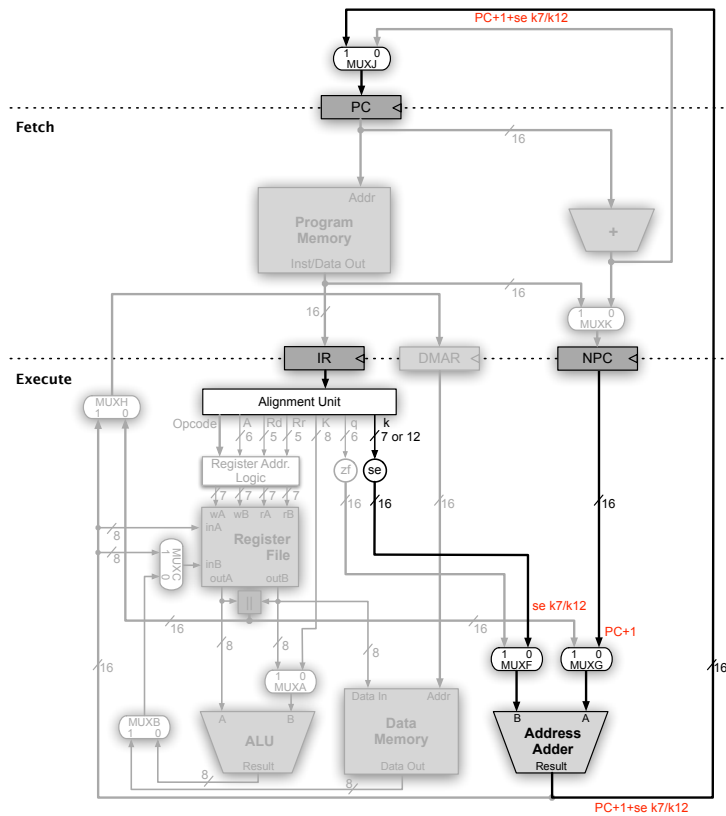
Figure 8.21: EX1 of PC-relative branch instruction.

conditionally or unconditionally latched at the end of the clock cycle.

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
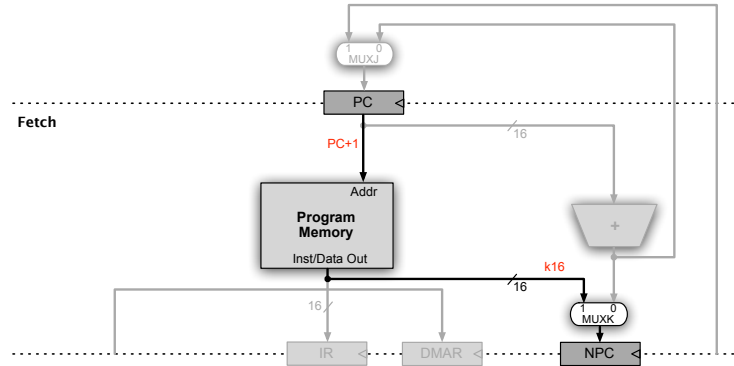
(a) EX1 for JMP k.
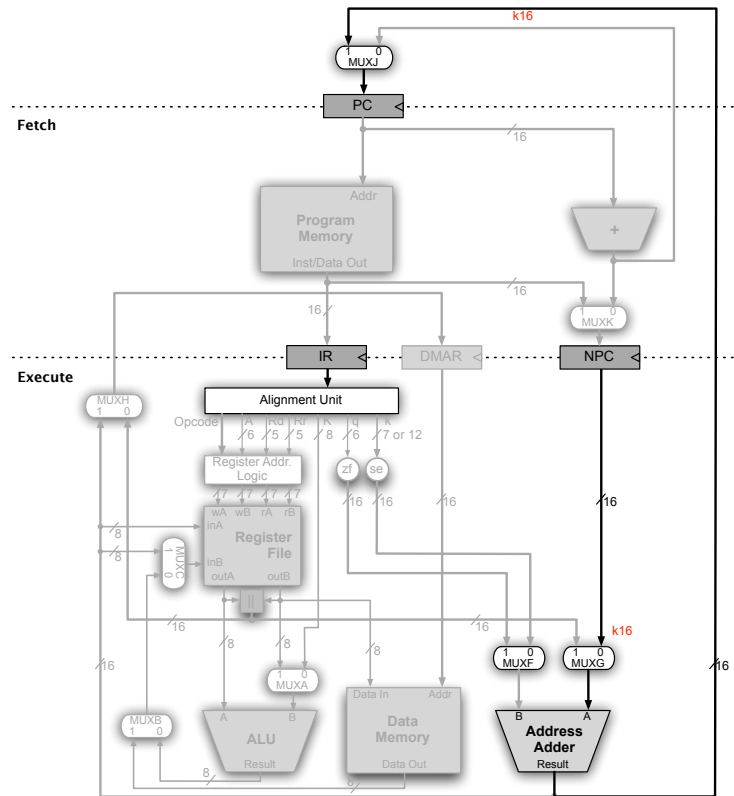
(b) EX2 for JMP k.

Figure 8.22: Micro-operations for direct jump instructions.
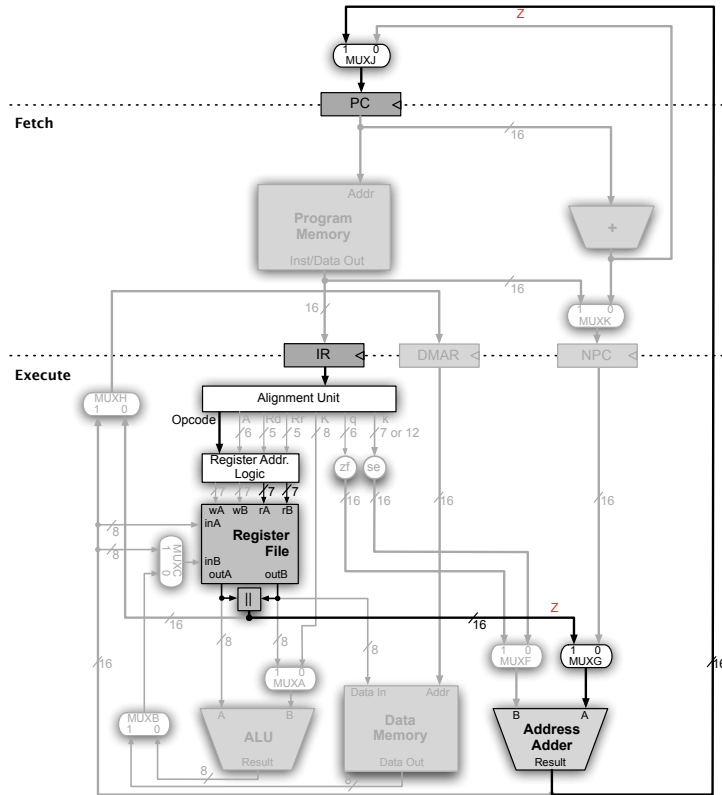
Figure 8.23: EX1 for indirect jump instruction.

where the second 16-bit of the instruction, k16, represents the target address of the jump. Therefore, unlike 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 during EX1. This can be thought of simply 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.

Figure 8.23 shows the portions of the datapath affected by the *indirect jump* (IJMP) instruction. In EX1, the high and low bytes of the address register Z (i.e., Zh and Zl) are read from the Register File and concatenated to generate a 16-bit target address. The resulting address is available at the input of PC via the Address Adder to be latched at the end of the clock cycle.
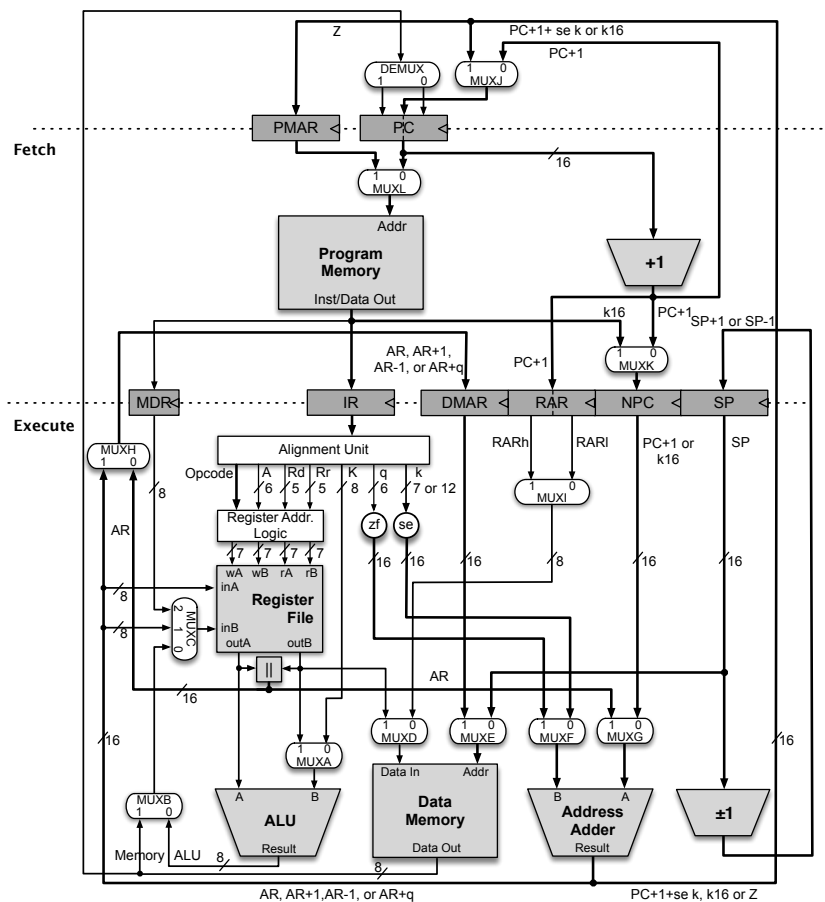
Figure 8.24: Enhanced 2-stage microarchitecture

8.5 Execution of More Complex Instructions

The basic datapath discussed thus far is capable of implementing many of the AVR instructions. However, some of the more complex AVR instructions require additional functionalities not available in the basic datapath. Figure 8.24 shows the *enhanced datapath* that handles 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 four additional special-purpose registers.

The *Return Address Register* (RAR) latches the return address for a subroutine call, which is either PC+1 for PC-relative (i.e., RCALL k12) and

Table 8.8: Operations of the Increment/Decrement Unit.

Operation	Description	Inc_Dec
SP+1	Increment	0
SP-1	Decrement	1

indirect subroutine (`ICALL`) calls and `PC+2` for the direct subroutine call (`CALL k16`). You may wonder why RAR is needed when it appears that NPC already latches `PC+1` or `PC+2`. The reason is that NPC can either hold `PC+1` (or `PC+2` in case of direct jumps) or `k16`. If `k16` is latched onto NPC for the 32-bit direct subroutine call (i.e., `CALL k16`), 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 MUXI. The write-port of the Data Memory (i.e., `Data In`) is preceded by MUXD to select between RARh/RARl and an operand from the Register File.

The *Stack Pointer* (SP) points to the top of the stack and is required by instructions that manipulate the stack. Note that high and low bytes of SP (i.e., SPH and SPL) are mapped to locations $3E and $3D in the 64 I/O registers' address space, and thus they can also be accessed from the Register File. However, there are several reasons why stack manipulations require SP to be treated as a special register together with the *Increment/Decrement Unit* (±1) to meet the clock cycle requirements of the AVR instruction set. First, subroutine call instructions require the address in the SP to be provided directly to the Data Memory rather than through DMAR. Second, the `ICALL` instruction requires accessing SP and `Z` registers at the same time. Third, one instruction in particular, `RCALL k12`, requires the calculation of PC-relative target address and the decrementing of the SP to occur at the same time. These situations can only be handled properly if SP is implemented as a separate register with a dedicated increment/decrement capability. In order to handle the SP, 2-to-1 MUXE has been added to the address-port of the Data Memory (i.e., `Addr`). Finally, the Increment/Decrement Unit is controlled by the `Inc_Dec` signal based on Table 8.8.

The *Program Memory Address Register* (PMAR) provides addresses for constants stored in the Program Memory using MUXL. These addresses are stored in the `Z`-register, which is used exclusively by `LPM` (*Load program memory*) instructions. The *Memory Data Register* (MDR) latches a

Table 8.9: Micro-operations for the Fetch Stage

Stage	Micro-operations
IF	IR ← M[PC], PC ← PC+1, NPC ← PC+1, RAR ← PC+1

constant accessed from the Program Memory, which together with MUXC
expanded to a 3-to-1 multiplexer allow it to be written to the Register File
in the Execute stage. Note that 8-bit constants are read from the Program
Memory using addresses that are shifted left by one bit. Thus, the least sig-
nificant bit of the address will select between first (left) and second (right)
constant within an instruction word, which is then latched onto MDR.

In addition to the four new special-purpose registers, there are some
other minor improvements made in the enhanced datapath. The PC is
separated into PCh and PCl representing the high and low bytes of the PC,
respectively. Finally, the inclusion of DEMUX allows a return address of a
subroutine call 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. In Section 8.6, we
will discuss how one of these complex instructions, CALL (*Direct subroutine
call*), can be implemented on the enhanced datapath.

The micro-operation for the Fetch stage for the enhanced datapath is
shown in Table 8.9, which is similar to Table 8.3 but requires one additional
data transfer operation of latching PC+1 onto RAR.

8.6   Control Unit Design

As we saw in Section 8.1, a microarchitecture consists of two major com-
ponents: datapath and Control Unit. The discussion up to now has been
on the datapath, which performs micro-operations using data transfer op-
erations. 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 multi-
cycle implementation of the pseudo-AVR Microarchitecture. In order to
illustrate the design of the control unit, we will base our discussion on a

small subset of AVR instructions. Table 8.10 shows descriptions of the instructions covered by the Control Unit.

Table 8.10: 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,S
	`ORI Rd, K`	Logical OR Register & Constant	Rd← Rd $\vee$ K	Z,N,V,S
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+2	None

8.6.1 Opcode Encoding

The Control Unit is responsible for decoding instructions and providing appropriate set of control signals to the various components of the datapath. In order to understand the instruction decoding process, we need to first understand how opcodes are encoded. Figure 8.25 shows the encoding of opcodes for instructions in Table 8.10 (see Figure E.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 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 that consists of data transfer, control transfer, 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 8.25, 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, or 11, and the next three bits (bits 11-9) further subdivide each type into different instructions. Note that for the instructions covered in Table 8.10, only 01 for bits 13-12 is listed. Furthermore, for some instructions, additional bits define different operations for similar instructions. For example,
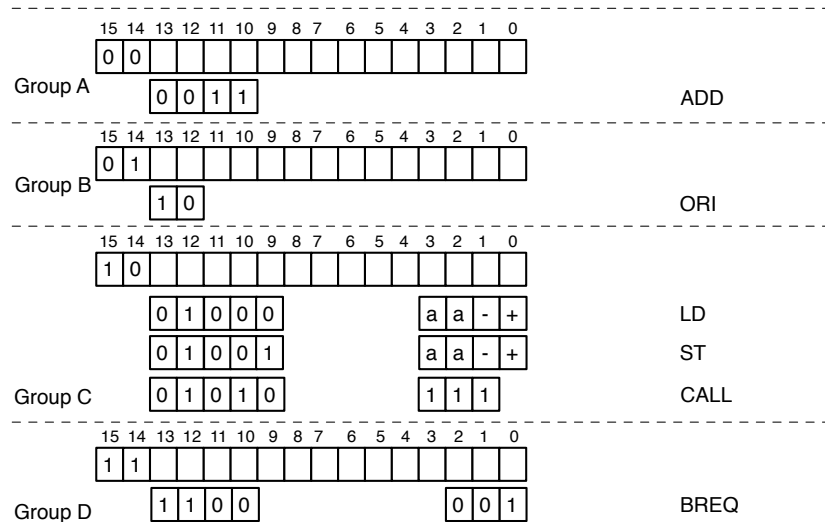
Figure 8.25: Opcode encoding for instructions in Table 8.10.

the `LD` 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. Furthermore, bits 3 and 2 ("aa") define one of the addresses registers `X`, `Y`, or `Z` (i.e., 11 for `X`-register, 10 for `Y`-register, and 00 for `Z`-register). 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 which condition is used to determine the outcome of the branch.

8.6.2   Control and Alignment Unit

Figure 8.26 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, and all the control signals. The control signals ALU_f, Adder_f, and Inc_Dec are defined in Tables 8.1, 8.2, and 8.8, respectively. The special-purpose registers IR, PC, NPC, and SP are enabled using control signals of the form xx_en, where xx represents the name of the special-purpose register (e.g., NPC_en). PMAR, MDR, and DMAR do not require enable signals and thus
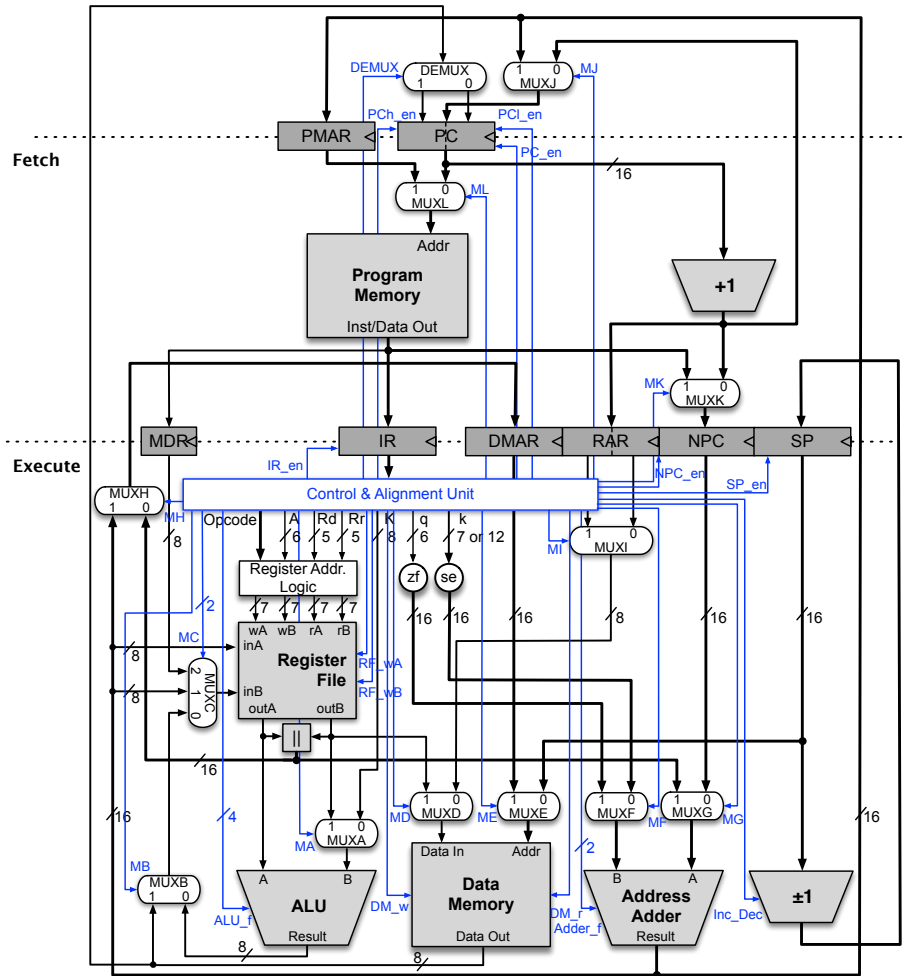
Figure 8.26: Control signals for the enhanced AVR datapath.

are simply latched with the clock. In addition, the control signal NPC_en is used to enable both RAR and NPC. Note that there are three way to latch an address onto PC. The first method is to latch the entire 16-bit address by asserting 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 to latch the return address of a subroutine call a byte at a time. Therefore, unless the instruction being executed is RET (*Subroutine return*), PCh_en and PCl_en are both not asserted (i.e., set to 0's). Moreover, only one of the three control signals (PC_en, PCh_en, or PCl_en) can be asserted at a time. The multiplexers are controlled using the control signals of the form Mx, where x represents the name of a multiplexer. Note that with the exception of MUXC, all the multiplexers are 2-to-1requiring only a single bit control signal. MUXC is 3-to-1 and thus requires 2 bits of control signals. Finally, as the name suggests, the DEMUX control signal controls the DEMUX.

