

Microcontrollers

ENGIN 2223

ALYSSA J. PASQUALE, PH.D.

College of DuPage
© ⓘ ⓘ Alyssa J. Pasquale, Ph.D.

Contents

1 Changelog	7
2 Introduction	9
2.1 Author Notes	9
2.2 License and Attribution Information	10
3 Course Prerequisites	11
4 Introduction to Microcontrollers and Embedded System Design	13
4.1 Computer Components	13
4.2 Microprocessor	15
4.3 Microcontroller	16
4.4 Embedded Systems	16
4.5 Choosing a Microcontroller	16
4.6 Atmel AVR Microcontroller and Arduino	17
4.7 Embedded System Programming	18
4.8 Compilers	20
4.9 Embedded System vs. Computer Program Design	20
4.10 Top-Down Design/Bottom-Up Implementation	22
4.11 Design Tools	23
4.12 Debugging	24
5 General Principles of Microcontrollers	27
5.1 CPU Architectures	27
5.2 Reduced Instruction Set Computing (RISC)	28
5.3 Machine Instructions	28
5.4 Instruction Decoder	31
5.5 Arithmetic and Logic Unit (ALU)	31
5.6 Registers	31
5.7 Program Counter (PC)	33
5.8 Memory	34
5.9 Instruction Execution Process and Timing	38
6 Status Register (SREG)	41
6.1 I – Global Interrupt Enable Flag	41

6.2 T – Bit Copy Storage Flag	41
6.3 H – Half Carry Flag	42
6.4 N – Negative Flag	42
6.5 V – z’s Complement Overflow Flag	42
6.6 S – Sign Flag	43
6.7 Z – Zero Flag	43
6.8 C – Carry Flag	43
6.9 Practice Problems	43
7 Memory Addressing Modes	45
7.1 Flash Program Memory	45
7.2 SRAM Data Memory	46
7.3 EEPROM Data Memory	46
7.4 Memory Addressing	47
7.5 Bit Addressing	55
7.6 The Stack and Stack Operations	55
7.7 Practice Problems	57
8 Model Microcontroller	59
8.1 Microcontroller Instructions	59
8.2 Microcontroller Operation Codes (Opcodes)	60
8.3 Model Microcontroller Program	62
8.4 Practice Problems	64
9 The ATmega328P Microcontroller	65
9.1 Pinout Diagrams	65
9.2 Pin Details	67
9.3 Writing Programs to Memory	68
9.4 Fuse Bytes	68
9.5 Practice Problems	69
10 I/O Port Registers	71
10.1 Data Direction	72
10.2 Electrical Characteristics	73
10.3 Internal Pull-Up Resistors	74
10.4 Alternate Pin Functions	74
10.5 Practice Problems	75
11 Analog to Digital Conversion	77
11.1 Sampling	78
11.2 Quantization	78
11.3 Encoding	80
11.4 ADC Architectures	80
11.5 The ADC on the ATmega328P	85
11.6 Analog Comparator on the ATmega328P	87

11.7 Digital to Analog Conversion	88
11.8 Practice Problems	88
12 Sensors and Sensor Calibration	91
12.1 Choosing Resistor Values	93
12.2 Sensor Calibration	95
12.3 Mitigating Fluctuating Data and Sensor Noise	97
12.4 ATmega328P Internal Temperature Sensor	99
13 Interrupts	101
13.1 Enabling and Disabling (Masking) Interrupts	102
13.2 Interrupt Service Routine (ISR)	102
13.3 Interrupts on the ATmega328P	104
13.4 Practice Problems	106
14 Clocks, Timer/Counters and Pulse-Width Modulation	107
14.1 ATmega328P Clock	108
14.2 ATmega328P Timer/Counters	110
14.3 ATmega328P Output Compare Unit	112
14.4 ATmega328P Timer/Counter Modes of Operation	113
14.5 Pulse-Width Modulation (PWM)	116
14.6 ATmega328P Timer/Counter PWM Modes of Operation	118
14.7 ATmega328P Input Capture Unit	126
14.8 ATmega328P Watchdog Timer (WDT)	129
14.9 Practice Problems	130
15 Serial Communication	133
15.1 Simplex and Duplex	134
15.2 Architecture	134
15.3 Data Transfer Protocol	134
15.4 ATmega328P Serial Peripheral Interface (SPI)	136
15.5 ATmega328P Universal Synchronous/Asynchronous Re- ceiver /Transmitter (USART)	140
15.6 ATmega328P Two-Wire Interface (TWI)	145
15.7 Practice Problems	147
16 Power Management and Sleep Modes	149
16.1 Sleep Modes	149
17 Control Systems and Feedback	153
17.1 Proportional Feedback Control	154
17.2 Proportional-Integral (PI) Control	155
17.3 Proportional-Integral-Derivative (PID) Control	156
18 C Concepts for Microcontrollers	159
18.1 Standard Datatypes	159

18.2 Variable Scope and Keywords	160
18.3 Arrays	162
18.4 Arithmetic and Assignment Operations	164
18.5 Bitwise Operations	166
18.6 Comparison and Boolean Operators	169
18.7 Compound Operators	170
18.8 Control Flow: Conditional	171
18.9 Control Flow: Iterative	175
19 Assembly	179
19.1 Data Transfer Instructions	179
19.2 Arithmetic and Logic Instructions	180
19.3 Branch (Control Flow) Instructions	182
19.4 Bit Manipulation Instructions	189
19.5 Miscellaneous Instructions	191
20 Index	193

1

Changelog

This section will be modified when changes are made to this textbook.

Table 1.1: Table of changes made to this textbook.

Date	Chapter(s)	Description of Change(s)
2021/02/15	all	Changed license to CC-BY-NC-SA
2021/07/12	all	Modified formatting and moved all figures to be in-line with the text
2021/07/12	4.10	Included clarifying details on top-down vs. bottom-up design
2021/07/12	4.11	Updated list of simulation software tools used in digital systems
2021/07/12	4.11	Added caveat about limitations to Tinkercad's Arduino simulation
2021/07/12	5.8	Added more explanation about what is meant by random-access memory and included information on its opposite (sequential-access memory)
2021/07/12	9.1	Included more information on each type of chip package
2021/07/12	15	Changed terminology used in serial communication architecture (see author note)
2021/07/14	note	Added attribution information
2022/03/23	14	Added more information on fast PWM and phase-correct PWM
2022/04/18	6	Added a reference for packed BCD addition in the half carry flag section
2022/04/20	10	Corrected a typo in output low current section and changed the wording in the current-level compatibility section to be more clear how current and logic levels are related.

2022/04/27	all	Made minor formatting changes
2024/03/23	4	Corrected ATmega328P datasheet information
2024/03/23	11	Edited and expanded chapter on analog to digital conversion
2024/03/24	14	Edited and expanded chapter on clocks, timer/counters, and pulse-width modulation
2024/03/25	2	Expanded the author note, and retitled “author notes”
2024/03/25	9	Added Pin Details section to the chapter
2024/03/25	12	Edited and expanded chapter on sensors
2024/03/25	13	Edited and expanded chapter on interrupts
2024/04/01	all	Added equation numbering
2024/04/01	14	Added information about glitches in fast PWM
2024/04/01	15	Edited and expanded chapter on serial communication
2024/04/08	7	Clarified that the stack pointer on the ATmega328P initializes to RAMEND
2024/04/08	11	Added information on the analog comparator on the ATmega328P
2024/04/08	all	Fixed index numbering so PDF bookmarks now point to the correct page
2024/04/08	–	Removed list of figures and list of tables
2024/04/15	14	Added equation numbering to equations in chapter 14
2024/05/22	16	Added additional information on sleep modes
2024/05/22	18	Added a section on arithmetic and assignment operations
2025/02/26	–	Fixed minor typos
2025/04/14	10	Expanded upon the description and labeling of the I/O pins and ports and added text about data direction, writing to output pins, and reading from input pins
2025/04/28	18	Rewrote most of the chapter to increase clarity
2025/04/28	19	Corrected indentation in assembly code blocks

2

Introduction

This book contains all of the information you will need to learn about the ATmega328P microcontroller as well as about embedded system design. First, microcontrollers and embedded systems will be described and explained at a zoomed-out level. Then, each of the peripheral features of the microcontroller will be explored in more detail. These topics include I/O port registers, analog to digital conversion, interrupts, timers/counters, clock systems, pulse-width modulation, serial communication, memory addressing, CPU registers and condition codes, and a basic overview of assembly language. Some chapters have additional practice problems to aid in studying the material.

This book should be used in conjunction with the ATmega328P datasheet¹ as well as with the course lab manual. A list of other useful and recommended textbooks is included in Chapter 3.

2.1 Author Notes

The original producer of the ATmega328P microcontroller, Atmel, was bought out by Microchip in 2016. Sometimes reference documents still contain the Atmel branding and authorship. When citing documents, I do my best to be consistent with the document itself. I first wrote this textbook prior to the purchase of Atmel by Microchip, so it's possible some references to Atmel are simply out of date. I apologize for any possible confusion this may cause.

The original versions of this textbook contained racist terminology for terms used in serial communication. I am sincerely sorry for using this terminology, and am making a concerted effort to become more educated. Because of the fact that Microchip still uses this nomenclature in their documentation, I will do my best to try to alleviate confusion due to the differences in terminology. There is a great article from Boston University that explains this situation.²

¹ Atmel, "ATmega328P Datasheet," January 2015

² Seele, Mike, "Striking Out Racist Terminology in Engineering," *The Brink*, July 2020.

2.2 License and Attribution Information

This book is licensed under creative commons as CC-BY-SA-NC. This license allows reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms. For more information, visit <https://creativecommons.org/>.

This license (CC-BY-SA-NC) includes the following elements:

- ① BY – Credit must be given to the creator
- ② NC – Only noncommercial uses of the work are permitted
- ③ SA – Adaptations must be shared under the same terms

The suggested attribution for this book is “**Microcontrollers**” by **Alyssa J. Pasquale, Ph.D., College of DuPage**, is licensed under **CC BY-NC-SA 4.0**.

The entirety of this work was created by Alyssa J. Pasquale, Ph.D. All circuit diagrams and figures in this text were created by the author using L^AT_EX libraries.

3

Course Prerequisites

The following topics should be well-understood before beginning this course.

- Number systems (especially decimal, binary and hexadecimal)
- Binary arithmetic, both signed and unsigned, including overflow detection
- ASCII code
- Boolean logic (particularly AND, OR, NOT, XOR, NAND, NOR)
- Multiplexers and decoders
- Combinational and sequential logic design
- Shift registers (especially PIPO, SIPO and PISO)
- Counters

Textbooks that may assist you in learning the material for this course are listed in Table 3.1.

Table 3.1: Recommended textbooks.

Title	Author
Introduction to Embedded Systems: Using ANSI C and the Arduino Development Environment	David Russell
The Atmel AVR Microcontroller: MEGA and XMEGA in Assembly and C	Han-Way Huang
AVR Microcontroller and Embedded Systems: Using Assembly and C	Muhammad Ali Mazidi, Janice Mazidi
Atmel AVR Microcontroller Primer: Programming and Interfacing	Steven F. Barrett, Daniel J. Pack

4

Introduction to Microcontrollers and Embedded System Design

TO UNDERSTAND MICROCONTROLLERS, one must first understand computers. What is a computer? These ubiquitous devices may be very familiar, but to understand their design more analytically, their components must be understood. A block diagram, shown in Figure 4.1, breaks down the components of a computer.

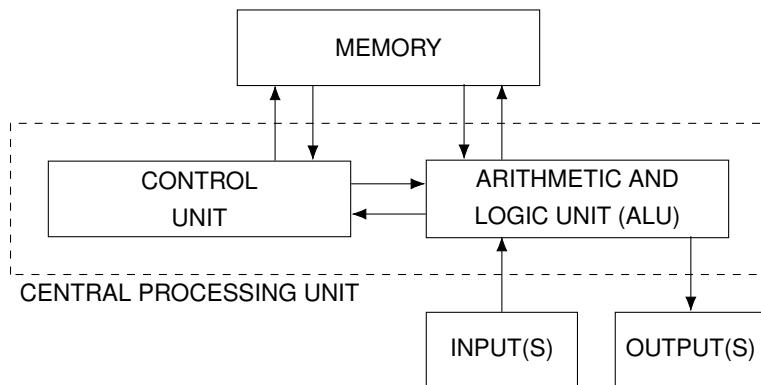


Figure 4.1: A block diagram of a computer, consisting of memory, a central processing unit, inputs and outputs.

4.1 Computer Components

The main computer components, as highlighted in this block diagram, include: memory, a control unit, an arithmetic and logic unit (ALU), inputs, outputs, and bus lines. Each of these components are explained briefly in this chapter, and then in more detail in chapter 5.

Computer Memory

Computer memory contains all of the information necessary to allow the computer to boot up, run programs, and access data. The instruc-

tions that tell the computer how to boot up is known as firmware. Program memory is referred to as software. As discussed later, this is much different from how microcontroller memory is organized. However, memory still plays a large role in storing program instructions and variable data information.

Control Unit

The control unit reads and interprets program instructions. It also sends control signals through the control bus. These signals instruct the computer to read or write to memory, control the timing of data transfer, and otherwise sequences the computer processor to take the necessary steps in the necessary timing to carry out the program.

The control unit uses something called a program counter (which is explored in more detail in [chapter 5](#)) to keep track of the current instruction, and a special piece of memory known as the status register (which is explained in [chapter 6](#)) to keep track of the most recently executed operation.

Arithmetic and Logic Unit (ALU)

As may seem obvious from the name, the arithmetic and logic unit (ALU) is capable of performing many arithmetic (addition, subtraction, multiplication) and logic (AND, XOR, OR) functions. The functions that are available in the ALU dictate the types of instructions that are possible to execute on the computer. The ALU together with the control unit form the central processing unit (CPU) of a computer.

Input and Output (I/O) Devices

Computers use a large number of input and output devices. Input devices gather information from the outside world to help the computer make decisions about what processes to carry out. Common computer inputs include: keyboard, mouse, touch screen, microphone, switches, and light detectors. Output devices allow the computer to display information for humans to see. Common computer outputs include: monitor, speakers, haptic feedback, LEDs, displays, and buzzers.

Bus Lines

Bus lines are interconnections between components. Bus lines are simply connections of multiple "normal" wires. Bus lines in this book are indicated by very thick lines. However, they may also be

drawn as lines with a slash. A number next to the slash indicates the number of bits in each bus. Both of these possibilities are shown schematically in Figure 4.2.

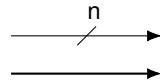


Figure 4.2: Two possible schematics of bus lines.

MANY "MICRO" DEVICES EXIST TODAY: microcomputers, microcontrollers, and microprocessors. Understanding this hierarchy of devices, outlined in Figure 4.3, will put microcontrollers in context, after which they will be focused on exclusively.

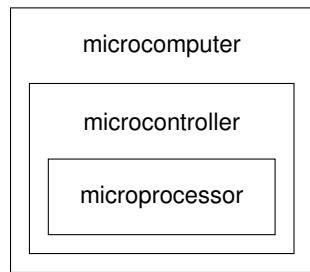


Figure 4.3: Hierarchy of "micro" devices: microcomputers consist of microcontrollers, which consist of microprocessors.

4.2 *Microp processor*

A microprocessor is a single integrated circuit (IC), generally consisting of just the CPU. If you have ever built a computer, this is the part that you purchase from AMD or Intel, and place into the motherboard. Thermal paste is put on top to thermally link it to a heat sink, which allows heat to safely dissipate away from the sensitive electronics inside of the processor. Less expensive, less powerful microprocessors are available for tasks that are not as extensive as those needed in general purpose computing.

The advantages of using a microprocessor (as opposed to a microcontroller) include

- small footprint,
- low cost, and
- easy to customize (the user can decide what and how many peripheral devices to include).

The disadvantages of using a microprocessor include

- lack of on-chip memory,

- no peripheral functions included, and
- cannot use directly with I/O devices.

4.3 Microcontroller

A microcontroller is a microprocessor that includes memory and I/O functionality. These are used in embedded systems, and are ubiquitous in today's technological world. Five of the major 8-bit microcontrollers are outlined in Table 4.1.

Vendor	Microcontroller(s)
Freescale Semiconductor	68HC08/68HC11
Intel	8051
Zilog	Z8
Microchip	AVR* PIC

Table 4.1: Five of the major 8-bit microcontrollers and their manufacturers.

Each of the microcontrollers listed in Table 4.1 has a specific instruction set, their own set of peripheral devices and I/O pins, and are generally not interchangeable. *Note: Atmel, the original producer of AVR microcontrollers, was acquired by Microchip in 2016. Many AVR manuals and datasheets still contain the Atmel logo.

4.4 Embedded Systems

Whereas a PC or a laptop is a general use machine (used for games, Internet, music, word processing, etc.) embedded systems refer to single-function devices. There are many examples of these in the world around us, but a good rule of thumb is that an embedded system is capable of performing computations without the use of an operating system (such as Windows, Linux, macOS, iOS, etc.). Embedded systems can be found in watches, MP3 players, vending machines, and more. In these examples, a full computer would be detrimental to the operation of the device. Imagine having to boot up Windows to run a dishwasher!

4.5 Choosing a Microcontroller

How does one choose a microcontroller to use in an embedded system project? It first must meet all project requirements and it must include peripherals and accessories that make it relatively simple to

develop products around the microcontroller. In addition, it is important to ensure that the microcontroller not only is available now, but will also be available into the future. Some considerations in making this determination are outlined in Table 4.2.

Project Requirements	
Speed	
Packaging (DIP, surface-mount, etc.)	
Power consumption	
Memory	
Peripherals (timers, ADC, etc.)	
Number of I/O pins	
Ease of upgrade	
Cost	
Additional Accessories	
An available assembler	
A debugger	
A compiler for high-level programming languages such as C	
Technical support	

Table 4.2: Project requirement and additional accessory considerations when choosing a microcontroller.

4.6 Atmel AVR Microcontroller and Arduino

The microcontroller that will be used in this class is the AVR ATmega328P. It is packaged in an Arduino Uno which contains extra features including (but not limited to): power regulator, bootloader, USB connection, I/O pins connected to headers, and the availability of the Arduino IDE for writing C code. The Arduino is a relatively inexpensive microcontroller package. By the end of this class, you will have all of the information you need to develop your own embedded systems and "smart projects" at home using this platform.

The ATmega328P datasheet¹ is the single most useful and important document to understanding everything that needs to be known about the microcontroller. It is available for download on the Microchip website. The first time you look at it, it may seem overwhelming. Perhaps you only understand a few words here and there. Do not be intimidated. Every time you refer to the datasheet, you will find that you understand a little more. Eventually, your knowledge will expand, and more and more of the datasheet will be part of your understanding. At the very least, become comfortable with the features of the ATmega328P, which are detailed on the front page of the datasheet. Some of this front page has been paraphrased in Table 4.3

¹ Atmel, "ATmega328P Datasheet," January 2015.

Table 4.3: ATmega328P datasheet front page.

High performance, low power AVR 8-bit microcontroller family
Advanced RISC architecture
131 powerful instructions
32 × 8 general purpose working registers
Up to 16 MIPS throughput at 16 MHz
High endurance non-volatile memory segments
32K bytes flash program memory
1K bytes EEPROM
2K bytes internal SRAM
Write/erase cycles: 10,000 flash / 100,000 EEPROM
Peripheral Features
Two 8-bit timer/counters
One 16-bit timer/counter
Six PWM channels
6-channel 10-bit ADC in PDIP package
Programmable serial USART
Primary/secondary SPI serial interface
Byte-oriented 2-wire serial interface (Philips I ² C compatible)
Programmable watchdog timer with separate on-chip oscillator
On-chip analog comparator
Interrupt and wake-up on pin change
Special microcontroller features
Power-on reset and programmable brown-out detection
Internal calibrated oscillator
External and internal interrupt sources
Six sleep modes
Operating voltage
2.7 V to 5.5 V for ATmega328P

4.7 Embedded System Programming

Computers and microcontrollers are useless without operating instructions, which are written in various programming languages. There is a hierarchy of language types, from machine language to high-level programming languages such as C. This hierarchy is depicted in Figure 4.4.

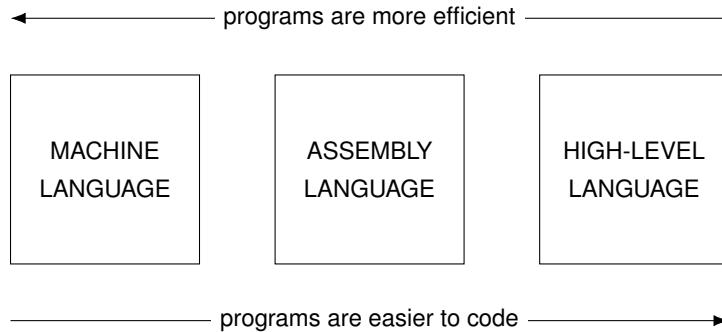


Figure 4.4: Hierarchy of language types and their relation to efficiency and simplicity.

Machine Language

Machine language is the instruction set of the microprocessor converted into binary. For example, the instruction 0001 1101 0100 1111 on the ATmega328P adds numbers from two registers (15 and 20) together. This code is then routed through hardware (the digital logic “guts” of the microprocessor) via the instruction decoder to route the appropriate control, address, and data signals to the associated hardware to carry out the desired task. Machine language can be very tedious to write and difficult to debug. It used to be accomplished using punch cards and magnetic tape.

Assembly Language

Assembly language uses mnemonic codes to refer to each instruction rather than the actual binary code. The above machine instruction could be rewritten into assembly as ADD r15, r20. Assembly is converted into machine code by a program called an assembler. Families of microcontrollers and microprocessors have specific instructions that they are capable of carrying out. For this reason, assembly code cannot be copy/pasted directly from an AVR microcontroller to an Intel processor, for example. The ATmega328P, being an AVR microcontroller, uses the AVR instruction set. Being familiar with the allowable instructions on a microcontroller means that code can be written very efficiently and compactly; assembly code generally uses less memory and takes less time to execute than code written using higher-level languages. AVR assembly is discussed in more detail in chapter 19.

High-Level Programming Language

High-level programming languages use functions to accomplish what assembly and machine language does. Addition can be carried out using familiar arithmetic symbols, for example: $f = 152 + 38$; High-

level language code is converted into assembly and from there to machine language by a compiler. Examples of high-level languages are C, C++, Python and Visual Basic.

Using high-level languages means that more time can be spent working on algorithms rather than on the specifics of what machine instructions to call in assembly. Initialization occurs without specifically having to do so; the stack, stack pointer, memory addresses, etc. are all allocated automatically by the compiler. In addition, high-level programs in the same language can more-or-less be recycled from one microcontroller to another (with caveats, not all registers have the same names, some functions may be a bit different, etc.). C concepts that are pertinent to microcontroller programming is discussed in more detail in [chapter 18](#).

4.8 Compilers

Compilers ensure that the high-level programming language code is correct, both in syntax and in memory allocation. Errors or warnings are usually displayed in these cases, and codes with errors are not loaded onto the microcontroller. Once the code is correct, the compiler takes the code and converts it into assembly language, and from there generates a HEX file which contains all of the machine code that needs to go into memory on the microcontroller.

4.9 Embedded System vs. Computer Program Design

Designing for embedded systems is very different from writing programs for computer (desktop, laptop, and mobile) applications. This has a lot to do with the resources, especially memory, available on each type of device. These differences are outlined in Table 4.4.

Table 4.4: Embedded vs. computer programming and resource differences.

	Embedded System	Computer
Program Data Location	ROM	RAM
RAM Capacity	2 kB (ATmega328P)	Nearly unlimited
ROM Capacity	32 kB (ATmega328P)	Nearly unlimited
Use of Peripherals (ADC, etc)	Frequent	Rare
Assembly Code Usage	Frequent	Rare

The use of peripheral features in embedded system design is so common that it forms the majority of the content in this textbook!

Desktop Memory

Desktop computers and laptops have a nearly unlimited supply of memory of all types.

ONBOARD FLASH MEMORY is used to store non-changing information containing instructions that tell the computer what to do when it boots up. This is usually known as BIOS (basic I/O system).

RANDOM-ACCESS MEMORY (RAM), stores program data while the program is running. For example, clicking on a PDF copies the entire Adobe program into RAM, and the program is run from that location in memory. This happens because there are many gigabytes of RAM available on most computers, and RAM tends to be faster than other computer memory types.

READ-ONLY MEMORY (ROM), which historically has mostly been made from magnetic disk or tape drives, and more recently solid-state memory, contains program executable storage and non-variable data files.

The term “read-only” is somewhat of a misnomer these days, as it is possible to write to most memory that we may call ROM. Flash, EEPROM, and SSD memory (all related technologies) are evolved from literal read-only memory, and the acronym continues to persist today.

Embedded Systems Memory

Embedded systems have much more limited memory capacity and options.

FLASH MEMORY (program memory) is non-volatile and stores program instructions and defined (constant) data.

RAM (data memory) is volatile memory that contains variable data. Because RAM is a limited resource, it is necessary to be aware of size constraints while writing applications. For example, an array with 500 floating-point values uses more RAM than the ATmega328P microcontroller can support.

Use of On-Chip Features

With embedded systems, a lot of peripheral features are used to interface with and control I/O devices, which doesn't happen on a PC. The peripheral features on the ATmega328P include

- an analog to digital converter (ADC),
- external interrupts
- timer/counter units,
- watchdog timer, and
- USART, SPI, and TWI serial communication protocols.

Configuring each of these peripheral functions requires a high level of understanding of each of their individual control registers.

Use of Assembly Code

Assembly code is frequently, but not always, used to program instructions onto a microcontroller. Assembly code uses less memory than equivalent C code, and their programs tend to be more efficient. Using assembly leads to great knowledge of the microcontroller.

4.10 Top-Down Design/Bottom-Up Implementation

For embedded control projects it is important to **design** before writing code or wiring up hardware. Rather than diving in head first and starting to wire things up and write code, it is important to first understand the problem. Break down the project into as many modules or parts as you can think of, and consider how they interconnect or interrelate. This is called top-down design (as if you are looking at the project from above and looking at each individual part). Once each module or part has been defined, then design and implement each one individually, rather than trying to tackle the entire project at once. This is called bottom-up implementation. This design process is explained in Table 4.5.

Table 4.5: Top-down design and bottom-up implementation process.

Top-Down Design

Understand the problem completely

- STOP – do not write any code yet!
- Specify requirements
- Think about possible errors
- Think about the "what", not the "how"
- Don't get caught up in minutiae

Design in levels

- Break the problem into parts
- Then break those parts into even more parts
- Build a block diagram of all parts and how they interrelate

- Start refining details
-

Bottom-Up Implementation

Implement one subsystem at a time

- Implement simpler subsystems first
 - Integrate everything at the end
-

At every step, it is important to keep detailed documentation of the project. Engineers keep notebooks in order to record ideas, thoughts, requirements, things that don't work, and things that do work.

The benefits of this design strategy is that it helps to clarify the problem. You can't design something if you're not completely sure what it should do. In addition, as parts get broken down into smaller pieces, they become less complicated. You may realize that parts of the solution may be reusable (e.g. a module built for one component may have code that can be used elsewhere in the design). Finally, when breaking a system down into parts, more than one person can work on the solution.

4.11 Design Tools

Design tools are always available to aid in project design and implementation.

One important design tool is the use of **flowcharts**. Flowcharts describe the steps that a program must implement, so that it is simpler to write the corresponding software code. Another benefit of flowcharts is that they are language-independent. In other words, given a flowchart, the program could then be implemented using the C programming language as easily as in Assembly. An example flowchart is shown in Figure 4.5.

Another important design tool is **circuit simulation software** which tests circuit functionality before wiring any hardware or writing any software code. Working with simulation software may be a large part of your future job. There are many different software packages that exist. You may have used [CircuitVerse](#), Logisim, or NI Multisim in Introduction to Digital Systems, which are hardware simulation packages. [Tinkercad](#) has circuit simulation software that allows testing of both the hardware and software of the Arduino Uno. This gives you the benefit of trying out new designs without necessarily having any access to an Arduino or any other components. (Note that not all Arduino functionality is possible on Tinkercad. Notably, many of the serial communication protocols have not been implemented in Tinkercad as of early 2020.)

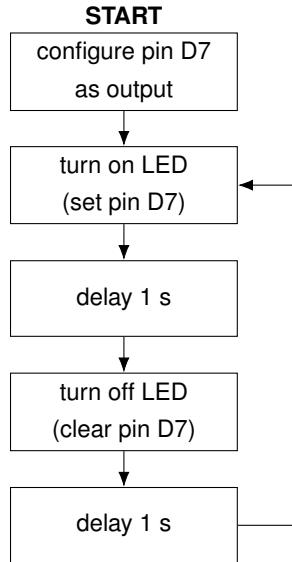


Figure 4.5: Example flowchart of a circuit that blinks an LED on and off with one second delays in between.

4.12 Debugging

Once a project has been designed implementation is begun, it will inevitably be time to start debugging. In order to make debugging as painless as possible, be smart about how code is implemented. Work on high-level functions first. Once they are working perfectly, add other subroutines only one at a time. If more than one bit of extra functionality is added at once, and something doesn't work, it will be difficult to determine what component of the design doesn't work. In the process, keep code well-commented, and document design decisions, research, implementation ideas, and notes about what does and doesn't work (and why) in a design notebook.

If something doesn't work (which will inevitably happen in any design), rather than diving in headfirst and changing code, find out what the code is actually doing. Then, and only then, can you change it to what it should be doing. This requires an intimate understanding of both the software and hardware components of your design. It may first need to be determined if hardware or software is at fault. For this reason it is imperative to test hardware before integrating it with software, to avoid problems later on. Several hardware testing and software debugging steps are suggested in Table 4.6.

Table 4.6: Hardware testing and software debugging procedures.

Hardware Testing

- Check that LEDs light before using them
- Check pushbuttons with a multimeter on continuity mode
- Use a tabletop multimeter or digital logic probe (for static signals)

Use an oscilloscope (for time-varying signals)

Software Debugging

Code walkthroughs: have your peers take a look at your code

Walk through the code step-by-step and see what results

Comment out lines of code one line at a time until the program works, then uncomment one line at a time until it doesn't work

5

General Principles of Microcontrollers

THE CENTRAL PROCESSING UNIT (CPU) of a microcontroller contains many parts that are crucial to carrying out all possible instructions, including the arithmetic and logic unit, registers, program counter, instruction decoder, and memory.

5.1 CPU Architectures

There are two main types of CPU architectures used in computing devices. **Harvard** architectures keep program and data memory separate. This allows the microcontroller to simultaneously read an instruction and write information to data memory. A simplified Harvard architecture CPU block diagram is shown in Figure 5.1. This simultaneous instruction and data access increases the speed of the device. Separate storage also means that program and data memories can be different widths. Data memory is restricted to 8 bits, but program memory is 16 bits on the ATmega328P.

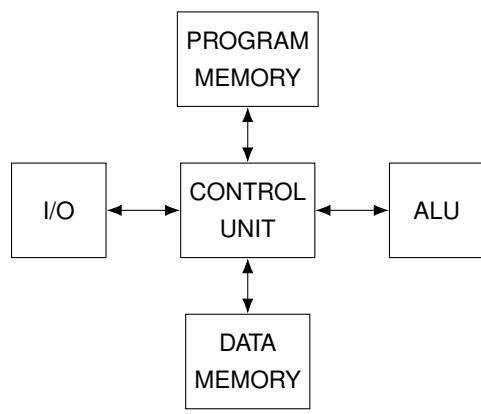


Figure 5.1: A Harvard CPU architecture features separate program and data memory.

In contrast, **von Neumann** architecture (named after mathematician and physicist John von Neumann) addresses program and data memory with a single bus. A simplified von Neumann architecture

CPU block diagram is shown in Figure 5.2. It is used extensively on PCs because RAM and ROM are external to the CPU and using separate wire-traces for each on the motherboard would be expensive.

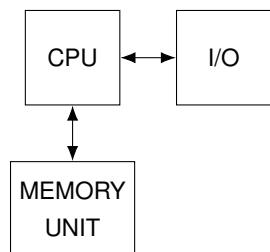


Figure 5.2: A von Neumann CPU architecture features a single bus for program and data memory.

ATmega328P Microcontroller

The ATmega328P microcontroller makes use of a Harvard architecture, meaning that program memory and data memory are stored in different locations. A block diagram of the ATmega328P CPU is shown in Figure 5.3. (Note that while virtually all of the interconnects are in actuality buses, bus notation is not used in this block diagram for simplicities sake.)

5.2 Reduced Instruction Set Computing (RISC)

Computers in the 1980s started to be programmed with every conceivable instruction, not all of which were used, which led to very complicated instruction codes and CPUs. Reduced instruction set computing (RISC) processors use only a limited number of instructions and have a fixed instruction size; most ATmega328P instructions are 16 bits (some are 32 bits).

Computers making use of RISC generally feature many registers, because data cannot be manipulated directly in memory (data must first be loaded into a register). RISC processors have a small instruction set, which is not a problem when using a high-level programming language such as C (but can make assembly coding more tedious). Most RISC instructions can be completed within a single clock cycle.

5.3 Machine Instructions

Instructions form the basis of all digital computing. A piece of binary data, called an instruction, contains information about the specific operation to be carried out. There are arithmetic operations such as addition and subtraction, branch operations such as jump to another

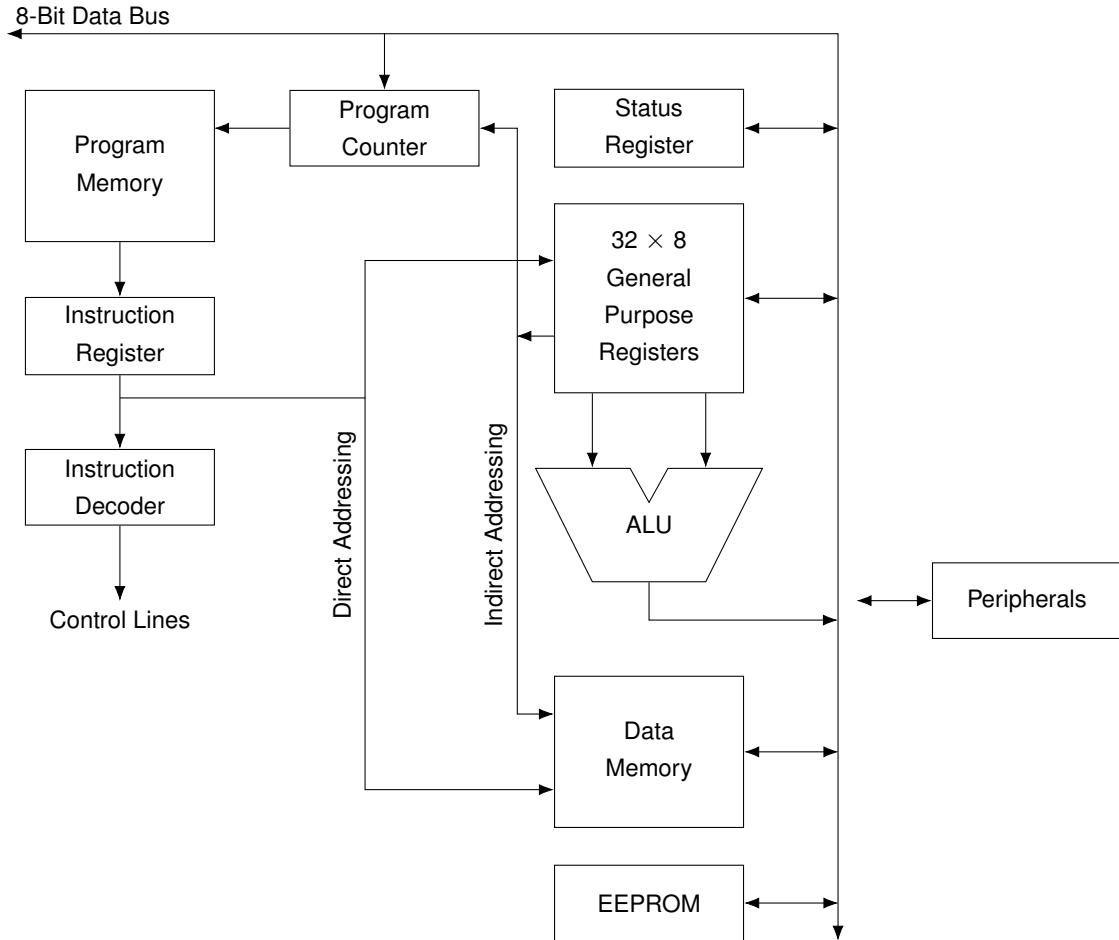


Figure 5.3: Block diagram of the ATmega328P central processing unit (CPU).

piece of code, data transfer operations such as loading or copying data from one place to another in memory, bit instructions such as clearing and shifting data, and control instructions such as sleeping or resetting the watchdog timer. In addition to containing information about the operation (known as the opcode), the instruction also contains information about what data is to be manipulated (known as the operand or operands). Each of these instructions is then parsed in hardware by the instruction decoder, which then routes the appropriate control and data signals to the arithmetic and logic unit (ALU).

In general, for every n opcodes, there must be a $\lceil \log_2(n) \rceil$ bit number to store it. The AVR instruction set for 8-bit microcontrollers, which is used for the ATmega328P microcontroller, has approximately 130 unique instructions.¹ $\lceil \log_2(130) \rceil = 8$, which means that 8 bits are needed in each 16-bit instruction to specify which operation will occur. This leaves only 8 bits for the remaining operand(s). There are 32 general purpose registers that can be used to temporarily store

¹ Atmel, "AVR Instruction Set Manual," November 2016.

data and many instructions exist that require the contents of two of these registers. In that case, 8 bits is insufficient to properly address both of these registers (each of which requires 5 bits to address).