There are two parts to the design of the CAU. First, a *set of control signals* need to be generated for each cycle or stage of an instruction execution. Second, the *sequence control* needs to be defined for the series of micro-operations required to execute instructions. These two requirements are met by implementing the CAU as a *finite state machine* (FSM).

The following discusses the requirements for generating the set of control signals for the fetch and execute cycles. Then, Section 8.6.4 will present the requirements for the sequence control.

Fetch Cycle

The Fetch cycle is the same for all instructions and is controlled by the following signals: MJ, MK, ML, PM_read, and PM_write. Since the discussion of the Fetch cycle 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, 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. The NPC_en signal also controls the latching of the RAR register. All other registers, i.e., DMAR and MDR, are simply latched with the clock.

Figure 8.27 shows the control signals needed to fetch an instruction. MUXL 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
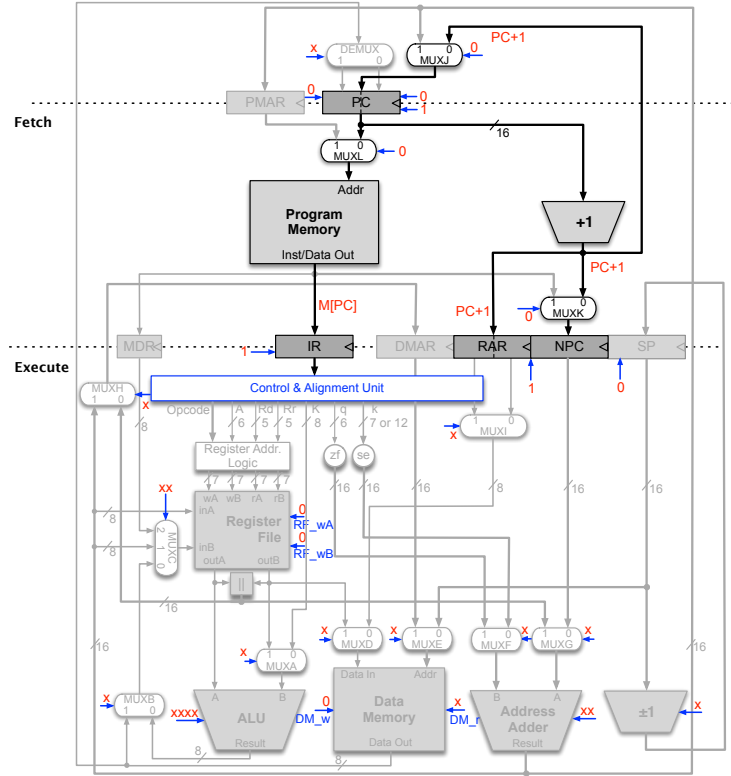
Figure 8.27: Control signals for the Fetch stage.

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. The IR is enabled to be latched 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 Execute cycles involving 32-bit instructions. Either PC+1 or PC+2 is latched onto RAR as well as NPC by selecting the 0-input of MUXK and asserting NPC_en. Again, NPC and RAR are enabled only during the Fetch cycle, and some Execute cycles involving 32-bit instructions.

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
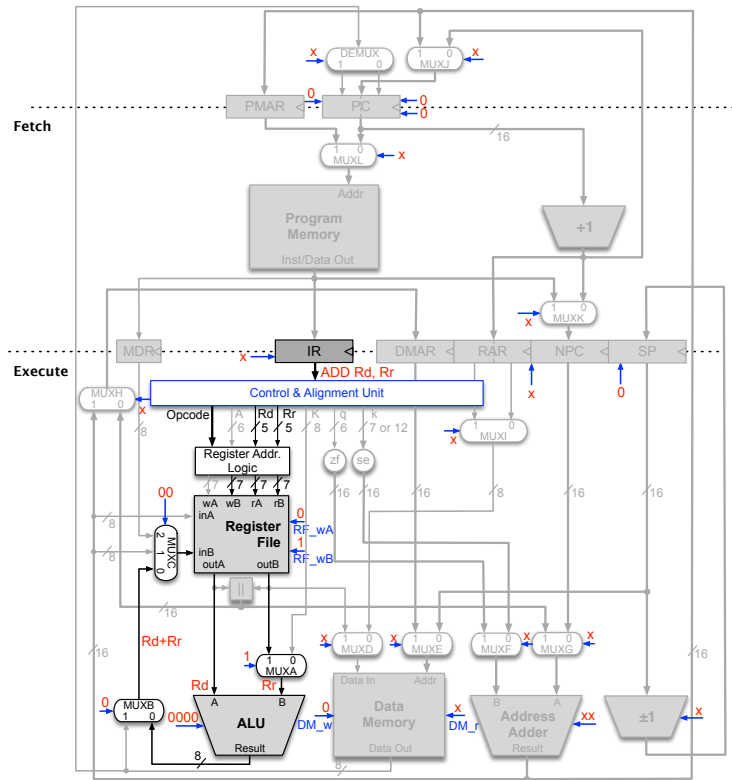
Figure 8.28: Control signals required in EX1 for `ADD Rd,Rr` instruction.

RF_wB=0 for the Register File.

Execute Cycle

As mentioned before, control signals required in the Execute (EX) stage depend on the instruction being executed.

Figure 8.28 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 is set to 1 to accept the Register File content specified by `Rr`. 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 input-0 of MUXB and input-0 of MUXC. Finally, the control signal RF_wB=1
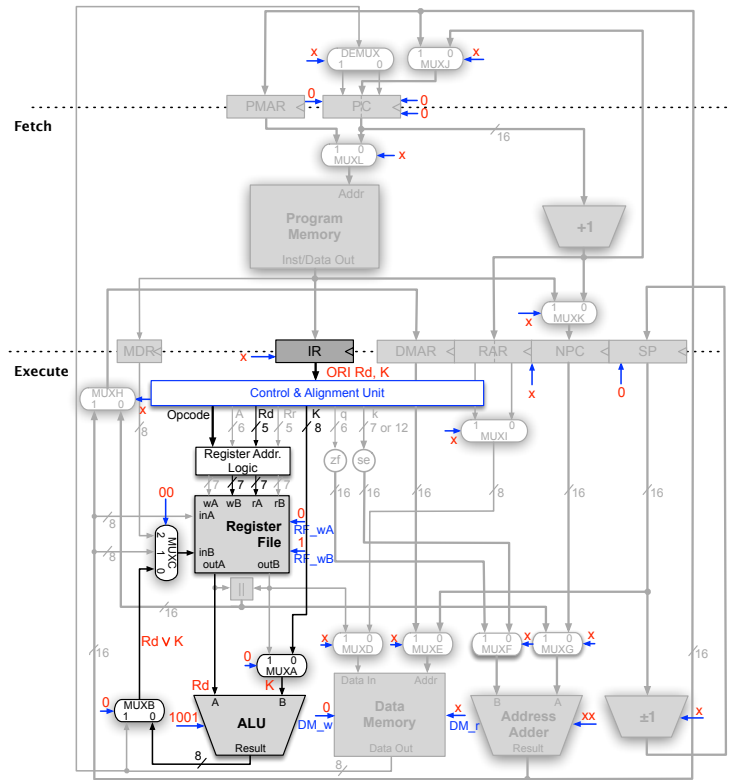
Figure 8.29: Control signals required in EX1 for `ORI Rd,K` instruction.

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 for DM_w and RF_wA, which have to be set to 0 to prevent the Data Memory and the Register File from being updated with unrelated or incorrect data. In addition, the control signals PC_en and SP_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 SP points to the top of the stack. Therefore, modifying these contents will be detrimental. In contrast, IR_en and NPC_en can all be "don't cares" because EX1 is the only execute cycle for ADD and whatever information latched to these registers at the end of the cycle will not be used and the fetch cycle will start all over again. MUXs and DEMUX in the Fetch stage can also be "don't cares" since PC is not updated.

Figure 8.29 shows the control signals required in EX1 for the `ORI` in-
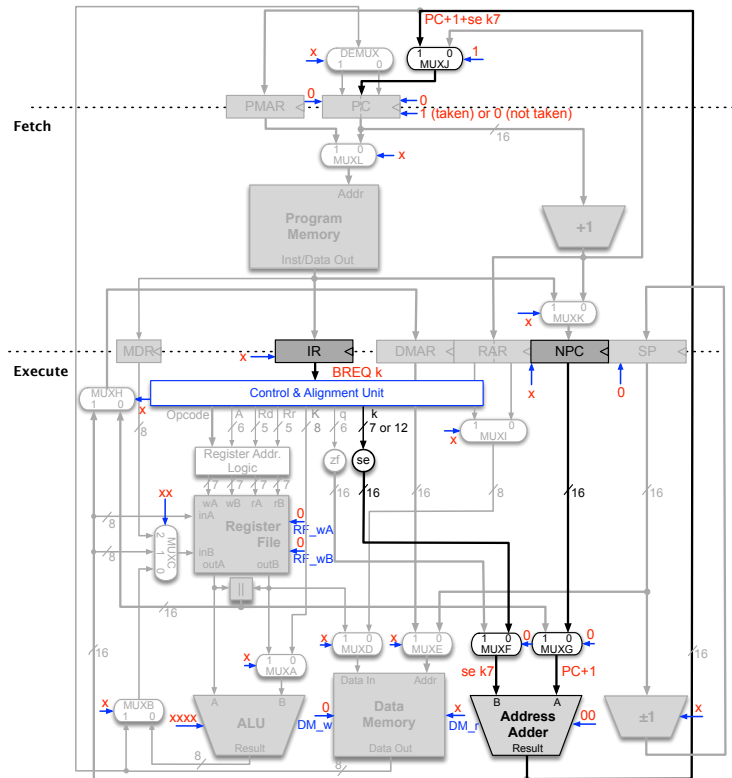
Figure 8.30: Control signals required in EX1 for `BREQ k` instruction.

struction. 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 MUXA is set to `0` to select the 8-bit constant `K`, and the ALU performs a logical OR operation by setting the control signal `ALU_f` to `1001` (see Table 8.1).

Figure 8.30 shows the control signals required in EX1 for the `BREQ` instruction. When the control signal `Adder_f = 00` is given (see Table 8.2), 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-flag is set, then `PC_en` is set to latch the branch target address onto PC. Otherwise, PC is not latched, i.e., `PC_en=0`.

Figure 8.31 shows the control signals required for `LD` and `ST` instructions in EX1, which are common for both instructions. In this example, the upper

Figure 8.31: Control signals required in EX1 for `LD Rd,Y` and `ST Y,Rr` instructions.

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 Address Adder by selecting the input-1 of MUXG. The control signal Adder_f = 11 causes the Address Adder to simply pass the content of the Y-register to the output, which then becomes available to the input of DMAR by setting MUXH 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 and IR_en are also set to zeros to prevent the contents of PC and IR from being overwritten. Control signals for RAR and NPC as well as all the other MUXs (as well as DEMUX) are "don't cares".

Figure 8.32 shows the data transfer operations in EX2 for `LD` and `ST` instructions. For both of these instructions, MUXE is set to 1 so that the
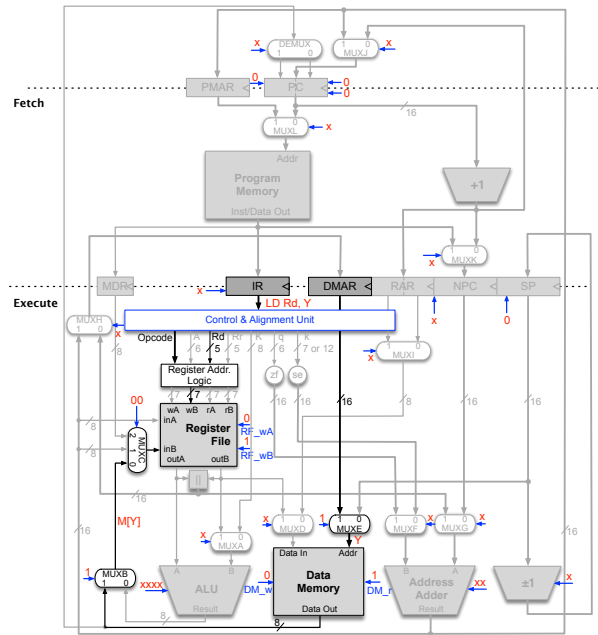
(a) EX2 for LD.
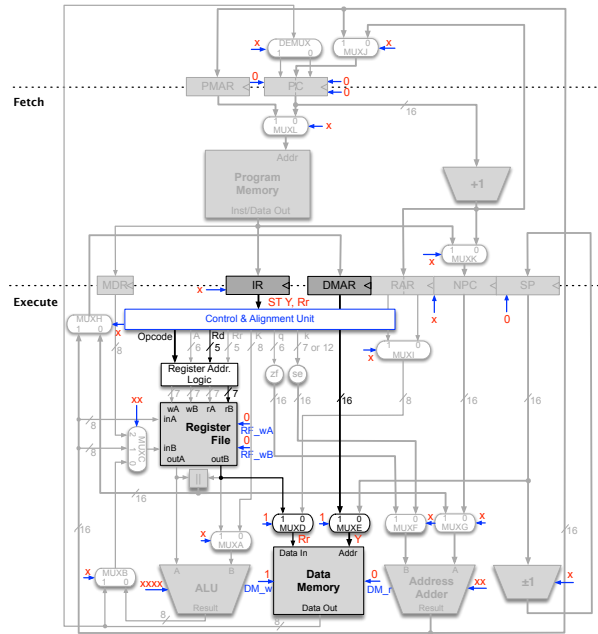
(b) EX2 for ST.

Figure 8.32: Control signals required in EX2 for LD Rd, Y and ST Y, Rr instructions.

effective address latched onto DMAR during EX1 can be used to access the Data Memory. For LD, the Data Memory is read by setting control signals DM_r = 1 and DM_w = 0, and the operand (i.e., M[Y]) is routed back to the lower write port (i.e., inB) of the Register File via MUXB and MUXC. The operand is written back to the Register File at the end of the clock cycle by setting RF_wB to 1. For ST, 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 MUXD 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, PC_en, and SP_en, which are all zeros. Note that Rd from the CAU to the Register Address Logic serves as the register identifier for both LD and ST since these instructions use the one-operand format shown in Figure 8.1(d).

Table 8.11 shows the sequence of micro-operations for CALL k. Figure 8.33 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 and latched onto NPC. This is achieved by setting MUXL to accept input-0, MUXK 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 RAR. Again, except for IR_en, RF_wA, RF_wB, DM_w, and SP_en, all other control signals can be "don't cares", including PC_en since the PC will be overwritten with the target address of CALL in EX3.

Table 8.11: Direct Subroutine Call.

Direct Subroutine Call	
Stage	Micro-operations
EX1	NPC ← M[PC], RAR ← PC+1
EX2	M[SP] ← RARl, SP ← SP-1
EX3	M[SP] ← RARh, SP ← SP-1, PC ← NPC

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 current address in SP to write RARl into the Data Memory. This is achieved by selecting input-0 for MUXD and input-0 for MUXE, and setting DM_w to
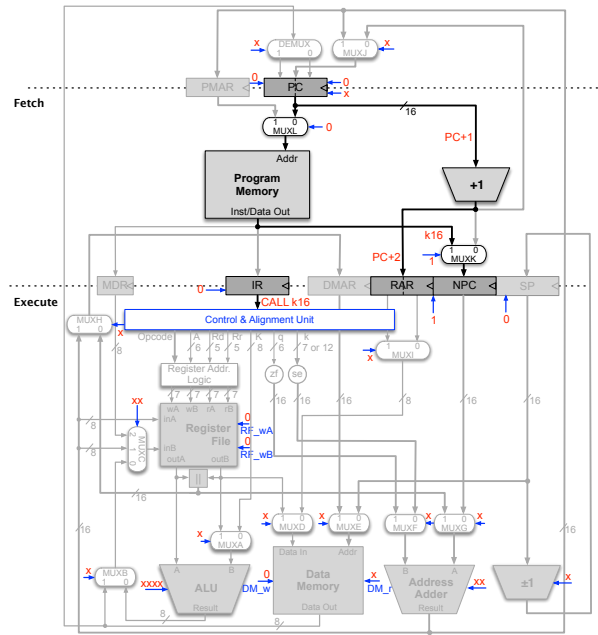
(a) EX1 of `CALL`.
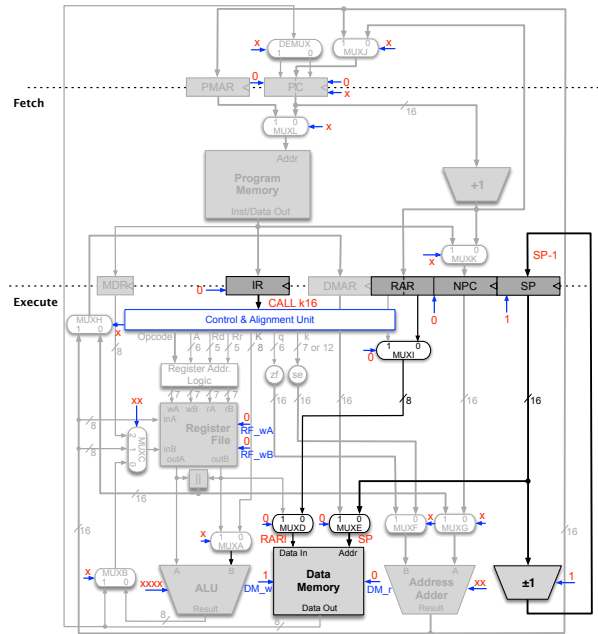
(b) EX2 of `CALL`.

Figure 8.33: Control signals required for `CALL  k` instruction.
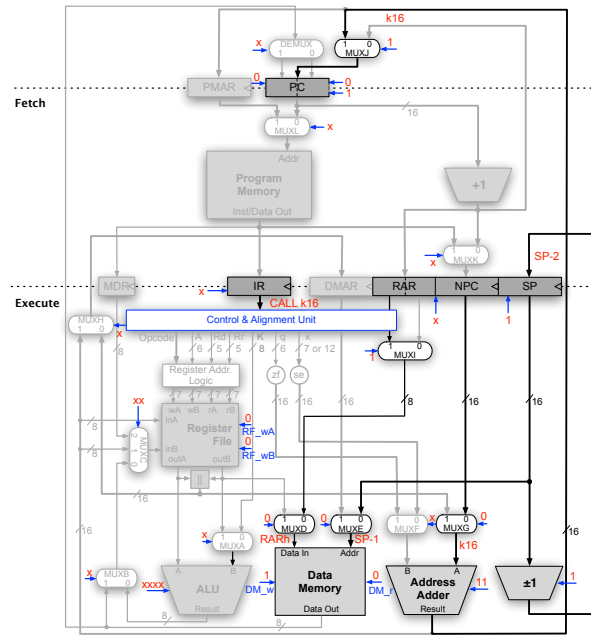
(c) EX3 of `CALL`.

Figure 8.33: Control signals required for `CALL k` instruction (*continued*).

1. The second operation is accomplished by setting both inc_dec and SP_en to 1. The decremented SP (i.e., SP-1) is then routed back and latched to SP at the end of the clock cycle. All other control signals can be "don't cares", except for IR_en, NPC_en, RF_wA, RF_wB, and DM_r.

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 (via MUXG) and latching it onto the PC by setting MUXJ to 1 and PC_en to 1. Otherwise, all other control signals are identical to EX2.

Table 8.12 summarizes the required control signals for the six instructions discussed in this section.

8.6.3   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 the two write ports

Table 8.12: Summary of control signals for instructions in Table 8.10

Control Signals	IF	ADD EX1	ORI EX1	BREQ EX1 Z=0	BREQ EX1 Z=1	LD EX1	LD EX2	ST EX1	ST EX2	CALL EX1	CALL EX2	CALL EX3
MJ	0	x	x	1	1	x	x	x	x	x	x	1
MK	0	x	x	x	x	x	x	x	x	1	x	x
ML	0	x	x	x	x	x	x	x	x	0	x	x
IR_en	1	x	x	x	x	0	x	0	x	0	0	x
PC_en	1	0	0	0	1	0	0	0	0	x	x	1
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
NPC_en	1	x	x	x	x	x	x	x	x	1	0	x
SP_en	0	0	0	0	0	0	0	0	0	0	1	1
DEMUX	x	x	x	x	x	x	x	x	x	x	x	x
MA	x	1	0	x	x	x	x	x	x	x	x	x
MB	x	0	0	x	x	x	1	x	x	x	x	x
ALU_f	xxxx	0000	1001	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
MC	xx	00	00	xx	xx	xx	00	xx	xx	xx	xx	xx
RF_wA	0	0	0	0	0	0	0	0	0	0	0	0
RF_wB	0	1	1	0	0	0	1	0	0	0	0	0
MD	x	x	x	x	x	x	x	x	1	x	0	0
ME	x	x	x	x	x	x	1	x	1	x	0	0
DM_r	x	x	x	x	x	x	1	x	0	x	0	0
DM_w	0	0	0	0	0	0	0	0	1	0	1	1
MF	x	x	x	0	0	x	x	x	x	x	x	x
MG	x	x	x	0	0	1	x	1	x	x	x	0
Adder_f	xx	xx	xx	00	00	11	xx	11	xx	xx	xx	11
Inc_Dec	x	x	x	x	x	x	x	x	x	x	1	1
MH	x	x	x	x	x	1	x	1	x	x	x	x
MI	x	x	x	x	x	x	x	x	x	x	0	1
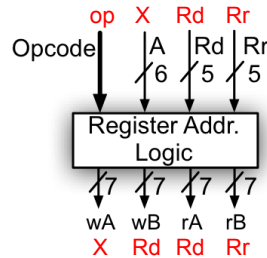
Figure 8.34: RAL Mapping for `ADD`.

(`wA` and `wB`).

    Figure 8.34 shows the RAL mapping for the `ADD` instruction, where `Rd` and `Rr` are *explicitly* defined by the instructions (`op` and `X` represent opcode bits and "don't care", respectively). This instruction directly maps `Rd` and `Rr` to inputs `rA`, `rB`, and `wB` of the Register File, i.e., `rA` = `Rd`, `rB` = `Rr`, and `wB` = `Rd`.

    Figure 8.35 shows the RAL mapping `ORI`. The main different between the RAL mapping for `ORI` and `ADD` is that, `Rr` is "don't care" for `ORI` because
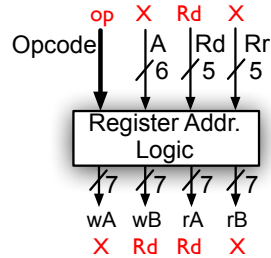
Figure 8.35: RAL Mapping for ORI.

it uses the constant K rather than the source operand Rr by selecting input-0 of MUXA.

The I/O register identifier A needs to be offset by 32 since the 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 in the six instructions in Table 8.10, 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) are more tricky and require the understanding of how opcodes are encoded. The address registers for these instructions are specified by bits 3 and 2 of the instruction format (indicated as "aa" in Figure 8.25). 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 need to be decoded from the IR and appropriately mapped to registers R29 and R28.

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 (see Section 8.6.4) to determine the RAL mapping. These two instructions share the same EX1and refer to the Y-register, and thus Yh and Yl have to be mapped to rA and rB, respectively. For LD in EX2, Rd serves as the destination register identifier, and thus it is mapped to wB. In contrast, for ST in EX2, Rd from the instruction format serves as the source register identifier, and is mapped to rB.

Finally, both BREQ and CALL instructions do not require RAL mapping because they do not use the Register File. Table 8.13 summarizes the RAL mapping for the six instructions.

Figure 8.37 shows the implementation of the RAL for the instructions
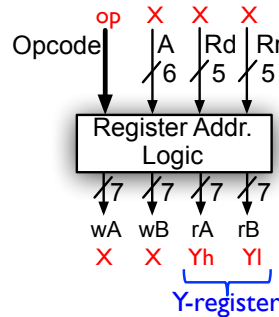
(a) Mapping for LD and ST in EX1.
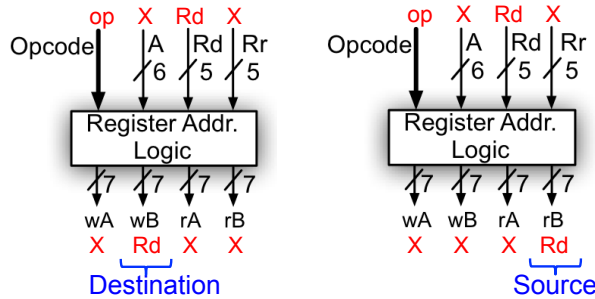
(b) Mapping for LD in EX2.   (c) Mapping for ST in EX2.

Figure 8.36: RAL Mapping for LD and ST.

in Table 8.10. The 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) cause 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 to choose Yh and Yl, which are hardwired to 0011101_2 (29) and 0011100_2 (28), respectively, based on bits 15-9 of the instruction and the current state (see Section 8.6.4).

8.6.4   Sequence Control

Now that all the control signals for Fetch and Execute cycles have been defined, we need a *sequence control* that governs the transitions from one

Table 8.13: Summary of RAL mapping for instructions in Table 8.10

RF R/W Ports	IF	Instructions									
		ADD	ORI	BREQ	LD		ST		CALL		
		EX1	EX1	EX1	EX1	EX2	EX1	EX2	EX1	EX2	EX3
wA	x	x	x	x	x	x	x	x	x	x	x
wB	x	Rd	Rd	x	x	Rd	x	x	x	x	x
rA	x	Rd	Rd	x	Yh	x	Yh	x	x	x	x
rB	x	Rr	x	x	Yl	x	Yl	Rd	x	x	x
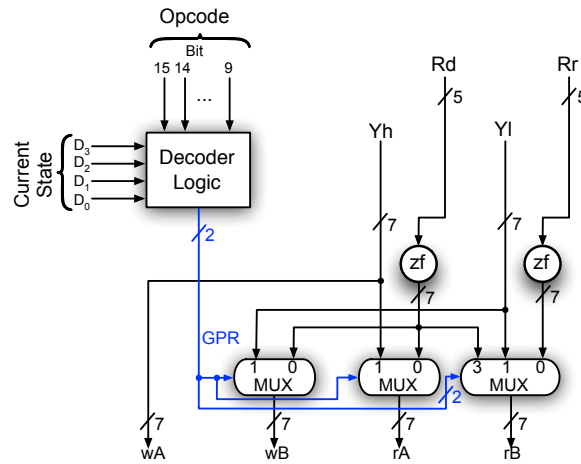
Figure 8.37: Register Address Logic.

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 8.38 shows the *finite state diagram* for the multi-cycle implementation of the six instructions in Table 8.10.

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 8.14 shows the *state table*, which was derived from the finite state diagram in Figure 8.38 and control signals from Table 8.12. 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 '–'

Table 8.14: Finite state table for the multi-cycle implementation.

Description	Current State	Inputs Opcode $b_{15}b_{14}b_{10}b_9$	Z	Next State	MJ	MK	ML	IR_en	PC_en	PCh_en	PCl_en	NPC_en	SP_en	DEMUX	MA	MB	ALU_f	MC	RF_wA	RF_wB	MD	ME	DM_r	DM_w	MF	MG	Adder_f	Inc_Dec	MH	MI
IF	0000	xxxx	×	—	0	0	0	1	1	0	0	1	0	×	×	0	xxxx	xx	0	0	×	×	×	0	×	0	xx	×	×	×
ADD	0001	00xx	×	0000	0	0	0	×	0	0	0	×	0	×	0	0	0000	00	1	1	×	×	×	0	×	0	xx	×	×	×
ORI	0010	01xx	×	0000	0	0	0	×	0	0	0	×	0	×	1	1	1001	00	0	1	×	×	×	0	×	×	xx	×	×	×
BREQ (Z=0)	0011	11xx	0	0000	×	×	×	×	0	0	0	×	0	×	×	×	xxxx	xx	0	0	×	×	×	0	×	×	00	×	×	×
BREQ (Z=1)	0100	11xx	1	0000	1	×	×	×	0	0	0	×	0	×	×	×	xxxx	xx	0	0	×	×	×	0	0	1	11	×	1	×
LD/ST (EX1)	0101	100x	×	—	×	×	×	0	0	0	0	×	0	×	×	×	xxxx	xx	0	0	×	1	×	0	×	×	xx	×	×	×
LD (EX2)	0110	1000	×	0000	×	×	×	×	0	0	0	×	0	×	×	1	xxxx	xx	0	1	1	1	1	0	1	×	xx	×	×	×
ST (EX2)	0111	1001	×	0000	×	1	0	×	×	0	0	1	0	×	×	×	xxxx	xx	0	×	×	×	×	1	×	×	xx	×	×	×
CALL (EX1)	1000	1010	×	1001	0	×	×	0	×	0	0	0	0	×	×	×	xxxx	xx	0	×	×	×	×	0	×	×	xx	×	×	×
CALL (EX2)	1001	xxxx	×	1010	×	×	×	0	×	0	0	×	1	×	1	0	xxxx	xx	0	0	0	0	0	1	×	×	xx	1	×	0
CALL (EX3)	1010	xxxx	×	0000	1	×	×	×	1	0	0	×	1	×	×	×	xxxx	xx	0	0	0	0	0	1	×	0	11	1	×	1

Figure 8.38: The Finite State Machine control for the multi-cycle datapath.

in the Next State field. The opcode field and the Z-flag are "don't cares" in this state indicating that instruction fetch occurs regardless of these inputs.

For instructions that require a single Execute cycle (i.e., ADD, ORI, and BREQ), the current state, i.e., 0000 (IF), together with the opcode of the fetched instruction (see Section 8.25) 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 uniquely defines this instruction as ORI and its next state as 0010. The BREQ instruction requires the Z-flag as well as the opcode as inputs. Therefore, if the bits 15-14 are 11, and if the Z-flag is set by the predecessor instruction, the sequence control transitions to state 0100. If the opcode bits indicate BREQ but the Z-flag 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 (i.e., LD, ST, and CALL), again the current state 0000 (IF) together with the opcode uniquely define the next state (e.g., EX1). Both LD and ST instructions 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 the 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 from state 1001 to state 1010 are independent of the input and defined only by the Current State.

After each instruction executes its last Execute cycle, the control sequence transitions back to state 0000, and the instruction cycle starts over.

8.7   FSM Implementation of the Control Unit

Under Construction!!!

8.8   Pipeline Implementation

Under Construction!!!

Chapter 9

Arithmetic and Logic Unit

Contents

9.1   Introduction

Arithmetic and Logic Unit (ALU) is one of the most important component in a processor. ALU is involved in not only arithmetic and logic operations but also in just about every micro-operation. For example, ALU or a variation of an ALU (depending on the microarchitecture) is used to calculate effective addresses for operands in memory, branch target addresses for conditional and unconditional branches, update stack pointer, etc. It is even used to transfer data from one register to another.

Our discussion starts with a review of binary number system in Section 9.2. Section 9.3 discusses shift operations. This is followed by a design of a basic ALU in Section 9.4. The objective of this section is to discuss how an ALU for a typical microcontroller, such as AVR, may be designed. Sections 9.5 and 9.6 present integer multiplication and division, respectively. These two sections also present techniques to speedup multiplication and
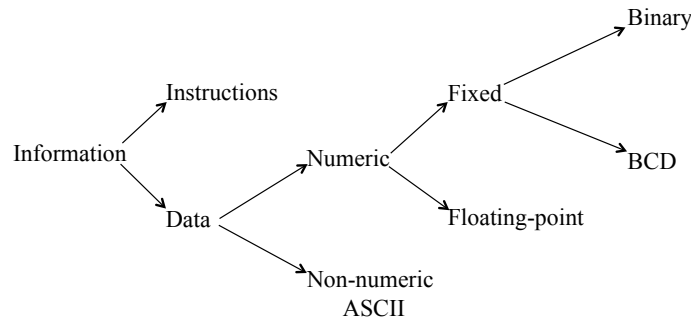
Figure 9.1: Information in a computer.

division operations, which is crucial for performance. Finally, Section 9.7 presents floating-point representation and operations.

9.2   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 understand 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

$$N = (b_{n-1}b_{n-2}\cdots b_1 b_0), \tag{9.1}$$

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.

9.2.1 Sign-Magnitude Representation

Signed-Magnitude representation of an n-bit binary number N is given as

$$N = (0b_{n-2}b_{n-3}\cdots b_1 b_0) \text{ and} \tag{9.3}$$
$$-N = (1b_{n-2}b_{n-3}\cdots b_1 b_0),$$

which has a range of

$$+0 \leq N \leq 2^{n-1} - 1 \text{ and} \tag{9.4}$$
$$-(2^{n-1} - 1) \leq N \leq -0.$$

The following example shows 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 (MSB) 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 9.1. Addition of two numbers with opposite signs in Sign-Magnitude representation.

$$
\begin{array}{cccccl}
 & 0 & 1 & 0 & 0 & (+4) \\
+ & 1 & 1 & 0 & 1 & (\text{-}5) \\
\hline
 & 1 & 0 & 0 & 1 & (\text{-}1)
\end{array}
$$

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 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.

9.2.2   1's-complement Representation

1's-complement representation of an n-bit binary number N is given as

$$N = (0b_{n-2}b_{n-3}\cdots b_1b_0) \text{ and} \tag{9.5}$$
$$-N = (1\bar{b}_{n-2}\bar{b}_{n-3}\cdots \bar{b}_1\bar{b}_0),$$

with a range given by

$$+0 \leq N \leq 2^{n-1} - 1 \text{ and} \tag{9.6}$$
$$-(2^{n-1} - 1) \leq N \leq -0.$$

The following shows 1's-complement representation with $n = 4$:

$$0111 = 7 \qquad\qquad 1111 = -0$$
$$0110 = 6 \qquad\qquad 1110 = -1$$
$$0101 = 5 \qquad\qquad 1101 = -2$$
$$0100 = 4 \qquad\qquad 1100 = -3$$
$$0011 = 3 \qquad\qquad 1011 = -4$$
$$0010 = 2 \qquad\qquad 1010 = -5$$
$$0001 = 1 \qquad\qquad 1001 = -6$$
$$0000 = 0 \qquad\qquad 1000 = -7$$

We learned from introductory course in digital logic design that 1's-complement of a number N, $OC(N)$, is obtained by complementing each bit, i.e., $\bar{b}_i$. However, there is a mathematical meaning of $OC(N)$ with n bits, which is given by

$$OC(N) = 2^n - N - 1. \qquad\qquad (9.7)$$

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 9.2. Addition of two positive numbers in 1's-complement.

$$
\begin{array}{ccccc}
 & 0 & 0 & 1 & 1 & (3) \\
+ & 0 & 0 & 1 & 0 & (2) \\
\hline
 & 0 & 1 & 0 & 1 & (5) \\
\end{array}
$$

This example adds two positive numbers, and thus it is performed using straightforward binary addition.

Now consider the following examples of adding two numbers of opposite signs.

Example 9.3. Addition of two numbers with opposite signs in 1's-complement:

$$
\begin{array}{ccccc}
 & 0 & 1 & 0 & 0 & (4) \\
+ & 1 & 0 & 1 & 0 & (-5) \\
\hline
 & 1 & 1 & 1 & 0 & (-1) \\
\end{array}
\qquad\qquad
\begin{array}{ccccc}
 & 0 & 1 & 0 & 1 & (5) \\
+ & 1 & 0 & 1 & 1 & (-4) \\
\hline
\mathbf{1} & 0 & 0 & 0 & 0 & \\
+ & & & & 1 & \text{EAC} \\
\hline
 & 0 & 0 & 0 & 1 & (1) \\
\end{array}
$$

In the first example, the second operand is a negative number. Therefore, the net effect is to subtract the magnitude of 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 MSB, and thus taking 1's-complement of 1110 gives 0001 indicating that the result 1110 is -1.

The second example generates 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 Equation 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:

$$
\begin{aligned}
N_1 + OC(N_2) &= N_1 + 2^n - N_2 - 1 \qquad\qquad (9.8)\\
&= 2^n + (N_1 - N_2) - 1.
\end{aligned}
$$

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. The process involved in removing these two terms is to add 1, which represents the 2^n term or the carry-out, to the partial result to eliminate 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 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 also 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.

The following example shows when an *overflow* occurs.

Example 9.4. Addition of two numbers that cause an overflow.

$$
\begin{array}{rcccccl}
 & 0 & 1 & 0 & 1 & & (5) \\
+ & 0 & 0 & 1 & 1 & & (3) \\
\hline
 & 1 & 0 & 0 & 0 & & (\text{-}7) \\
\end{array}
$$

The above example adds two positive numbers but the MSB of the result indicates it is negative. This indicates an *overflow*, which occurs when the result is larger than the maximum range of $2^{n-1} - 1$.

9.2.3 2's-complement Representation

2's-complement representation of an n-bit binary number N is given as

$$N = (0b_{n-2}b_{n-3}\cdots b_1 b_0) \text{ and} \tag{9.9}$$
$$-N = (1\bar{b}_{n-2}\bar{b}_{n-3}\cdots \bar{b}_1 \bar{b}_0 + 1),$$

with a range given by

$$-2^{n-1} \le N \le 2^{n-1} - 1. \tag{9.10}$$

The following shows 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 and 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.

The following examples illustrate adding two numbers of opposite signs.

Example 9.5. Addition of two numbers with opposite signs:

$$
\begin{array}{cccccc}
 & 0 & 1 & 0 & 0 & (4) \\
+ & 1 & 0 & 1 & 0 & (\text{-}6) \\
\hline
 & 1 & 1 & 1 & 0 & (\text{-}2)
\end{array}
\qquad
\begin{array}{cccccl}
 & 0 & 1 & 1 & 0 & (6) \\
+ & 1 & 1 & 0 & 1 & (\text{-}3) \\
\hline
\mathbf{1} & 0 & 0 & 1 & 1 & \text{Discard carry} \\
 & 0 & 0 & 1 & 1 & (3)
\end{array}
$$

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 Equation 9.11. Consider two positive numbers N_1 and N_2. Performing $N_1 - N_2$ is equivalent to

$$N_1 + TC(N_2) = N_1 + 2^n - N_2 = 2^n + (N_1 - N_2). \qquad (9.12)$$

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 $TC(N_1 - N_2)$, 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 also 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.

9.3   Shift Operations

Under Construction!!!

9.4   Basic ALU Design

Under Construction!!!

9.5   Multiplication

Under Construction!!!

9.6 Division

Under Construction!!!

9.7 Floating-Point Number

Under Construction!!!

Appendix A

AVR Instruction Set Summary

This appendix provides descriptions of all the AVR instructions and it serves as a quick reference for assembly programming. There are five categories of instructions in the AVR instruction set:

- Arithmetic and Logic
- Data Transfer
- Branch
- Bit and Bit-test
- MCU control

The following tables list these instructions.