To get around this limitation, the AVR instruction set uses the concept of variable length instructions, or **expanding opcodes**. Instructions that have many or lengthy operands are designed to have short opcodes while instructions with few or no operands are designed to have long opcodes. Most AVR instructions are 16 bits, with a few 32-bit instructions used as needed. There are 32 general purpose (GP) registers, and the ATmega328P additionally has 32,768 bytes of program memory and 2,048 bytes of data memory, which need to be addressed. In addition, there are other specialty registers such as a status register and stack register that can be affected by machine instructions.

To highlight the use of expanding opcodes, consider the following instructions which run the gamut from having a long opcode to having a short opcode.

- The instruction `CLI` is used to clear the global interrupt flag in the status and control register `SREG`. Because this instruction requires no operands, the full 16 bits of instruction are used to store the opcode.
- The instruction `NEG` converts the contents of a single GP register into a 2's complement number. Because there are 32 GP registers, 5 bits are required for the operand, leaving 11 bits remaining for the opcode.
- Most instructions have two GP registers as operands. For example, the instruction `ADD` will sum the contents of two GP registers. 10 bits are required to address both of the operands, leaving 6 bits for the opcode.
- A few instructions require very long operands. In order to properly address up to 4 MB of memory, it is necessary to use 22 bits of data for the operand. In order to accomplish this, a 32-bit instruction is used. 22 bits are dedicated to the memory address with the other 10 bits used for the opcode.

While expanding opcodes allows for much greater flexibility in instruction operands, it also makes for more difficult instruction decoding. This is because there is no longer a fixed position and length for each opcode within each instruction.

5.4 Instruction Decoder

The instruction decoder translates the instruction into appropriate control and address signals. This can take place using combinational logic gates or by using a programmed ROM. As an example, the ADD (add without carry) instruction requires two general purpose (GP) registers as operands (one of which is also designated as the destination register where the result will be saved). The instruction decoder will address these two GP registers and route their contents to adder hardware in the ALU, then route the solution back to the destination register.

5.5 Arithmetic and Logic Unit (ALU)

The ALU contains all of the hardware that is necessary to perform arithmetic and logic operations. While most ALU hardware designs are proprietary, taking a look at the instruction set gives a good idea of the type of hardware that must be included within the ALU. For example, the AVR ALU must contain, at the very least, an adder, logic gates such as AND, OR, XOR, and NOT, and other similar hardware.

5.6 Registers

Registers are ubiquitous in microcontroller design. **General purpose registers** contain data that we can immediately operate on, and allows us to store data with easy access to the ALU before saving results back to memory. **Memory address registers** or **pointer registers** tell the microcontroller where to look in program data for the next instruction to perform. **Status registers** (SREG on the ATmega328P) store information related to the most recent operation that has been executed on the microcontroller. **I/O registers** and extended I/O registers (sometimes called **peripheral registers**) store information regarding the operation of the I/O pins and their peripheral features. Some of these, DDRxn, PORTxn, and PINxn are discussed in [chapter 10](#). In addition, there is a **stack pointer register** that is used to hold memory addresses in short term storage.

Register Architectures

There are four main types of register architectures: serial in/serial out (SISO), parallel in/parallel out (PIPO), serial in/parallel out (SIPO) and parallel in/serial out (PISO). They are defined by their size, which has to do with the number of flip-flops that they consist

of, and ultimately tells us how many bits of data can be stored within them. Data can be loaded either serially or in parallel. A shift operation can be added to registers to increase their functionality, at the expense of added electronics.

Serial in/serial out (SISO) registers can be used in data buffering, storing data temporarily while it is in between its source and its destination. Serial data is sent to the input of the first flip-flop, and at each clock cycle the signal is shifted through subsequent flip-flops. Registers with serial input take the longest time for data to transmit from one end to the other. Registers with serial output additionally take longer than parallel output devices to access the data stream at the end. A schematic of a 4-bit SISO register is shown in Figure 5.4.

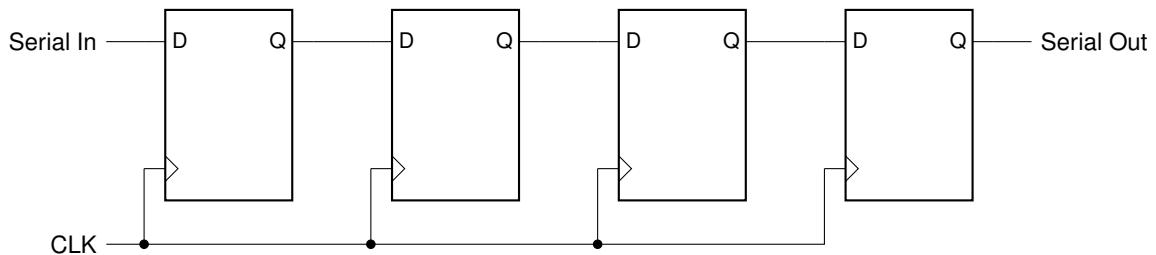


Figure 5.4: Schematic of a 4-bit serial in/serial out (SISO) register.

Parallel in/parallel out (PIPO) registers are used in general purpose I/O registers, among other applications. Data is immediately made accessible to all flip-flops simultaneously. In addition, all of the outputs are immediately available. At each clock cycle, the outputs are updated to reflect any new data available on the inputs. PIPO shift registers are available that have control signals to either parallel load the data or to shift data in either direction. A schematic of a 4-bit PIPO (non-shift) register is shown in Figure 5.5.

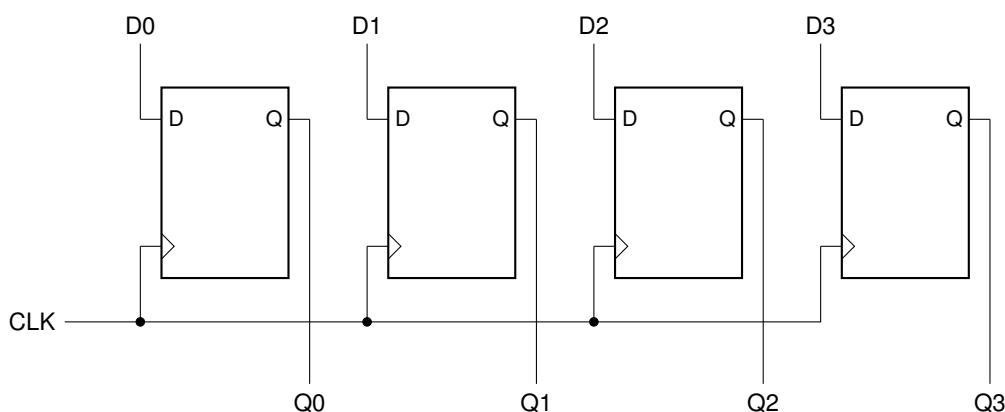


Figure 5.5: Schematic of a 4-bit parallel in/parallel out (PIPO) register.

Serial in/parallel out (SIPO) registers are used to input serial data from a serial communication protocol, and then output that data

either to a general purpose register or memory. Serial data is sent to the input of the first flip-flop, and at each clock cycle the signal is shifted through subsequent flip-flops. All outputs are immediately available. A schematic of a 4-bit SIPO register is shown in Figure 5.6.

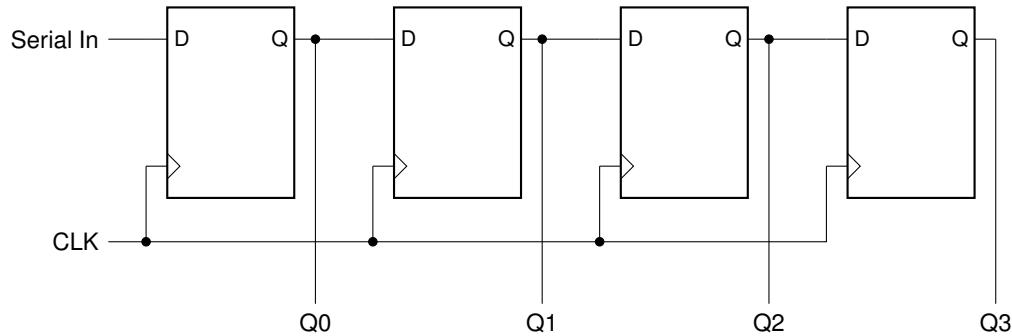


Figure 5.6: Schematic of a 4-bit serial in/parallel out (SIPO) register.

Parallel in / serial out (PISO) registers are used to input data from a general purpose register or memory and to output that data serially through a serial communication protocol. A schematic of a 2-bit PISO register is shown in Figure 5.7 (only two bits are shown to save space). A control signal $\overline{W/S}$ is used to control if the data loads into the flip-flops (which occurs when the signal is held LOW) or if data shifts through the flip-flops (which occurs when the signal is held HIGH).

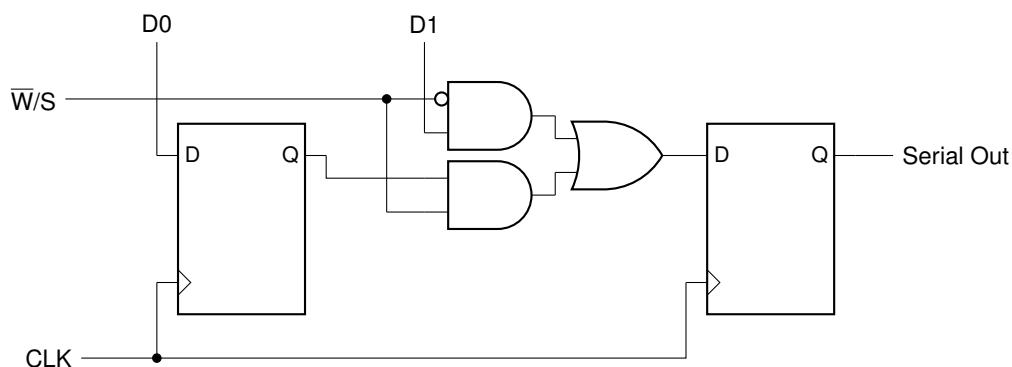


Figure 5.7: Schematic of a 2-bit parallel in/serial out (PISO) register.

5.7 Program Counter (PC)

The program counter contains the address (in memory) of the first byte of the instruction to be executed. It is incremented after every instruction. A schematic, shown in Figure 5.8, shows the inner workings of a program counter. When the power is switched on, the program counter is forced to 0 and the instruction fetch starts from address 0 ($\overline{\text{reset}} = 0$ forces the flip-flops to a value of 0).

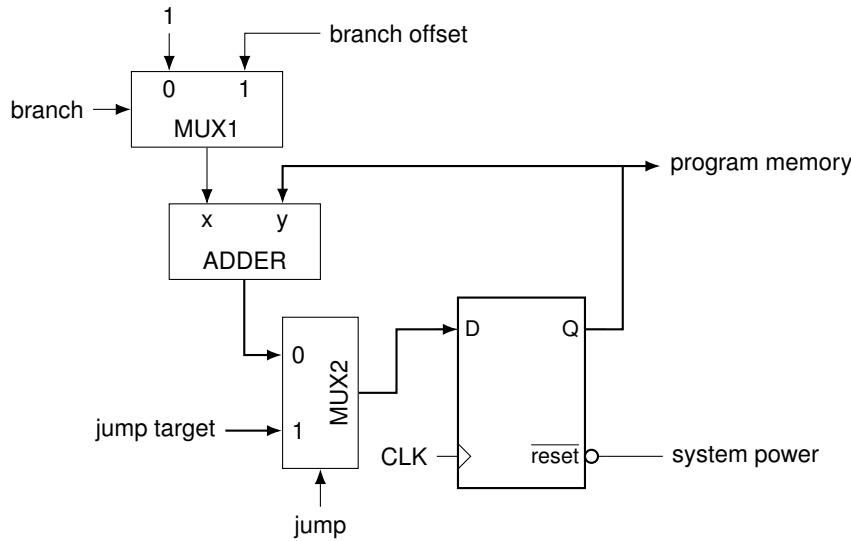


Figure 5.8: Schematic of a program counter (PC).

If the instruction is a conditional branch instruction (which is discussed in [chapter 6](#)), $\text{branch} = 1$, and the sum of the current PC value (Q_s) is added to the branch offset (which is passed through MUX1) and that value is sent to D_s to update the PC value.

If the instruction is a jump instruction, then the value of jump target (which is passed through MUX2) is loaded into D_s to update the PC value.

If the instruction is not a program flow control instruction (i.e. it is not a branch or jump instruction), then the PC is incremented by 1 after each instruction is fetched.

5.8 Memory

The two types of memory are volatile and non-volatile. This refers to whether or not data can persist after power has been removed from the microcontroller or computer. Before discussing these types of memory systems, however, it is important to note that the way that memory is addressed (i.e. how each piece of memory is stored or recalled) is also of critical importance to the operation of a microcontroller. This is discussed in [chapter 7](#).

Volatile Memory

Volatile memory is used for temporary storage because data is unable to persist after power has been removed from the device. It is usually called random-access memory (RAM). This concept of random-access refers to the ability to obtain data from any arbitrary (or random)

address in memory at any given time. Types of RAM include static RAM (SRAM), dynamic RAM (DRAM), and registers.

RAM differs from sequential-access memory, in which the data must be accessed in the same order in which it was stored. Sequential-access memory is the type of memory obtained when using rolls of magnetic tape (for example). A tape reader must spool through the entire reel to read from beginning to end in memory, and cannot skip around.

SRAM is the type of volatile memory used in the ATmega328P. It is characterized by

- use of flip-flops,
- faster operating speed than DRAM,
- higher cost than DRAM, and
- low power consumption.

A schematic of SRAM is given in Figure 5.9. Not shown in the schematic are connections to Vcc and GND for each inverter.

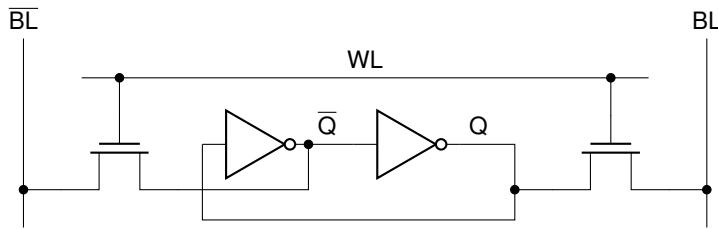


Figure 5.9: Schematic of a Static RAM cell.

The modes of operation of the SRAM depicted in Figure 5.9 are explained in Table 5.1.

Modes	Description
Standby	WL = LOW, mosfets are OFF (open switches) Q and \bar{Q} will not change and are floating
Write	WL = HIGH, mosfets are ON (closed switches) BL and \bar{BL} are asserted $Q = BL, \bar{Q} = \bar{BL}$
Read	WL = HIGH, mosfets are ON (closed switches) read data from BL and \bar{BL}

Table 5.1: Description of Static RAM modes of operation.

DRAM is the type of volatile memory used in computers due to its small size and low cost. DRAM is characterized by

- use of only one transistor and one capacitor,

- slower operating speed than SRAM,
- lower cost than SRAM,
- smaller footprint than SRAM,
- high power consumption, and
- requirement of constant refreshing (as otherwise the capacitors will slowly discharge over time).

A schematic of DRAM is given in Figure 5.10. If the capacitor is not continuously refreshed with either a value of one or zero, it will discharge and memory will be lost.

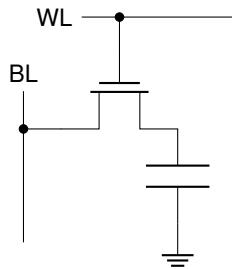


Figure 5.10: Schematic of a Dynamic RAM cell.

The modes of operation of the DRAM depicted in Figure 5.10 are explained in Table 5.2.

Modes	Description
Write	WL = HIGH, mosfets are ON (closed switches) BL is asserted, must wait for cap to charge or discharge
Read	WL = HIGH, mosfets are ON (closed switches) charge stored on cap goes to BL value must be re-written after every read

Table 5.2: Description of Dynamic RAM modes of operation.

A subset of DRAM known as pseudostatic RAM (PSRAM) uses memory cells similar to that of DRAM, but includes circuitry that allows it to refresh itself, which means that it has the same ease-of-use as SRAM.

Non-Volatile Memory

Non-volatile memory is used for more permanent storage because the data persists after power has been removed from the device. Over the years, non-volatile memory has taken on many different diverse forms: paper punch cards; wax cylinders and records; optical CD

and DVD disks; magnetic disks, drives and tape; and semiconductor drives. Because non-volatile memory doesn't require power to hold on to stored data, it is used to store firmware on computers, and program instructions and constants on microcontrollers. It is usually referred to as read-only memory (ROM), due to the fact that in regular operation the memory is effectively read-only. Writing to ROM typically requires a higher voltage than a read operation requires, and in addition write operations take more time than read operations.

The first types of ROM included mask ROM, which was manufactured in a clean room in a manner similar to the fabrication of integrated circuits. This meant that all of the bits of memory were hard-wired onto the device and could not be erased or rewritten after the fact. Mask ROM is therefore inefficient for small projects that may require debugging, or for embedded systems that require firmware updates or other types of memory changes.

EPROM is known as electrically programmable ROM. These chips generally contain a quartz window over the integrated circuit; exposure to UV light for an extended period of time erases the memory and sets all bits to a value of 1. An external programmer is used to electrically program all of the desired data to memory. EPROM has the advantage of being re-writable, but suffers from needing to be physically removed from a circuit to be erased and reprogrammed. In addition, the quartz window is expensive to make and EPROM chips can be more expensive than other types of memory. However, they were an improvement over one-time-programmable (PROM) chips, which, as the name implies, could only be programmed a single time and could not be erased and re-written.

Flash memory is used as program memory in the ATmega328P. It uses floating-gate transistors, one of which is shown in Figure 5.11 in "NOR" architecture. CG stands for control gate, and is where control signals are asserted. FG stands for floating gate, and is where electrons are either injected or removed to store data. S stands for source, which is generally the low potential (or ground) connection of a transistor. D is the drain, which is generally the high potential connection of a transistor.

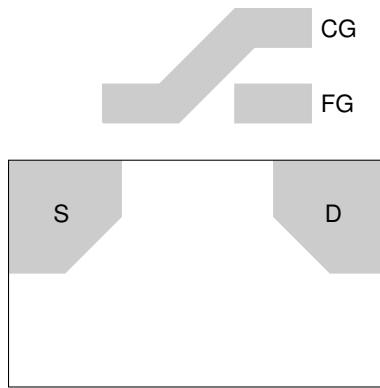


Figure 5.11: Schematic of a flash floating-gate cell.

The modes of operation of the flash floating-gate cell are detailed in Table 5.3.

Mode	Description
Clear	≈ 12 V applied to CG, channel on electrons travel from S to D in the process they tunnel through oxide to FG
Set	≈ -12 V applied to CG electrons tunnel from FG to CG and are removed
Read	5 V applied to CG if FG has electrons, a channel will not form (0) if FG has no electrons, a channel will form (1) from S to D

Table 5.3: Description of flash modes of operation.

Flash ROM is a subset of **EEPROM**, also known as electrically erasable programmable ROM. Today, there is little difference between EEPROM and flash architectures. However, the ATmega328P contains both flash and EEPROM storage for non-volatile memory. Flash is where program data and an optional bootloader are stored. The EEPROM on the ATmega328P is used for data storage that can be re-written but is not intended to change frequently. (For example, a passcode for a garage door opener can be reprogrammed, but mostly remains the same, and must be saved during power outages.) A fuse bit on the ATmega328P can be configured to allow the EEPROM storage to remain programmed in between chip erases, allowing additional functionality for the microcontroller.

5.9 Instruction Execution Process and Timing

This is the process by which the microcontroller executes each instruction. In this fashion, the CPU is nothing more than a very com-

plicated finite state machine. The state diagram is shown in Figure 5.12.

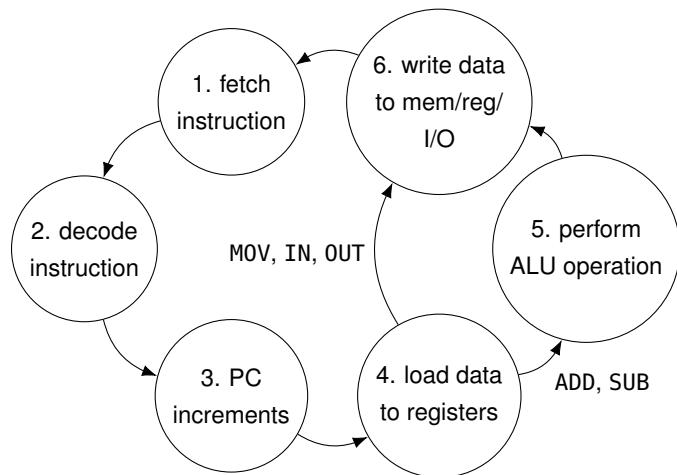


Figure 5.12: State diagram of microcontroller instruction execution process.

1. The next instruction in memory is read, indicated by the address stored in the program counter
2. The instruction is decoded into a set of commands or signals for each of the components in the processor
3. The program counter increments so that it points to the next location in memory
4. Data is loaded from memory (or input device(s)) into register(s), the location of this data is usually stored in the instruction code as an operand
5. If the ALU is required to execute the operation, the processor instructs the hardware to carry this out
6. The result is written back to a memory location, to a register, or even to an output device
7. Jump back to step 1

The ATmega328P uses a pipelining concept to speed up the rate at which the CPU can process the next instruction. This is enabled by its Harvard architecture, which allows the CPU to fetch the next instruction from memory while executing the current instruction. This is known as a parallel instruction fetch and is depicted in Figure 5.13.

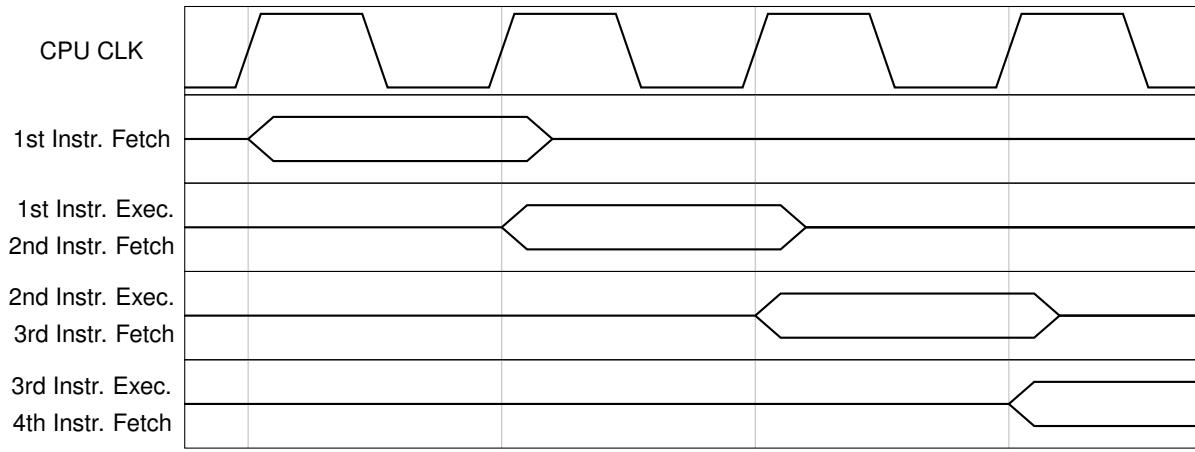


Figure 5.13: Timing diagram of parallel instruction fetch process.

Figure 5.14 shows the internal timing for the CPU, and shows how most of the instructions available on the ATmega328P are able to occur within a single clock cycle.

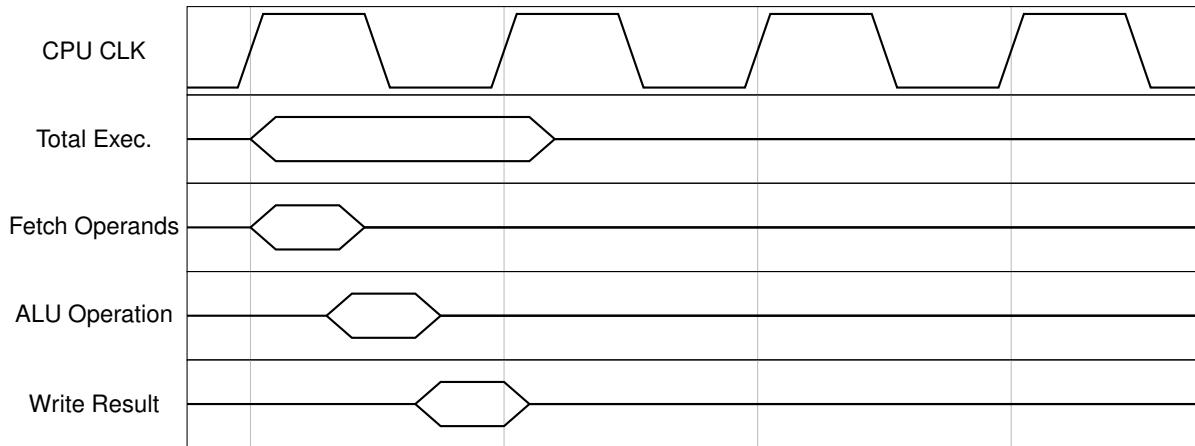


Figure 5.14: Timing diagram of single cycle ALU operation timing.

6

Status Register (SREG)

THE STATUS REGISTER ON THE ATMEGA328P contains information about the result of the most recently executed instruction on the microcontroller. The information in SREG can be used to control the flow of the program being executed by way of conditional branch instructions (which is discussed in [chapter 19](#)). All of the machine instructions that are capable of affecting the flags in SREG are listed in the AVR instruction manual ¹. The status register contains eight bits (known as flags) as shown in Table [6.1](#).

bit:	7	6	5	4	3	2	1	0
flag:	I	T	H	S	V	N	Z	C

¹ Atmel, "AVR Instruction Set Manual," November 2016.

Table 6.1: Contents of the status register (SREG).

6.1 I – Global Interrupt Enable Flag

This bit, as will be discussed in [chapter 13](#), the global interrupt enable flag is used to enable and disable interrupts. It must be set to 1 to globally enable interrupts, and it must be cleared to disable interrupts. This bit is automatically cleared by hardware on the ATmega328P when an interrupt service routine is invoked, and is subsequently set again when the execution of the ISR has been completed.

6.2 T – Bit Copy Storage Flag

There are instructions available in the AVR instruction set that use this bit as a source or destination for the operated bit. These two operations are BST (bit store from bit in register to T flag in SREG), and BLD (bit load from the T flag in SREG to a bit in register).

6.3 *H – Half Carry Flag*

The half carry flag is useful when dealing with binary coded decimal (BCD) numbers. The half carry flag is set if a carry was generated by the least significant 4 bits of the byte in the most recently executed instruction. In BCD arithmetic, this carry has a value of 16. The most significant bit (MSB) of 16 (the number 1) belongs in the 10's place of the BCD number. The remaining value of 6 needs to be added back in to the least significant nibble. An example of BCD addition with the half carry flag is shown in Table 6.2.

$ \begin{array}{r} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ + & 0 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 \\ + & 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 \end{array} $	$ \begin{array}{r} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \\ + & 0 & 1 & 1 & 0 \\ \hline 0 & 1 & 1 & 1 \end{array} $	$ \begin{array}{r} = 39_{10} \\ = 48_{10} \\ = 81_{10} \text{ WRONG!} \\ = 6_{10} \\ = 87_{10} \text{ Correct!} \end{array} $
--	--	---

An algorithm for “packed BCD addition” (when two BCD digits are represented in an 8-bit binary number) is provided in an application note by Microchip.²

Table 6.2: BCD arithmetic, adding 39 and 48 yields an incorrect sum unless 6 is added again into the least significant nibble of the result.

² Atmel, "AVR204: BCD Arithmetics," January 2003.

6.4 *N – Negative Flag*

This bit is only generated with signed numbers. The negative flag is the MSB of the result of the most recently executed operation. A value of 0 indicates a positive number, and a value of 1 indicates a negative number.

6.5 *V – 2's Complement Overflow Flag*

This bit is only generated with signed numbers. When set, the most recently executed operation resulted in an overflow condition. When cleared, the most recently executed operation did not result in an overflow. There are different equations for its generation based on the particular instruction that is in use.

In **signed addition**, an overflow occurs if the addition of two negative numbers results in a positive number, or when the addition of two positive numbers results in a negative number. Overflow is not possible when adding a positive to a negative number.

In **signed subtraction**, an overflow occurs if a positive number minus a negative number results in a negative number, or when a negative number minus a positive number results in a positive number. Overflow is not possible when subtracting a positive number

from a positive number, or when subtracting a negative number from a negative number.

In **signed multiplication**, an overflow occurs if not all of the overflow bits are equal.

6.6 S – Sign Flag

The sign flag is always an exclusive OR operation between the N and V flags,

$$S = N \oplus V.$$

It gives the true sign of the result of the most recent operation, as indicated in Table 6.3.

N	V	S	
0	0	0	sign is positive
0	1	1	sign is negative (overflow with false positive answer)
1	0	1	sign is negative
1	1	0	sign is positive (overflow with false negative answer)

Table 6.3: Information about the sign flag (S) flag based on the negative flag (N) and the z's complement overflow flag (V).

6.7 Z – Zero Flag

This bit is set if the result of the most recently executed operation is zero.

6.8 C – Carry Flag

This bit is set if the most recently executed operation resulted in a carry or a borrow. This flag is particularly useful when adding numbers that require more than 8-bits to store. For example, when adding 16-bit numbers, the microcontroller must work 8-bits at a time due to the size limitation of the CPU registers. A carry on the least significant byte (carry flag of 1) indicates that one must be added to the most significant byte.

In unsigned addition and subtraction, the presence of a carry or borrow indicates overflow.

6.9 Practice Problems

- Determine the values of the C, Z, N, V and S flags for the following.

(a) $1010\ 1101 + 0101\ 0011$

C=1, Z=1, N=0, V=0, S=0

(b) 0000 1010 — 0000 1111

 $C=1, Z=0, N=1, V=0, S=1$

(c) 0101 0101 + 0101 0010

 $C=0, Z=0, N=1, V=1, S=0$

7

Memory Addressing Modes

MEMORY MAPS OF MICROCONTROLLERS indicate what addresses are used for what type of memory. They are important to understand the functionality of the microcontroller memory space, and how the memory can be accessed by hardware.

7.1 Flash Program Memory

The ATmega328P contains 32 kBytes of non-volatile program flash memory. Because all of the instructions are 16-bits or 32-bits wide, data memory is formatted as 16 kBytes of 16-bit words. This means that 14 bits ($\log_2 16384$) are required to address the full program memory space on the ATmega328P. Indeed, the program counter on the ATmega328P is a 14-bit counter.

Program memory space is divided into two sections: an application section and a bootloader section. The size of the bootloader section is configured with the high fuse byte, as discussed in [chapter 9](#). A map of the program memory on the ATmega328P is shown in Figure 7.1.

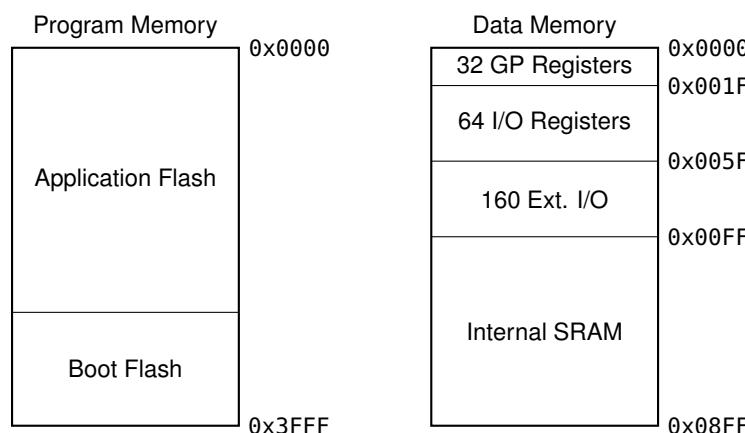


Figure 7.1: Memory map of the flash program memory and SRAM data memory on the ATmega328P.

7.2 SRAM Data Memory

The volatile data memory on the ATmega328P is composed of 2 kBytes of SRAM. This data memory space, the memory map of which is given in Figure 7.1, is split into several sections.

32 General Purpose Registers

These 32 general purpose registers contain information to be entered into the ALU on the microcontroller. They are used to execute instructions. Registers R26 to R31 (known as X, Y, and Z) are indirect addressing pointer registers. They are used in pairs to indirectly address program memory.

64 I/O Registers

In addition to the general purpose registers, data memory contains 64 I/O registers. This memory space contains the PIN_x, PORT_x, and DDR_x registers, as well as SREG and registers that control the operation of timer/counter 0, SPI communication, external interrupts, among others. These registers can be directly addressed using assembly instructions. Using assembly commands such as LOAD, data from the 64 I/O registers can be loaded into the general purpose registers to allow further operations to take place.

160 Extended I/O Registers

There are a lot more peripheral units than can be supported with 64 memory locations, therefore there are an additional 160 extended I/O registers that control all of the other peripheral features of the ATmega328P, including serial communication, interrupts, timer/counters, ADC, and more. These memory locations can only be accessed indirectly using load and store instructions in assembly (which takes more clock cycles than the direct addressing that can be used on the 64 I/O registers).

Internal SRAM

The remaining data space is used to store variable data in memory, as well as to house the stack.

7.3 EEPROM Data Memory

In addition to the above mentioned memory spaces, 1 kByte of EEPROM non-volatile data memory is available as a separate data space

from SRAM. EEPROM is used to save data that can be changed but should remain stored in memory even in the absence of power to the device.

7.4 Memory Addressing

Memory addressing refers to the ways in which a microcontroller accesses data to carry out an instruction. The instruction contains operands that specify the data to be operated on, while the program counter provides the address for the next instruction in program memory. Memory addressing modes are determined when a microcontroller is designed, and cannot be changed by a programmer.

In a simple device such as an EPROM chip, memory addressing occurs directly by means of an internal decoder. A memory address is asserted on the address pins, and the internal circuitry asserts the contents of that address onto the output pins of the device. This is depicted for a 65 kB memory in Figure 7.2.

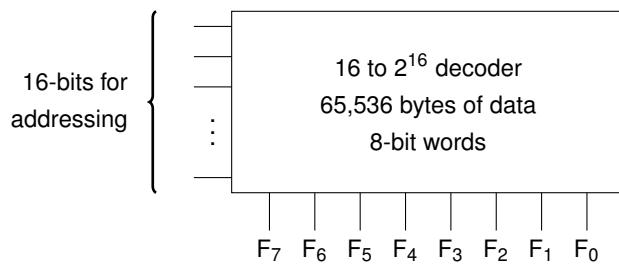


Figure 7.2: Memory addressing occurs in a simple system with a decoder. A 16×2^{16} decoder is capable of addressing 65,536 words of data.

In general, memory can be addressed **directly** or **indirectly**. Direct addressing occurs when the memory address is stored as part of the instruction itself. Direct addressing is typically fast (the data to be operated on in memory is pointed to directly), and has the flexibility of allowing variable data to be manipulated, rather than constant data (which is used in immediate addressing). Indirect addressing occurs when a pointer is used instead of the memory address. Indirect addressing enables a microcontroller to access memory contents even if the number of addresses available exceeds the size of the instruction itself, but suffers the drawback of being slower than direct addressing (first the pointer must be loaded, then the contents of the memory address that the pointer points to is operated on).

General Purpose Register Addressing

General purpose register addressing occurs when an assembly instruction contains the address of one or more general purpose register to be operated upon. Instructions with only a single register (Rd)

operand address memory in a way known as **direct, single register addressing**, which is shown schematically in Figure 7.3.

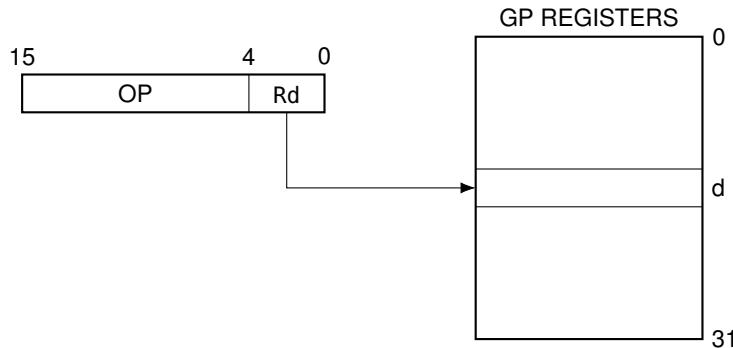


Figure 7.3: Direct, single register addressing occurs when an instruction contains a single general purpose register as an operand.

An instruction that directly addresses a single GP register is NEG, which takes the two's complement value of a register and saves the result back into that same register. It is capable of addressing all of the 32 GP registers using 5 bits in the instruction. The remaining 11 bits are used for the opcode.

Instructions with two general purpose register operands (Rd and Rr) address memory in a way known as **direct, two register addressing**, which is shown schematically in Figure 7.4. The result of these instructions is always stored in Rd, the destination register.

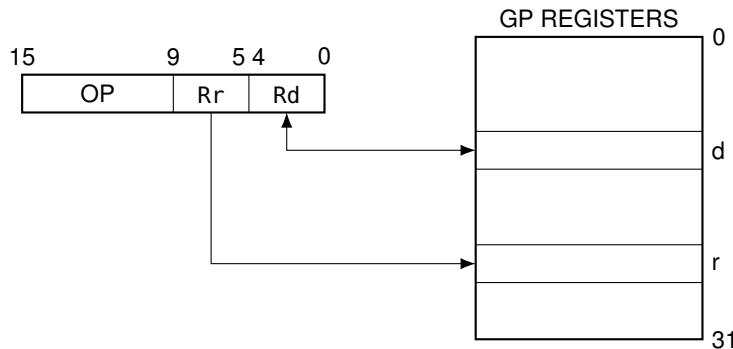


Figure 7.4: Direct, two register addressing occurs when an instruction contains two general purpose registers as operands.

An instruction that directly addresses a single GP register is ADD, which takes the value stored in a source register, adds it to the value stored in a destination register, and saves the result back into the destination register. It is capable of addressing all of the 32 GP registers, which means that 10 bits of the instruction are used for register addressing. The remaining 6 bits are used for the opcode.