Table A.1: AVR Arithmetic and Logic Instructions

ARITHMETIC AND LOGIC INSTRUCTIONS					
Mnemonics	Operands	Description	Operation	Flags	#Clks
Two Registers					
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
SUB	Rd, Rr	Subtract two Registers	Rd←Rd-Rr	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry two Registers	Rd←Rd-Rr-C	Z,C,N,V,H	1
AND	Rd, Rr	Logical AND Registers	Rd← Rd∧Rr	Z,N,V	1
OR	Rd, Rr	Logical OR Registers	Rd← Rd∨Rr	Z,N,V	1
EOR	Rd, Rr	Exclusive OR Registers	Rd←Rd⊕Rr	Z,N,V	1
				Continued on next page	

277

Table A.1 continued from previous page

Mnemonics	Operands	Description	Operation	Flags	#Clks
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
Register and Constant					
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl←Rdh:Rdl+K	Z,C,N,V,S	2
SUBI	Rd, K	Subtract Constant from Register	Rd←Rd-K	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
ANDI	Rd, K	Logical AND Register and Constant	Rd←Rd∧K	Z,N,V	1
ORI	Rd, K	Logical OR Register and Constant	Rd← Rd∨K	Z,N,V	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∧($FF-K)	Z,N,V	1
One Register					
COM	Rd	One's Complement	Rd←$FF-Rd	Z,C,N,V	1
NEG	Rd	Two's Complement	Rd←$00-Rd	Z,C,N,V,H	1
INC	Rd	Increment	Rd←Rd+1	Z,N,V	1
DEC	Rd	Decrement	Rd←Rd-1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd←Rd∧Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd←Rd⊕Rd	Z,N,V	1
SER	Rd	Set Register	Rd←$FF	None	1

Table A.2: Data Transfer Instructions

DATA TRANSFER INSTRUCTIONS					
Mnemonics	Operands	Description	Operation	Flags	#Clks
Register(s) to Register(s) Move					
MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None	1
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd ← Rr+1:Rr	None	1

Table A.2 – continued from previous page

Mnemonics	Operands	Description	Operation	Flags	#Clks
Load Constant to Register					
LDI	Rd, K	Load Immediate	Rd ← K	None	1
Load from Memory					
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
Store to Memory					
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
Load from Program Memory					
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

Table A.2 – continued from previous page

Mnemonics	Operands	Description	Operation	Flags	#Clks
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
Store to Program Memory					
SPM		Store Program Memory	(Z) ← R1:R0	None	-
Load/Store from/to I/O Register					
IN	Rd, P	In Port	Rd ← P	None	1
OUT	P, Rr	Out Port	P ← Rr	None	1
Stack Manipulation					
PUSH	Rr	Push Register on Stack	STACK ← Rr	None	2
POP	Rd	Pop Register from Stack	Rd ← STACK	None	2

Table A.3: Branch Instructions

BRANCH INSTRUCTIONS[1]					
Mnemonics	Operands	Description	Operation	Flags	#Clks
Jump					
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
Subroutine Calls and Return					
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		Return from Interrupt	PC←STACK	I	4
Compare					
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
Skip if *cond*					
				Continued on next page	

Table A.3 – continued from previous page

Mnemonics	Operands	Description	Operation	Flags	#Clks
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
Conditional Branch					
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$\oplus$V= 0) then PC←PC+k+1	None	1/2
BRLT	k	Branch if Less Than Zero, Signed	if (N$\oplus$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

Table A.3 − continued from previous page

Mnemonics	Operands	Description	Operation	Flags	#Clks
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

Table A.4: Bit and Bit-test Instruction

BIT AND BIT-TEST INSTRUCTIONS					
Mnemonics	Operands	Description	Operation	Flags	#Clks
SBI	P,b	Set Bit in I/O Register	(P,b) ← 1	None	2
CBI	P,b	Clear Bit in I/O Register	(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
				Continued on next page	

Table A.4 – continued from previous page

Mnemonics	Operands	Description	Operation	Flags	#Clks
CLS		Clear Signed Test Flag	$S \leftarrow 0$	S	1
SEV		Set Twos Complement Overflow.	$V \leftarrow 1$	V	1
CLV		Clear Twos Complement Overflow	$V \leftarrow 0$	V	1
SET		Set T in SREG	$T \leftarrow 1$	T	1
CLT		Clear T in SREG	$T \leftarrow 0$	T	1
SEH		Set Half Carry Flag in SREG	$H \leftarrow 1$	H	1
CLH		Clear Half Carry Flag in SREG	$H \leftarrow 0$	H	1

Table A.5: MCU Control Instructions

MCU CONTROL INSTRUCTIONS			
Mnemonics	**Description**	**Flags**	**#Clks**
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

Appendix B

AVR Assembler Directives

The AVR assembler supports a number of directives. These directives are not a part of the AVR instruction set and do not get assembled into machine code. Instead, they are used to adjust the location of the program in memory, initialize memory locations, define variables and names, etc. The set of directives supported by the AVR assembler is shown in Table B.1

Table B.1: AVR Assembler Directives

AVR Assembler Directives	
Directive	**Description**
Header	
.DEVICE	Defines the type of the target processor and the applicable set of instructions. Example usage: `.DEVICE AT90S8515`
.DEF	Defines a symbol to refer to a register. Example usage: `.DEF MyReg = R16`)
.EQU	Defines a symbol and sets its value. This value cannot be changed later. Example usage: `.EQU test = 1234567`)
.SET	Defines a symbol and sets its value. This value can be changed later. Example usage: `.SET io_offset = 0x23`
Continued on next page	

Table B.1 – continued from previous page

Directive	Description
`.INCLUDE`	Includes a file and assembles its content. Example usage: `.INCLUDE "iodefs.asm"`.
Code	
`.CSEG`	Defines the start of a code segment.
`.DB`	Initializes program memory or EEPROM with 8-bit values. The number of inserted bytes must be even; otherwise, an additional zero byte will be inserted by the assembler. Example usage: `array:   .DB 1, 2, 3, 4, 5`
`.DW`	Initializes program memory or EEPROM with 16-bit values. Example usage: `constArry:   .DW 0, 0xFFFF, 0x7FFF, 65536`
`.LISTMAC`	Macros will be listed in the listfile, which contains assembly source code, addresses, and opcodes, generated by the assembler.
`.MACRO`	Defines the beginning of a macro. Example usage: `.MACRO macroname`
`.ENDMACRO`	Defines the end of the macro.
EEPROM	
`.ESEG`	Defines the start of an EEPROM segment.
`.DB`	Initializes program memory or EEPROM with 8-bit values. The number of inserted bytes must be even; otherwise, an additional zero byte will be inserted by the assembler
`.DW`	Initializes program memory or EEPROM with 16-bit values.
SRAM	
`.DSEG`	Defines the start of a new data segment.
`.BYTE`	Reserves memory spaces in the SRAM. Example usage: `buffer:` `.BYTE 20`

Table B.1 – continued from previous page

Directive	Description
Everywhere	
`.ORG`	Defines the address within the respective segment. Example usage: `.ORG 0x0100`
`.LIST`	Generates a listfile, which contains assembly source code, addresses, and opcodes.
`.NOLIST`	Turns off list file generation.
`.INCLUDE`	Includes a file and assembles its content. Example usage: `.INCLUDE "iodefs.asm"`.
`.EXIT`	Indicates the end of the assembler-source code.

Appendix C

AVR I/O Registers – ATmega128

This appendix provides descriptions of all the I/O registers for the AT-mega128 microcontroller. Table C.1 shows the registers in the 64 I/O register space, while Table C.2 shows the registers in the extended I/O register space. In Table C.1, the address of the form $xx represents the I/O address (used by `IN` and `OUT` instructions). On the other hand, the address of the form ($xx) represents the location of the register in memory. Thus, the `LDS` and `STS` instructions can also be used to access these registers. Note that the registers in the extended I/O spec shown in Table C.2 can only be accessed using the `LDS` and `STS` instructions.

Table C.1: 64 I/O Registers

Address	I/O	Register	Description
$3F ($5F)	Status Register	SREG	Status Register
$3E ($5E)	Stack Pointer	SPH	Stack Pointer High Byte
$3D ($5D)		SPL	Stack Pointer Low Byte
$3C ($5C)	Clock	XDIV	XTAL Divide Control Register
$3B ($5B)	Program Memory Store	RAMP	RAM Page Z Select Register
$3A ($5A)		EICRB	External Interrupt Control Register B
$39 ($59)	External Interrupts	EIMSK	External Interrupt Mask Register
$38 ($58)		EIFR	External Interrupt Flag Register
$37 ($57)	Timer/Counter0 & 1	TIMSK	Timer Interrupt Mask Register
$36 ($56)		TIFR	Timer Interrupt Flag Register
			Continued on next page

289

Table C.1 – *Continued from previous page*

Address	I/O	Register	Description
$35 ($55)	MCU	MCUCR	MCU Control Register
$34 ($54)		MCUCSR	MCU Control and Status Register
$33 ($53)	Timer/Counter0	TCCR0	Timer/Counter Control Register 0
$32 ($52)		TCNT0	Timer/Counter0 (8 Bit)
$31 ($51)		OCR0	Timer/Counter0 Output Compare Register
$30 ($50)		ASSR	Asynchronous Status Register
$2F ($4F)	Timer/Counter1	TCCR1A	Timer/Counter1 Control Register A
$2E ($4E)		TCCR1B	Timer/Counter1 Control Register B
$2D ($4D)		TCNT1H	Timer/Counter1 High Byte
$2C ($4C)		TCNT1L	Timer/Counter1 Low Byte
$2B ($4B)		OCR1AH	Output Compare Register 1A High Byte
$2A ($4A)		OCR1AL	Output Compare Register 1A Low Byte
$29 ($49)		OCR1BH	Output Compare Register 1B High Byte
$28 ($48)		OCR1BL	Output Compare Register 1B Low Byte
$27 ($47)		ICR1H	Timer/Counter1–Input Capture Register 1 High Byte
$26 ($46)		ICR1L	Timer/Counter1–Input Capture Register 1 Low Byte
$25 ($45)	Timer/Counter2	TCCR2	Timer/Counter Control Register 2
$24 ($44)		TCNT2	Timer/Counter2 (8 Bit)
$23 ($43)		OCR2	Output Compare Register 2
$22 ($42)	Debugging	OCDR	On-chip Debug Register
$21 ($41)	Watchdog Timer	WDTCR	Watchdog Timer Control Register
$20 ($40)	Special Function	SFIOR	Special Function IO Register
$1F ($3F)	EEPROM	EEARH	EEPROM Address Register High Byte
$1E ($3E)		EEARL	EEPROM Address Register Low Byte
$1D ($3D)		EEDR	EEPROM Data Register
$1C ($3C)		EECR	EEPROM Control Register
$1B ($3B)	PORTA	PORTA	Port A Data Register
$1A ($3A)		DDRA	Port A Data Direction Register
$19 ($39)		PINA	Port A Input Pins Address
$18 ($38)	PORTB	PORTB	Port B Data Register
$17 ($37)		DDRB	Port B Data Direction Register
$16 ($36)		PINB	Port B Input Pins Address
$15 ($35)	PORTC	PORTC	Port C Data Register
$14 ($34)		DDRC	Port C Data Direction Register
$13 ($33)		PINC	Port C Input Pins Address

Table C.1 – *Continued from previous page*

Address	I/O	Register	Description
$12 ($32)		PORTD	Port D Data Register
$11 ($31)	PORTD	DDRD	Port D Data Direction Register
$10 ($30)		PIND	Port D Input Pins Address
$0F ($2F)		SPDR	SPI Data Register
$0E ($2E)	SPI	SPSR	SPI Status Register
$0D ($2D)		SPCR	SPI Control Register
$0C ($2C)		UDR0	USART0 I/O Data Register
$0B ($2B)	USART0	UCSR0A	USART0 Control and Status Register A
$0A ($2A)		UCSR0B	USART0 Control and Status Register B
$09 ($29)		UBRR0L	USART0 Baud Rate Register Low
$08 ($28)	Analog Comparator	ACSR	Analog Comparator Control and Status Register
$07 ($27)		ADMUX	ADC Multiplexer Selection Register
$06 ($26)	ADC	ADCSRA	ADC Control and StatusRegister A
$05 ($25)		ADCH	ADC Data Register High Byte
$04 ($24)		ADCL	ADC Data Register Low Byte
$03 ($23)		PORTE	Port E Data Registe
$02 ($22)	PORTE	DDRE	Port E Data Direction Register
$01 ($21)		PINE	Port E Input Pins Address
$00 ($20)	PORTF	PINF	Port F Input Pins Address

Table C.2: Extended I/O Registers

Address	I/O	Register	Description
($FF)		Reserved	-
(...)		Reserved	-
($9E)		Reserved	-
($9D)		UCSR1C	USART 1 Control and Status Register A
($9C)		UDR1	USART 1 I/O Data Register
($9B)		UCSR1A	USART 1 Control and Status Register A
($9A)	USART1	UCSR1B	USART 1 Control and Status Register B
($99)		UBRR1L	USART1 Baud Rate Register Low
($98)		UBRR1H	USART1 Baud Rate Register High
($97)		Reserved	-
($96)		Reserved	-
			Continued on next page

Table C.2 – *Continued from previous page*

Address	I/O	Register	Description
($95)		UCSR0C	USART 0 Control and Status Register C
($94)		Reserved	-
($93)	USART0	Reserved	-
($92)		Reserved	-
($91)		Reserved	-
($90)		UBRR0H	USART0 Baud Rate Register High
($8F)		Reserved	-
($8E)		Reserved	-
($8D)		Reserved	-
($8C)		TCCR3C	Timer/Counter1 High Byte
($8B)		TCCR3A	Timer/Counter3 Control Register A
($8A)		TCCR3B	Timer/Counter3 Control Register B
($89)		TCNT3H	Timer/Counter3–Counter Register High Byte
($88)		TCNT3L	Timer/Counter3–Counter Register Low Byte
($87)		OCR3AH	Timer/Counter3–Output Compare Register A High Byte
($86)		OCR3AL	Timer/Counter3–Output Compare Register A Low Byte
($85)		OCR3BH	Timer/Counter3–Output Compare Register B High Byte
($84)		OCR3BL	Timer/Counter3–Output Compare Register B Low Byte
($83)		OCR3CH	Timer/Counter3–Output Compare Register C High Byte
($82)	Timer/Counter1 & 3	OCR3CL	Timer/Counter3–Output Compare Register C Low Byte
($81)		ICR3H	Timer/Counter3–Input Capture Register High Byte
($80)		ICR3L	Timer/Counter3–Input Capture Register Low Byte
($7F)		Reserved	-
($7E)		Reserved	-
($7D)		ETIMSK	Extended Timer Interrupt Mask Register
($7C)		ETIFR	Extended Timer Interrupt Flag Register
($7B)		Reserved	-
($7A)		TCCR1C	Timer/Counter1 Control Register C
($79)		OCR1CH	Timer/Counter1 ? Output Compare Register C High Byte
($78)		OCR1CL	Timer/Counter1 ? Output Compare Register C Low Byte
($77)		Reserved	-
($76)		Reserved	-
($75)		Reserved	-
			Continued on next page

Table C.2 – *Continued from previous page*

Address	I/O	Register	Description
($74)		TWCR	TWI Control Register
($73)		TWDR	TWI Data Register
($72)	TWI	TWAR	TWI Address Register
($71)		TWSR	TWI Status Register
($70)		TWBR	TWI Bit Rate Register
($6F)	Internal Oscillator	OSCCAL	Oscillator Calibration Register
($6E)		Reserved	-
($6D)	External Memory	XMCRA	External Memory Control Registers A
($6C)		XMCRB	External Memory Control Register B
($6B)		Reserved	-
($6A)	External Interrupts	EICRA	External Interrupt Control Register A
($69)		Reserved	-
($68)	Boot Loader	SPMCSR	Store Program Memory Control and Status Register
($67)		Reserved	-
($66)		Reserved	-
($65)		PORTG	Port G Data Registe
($64)	PORTG	DDRG	Port G Data Direction Register
($63)		PING	Port G Input Pins Address
($62)	PORTF	PORTF	Port F Data Register
($61)		DDRF	Port F Data Direction Register
($60)		Reserved	-

Appendix D

AVR ATmega128 Definition File (m128def.inc)

This appendix provides the `m128def.inc` definition file that contains all the I/O register names, I/O register bit names, names of high and low bytes of X, Y, and X address registers, and the highest address for the internal SRAM for the ATmega128 microcontroller.

```
;***** THIS IS A MACHINE GENERATED FILE - DO NOT EDIT ********************
;***** Created: 2007-09-11 14:24 ******* Source: ATmega128.xml ***********
;************************************************************************
;* A P P L I C A T I O N   N O T E   F O R   T H E   A V R   F A M I L Y
;*
;* Number          : AVR000
;* File Name        : "m128def.inc"
;* Title            : Register/Bit Definitions for the ATmega128
;* Date             : 2007-09-11
;* Version          : 2.24
;* Support E-mail   : avr@atmel.com
;* Target MCU       : ATmega128
;*
;* DESCRIPTION
;* When including this file in the assembly program file, all I/O register
;* names and I/O register bit names appearing in the data book can be used.
;* In addition, the six registers forming the three data pointers X, Y and
;* Z have been assigned names XL - ZH. Highest RAM address for Internal
;* SRAM is also defined
;*
;* The Register names are represented by their hexadecimal address.
;*
;* The Register Bit names are represented by their bit number (0-7).
;*
;* Please observe the difference in using the bit names with instructions
;* such as "sbr"/"cbr" (set/clear bit in register) and "sbrs"/"sbrc"
;* (skip if bit in register set/cleared). The following example illustrates
```

```
;* this:
;*
;* in    r16,PORTB           ;read PORTB latch
;* sbr   r16,(1<<PB6)+(1<<PB5) ;set PB6 and PB5 (use masks, not bit#)
;* out   PORTB,r16           ;output to PORTB
;*
;* in    r16,TIFR            ;read the Timer Interrupt Flag Register
;* sbrc  r16,TOV0            ;test the overflow flag (use bit#)
;* rjmp  TOV0_is_set         ;jump if set
;* ...                       ;otherwise do something else
;**************************************************************************

#ifndef _M128DEF_INC_
#define _M128DEF_INC_


#pragma partinc 0

; ***** SPECIFY DEVICE *****************************************************
.device ATmega128
#pragma AVRPART ADMIN PART_NAME ATmega128
.equ    SIGNATURE_000  = 0x1e
.equ    SIGNATURE_001  = 0x97
.equ    SIGNATURE_002  = 0x02

#pragma AVRPART CORE CORE_VERSION V2E


; ***** I/O REGISTER DEFINITIONS ******************************************
; NOTE:
; Definitions marked "MEMORY MAPPED"are extended I/O ports
; and cannot be used with IN/OUT instructions
.equ    UCSR1C = 0x9d  ; MEMORY MAPPED
.equ    UDR1   = 0x9c  ; MEMORY MAPPED
.equ    UCSR1A = 0x9b  ; MEMORY MAPPED
.equ    UCSR1B = 0x9a  ; MEMORY MAPPED
.equ    UBRR1H = 0x98  ; MEMORY MAPPED
.equ    UBRR1L = 0x99  ; MEMORY MAPPED
.equ    UCSR0C = 0x95  ; MEMORY MAPPED
.equ    UBRR0H = 0x90  ; MEMORY MAPPED
.equ    TCCR3C = 0x8c  ; MEMORY MAPPED
.equ    TCCR3A = 0x8b  ; MEMORY MAPPED
.equ    TCCR3B = 0x8a  ; MEMORY MAPPED
.equ    TCNT3L = 0x88  ; MEMORY MAPPED
.equ    TCNT3H = 0x89  ; MEMORY MAPPED
.equ    OCR3AL = 0x86  ; MEMORY MAPPED
.equ    OCR3AH = 0x87  ; MEMORY MAPPED
.equ    OCR3BL = 0x84  ; MEMORY MAPPED
.equ    OCR3BH = 0x85  ; MEMORY MAPPED
.equ    OCR3CL = 0x82  ; MEMORY MAPPED
.equ    OCR3CH = 0x83  ; MEMORY MAPPED
.equ    ICR3L  = 0x80  ; MEMORY MAPPED
.equ    ICR3H  = 0x81  ; MEMORY MAPPED
.equ    ETIMSK = 0x7d  ; MEMORY MAPPED
.equ    ETIFR  = 0x7c  ; MEMORY MAPPED
.equ    TCCR1C = 0x7a  ; MEMORY MAPPED
```

```
.equ    OCR1CL  = 0x78  ; MEMORY MAPPED
.equ    OCR1CH  = 0x79  ; MEMORY MAPPED
.equ    TWCR    = 0x74  ; MEMORY MAPPED
.equ    TWDR    = 0x73  ; MEMORY MAPPED
.equ    TWAR    = 0x72  ; MEMORY MAPPED
.equ    TWSR    = 0x71  ; MEMORY MAPPED
.equ    TWBR    = 0x70  ; MEMORY MAPPED
.equ    OSCCAL  = 0x6f  ; MEMORY MAPPED
.equ    XMCRA   = 0x6d  ; MEMORY MAPPED
.equ    XMCRB   = 0x6c  ; MEMORY MAPPED
.equ    EICRA   = 0x6a  ; MEMORY MAPPED
.equ    SPMCSR  = 0x68  ; MEMORY MAPPED
.equ    PORTG   = 0x65  ; MEMORY MAPPED
.equ    DDRG    = 0x64  ; MEMORY MAPPED
.equ    PING    = 0x63  ; MEMORY MAPPED
.equ    PORTF   = 0x62  ; MEMORY MAPPED
.equ    DDRF    = 0x61  ; MEMORY MAPPED
.equ    SREG    = 0x3f
.equ    SPL     = 0x3d
.equ    SPH     = 0x3e
.equ    XDIV    = 0x3c
.equ    RAMPZ   = 0x3b
.equ    EICRB   = 0x3a
.equ    EIMSK   = 0x39
.equ    EIFR    = 0x38
.equ    TIMSK   = 0x37
.equ    TIFR    = 0x36
.equ    MCUCR   = 0x35
.equ    MCUCSR  = 0x34
.equ    TCCR0   = 0x33
.equ    TCNT0   = 0x32
.equ    OCR0    = 0x31
.equ    ASSR    = 0x30
.equ    TCCR1A  = 0x2f
.equ    TCCR1B  = 0x2e
.equ    TCNT1L  = 0x2c
.equ    TCNT1H  = 0x2d
.equ    OCR1AL  = 0x2a
.equ    OCR1AH  = 0x2b
.equ    OCR1BL  = 0x28
.equ    OCR1BH  = 0x29
.equ    ICR1L   = 0x26
.equ    ICR1H   = 0x27
.equ    TCCR2   = 0x25
.equ    TCNT2   = 0x24
.equ    OCR2    = 0x23
.equ    OCDR    = 0x22
.equ    WDTCR   = 0x21
.equ    SFIOR   = 0x20
.equ    EEARL   = 0x1e
.equ    EEARH   = 0x1f
.equ    EEDR    = 0x1d
.equ    EECR    = 0x1c
.equ    PORTA   = 0x1b
.equ    DDRA    = 0x1a
.equ    PINA    = 0x19
```

```
.equ    PORTB   = 0x18
.equ    DDRB    = 0x17
.equ    PINB    = 0x16
.equ    PORTC   = 0x15
.equ    DDRC    = 0x14
.equ    PINC    = 0x13
.equ    PORTD   = 0x12
.equ    DDRD    = 0x11
.equ    PIND    = 0x10
.equ    SPDR    = 0x0f
.equ    SPSR    = 0x0e
.equ    SPCR    = 0x0d
.equ    UDR0    = 0x0c
.equ    UCSR0A  = 0x0b
.equ    UCSR0B  = 0x0a
.equ    UBRR0L  = 0x09
.equ    ACSR    = 0x08
.equ    ADMUX   = 0x07
.equ    ADCSRA  = 0x06
.equ    ADCH    = 0x05
.equ    ADCL    = 0x04
.equ    PORTE   = 0x03
.equ    DDRE    = 0x02
.equ    PINE    = 0x01
.equ    PINF    = 0x00


; ***** BIT DEFINITIONS **************************************************

; ***** ANALOG_COMPARATOR ************
; SFIOR - Special Function IO Register
.equ    ACME    = 3       ; Analog Comparator Multiplexer Enable

; ACSR - Analog Comparator Control And Status Register
.equ    ACIS0   = 0       ; Analog Comparator Interrupt Mode Select bit 0
.equ    ACIS1   = 1       ; Analog Comparator Interrupt Mode Select bit 1
.equ    ACIC    = 2       ; Analog Comparator Input Capture Enable
.equ    ACIE    = 3       ; Analog Comparator Interrupt Enable
.equ    ACI     = 4       ; Analog Comparator Interrupt Flag
.equ    ACO     = 5       ; Analog Compare Output
.equ    ACBG    = 6       ; Analog Comparator Bandgap Select
.equ    ACD     = 7       ; Analog Comparator Disable


; ***** SPI *************************
; SPDR - SPI Data Register
.equ    SPDR0   = 0       ; SPI Data Register bit 0
.equ    SPDR1   = 1       ; SPI Data Register bit 1
.equ    SPDR2   = 2       ; SPI Data Register bit 2
.equ    SPDR3   = 3       ; SPI Data Register bit 3
.equ    SPDR4   = 4       ; SPI Data Register bit 4
.equ    SPDR5   = 5       ; SPI Data Register bit 5
.equ    SPDR6   = 6       ; SPI Data Register bit 6
.equ    SPDR7   = 7       ; SPI Data Register bit 7

; SPSR - SPI Status Register
```

```
        .equ    SPI2X   = 0     ; Double SPI Speed Bit
        .equ    WCOL    = 6     ; Write Collision Flag
        .equ    SPIF    = 7     ; SPI Interrupt Flag

; SPCR - SPI Control Register
        .equ    SPR0    = 0     ; SPI Clock Rate Select 0
        .equ    SPR1    = 1     ; SPI Clock Rate Select 1
        .equ    CPHA    = 2     ; Clock Phase
        .equ    CPOL    = 3     ; Clock polarity
        .equ    MSTR    = 4     ; Master/Slave Select
        .equ    DORD    = 5     ; Data Order
        .equ    SPE     = 6     ; SPI Enable
        .equ    SPIE    = 7     ; SPI Interrupt Enable


; ***** TWI **************************
; TWBR - TWI Bit Rate register
        .equ    I2BR    = TWBR  ; For compatibility
        .equ    TWBR0   = 0     ;
        .equ    TWBR1   = 1     ;
        .equ    TWBR2   = 2     ;
        .equ    TWBR3   = 3     ;
        .equ    TWBR4   = 4     ;
        .equ    TWBR5   = 5     ;
        .equ    TWBR6   = 6     ;
        .equ    TWBR7   = 7     ;

; TWCR - TWI Control Register
        .equ    I2CR    = TWCR  ; For compatibility
        .equ    TWIE    = 0     ; TWI Interrupt Enable
        .equ    I2IE    = TWIE  ; For compatibility
        .equ    TWEN    = 2     ; TWI Enable Bit
        .equ    I2EN    = TWEN  ; For compatibility
        .equ    ENI2C   = TWEN  ; For compatibility
        .equ    TWWC    = 3     ; TWI Write Collition Flag
        .equ    I2WC    = TWWC  ; For compatibility
        .equ    TWSTO   = 4     ; TWI Stop Condition Bit
        .equ    I2STO   = TWSTO ; For compatibility
        .equ    TWSTA   = 5     ; TWI Start Condition Bit
        .equ    I2STA   = TWSTA ; For compatibility
        .equ    TWEA    = 6     ; TWI Enable Acknowledge Bit
        .equ    I2EA    = TWEA  ; For compatibility
        .equ    TWINT   = 7     ; TWI Interrupt Flag
        .equ    I2INT   = TWINT ; For compatibility

; TWSR - TWI Status Register
        .equ    I2SR    = TWSR  ; For compatibility
        .equ    TWPS0   = 0     ; TWI Prescaler
        .equ    TWS0    = TWPS0 ; For compatibility
        .equ    I2GCE   = TWPS0 ; For compatibility
        .equ    TWPS1   = 1     ; TWI Prescaler
        .equ    TWS1    = TWPS1 ; For compatibility
        .equ    TWS3    = 3     ; TWI Status
        .equ    I2S3    = TWS3  ; For compatibility
        .equ    TWS4    = 4     ; TWI Status
        .equ    I2S4    = TWS4  ; For compatibility
```

```
.equ    TWS5    = 5      ; TWI Status
.equ    I2S5    = TWS5   ; For compatibility
.equ    TWS6    = 6      ; TWI Status
.equ    I2S6    = TWS6   ; For compatibility
.equ    TWS7    = 7      ; TWI Status
.equ    I2S7    = TWS7   ; For compatibility

; TWDR - TWI Data register
.equ    I2DR    = TWDR   ; For compatibility
.equ    TWD0    = 0      ; TWI Data Register Bit 0
.equ    TWD1    = 1      ; TWI Data Register Bit 1
.equ    TWD2    = 2      ; TWI Data Register Bit 2
.equ    TWD3    = 3      ; TWI Data Register Bit 3
.equ    TWD4    = 4      ; TWI Data Register Bit 4
.equ    TWD5    = 5      ; TWI Data Register Bit 5
.equ    TWD6    = 6      ; TWI Data Register Bit 6
.equ    TWD7    = 7      ; TWI Data Register Bit 7

; TWAR - TWI (Slave) Address register
.equ    I2AR    = TWAR   ; For compatibility
.equ    TWGCE   = 0      ; TWI General Call Recognition Enable Bit
.equ    TWA0    = 1      ; TWI (Slave) Address register Bit 0
.equ    TWA1    = 2      ; TWI (Slave) Address register Bit 1
.equ    TWA2    = 3      ; TWI (Slave) Address register Bit 2
.equ    TWA3    = 4      ; TWI (Slave) Address register Bit 3
.equ    TWA4    = 5      ; TWI (Slave) Address register Bit 4
.equ    TWA5    = 6      ; TWI (Slave) Address register Bit 5
.equ    TWA6    = 7      ; TWI (Slave) Address register Bit 6


; ***** USART0 ***********************
; UDR0 - USART I/O Data Register
.equ    UDR00   = 0      ; USART I/O Data Register bit 0
.equ    UDR01   = 1      ; USART I/O Data Register bit 1
.equ    UDR02   = 2      ; USART I/O Data Register bit 2
.equ    UDR03   = 3      ; USART I/O Data Register bit 3
.equ    UDR04   = 4      ; USART I/O Data Register bit 4
.equ    UDR05   = 5      ; USART I/O Data Register bit 5
.equ    UDR06   = 6      ; USART I/O Data Register bit 6
.equ    UDR07   = 7      ; USART I/O Data Register bit 7

; UCSR0A - USART Control and Status Register A
.equ    MPCM0   = 0      ; Multi-processor Communication Mode
.equ    U2X0    = 1      ; Double the USART transmission speed
.equ    UPE0    = 2      ; Parity Error
.equ    DOR0    = 3      ; Data overRun
.equ    FE0     = 4      ; Framing Error
.equ    UDRE0   = 5      ; USART Data Register Empty
.equ    TXC0    = 6      ; USART Transmit Complete
.equ    RXC0    = 7      ; USART Receive Complete

; UCSR0B - USART Control and Status Register B
.equ    TXB80   = 0      ; Transmit Data Bit 8
.equ    RXB80   = 1      ; Receive Data Bit 8
.equ    UCSZ02  = 2      ; Character Size
.equ    UCSZ2   = UCSZ02        ; For compatibility
```

```
        .equ    TXEN0   = 3     ; Transmitter Enable
        .equ    RXEN0   = 4     ; Receiver Enable
        .equ    UDRIE0  = 5     ; USART Data register Empty Interrupt Enable
        .equ    TXCIE0  = 6     ; TX Complete Interrupt Enable
        .equ    RXCIE0  = 7     ; RX Complete Interrupt Enable

    ; UCSR0C - USART Control and Status Register C
        .equ    UCPOL0  = 0     ; Clock Polarity
        .equ    UCSZ00  = 1     ; Character Size
        .equ    UCSZ01  = 2     ; Character Size
        .equ    USBS0   = 3     ; Stop Bit Select
        .equ    UPM00   = 4     ; Parity Mode Bit 0
        .equ    UPM01   = 5     ; Parity Mode Bit 1
        .equ    UMSEL0  = 6     ; USART Mode Select

    ; UBRR0H - USART Baud Rate Register High Byte
        .equ    UBRR8   = 0     ; USART Baud Rate Register bit 8
        .equ    UBRR9   = 1     ; USART Baud Rate Register bit 9
        .equ    UBRR10  = 2     ; USART Baud Rate Register bit 10
        .equ    UBRR11  = 3     ; USART Baud Rate Register bit 11

    ; UBRR0L - USART Baud Rate Register Low Byte
        .equ    UBRR0   = 0     ; USART Baud Rate Register bit 0
        .equ    UBRR1   = 1     ; USART Baud Rate Register bit 1
        .equ    UBRR2   = 2     ; USART Baud Rate Register bit 2
        .equ    UBRR3   = 3     ; USART Baud Rate Register bit 3
        .equ    UBRR4   = 4     ; USART Baud Rate Register bit 4
        .equ    UBRR5   = 5     ; USART Baud Rate Register bit 5
        .equ    UBRR6   = 6     ; USART Baud Rate Register bit 6
        .equ    UBRR7   = 7     ; USART Baud Rate Register bit 7


    ; ***** USART1 ***********************
    ; UDR1 - USART I/O Data Register
        .equ    UDR10   = 0     ; USART I/O Data Register bit 0
        .equ    UDR11   = 1     ; USART I/O Data Register bit 1
        .equ    UDR12   = 2     ; USART I/O Data Register bit 2
        .equ    UDR13   = 3     ; USART I/O Data Register bit 3
        .equ    UDR14   = 4     ; USART I/O Data Register bit 4
        .equ    UDR15   = 5     ; USART I/O Data Register bit 5
        .equ    UDR16   = 6     ; USART I/O Data Register bit 6
        .equ    UDR17   = 7     ; USART I/O Data Register bit 7

    ; UCSR1A - USART Control and Status Register A
        .equ    MPCM1   = 0     ; Multi-processor Communication Mode
        .equ    U2X1    = 1     ; Double the USART transmission speed
        .equ    UPE1    = 2     ; Parity Error
        .equ    DOR1    = 3     ; Data overRun
        .equ    FE1     = 4     ; Framing Error
        .equ    UDRE1   = 5     ; USART Data Register Empty
        .equ    TXC1    = 6     ; USART Transmitt Complete
        .equ    RXC1    = 7     ; USART Receive Complete

    ; UCSR1B - USART Control and Status Register B
        .equ    TXB81   = 0     ; Transmit Data Bit 8
        .equ    RXB81   = 1     ; Receive Data Bit 8
```

```
.equ    UCSZ12  = 2      ; Character Size
.equ    TXEN1   = 3      ; Transmitter Enable
.equ    RXEN1   = 4      ; Receiver Enable
.equ    UDRIE1  = 5      ; USART Data register Empty Interrupt Enable
.equ    TXCIE1  = 6      ; TX Complete Interrupt Enable
.equ    RXCIE1  = 7      ; RX Complete Interrupt Enable

; UCSR1C - USART Control and Status Register C
.equ    UCPOL1  = 0      ; Clock Polarity
.equ    UCSZ10  = 1      ; Character Size
.equ    UCSZ11  = 2      ; Character Size
.equ    USBS1   = 3      ; Stop Bit Select
.equ    UPM10   = 4      ; Parity Mode Bit 0
.equ    UPM11   = 5      ; Parity Mode Bit 1
.equ    UMSEL1  = 6      ; USART Mode Select

; UBRR1H - USART Baud Rate Register Hight Byte
;.equ   UBRR8   = 0      ; USART Baud Rate Register bit 8
;.equ   UBRR9   = 1      ; USART Baud Rate Register bit 9
;.equ   UBRR10  = 2      ; USART Baud Rate Register bit 10
;.equ   UBRR11  = 3      ; USART Baud Rate Register bit 11

; UBRR1L - USART Baud Rate Register Low Byte
;.equ   UBRR0   = 0      ; USART Baud Rate Register bit 0
;.equ   UBRR1   = 1      ; USART Baud Rate Register bit 1
;.equ   UBRR2   = 2      ; USART Baud Rate Register bit 2
;.equ   UBRR3   = 3      ; USART Baud Rate Register bit 3
;.equ   UBRR4   = 4      ; USART Baud Rate Register bit 4
;.equ   UBRR5   = 5      ; USART Baud Rate Register bit 5
;.equ   UBRR6   = 6      ; USART Baud Rate Register bit 6
;.equ   UBRR7   = 7      ; USART Baud Rate Register bit 7


; ***** CPU **************************
; SREG - Status Register
.equ    SREG_C  = 0      ; Carry Flag
.equ    SREG_Z  = 1      ; Zero Flag
.equ    SREG_N  = 2      ; Negative Flag
.equ    SREG_V  = 3      ; Two's Complement Overflow Flag
.equ    SREG_S  = 4      ; Sign Bit
.equ    SREG_H  = 5      ; Half Carry Flag
.equ    SREG_T  = 6      ; Bit Copy Storage
.equ    SREG_I  = 7      ; Global Interrupt Enable

; MCUCR - MCU Control Register
.equ    IVCE    = 0      ; Interrupt Vector Change Enable
.equ    IVSEL   = 1      ; Interrupt Vector Select
.equ    SM2     = 2      ; Sleep Mode Select
.equ    SM0     = 3      ; Sleep Mode Select
.equ    SM1     = 4      ; Sleep Mode Select
.equ    SE      = 5      ; Sleep Enable
.equ    SRW10   = 6      ; External SRAM Wait State Select
.equ    SRE     = 7      ; External SRAM Enable

; XMCRA - External Memory Control Register A
.equ    SRW11   = 1      ; Wait state select bit upper page
```

```
            .equ   SRW00  = 2      ; Wait state select bit lower page
            .equ   SRW01  = 3      ; Wait state select bit lower page
            .equ   SRL0   = 4      ; Wait state page limit
            .equ   SRL1   = 5      ; Wait state page limit
            .equ   SRL2   = 6      ; Wait state page limit

    ; XMCRB - External Memory Control Register B
            .equ   XMM0   = 0      ; External Memory High Mask
            .equ   XMM1   = 1      ; External Memory High Mask
            .equ   XMM2   = 2      ; External Memory High Mask
            .equ   XMBK   = 7      ; External Memory Bus Keeper Enable

    ; OSCCAL - Oscillator Calibration Value
            .equ   CAL0   = 0      ; Oscillator Calibration Value
            .equ   CAL1   = 1      ; Oscillator Calibration Value
            .equ   CAL2   = 2      ; Oscillator Calibration Value
            .equ   CAL3   = 3      ; Oscillator Calibration Value
            .equ   CAL4   = 4      ; Oscillator Calibration Value
            .equ   CAL5   = 5      ; Oscillator Calibration Value
            .equ   CAL6   = 6      ; Oscillator Calibration Value
            .equ   CAL7   = 7      ; Oscillator Calibration Value

    ; XDIV - XTAL Divide Control Register
            .equ   XDIV0  = 0      ; XTAl Divide Select Bit 0
            .equ   XDIV1  = 1      ; XTAl Divide Select Bit 1
            .equ   XDIV2  = 2      ; XTAl Divide Select Bit 2
            .equ   XDIV3  = 3      ; XTAl Divide Select Bit 3
            .equ   XDIV4  = 4      ; XTAl Divide Select Bit 4
            .equ   XDIV5  = 5      ; XTAl Divide Select Bit 5
            .equ   XDIV6  = 6      ; XTAl Divide Select Bit 6
            .equ   XDIVEN = 7      ; XTAL Divide Enable

    ; MCUCSR - MCU Control And Status Register
            .equ   PORF   = 0      ; Power-on reset flag
            .equ   EXTRF  = 1      ; External Reset Flag
            .equ   BORF   = 2      ; Brown-out Reset Flag
            .equ   WDRF   = 3      ; Watchdog Reset Flag
            .equ   JTRF   = 4      ; JTAG Reset Flag
            .equ   JTD    = 7      ; JTAG Interface Disable

    ; RAMPZ - RAM Page Z Select Register
            .equ   RAMPZ0 = 0      ; RAM Page Z Select Register Bit 0


    ; ***** BOOT_LOAD ********************
    ; SPMCSR - Store Program Memory Control Register
            .equ   SPMCR  = SPMCSR        ; For compatibility
            .equ   SPMEN  = 0      ; Store Program Memory Enable
            .equ   PGERS  = 1      ; Page Erase
            .equ   PGWRT  = 2      ; Page Write
            .equ   BLBSET = 3      ; Boot Lock Bit Set
            .equ   RWWSRE = 4      ; Read While Write section read enable
            .equ   ASRE   = RWWSRE        ; For compatibility
            .equ   RWWSB  = 6      ; Read While Write Section Busy
            .equ   ASB    = RWWSB  ; For compatibility
            .equ   SPMIE  = 7      ; SPM Interrupt Enable
```

```
; ***** JTAG **************************
; OCDR - On-Chip Debug Related Register in I/O Memory
.equ    OCDR0   = 0     ; On-Chip Debug Register Bit 0
.equ    OCDR1   = 1     ; On-Chip Debug Register Bit 1
.equ    OCDR2   = 2     ; On-Chip Debug Register Bit 2
.equ    OCDR3   = 3     ; On-Chip Debug Register Bit 3
.equ    OCDR4   = 4     ; On-Chip Debug Register Bit 4
.equ    OCDR5   = 5     ; On-Chip Debug Register Bit 5
.equ    OCDR6   = 6     ; On-Chip Debug Register Bit 6
.equ    OCDR7   = 7     ; On-Chip Debug Register Bit 7
.equ    IDRD    = OCDR7 ; For compatibility

; MCUCSR - MCU Control And Status Register
;.equ   JTRF    = 4     ; JTAG Reset Flag
;.equ   JTD     = 7     ; JTAG Interface Disable


; ***** MISC **************************
; SFIOR - Special Function IO Register
.equ    PSR321  = 0     ; Prescaler Reset Timer/Counter3, Timer/Counter2, and Timer/Counter1
.equ    PSR1    = PSR321        ; For compatibility
.equ    PSR2    = PSR321        ; For compatibility
.equ    PSR3    = PSR321        ; For compatibility
.equ    PSR0    = 1     ; Prescaler Reset Timer/Counter0
.equ    PUD     = 2     ; Pull Up Disable
;.equ   ACME    = 3     ; Analog Comparator Multiplexer Enable
.equ    TSM     = 7     ; Timer/Counter Synchronization Mode


; ***** EXTERNAL_INTERRUPT ***********
; EICRA - External Interrupt Control Register A
.equ    ISC00   = 0     ; External Interrupt Sense Control Bit
.equ    ISC01   = 1     ; External Interrupt Sense Control Bit
.equ    ISC10   = 2     ; External Interrupt Sense Control Bit
.equ    ISC11   = 3     ; External Interrupt Sense Control Bit
.equ    ISC20   = 4     ; External Interrupt Sense Control Bit
.equ    ISC21   = 5     ; External Interrupt Sense Control Bit
.equ    ISC30   = 6     ; External Interrupt Sense Control Bit
.equ    ISC31   = 7     ; External Interrupt Sense Control Bit

; EICRB - External Interrupt Control Register B
.equ    ISC40   = 0     ; External Interrupt 7-4 Sense Control Bit
.equ    ISC41   = 1     ; External Interrupt 7-4 Sense Control Bit
.equ    ISC50   = 2     ; External Interrupt 7-4 Sense Control Bit
.equ    ISC51   = 3     ; External Interrupt 7-4 Sense Control Bit
.equ    ISC60   = 4     ; External Interrupt 7-4 Sense Control Bit
.equ    ISC61   = 5     ; External Interrupt 7-4 Sense Control Bit
.equ    ISC70   = 6     ; External Interrupt 7-4 Sense Control Bit
.equ    ISC71   = 7     ; External Interrupt 7-4 Sense Control Bit

; EIMSK - External Interrupt Mask Register
.equ    GICR    = EIMSK ; For compatibility
.equ    GIMSK   = EIMSK ; For compatibility
.equ    INT0    = 0     ; External Interrupt Request 0 Enable
```

```
.equ    INT1    = 1     ; External Interrupt Request 1 Enable
.equ    INT2    = 2     ; External Interrupt Request 2 Enable
.equ    INT3    = 3     ; External Interrupt Request 3 Enable
.equ    INT4    = 4     ; External Interrupt Request 4 Enable
.equ    INT5    = 5     ; External Interrupt Request 5 Enable
.equ    INT6    = 6     ; External Interrupt Request 6 Enable
.equ    INT7    = 7     ; External Interrupt Request 7 Enable

; EIFR - External Interrupt Flag Register
.equ    GIFR    = EIFR  ; For compatibility
.equ    INTF0   = 0     ; External Interrupt Flag 0
.equ    INTF1   = 1     ; External Interrupt Flag 1
.equ    INTF2   = 2     ; External Interrupt Flag 2
.equ    INTF3   = 3     ; External Interrupt Flag 3
.equ    INTF4   = 4     ; External Interrupt Flag 4
.equ    INTF5   = 5     ; External Interrupt Flag 5
.equ    INTF6   = 6     ; External Interrupt Flag 6
.equ    INTF7   = 7     ; External Interrupt Flag 7


; ***** EEPROM ***********************
; EEDR - EEPROM Data Register
.equ    EEDR0   = 0     ; EEPROM Data Register bit 0
.equ    EEDR1   = 1     ; EEPROM Data Register bit 1
.equ    EEDR2   = 2     ; EEPROM Data Register bit 2
.equ    EEDR3   = 3     ; EEPROM Data Register bit 3
.equ    EEDR4   = 4     ; EEPROM Data Register bit 4
.equ    EEDR5   = 5     ; EEPROM Data Register bit 5
.equ    EEDR6   = 6     ; EEPROM Data Register bit 6
.equ    EEDR7   = 7     ; EEPROM Data Register bit 7

; EECR - EEPROM Control Register
.equ    EERE    = 0     ; EEPROM Read Enable
.equ    EEWE    = 1     ; EEPROM Write Enable
.equ    EEMWE   = 2     ; EEPROM Master Write Enable
.equ    EERIE   = 3     ; EEPROM Ready Interrupt Enable


; ***** PORTA ***********************
; PORTA - Port A Data Register
.equ    PORTA0  = 0     ; Port A Data Register bit 0
.equ    PA0     = 0     ; For compatibility
.equ    PORTA1  = 1     ; Port A Data Register bit 1
.equ    PA1     = 1     ; For compatibility
.equ    PORTA2  = 2     ; Port A Data Register bit 2
.equ    PA2     = 2     ; For compatibility
.equ    PORTA3  = 3     ; Port A Data Register bit 3
.equ    PA3     = 3     ; For compatibility
.equ    PORTA4  = 4     ; Port A Data Register bit 4
.equ    PA4     = 4     ; For compatibility
.equ    PORTA5  = 5     ; Port A Data Register bit 5
.equ    PA5     = 5     ; For compatibility
.equ    PORTA6  = 6     ; Port A Data Register bit 6
.equ    PA6     = 6     ; For compatibility
.equ    PORTA7  = 7     ; Port A Data Register bit 7
.equ    PA7     = 7     ; For compatibility
```

```
; DDRA - Port A Data Direction Register
.equ    DDA0    = 0      ; Data Direction Register, Port A, bit 0
.equ    DDA1    = 1      ; Data Direction Register, Port A, bit 1
.equ    DDA2    = 2      ; Data Direction Register, Port A, bit 2
.equ    DDA3    = 3      ; Data Direction Register, Port A, bit 3
.equ    DDA4    = 4      ; Data Direction Register, Port A, bit 4