Immediate Addressing

In immediate addressing, the instruction contains data to be operated on immediately. In other words, the data to be operated on is part of the instruction. Because the data is part of the instruction, immediate addressing can be fast. However, it is also less flexible when it comes to changing or repurposing code.

Most of the instructions that contain immediate addressing in the ATmega328P allow for the immediate data to take on values between 0–255. This requires 8 bits of data, which means that only 8 bits remain in the instruction for the opcode and the other operand. The number of GP registers that can be addressed is therefore limited to registers 16–31, which means that 4 bits are used to address the register and the remaining 4 bits are used as the opcode.

The AND and ANDI instructions demonstrate the difference between two-register addressing and immediate addressing. AND is a logical AND operation that occurs between two GP registers, Rd and Rr. The data is fetched from each of the registers, routed to the ALU, and then stored in the destination register. All of the GP registers can be addressed in this instruction, leaving 6 bits for the opcode. The machine instruction is:

```
0010 00rd dddd rrrr,
```

where 001000 is the opcode, rrrrr indicates the binary value of the source register, and dddd indicates the binary value of the destination register.

ANDI is a logical AND with an immediate. This means that the microcontroller performs a logical AND between the contents of GP register Rd and a constant (the immediate), with the result stored in the destination register. 8 bits of the instruction are used for the immediate, 4 bits are used for the GP register, leaving 4 bits for the opcode. The machine instruction for ANDI is:

```
0111 KKKK dddd KKKK,
```

where 0111 is the opcode, dddd indicates the binary value of the destination register (to be added to 16; the instruction uses GP registers 16–31), and KKKKKKKK indicates the binary value of the immediate data.

Tying this together with GP register addressing, assembly code using AVR instructions can be written to load immediate data into GP registers, add the data together, and store it in a destination register. The assembly code with descriptions is given in Table 7.1.

LDI r20, 230	Load the number 230 into GP register 20
LDI r22, 15	Load the number 15 into GP register 22
ADD r20, r22	Add the contents of GP registers 20 and 22, store the result in register 20

Table 7.1: Assembly code that loads immediate data to GP registers, adds the contents together, and stores the result in a destination register.

I/O Register Addressing

I/O register addressing, shown schematically in Figure 7.5, is used when the instruction contains the address of the I/O register to be operated on. As there are 64 I/O registers, each instruction requires 6 bits to indicate the memory location. Therefore, this mode cannot be used to access extended I/O memory.

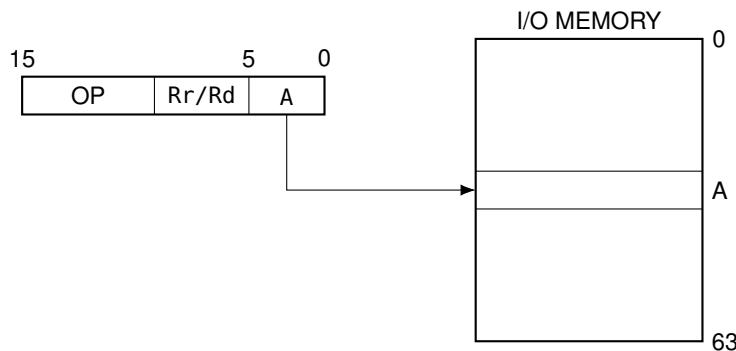


Figure 7.5: I/O register addressing occurs when an instruction contains an I/O registers as an operand.

In **I/O direct addressing**, instructions contain an I/O address (A) as well as a destination general purpose register (Rd) and/or a source general purpose register (Rr).

Notable instructions that use direct I/O addressing are IN and OUT, which either loads data from an I/O register into a destination GP register or stores the contents from a source register into one of the I/O registers, respectively. 6 bits are used to address the I/O register, 5 bits are used to address the (source or destination) GP register, and the remaining 5 bits are used for the opcode.

Assembly code can be written to load data from PIN registers into GP registers, perform a subtraction, and store the result in a destination register. The assembly code with descriptions is given in Table 7.2.

IN r5, PIND	Load the data from port D into GP register 5
IN r6, PINB	Load the data from port B into GP register 6
SUB r5, r6	Subtracts the contents of GP register 6 from the contents of register 5 and stores the result in register 5

Table 7.2: Assembly code that loads PIN data to GP registers, subtracts the contents, and stores the result in a destination register.

Data Memory Addressing

Data memory addressing is used to access the data memory (SRAM). These instructions contain two 16-bit words; the least significant word contains the 16-bit data address. **Data direct addressing**, shown schematically in Figure 7.6, contains destination and/or source general purpose registers (Rd and/or Rr). Because there are two instruction words to decode, these instructions require at least 2 clock cycles to execute.

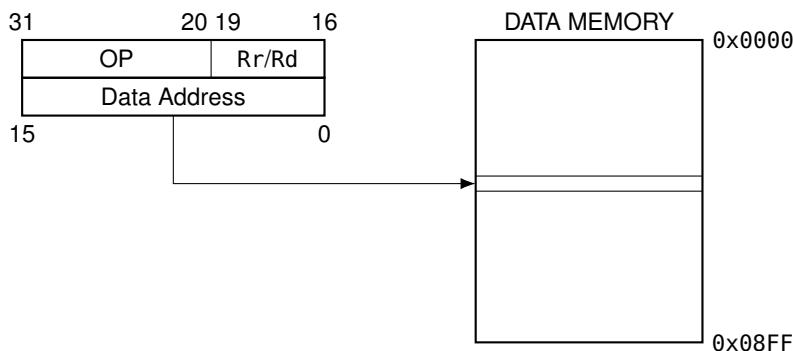


Figure 7.6: Data direct addressing occurs when an instruction contains a 16-bit data address as an operand.

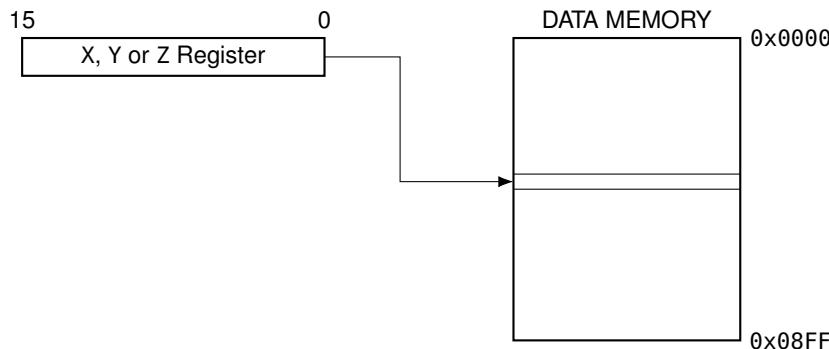
An instruction that uses direct data memory addressing is LDS, which loads data from a given address in SRAM into a destination GP register. 16 bits are available for the data memory address, which means that it is limited to addressing the first 64 KB of data stored in SRAM. This is not a problem on the ATmega328P which only has 2 KB of memory. The remaining part of the instruction contains 5 bits to address the GP register, with the remaining 11 bits used for the opcode. The machine instruction for LDS is:

```
1001 000d dddd 0000
kkkk kkkk kkkk kkkk,
```

where 100100000000 is the opcode, dddd indicates the binary value of the destination register, and kkkkkkkkkkkkkkk indicates the address in SRAM to be accessed.

Data indirect addressing occurs when the operand address is the contents of the X, Y or Z pointer register. This mode, shown schematically in Figure 7.7, is used to access extended I/O registers or the

internal SRAM. This type of addressing is also used when the address of a desired memory location is not known at the time that a program is written. Just as there are pros and cons to using direct and indirect addressing for the GP registers, there are pros and cons to using direct and indirect addressing to access data memory.

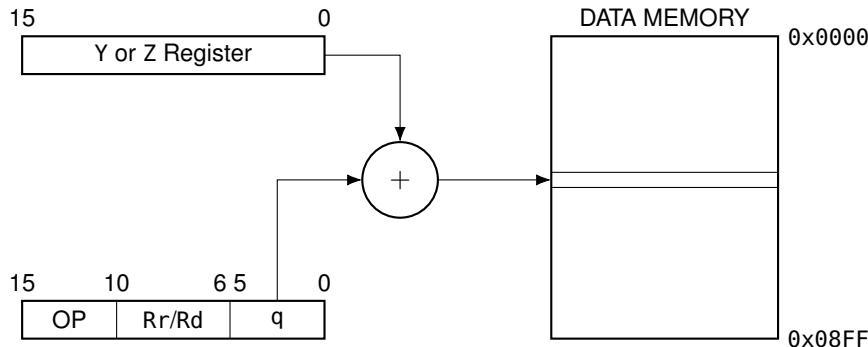


Indirect data memory addressing can also occur with any one of the following modifications:

- displacement – add a constant value to the address in pointer register Y or Z,
- post-increment – add one to the value in pointer register Y or Z after the operation, and
- pre-decrement – subtract one from the value in pointer register Y or Z before the operation.

Data indirect with displacement mode, shown in Figure 7.8 is used when the operand address is the result of the contents of Y or Z (indirect addressing pointer registers) added to the address contained in 6-bits (q) of the instruction word. This mode is useful for writing position independent code, since an address register can be used to stored a base address, with data access using displacements relative to the base.

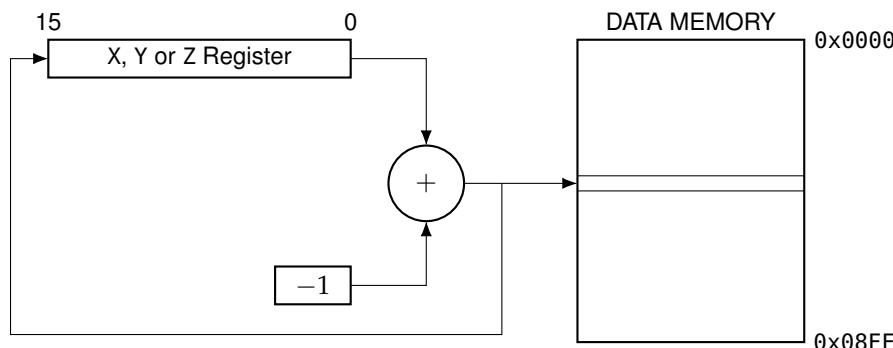
Figure 7.7: Data indirect addressing occurs when an operand address is the contents of an indirect addressing pointer register.



As an example, say that the contents of an array are stored in a known position in SRAM. The address of element 0 (150 in the example in Table 7.3) can be stored in either pointer register Y or Z. If element 15 needs to be accessed, the displacement offset would be equal to 15. This assembly code is given in Table 7.3. Note that data indirect with displacement is not available with pointer register X.

CLR r29	Clear the HIGH byte of pointer register Y
LDI r28, 150	Load the value 150 into the LOW byte of pointer register Y
LDD r4, Y+15	Loads the value stored in address 165 ($Y + 15$) into GP register 4

Data indirect with pre-decrement mode is used when the operand address is the contents of the X, Y or Z register, which is automatically decremented before the operation. This addressing mode is similarly useful for accessing array contents, for example. It is depicted schematically in Figure 7.9.



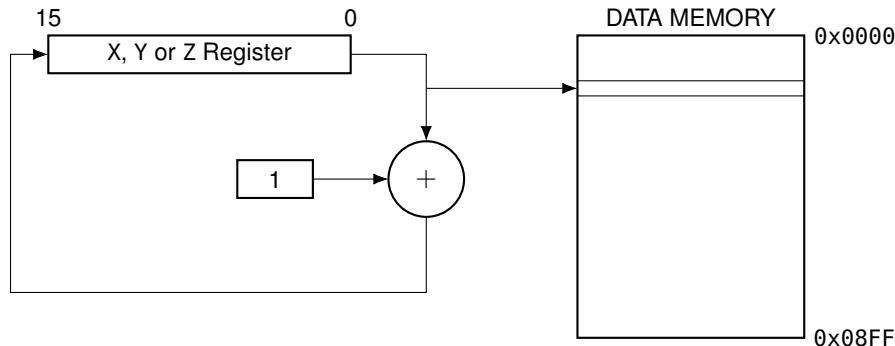
Data indirect with post-increment addressing is used when the operand address is the contents of the X, Y or Z register, which is automatically incremented after the operation. This addressing mode

Figure 7.8: Data indirect addressing with displacement occurs when an operand address is the result of the contents of an indirect addressing pointer register added to 6 bits (q) contained in the instruction word.

Table 7.3: Assembly code that loads indirect from pointer register Y with displacement.

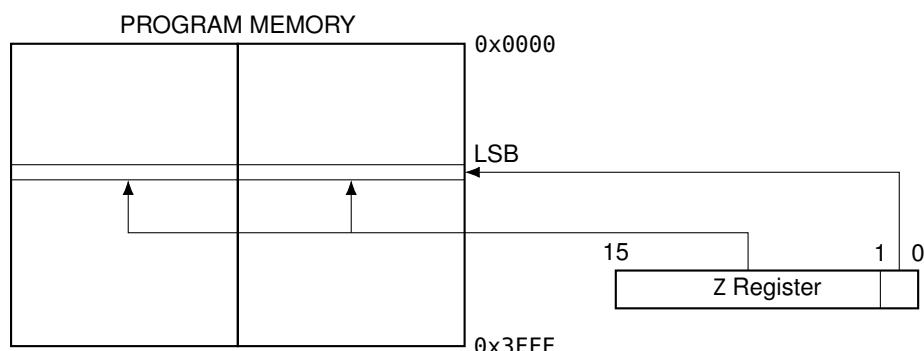
Figure 7.9: Data indirect with pre-decrement addressing occurs when an operand address is the contents of an indirect addressing pointer register, which is decremented prior to the instruction.

is also useful for accessing array contents. It is depicted schematically in Figure 7.10.



Program Memory Addressing

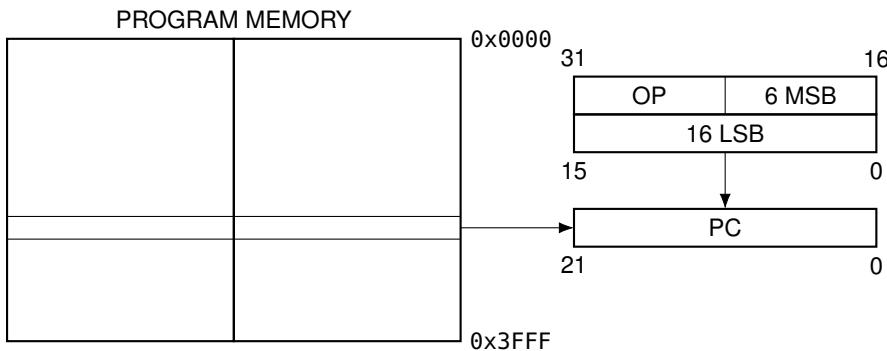
Program memory addressing is used to access the full program memory space. It requires at least 14 bits to address the total flash memory section. The Z pointer register is used to refer to the address in program memory. Because program memory stores 16-bit words, and the destination register can only store 8 bits of data, the Z register additionally specifies if the LOW byte (accessed if Z_{LSB} is 0) or HIGH byte (accessed if Z_{LSB} is 1) should be returned. Program memory addressing can also be done with a post-increment or with a pre-decrement. Program memory addressing is shown schematically in Figure 7.11.



In **direct program addressing**, instructions can contain a pointer (indirect) to be loaded to the program counter, or an 16-bit word (immediate) following the instruction to load to the program counter. The program counter can also be incremented by a constant k . Program execution will then continue at that space in program memory.

Figure 7.10: Data indirect with post-increment addressing occurs when an operand address is the contents of an indirect addressing pointer register, which is incremented after the instruction.

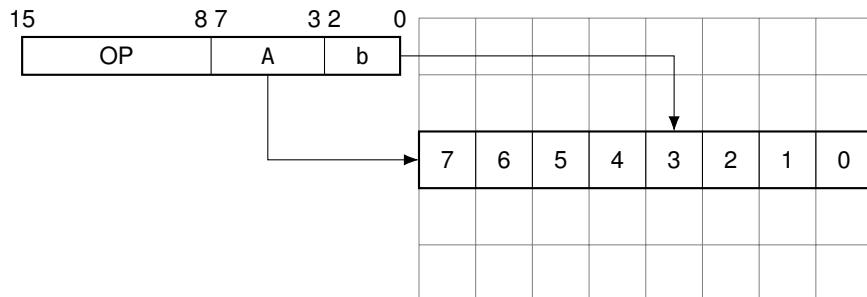
Figure 7.11: Program memory addressing accesses the full memory space given by the address in pointer register Z. The LSB of Z indicates if the low or high byte should be returned as a result.



The types of instructions that use direct programming addressing include **JMP**, which causes the program counter to load an immediate value and continue executing instructions from that location in memory and **EIJUMP**, which has the program counter jump to the memory location stored in pointer register Z.

7.5 Bit Addressing

Bit addressing is used to access specific bits in registers (especially SREG). Because most registers are 8 bits, the instruction requires 3 bits (b) to specify which bit (from 0–7) is to be read from/written to.



The instruction **CLI**, for example, is used to clear the global interrupt flag bit in SREG, which prevents interrupts from executing on the microcontroller. The global interrupt flag bit can be set again using the **SEI** instruction. These instructions do not require any operands, and thus use 16-bit opcodes.

7.6 The Stack and Stack Operations

The stack is a special area of RAM reserved for temporary data storage. It cannot overlap with the general purpose, I/O, or extended I/O memory registers. Therefore, the memory address at which it

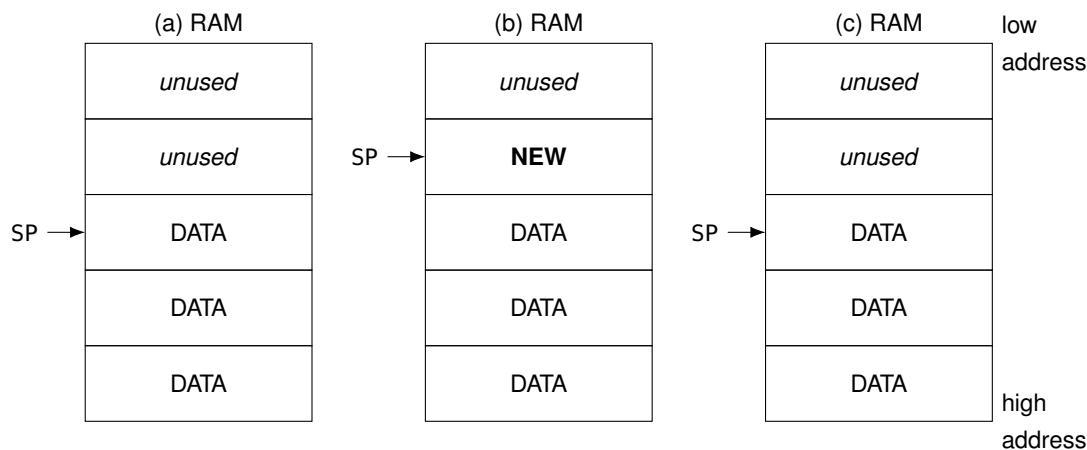
Figure 7.12: Direct program addressing accesses the full memory space given by a pointer loaded into the program counter.

Figure 7.13: Bit addressing is used to access specific bits in registers.

starts must be greater than or equal to `0x0100`. Usually, the stack is initialized at the highest available address in RAM. (This is done automatically on the ATmega328P microcontroller, which initializes the stack pointer register at a value equal to `RAMEND`, the last address of the internal SRAM.) The current location of the most recent data byte in the stack is known as the stack pointer (SP). The stack pointer consists of two 8-bit registers that exist in the I/O space.

The stack itself can be visualized as a stack of plates in a cafeteria; new plates can be added to the stack, and when plates are accessed, the last one in is always the first one out (last-in, first-out).

Data can be added to the stack using the `PUSH` instruction. When this occurs, new information is placed into temporary storage, and the value of SP is decremented. Removing data from the stack occurs using the `POP` instruction. The value of SP is incremented during this instruction. The contents of the stack, as well as the location of the stack pointer, is depicted schematically in Figure 7.14 after both a `PUSH` and `POP` operation.



Use of the stack is essential during a subroutine call (external function or an ISR). In order to return to the exact spot in memory where the program left to execute the subroutine, the return address is `PUSHed` to the top of the stack before the location of the subroutine is accessed. After the subroutine is complete, the program goes back to the location and the return address is `POPed` out of the stack.

Figure 7.14: Stack operations. (a) Memory in the stack, (b) after a `PUSH` instruction, and (c) after a `POP` instruction.

Stack Problems

Care must be taken when allocating data memory to be used in the stack. **Stack overflow** is a situation in which too much data is pushed onto the stack which causes the SP to point to an address outside of the stack memory area. **Stack underflow** is a situation in which

too much data is popped from the stack so that the SP points to an address below the stack bottom. A **stack collision** occurs when the stack is allocated to memory addresses which overlap with data registers.

7.7 Practice Problems

1. How many bytes in memory can be accessed if 6 bits are used in the instruction for the memory address? 64
2. In indirect addressing, a 16-bit pointer register is used to address memory. How many bytes can be accessed in this scenario? 64 K
3. Use the table below and $Y = 0x0103$ to determine the value stored in the destination register, as well as the value of pointer register Y, after each of the following subsequent operations.

Address	SRAM Contents
0x0100	0xC1
0x0101	0xFB
0x0102	0x3F
0x0103	0xE9
0x0104	0x95
0x0105	0x9A
0x0106	0x1C
0x0107	0xDC

- (a) LD r4, Y $r4 = 0xE9, Y = 0x0103$
- (b) LD r5, Y+ $r5 = 0xE9, Y = 0x0104$
- (c) LD r6, Y $r6 = 0x95, Y = 0x0104$
- (d) LD r7, Y+3 $r7 = 0xDC, Y = 0x0107$
- (e) LD r8, Y- $r8 = 0x1C, Y = 0x0106$
- (f) LDS r9, 0x0102 $r9 = 0x3F, Y = 0x0106$

8

Model Microcontroller

TO BETTER UNDERSTAND HOW A MICROCONTROLLER WORKS, a simple model microcontroller will be designed. It will have an ALU to perform all required operations, two data registers, four input devices, and four output devices. Each of the registers, input devices, and output devices has its own numerical label to determine with which component each instruction should interact. This model microcontroller is shown in Figure 8.1.

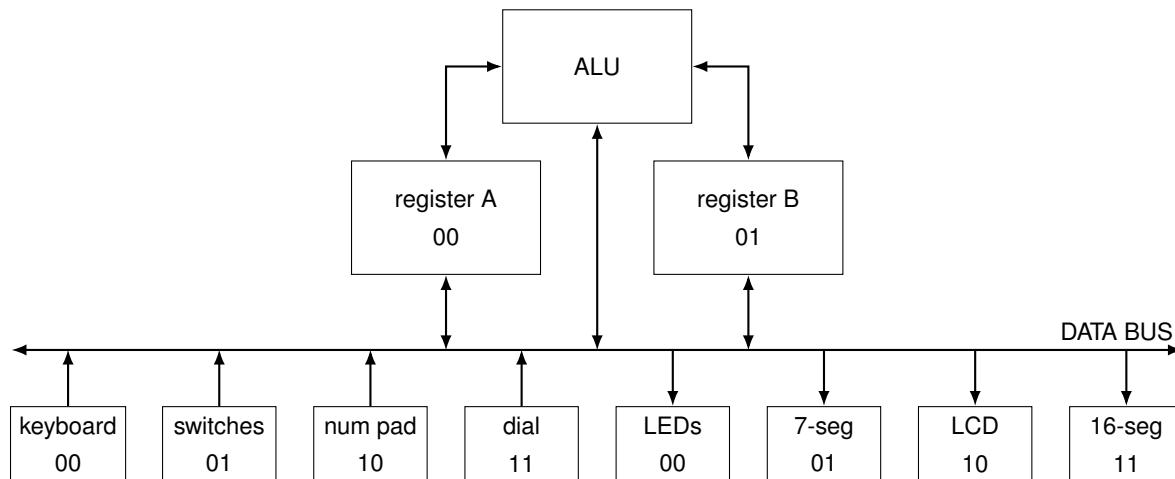


Figure 8.1: A block diagram of the model microcontroller with input and output devices, two registers, and an ALU.

8.1 Microcontroller Instructions

Microcontrollers are capable of carrying out specific machine instructions. An instruction consists of an operation code (opcode), which dictates what function to do (add, subtract, XOR, etc.). The ALU contains specialized hardware to accomplish each of these functions. Each source of data used to carry out the function is known as an

operand. A full listing of the ATmega328P instructions is available in the AVR instruction manual.

8.2 Microcontroller Operation Codes (Opcodes)

Recall that for every n opcodes, there must be a $\lceil \log_2(n) \rceil$ bit number to store it, assuming that expanding opcodes are not used. If there are 40 operations, 6 bits of the instruction code are necessary to store the opcode. The model microcontroller will have 7 operations. Therefore, 3 bits are necessary for storing this information. However 4 bits will be used to create 8-bit instructions.

It is now possible to come up with a list of instructions that the model microcontroller can perform, and indicate the operands that are required for each. These instructions are given in Table 8.1.

Instruction	Opcode	Description	Operands
IN	0000	Load data from input	I/O(A), Rd
OUT	0001	Store data to output	I/O(A), Rr
MOV	0010	Copy data	Rr, Rd
ADD	0011	Add data	Rr, Rd
SUB	0100	Subtract data	Rr, Rd
AND	0101	Logical AND	Rr, Rd
OR	0110	Logical OR	Rr, Rd

Table 8.1: Listing and description of the instructions available from a model microcontroller.

IN – Load data from input to register

The IN instruction loads data from an input device to one of the registers. In order to carry out this instruction, the designation of the input device (A) as well as the designation of the destination register (Rd) need to be known.

OUT – Store data from register to output

The OUT instruction takes data from a source register and sends it to one of the output devices. In order to carry out this instruction, the designation of the output device (A) as well as the designation of the source register (Rr) need to be known.

MOV – Copy data from one register to another

The MOV instruction takes data from a source register and copies it into a destination register. In order to carry out this instruction, the

designations of both the source (Rr) and the destination (Rd) registers need to be known.

ADD – Add data from register Rr to Rd and store in Rd

The ADD instruction takes data from two registers, adds the contents, and stores the sum into a destination register. In order to carry out this instruction, the designations of both the source (Rr) and the destination (Rd) registers need to be known.

SUB – Subtract data in register Rr from Rd and store in Rd

The SUB instruction takes data from a source register, subtracts it from the contents of the destination register, and then stores the result into a destination register. In order to carry out this instruction, the designations of both the source (Rr) and the destination (Rd) registers need to be known.

AND – Logical AND data in register Rr with Rd and store in Rd

The AND instruction takes data from a source register, performs a logical AND operation with the contents of the destination register, and then stores the result into a destination register. In order to carry out this instruction, the designations of both the source (Rr) and the destination (Rd) registers need to be known.

OR – Logical OR data in register Rr with Rd and store in Rd

The OR instruction takes data from a source register, performs a logical OR operation with the contents of the destination register, and then stores the result into a destination register. In order to carry out this instruction, the designations of both the source (Rr) and the destination (Rd) registers need to be known.

Each of the model microcontroller instructions can be expressed in machine code (binary), assembly (using an opcode and operands), or using register-transfer language (depicting the movement of data from one register to another). This is outlined in Table 8.2.

Machine Instruction	Assembly	Register Transfer
0000 AAdd	IN Rd,A	Rd \leftarrow I/O (A)
0001 rrAA	OUT A,Rr	I/O (A) \leftarrow Rr
0010 rrrd	MOV Rd,Rr	Rd \leftarrow Rr
0011 rrdd	ADD Rd,Rr	Rd \leftarrow Rd + Rr
0100 rrdd	SUB Rd,Rr	Rd \leftarrow Rd - Rr

Table 8.2: Model microcontroller instructions in machine code, register transfer language, and assembly.

8.3 Model Microcontroller Program

Now a set of instructions that the model microcontroller can understand can be created. A sample set of instructions is provided in Table 8.3. The program will be written in machine code, assembly language, and register transfer language.

Step	Instruction
1	Input data from the switches into register A
2	Input data from the dial into register B
3	Output the contents of register A to the LCD screen
4	Output the contents of register B to the LEDs
5	Add the contents of both registers, and store the result in register A
6	Output the contents of register A to the 16-seg. display

Table 8.3: Program instructions for the model microcontroller.

Machine Code

The binary numbers in the machine code are what's actually saved to the microcontroller memory. Using hexadecimal makes the list of instructions somewhat less unwieldy. Based on the sample program defined in Table 8.3, the corresponding machine instructions are given in Table 8.4.

Step	Machine Code	Hex
1	0000 0100	0x04
2	0000 1101	0x0D
3	0001 0010	0x12
4	0001 0100	0x14
5	0011 0100	0x34
6	0001 0011	0x13

Table 8.4: Program instructions for the model microcontroller in machine code given in both binary and hexadecimal.

Assembly Language and Register Transfer Language

Assembly language allows instructions to be invoked while having a somewhat easier time reading, understanding, and debugging the program contents. Register transfer language is another abstraction that allows for the visualization of movement of data from one location to another, along the location of the arrow. Based on the sample program defined in Table 8.3, the corresponding machine instructions are given in Table 8.5.

Step	Assembly	Register Transfer Language
1	IN A,switches	$A \leftarrow \text{I/O}(\text{switches})$
2	IN B,dial	$B \leftarrow \text{I/O}(\text{dial})$
3	OUT LCD,A	$\text{I/O}(\text{LCD}) \leftarrow A$
4	OUT LEDs,B	$\text{I/O}(\text{LEDs}) \leftarrow B$
5	ADD A,B	$A \leftarrow A+B$
6	OUT 16seg,A	$\text{I/O}(16\text{seg}) \leftarrow A$

Table 8.5: Program instructions for the model microcontroller in assembly language and register transfer language.

Model Microcontroller Arithmetic and Logic Unit (ALU)

The ALU of the model microcontroller is depicted schematically in Figure 8.2. This schematic diagram shows all of the internal hardware that is required to carry out each operation. The intricacies of the instruction decoder have been omitted for simplicities sake.

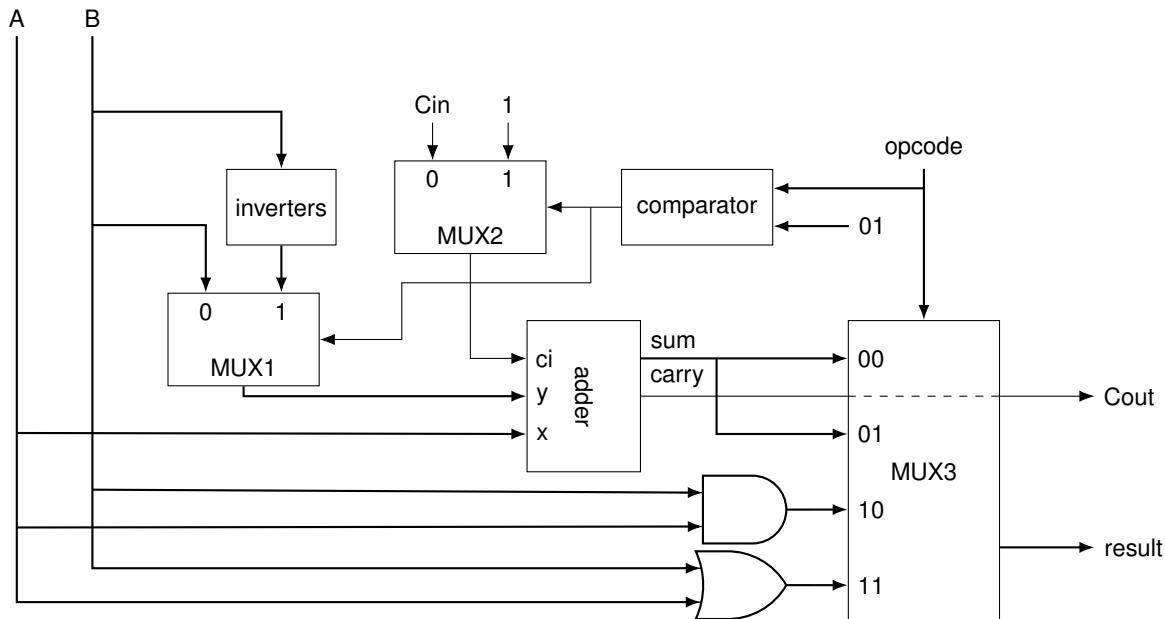


Table 8.6 describes the operation of the hardware of each of these instructions.

Opcode oo: Add

Opcode oo is sent to the comparator, which outputs a value of o
MUX2 output: Cin, which enters the adder
MUX1 output: B, which enters the adder

Figure 8.2: Hardware schematic of a simple arithmetic and logic unit (ALU).

Table 8.6: Listing and description of the four opcodes available on the simple arithmetic and logic unit.

A also enters the adder

MUX₃ output: sum

The carry is sent as Cout

Opcode 01: Subtract

Opcode 01 is sent to the comparator, which outputs a value of 1

MUX₂ output: 1, which enters the adder

MUX₁ output: B', which enters the adder (this creates a 2's complement value of B)

A also enters the adder

MUX₃ output: (A + (-B))

The carry is sent as Cout

Opcode 10: AND

Opcode 10 is sent to the comparator, which outputs a value of 0

A and B both go to the AND gate

MUX₃ output: AB

Opcode 11: OR

Opcode 11 is sent to the comparator, which outputs a value of 0

A and B both go to the OR gate

MUX₃ output: A+B

8.4 Practice Problems

Using the available instructions for the model microcontroller, write assembly commands to execute the following tasks.

1. Load data from the keyboard into register A. IN A,keyboard

2. Load data from the number pad into register B. IN B,numpad

3. Subtract the contents of register A from register B and store the result in register B. SUB B,A

4. Move the contents of register B to register A. MOV A,B

5. Display the contents of register A on the 7-segment display. OUT 7seg,A

9

The ATmega328P Microcontroller

THE ATMEGA328P microcontroller is featured on the Arduino Uno board and contains a number of peripheral functions that will be described in detail in this book.

9.1 Pinout Diagrams

The ATmega328P is available in four different packages. The 28-pin plastic dual in-line package (PDIP) is used in breadboarded designs. The pinout diagram of the ATmega328P's PDIP integrated circuit package is shown in Figure 9.1.

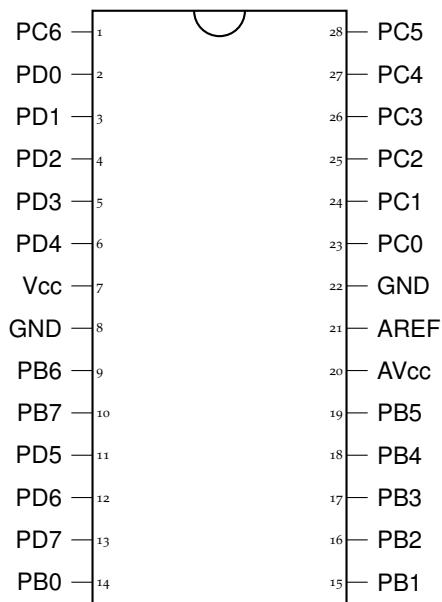


Figure 9.1: Pinout diagram for 28-pin PDIP package.

The other three packages are surface mount designs. First, there is a 32-lead thin quad flat package (TQFP). The close spacing of pins allows for more connections to be made (relative to the DIP architecture) in the same (or less) area of space. This chip has four more

pins than the PDIP. Two of the pins are extra ground connections. The other two include connections to additional analog to digital conversion channels. However, this type of pin configuration can be much more complicated for use in hobbyist projects, especially without knowledge of surface mount soldering techniques. The pinout diagram for the TQFP integrated circuit chip is shown in Figure 9.2.

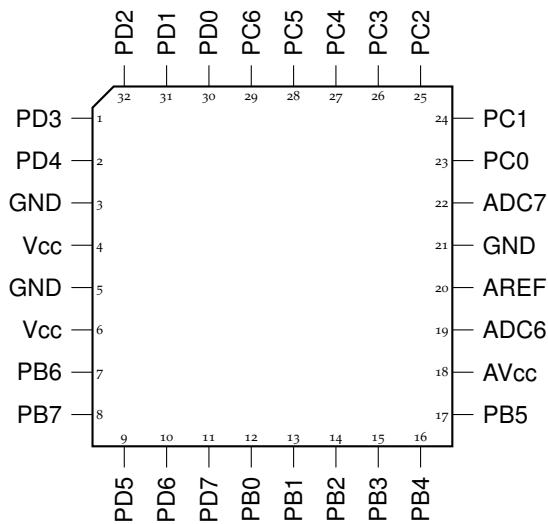


Figure 9.2: Pinout diagram for 32-lead TQFP package.

The second surface mount package is a 28-pad quad flat no-leads package (QFN) which, as the name implies, has no leads protruding from the integrated circuit chip. As with the 28-pin PDIP format, this lacks the two extra analog to digital conversion channels that are available in the 32-pin chips. The pinout diagram for the 28-pad QFN integrated circuit chip is shown in Figure 9.3.

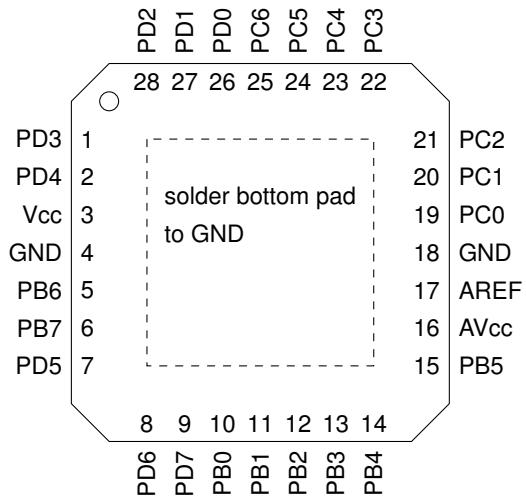


Figure 9.3: Pinout diagram for 28-pad QFN package.

The final surface mount format is a 32-pad QFN package. This chip also contains no physical leads, and, as a 32-pad device, contains the two extra analog to digital converter channels. This format's pinout diagram is given in Figure 9.4.

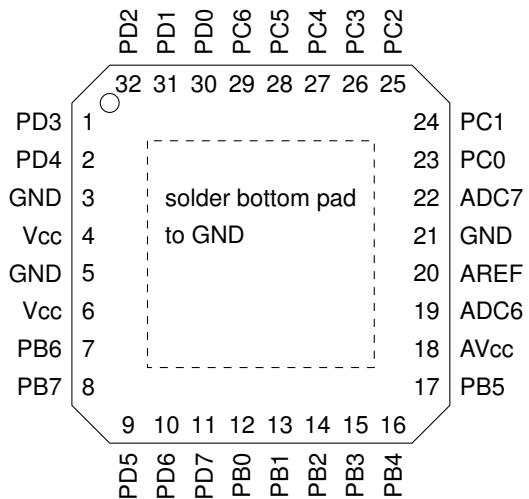


Figure 9.4: Pinout diagram for 32-pad QFN package.

9.2 Pin Details

As will be mentioned in [chapter 10](#), each of the pins on the ATmega328P serves multiple purposes. This allows a pin to act either as an input/output interface or to provide a connection to a peripheral feature.

Pin PC6 can either be used as an active-LOW reset pin or as a general purpose I/O pin, depending on the fuse settings (described later in this chapter). On the Arduino Uno, pin PC6 is used as an active-LOW reset and is connected to a pushbutton on the printed circuit board.

Supply Pins

Importantly, the Vcc pin must be connected to a proper operating voltage, which is between 2.7 V and 5.5 V. It is always recommended to use a bypass capacitor (a 100 nF ceramic capacitor will likely be adequate for the speeds at which the ATmega328P will be operated) between power and ground connections in close proximity to the chip on any digital device operated at high frequencies. A bypass capacitor will reduce noise on the power and ground connections and provide a stable operating voltage.

The AVcc pin is used as a supply voltage for the analog to digital converter (ADC). This pin should be externally connected to Vcc even

if the ADC is not being used in a circuit design. If the circuit is not sensitive to noise, connect AVcc to Vcc and insert a bypass capacitor to the pin. If the circuit needs more extensive noise filtering, a low-pass filter should be connected to the pin in accordance with the microcontroller datasheet.

9.3 Writing Programs to Memory

By connecting the Arduino Uno to a computer with a USB cable and using the Arduino IDE, programs and data can be written to the ATmega328P memory using a serial communication protocol called USART. A section of program memory known as the **bootloader** instructs the microcontroller to receive external information and write it to program memory. The Arduino Uno can also be programmed using another serial protocol known as SPI. Otherwise, a conventional non-volatile memory programmer can be used to program the microcontroller.

A non-volatile memory programmer can be used to program the ATmega328P microcontroller chip external to the Arduino Uno. While the Arduino package has many features and is convenient to use, it is useful to be able to program individual microcontroller chips for small-size, low-power, or low-cost situations.

9.4 Fuse Bytes

Although the microcontroller contains flash memory, EEPROM, SRAM, and general purpose registers, it also contains non-volatile fuse bytes to affect the functioning of the microcontroller external to the program operations. There are three fuse bytes used to control the microcontroller: the extended fuse byte, the low fuse byte, and the high fuse byte. Programmed fuses have a value of 0, while unprogrammed fuses have a value of 1.

Extended Fuse Byte

This fuse configures the brown-out detection unit. If a power supply fluctuates, it is possible for the microcontroller to malfunction if the voltage supplied to the Vcc pin dips below a certain value. The brown-out detection unit resets the microcontroller while the voltage is underneath a given threshold as programmed by the extended fuse byte.

High Fuse Byte

This fuse configures features that are relevant to programming and debugging the ATmega328P microcontroller. Among other things, the bits in the high fuse byte control the ability to use serial programming, to use the watchdog timer (which is explained in [chapter 14](#)), to preserve EEPROM memory when writing new program memory to the device, and to configure the amount of memory set aside to act as a bootloader.

Low Fuse Byte

This fuse configures features that are relevant to the clock source used to operate the microcontroller. There are two clock sources internal to the ATmega328P, but the device may also be clocked from an external source. The start-up time is also affected by changing the low fuse byte. Clock sources require a sufficiently high value of Vcc before it starts oscillating, and it must oscillate a sufficient number of times before the clock can be considered stable. A time-out delay is used after device reset to ensure a sufficient value of Vcc.

9.5 Practice Problems

1. The reset pin is active-_____. LOW
2. What is the operating voltage range of the ATmega328P? 2.7–5.5 V
3. How many fuse bytes are available on the ATmega328P? 3
4. What are the names of the fuse bytes? extended, high, low
5. True or false: Upon power-up, the CPU starts working immediately. FALSE
6. What is the purpose of brown-out detection? To protect the CPU from malfunction if the power supply dips below a given voltage level.

10

I/O Port Registers

COMPUTERS ARE ONLY INTERESTING insofar as they can interact with input and output devices. The gatekeeper between a microcontroller and external devices are I/O pins. The ATmega328P has three ports that connect to each of the I/O pins. These ports are called Port B, Port C, and Port D.

The pins associated with each of these ports is listed in Table 10.1. The labeling used for the ATmega328P differs from the labeling used by Arduino. Note that the ATmega328P has more available I/O pins than the Arduino. The Arduino platform has external hardware connected to pins PB6, PB7, and PC5, so they are not available for general I/O usage. The Arduino labeling of pins in Port C with the letter "A" is due to the fact that these pins can act as an input source to the analog to digital converter (ADC), which is discussed in chapter 11. This does not mean that the pins on Port C are "analog" pins. All I/O pins on the ATmega328P are digital I/O pins.

Table 10.1: I/O pin labeling on the ATmega328P and Arduino.

Port B								
bit:	7	6	5	4	3	2	1	0
ATmega328P:	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
Arduino:	-	-	D13	D12	D11	D10	D9	D8

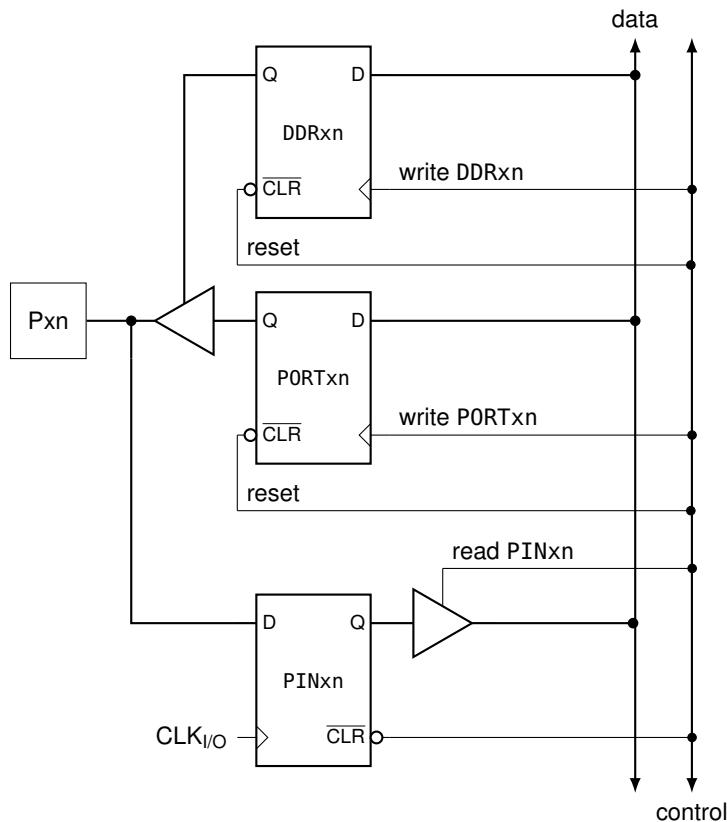
Port C								
bit:	7	6	5	4	3	2	1	0
ATmega328P:	-	PC6	PC5	PC4	PC3	PC2	PC1	PC0
Arduino:	-	-	A5	A4	A3	A2	A1	A0

Port D								
bit:	7	6	5	4	3	2	1	0
ATmega328P:	PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0
Arduino:	D7	D6	D5	D4	D3	D2	D1	D0

10.1 Data Direction

Each I/O pin on the ATmega328P is bidirectional. Through associated registers, each pin can be configured as an input (which allows the microcontroller to read its corresponding logic level) or an output (which allows the microcontroller to send data to the pin).

Figure 10.1 shows a schematic of how this works.



The directionality of each I/O pin can be configured with the data direction register associated with that pin's port. The three data direction registers on the ATmega328P are **DDRB**, **DDRC**, and **DDRD**. By default upon startup, each bit in each data direction register has a value of 0, which corresponds to each I/O pin being configured as an input pin. By writing a 1 to a pin's bit in its corresponding data direction register, that pin will be configured as an output pin.

Writing data to an output pin can be accomplished by sending data to that pin's port data register. Each port on the microcontroller has its own port data register: **PORTB**, **PORTC**, and **PORTD**. Reading data from an input pin can be accomplished by reading data from the pin's pin register. The three pin registers are **PINB**, **PINC**, and **PIND**.

The data direction, port data, and pin registers are all depicted in

Figure 10.1: Hardware schematic of I/O port registers: **DDRxn** controls the data direction, **PORTxn** controls the data sent to output pins, and **PINxn** contains the data state of input pins.

Figure 10.1.

10.2 Electrical Characteristics

It is important to understand the electrical characteristics of I/O pins in order to interface external devices with the microcontroller properly. The two characteristics of primary concern are **voltage-level compatibility** and **current-level compatibility**.

Voltage-Level Compatibility

To ensure that the voltage levels of external devices are able to properly interface with the ATmega328P, the voltages required for a logic HIGH and logic LOW must be known for both devices. (The values for the ATmega328P can be found in the Common DC Characteristics section of the datasheet.) The following four voltages must be compared and checked for compatibility:

- **Input HIGH voltage (V_{IH})** – This is the voltage level that is treated as a logic 1 when applied to the input of a device.
- **Input LOW voltage (V_{IL})** – This is the voltage level that is treated as a logic 0 when applied to the input of a device.
- **Output HIGH voltage (V_{OH})** – This is the output voltage level when a digital circuit outputs a logic 1. The output HIGH voltage of circuit X must be greater than or equal to the input HIGH voltage of circuit Y, in order for circuit X to properly drive circuit Y.
- **Output LOW voltage (V_{OL})** – This is the output voltage level when a digital circuit outputs a logic 0. The output LOW voltage of circuit X must be less than or equal to the input LOW voltage of circuit Y, in order for circuit X to properly drive circuit Y.

Current-Level Compatibility

In order for the ATmega328P to properly drive external devices, the current drive capability must be understood. The ATmega328P is capable of supplying current when the output voltage is HIGH (current flows from the microcontroller to the device), or sinking current when the output level is LOW (current flows from the device to the microcontroller).

Each I/O pin on the ATmega328P is capable of sourcing and sinking 40 mA. The maximum current rating of the microcontroller is 200 mA. All logic devices have four current values that are involved

in determining how much current is necessary to properly derive the device:

- **Input HIGH current (I_{IH})** – This is the input current that must flow into the input pin to create a HIGH logic level.
- **Input LOW current (I_{IL})** – This is the input current that must flow out of the input pin to create a LOW logic level.
- **Output HIGH current (I_{OH})** – This is the output current that flows out of the output pin when the output logic level is HIGH.
- **Output LOW current (I_{OL})** – This is the output current that flows into the output pin when the output logic level is LOW.

10.3 Internal Pull-Up Resistors

There are internal pull-up resistors built in to each digital I/O pin on the ATmega328P. They have a value of approximately $36\text{ k}\Omega$. In order to make use of the pull-up resistors, write a value of 1 to a pin when it is configured as an input. The input pull-up resistors can be globally turned off by setting bit 4 (PUD, pull-up disable) in the Microcontroller Control Register (MCUCR), but by default this bit is configured to enable the usage of the internal pull-ups. The I/O pin hardware that controls the input pull-up is shown in Figure 10.2.

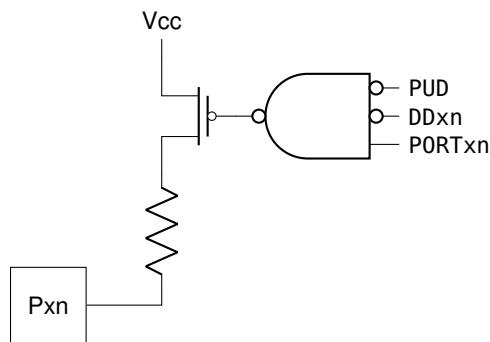


Figure 10.2: Hardware schematic of I/O port input pull-up resistors.

10.4 Alternate Pin Functions

All of the I/O pins on the ATmega328P microcontroller serve alternative purposes. This increases the functionality of the microcontroller without requiring extra space for extra pins. Some of these alternative functions include acting as sources for external interrupts, providing functionality for serial communication, converting analog to digital signals, etc. Each pin has extra hardware that allows for

this extra functionality. Multiplexers route control signals to either have the pins act as regular I/O pins, or to function with their alternate purpose. The I/O pin multiplexers are shown schematically in Figure 10.3.

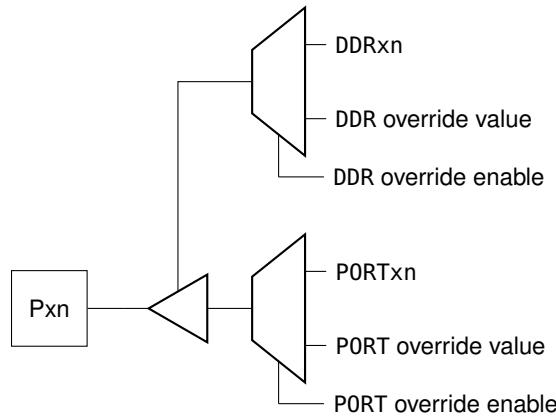


Figure 10.3: Hardware schematic of alternate I/O port functionality.

10.5 Practice Problems

1. True or false: All bits of Port B are accessible on the ATmega328P. FALSE
2. True or false: To change the data direction of pins on Port B, the register **DDRB** is used. TRUE
3. What is the difference between **PORTC=0x00** and **DDRC=0x00**? **PORTC=0x00** sets the output value of all pins on Port C to be equal to logic LOW. **DDRC=0x00** sets the directionality of all pins on Port C to be input pins.
4. True or false: All of the ATmega328P ports have 8 bits. TRUE
5. True or false: Upon power-up, the I/O pins are configured as output ports. FALSE
6. True or false: The **PORTx** register is used to send data out to ATmega328P pins. TRUE
7. True or false: The **PINx** register is used to bring data into the CPU from ATmega328P pins. TRUE

11

Analog to Digital Conversion

As will be discussed in [CHAPTER 12](#), many sensors output analog values. Table 11.1 gives a list of various types of digital and analog sensors.

Digital	Analog
Pushbutton	Potentiometer
Toggle Switch	Microphone
Keypad	Temperature sensor
Encoder	Photo-diode, -resistor

Table 11.1: Listing of digital and analog sensors.

Analog values must be converted into digital values in order to work with the digital functionality of the microcontroller. By nature, analog signals are continuous, and they contain an infinite number of points, as shown in Figure 11.1.

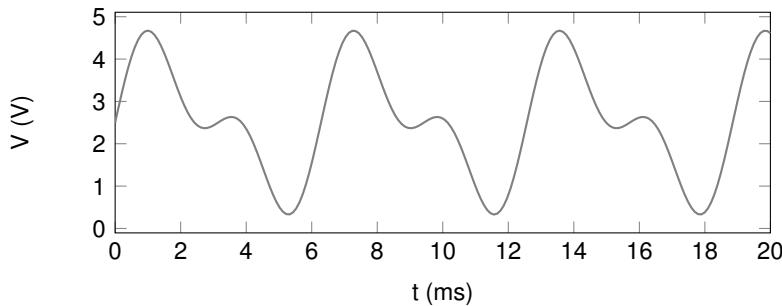


Figure 11.1: An analog signal takes on continuous values and contains an infinite number of points.

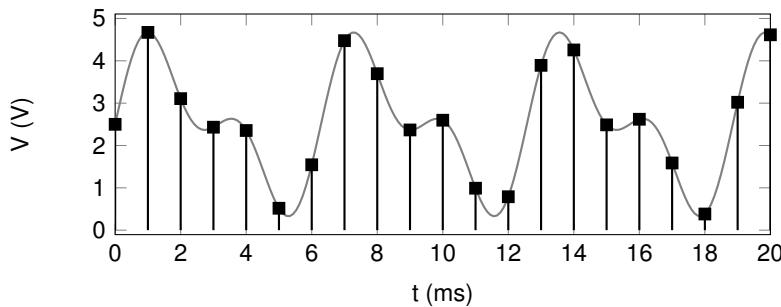
The voltage values of an analog signal can take on any real number between some minimum and maximum voltage. This textbook will consider the minimum voltage to be ground and the maximum voltage to be equal to a reference voltage (V_{REF}).

To convert an analog voltage signal into digital values, the analog signal must be sampled, quantized, and encoded. A so-called single-

ended analog to digital converter will convert a single analog signal into a digital value. A differential analog to digital converter will determine the difference between two analog signals.¹ This textbook will consider single-ended analog to digital conversion (ADC).

11.1 Sampling

An infinite number of points can't be digitized by a microcontroller because it would require infinite processing power and infinite memory. Therefore, the analog signal must be sampled periodically. This sample rate is very important when doing signal processing (audio, video, etc.). It is inversely related to conversion accuracy. The fine details of sampling rate are beyond the scope of this course; you will very likely study the topic in a signals and systems electrical engineering course. Figure 11.2 shows the same analog signal sampled every millisecond. At each point in time, the sampled data is held in memory and then quantized and encoded.



¹ Microchip, "Differential and Single-Ended ADC," August 2019.

11.2 Quantization

Because digital data cannot take on a continuum of values, the sampled analog data must be set equal to one of a number of discrete values. Figure 11.3 shows an analog signal that has been quantized. At every sampled point in the signal, that value is rounded down to the nearest possible value of $V_{REF} \div 2^n$, where n refers to the number of bits of system resolution.

Figure 11.2: An analog signal sampled every millisecond.

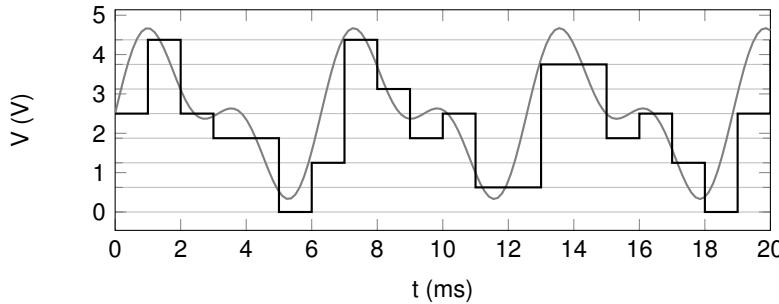


Figure 11.3: An analog signal sampled every 1 ms that has been quantized in a 3-bit system.

The number of possible discrete values available depends on the **resolution** of the system. The resolution of an ADC is represented in bits. An n -bit ADC has 2^n possible discrete values.

While low resolution data can suffer quantization error (described below), higher resolution does not always lead to “better” results. The higher the resolution of the ADC, the more memory is required to store converted data. Higher resolution ADCs also typically require a lower sampling rate. In circuits prone to noise, high resolution quantization may not provide accurate data in the least-significant bits of the result. It is therefore important to understand the limitations of circuit hardware when selecting the resolution of an ADC.

Other ADC metrics include the **step size**, which is the minimum change in voltage required to change the output value. The step size is given by Equation 11.1.

$$\Delta V = \frac{V_{REF}}{2^n} \quad (11.1)$$

Quantization Error

The **quantization error** refers to the difference between the actual and quantized voltage and is defined by Equation 11.2.

$$\text{error} = V_{\text{quantized}} - V_{\text{actual}} \quad (11.2)$$

Quantization error can be reduced by increasing the resolution of the ADC (as shown in Figure 11.4), or by increasing the sample rate (not shown).

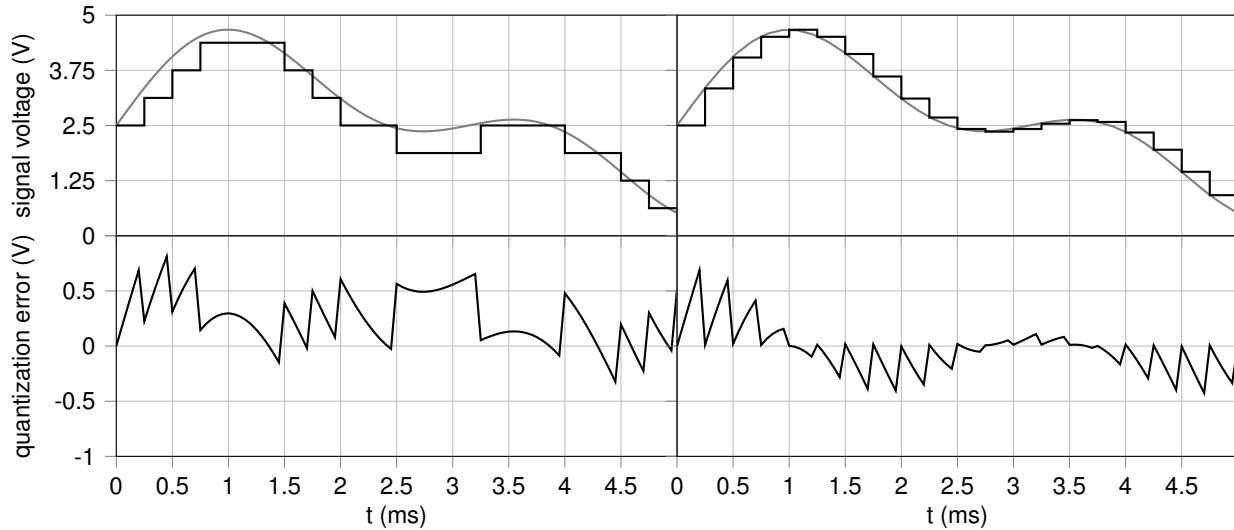


Figure 11.4: Quantized analog signals and associated quantization error for a 3-bit ADC (left) and an 8-bit ADC (right). The sample rate is 4× per millisecond.

Other types of error are possible in analog to digital converters and may depend on the architecture of the ADC. These will typically be listed in an ADC datasheet.

11.3 Encoding

The encoding part of ADC comes when the sampled and quantized signal is then converted into a binary number. In hardware, this can be accomplished with a priority encoder.

The binary value stored by the ADC is given by Equation 11.3, where V_{IN} is the analog voltage input.

$$\text{ADC result} = 2^n \times \left(\frac{V_{IN}}{V_{REF}} \right) \quad (11.3)$$

11.4 ADC Architectures

There are many different hardware approaches to realizing an ADC. Each architecture has tradeoffs between size, resolution, and data processing time. While only three architectures are outlined here, there are many other possibilities that exist to convert analog signals to digital values.²

Direct Conversion (Flash-Type) ADC

A flash-type ADC uses analog comparators to quantize an analog input signal. An analog comparator is a circuit element that has two inputs, V_p (positive input) and V_n (negative input), and one output

² Kester, Walt, "Which ADC Architecture Is Right for Your Application?", *Analog Dialogue*, June 2005.

V_{out} . The output becomes equal to Vcc if $V_p \geq V_n$. Otherwise, the output will be equal to GND. The circuit symbol of a comparator is shown in Figure 11.5.

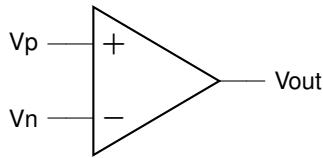


Figure 11.5: The circuit symbol of an analog comparator.

A 3-bit flash-type ADC (which requires seven analog comparators) is shown in Figure 11.6. A voltage divider (composed of multiple identical resistors wired up in series between Vcc and GND) is used to create fractional values of Vcc at each comparator's negative input pin.

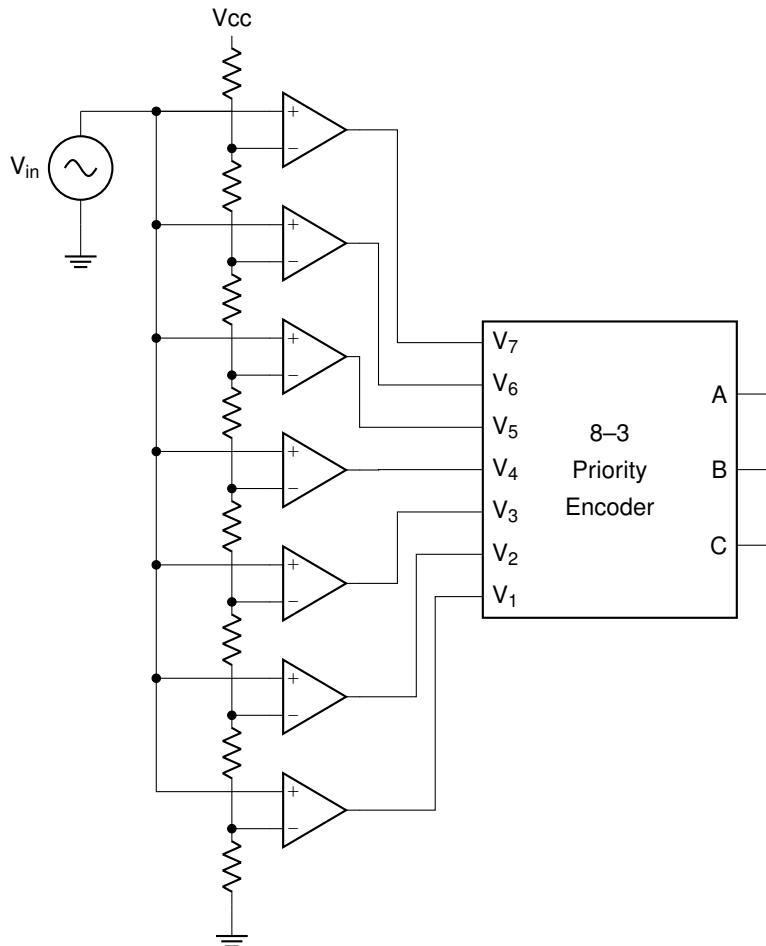


Figure 11.6: A 3-bit flash analog to digital converter.

Each comparator output ($V_{out,x}$) becomes equal to Vcc if the inequality in Equation 11.4 is satisfied, where the input voltage (applied to the positive pin) is V_{in} , x is the number of the comparator,

and m is the number of bits of system resolution.

$$V_{in} > V_{cc} \times \left(\frac{x}{2^m} \right) \quad (11.4)$$

For example, in the 3-bit flash-type ADC shown in Figure 11.6, the output $V_{out,6}$ becomes equal to V_{cc} when the inequality in Equation 11.5 is satisfied.

$$V_{in} > V_{cc} \times \left(\frac{6}{2^3} \right) \quad (11.5)$$

The output of each comparator is connected to a priority encoder, which converts the quantized signal into a binary output value that can be stored in a data register.

Flash ADCs are quick. Because they are able to process all bits of data simultaneously, they require no sample-and-hold circuitry and would only need a single clock cycle to compute the output (assuming the comparators and encoder are fast enough to have a stable output value at that time). However, they consume relatively high amounts of power and also require $2^m - 1$ comparators, which means they require a form factor that is unreasonably large for high resolution data converters. A 10-bit ADC would require 1023 analog comparators!

Successive Approximation Register (SAR) ADC

A successive approximation register works by comparing the analog input to half of the value of V_{REF} , then depending on the result, comparing in intervals of half the remaining difference until the final result is determined.

A process flow of how a 4-bit SAR ADC calculates values is shown in Figure 11.7. The analog input value is first compared with one half of the maximum value. If the analog input is larger than this value (indicated by the letter L), then it is compared to successively larger approximations of the value. If the analog input is smaller than this value (indicated by the letter S), then it is compared to successively smaller approximations of the value until settling on the result.

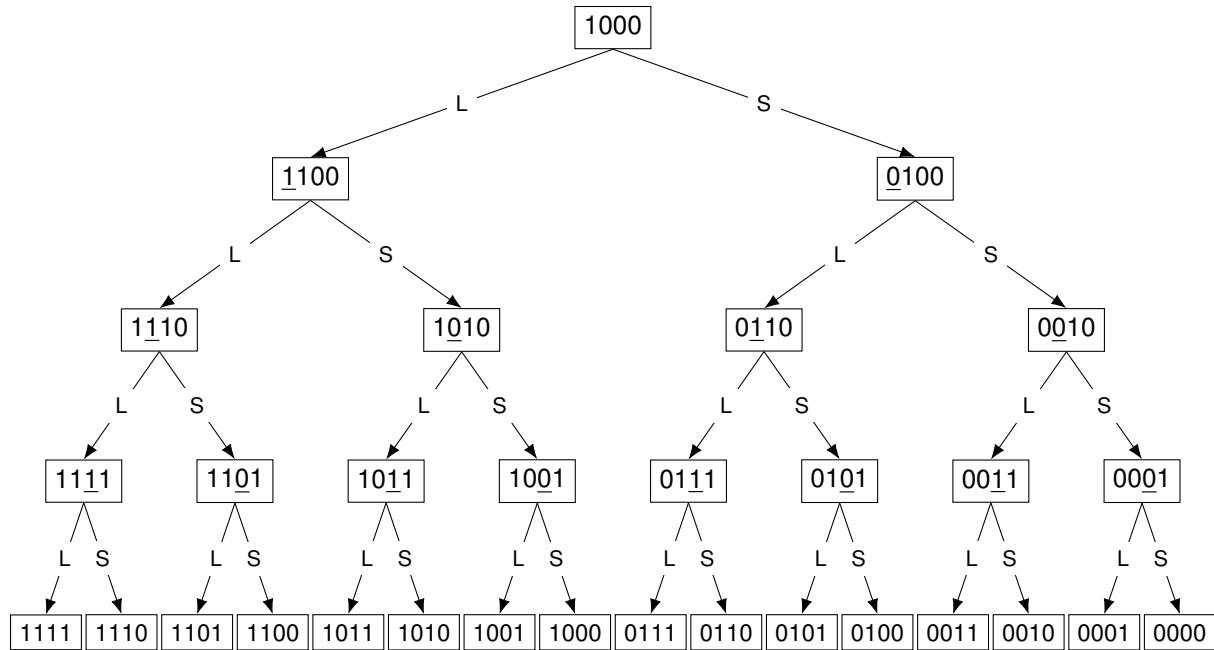


Figure 11.7: Process flow for a 4-bit successive approximation register analog to digital converter.

An SAR ADC requires a properly calibrated digital to analog converter (DAC), but only requires a single analog comparator, so the form factor is relatively small. It takes n clock cycles to compute a final result, where n is equal to the number of bits of system resolution. An SAR ADC consumes much less power than a flash ADC.³

Pipelined ADC

A pipelined ADC has q stages, where q is equal to the total device resolution divided by stage resolution. Each stage (which has m bits of resolution) uses an m -bit flash-type ADC to calculate m bits of the result at a time. For example, in Figure 11.8, a 12-bit pipelined ADC uses four 3-bit flash stages.

³ Maxim Integrated, "Understanding SAR ADCs: Their Architecture and Comparison with Other ADCs," October 2001.

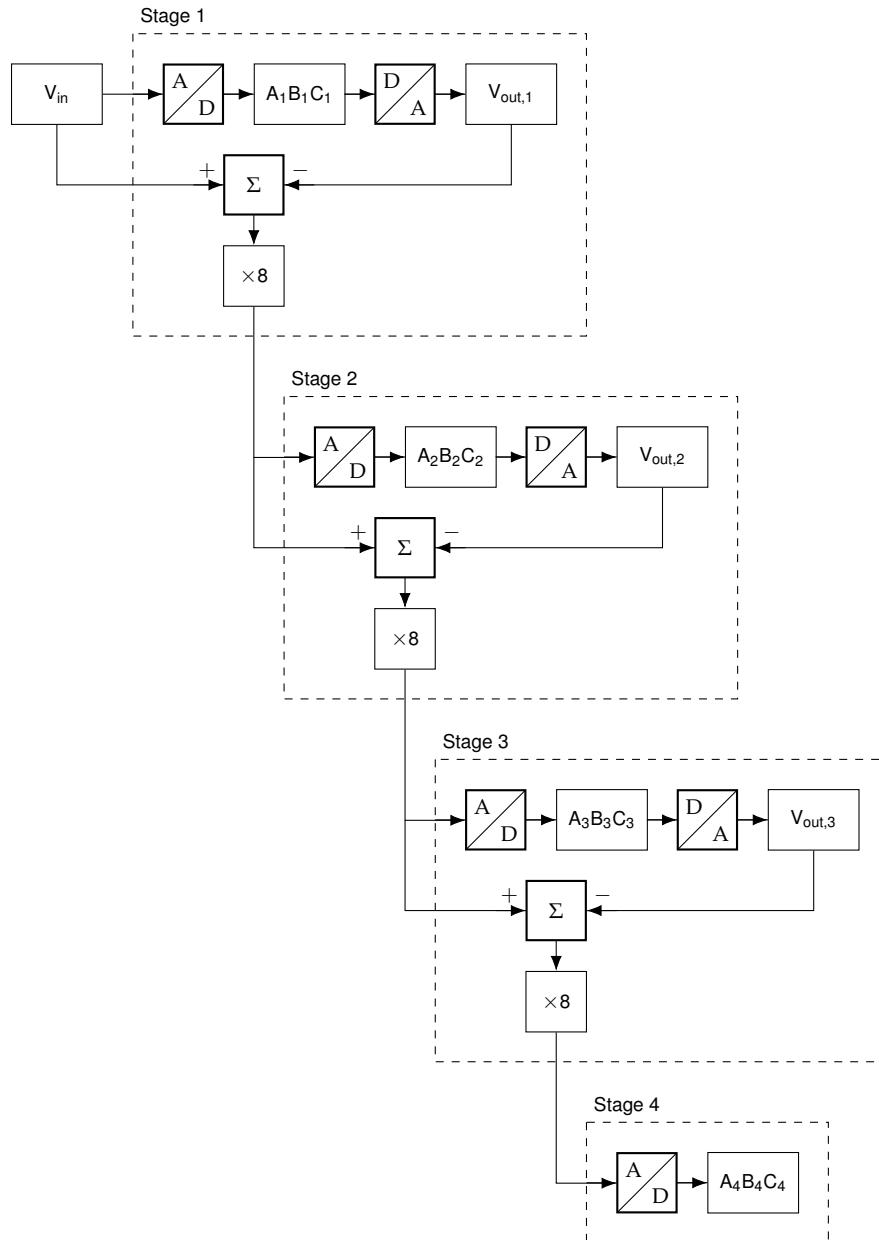


Figure 11.8: A 12-bit pipelined ADC with 3-bit flash ADC stage.

The first stage of the pipelined ADC calculates the first 3 most significant bits of the result. This is then subtracted from the input value, and then the next 3 significant bits are calculated. This process iterates until the final result has been calculated, giving an output of the form $\underline{A_1B_1C_1} \underline{A_2B_2C_2} \underline{A_3B_3C_3} \underline{A_4B_4C_4}$. In this manner, a pipelined ADC requires q clock cycles to compute the output.

11.5 The ADC on the ATmega328P

The ATmega328P contains a 10-bit successive approximation ADC. It is connected to an 8-channel analog multiplexer that selects from eight different input sources (including all of the pins in port C). The input source is selected by configuring the ADC multiplexer selection register (ADMUX). The minimum voltage level is 0 V (ground) and the reference voltage can be selected from different sources (either an externally applied reference voltage AREF, the value on pin AVcc, or 1.1 V). The voltage reference source is selected by configuring the ADMUX register. A block diagram of the ATmega328P ADC is shown in Figure 11.9.

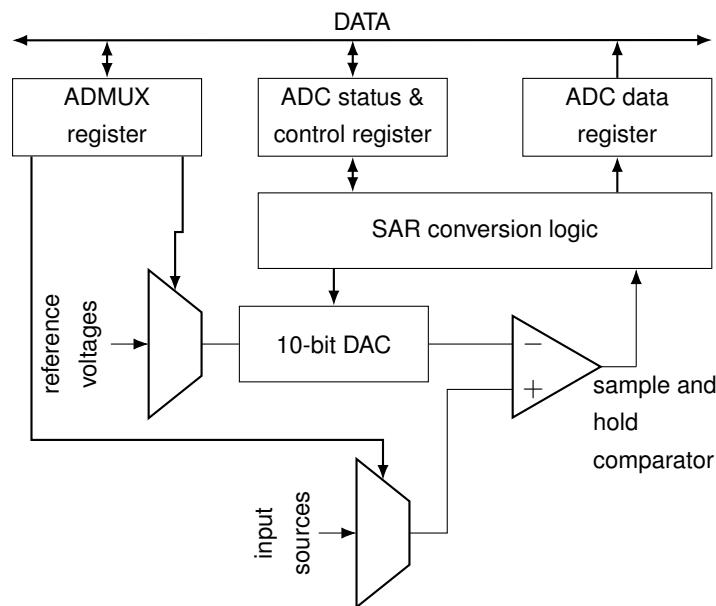


Figure 11.9: Analog to digital converter block diagram.

The number of available input sources depends on the package of the ATmega328P. The 28-pin chips do not contain pins for ADC6 and ADC7. However, the 32-pin chips do.

Data Triggering and Sampling

When using the ADC on the ATmega328P, the sample rate can be selected by configuring the trigger source for the ADC. If the ADC auto trigger enable (ADATE) bit is set in ADC control and status register A (ADCSRA), the ADC can be set to run continuously in free-running mode or can sample on particular trigger events given by the trigger source selected in ADC control and status register B (ADCSR_B).