.equ    DDA5    = 5      ; Data Direction Register, Port A, bit 5
.equ    DDA6    = 6      ; Data Direction Register, Port A, bit 6
.equ    DDA7    = 7      ; Data Direction Register, Port A, bit 7

; PINA - Port A Input Pins
.equ    PINA0   = 0      ; Input Pins, Port A bit 0
.equ    PINA1   = 1      ; Input Pins, Port A bit 1
.equ    PINA2   = 2      ; Input Pins, Port A bit 2
.equ    PINA3   = 3      ; Input Pins, Port A bit 3
.equ    PINA4   = 4      ; Input Pins, Port A bit 4
.equ    PINA5   = 5      ; Input Pins, Port A bit 5
.equ    PINA6   = 6      ; Input Pins, Port A bit 6
.equ    PINA7   = 7      ; Input Pins, Port A bit 7


; ***** PORTB ************************
; PORTB - Port B Data Register
.equ    PORTB0  = 0      ; Port B Data Register bit 0
.equ    PB0     = 0      ; For compatibility
.equ    PORTB1  = 1      ; Port B Data Register bit 1
.equ    PB1     = 1      ; For compatibility
.equ    PORTB2  = 2      ; Port B Data Register bit 2
.equ    PB2     = 2      ; For compatibility
.equ    PORTB3  = 3      ; Port B Data Register bit 3
.equ    PB3     = 3      ; For compatibility
.equ    PORTB4  = 4      ; Port B Data Register bit 4
.equ    PB4     = 4      ; For compatibility
.equ    PORTB5  = 5      ; Port B Data Register bit 5
.equ    PB5     = 5      ; For compatibility
.equ    PORTB6  = 6      ; Port B Data Register bit 6
.equ    PB6     = 6      ; For compatibility
.equ    PORTB7  = 7      ; Port B Data Register bit 7
.equ    PB7     = 7      ; For compatibility

; DDRB - Port B Data Direction Register
.equ    DDB0    = 0      ; Port B Data Direction Register bit 0
.equ    DDB1    = 1      ; Port B Data Direction Register bit 1
.equ    DDB2    = 2      ; Port B Data Direction Register bit 2
.equ    DDB3    = 3      ; Port B Data Direction Register bit 3
.equ    DDB4    = 4      ; Port B Data Direction Register bit 4
.equ    DDB5    = 5      ; Port B Data Direction Register bit 5
.equ    DDB6    = 6      ; Port B Data Direction Register bit 6
.equ    DDB7    = 7      ; Port B Data Direction Register bit 7

; PINB - Port B Input Pins
.equ    PINB0   = 0      ; Port B Input Pins bit 0
.equ    PINB1   = 1      ; Port B Input Pins bit 1
.equ    PINB2   = 2      ; Port B Input Pins bit 2
.equ    PINB3   = 3      ; Port B Input Pins bit 3
```

```
.equ    PINB4   = 4     ; Port B Input Pins bit 4
.equ    PINB5   = 5     ; Port B Input Pins bit 5
.equ    PINB6   = 6     ; Port B Input Pins bit 6
.equ    PINB7   = 7     ; Port B Input Pins bit 7


; ***** PORTC ************************
; PORTC - Port C Data Register
.equ    PORTC0  = 0     ; Port C Data Register bit 0
.equ    PC0     = 0     ; For compatibility
.equ    PORTC1  = 1     ; Port C Data Register bit 1
.equ    PC1     = 1     ; For compatibility
.equ    PORTC2  = 2     ; Port C Data Register bit 2
.equ    PC2     = 2     ; For compatibility
.equ    PORTC3  = 3     ; Port C Data Register bit 3
.equ    PC3     = 3     ; For compatibility
.equ    PORTC4  = 4     ; Port C Data Register bit 4
.equ    PC4     = 4     ; For compatibility
.equ    PORTC5  = 5     ; Port C Data Register bit 5
.equ    PC5     = 5     ; For compatibility
.equ    PORTC6  = 6     ; Port C Data Register bit 6
.equ    PC6     = 6     ; For compatibility
.equ    PORTC7  = 7     ; Port C Data Register bit 7
.equ    PC7     = 7     ; For compatibility

; DDRC - Port C Data Direction Register
.equ    DDC0    = 0     ; Port C Data Direction Register bit 0
.equ    DDC1    = 1     ; Port C Data Direction Register bit 1
.equ    DDC2    = 2     ; Port C Data Direction Register bit 2
.equ    DDC3    = 3     ; Port C Data Direction Register bit 3
.equ    DDC4    = 4     ; Port C Data Direction Register bit 4
.equ    DDC5    = 5     ; Port C Data Direction Register bit 5
.equ    DDC6    = 6     ; Port C Data Direction Register bit 6
.equ    DDC7    = 7     ; Port C Data Direction Register bit 7

; PINC - Port C Input Pins
.equ    PINC0   = 0     ; Port C Input Pins bit 0
.equ    PINC1   = 1     ; Port C Input Pins bit 1
.equ    PINC2   = 2     ; Port C Input Pins bit 2
.equ    PINC3   = 3     ; Port C Input Pins bit 3
.equ    PINC4   = 4     ; Port C Input Pins bit 4
.equ    PINC5   = 5     ; Port C Input Pins bit 5
.equ    PINC6   = 6     ; Port C Input Pins bit 6
.equ    PINC7   = 7     ; Port C Input Pins bit 7


; ***** PORTD ************************
; PORTD - Port D Data Register
.equ    PORTD0  = 0     ; Port D Data Register bit 0
.equ    PD0     = 0     ; For compatibility
.equ    PORTD1  = 1     ; Port D Data Register bit 1
.equ    PD1     = 1     ; For compatibility
.equ    PORTD2  = 2     ; Port D Data Register bit 2
.equ    PD2     = 2     ; For compatibility
.equ    PORTD3  = 3     ; Port D Data Register bit 3
.equ    PD3     = 3     ; For compatibility
```

```
.equ    PORTD4  = 4     ; Port D Data Register bit 4
.equ    PD4     = 4     ; For compatibility
.equ    PORTD5  = 5     ; Port D Data Register bit 5
.equ    PD5     = 5     ; For compatibility
.equ    PORTD6  = 6     ; Port D Data Register bit 6
.equ    PD6     = 6     ; For compatibility
.equ    PORTD7  = 7     ; Port D Data Register bit 7
.equ    PD7     = 7     ; For compatibility

; DDRD - Port D Data Direction Register
.equ    DDD0    = 0     ; Port D Data Direction Register bit 0
.equ    DDD1    = 1     ; Port D Data Direction Register bit 1
.equ    DDD2    = 2     ; Port D Data Direction Register bit 2
.equ    DDD3    = 3     ; Port D Data Direction Register bit 3
.equ    DDD4    = 4     ; Port D Data Direction Register bit 4
.equ    DDD5    = 5     ; Port D Data Direction Register bit 5
.equ    DDD6    = 6     ; Port D Data Direction Register bit 6
.equ    DDD7    = 7     ; Port D Data Direction Register bit 7

; PIND - Port D Input Pins
.equ    PIND0   = 0     ; Port D Input Pins bit 0
.equ    PIND1   = 1     ; Port D Input Pins bit 1
.equ    PIND2   = 2     ; Port D Input Pins bit 2
.equ    PIND3   = 3     ; Port D Input Pins bit 3
.equ    PIND4   = 4     ; Port D Input Pins bit 4
.equ    PIND5   = 5     ; Port D Input Pins bit 5
.equ    PIND6   = 6     ; Port D Input Pins bit 6
.equ    PIND7   = 7     ; Port D Input Pins bit 7


; ***** PORTE ************************
; PORTE - Data Register, Port E
.equ    PORTE0  = 0     ;
.equ    PE0     = 0     ; For compatibility
.equ    PORTE1  = 1     ;
.equ    PE1     = 1     ; For compatibility
.equ    PORTE2  = 2     ;
.equ    PE2     = 2     ; For compatibility
.equ    PORTE3  = 3     ;
.equ    PE3     = 3     ; For compatibility
.equ    PORTE4  = 4     ;
.equ    PE4     = 4     ; For compatibility
.equ    PORTE5  = 5     ;
.equ    PE5     = 5     ; For compatibility
.equ    PORTE6  = 6     ;
.equ    PE6     = 6     ; For compatibility
.equ    PORTE7  = 7     ;
.equ    PE7     = 7     ; For compatibility

; DDRE - Data Direction Register, Port E
.equ    DDE0    = 0     ;
.equ    DDE1    = 1     ;
.equ    DDE2    = 2     ;
.equ    DDE3    = 3     ;
.equ    DDE4    = 4     ;
.equ    DDE5    = 5     ;
```

```
.equ    DDE6    = 6       ;
.equ    DDE7    = 7       ;

; PINE - Input Pins, Port E
.equ    PINE0   = 0       ;
.equ    PINE1   = 1       ;
.equ    PINE2   = 2       ;
.equ    PINE3   = 3       ;
.equ    PINE4   = 4       ;
.equ    PINE5   = 5       ;
.equ    PINE6   = 6       ;
.equ    PINE7   = 7       ;


; ***** PORTF ************************
; PORTF - Data Register, Port F
.equ    PORTF0  = 0       ;
.equ    PF0     = 0       ; For compatibility
.equ    PORTF1  = 1       ;
.equ    PF1     = 1       ; For compatibility
.equ    PORTF2  = 2       ;
.equ    PF2     = 2       ; For compatibility
.equ    PORTF3  = 3       ;
.equ    PF3     = 3       ; For compatibility
.equ    PORTF4  = 4       ;
.equ    PF4     = 4       ; For compatibility
.equ    PORTF5  = 5       ;
.equ    PF5     = 5       ; For compatibility
.equ    PORTF6  = 6       ;
.equ    PF6     = 6       ; For compatibility
.equ    PORTF7  = 7       ;
.equ    PF7     = 7       ; For compatibility

; DDRF - Data Direction Register, Port F
.equ    DDF0    = 0       ;
.equ    DDF1    = 1       ;
.equ    DDF2    = 2       ;
.equ    DDF3    = 3       ;
.equ    DDF4    = 4       ;
.equ    DDF5    = 5       ;
.equ    DDF6    = 6       ;
.equ    DDF7    = 7       ;

; PINF - Input Pins, Port F
.equ    PINF0   = 0       ;
.equ    PINF1   = 1       ;
.equ    PINF2   = 2       ;
.equ    PINF3   = 3       ;
.equ    PINF4   = 4       ;
.equ    PINF5   = 5       ;
.equ    PINF6   = 6       ;
.equ    PINF7   = 7       ;


; ***** PORTG ************************
; PORTG - Data Register, Port G
```

```
.equ    PORTG0  = 0     ;
.equ    PG0     = 0     ; For compatibility
.equ    PORTG1  = 1     ;
.equ    PG1     = 1     ; For compatibility
.equ    PORTG2  = 2     ;
.equ    PG2     = 2     ; For compatibility
.equ    PORTG3  = 3     ;
.equ    PG3     = 3     ; For compatibility
.equ    PORTG4  = 4     ;
.equ    PG4     = 4     ; For compatibility

; DDRG - Data Direction Register, Port G
.equ    DDG0    = 0     ;
.equ    DDG1    = 1     ;
.equ    DDG2    = 2     ;
.equ    DDG3    = 3     ;
.equ    DDG4    = 4     ;

; PING - Input Pins, Port G
.equ    PING0   = 0     ;
.equ    PING1   = 1     ;
.equ    PING2   = 2     ;
.equ    PING3   = 3     ;
.equ    PING4   = 4     ;


; ***** TIMER_COUNTER_0 **************
; TCCR0 - Timer/Counter Control Register
.equ    CS00    = 0     ; Clock Select 0
.equ    CS01    = 1     ; Clock Select 1
.equ    CS02    = 2     ; Clock Select 2
.equ    WGM01   = 3     ; Waveform Generation Mode 1
.equ    CTC0    = WGM01 ; For compatibility
.equ    COM00   = 4     ; Compare match Output Mode 0
.equ    COM01   = 5     ; Compare Match Output Mode 1
.equ    WGM00   = 6     ; Waveform Generation Mode 0
.equ    PWM0    = WGM00 ; For compatibility
.equ    FOC0    = 7     ; Force Output Compare

; TCNT0 - Timer/Counter Register
.equ    TCNT0_0 = 0     ;
.equ    TCNT0_1 = 1     ;
.equ    TCNT0_2 = 2     ;
.equ    TCNT0_3 = 3     ;
.equ    TCNT0_4 = 4     ;
.equ    TCNT0_5 = 5     ;
.equ    TCNT0_6 = 6     ;
.equ    TCNT0_7 = 7     ;

; OCR0 - Output Compare Register
.equ    OCR0_0  = 0     ;
.equ    OCR0_1  = 1     ;
.equ    OCR0_2  = 2     ;
.equ    OCR0_3  = 3     ;
.equ    OCR0_4  = 4     ;
.equ    OCR0_5  = 5     ;
```

```
.equ    OCRO_6  = 6      ;
.equ    OCRO_7  = 7      ;

; ASSR - Asynchronus Status Register
.equ    TCROUB  = 0      ; Timer/Counter Control Register 0 Update Busy
.equ    OCROUB  = 1      ; Output Compare register 0 Busy
.equ    TCNOUB  = 2      ; Timer/Counter0 Update Busy
.equ    ASO     = 3      ; Asynchronus Timer/Counter 0

; TIMSK - Timer/Counter Interrupt Mask Register
.equ    TOIE0   = 0      ; Timer/Counter0 Overflow Interrupt Enable
.equ    OCIE0   = 1      ; Timer/Counter0 Output Compare Match Interrupt register

; TIFR - Timer/Counter Interrupt Flag register
.equ    TOV0    = 0      ; Timer/Counter0 Overflow Flag
.equ    OCF0    = 1      ; Output Compare Flag 0

; SFIOR - Special Function IO Register
;.equ   PSR0    = 1      ; Prescaler Reset Timer/Counter0
;.equ   TSM     = 7      ; Timer/Counter Synchronization Mode


; ***** TIMER_COUNTER_1 **************
; TIMSK - Timer/Counter Interrupt Mask Register
.equ    TOIE1   = 2      ; Timer/Counter1 Overflow Interrupt Enable
.equ    OCIE1B  = 3      ; Timer/Counter1 Output CompareB Match Interrupt Enable
.equ    OCIE1A  = 4      ; Timer/Counter1 Output CompareA Match Interrupt Enable
.equ    TICIE1  = 5      ; Timer/Counter1 Input Capture Interrupt Enable

; ETIMSK - Extended Timer/Counter Interrupt Mask Register
.equ    OCIE1C  = 0      ; Timer/Counter 1, Output Compare Match C Interrupt Enable

; TIFR - Timer/Counter Interrupt Flag register
.equ    TOV1    = 2      ; Timer/Counter1 Overflow Flag
.equ    OCF1B   = 3      ; Output Compare Flag 1B
.equ    OCF1A   = 4      ; Output Compare Flag 1A
.equ    ICF1    = 5      ; Input Capture Flag 1

; ETIFR - Extended Timer/Counter Interrupt Flag register
.equ    OCF1C   = 0      ; Timer/Counter 1, Output Compare C Match Flag

; SFIOR - Special Function IO Register
;.equ   PSR321  = 0      ; Prescaler Reset, T/C3, T/C2, T/C1
;.equ   TSM     = 7      ; Timer/Counter Synchronization Mode

; TCCR1A - Timer/Counter1 Control Register A
.equ    WGM10   = 0      ; Waveform Generation Mode Bit 0
.equ    PWM10   = WGM10  ; For compatibility
.equ    WGM11   = 1      ; Waveform Generation Mode Bit 1
.equ    PWM11   = WGM11  ; For compatibility
.equ    COM1C0  = 2      ; Compare Output Mode 1C, bit 0
.equ    COM1C1  = 3      ; Compare Output Mode 1C, bit 1
.equ    COM1B0  = 4      ; Compare Output Mode 1B, bit 0
.equ    COM1B1  = 5      ; Compare Output Mode 1B, bit 1
.equ    COM1A0  = 6      ; Compare Ouput Mode 1A, bit 0
.equ    COM1A1  = 7      ; Compare Output Mode 1A, bit 1
```

```
; TCCR1B - Timer/Counter1 Control Register B
.equ    CS10    = 0     ; Clock Select bit 0
.equ    CS11    = 1     ; Clock Select 1 bit 1
.equ    CS12    = 2     ; Clock Select1 bit 2
.equ    WGM12   = 3     ; Waveform Generation Mode
.equ    CTC10   = WGM12 ; For compatibility
.equ    WGM13   = 4     ; Waveform Generation Mode
.equ    CTC11   = WGM13 ; For compatibility
.equ    ICES1   = 6     ; Input Capture 1 Edge Select
.equ    ICNC1   = 7     ; Input Capture 1 Noise Canceler

; TCCR1C - Timer/Counter1 Control Register C
.equ    FOC1C   = 5     ; Force Output Compare for channel C
.equ    FOC1B   = 6     ; Force Output Compare for channel B
.equ    FOC1A   = 7     ; Force Output Compare for channel A


; ***** TIMER_COUNTER_2 **************
; TCCR2 - Timer/Counter Control Register
.equ    CS20    = 0     ; Clock Select
.equ    CS21    = 1     ; Clock Select
.equ    CS22    = 2     ; Clock Select
.equ    WGM21   = 3     ; Waveform Generation Mode
.equ    CTC2    = WGM21 ; For compatibility
.equ    COM20   = 4     ; Compare Match Output Mode
.equ    COM21   = 5     ; Compare Match Output Mode
.equ    WGM20   = 6     ; Wafeform Generation Mode
.equ    PWM2    = WGM20 ; For compatibility
.equ    FOC2    = 7     ; Force Output Compare

; TCNT2 - Timer/Counter Register
.equ    TCNT2_0 = 0     ; Timer/Counter Register Bit 0
.equ    TCNT2_1 = 1     ; Timer/Counter Register Bit 1
.equ    TCNT2_2 = 2     ; Timer/Counter Register Bit 2
.equ    TCNT2_3 = 3     ; Timer/Counter Register Bit 3
.equ    TCNT2_4 = 4     ; Timer/Counter Register Bit 4
.equ    TCNT2_5 = 5     ; Timer/Counter Register Bit 5
.equ    TCNT2_6 = 6     ; Timer/Counter Register Bit 6
.equ    TCNT2_7 = 7     ; Timer/Counter Register Bit 7

; OCR2 - Output Compare Register
.equ    OCR2_0  = 0     ; Output Compare Register Bit 0
.equ    OCR2_1  = 1     ; Output Compare Register Bit 1
.equ    OCR2_2  = 2     ; Output Compare Register Bit 2
.equ    OCR2_3  = 3     ; Output Compare Register Bit 3
.equ    OCR2_4  = 4     ; Output Compare Register Bit 4
.equ    OCR2_5  = 5     ; Output Compare Register Bit 5
.equ    OCR2_6  = 6     ; Output Compare Register Bit 6
.equ    OCR2_7  = 7     ; Output Compare Register Bit 7

; TIMSK -
.equ    TOIE2   = 6     ;
.equ    OCIE2   = 7     ;

; TIFR - Timer/Counter Interrupt Flag Register
```

```
.equ    TOV2    = 6      ; Timer/Counter2 Overflow Flag
.equ    OCF2    = 7      ; Output Compare Flag 2


; ***** TIMER_COUNTER_3 **************
; ETIMSK - Extended Timer/Counter Interrupt Mask Register
.equ    OCIE3C  = 1      ; Timer/Counter3, Output Compare Match Interrupt Enable
.equ    TOIE3   = 2      ; Timer/Counter3 Overflow Interrupt Enable
.equ    OCIE3B  = 3      ; Timer/Counter3 Output CompareB Match Interrupt Enable
.equ    OCIE3A  = 4      ; Timer/Counter3 Output CompareA Match Interrupt Enable
.equ    TICIE3  = 5      ; Timer/Counter3 Input Capture Interrupt Enable

; ETIFR - Extended Timer/Counter Interrupt Flag register
.equ    OCF3C   = 1      ; Timer/Counter3 Output Compare C Match Flag
.equ    TOV3    = 2      ; Timer/Counter3 Overflow Flag
.equ    OCF3B   = 3      ; Output Compare Flag 1B
.equ    OCF3A   = 4      ; Output Compare Flag 1A
.equ    ICF3    = 5      ; Input Capture Flag 1

; SFIOR - Special Function IO Register
;.equ   PSR321  = 0      ; Prescaler Reset, T/C3, T/C2, T/C1
;.equ   PSR1    = PSR321       ; For compatibility
;.equ   PSR2    = PSR321       ; For compatibility
;.equ   TSM     = 7      ; Timer/Counter Synchronization Mode

; TCCR3A - Timer/Counter3 Control Register A
.equ    WGM30   = 0      ; Waveform Generation Mode Bit 0
.equ    PWM30   = WGM30 ; For compatibility
.equ    WGM31   = 1      ; Waveform Generation Mode Bit 1
.equ    PWM31   = WGM31 ; For compatibility
.equ    COM3C0  = 2      ; Compare Output Mode 3C, bit 0
.equ    COM3C1  = 3      ; Compare Output Mode 3C, bit 1
.equ    COM3B0  = 4      ; Compare Output Mode 3B, bit 0
.equ    COM3B1  = 5      ; Compare Output Mode 3B, bit 1
.equ    COM3A0  = 6      ; Comparet Ouput Mode 3A, bit 0
.equ    COM3A1  = 7      ; Compare Output Mode 3A, bit 1

; TCCR3B - Timer/Counter3 Control Register B
.equ    CS30    = 0      ; Clock Select 3 bit 0
.equ    CS31    = 1      ; Clock Select 3 bit 1
.equ    CS32    = 2      ; Clock Select3 bit 2
.equ    WGM32   = 3      ; Waveform Generation Mode
.equ    CTC30   = WGM32 ; For compatibility
.equ    WGM33   = 4      ; Waveform Generation Mode
.equ    CTC31   = WGM33 ; For compatibility
.equ    ICES3   = 6      ; Input Capture 3 Edge Select
.equ    ICNC3   = 7      ; Input Capture 3  Noise Canceler

; TCCR3C - Timer/Counter3 Control Register C
.equ    FOC3C   = 5      ; Force Output Compare for channel C
.equ    FOC3B   = 6      ; Force Output Compare for channel B
.equ    FOC3A   = 7      ; Force Output Compare for channel A

; TCNT3L - Timer/Counter3 Low Byte
.equ    TCN3L0  = 0      ; Timer/Counter 3 bit 0
.equ    TCN3L1  = 1      ; Timer/Counter 3 bit 1
```

```
.equ    TCN3L2  = 2      ; Timer/Counter 3 bit 2
.equ    TCN3L3  = 3      ; Timer/Counter 3 bit 3
.equ    TCN3L4  = 4      ; Timer/Counter 3 bit 4
.equ    TCN3L5  = 5      ; Timer/Counter 3 bit 5
.equ    TCN3L6  = 6      ; Timer/Counter 3 bit 6
.equ    TCN3L7  = 7      ; Timer/Counter 3 bit 7


; ***** WATCHDOG *********************
; WDTCR - Watchdog Timer Control Register
.equ    WDTCSR  = WDTCR ; For compatibility
.equ    WDP0    = 0      ; Watch Dog Timer Prescaler bit 0
.equ    WDP1    = 1      ; Watch Dog Timer Prescaler bit 1
.equ    WDP2    = 2      ; Watch Dog Timer Prescaler bit 2
.equ    WDE     = 3      ; Watch Dog Enable
.equ    WDCE    = 4      ; Watchdog Change Enable
.equ    WDTOE   = WDCE  ; For compatibility


; ***** AD_CONVERTER *****************
; ADMUX - The ADC multiplexer Selection Register
.equ    MUX0    = 0      ; Analog Channel and Gain Selection Bits
.equ    MUX1    = 1      ; Analog Channel and Gain Selection Bits
.equ    MUX2    = 2      ; Analog Channel and Gain Selection Bits
.equ    MUX3    = 3      ; Analog Channel and Gain Selection Bits
.equ    MUX4    = 4      ; Analog Channel and Gain Selection Bits
.equ    ADLAR   = 5      ; Left Adjust Result
.equ    REFS0   = 6      ; Reference Selection Bit 0
.equ    REFS1   = 7      ; Reference Selection Bit 1

; ADCSRA - The ADC Control and Status register
.equ    ADCSR   = ADCSRA          ; For compatibility
.equ    ADPS0   = 0      ; ADC  Prescaler Select Bits
.equ    ADPS1   = 1      ; ADC  Prescaler Select Bits
.equ    ADPS2   = 2      ; ADC  Prescaler Select Bits
.equ    ADIE    = 3      ; ADC Interrupt Enable
.equ    ADIF    = 4      ; ADC Interrupt Flag
.equ    ADFR    = 5      ; ADC  Free Running Select
.equ    ADSC    = 6      ; ADC Start Conversion
.equ    ADEN    = 7      ; ADC Enable

; ADCH - ADC Data Register High Byte
.equ    ADCH0   = 0      ; ADC Data Register High Byte Bit 0
.equ    ADCH1   = 1      ; ADC Data Register High Byte Bit 1
.equ    ADCH2   = 2      ; ADC Data Register High Byte Bit 2
.equ    ADCH3   = 3      ; ADC Data Register High Byte Bit 3
.equ    ADCH4   = 4      ; ADC Data Register High Byte Bit 4
.equ    ADCH5   = 5      ; ADC Data Register High Byte Bit 5
.equ    ADCH6   = 6      ; ADC Data Register High Byte Bit 6
.equ    ADCH7   = 7      ; ADC Data Register High Byte Bit 7

; ADCL - ADC Data Register Low Byte
.equ    ADCL0   = 0      ; ADC Data Register Low Byte Bit 0
.equ    ADCL1   = 1      ; ADC Data Register Low Byte Bit 1
.equ    ADCL2   = 2      ; ADC Data Register Low Byte Bit 2
.equ    ADCL3   = 3      ; ADC Data Register Low Byte Bit 3
```

```
.equ    ADCL4   = 4     ; ADC Data Register Low Byte Bit 4
.equ    ADCL5   = 5     ; ADC Data Register Low Byte Bit 5
.equ    ADCL6   = 6     ; ADC Data Register Low Byte Bit 6
.equ    ADCL7   = 7     ; ADC Data Register Low Byte Bit 7


; ***** LOCKSBITS *******************************************
.equ    LB1     = 0     ; Lock bit
.equ    LB2     = 1     ; Lock bit
.equ    BLB01   = 2     ; Boot Lock bit
.equ    BLB02   = 3     ; Boot Lock bit
.equ    BLB11   = 4     ; Boot lock bit
.equ    BLB12   = 5     ; Boot lock bit


; ***** FUSES **********************************************
; LOW fuse bits
.equ    CKSEL0  = 0     ; Select Clock Source
.equ    CKSEL1  = 1     ; Select Clock Source
.equ    CKSEL2  = 2     ; Select Clock Source
.equ    CKSEL3  = 3     ; Select Clock Source
.equ    SUT0    = 4     ; Select start-up time
.equ    SUT1    = 5     ; Select start-up time
.equ    BODEN   = 6     ; Brown out detector enable
.equ    BODLEVEL        = 7     ; Brown out detector trigger level

; HIGH fuse bits
.equ    BOOTRST = 0     ; Select Reset Vector
.equ    BOOTSZ0 = 1     ; Select Boot Size
.equ    BOOTSZ1 = 2     ; Select Boot Size
.equ    EESAVE  = 3     ; EEPROM memory is preserved through chip erase
.equ    CKOPT   = 4     ; Oscillator Options
.equ    SPIEN   = 5     ; Enable Serial programming and Data Downloading
.equ    JTAGEN  = 6     ; Enable JTAG
.equ    OCDEN   = 7     ; Enable OCD

; EXTENDED fuse bits
.equ    WDTON   = 0     ; Watchdog timer always on
.equ    M103C   = 1     ; ATmega103 compatibility mode


; ***** CPU REGISTER DEFINITIONS ***************************************
.def    XH      = r27
.def    XL      = r26
.def    YH      = r29
.def    YL      = r28
.def    ZH      = r31
.def    ZL      = r30


; ***** DATA MEMORY DECLARATIONS ***************************************
.equ    FLASHEND        = 0xffff        ; Note: Word address
.equ    IOEND   = 0x00ff
```

```
.equ    SRAM_START     = 0x0100
.equ    SRAM_SIZE      = 4096
.equ    RAMEND  = 0x10ff
.equ    XRAMEND = 0xffff
.equ    E2END   = 0x0fff
.equ    EEPROMEND      = 0x0fff
.equ    EEADRBITS      = 12
#pragma AVRPART MEMORY PROG_FLASH 131072
#pragma AVRPART MEMORY EEPROM 4096
#pragma AVRPART MEMORY INT_SRAM SIZE 4096
#pragma AVRPART MEMORY INT_SRAM START_ADDR 0x100



; ***** BOOTLOADER DECLARATIONS ****************************************
.equ    NRWW_START_ADDR = 0xf000
.equ    NRWW_STOP_ADDR  = 0xffff
.equ    RWW_START_ADDR  = 0x0
.equ    RWW_STOP_ADDR   = 0xefff
.equ    PAGESIZE       = 128
.equ    FIRSTBOOTSTART = 0xfe00
.equ    SECONDBOOTSTART = 0xfc00
.equ    THIRDBOOTSTART  = 0xf800
.equ    FOURTHBOOTSTART = 0xf000
.equ    SMALLBOOTSTART  = FIRSTBOOTSTART
.equ    LARGEBOOTSTART  = FOURTHBOOTSTART



; ***** INTERRUPT VECTORS **********************************************
.equ    INT0addr       = 0x0002       ; External Interrupt Request 0
.equ    INT1addr       = 0x0004       ; External Interrupt Request 1
.equ    INT2addr       = 0x0006       ; External Interrupt Request 2
.equ    INT3addr       = 0x0008       ; External Interrupt Request 3
.equ    INT4addr       = 0x000a       ; External Interrupt Request 4
.equ    INT5addr       = 0x000c       ; External Interrupt Request 5
.equ    INT6addr       = 0x000e       ; External Interrupt Request 6
.equ    INT7addr       = 0x0010       ; External Interrupt Request 7
.equ    OC2addr = 0x0012       ; Timer/Counter2 Compare Match
.equ    OVF2addr       = 0x0014       ; Timer/Counter2 Overflow
.equ    ICP1addr       = 0x0016       ; Timer/Counter1 Capture Event
.equ    OC1Aaddr       = 0x0018       ; Timer/Counter1 Compare Match A
.equ    OC1Baddr       = 0x001a       ; Timer/Counter Compare Match B
.equ    OVF1addr       = 0x001c       ; Timer/Counter1 Overflow
.equ    OC0addr = 0x001e       ; Timer/Counter0 Compare Match
.equ    OVF0addr       = 0x0020       ; Timer/Counter0 Overflow
.equ    SPIaddr = 0x0022       ; SPI Serial Transfer Complete
.equ    URXC0addr       = 0x0024       ; USART0, Rx Complete
.equ    UDRE0addr       = 0x0026       ; USART0 Data Register Empty
.equ    UTXC0addr       = 0x0028       ; USART0, Tx Complete
.equ    ADCCaddr       = 0x002a       ; ADC Conversion Complete
.equ    ERDYaddr       = 0x002c       ; EEPROM Ready
.equ    ACIaddr = 0x002e       ; Analog Comparator
.equ    OC1Caddr       = 0x0030       ; Timer/Counter1 Compare Match C
.equ    ICP3addr       = 0x0032       ; Timer/Counter3 Capture Event
.equ    OC3Aaddr       = 0x0034       ; Timer/Counter3 Compare Match A
```

```
    .equ    OC3Baddr        = 0x0036        ; Timer/Counter3 Compare Match B
    .equ    OC3Caddr        = 0x0038        ; Timer/Counter3 Compare Match C
    .equ    OVF3addr        = 0x003a        ; Timer/Counter3 Overflow
    .equ    URXC1addr       = 0x003c        ; USART1, Rx Complete
    .equ    UDRE1addr       = 0x003e        ; USART1, Data Register Empty
    .equ    UTXC1addr       = 0x0040        ; USART1, Tx Complete
    .equ    TWIaddr = 0x0042        ; 2-wire Serial Interface
    .equ    SPMRaddr        = 0x0044        ; Store Program Memory Read

    .equ    INT_VECTORS_SIZE        = 70    ; size in words

#endif  /* _M128DEF_INC_ */

; ***** END OF FILE ****************************************************
```