With the exception of the first conversion, which requires 38.5 clock cycles to complete, auto triggered conversions require 15.5 clock cycles to complete. Other conversions require 14.5 clock cycles to complete. This includes the clock cycles required to sample and hold data as well as to convert the sampled data to a digital value and store it in the ADC data registers.

In contrast, single conversions are possible by clearing the ADATE bit. In this case, data can be sampled from the ADC on demand by setting the ADC start conversion (ADSC) bit in ADCSRA. This is **strongly recommended** if there is any need to change the input source pin using ADMUX. When auto triggering is enabled, updating the ADMUX register can lead to indeterministic timing of the ADC trigger, which can lead to unpredictable results.

When selecting between multiple ADC input sources, use single conversion mode by clearing ADATE. Write to ADSC when ready to make a conversion. After the conversion is complete, save the result and select the new input channel by reconfiguring ADMUX. Then, this cycle can be repeated.

The length of each clock cycle on the ADC depends on the frequency of the clock source as well as the configuration of something called a prescaler (both of which will be discussed in Chapter 14 of this textbook). The ADC prescaler can be set independently of other prescalers used for other peripheral features on the microcontroller.

Resolution

The ADC on the ATmega328P is 10 bits, meaning the output stored in the ADC data register(s) will be between 0–1023. The converted binary data is stored between two 8-bit registers ADCH and ADCL. When using the ADC in full-precision (10-bit) mode, the ADC left adjust result (ADLAR) bit in the ADMUX register is cleared, enabling the use of the variable ADC which accesses the full 10-bits of data.

When using the ADC in full-precision mode, an ADC frequency of 50–200 kHz is required. (Depending on the clock source used to operate the microcontroller, this will affect the prescaler used to operate the ADC.)

It is also possible to obtain lower-precision data from the ADC.

Simply store the most significant n bits of data to obtain an n -bit precision value! Typically, 8-bit precision is convenient as it lends itself nicely to the bit-width of the microcontroller in general. In this case, the ADLAR bit can be set, and the variable ADCH can be used to store the 8-bit result of the ADC.

Lower resolution can be obtained by setting ADLAR and bitshifting the ADCH register to the right as many bits as desired. (For example, bitshift right four times to obtain a 4-bit value.)

Resources

Additional information about the ADC used on the ATmega328P can be obtained in the datasheet, as well as in

- Microchip, Application Note AN2538, "AVR126: ADC of megaAVR in Single-Ended Mode," August 2017,
- Microchip, Application Note AN2447, "Measure VCC/Battery Voltage Without Using I/O Pin on tinyAVR and megaAVR," May 2017, and
- Microchip, Application Note AN2519, "AVR Microcontroller Hardware Design Considerations," February 2018.

11.6 Analog Comparator on the ATmega328P

As discussed above, an analog comparator is capable of comparing two analog voltages and either setting or clearing an output voltage based on the solution to the inequality given in Equation 11.6.

$$V_{out} = \begin{cases} V_{cc}, & \text{if } V_p \geq V_n \\ 0, & \text{otherwise} \end{cases} \quad (11.6)$$

On the ATmega328P, there is an analog comparator that can be used in this manner. The positive pin is AIN0 which is located on port D. The negative pin can be AIN1 (also located on port D), or selected from one of the ADC input channels, based on the values configured in ADCSRA and ADMUX. The value of the comparator output is saved in the Analog Comparator Control and Status Register (ACSR) in the Analog Comparator Output (AC0) bit.

The analog comparator can be used to trigger interrupts when the output of the analog comparator toggles, has a falling edge, or has a rising edge.

When using analog values on pins AIN0 and/or AIN1, the digital input buffer on whichever of those pins is in use should be disabled. This can be accomplished by setting the corresponding bit in the Digital Input Disable Register 1 (DIDR1) register.

11.7 Digital to Analog Conversion

While the ATmega328P does not contain a standalone digital to analog converter (DAC) module to provide analog voltage levels, it is useful to understand the concept of digital to analog conversion. A DAC performs the reverse function of an ADC, and as indicated previously in this chapter, is a crucial component included in SAR and pipelined ADC architectures. The analog voltage output of a DAC is given by Equation 11.7, where n is equal to the resolution of the converter.

$$V_{OUT} = V_{cc} \times \left(\frac{\text{value}}{2^n} \right) \quad (11.7)$$

In a DAC circuit, a digital datastream can be sampled and held (or sampled continuously), in which case the digital value is then converted into an analog voltage level. An R-2R ladder is one of many common architectures used to convert a binary value into a voltage, and is depicted schematically in Figure 11.10.

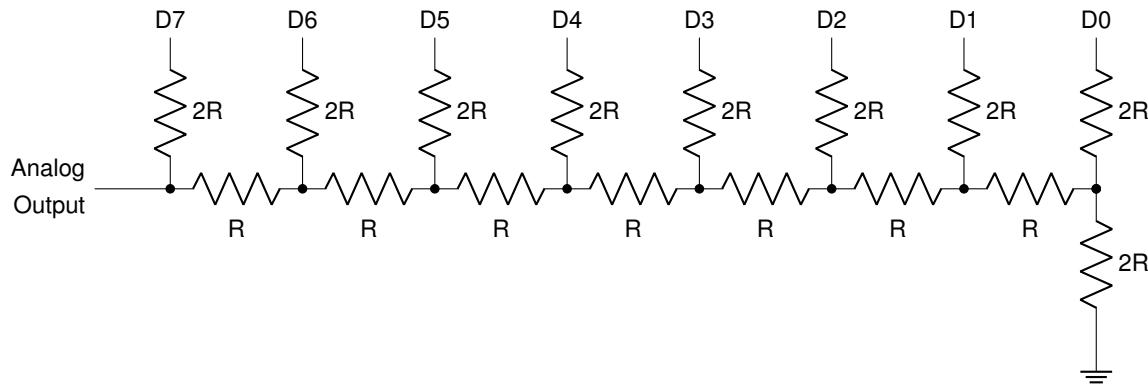


Figure 11.10: Schematic of a 3-bit R-2R digital to analog converter.

11.8 Practice Problems

1. Find the step size for an 8-bit ADC, if $V_{ref} = 1.28$ V. $\Delta V = 5$ mV
2. Given the situation in Question 1, calculate the output if the analog input is 0.7 V. $1000\ 1100$
3. Given the situation in Question 1, calculate the output if the analog input is 1.0 V. $1100\ 1000$
4. How many bits of resolution does the ATmega328P microcontroller have? 10 bits

5. True or false: The output of most sensors is analog. TRUE
6. Calculate the step size for the following ADCs, given $V_{ref} = 5$ V.
- (a) 8-bit ADC $\Delta V = 19.5$ mV
 - (b) 10-bit ADC $\Delta V = 4.9$ mV
 - (c) 12-bit ADC $\Delta V = 1.2$ mV
 - (d) 16-bit ADC $\Delta V = 15.3$ μ V
7. Given a V_{ref} of 2.56 V, find the corresponding V_{in} for each of the 8-bit ADC outputs.
- (a) 1111 1111 $V_{in} = 2.55$ V
 - (b) 1001 1001 $V_{in} = 1.53$ V
 - (c) 0110 1100 $V_{in} = 1.08$ V

12

Sensors and Sensor Calibration

IN ORDER TO OBTAIN INFORMATION about the world around us, sensors must be used. Sensors take information about the external environment and convert (transduce) them into electrical signals. Possible electrical signals include voltage, current, and resistance. Devices with electrical properties dependent on a particular physical quantity are chosen (photoresistors, temperature-sensitive transistors, piezoelectric ceramics) to measure that property. Figure 12.1 shows example block diagrams of this property in a temperature sensor (such as the TMP36) and a photoresistor.

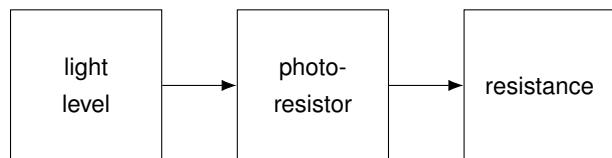
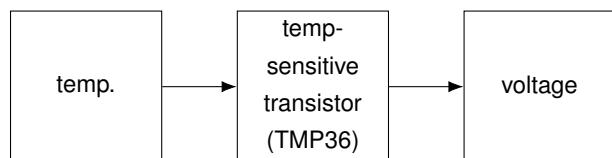


Figure 12.1: Diagram of how the TMP36 temperature sensor transduces temperature to voltage, as well as how a photoresistor transduces light level to resistance.

Several types of sensors are listed in Table 12.1.

Table 12.1: Various types of sensors.

Light Level Sensors

- Photoresistor
 - Photodiode
 - Phototransistor
-

Distance, Speed, Acceleration Sensors

Ultrasonic detector

Accelerometer

Wheel encoder/tachometer

Sound Intensity

Microphone

Digital Sensors

Switches

Toggles

Pushbuttons

Environmental Conditions

Temperature sensor

Humidity sensor

Barometer

Some sensors provide digital data output. For example, a pushbutton is capable of generating a digital logic value of HIGH or LOW depending on whether or not the button is pressed. Simple digital sensors such as this can be directly connected to any pin on the ATmega328P that has been configured as an input pin.

Of sensors that provide digital data output, some have been built to send data using a serial communication protocol. Devices that make use of USART, SPI, or TWI can interface with the ATmega328P using serial I/O, as is discussed in [chapter 15](#). Making sense of this data will require consulting the sensor datasheet.

Many sensors, however, provide analog information, which may be in the form of a voltage output or a changing resistance. Sensors that output a voltage can interface directly with the ATmega328P using an analog pin by confirming that the minimum and maximum voltage output values fall within the range of 0 V to Vcc.

Sensors that output a resistance must first be configured with another resistor in series (R_C) in order to convert the variable resistance into an analog voltage value, as shown in Figure [12.2](#). By connecting this resistor in series with the sensor, a voltage divider is created, causing the value of V_{OUT} to take on a value between 0 V and Vcc, depending on the value of the sensor resistance.

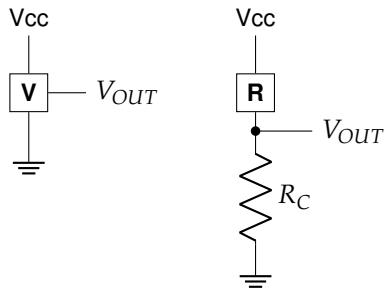


Figure 12.2: Schematics of how to connect sensors that output voltage (V) and resistance (R) to the ADC on the ATmega328P.

12.1 Choosing Resistor Values

With sensors that output variable resistance values, it is important to correctly choose the value of R_C in order to obtain an appropriate output value. The relationship between output voltage, V_{cc} , and both resistors in Figure 12.2 is given by Equation 12.1.

$$V_{OUT} = V_{cc} \times \left(\frac{R_C}{R_C + R} \right) \quad (12.1)$$

The output voltage will be maximum (V_{MAX}) when the sensor resistance (R) is lowest, and it will be minimum (V_{MIN}) when the sensor resistance is greatest. A value of R_C must be chosen such that the contrast of the circuit, defined in Equation 12.2, is greatest.

$$contrast = V_{MAX} - V_{MIN} \quad (12.2)$$

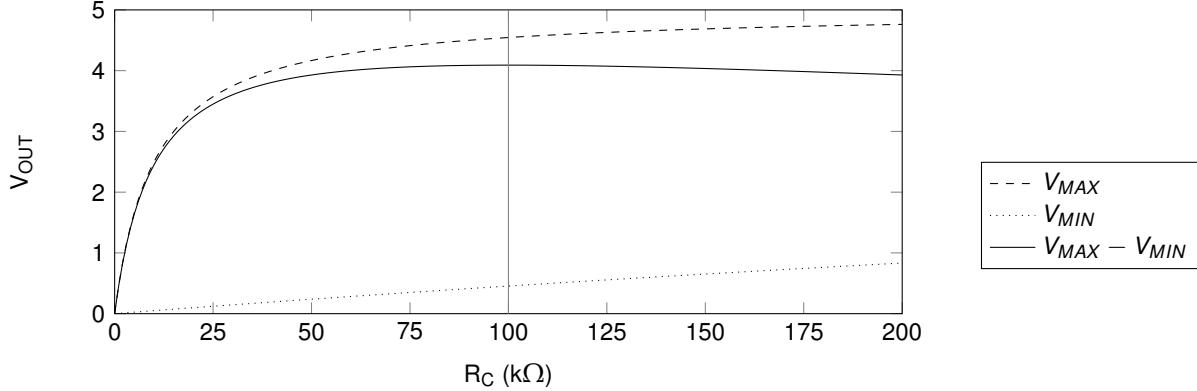
The value of R_C that leads to the greatest contrast can be determined empirically by determining the minimum and maximum sensor resistance values. Using calculus, the first derivative of contrast with respect to R_C can be calculated, set equal to zero, and solved for R_C . This will be the value of R_C that leads to the most contrast. For an alternative solving method that does not require calculus, use a spreadsheet program such as Excel to plot the values of V_{MAX} and V_{MIN} and their difference for many versions of R_C , and use the MAX() function to determine the value of R_C that leads to the maximum contrast.

In the specific case of the photoresistor, the maximum resistance in the device corresponds to when it is dark ($R_{DARK} = 1 \text{ M}\Omega$), and the minimum resistance in the device corresponds to when it is light ($R_{LIGHT} = 10 \text{ k}\Omega$). Using Equation 12.1, V_{MAX} is derived as a function of R_C in Equation 12.3 and V_{MIN} is derived as a function of R_C in Equation 12.4.

$$V_{MAX} = V_{cc} \times \left(\frac{R_C}{R_C + 10k\Omega} \right) \quad (12.3)$$

$$V_{MIN} = V_{cc} \times \left(\frac{R_C}{R_C + 1M\Omega} \right) \quad (12.4)$$

These two functions, and their difference (the contrast), have been plotted as a function of R_C in Figure 12.3. From this graph, the value of R_C that leads to the maximum contrast is equal to $100 k\Omega$.



After determining the value of R_C , it is next imperative to ensure that the power rating of this resistor is sufficient. The power consumed by this resistor is given by Equation 12.5.

$$P = V_{cc}^2 \times \left(\frac{R_C}{(R + R_C)^2} \right) \quad (12.5)$$

This power will be maximized when the sensor resistance is at a minimum value, so that value should be used to determine the maximum power consumption of the resistor R_C .

It's also important to ensure that the power consumed by the sensor itself will be within the maximum value allowed by the datasheet specifications. The power consumed by the sensor is given by Equation 12.6. This will also be at a maximum when the sensor resistance is lowest. If this value is out of spec, it may be necessary to change the value of R_C to get it into an allowable range. (This is more important than contrast, so be sure to prioritize power consumption as needed. If the contrast becomes too small, it may be necessary to buy a sensor that is capable of handling a larger power load.)

$$P = V_{cc}^2 \times \left(\frac{R}{(R + R_C)^2} \right) \quad (12.6)$$

Returning to the example of the photoresistor, the maximum power consumed by the $100 k\Omega$ resistor is 0.2 mW , which means

Figure 12.3: The maximum and minimum voltages and contrast as a function of the value of R_C .

If you are interested in the derivation of this equation, I suggest you read the first two chapters of the Circuit Analysis textbook that I have authored, particularly the sections on power and the voltage divider rule.

that a $1/4$ watt resistor (which is quite standard) will be more than sufficient to ensure that the resistor R_C will not overheat and melt.

12.2 Sensor Calibration

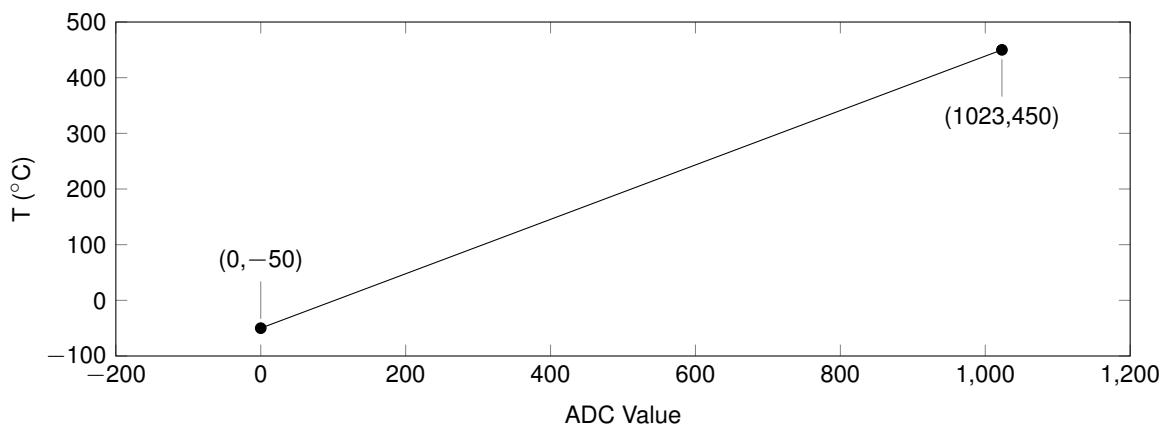
In order to obtain meaningful information from sensors, they must be characterized and calibrated. In other words, it is critical to know how to convert the output voltage into the physical quantity that it is meant to represent. This can be accomplished by checking the sensor datasheet or by performing a manual calibration.

Datasheet Calibration

Some sensors have datasheets which explain how the output corresponds to the physical quantity. For example, the TMP36 temperature sensor datasheet states that the output has a slope of $10 \text{ mV}/^\circ\text{C}$ and that an output voltage of 750 mV indicates a temperature of 25°C . From this, the equation between temperature (T , measured in $^\circ\text{C}$) and output voltage (V_{OUT} , measured in V) can be derived as given in Equation 12.7.

$$T = 100V_{OUT} - 50 \quad (12.7)$$

As discussed in chapter 11, the ATmega328P has a 10-bit analog to digital converter (ADC), which outputs a value of 0 when the input voltage is 0 V, and a value of 1023 when the input voltage is 5 V. Therefore an equation can be derived directly between the value of the ADC and the temperature. This data is depicted graphically in Figure 12.4.



Deriving an equation for the slope of the line connecting the two datapoints yields a relationship between the value output by the

Figure 12.4: Linear relationship between temperature and ADC value of the TMP36 temperature sensor, given by the TMP36 datasheet.

ADC and the temperature. This is what will be used to calculate the temperature using the ADC value provided by the microcontroller. This equation is given for the TMP36 sensor in Equation 12.8.

$$T = 0.489ADC - 50 \quad (12.8)$$

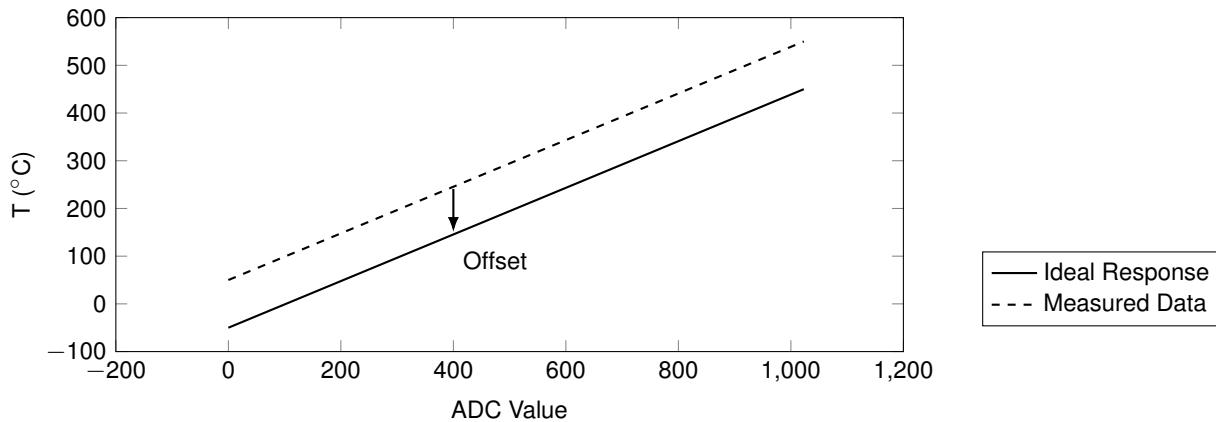
This can be programmed onto the ATmega328P, avoiding the use of floating-point math, using the equation

$$T = 500L*ADC>>n - 50,$$

where n is the resolution of the ADC. (This makes it easy to port code between an ADC run in 8-bit mode and an ADC run in 10-bit mode, for example.)

One-Point Calibration

When put into practice, the sensor may not give the output that was expected. If there is a discrepancy between the output of the sensor and that of a trustworthy reference, the data must be further calibrated. One-point calibration is used when the slope is correct but there's an offset between measured and actual data (as measured by a trustworthy reference). If the measured data is greater than the actual data, the offset must be subtracted, and if the measured data is less than the actual data, the offset must be added. This is shown graphically in Figure 12.5.



Multiple-Point Calibration

If the output response of a sensor is not known, it must be found using a reference instrument to measure the physical quantity and compare it to the ADC values output by the sensor. Use as many

Figure 12.5: Example of one-point calibration; finding an offset value changes the measured response to the ideal (expected) result.

known reference points as possible, and use plotting software such as Excel to find a best fit line, keeping in mind that the best fit line may not always be linear. This best fit line will then be used in software to convert the ADC value into the corresponding physical quantity. A hypothetical example is shown in Figure 12.6 with a linear relationship found between sound intensity and ADC values.

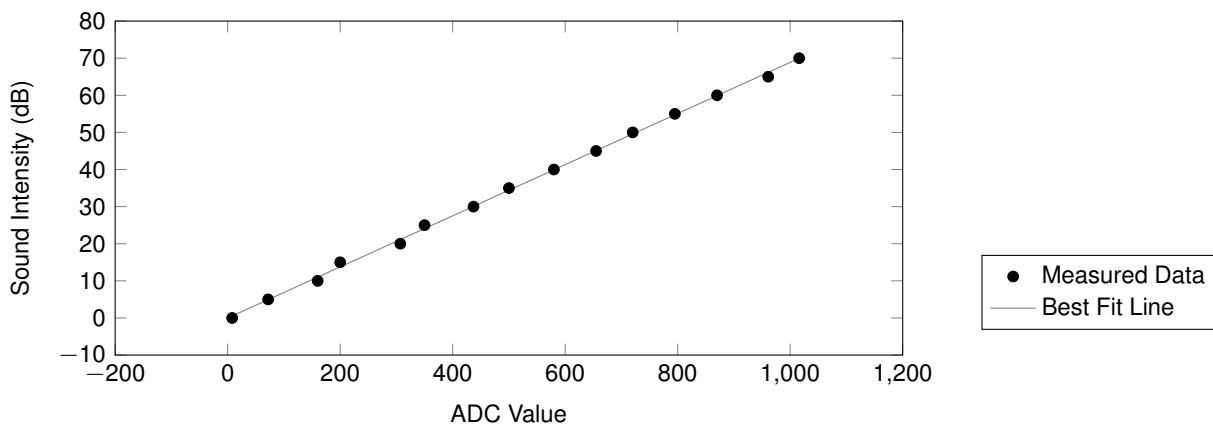


Figure 12.6: Example of multiple point calibration to determine the ADC value given at different sound intensity levels.

Nonlinear Data

When data is nonlinear, there are a couple options to consider for calibrating the sensor.

If the sensor is nonlinear throughout its entire range, but will be essentially linear throughout the expected operating range, then it may be acceptable to treat the output as a linear function of the ADC value. (Do this with caution: if the sensor is operated outside of the expected range, the output values will be invalid!)

If there is plenty of processing power and program memory (but perhaps not much data memory), then a best-fit equation can be derived. The non-linear math can be handled using trustworthy subroutines.

If instead there is plenty of data memory (but perhaps not much processing power or program memory), a lookup table of output values can be stored in data memory as an array based on the ADC values. For example, when using the ADC in 8-bit mode, a table of 256 output values can be stored in data memory, one corresponding to each output given an ADC value from 0–255.

12.3 Mitigating Fluctuating Data and Sensor Noise

Sensor data can fluctuate over time due to environmental conditions or noise. It may be important to perform a rolling average to obtain

a steady, reliable readout. A rolling average uses n previous values (stored in an array) to calculate the current average value. Care must be taken in choosing an appropriate value for n , which must be specifically tailored to each different sensor and situation in which it is to be used. Table 12.2 explains what happens if n is small or large.

If n is small...

- Program uses less memory
 - Program takes less time to initialize
 - Sensor is more susceptible to noise
-

If n is large...

- Program takes more memory
 - Program takes more time to initialize
 - Sensor is less susceptible to noise
 - If n is too large, the sensor is not sensitive to short-term or quick changes
-

Table 12.2: Issues that can result if n is chosen to be either small or large..

Sensor data from a temperature sensor that has been averaged out using three different values of n (as well as the raw data, which has an n value of 1) is shown in Figure 12.7.

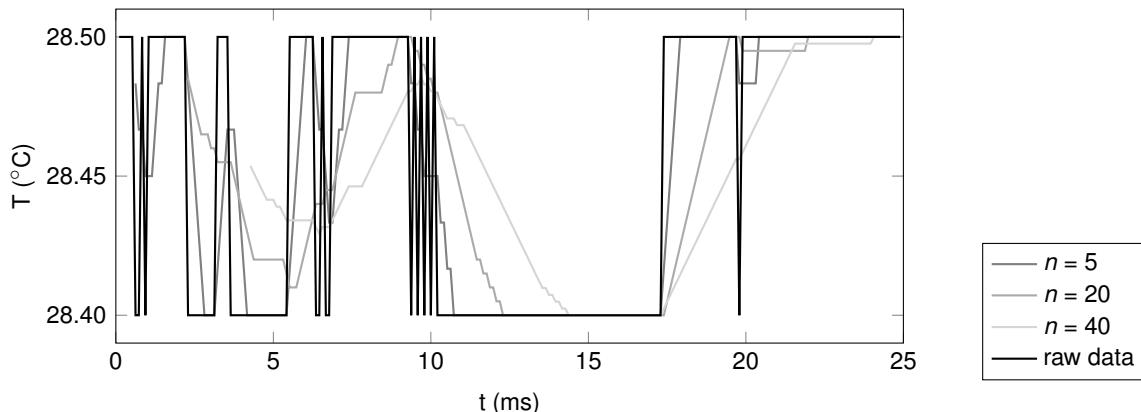


Figure 12.7: Temperature data subjected to rolling average with different values of n .

Circular Buffer

A circular buffer is a method of taking a rolling average to clean up sensor noise and fluctuations. It is simply an array of n sensor values, where the first value in is also the first value out. (This type of situation is known as FIFO: first in first out.) A flowchart and sample array is shown in Figure 12.8 to explain this process.

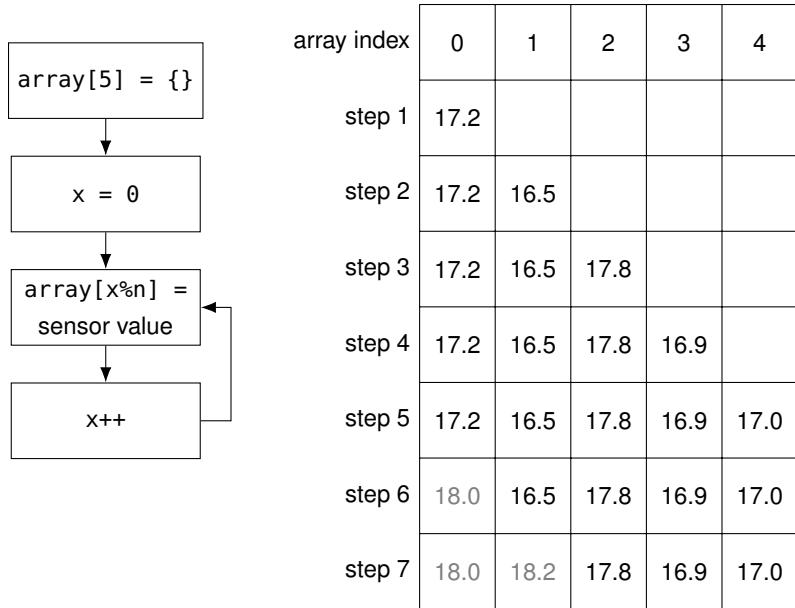


Figure 12.8: Flow chart and example array of values in a circular buffer.

The data in the circular buffer can be passed to a subroutine and averaged to determine the average over the given timespan.

Timed Updates

Instead of retrieving continuous updates from the sensor, the sensor value can be updated only at finite, regularly spaced intervals of time. This can be accomplished using timed interrupts. The concept of interrupts is explained in Chapter 13, and the timer/counter units are explained in Chapter 14.

For maximum noise filtering, this method can be combined with the circular buffer method.

12.4 ATmega328P Internal Temperature Sensor

The ATmega328P contains an internal temperature sensor whose output is connected to one of the input channels on the ADC Multiplexer Selection Register (ADMUX). When using this input channel, the internal 1.1 V reference voltage must also be selected in ADMUX.

When using the ADC in full-precision (10-bit) mode, the relationship between temperature (T , measured in $^{\circ}\text{C}$) and the ADC value is given in Equation 12.9.

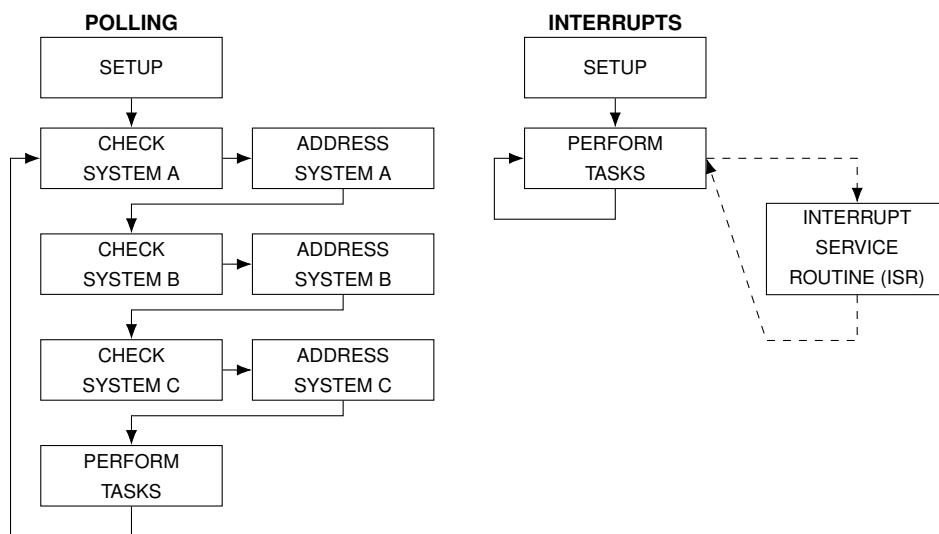
$$T = \frac{781 \times \text{ADC}}{1000} - 250 \quad (12.9)$$

13

Interrupts

AN INTERRUPT IS AN IMPORTANT EVENT that requires immediate attention. For example, in monitoring an oil refinery, many sensors may be checked in sequence and cycled through (a system known as **polling**). If a fire breaks out, it would not be prudent to wait until the fire sensor is checked to know about it. It would also not be wise to only check the fire sensor, or to check it excessively, because the program must do other things as well. Instead the fire sensor can generate an **interrupt request** if a fire is detected. This would cause the software to deal with the immediate issue, and then return to regular operation when ready.

An example program flow that compares continuous polling to interrupts is shown in Figure 13.1.



Some interrupts are synchronous (for example, interrupts can be configured to operate at particular time intervals using the timer/counter unit, as will be discussed in Chapter 14). Other interrupts are asyn-

Figure 13.1: Program flow of software that uses continuous polling to monitor systems, vs. software that uses interrupts to asynchronously handle important events.

chronous and can correspond to particular values on I/O pins or status updates from peripheral features.

When an interrupt request is set, the current value in the program counter is stored, and the code jumps to handle the corresponding interrupt service routine (ISR). Once the ISR has concluded, the old program counter location is retrieved, and the code jumps back to its initial location and continues executing code.

It's important to note that the value of SREG is not necessarily automatically stored when an ISR is invoked by an interrupt request. This means that any critical functionality of the microcontroller (for example, performing conditional and/or Boolean logic to determine a branch using control flow) will be affected. Either the value of SREG must be stored upon executing an ISR and retrieved upon concluding an ISR, or core functionality must be protected by masking interrupts.

13.1 Enabling and Disabling (Masking) Interrupts

At times, it is useful to enable or disable interrupts. This can happen globally, in that **all** interrupts are enabled/disabled, or locally, in which only individual interrupts are enabled/disabled. When the ATmega328P resets, all interrupts are disabled to allow the device to reset without interrupt. Interrupts can be globally disabled by clearing the interrupt flag in SREG ($I=0$). Interrupts can be globally enabled by setting the interrupt flag in SREG ($I=1$). Interrupts are globally disabled upon entering an ISR, and then globally enabled once the ISR has been serviced. **Nonmaskable** interrupts cannot be disabled. The reset interrupt is an example of a nonmaskable interrupt.

Locally enabling or disabling individual interrupts requires configuration in the corresponding configuration or setup register. For example, enabling the ADC interrupt requires setting the ADC Interrupt Enable (ADIE) bit in the ADC Control and Status Register A (ADCSRA).

13.2 Interrupt Service Routine (ISR)

An interrupt service routine (ISR) is a subroutine that the microcontroller executes when an interrupt is invoked. ISRs should be written as efficiently as possible, so as to deal with the asynchronous event without detracting from the functionality of the program in general. Because an ISR is never formally invoked in software (it is only triggered upon a specific event), the function cannot be passed any parameters and cannot return any variables. Any variable that

must be shared with an ISR must be global and protected with the **volatile** keyword.

When the interrupt is invoked, the program jumps to the memory location of the associated **interrupt vector**. Each of these vectors can have an associated subroutine written for it. The group of memory locations that holds the ISR addresses is called the **interrupt vector table**. The ATmega328P interrupt vector table is given in Table 13.1. The lower the address of the ISR location, the higher its **priority**. If two interrupts are triggered simultaneously, the one with the highest priority will be serviced first, followed by the second.

Table 13.1: ATmega328P interrupt vector table.

Vector	Address	Source	Interrupt Definition
1	0x0000	RESET	External pin, power-on reset
2	0x0002	INT0	External interrupt request 0
3	0x0004	INT1	External interrupt request 1
4	0x0006	PCINT0	Pin change interrupt request 0
5	0x0008	PCINT1	Pin change interrupt request 1
6	0x000A	PCINT2	Pin change interrupt request 2
7	0x000C	WDT	Watchdog time-out interrupt
8	0x000E	TIMER2 COMPA	Timer/counter 2 compare match A
9	0x0010	TIMER2 COMPB	Timer/counter 2 compare match B
10	0x0012	TIMER2 OVF	Timer/counter 2 overflow
11	0x0014	TIMER1 CAPT	Timer/counter 1 capture event
12	0x0016	TIMER1 COMPA	Timer/counter 1 compare match A
13	0x0018	TIMER1 COMPB	Timer/counter 1 compare match B
14	0x001A	TIMER1 OVF	Timer/counter 1 overflow
15	0x001C	TIMER0 COMPA	Timer/counter 0 compare match A
16	0x001E	TIMER0 COMPB	Timer/counter 0 compare match B
17	0x0020	TIMER0 OVF	Timer/counter 0 overflow
18	0x0022	SPI STC	SPI serial transfer complete
19	0x0024	USART RX	USART Rx complete
20	0x0026	USART UDRE	USART data register empty
21	0x0028	USART TX	USART Tx complete
22	0x002A	ADC	ADC conversion complete
23	0x002C	EE READY	EEPROM ready
24	0x002E	ANALOG COMP	Analog comparator
25	0x0030	TWI	2-wire serial interface

When a particular interrupt is enabled, and interrupts are globally enabled (I in SREG is set), as soon as the ISR is invoked, the address associated with that interrupt vector will be loaded into the program counter. If the ISR is not defined in code, then the program will jump to an invalid memory location and cause the software to act unpredictably. For this reason, when using interrupts, it is extremely important to ensure that each enabled ISR is defined!

ISR Execution

When an ISR is invoked, the microcontroller completes the following operations.

1. Finishes executing the current instruction
2. Saves the address of the next instruction in the stack
3. Jumps to interrupt vector table
4. Loads memory address of the associated ISR into the program counter
5. Clears global interrupt enable flag in SREG (I=0)
6. Runs the ISR subroutine until its end (RETI = return from interrupt instruction)
7. Sets global interrupt enable flag in SREG (I=1)
8. Loads the memory address to return to from the stack into the program counter
9. Executes the next instruction
10. Either
 - (a) Continues fetch/execute as normal
 - (b) Services the next interrupt (if there is one)

13.3 Interrupts on the ATmega328P

As listed in Table 13.1, there are many available interrupt sources on the ATmega328P microcontroller. This chapter will consider the reset, external, and pin-change interrupts. Other chapters in this textbook will deal with their appropriate interrupts as needed.

Interrupt Flags

Interrupt flags are used to trigger an ISR. They are stored in a register and set by hardware once the interrupt condition is met. If I in SREG is set (interrupts are globally enabled) and an interrupt flag is set, that ISR will be invoked. An interrupt flag is cleared by hardware when executing the ISR. If the flag is being used but an ISR is not needed (this is useful when triggering the ADC, for example), then the interrupt flag can be cleared by writing a one to the flag bit.

External Interrupts

External interrupts are triggered by the INT0 and INT1 pins on the microcontroller. Each of the individual interrupts can be configured to trigger on a

- LOW signal,
- toggle,
- falling edge, or
- rising edge.

The utility of these two interrupts is that each individual pin can be uniquely triggered, and has its own corresponding ISR. This means that the exact status of the pin is known at the moment the ISR is invoked. However, these interrupts are only available on two of the approximately 20 I/O pins on the microcontroller.

Pin Change Interrupts

Pin change interrupts are available on all I/O pins on the ATmega328P. The pin change interrupt for each I/O pin can be individually enabled in the I/O port's Pin Change Mask Register (PCMSKn). As there are three I/O ports on the ATmega328P, there are three mask registers. Pin change interrupts must also be enabled on the port level using the Pin Change Interrupt Control Register (PCICR).

When enabled, a pin change interrupt flag will be set when a toggle condition is present on any enabled I/O pin.

The benefit of pin change interrupts is that they are available on all of the I/O pins on the microcontroller. However, because each port has its own ISR (instead of each pin), if more than one pin interrupt is enabled on a port, conditional logic may be required to determine which of the pins triggered the interrupt. Also, because pin change interrupts are invoked on a pin toggle, if a particular logic level is desired, conditional logic will be required to determine if that logic level has been obtained.

Reset Interrupt

Reset is a special category of interrupt on the ATmega328P. It is the highest priority ISR, and cannot be disabled. When the ATmega328P is first powered on, initial values of the program counter, flip-flops, I/O control registers, etc. are unknown. The reset ISR sets these critical registers to initial values upon reset. The program counter is set to `0x0000` upon reset (shown in Figure 13.2), therefore it begins by servicing the reset subroutine before carrying out the rest of the program. There are several reset sources on the ATmega328P microcontroller. One of them occurs when the microcontroller is first powered on. An external button on the Arduino is connected to the `RESET` pin on the ATmega328P, which creates a reset condition when the pin is held low for longer than a specified minimum amount of time. In addition, brown-out resets and watchdog resets can be enabled to trigger the reset condition.

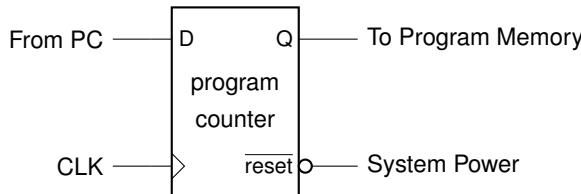


Figure 13.2: The program counter is initialized to a value of `0x0000` when system power is shut off.

13.4 Practice Problems

1. Which of the two techniques (interrupts or polling) requires more system resource usage on the microcontroller?
polling
2. What is the memory address of the watchdog timer interrupt?
`0x000C`
3. True or false: While servicing an interrupt service routine, interrupts are globally enabled.
FALSE
4. Is the reset interrupt maskable or nonmaskable?
nonmaskable
5. When two interrupts are invoked simultaneously, the one with the _____ memory address will be serviced first.
lower
6. If `INT0` and `INT1` are invoked simultaneously, which will be serviced first?
`INT0`

14

Clocks, Timer/Counters and Pulse-Width Modulation

A MICROCONTROLLER NEEDS A CLOCK to keep time for all of its hardware components and peripherals. A clock signal is a square wave that oscillates between 0 V and Vcc at regular intervals. A clock is also useful to create time delays in code (for example, to blink an LED or to check sensors at regular intervals).

The frequency (f) of a clock describes the number of oscillations it makes every second and is measured in units of Hz. (For a square wave clock signal, an oscillation corresponds to subsequent rising edges of the signal or to subsequent falling edges of the signal.) The period (T) of the wave describes the number of seconds it takes to complete one oscillation. These two quantities are inversely related to each other given by Equation 14.1.

$$T = \frac{1}{f} \quad (14.1)$$

A list of different technologies that can be used to generate clock sources is provided in Table 14.1.

Table 14.1: A list of various clock sources.

Ceramic Resonators

- Piezoelectric ceramic material
 - Internal vibrations create an oscillating voltage at a specific frequency
 - Not accurate enough to be used for a CPU clock (0.5% tolerance)
-

Crystal Oscillators

- Piezoelectric crystal
- Internal vibrations create an oscillating voltage at a specific frequency

- Highly accurate and can be used for system clocks (0.001% tolerance)
-

RC Circuits

- Amplifiers
 - 555 timers
 - RLC circuits
-

Not all of these technologies directly generate a clock signal. Some of them (such as an RLC circuit) generate a sinusoidal wave and some (such as piezoelectric crystals) may not generate the proper voltage levels necessary for a digital clock. Microcontrollers contain circuitry to condition an input oscillation and convert it into a proper clock signal. An understanding of a microcontroller's capabilities is important before selecting a clock source. It's possible that external capacitors or other conditioning circuitry may be required to obtain a proper clock signal.¹

Microcontrollers also have the capability to change the clock frequency. **Clock multipliers** increase the frequency by using a phase-locked loop. The advantage of this is to have faster code execution. However, all microcontrollers have maximum frequencies beyond which they will not work reliably, which must be taken into consideration before multiplying the frequency of any clock signal. Increasing the clock frequency also has the side effect of increasing the power consumption of the microcontroller.

Clock dividers (prescalers) decrease the frequency. They use a timer to count to a particular prescaler value (2, 8, 1024, etc) and when the timer reaches that value, an overflow causes a pin to toggle. That toggle becomes the new clock signal. Slower clocks indicate a longer time to execute code but use less power.

¹ Microchip, "AVR Microcontroller Hardware Design Considerations," February 2018.

14.1 ATmega328P Clock

The ATmega328P has two internal clocks: an 8 MHz RC oscillator and a 128 kHz lower power oscillator. An external clock (connected to the XTAL1 pin), or a crystal or ceramic oscillator (connected to the XTAL1 and XTAL2 pins) may be used instead of one of the internal oscillators. The clock source is selected by changing fuse bits in the low fuse byte. The ATmega328P is rated for a fastest clock speed of 16 MHz. It has no clock multiplier, but is capable of using a global prescaler to divide the clock frequency. (In fact, the factory default setting of an ATmega328P microcontroller is to use the 8 MHz oscillator with a prescaler of 8, leading to a system clock frequency of 1 MHz.)

The clock control unit, as shown in Figure 14.1, routes the clock signals to all device peripherals, as different peripherals require their own clock signals.

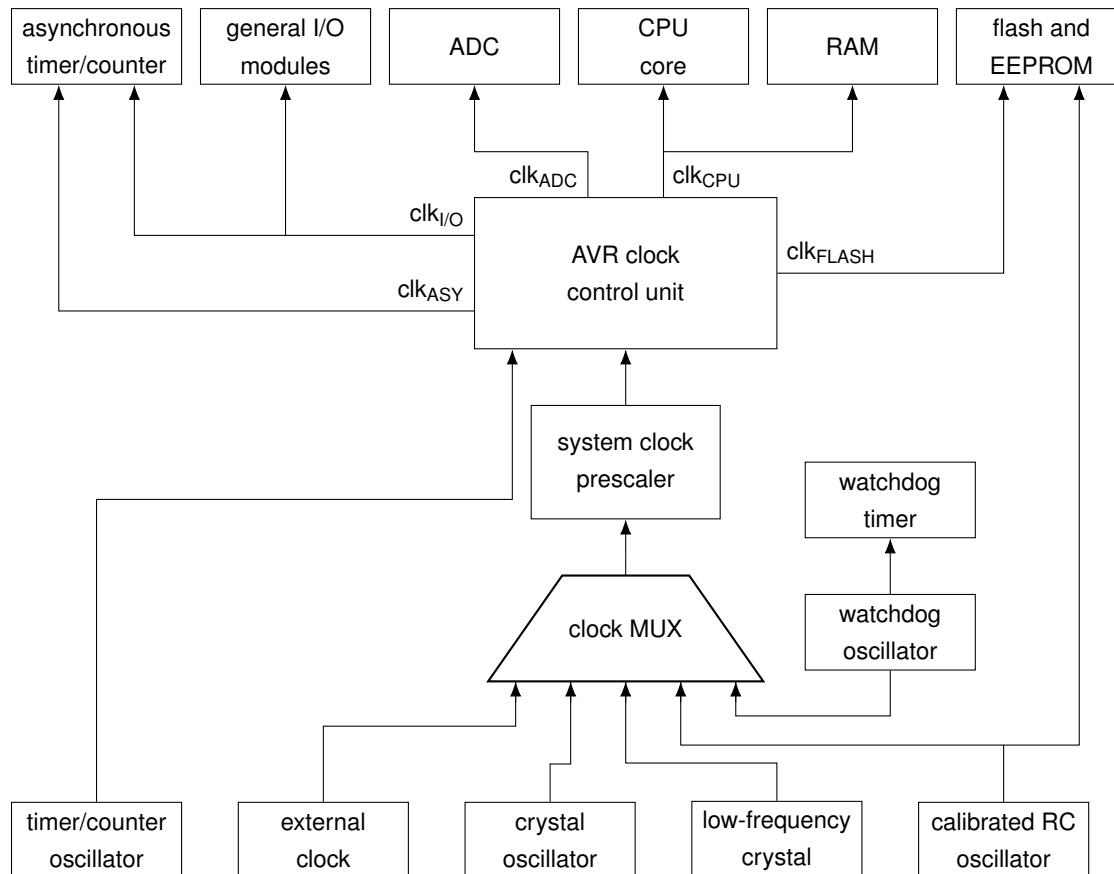


Figure 14.1: Schematic of the AVR clock distribution system.

Each peripheral clock is described below.

- The CPU clock (CLK_{CPU}) is routed to all parts involved with core operations.
- The I/O clock ($\text{CLK}_{\text{I/O}}$) is routed to all parts involved with input and output operations, including timer/counters, serial communication peripherals, and external interrupts.
- The flash clock ($\text{CLK}_{\text{FLASH}}$) is routed to the program memory. When externally programming the chip, it is important to have access to program memory without the CPU running.
- The asynchronous clock (CLK_{ASY}) allows the asynchronous timer/counter to be clocked directly from an external source.
- The ADC clock (CLK_{ADC}) is used to power the ADC independently of the CPU clock. The ADC can be run with the CPU clock

off to reduce noise and increase the precision of the analog to digital conversion process.

14.2 ATmega328P Timer/Counters

Timer/counters are integral in the use of timed interrupts and generation of square waves. A counter counts up values from $0 \rightarrow 2^n - 1$, where n is the resolution of the counter. A timer is simply a counter that is clocked with the CPU clock. Timer/counters are used to generate waveforms (for example in pulse-width modulation), measure time intervals, generate interrupts at specific intervals, and capture or count external events.

There are three timer/counters on the ATmega328P.

- Timer/counter 0 – 8 bits of resolution
- Timer/counter 1 – 16 bits of resolution
- Timer/counter 2 – 8 bits of resolution

Each counter can be independently configured and can also be assigned its own prescaler value to allow for operation at independent frequencies.²

² Atmel, "AVR130: Setup and Use of AVR Timers," March 2016.

Clock Sources

There are different clock sources that can be used for each timer/counter to provide flexibility in timing. The first option, available on all three timer/counters, is to use the CPU clock. It is possible to use a prescaler to slow down the timing intervals, however there are a finite number of prescalers available to use.

The second option, available on timer/counters 0 and 1, is to use an external clock. The microcontroller then synchronizes the external clock to the system clock. This limits the range of allowable frequencies of the external clock. Because it is sampled by the CPU clock, it must be slower than half the frequency of the CPU clock. (In fact, the ATmega328P datasheet recommends that the maximum external clock frequency be no greater than the CPU clock divided by 2.5.) The external clock frequency cannot be prescaled. While an external clock is being used, due to the synchronization of the CPU clock, this is not a truly asynchronous timing option.

The last option, available only on timer/counter 2, is to use a truly asynchronous oscillator. That is, the oscillator will independently clock the timer/counter without any synchronization to the CPU clock. Timing events (such as interrupts) will by definition occur outside of the timescale of the CPU clock, yet the CPU clock drives

all of the hardware that deals with those events. Therefore, it is still necessary to limit the frequency of the asynchronous clock. It is recommended that the frequency be at least four times less than the CPU clock. In particular, the asynchronous timer/counter is meant for 32,768 Hz oscillators which, when prescaled, can be used to develop real time clocks.³

³ Atmel, "AVR134: Real Time Clock (RTC) Using the Asynchronous Timer," September 2016.

Timer/Counter Unit

Each of the timer/counter units can be simplified as a block diagram as shown in Figure 14.2. For each timer/counter, the variable n corresponds to the timer/counter number.

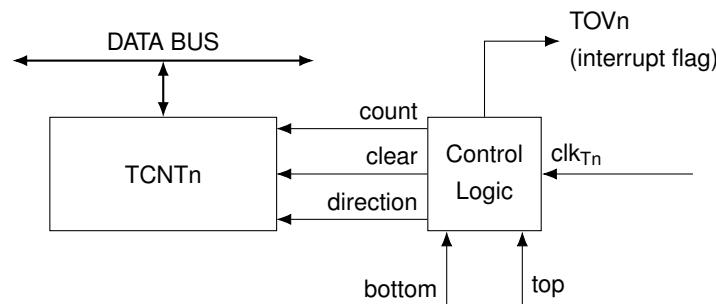


Figure 14.2: Block diagram of each timer/counter unit on the ATmega328P.

The definitions given in Table 14.2 are important to understand the operation of each timer/counter.

Name	Definition
bottom	0x00 (or 0x0000)
max	0xFF (or 0xFFFF)
top	highest value in the count sequence
direction	select between increment and decrement
count	signal to increment/decrement by 1
clk _{Tn}	timer/counter clock

Table 14.2: Important definitions to understand the operation of the timer/counter system.

Each timer/counter has two independent **output compare units**. Each output compare unit allows the timer/counter to trigger timed interrupts and/or generate square waves with differing frequencies and duty cycles. It is capable of generating pulse-width modulation (PWM) signals. The **input capture unit** (which is available only on timer/counter 1) captures input signals and can calculate their frequencies and duty cycles.

14.3 ATmega328P Output Compare Unit

The output compare unit block diagram is shown in Figure 14.3. The variable x corresponds to the output compare unit. In each output compare unit, the value on the timer/counter register TCNT_n is continuously compared with the values in an output compare register OCR_nx. The output compare unit consists of the following components:

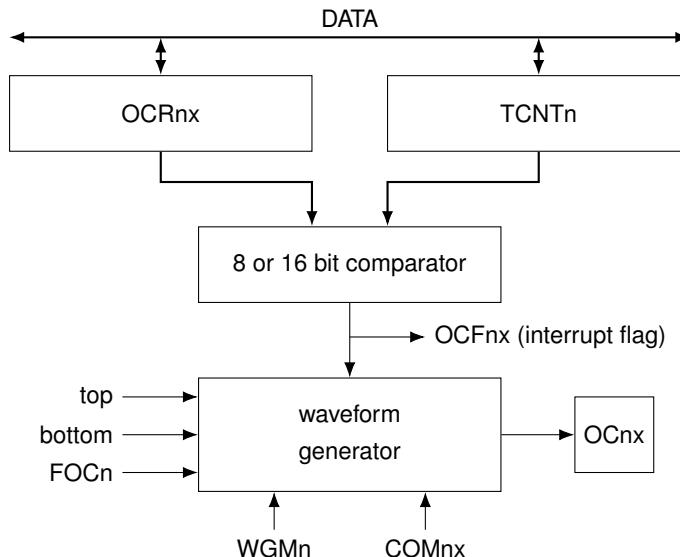


Figure 14.3: Block diagram of the timer/counter output compare unit.

One of the output compare units continuously compares the value stored in TCNT_n with the value stored in output compare register A (OCRnA). A match will set the output compare flag OCFnA. If interrupts are enabled, this will generate a compare match A interrupt. Depending on the mode of operation configured in the timer/counter control registers, the value stored in OCRnA can be used to modify the value of TOP and can also change the count direction. Based on the values stored in the COMnA bits in the timer/counter control registers, a compare match can modify a waveform generated on the output compare A (OCnA) pin.

The second output compare unit continuously compares the value stored in TCNT_n with the value stored in output compare register B (OCRnB). A match will set the output compare flag OCFnB. If interrupts are enabled, this will generate a compare match B interrupt. In addition, based on the values stored in the COMnB bits in the timer/counter control registers, a compare match can modify a waveform generated on the output compare B (OCnB) pin.

Timer/counters 0 and 1 can additionally be used as trigger sources for the analog to digital converter (ADC). Timer/counter 0 can trigger an ADC conversion on an overflow of the timer/counter (which

occurs when the flag T0V0 is set) or on a compare match between TCNT0 and 0CR0A (which occurs when the flag 0CF0A is set).

Timer/counter 1 can trigger an ADC conversion on an overflow of the timer/counter (which occurs when the flag T0V1 is set), on a compare match between TCNT1 and 0CR1B (which occurs when the flag 0CF1B is set), or when the flag ICF1 is set, which can occur either on an input capture event or on a compare match between TCNT1 and input capture register ICR1 (when ICR1 is used to define T0P).

Output signals (waveforms) on 0CnA and 0CnB can be generated based on the values of WGM_n and COM_n, which are initialized in the timer/counter control registers.

14.4 ATmega328P Timer/Counter Modes of Operation

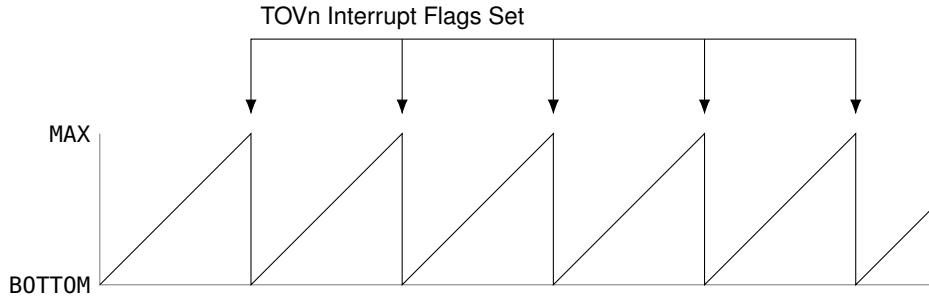
Using these two output compare units, the timer/counter can carry out one of its many modes of operation. These are

- normal mode,
- clear timer on compare match (CTC) mode,
- fast PWM mode,
- phase-correct PWM mode, and
- phase- and frequency-correct PWM mode (only available on timer/counter 1).

Normal and CTC modes will be discussed below. The PWM modes of operation will be discussed later in this chapter.

Normal Mode

In normal mode, the timer/counter unit will count up (increment) from BOTTOM to MAX. When the value in TCNT_n reaches MAX, the counter overflows and restarts from BOTTOM. In normal mode, the timer/counter overflow flag T0V_n will be set when the value in the timer/counter register becomes zero. A timing diagram depicting the value stored in TCNT_n over time in normal mode is shown in Figure 14.4.



The amount of time it takes for the timer/counter to overflow is defined by Equation 14.2, where N is the value of the timer/counter prescaler, n is the resolution of the timer/counter (either 8 or 16), and $f_{CLK,I/O}$ is the frequency of the I/O clock.

$$T_{normal} = \frac{N \times 2^n}{f_{CLK,I/O}} \quad (14.2)$$

In this manner, normal mode can be used to generate regularly occurring interrupts using the timer/counter overflow flag. Using the output compare unit to generate a waveform in normal mode is not recommended.

Clear Timer on Compare Match (CTC) Mode

In clear timer on compare match (CTC) mode, the timer/counter unit will count up (increment) from BOTTOM to TOP, where TOP is equal to the value stored in the output compare A register (OCRnA). (Timer/counter 1 has the additional possibility to have TOP equal to the value stored in the input capture register (ICR1) instead of from OCR1A.) Upon reaching the value of TOP, the timer/counter will clear to zero and continue incrementing again.

The amount of time it takes for the timer/counter to reach the TOP value is defined by Equation 14.3

$$T_{CTC} = \frac{N \times (TOP + 1)}{f_{CLK,I/O}} \quad (14.3)$$

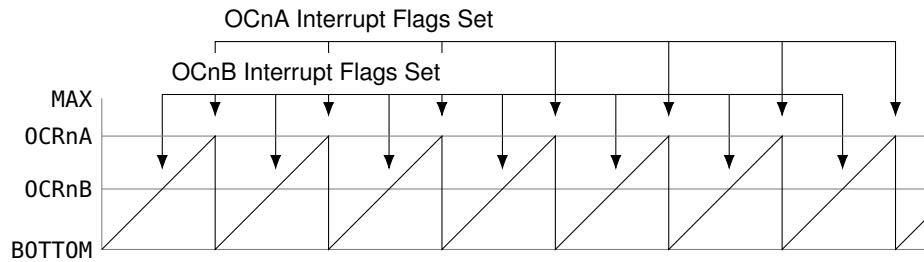
Because the value of TOP is configurable (as well as the value of the prescaler), the period of operation is much more flexible than what is possible in normal mode.

Upon reaching the value of TOP, the output compare flag A (OCFnA) will be set. (In timer/counter 1 counting to ICR1, the input capture flag ICF1 will be set.) In this manner, CTC mode can be used to generate regularly occurring interrupts using the timer/counter compare match A interrupt (or the input capture event interrupt).

Figure 14.4: Timing diagram of the value stored in register TCNTn when operating in normal mode.

Compare match B interrupts can also be enabled, as the output compare flag B (OCFnB) will be set when the value stored in TCNTn is equal to the value stored in OCRnB. The interval of time between compare match B interrupts will be the same as the interval of time between compare match A interrupts, with the interval of time controlled by the value of T0P. For example, if compare match A interrupts occur every 6 ms, compare match B interrupts will also occur every 6 ms. If the two output compare registers (OCRnA and OCRnB) contain different values, then there will be a time offset between the triggering of each individual compare match interrupt.

A timing diagram of a timer/counter in CTC mode is shown in Figure 14.5. The timer/counter compare match interrupt flags are both set at periodic intervals, controlled by the value stored in register OCRnA.



Note that the value of T0P can be changed in software (it does not need to be set to a static value that never changes). This would modify the timing of any interrupts or output compare waveforms (described below). If a new value is stored as T0P while the timer/counter is higher than that value, it will be unable to trigger a compare match that cycle. The timer/counter will increment to MAX, reset to zero, and then compare on the next round.

In CTC mode, the overflow flag T0Vn is set only when the timer/counter unit counts from MAX to BOTTOM. As this typically does not happen in CTC mode, it is recommended not to rely on overflow interrupts when using a timer/counter in CTC mode.

CTC mode can also be used to generate square waves with 50% duty cycle either of the output compare pins (OCnx). This occurs when the COMnx bits are configured to cause the output compare pin to toggle on a compare match and are configured as output pins in the corresponding data direction register. The period of the square wave is defined by Equation 14.4.

$$T_{OCnx} = \frac{2 \times N \times (TOP + 1)}{f_{CLK,I/O}} \quad (14.4)$$

The multiplication by 2 is due to the compare pin toggling each time

the timer/counter reaches T_{OP}. Therefore, it will take two cycles of reaching T_{OP} before the compare pin reaches its initial state, completing one full oscillation.

Similar to how compare match A and compare match B interrupts will occur with the same periodicity in CTC mode, square waves generated on each compare pin will also have the same period. However, depending on the values stored in each output compare register, there may be a time offset between each waveform.

An example of waveforms generated on both output compare pins of a single timer/counter using CTC mode is shown in Figure 14.6.

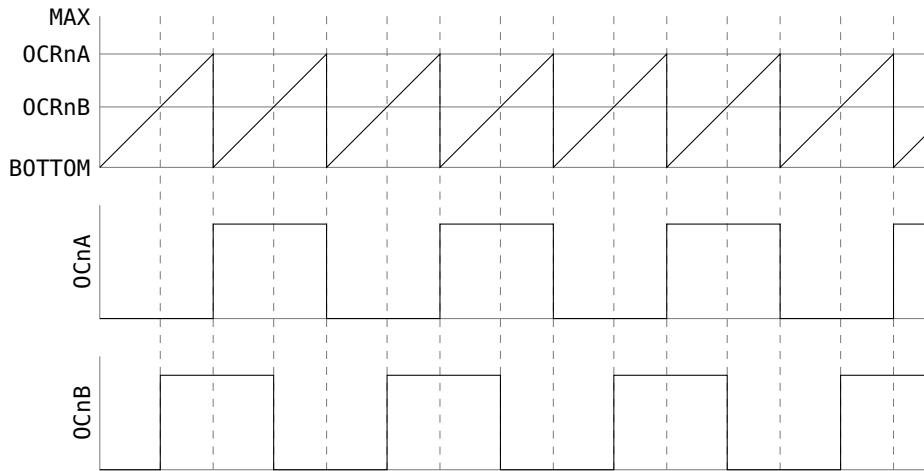


Figure 14.6: Output compare pins OCnA and OCnB configured to toggle on compare match for a timer/counter in CTC mode.

14.5 Pulse-Width Modulation (PWM)

Pulse-width modulation (PWM) is used to generate digital signals with varying average voltage levels. Instead of outputting an analog value, which is not possible without a digital to analog converter (DAC), a digital pulse with varying frequency and duty cycle is generated.

Duty Cycle

The duty cycle (D) of a square wave is the fraction of time that a signal is HIGH (T_{HIGH}) compared to the total period of the wave (T). The duty cycle can be calculated as defined in Equation 14.5.

$$D = \frac{T_{HIGH}}{T} \quad (14.5)$$

Duty cycle is typically expressed as a percentage. A square wave that is LOW all the time has a duty cycle of 0%, and a square wave that is

HIGH all the time has a duty cycle of 100%. Equal HIGH and LOW intervals would result in a duty cycle of 50%.

Average Voltage

Modifying the duty cycle of a square wave changes the signal's average voltage (\bar{V}). This average voltage can be calculated as defined in Equation 14.6, where V_{MAX} is the maximum signal voltage (usually V_{cc}), and V_{MIN} is the minimum signal voltage (usually 0 V).

$$\bar{V} = D \times V_{MAX} + (1 - D) \times V_{MIN} \quad (14.6)$$

Figure 14.7 shows five PWM waveforms, all with a period of 20 ms (frequency of 50 Hz). Each of the waveforms in Figure 14.7 is described below.

- Waveform A has a duty cycle of 0% and average voltage of 0 V.
- Waveform B has a duty cycle of 25% and average voltage of 1.25 V.
- Waveform C has a duty cycle of 50%, and average voltage of 2.5 V.
- Waveform D has a duty cycle of 75%, and average voltage of 3.75 V.
- Waveform E has a duty cycle of 100% and average voltage of 5 V.

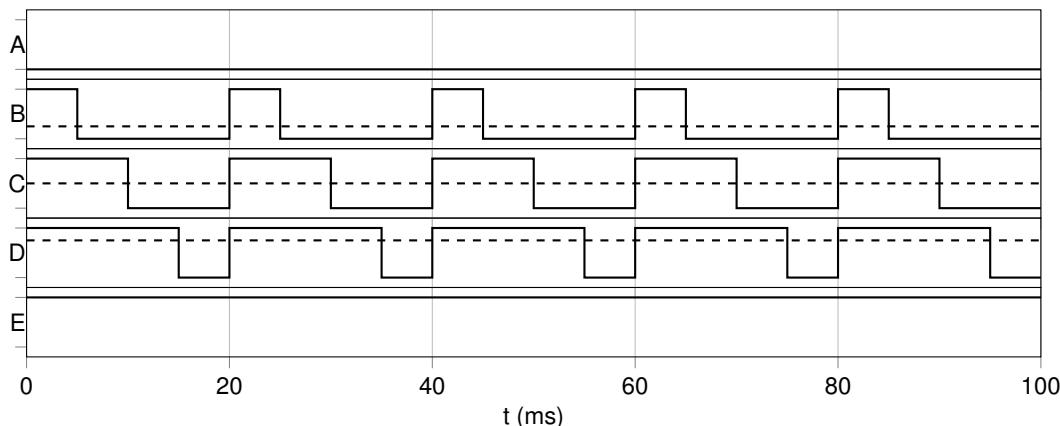


Figure 14.7: Pulse-width modulation waveforms with various duty cycles. The average voltage is indicated with a dashed line.

Frequency

The frequency of a PWM wave is equal to the number of complete cycles that occur per unit of time. It is independent of the waveform duty cycle. Figure 14.8 shows waveforms, all with 25% duty cycles, with different frequencies. The frequency of each waveform shown in Figure 14.8 is described below.

- Waveform A has a frequency of 200 Hz.
- Waveform B has a frequency of 100 Hz.
- Waveform C has a frequency of 50 Hz.
- Waveform D has a frequency of 20 Hz.

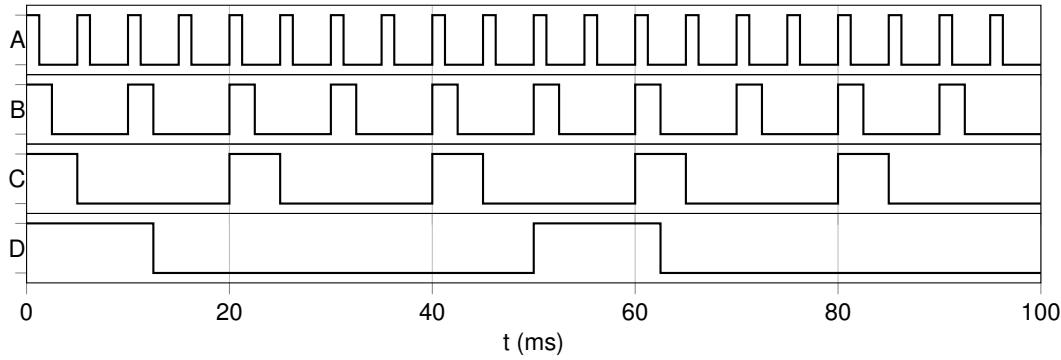


Figure 14.8: Pulse-width modulation waveforms with various frequencies. All have a duty cycle of 25%.

14.6 ATmega328P Timer/Counter PWM Modes of Operation

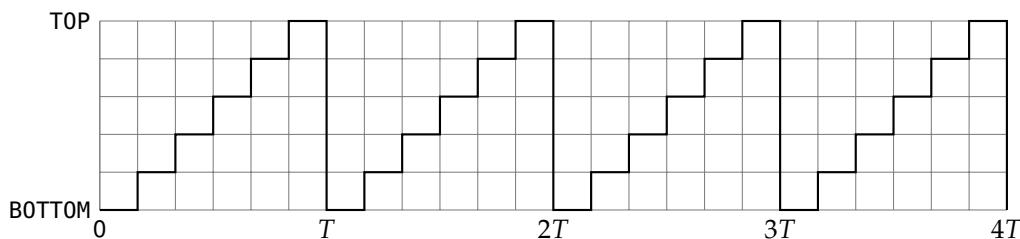
The ATmega328P has different modes of generating pulse-width modulation (PWM) signals.

Fast PWM

In fast PWM mode, the timer/counter increments from BOTTOM to TOP, and then restarts again from BOTTOM. The period of a fast PWM signal is given by Equation 14.7.

$$T_{fastPWM} = \frac{N \times (TOP + 1)}{f_{CLK,I/O}} \quad (14.7)$$

This equation can be understood by looking at how the value stored in the TCNT_n register increments over time. Counting between zero and TOP requires TOP plus one intervals of time. This is depicted in Figure 14.9.



Similar to the distinction between normal and CTC modes, fast PWM can either be configured to count to MAX or to a different T_{OP} value. These options are presented for each timer/counter in Table 14.3. (Note that these are also the same options used in phase-correct PWM mode, described in the next section.)

Timer/Counter 0	Timer/Counter 1	Timer/Counter 2
0xFF (MAX)	0xFF (8-bit mode)	0xFF (MAX)
OCR0A	0x1FF (9-bit mode)	OCR2A
	0x3FF (10-bit mode)	
	OCR1A	
	ICR1	

Table 14.3: Sources of T_{OP} for each timer/counter in both fast PWM and phase-correct PWM modes.

The timer/counter overflow flag (T_{OVn}) will be set each time the counter reaches T_{OP}. This allows for the use of overflow interrupts in fast PWM mode. Output compare flags will be set each time the counter reaches the value stored in an output compare register. This allows for the use of output compare interrupts as well.

Typically, fast PWM is used to generate waveforms on one or more timer/counter output compare pins. When the COM_nx bits are configured to cause the output compare pin(s) to operate in non-inverting mode, the output compare pin(s) will clear on a compare match and set when the timer/counter is reset to BOTTOM. (The opposite is true in inverting mode.)

If an exact frequency is desired, that cannot necessarily be generated by changing the prescaler and using MAX, the value stored in OCR_nA can be used as T_{OP} to set a specific frequency. The output compare pin B can be used to generate the PWM waveform, with the duty cycle set by the value stored in the OCR_nB register. The duty cycle in this case is defined by Equation 14.8.

$$D_{OCnB} = \frac{OCRnB}{OCRnA} \quad (14.8)$$

An example fast PWM waveform of this nature is shown in Figure 14.10.

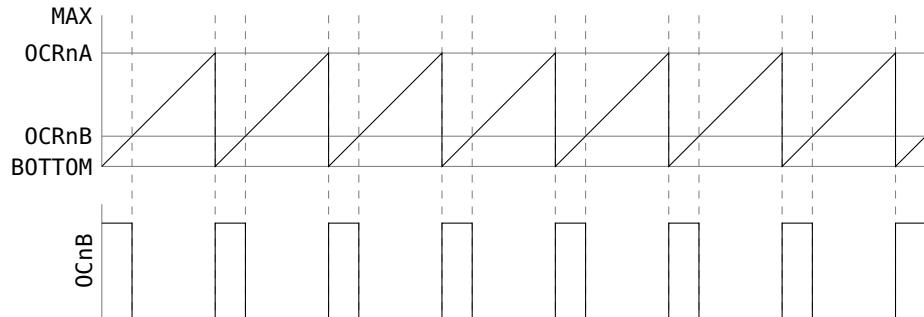


Figure 14.10: Example of a fast PWM waveform generated on pin OCnB using OCRnA as TOP. The duty cycle is set by the value stored in OCRnB.

When output compare register A is used as the value of TOP, output compare pin A cannot be used to generate a meaningful PWM waveform. This is because the pin is set every time there is a compare match, and cleared when the timer/counter resets to BOTTOM. However, if OCRnA is equal to TOP, then those two events occur simultaneously, and the value on the output compare A pin will always be HIGH.

An exception to this is timer/counter 1, where value stored in ICR1 can be used to define TOP. In this case, the frequency can be precisely tuned using the value of ICR1 and a prescaler. In addition, both output compare pins can have independent duty cycles configured by their corresponding output compare registers, defined by Equation 14.9.

$$D_{OC1x} = \frac{OCR1x}{ICR1} \quad (14.9)$$

If the frequency of the PWM waveform is not as strictly circumscribed, both output compare pins can be used to generate PWM signals in any timer/counter by using MAX as the value of TOP. In timer/counter 1, MAX can be configured to operate in 8-bit mode, 9-bit mode, or 10-bit mode, as described in Table 14.3. Both output compare pins will have PWM waveforms of the same frequency, but each duty cycle can independently be configured by the corresponding output compare register, with a duty cycle defined by Equation 14.10.

$$D_{OCnx} = \frac{OCRnx}{MAX} \quad (14.10)$$

Regardless of the value of TOP, when the value in the output compare register used to create the duty cycle of the wave is equal to BOTTOM, the output will be HIGH for one clock cycle. This can be considered to be a glitch, as the output would be expected to be LOW if the output compare register is equal to zero. This is depicted in Figure 14.11.

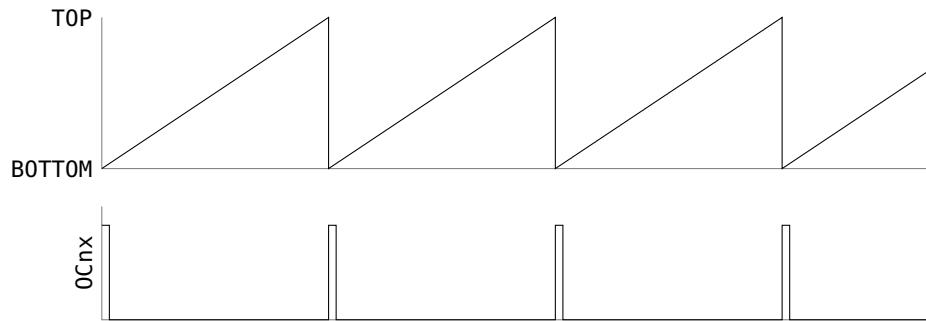


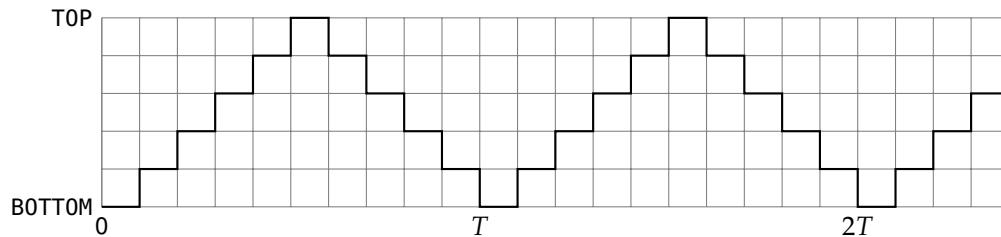
Figure 14.11: Example of the output glitch that occurs in fast PWM when the output compare register is equal to BOTTOM.

Phase-Correct PWM

In phase-correct PWM mode, the timer/counter increments from BOTTOM to TOP, and then decrements back to BOTTOM. The timer/counter remains at TOP for one clock cycle. The frequency of a phase-correct PWM signal is defined by Equation 14.11.

$$T_{\text{phase-correctPWM}} = \frac{2 \times N \times \text{TOP}}{f_{\text{CLK,I/O}}} \quad (14.11)$$

This equation can be understood by looking at how the value stored in the TCNT_n register increments over time. Because the timer/counter only remains at TOP for one interval of time, it takes TOP plus one intervals of time to increment to TOP, but then TOP minus one intervals of time to decrement back to BOTTOM, for a total of $2 \times \text{TOP}$ intervals of time. This is depicted in Figure 14.12.



Phase-correct PWM can either be configured to count to MAX or to a different TOP value. These options are given for each timer/counter in Table 14.3.