Appendix E

AVR Opcode Encoding
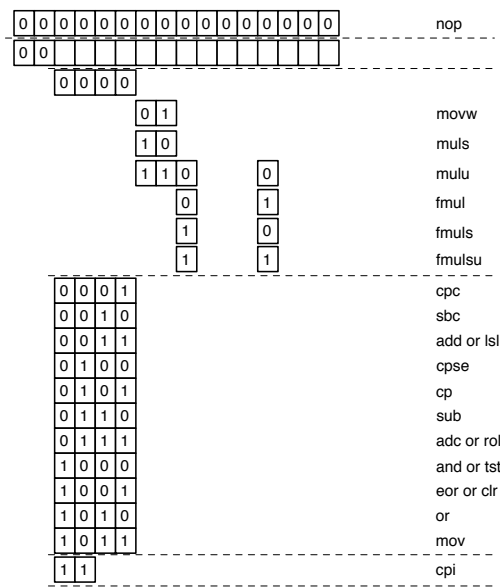
Figure E.1: Category 1 opcode encoding.
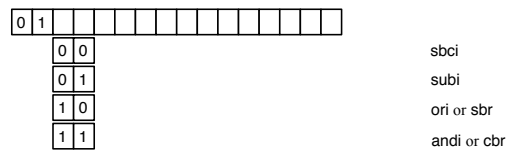
Figure E.2: Category 2 opcode encoding.