The timer/counter overflow flag (T0V_n) will be set each time the counter reaches BOTTOM. This allows for the use of overflow interrupts in fast PWM mode. Output compare flags will be set each time the counter reaches the value stored in an output compare register. This allows for the use of output compare interrupts as well.

Phase-correct PWM can be used to generate waveforms on one or more timer/counter output compare pins. When the COM_nx bits are configured to cause the output compare pin(s) to operate in non-

Figure 14.12: The value stored in a timer/counter in phase-correct PWM mode. It takes $2 \times \text{TOP}$ intervals of time to increment from BOTTOM to TOP and then decrement back to BOTTOM.

inverting mode, the output compare pin(s) will clear on a compare match when incrementing and set on a compare match when decrementing. (The opposite is true in inverting mode.) This creates an output waveform that is symmetric with respect to BOTTOM. This symmetry makes spikes or glitches in the output less likely (particularly in the case where the value of T0P is changed), and is preferred for operating motors.

Similarly to fast PWM mode, if a specific frequency is required that necessitates the use of 0CRnA as T0P, then output compare pin B can be used to generate the PWM waveform, with the duty cycle set by the value stored in the 0CRnB register. The duty cycle is defined in Equation 14.8.

An example phase-correct PWM waveform of this nature is shown in Figure 14.13.

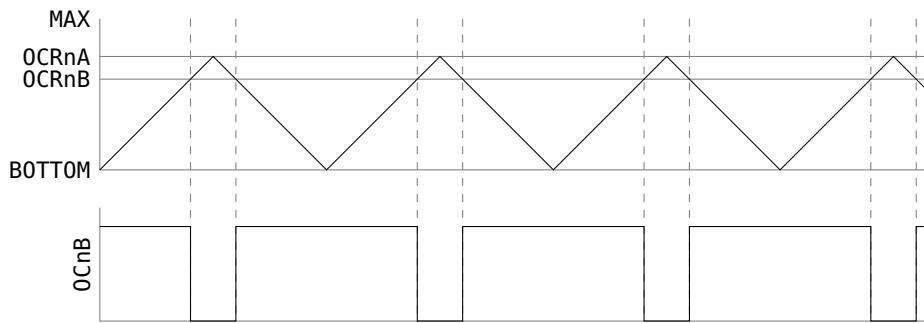


Figure 14.13: Example of a phase-correct PWM waveform generated on pin 0CnB using 0CRnA as T0P. The duty cycle is set by the value stored in 0CRnB.

When output compare register A is used as the value of T0P, output compare pin A cannot be used to generate a meaningful PWM waveform. This is because the pin is cleared on a compare match when incrementing, but set on a compare match when decrementing. However, these two events occur simultaneously, and the value on the output compare A pin will always be HIGH.

An exception to this is timer/counter 1, where value stored in ICR1 can be used to define T0P. In this case, the frequency can be precisely tuned using the value of ICR1 and a prescaler. In addition, both output compare pins can have independent duty cycles configured by their corresponding output compare registers, defined by Equation 14.9.

If the frequency of the PWM waveform is not as strictly circumscribed, both output compare pins can be used to generate PWM signals in any timer/counter by using MAX as the value of T0P. In timer/counter 1, MAX can be configured to operate in 8-bit mode, 9-bit mode, or 10-bit mode, as described in Table 14.3. Both output compare pins will have PWM waveforms of the same frequency, but each duty cycle can independently be configured by the corresponding

output compare register, with a duty cycle defined by Equation 14.10.

Unlike in fast PWM, there is no glitch that occurs in phase-correct PWM when the output compare register used to set the duty cycle is equal to BOTTOM . This makes phase-correct PWM a better choice for driving motors than fast PWM.

Updating TOP Values

The value of TOP can be modified in code; it does not need to be a fixed constant. This allows for the modification of the timer/counter frequency as the code executes. In order to prevent synchronization issues, the output compare A register is double buffered in PWM modes. (This buffering does **not** occur in normal or CTC modes!) This means that the CPU will write the new value of TOP to a buffer register and only update the value in OCRnA when a particular timing event occurs (either reaching the previous value of TOP or upon reaching BOTTOM).

The presence of the buffer register used with OCRnA is shown in Figure 14.14.

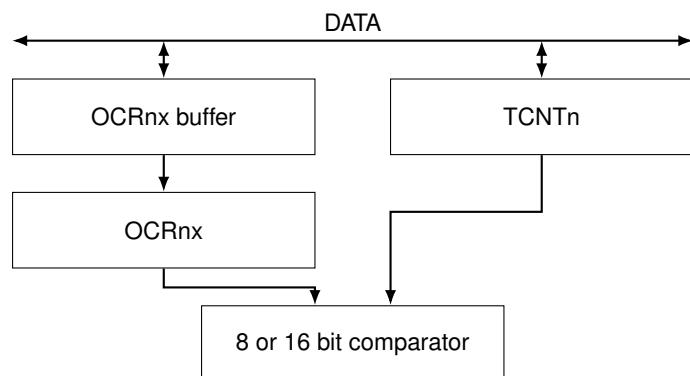
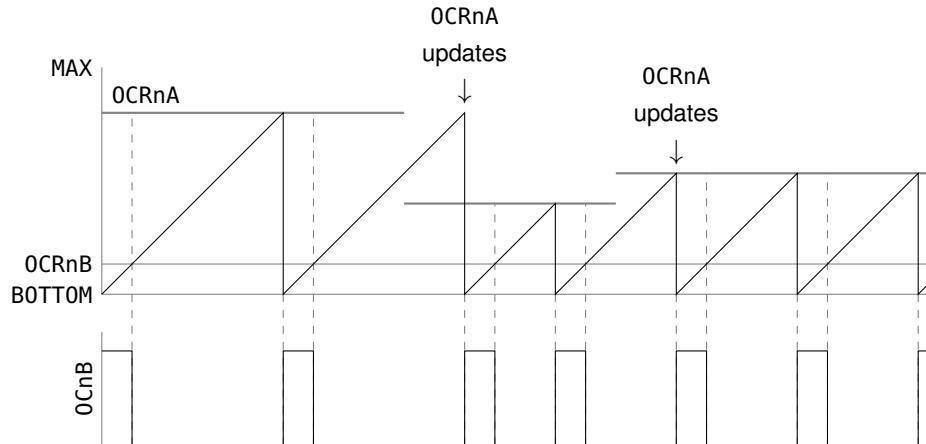


Figure 14.14: Block diagram of the timer/counter output compare register and the buffer register.

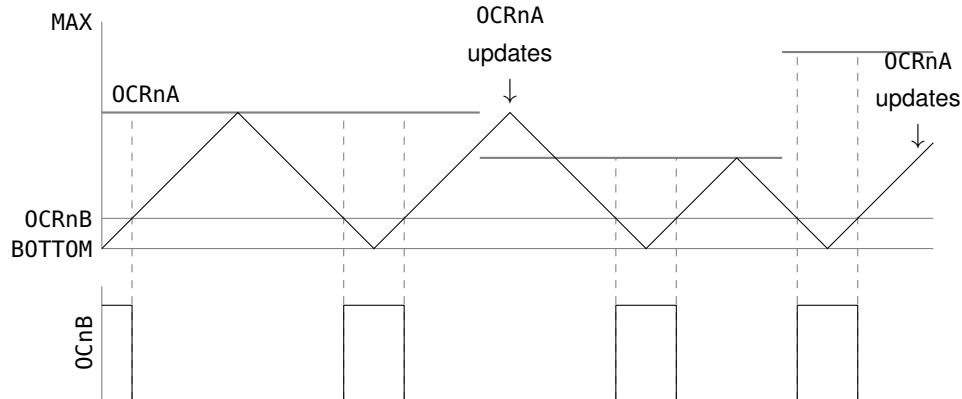
In fast PWM, changes to OCRnA are only updated the next time the counter reaches a value of TOP and subsequently clears back to zero. A waveform depicting this phenomenon is shown in Figure 14.15.



Even though the value of $OCRnA$ is modified, that change is only made immediately in the buffer register. The update to $OCRnA$ itself is only made once the timer/counter has reached $T0P$ and subsequently reset back to zero. The output waveform is also shown, depicting the change in PWM frequency with changing values of $OCRnA$.

The same feature is present in phase-correct PWM. Changes to $OCRnA$ are only updated the next time the counter reaches a value of $T0P$. This is depicted in Figure 14.16.

Figure 14.15: Example of a fast PWM waveform generated on pin $OCnB$ using $OCRnA$ as $T0P$. The value of $OCRnA$ is changed over time, but only updates when the value in the timer/counter becomes $T0P$ and clears to zero.

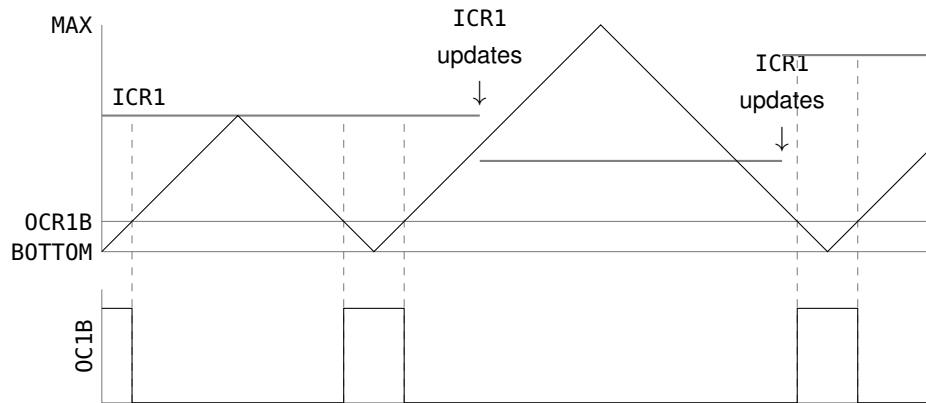


When an updated value of $OCRnA$ is higher than the previous value, the change effectively takes place immediately. When the updated value of $OCRnA$ is lower than the previous value, the change will not take effect until after the next waveform is generated (or after the next interrupt is invoked).

When using timer/counter 1, it is possible to use $ICR1$ as the value of $T0P$. However, this register is not double buffered. This means that the value in the register will update immediately. In either PWM mode, if the updated value of $ICR1$ is lower than the previous value,

Figure 14.16: Example of a phase-correct PWM waveform generated on pin $OCnB$ using $OCRnA$ as $T0P$. The value of $OCRnA$ is changed over time, but only updates when the value in the timer/counter becomes $T0P$.

the timer/counter will not make a compare match and will count to MAX (65,535) before beginning the next cycle. Once the next cycle begins, a compare match will occur as expected. However, one cycle of the waveform (or interrupt timing) will be significantly affected. This is depicted with phase-correct PWM in Figure 14.17. Because of this issue, it is recommended to use ICR1 only if the value of T0P will remain fixed.



When the value of T0P is changed in phase-correct PWM mode using OCRnA as T0P, even though the register is double buffered, there is still one issue that can occur when the value of OCRnA is changed. This issue is that waveforms are not completely symmetric about BOTTOM, which may be an issue when working with motors. This issue will be considered in the next section on phase- and frequency-correct PWM.

Phase- and Frequency-Correct PWM

Phase- and frequency-correct PWM is much the same as phase-correct PWM, except that the output periods are all symmetrical. This is produced by updating the value of OCRnA on BOTTOM, rather than on T0P. This PWM mode of operation is only available on timer/counter 1.

Because the only difference in operation regards how the value of T0P updates, this mode of operation has the same frequency and duty cycle considerations as phase-correct PWM. The values available for T0P are given in Table 14.4.

Timer/Counter 0	Timer/Counter 1	Timer/Counter 2
unavailable	OCR1A	unavailable
	ICR1	

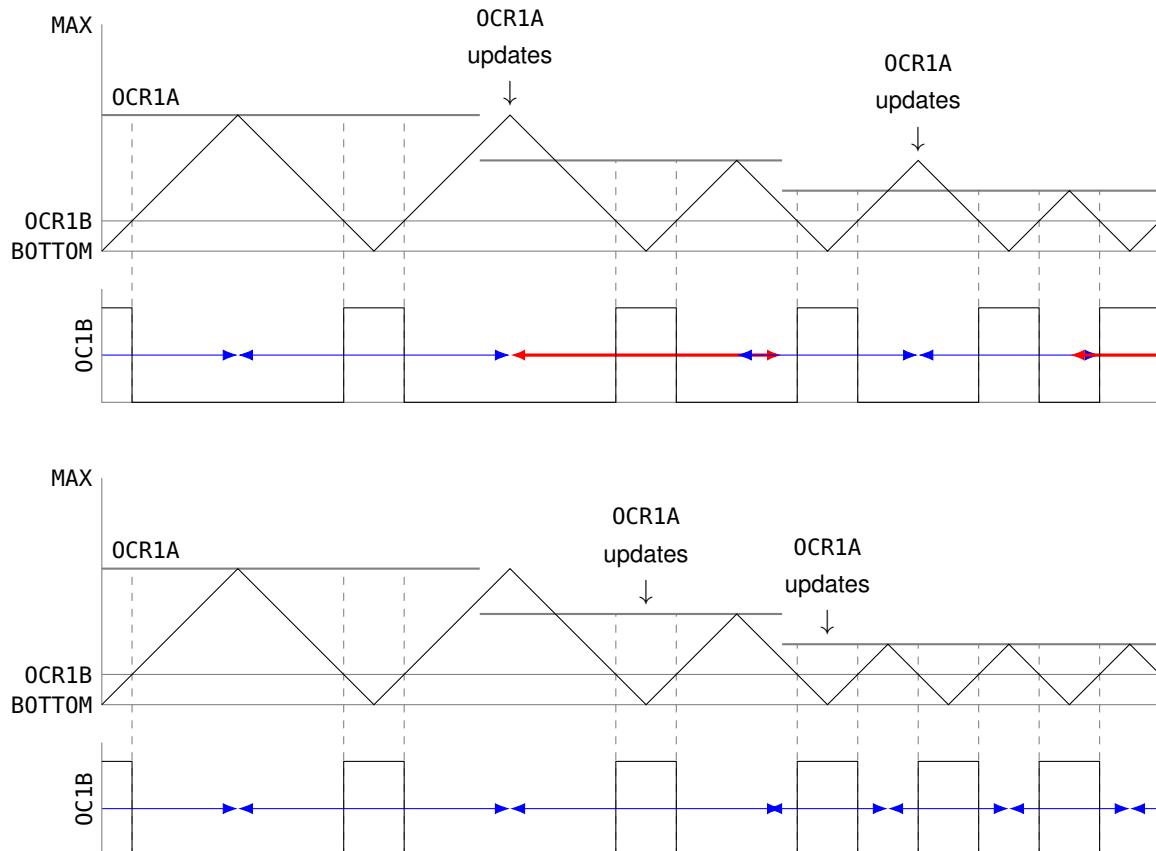
Because of the problematic nature of ICR1 not being double

Figure 14.17: Example of a phase-correct PWM waveform generated on pin OC1B using ICR1 as TOP. The value of ICR1 is changed immediately, which may cause timing issues if the value of ICR1 is lower than the value stored in TCNT1.

Table 14.4: Sources of T0P for timer/counter1 in phase- and frequency-correct PWM modes.

buffered (which was explained in the previous section), it is not recommended to use that register as the value for T0P when the value needs to be changed over time. Because this PWM mode of operation is designed to be used in the specific instance when T0P needs to change, it is strongly recommended that any designs using this PWM mode of operation use OCR1A instead of ICR1.

The issue of waveform asymmetry and symmetry is demonstrated in Figure 14.18, which depicts a phase-correct PWM signal (top) that is not symmetric around BOTTOM. However, the phase- and frequency-correct PWM signal (bottom) is completely symmetric around BOTTOM.



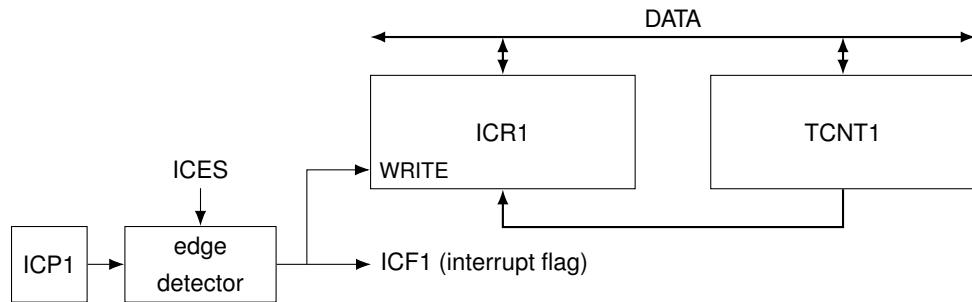
If a symmetric waveform is required, and the value of T0P can change over time, then it is recommended to use phase- and frequency-correct PWM counting to a T0P value of OCR1.

14.7 ATmega328P Input Capture Unit

Timer/counter 1 on the ATmega328P has an input capture unit in addition to its two output compare units. The capture unit is capable

Figure 14.18: Example of a phase-correct PWM waveform (top) and phase- and frequency-correct PWM waveform (bottom) generated by timer/counter 1. The phase-correct PWM waveform is not completely symmetric about BOTTOM, but the phase- and frequency-correct PWM waveform is.

of detecting rising or falling edges applied to the input capture pin (ICP1). A high-level block diagram of the input capture unit is shown in Figure 14.19.



The trigger event on ICP1 is based on the value of the input capture edge select (ICES) bit in timer/counter 1 control register B. When $\text{ICES} = 0$, an input capture is triggered on a falling edge of the signal. Otherwise, an input capture is triggered on a rising edge.

At the moment of a capture event, the value of TCNT1 is written to the input capture register ICR1. At the same time, the input capture flag ICF1 is set.

It is recommended to use the input capture unit only when the value of TOP used with timer/counter 1 is unchanging, and only when the value in the timer/counter will exclusively increment (in other words: do not use the input capture unit in phase-correct or phase- and frequency-correct PWM modes). It's important to understand the time interval between timer/counter increments, and ensure that any input signal will be sampled often enough by the timer/counter to adequately trigger on rising or falling edges. Care should also be taken to understand the timing of the input capture event interrupt. If the interrupt service routine (ISR) is lengthy enough, a second trigger event could occur, causing the value of ICR1 to change. Therefore, it's important to store the value of ICR1 to a variable as early in the ISR as possible.

Using the Input Capture Unit to Measure Signal Timing

The input capture unit can be used to measure the period and duty cycle of devices that output a square wave signal. To calculate the waveform period, two subsequent identical edges can be captured (either both rising edges or both falling edges). The value stored in ICR1 at each edge denotes the timestamp when the edge occurred. Subtracting the timestamp of the second edge (K_2) from the timestamp (K_1) of the first edge gives the number of timer/counter increments between the edges (K), as defined in Equation 14.12.

Figure 14.19: High-level block diagram of the timer/counter 1 input capture unit.

$$K = K_1 - K_2 \quad (14.12)$$

The amount of time that elapses between each increment of the timer/counter can be calculated based on the prescaler (N) of the timer/counter and the I/O clock frequency ($f_{CLK,I/O}$), and is defined in Equation 14.13.

$$\Delta t = \frac{N}{f_{CLK,I/O}} \quad (14.13)$$

The period of the wave can therefore be calculated as defined in Equation 14.14.

$$T = K \times \Delta t \quad (14.14)$$

An example waveform applied to ICP1 is shown in Figure 14.20, with ICES configured to trigger a capture event on a rising edge of the input signal. The timestamp at each rising edge is shown.

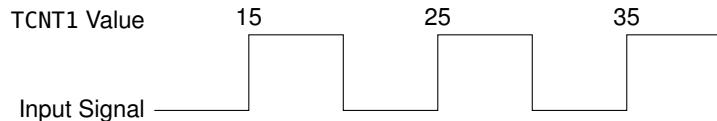


Figure 14.20: Using the timer/counter 1 input capture unit to calculate the period of an input signal.

If the prescaler of the timer/counter is equal to 64, and the clock frequency is 8 MHz, the period of the wave can be calculated as shown in Equation 14.15.

$$T = (25 - 15) \times \frac{64}{8 \times 10^6 \text{ Hz}} = 80 \mu\text{s} \quad (14.15)$$

The HIGH period of the input wave can be measured, but requires capturing a rising edge and subsequent falling edge of the input signal. This means that ICES needs to be modified between capture events. The timestamp of the rising edge (K_R) can be subtracted from the timestamp of the falling edge (K_F) to calculate the number of increments that occurred between the two events, as shown in Equation 14.16.

$$K = K_F - K_R \quad (14.16)$$

The relationship between time between increments is the same as that used for calculating the period. The HIGH period is also calculated by multiplying the number of increments by the amount of time that elapses between increments.

An example waveform applied to ICP1 is shown in Figure 14.21, with ICES configured to trigger a capture event on a rising edge of the input signal, and then switch to triggering a capture event on a falling edge. The timestamp at each trigger event is shown.

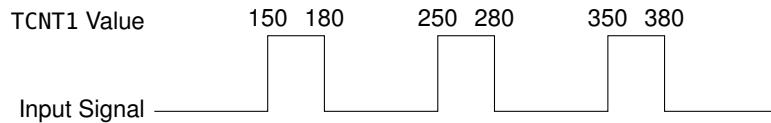


Figure 14.21: Using the timer/counter 1 input capture unit to calculate the HIGH period of an input signal.

If the prescaler of the timer/counter is equal to 1024, and the clock frequency is 1 MHz, the HIGH period of the wave can be calculated as shown in Equation 14.17.

$$T_{HIGH} = (180 - 150) \times \frac{1024}{1 \times 10^6 \text{ Hz}} = 30.72 \text{ ms} \quad (14.17)$$

The overall period of the wave is calculated as shown in Equation 14.18.

$$T = (250 - 150) \times \frac{1024}{1 \times 10^6 \text{ Hz}} = 102.4 \text{ ms} \quad (14.18)$$

To calculate the duty cycle, simply use the relationship between duty cycle, period, and HIGH period. For the waveform depicted in Figure 14.21, the duty cycle is equal to the calculation shown in Equation 14.19.

$$D = \frac{30.72 \text{ ms}}{102.4 \text{ ms}} = 30\% \quad (14.19)$$

14.8 ATmega328P Watchdog Timer (WDT)

The watchdog timer (WDT) is a feature on AVR microcontrollers that allows for system resets if the program is unresponsive after a certain period of time.⁴ The WDT is clocked with a 128 kHz clock that is independently of the on-chip oscillator. The WDT has three operating modes, which are

- interrupt,
- system reset,
- interrupt then system reset.

The WDT is a timer that will increment at intervals of time given by the 128 kHz clock and the WDT prescaler (which can be configured in the WDT control register). Whenever the WDT is reset (using the machine instruction WDR), the timer will reset to zero. However, if the WDT overflows, the WDT will either invoke an interrupt, reset the system, or invoke an interrupt and then reset the system (based on the selection made in the WDT control register).

In this manner, the WDT can be used to restart the microcontroller program after a certain time interval, which can be beneficial for

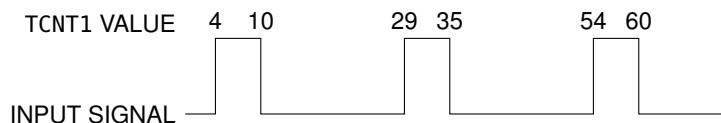
⁴ Atmel, "AVR132: Using the Enhanced Watchdog Timer," June 2008.

avoiding unresponsive code or runaway timing errors. Alternatively (or in addition), an interrupt can be invoked that can be used to store important program information or wake up from a sleep mode.

14.9 Practice Problems

Assume a clock frequency of 16 MHz to answer all of the following questions.

1. How many timer/counters are available on the ATmega328P? 3
2. True or false: Timer/counter 0 is a 16-bit counter. FALSE
3. On timer/counter 0 and timer/counter 2, in normal mode, when the counter rolls over, it goes from _____ to _____. 0xFF to 0x00
4. What is the period and duty cycle of the following waveform, given an I/O clock frequency of 1 MHz? $T = 25 \mu\text{s}, D = 24\%$



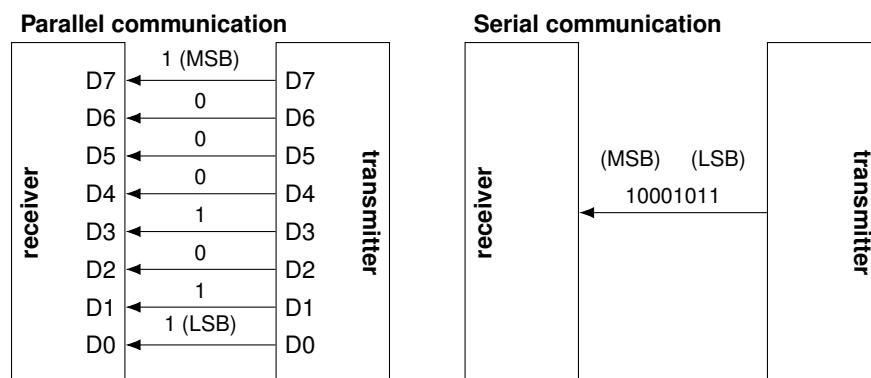
5. In normal mode, what is the longest possible delay that can be obtained using timer/counter 0 or timer/counter 2? With what prescaler value is this achieved? $T_{MAX} = 16.38 \text{ ms}, N = 1024$
6. In normal mode, what is the longest possible delay that can be obtained using timer/counter 1? With what prescaler value is this achieved? $T_{MAX} = 4.19 \text{ s}, N = 1024$
7. In normal mode, how much time elapses between timer/counter overflows when using timer/counter 1 and a prescaler $N = 8$? 32.768 ms
8. In normal mode, what prescaler must be used with timer/counter 2 to achieve a delay of 0.512 ms? Is this possible with timer/counter 0? Why or why not? 32, Not possible with TCNTO because it does not have a prescaler option of 32.

9. In CTC mode, what delay is obtained when $0\text{CR1A} = 1499$ and the prescaler $N = 64$? 6 ms
10. In CTC mode using timer/counter 2 and a prescaler $N = 32$, what value must be loaded into 0CR2A to obtain a delay of 0.38 ms? 189
11. In CTC mode, what prescaler value must be used if $0\text{CR0A} = 140$ to obtain a delay of ≈ 9 ms? 1024

15

Serial Communication

SERIAL COMMUNICATION PROTOCOLS allow data to be sent with far fewer wires than with parallel I/O. The schematic difference between serial and parallel communication protocols is shown in Figure 15.1.



While parallel data transmission allows many bits to be sent simultaneously, it requires as many wires as bits. This can be prohibitive in a microcontroller as the number of I/O ports is quite limited. In addition, parallel I/O can suffer from noise (crosstalk) and signal reflections if the source and destination are not close. Long distance synchronization can be difficult to achieve if there is significant delay in transmitting from one device to another.

Serial data is sent one bit at a time. Few wires are used, which minimizes crosstalk, meaning that it can be used over long distances. It is also cheaper to implement. There are many features of serial communication that may change the configuration and number of wires required to implement the communication protocol.

Figure 15.1: Schematic difference between parallel I/O (which uses many wires) and serial I/O (which can use as few as one wire).

15.1 Simplex and Duplex

Simplex communication is capable of sending data in only one direction. Consider for example a radio. Information is broadcast to the radio receiver, but no information is transmitted back. It is simply a unidirectional flow of data. A duplex communication system consists of two devices that can communicate with each other in both directions (i.e. as receivers and as transmitters).

In **half-duplex** communication, only one wire is present to send and receive data, therefore information can be sent in both directions, but only one direction at a time. Half-duplex communication can be considered similar to speaking on a walkie-talkie; one individual speaks at a time and indicates the end of their message by saying "over."

In **full-duplex** communication, information can be sent in both directions simultaneously, which requires the use of one more wire than half-duplex communication. Full-duplex communication is possible using phones where two people can speak simultaneously (although limitations of humans may make it difficult for both parties to understand; this complication is not present in computing devices).

15.2 Architecture

Many serial protocols use a primary-secondary configuration. The primary device has unidirectional control over other devices (secondaries). In some serial communication protocols, the primary selects the active secondary device, and also supplies the clock signal. Some serial protocols allow multiple primary devices. In addition, some protocols allow the roles of primary and secondary to be changed between message transmissions.

15.3 Data Transfer Protocol

In serial communication, data can be sent synchronously or asynchronously. **Synchronous** communication requires a clock signal to be provided by the primary (which requires the existence of another wire to carry the clock signal). With synchronous communication, once (8-bit) data transfer is initiated, the receiver has only to wait 8 clock cycles to obtain the information.

In **asynchronous** communication, START and STOP bits are required to signal to the receiver that transfer has been initiated and completed. Certain serial protocols allow the user to specify whether data should be sent LSB-first or MSB-first. In asynchronous communication, START and STOP bits are required to specify when data

communication has commenced or completed. In addition, data rate, electrical signal definition for HIGH and LOW, and handshaking protocols may also need to be defined in each serial protocol.

Parity Error Detection

Parity refers to the number of 1's in a variable or data stream. Parity is ODD if there is an odd number of 1's, and EVEN if there is an even number of 1's. In order to create a data stream with a particular parity (which is decided upon by both the transmitter and receiver beforehand), a parity bit must be generated. Parity checking is a simple way to check for transmission errors, although it is not foolproof (it is capable of producing false positives, but not false negatives). It is simple to implement, requiring only XOR or XNOR gates, and only requires a single bit added to a signal. However it is only able to detect an error if an odd number of bits are corrupted during transmission.

Even Parity: Success

Person A wants to send 1011

$$\text{Compute parity: } P = 1 \oplus 0 \oplus 1 \oplus 1 = 1$$

Send signal 10111 (the last bit is P , the parity bit)

Person B receives 10111

$$\text{Compute parity: } 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 0 = \text{even}$$

Transmission assumed successful

Even Parity: Failure

Person A wants to send 1011

$$\text{Compute parity: } P = 1 \oplus 0 \oplus 1 \oplus 1 = 1$$

Send signal 10111 (the last bit is P , the parity bit)

Person B receives 10011

$$\text{Compute parity: } 1 \oplus 0 \oplus 0 \oplus 1 \oplus 1 = 1 = \text{odd}$$

Transmission was not successful

Table 15.1: Example of error detection using parity-checking.

Serial Protocols on the ATmega328P

The serial communication protocols supported by the ATmega328P microcontroller are

- serial peripheral interface (SPI),
- universal synchronous/asynchronous receiver/transmitter (USART), and
- two-wire interface (TWI).

15.4 ATmega328P Serial Peripheral Interface (SPI)

SPI communication on the ATmega328P is capable of full-duplex synchronous communication using a maximum of four wires. The Arduino can be configured as either a primary or a secondary, and can send data bytes either MSB first or LSB first.

The ATmega328P SPI bus uses four logic signals, given in Table 15.2. Each of the signals is associated with a particular pin on Port B. (Please note that there is an inconsistency between the pin names and the descriptions, see the author note at the start of this book for more information.)

Name	Pin	Description
SCK	D13	Serial clock (output from primary)
MOSI	D11	Primary output, secondary input
MISO	D12	Primary input, secondary output
SS	D10	Secondary select (active low, output from primary)

Table 15.2: SPI I/O pins.

The primary and secondary devices are connected as shown in Figure 15.2. As can be seen, shift registers exist on both the primary and secondary devices. As a data byte is transmitted from the primary to the secondary along the MOSI line, the data that was previously stored in the secondary's shift register transmits from the secondary to the primary along the MISO line.

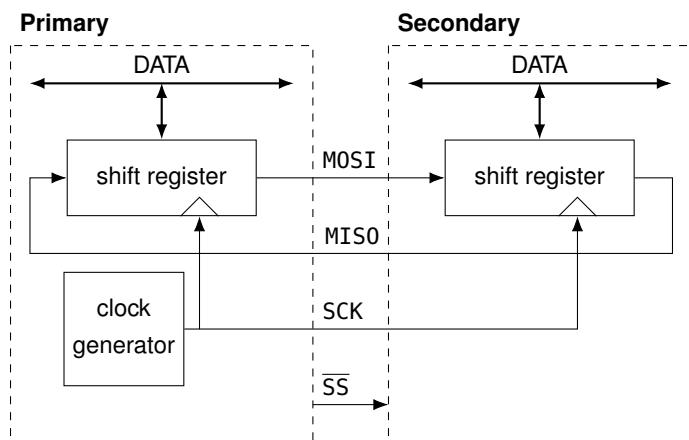


Figure 15.2: SPI primary-secondary connection diagram.

The ATmega328P can be configured either as a primary device or as a secondary device.¹

¹ Atmel, "AVR151: Setup and Use of the SPI," February 2016.

ATmega328P in Primary Mode

The microcontroller can be configured as a primary device by setting the SPI Primary/Secondary Select bit (MSTR) in the SPI Control Register (SPCR). The microcontroller hardware will override any data direction on the MISO pin to ensure that it is an input pin. The other three SPI pin data directions can be set at the programmer's discretion.

If the microcontroller won't be transmitting any data, the MOSI pin need not be used. However, as the primary device asserts the clock signal, the SCK pin will likely need to be connected and configured as an output pin. The secondary select pin (D10) must either be configured as an output pin or as an input pin that is held HIGH. (This is true regardless of whether or not the secondary select pin is necessarily used in a circuit design!) Typically, it is simpler to ensure the secondary select pin is an output pin by setting the appropriate bit in DDRB.

ATmega328P in Secondary Mode

The microcontroller can be configured as a secondary device by ensuring that the SPI Primary/Secondary Select bit (MSTR) in the SPI Control Register (SPCR) is cleared. The microcontroller hardware will override any data direction on the MOSI, SS, and SCK pins to ensure that they are input pins. The MISO pin data direction can be set at the programmer's discretion.

SPI will be activated and capable of receiving data when the SS bit is held LOW. When held HIGH, the SPI hardware will not receive any incoming data.

SPI Primary-Secondary Configuration

Multiple secondary devices can be supported with SPI. This can be accomplished independently as shown in Figure 15.3. Independent secondary devices all require their own secondary select signal. Therefore using many independent secondary devices can require a large number of I/O pins.

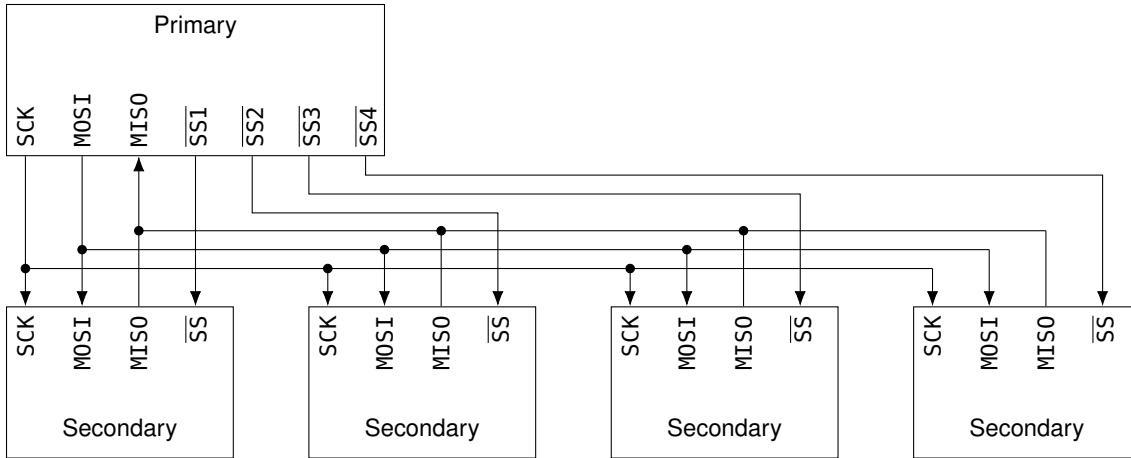


Figure 15.3: Independent secondary configuration in SPI.

Multiple secondary devices can also be connected in a daisy-chained configuration as shown in Figure 15.4. Daisy-chained secondary devices have the output of one secondary feeding into the input of the next, sharing a common secondary select signal.

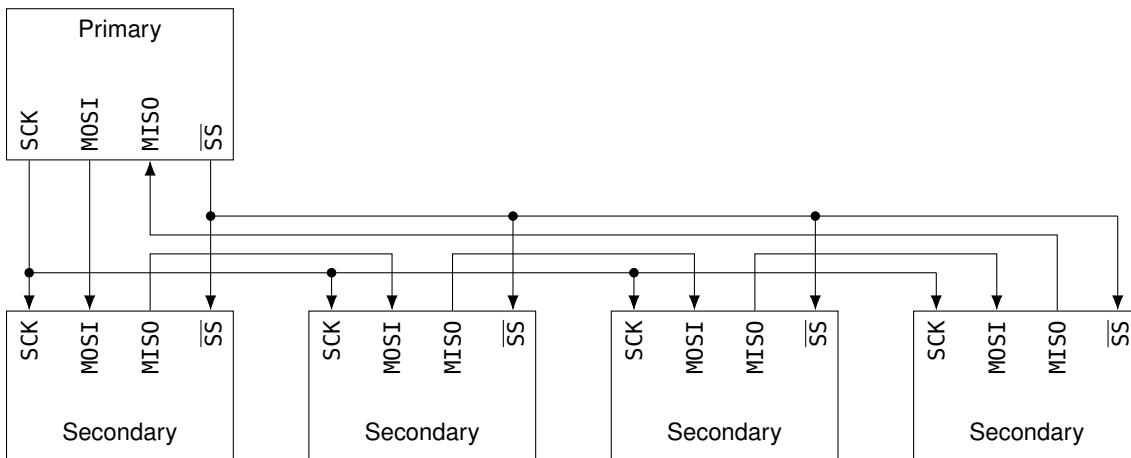


Figure 15.4: Daisy-chained secondary configuration in SPI.

SPI Clock Polarity and Phase

The clock polarity indicates the idle value of the clock signal. When idle, the SCK will be kept LOW if the polarity bit (known as CPOL) in the SPI control register is 0. Otherwise, if CPOL is 1, the clock signal will be kept HIGH when idle. A timing diagram depicting the value on SCK is shown in Figure 15.5 for both values of CPOL.