bits	instruction
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 E.3: Category 3 opcode encoding.

1	1															

0	0										rjmp
0	1										rcall
1	0										ldi
1	0	1	1	1	1		1	1	1	1	ser
1	1	0	0				0	0	0		brcs or brlo
1	1	0	0				0	0	1		breq
1	1	0	0				0	1	0		brmi
1	1	0	0				0	1	1		brvs
1	1	0	0				1	0	0		brlt
1	1	0	0				1	1	0		brts
1	1	0	0				1	1	1		brie
1	1	0	1				0	0	0		brcc or brsh
1	1	0	1				0	0	1		brne
1	1	0	1				0	1	0		brpl
1	1	0	1				0	1	1		brvc
1	1	0	1				1	0	0		brge
1	1	0	1				1	0	1		brhc
1	1	0	1				1	1	0		brtc
1	1	0	1				1	1	0		brid
1	1	1	0	0			0				bld
1	1	1	0	1			0				bst
1	1	1	1	0			0				sbrc
1	1	1	1	1			0				sbrs

Figure E.4: Category 4 opcode encoding.

Appendix F

Atmel Studio 6

Atmel Studio 6 is Atmel's official Integrated Development Environment (IDE) used for writing and debugging AVR applications on the Windows platform. Atmel Studio 6 is available for free, and can be downloaded at: http://www.atmel.com/tools/atmelstudio.aspx.

This section provides general information on how to successfully use Atmel Studio 6 to create, compile, and debug AVR assembly projects. Not every aspect of Atmel Studio 6 will be covered here, but for those who choose to learn the program in more detail, additional information can be obtained from Atmel's website at: http://www.atmel.com.

F.1 Startup Tutorial

This tutorial will give a step-by-step guide on how to install Atmel Studio 6, create a project, add code (new or existing) to the project, and simulate the project.

F.1.1 Installation

The installation of Atmel Studio 6 is straightforward and involves only a few steps:

1. Go to http://www.atmel.com/tools/atmelstudio.aspx, and click the download icon next to **Atmel Studio Installer**.
2. At this point, you can create a "myAtmel" account, or choose to download Atmel Studio 6 as a guest. In either case, follow the directions

and download the executable installer.

3. Locate the .exe file you just downloaded and run the setup program by double-clicking on it.

4. Follow the instructions in the setup program. Most of the default installation directories will work just fine.

5. When the installer is finished, click on the **Finish** button to complete the setup process. Atmel Studio 6 is now successfully installed.

F.1.2   Project Creation

Atmel Studio 6 is an Integrated Development Environment (IDE). Just like any other IDE, Atmel Studio 6 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. The following discusses the steps needed to create a new project:

1. Start Atmel Studio 6 by navigating through the Windows start menu: **Start ⇒ Programs ⇒ Atmel ⇒ Atmel Studio 6**. The path could be different if changed during installation.

2. Atmel Studio 6 should launch and display a Start Page. To create a new AVR project, click on the **New Project...** button, or navigate to **File ⇒ New ⇒ Project...**

3. The dialogue box that appears should look similar to Figure F.1. Under **Installed Templates**, make sure **Assembler** is selected.

4. Select **AVR Assembler Project** as the project type.

5. In the **Name** text box, type the name of the project, such as **Lab1**.

6. Make sure that that the checkbox for **Create directory for solution** is checked.

7. The location of the project can be changed by clicking on the **Browse...** button next to the path name, and navigating to the desired location for the new project.

8. Click **OK** to continue.

9. The next dialogue requires a device selection. First, ensure that the drop-down menu labeled **Device Family:** selects either **All** or **megaAVR, 8-bit**.

10. Scroll through the list of devices and select **ATmega128**.

11. Click **OK** to complete the project creation.

At this point, an editor window appears within Atmel Studio 6 and you
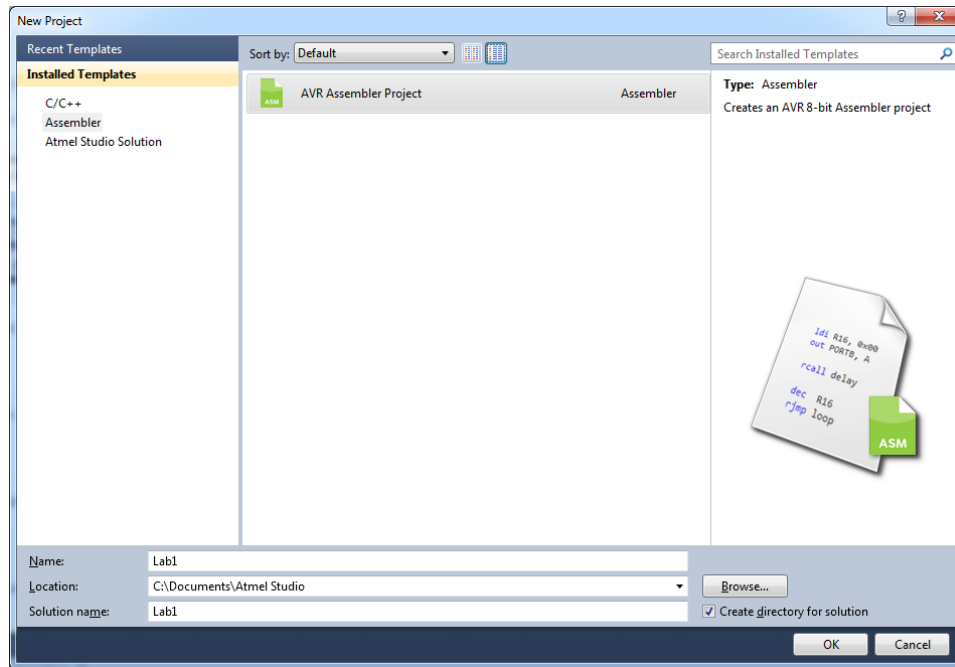
Figure F.1: AVR Studio Project Creation.

are able to begin composing your assembly program. Notice that Atmel Studio 6 has already created an empty assembly file for you, based on the name given earlier as the project name. For example, if you named your project **Lab1** as in Figure F.1 then the automatically-created assembly file would be named **Lab1.asm**.

If you want to incorporate some code that you have already written into this new project, then you can do so in one of two ways. First, you can simply open your existing code file with a text editor and copy-paste some or all of its contents directly into the open editor window within Atmel Studio 6 - this copies your code into the file created for you, e.g., **Lab1.asm**. If you want to include an entire existing file into your newly-created project, use the following steps:

1. In the **Solution Explorer** on the righthand side of the Atmel Studio 6 window, right-click on the name of your project (e.g., **Lab1**) and select **Add** ⇒ **Existing Item...**
2. Navigate to the existing assembly code file that you would like to use for this project, select it, and click **Add**.

3. Your existing code file will now appear in the **Solution Explorer** under the heading of your project. Double-click on the file name and it will open in a new editor tab.

4. If this existing file is to be the "main" assembly file of your project, right-click on the file name and select **Set As EntryFile**. Now this existing file that you included in the project will be considered the main entry point during compilation. Feel free to remove the automatically-created file (e.g. **Lab1.asm**) if you are not going to use it, by right-clicking on the file name and selecting **Remove**.

F.1.3 Project Simulation

Once a project has been created, and you have written an assembly program, it will need to be tested. This is accomplished by running the program on a simulated microcontroller built into Atmel Studio 6. Atmel Studio 6 has the capability to simulate almost every AVR microcontroller offered by Atmel. For the purposes of this tutorial, the ATmega128 will be the microcontroller that will be simulated. This microcontroller was selected earlier during the project creation phase. (To change the microcontroller, right-click on your project name in the **Solution Explorer** and select **Properties**. This will open a tab that allows you to configure various properties of your project. Make sure the **Build** tab is selected, and then click the **Change Device...** button and select a different microcontroller.)

1. Before the program can be simulated, it must first be compiled. There are three ways to do this:
   (a) In the main Atmel Studio 6 menu, navigate to Build $\Rightarrow$ Build Solution.
   (b) Click on the **Build Solution icon** on the main toolbar.
   (c) Press the **F7** key.

2. If the code was successfully compiled, a message in the **Output** window at the bottom should read "Build succeeded". If it does not say this, then there were some errors in the code. Clicking on the errors in the **Error List** will highlight the line of code causing the error 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, which allows a line-by-line simulation, and run mode, which continuously runs the program.
   (a) There are a few ways to run in debug mode:

       i. Follow the menu **Debug ⇒ Start Debugging and Break**.
      ii. Click on the **Start Debugging And Break icon**.
    iii. Press **Alt+F5**.
  (b) To start the run mode:
       i. Follow the menu **Debug ⇒ Continue**.
      ii. Click on the **Start Debugging icon**.
    iii. Press **F5**.

4. To stop the simulation at any point:
  (a) Follow the menu **Debug ⇒ Stop Debugging**.
  (b) Click on the **Stop Debugging Icon**.
  (c) Press **Ctrl+Shift+F5**.

5. That is how to simulate a program. For more detailed simulation tips and strategies, see Simulation Tips below.

F.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.

F.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 **Start Debugging and Break icon**. When the program is in run mode, hitting the **Break All icon** 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 subroutine calls such as the `RCALL` command.
- Step Over (**F10)** - Steps over subroutine calls. Normal operation will run program line-by-line, but will treat subroutine calls as a single instruction and not jump to the subroutine instructions.

- Step Out (**Shift+F11**) - Steps out of subroutine calls. This will temporarily put the simulation into run mode for the remainder of the subroutine and will pause at the next instruction after the subroutine 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.
- Reset (**Shift+F5**) - Simulates a reset of the microcontroller; returns the simulator to the first instruction of the program.

After experimenting around with these five commands, you should be able to navigate through the code with ease.

F.2.2   Workspace Window

When debugging, the **Solution Explorer** window is supplemented by tabs such as **IO View** and **Processor**, which provide a look at the current state of the microcontroller during the course of simulation. The **IO View** tab contains all the configuration registers associated with the simulated chip. By default, this window should automatically be displayed when simulation is run in line-by-line mode. Figure F.2 shows an example of what the **IO View** tab looks like during simulation. By expanding some of the contents of this window, additional information is available such as the current bit values, and address, of configuration registers. It is in this window where you can simulate input on the ports.

The **Processor** tab displays the current contents of the Program Counter, Stack Pointer, the 16-bit pointer registers X, Y, and Z, and the Status Register. Figure F.3 shows an example of what the **Processor** tab looks like during simulation. The **Processor** tab also shows the current values contained in each of the general purpose registers (in the case of the ATmega128, registers `R00 - R31`).

F.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 Atmel Studio 6 is no exception.
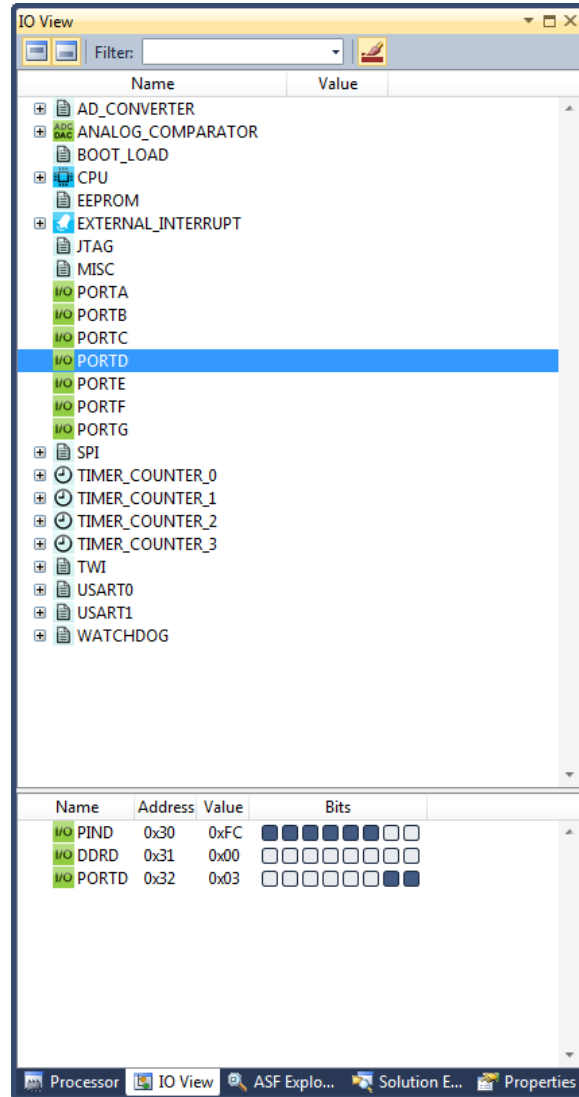
Figure F.2: I/O View tab in Workspace.

To view the **Memory** window, follow the menu command **Debug** ⇒ **Windows** ⇒ **Memory** ⇒ **Memory 1** or hit **Alt+6**. The **Memory** window, shown in Figure F.4, may pop up on top and obscure other windows, but it can be docked below the **Processor** and **IO View** tabs in order to be less intrusive.

The main area of the **Memory** window contains three sets of informa-
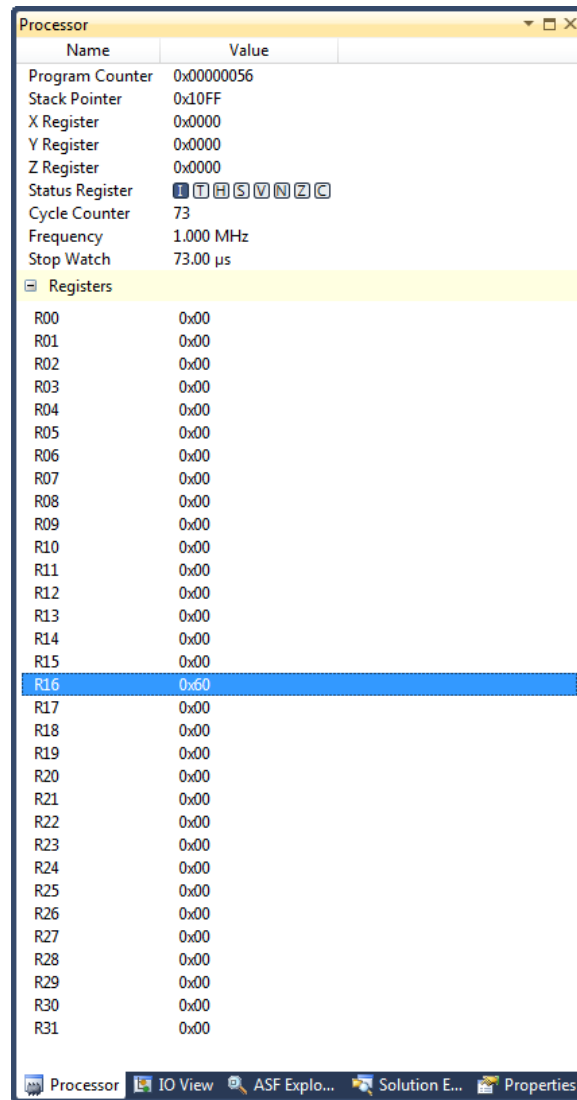
Figure F.3: Processor tab in Workspace.

tion; the starting address of each line of memory shown, 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. In Figure F.4, the contents of Program Memory are being displayed, with 0x000000 as the starting address of the first line shown. To edit the memory, just place the cursor in the
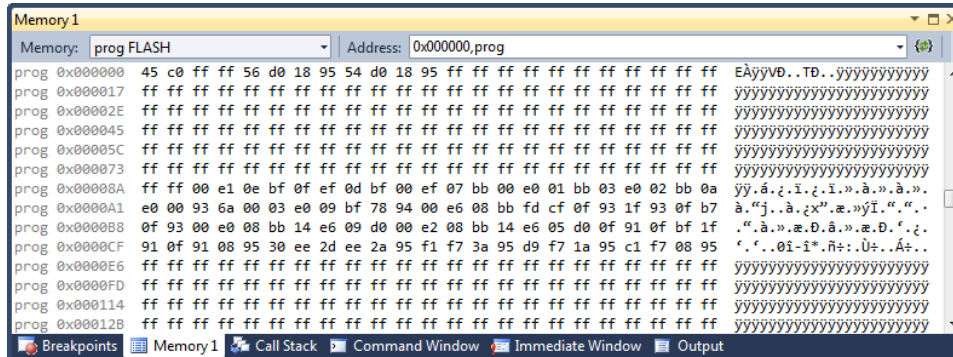
Figure F.4: Memory Window.

hexadecimal data area and type in the new data.

F.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 and 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 programming style and stick with it. The style is how you lay out your code, and having a consistent programming style 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 subroutines 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 View** tab, **Memory** tab, and **Processor** tab throughout the simulation. These windows will indicate any

    problem.

- Make sure the AVR instruction is actually supported by the AT-mega128.
- The ATmega128 has certain memory ranges; so make sure that when manipulating data, the addresses are within range.

Index