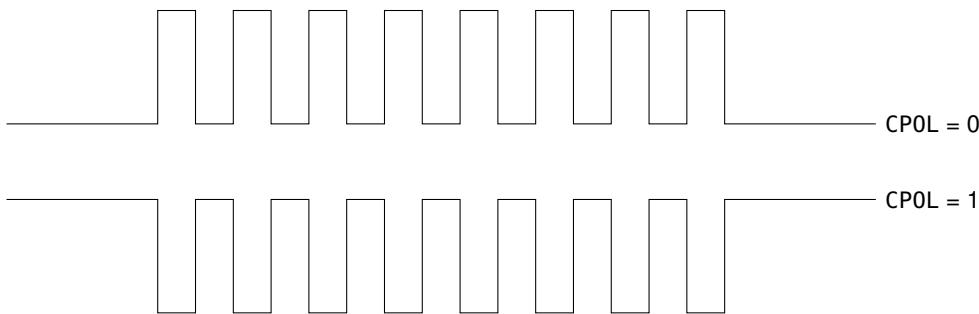


Figure 15.5: SPI clock signal (SCK) for different values of CPOL.

The clock phase, configured by the CPHA bit in the SPI control register, indicates the timing of sampling (reading data from the receive pin) shifting (sending data to the transmit pin). The times at which data will be sampled and shifted based on the values of CPOL and CPHA are listed in Table 15.3.

CPOL	CPHA	Shift Data	Sample Data
0	0	First SS, then falling edges	Rising edges
0	1	Rising edges	Falling edges
1	0	First SS, then rising edges	Falling edges
1	1	Falling edges	Rising edges

Table 15.3: SPI modes of operation.

When two microcontrollers are communicating to each other using the SPI protocol, each must have the same values configured for CPOL and CPHA. When a single microcontroller is communicating with a piece of hardware (say, a shift register chip), then the datasheet must be consulted to determine the values to set for CPOL and CPHA based on the piece of hardware. (For example, the 74595 shift register samples on rising edges, so CPHA must be zero.)

Advantages and Disadvantages of SPI

The SPI protocol has a number of advantages and disadvantages. The advantages of SPI include

- full duplex communication,
- simple hardware requirements (including no requirement for a secondary address assignment as in TWI),
- no requirement for built-in hardware as the protocol can be achieved by “bit-banging” (manually setting and clearing I/O pins to achieve the desired signals),
- no protocol-level limit to clock speed (this is only limited by hardware and microcontroller clock speeds),

- no protocol-level limit to the number of bits that can be sent at a time (however the ATmega328P architecture limits to 8 bits at a time), and
- typically lower power consumption than protocols such as TWI due to the absence of pull-up resistors on secondary devices.

The disadvantages of SPI include

- more pins are required in SPI than in USART or TWI protocols,
- capable of only short-distance communication,²
- lack of error-checking protocol, and
- lack of secondary device acknowledgement (the primary could send data to nowhere and not realize it).

² Kugelstadt, Thomas, "Extending the SPI bus for long-distance communication," 2011.

15.5 ATmega328P Universal Synchronous/Asynchronous Receiver/Transmitter (USART)

USART is a serial protocol commonly included in microcontrollers. The ATmega328P USART is a full duplex protocol that is capable of operating either asynchronously or synchronously. The USART on the ATmega328P is also capable of operating in SPI as a primary device. Regardless of the mode of operation, the USART protocol operates by sending **frames** of information. A frame consists of

- one LOW start bit,
- data bits (LSB first),
- (optional) even or odd parity bit, and
- one or two HIGH stop bits.

The number of data bits that can be transmitted or received can vary between 5–9 bits, and is configured using the USART Character Size bits (UCSZ0) in the USART Control and Status Registers B and C (UCSR0B and UCSR0C).

When configured as a transmitter, the microcontroller can generate either an even or odd parity bit to include in the frame. When configured as a receiver, the microcontroller will receive a parity bit, if configured, and use it to for parity error detection.

The number of stop bits included in the frame is also configurable.

Flags are generated upon completion of transmission or receiving data, or when the transfer data register is empty. When enabled, these flags can generate interrupts.

A block diagram of USART features and functionality is given in Figure 15.6.

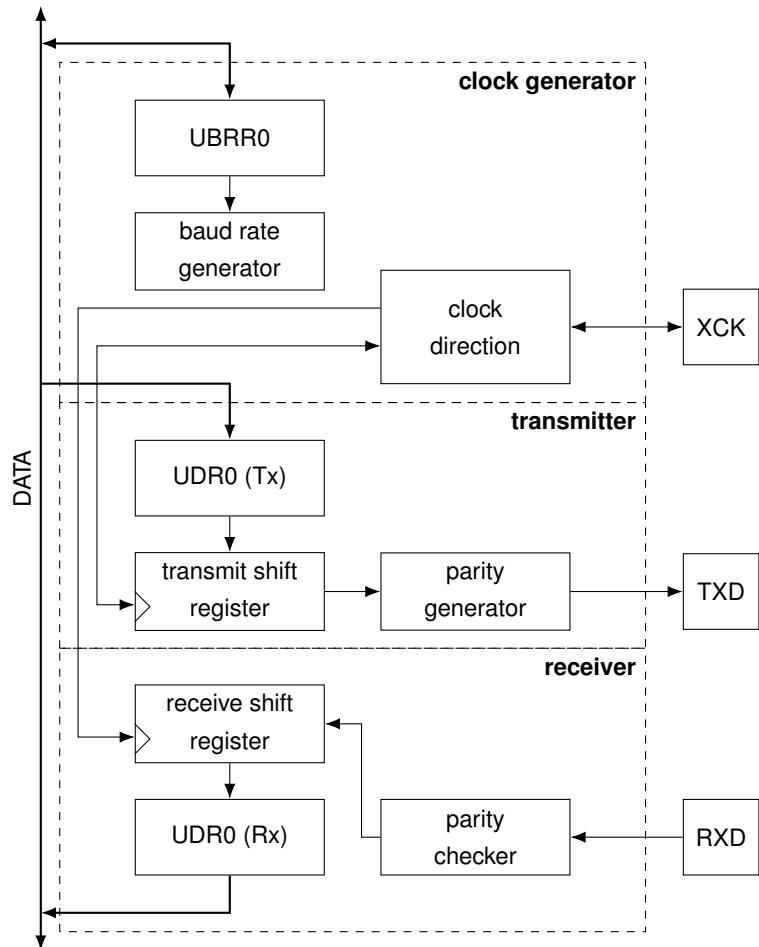


Figure 15.6: USART block diagram.

USART I/O pins

In asynchronous mode, two I/O pins are used, one to transmit data (TXD), and one to receive data (RXD). In synchronous mode, an additional pin is required for the clock signal (XCK). These I/O pins are listed in Table 15.4.

Name	Pin	Description
XCK	D4	USART clock
TXD	D1	Transmit data (data output)
RXD	D0	Receive data (data input)

Table 15.4: USART I/O pins.

When the USART transmitter is enabled, the TXD pin is configured

as an output, overriding any other value on the DDRD register. When the USART receiver is enabled, the RXD pin is configured as an input, overriding any other value on the DDRD register.

USART Clock and Baud Rate Generator

Regardless of whether or not the USART is operated synchronously or asynchronously, the data must be received or transmitted at a consistent rate. (The difference is that in synchronous mode, the primary device on the USART supplies a clock source consistent with this data transmission/receive rate.) In the USART, this is known as a baud rate, which is defined in bits per second (bps). The baud rate should always be configured prior to initializing the USART using the Control and Status registers.³

The baud rate can be set using the USART baud rate register (UBRR0), and the baud rate that is generated depends on the mode of operation, as described in Table 15.5.

³ Atmel, "AVR306: Using the AVR USART on tinyAVR and megaAVR devices," April 2016.

Table 15.5: Equations to define the BAUD rate and UBRR0 register in each operating mode of the USART.

Operating Mode	Baud Rate Equation	UBRR0 Value Equation
Asynchronous normal mode	Equation 15.1	Equation 15.4
Asynchronous double-speed mode	Equation 15.2	Equation 15.5
Synchronous primary mode	Equation 15.3	Equation 15.6

The baud rate equations, given a value stored in UBRR0 are defined below.

$$BAUD = \frac{f_{CLK,I/O}}{16 \times (UBRR0 + 1)} \quad (15.1)$$

$$BAUD = \frac{f_{CLK,I/O}}{8 \times (UBRR0 + 1)} \quad (15.2)$$

$$BAUD = \frac{f_{CLK,I/O}}{2 \times (UBRR0 + 1)} \quad (15.3)$$

Equations 15.1–15.3 can be instead solved for UBRR0 to determine the value to store in that register, given a desired baud rate. The register values are defined below.

$$UBRR0 = \frac{f_{CLK,I/O}}{16 \times BAUD} - 1 \quad (15.4)$$

$$UBRR0 = \frac{f_{CLK,I/O}}{8 \times BAUD} - 1 \quad (15.5)$$

$$UBRR0 = \frac{f_{CLK,I/O}}{2 \times BAUD} - 1 \quad (15.6)$$

Depending on the I/O clock frequency, and the fact that the UBRR0 register can only store integer values, there may be an error between the desired baud rate and the actual baud rate. This error is defined in Equation 15.7.

$$\text{error} = \frac{\text{actual baud rate}}{\text{desired baud rate}} - 1 \quad (15.7)$$

For example, consider running the ATmega328P in asynchronous normal mode with a 1 MHz clock at a baud rate of 28.8 kbps. The value that should be stored in UBRR0 can be calculated as

$$\text{UBRR0} = \frac{1 \times 10^6}{16 \times 28800} - 1 = 1. \quad (15.8)$$

A more exact value of the calculation is 1.17, but fractional values cannot be stored in a register. The integer value to be stored is 1. The actual baud rate that results from this register value is

$$\text{BAUD} = \frac{1 \times 10^6}{16 \times (1 + 1)} = 31250 \text{ bps.} \quad (15.9)$$

The error, calculated using Equation 15.7, is

$$\text{error} = \left(\frac{31250 \text{ bps}}{28800 \text{ bps}} - 1 \right) \times 100\% = 8.5\% \quad (15.10)$$

When the microcontroller is configured in synchronous primary mode, a clock signal will be generated on the XCK pin. When configured in synchronous secondary mode, a clock signal needs to be supplied to the XCK pin.

Transmitting Data

The USART can be configured to transmit data by setting the Transmitter Enable bit (TXEN0) in the USART Control and Status Register B (UCSR0B). (This configuration must occur after setting the baud rate in UBRR0.)

Prior to transmitting data, it is important to ensure that the transmit buffer is empty. This is accomplished by ensuring that the USART Data Register Empty flag (UDRE0) is set. Data transmission between 5 and 8 bits is initiated when data is written to the USART I/O Data Register (UDR0). This will send a data frame out using the TXD pin.

To send 9 bits of data, first ensure that the transmit buffer is empty. Then, write the ninth bit to the Transmit Data Bit 8 bit TXB80 in USART Control and Status Register B (UCSR0B). Finally, write the remaining low byte to the UDR0 register, which will initiate the transmission of the data frame using the TXD pin.

Receiving Data

The USART can be configured to transmit data by setting the Receiver Enable bit (RXEN0) in the USART Control and Status Register B (UCSR0B). (This configuration must occur after setting the baud rate in UBRR0.)

Data is received when the USART unit detects a start bit. Each subsequent bit in the frame will be sampled either using the baud rate (asynchronous mode) or on the clock signal input to the XCK pin (synchronous mode). The sampling will continue until the first stop bit is received. When data has been written into the receive buffer, the USART Receive Complete flag (RXC0) will be set in USART Control and Status Register A (UCSR0A). If interrupts are not used to determine if data is received, then polling that flag can be used to determine if new data is ready to be read out of the UDR0 register.

If 9 bits of data are to be received, the ninth bit must be read from the Receive Data Bit 8 bit (RXB80) before reading the remaining low byte of data from UDR0.

Data is buffered as it is received by the USART. If it's necessary to clear the buffer, simply continue to read the contents of UDR0 until the USART Receive Complete flag (RXC0) is cleared.

Receiver Errors

The USART on the ATmega328P has three receiver error flags that can be read in the USART Control and Status Register A (UCSR0A). Note that this register must be read **before** reading data from UDR0 or the Data Bit 8 bit in UCSR0B. (When writing to the UCSR0A register, ensure that the flag bits are written as zero.)

The receiver errors are listed below.

- A **frame error** (indicated by the Frame Error flag FE0) occurs when a stop bit is not recorded at the expected time.
- A **data overrun error** (indicated by the Data Overrun flag DOR0) occurs when the receiver cannot process data quickly enough. New data is arriving before previous data has been buffered or written into the UDR0 register.
- A **parity error** (indicated by the Parity Error flag UPE0) occurs when there is a parity error with the received data. This flag can only be set if parity check is enabled.

USART in SPI Mode

The USART module can be configured to run in SPI primary mode if so configured using the USART Mode Select bits (UMSEL0) in the US-

ART Control and Status Register C (UCSR0C). When using the USART in this manner, the bits in the USART Control and Status Registers B and C are modified to eliminate some of the features of the USART and align it with the SPI protocol. For example, frames only consist of 8 bits (it cannot be configured for 5, 6, 7, or 9-bit mode) and parity generation and detection is removed. When operating the USART in SPI mode, the registers give the option to set the clock polarity and phase, which is consistent with the description provided in the previous section in this chapter.

15.6 ATmega328P Two-Wire Interface (TWI)

Two-wire interface (TWI), also referred to as inter-integrated circuit (I^2C), is a serial protocol used to connect with one or more secondary devices (connected as shown in Figure 15.7, note the use of pull-up resistors on both signal lines) using only two I/O pins. One wire is a bidirectional data line called SDA, which indicates that TWI is capable of half-duplex communication. The other wire, SCL (also bidirectional), carries the clock signal, indicating that TWI is a synchronous communication system. Both of these pins reside in port C on the ATmega328P.

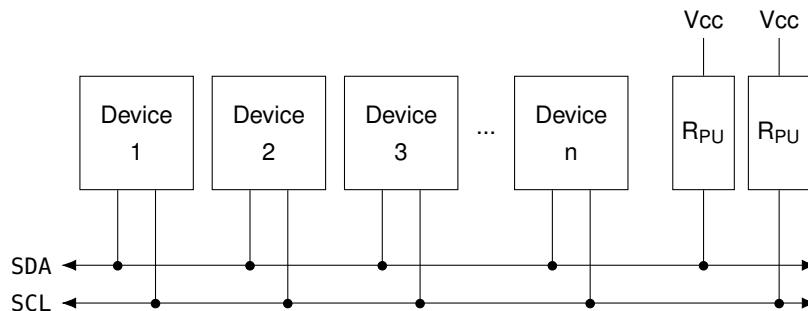


Figure 15.7: Device configuration using the TWI protocol.

TWI Modules

The TWI hardware consists of several modules that work together to operate the communication system. This overview is depicted schematically in Figure 15.8. The two pins SCL and SDA interface the microcontroller with external devices. The **bus interface unit** contains four units. TWDR, the data and address shift register, which contains the address or data bytes to be transmitted, or the address or data bytes that have been received. The START/STOP controller generates and detects START and STOP bits used for framing the received and transmitted data. Spike suppression filters out short bursts of

data that may otherwise interfere with serial communication. Finally, the arbitration detection unit monitors communications to ensure that only one primary is communicating at a time in a multi-primary system.

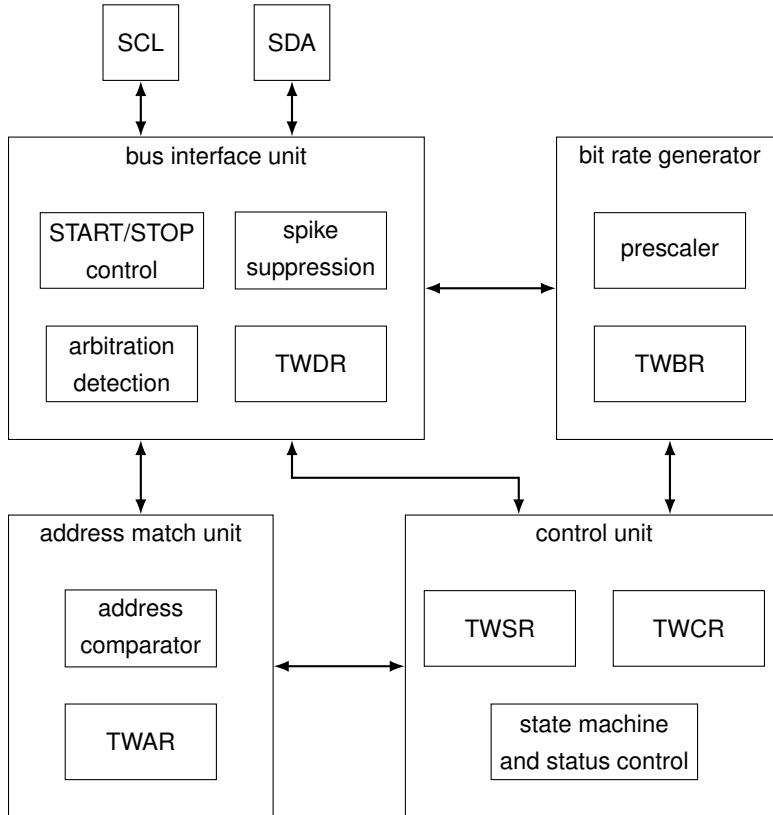


Figure 15.8: Overview schematic of the TWI module on the ATmega328P.

The **bit rate generator** controls the period of the clock signal on SCL when the device is operating in primary mode. The bit rate is stored in a register known as TWBR, and together with a prescaler bit they control the period given by Equation 15.11, where N is the value of the prescaler.

$$T_{SCL} = T_{CPU} \times [16 + (2 \times TWBR \times N)] \quad (15.11)$$

The **address match unit** checks to ensure that received address bytes match the 7-bit address given in the address register TWAR.

Finally, the **control unit** monitors the TWI bus and generates signals in response to the settings that have been detected by all of the other units. TWSR is the TWI status register, which contains data corresponding to the status of the most recently executed communication operation on the bus. The TWI control register, TWCR, contains the bus specification and settings.

TWI Communication Process

While TWI has the advantage of using fewer wires than SPI communication, the drawback is in the complexity of using the TWI protocol. Communication begins when the primary transmits a START bit followed by the 7-bit address of the secondary with which it intends to communicate and a bit indicating a read or write operation. (To be assigned a unique secondary address for a TWI-compatible device, a fee must be paid to NXP Semiconductors.) The corresponding secondary device (if it is properly connected to the interface bus) then sends an acknowledge bit, which the primary can use to ensure that it is indeed communicating with the correct device. The primary then continues to send a clock signal while either transmitting messages to the secondary or receiving data from the secondary. A STOP bit is transmitted when communication with the secondary is complete.

TWI Status Codes

Multiple status codes exist for all TWI communication modes (primary transmit, primary receive, secondary transmit, and secondary receive). These codes give information about the status of the bus and connected hardware. They can be used to generate error messages, or to stop program flow if proper handshaking and acknowledgement between primary and secondary has not been successful.

15.7 Practice Problems

1. Is the USART capable of full-duplex, half-duplex, or simplex communication?
full-duplex
2. Does the USART operate synchronously or asynchronously?
it can operate either synchronously or asynchronously
3. True or false: the SPI bus requires an external clock to operate.
FALSE
4. The secondary select line on the SPI bus is active _____.
LOW
5. Using the SPI bus, data is received and transmitted in chunks of how many bits?
8
6. How many specific secondary devices can be supported on the TWI bus?
 $2^7 = 128$

7. True or false: the TWI bus requires an external clock to operate. FALSE
8. Calculate the parity of the data byte 1101 1001. odd
9. Calculate the parity of the data byte 0101 0011. even
10. Generate an odd parity bit for the data byte 1101 0011. $P = 0$
11. Generate an even parity bit for the data byte 1101 0011. $P = 1$

16

Power Management and Sleep Modes

EMBEDDED SYSTEMS FREQUENTLY RUN ON BATTERIES. In system designs for applications when it is monetarily or physically difficult to frequently change batteries, it is important to design the project to consume as little power as possible. The ATmega328P is built from CMOS technology; CMOS devices use no power when they are not switching. There are sleep modes available that reduce power consumption by turning off the clock signal to peripheral units.

In addition, a power reduction register (PRR) can also be used to disable modules (by turning off the associated clock signal) that are not needed. The datasheet additionally recommends ensuring that any disconnected I/O pins have a defined voltage levels. For input pins, a float is not a defined voltage level. The simplest way to define a voltage level on unused input pins is to activate the internal pull-up resistor.

16.1 Sleep Modes

To enter a sleep mode, the Sleep Enable bit (SE) in the Sleep Mode Control Register (SMCR) must be set with the Sleep Mode Select bits (SM) configured as desired. This must be followed by a sleep command (SLEEP instruction in assembly). If an enabled interrupt corresponding to a wake-up source occurs when the microcontroller is in a sleep mode, the microcontroller will exit the sleep mode, halt for four clock cycles, and then continue executing the program code at the address following the SLEEP instruction.

If the brown-out detection (BOD) unit is enabled in the extended fuse byte, it is recommended to disable the BOD when the microcontroller is in a sleep mode. This can be accomplished using the MCU Control Register (MCUCR).

There are six sleep modes available on the ATmega328P, as shown in Table 16.1.

Name	clk _{CPU}	clk _{FLASH}	clk _{I/O}	clk _{ADC}	clk _{ASY}
Idle			×	×	×
ADC Noise Reduction				×	×
Power-down					
Power-save					×
Standby					
Extended Standby					×

Idle Mode

Idle mode stops the CPU clock but allows the SPI, USART, analog comparator, ADC, TWI, timer/counters, watchdog timer, and system interrupts to continue operating. Wake-up sources in idle mode include

- external and pin-change interrupts,
- TWI address match,
- timer/counter 2,
- SPM and EEPROM ready,
- ADC,
- WDT, and
- other I/O.

ADC Noise Reduction Mode

ADC noise reduction mode stops the CPU clock, I/O clock, and flash clock. The peripherals that can continue to run under this restriction are the ADC, external interrupts, TWI, timer/counter 2 (clocked asynchronously), and the watchdog timer. As is indicated from the name, shutting down superfluous clock signals leads to lower noise in the ADC, which leads to more accurate results. Wake-up sources in ADC noise reduction mode include

- external and pin-change interrupts,
- TWI address match,
- timer/counter 2,
- SPM and EEPROM ready,
- ADC, and
- WDT.

Table 16.1: Sleep modes on the ATmega328P, indicating the active peripheral clocks.

Power-Down and Power-Save Modes

Power-down mode shuts off all generated clocks, allowing only asynchronous modules to run. Wake-up sources in power-down mode include

- external and pin-change interrupts,
- TWI address match, and
- WDT.

Power-save mode is virtually identical to power-down mode, except that timer/counter 2, if enabled and clocked asynchronously, will continue running during sleep. Timer/counter 2 can additionally be used as a wake-up source in power-save mode.

Standby and Extended Standby Modes

When clocked from an external source, standby mode can be used; it is identical to power-down mode except that the external oscillator is kept running. Wake-up sources in standby mode include

- external and pin-change interrupts,
- TWI address match, and
- WDT.

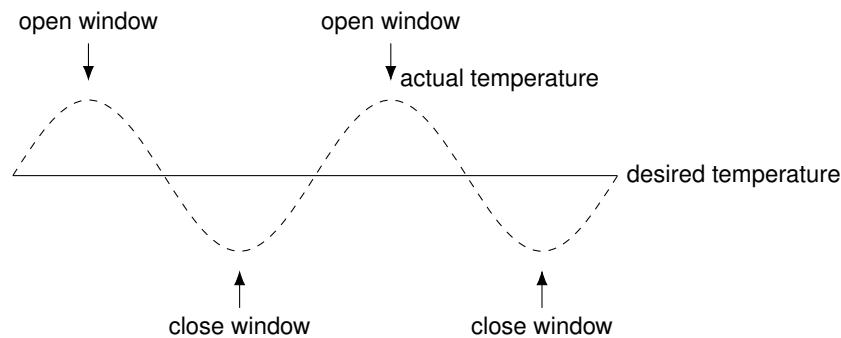
Similarly, extended standby mode is virtually identical to power-save except that an external source is used to clock the system and the external clock will continue running during sleep. Therefore, timer/counter 2 can be used as a wake-up source in this sleep mode.

17

Control Systems and Feedback

WHEN CONTROLLING A DEVICE, it is important to use some form of feedback to monitor the current status and see if changes need to be made.

Consider, for example, that you are sitting inside of a room with only a window. The window can only be opened and closed completely. If it gets too warm in the room, you open the window. When it gets too cold, you close the window. This leads to a temperature response as shown in Figure 17.1.



This is known as a negative feedback system. When you start to feel uncomfortable (negative feedback), you take an action that changes the surroundings to better suit your needs. However, in this situation, not only does the temperature fluctuate between extremes without ever settling on an ideal temperature, it also requires you to continue moving over to the window to open and close it. It would be better to have an automated system to take care of this functionality.

In a **closed-loop negative feedback** system, a setpoint is required. This is the value at which the system will attempt to keep the value of the output (temperature, in the case of the window example). When the measured output value is greater than the setpoint, action

Figure 17.1: The temperature response in the room with a single window.

will be taken to attempt to decrease the output value. When the measured output value is less than the setpoint, action will be taken to attempt to increase the output value. Instead of turning a system on and off completely (or opening a window fully or closing it fully), which leads to oscillating behavior, it is necessary to use a proportional feedback mechanism to more finely tune the desired changes in output value.

For example, a home central air-conditioning unit, when initially switched on in the middle of summer, may have a setpoint of 20 °C, but the surrounding temperature is 30 °C. This signals to the AC to turn on at great intensity. When turned on in the evening, the surrounding temperature of 25 °C signals to the AC to turn on but without as much intensity as before. This is known as proportional feedback because the amount of action requested at the output is proportional to the difference (hence the term negative feedback) between the setpoint value $r(t)$ and the actual value $y(t)$ (known as the process variable). This difference is also known as the error $e(t)$, and is defined by equation 17.1.

$$e(t) = r(t) - y(t) \quad (17.1)$$

17.1 Proportional Feedback Control

In proportional feedback control, the error is multiplied by a constant known as K_P . This term is then added to the process variable to create a new output, as defined by equation 17.2, where $y(t)$ is the value of the process variable (i.e. what the value is right now).

$$u(t) = y(t) + K_P e(t) \quad (17.2)$$

Proportional control error correction is depicted schematically in Figure 17.2.

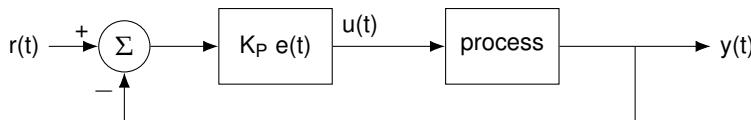


Figure 17.2: Block diagram of proportional control.

The constant K_P should be chosen with careful consideration. Values that are too low lead to sluggish, unresponsive feedback (this is known as an overdamped system). Values that are too high become unstable and can oscillate rapidly between values (this is known as an underdamped system). In the worst case, an underdamped system won't ever settle down to a proper value. Overdamped (red curve corresponding to $K_P = 0.1$) and underdamped (blue curve corresponding to $K_P = 1.8$) conditions are shown in Figure 17.3. When

K_P is just right, the output is stable and has no oscillations. This is known as a **critically damped** system and is the desired outcome of most control systems.

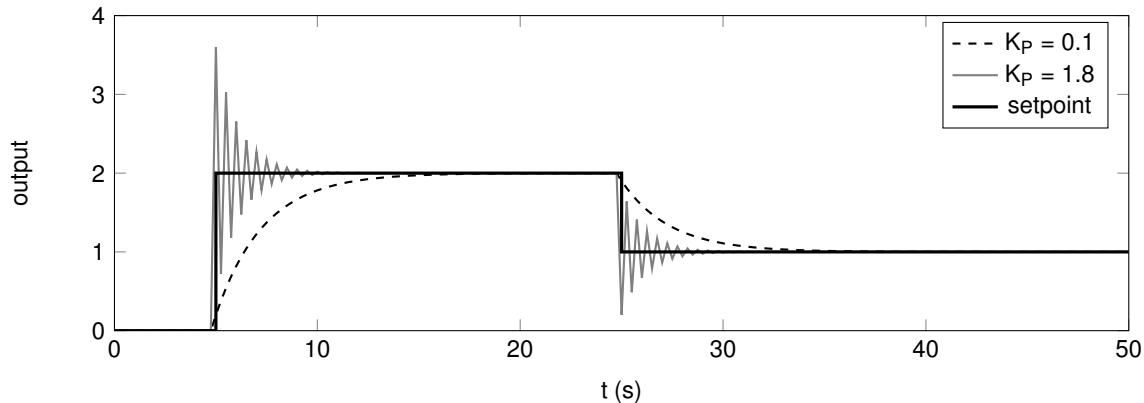


Figure 17.3: Proportional control output. The setpoint value is the thick black curve. The outputs are shown for a small value of K_P (dashed line, overdamped system) and a large value of K_P (gray solid line, underdamped system).

17.2 Proportional-Integral (PI) Control

In addition to the proportionality constant K_P that considers the current amount of error in the system, an integral term K_I can also be included that takes into account past values of the error. The past error is defined in Equation 17.3.

$$\text{past error} = \int_0^t e(\tau) d\tau \quad (17.3)$$

In a discrete time system such as a microcontroller, an integral becomes a sum, in which case the past error is defined by Equation 17.4. This sum can be considered to be very similar to the circular buffer discussed in Chapter 12. The larger the value of τ , the more past error is taken into account (and the more memory that is required to store all of the entries in the circular buffer). The smaller the value of τ , the less past error is taken into account.

$$\text{past error} = \sum_{n=0}^t e(\tau) \quad (17.4)$$

To affect the output in PI control, the process variable is now modified as shown in Equation 17.5.

$$u(t) = y(t) + K_P e(t) + K_I \int_0^t e(\tau) d\tau \quad (17.5)$$

PI control is depicted schematically in Figure 17.4.

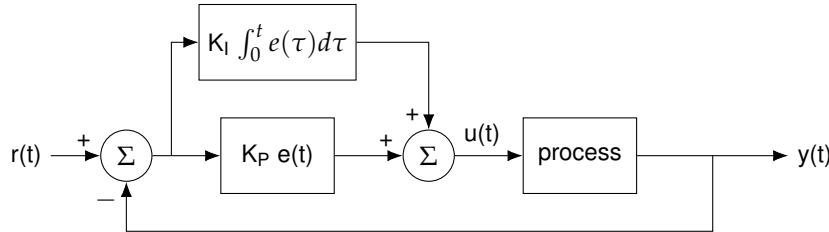


Figure 17.4: Block diagram of proportional-integral control.

This integral constant (which must be less than K_P) helps the feedback system achieve a steady-state value much quicker, based on the fact that it helps compensate for errors that have not yet been cleaned up by the linear proportionality constant K_P . However, a careful value of K_I still needs to be chosen, due to the fact that a poorly chosen value can still lead to an overdamped (K_I too low) or underdamped (K_I too high) output.

Figure 17.5 shows a PI control system with $K_P = 0.1$ that would ordinarily lead to a highly overdamped system which responds very slowly to change (indicated by the red curve). By adding integral control to the system, it is able to respond much quicker to change, with the drawback that it introduces overshoot.

An integral constant can be necessary to eliminate **steady-state error**, which refers to error in a system that persists even when the system has reached a stable state.

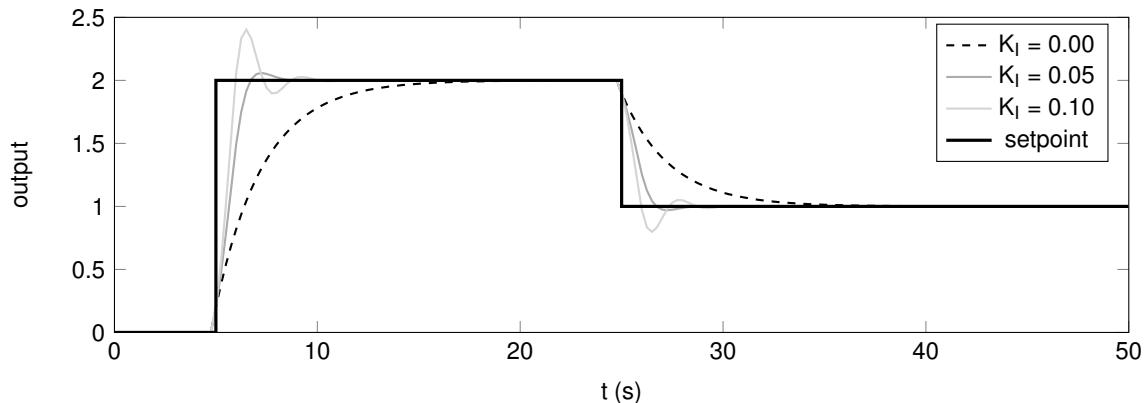


Figure 17.5: Proportional-integral control output. The setpoint value is the thick black curve. All outputs have $K_P = 0.1$. The dashed curve has no integral constant, the dark gray curve has $K_I = 0.05$, and the light gray curve has $K_I = 0.10$.

17.3 Proportional-Integral-Derivative (PID) Control

A full proportional-integral-derivative (PID) feedback system takes into account also anticipated future values of the error based on taking a derivative of the current error. The error term is now described in Equation 17.6.

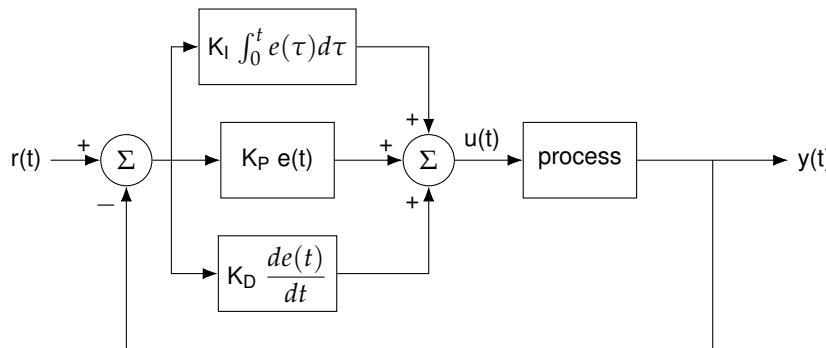
$$\text{future error} = \frac{d}{dt}e(t) \quad (17.6)$$

In a discrete time system such as a microcontroller, the derivative of the error can be calculated by determining how much the output has changed over time by taking the difference between current and past values and dividing by the elapsed time.

To affect the output in PID control, the process variable is now modified as shown in Equation 17.7.

$$u(t) = y(t) + K_P e(t) + K_I \int_0^t e(\tau) d\tau + K_D \frac{d}{dt}e(t) \quad (17.7)$$

PID control is depicted schematically in Figure 17.6.



Adding a derivative proportionality constant K_D may make the feedback mechanism much more sensitive to noise, and is only recommended in cases where noise will be minimum.

Example data with proportional control only, PI control, and PID control is shown in Figure 17.7.

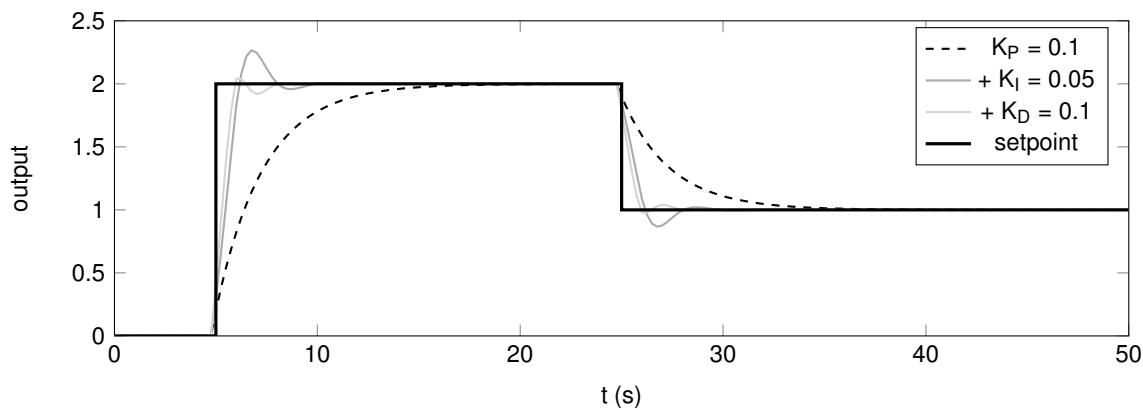


Figure 17.6: Block diagram of proportional-integral-derivative control.

Values for each of the constants must be chosen specifically for each application in which they are to be used. Changing any other part of the process may also require a change in the constants.

Figure 17.7: Proportional-integral-derivative control output. The setpoint value is the thick black curve. The dashed curve corresponds to only a proportional term with $K_P = 0.1$. The dark gray curve adds an integral term of $K_I = 0.05$. The light gray curve adds a derivative term of $K_D = 0.10$.

18

C Concepts for Microcontrollers

THE C PROGRAMMING LANGUAGE is an invaluable tool for programming microcontrollers using a high-level language. As discussed in [chapter 4](#), writing C programs for embedded systems is quite different from writing programs for general computing applications. This chapter is not intended to serve as an exhaustive reference for C; it will merely outline the most important concepts in programming C for microcontroller applications.

18.1 Standard Datatypes

All variables in C must be declared before usage. This declaration includes the datatype that the variable takes on, which indicates how much memory the microcontroller must use to store the variable and the minimum and maximum values it can take on. A list of C datatypes is given in Table [18.1](#).

Datatype	Memory	Data Range
Integer Datatypes		
char	8-bits	-128 to 127
unsigned char	8-bits	0 to 255
int	16-bits	-32,768 to 32,767
unsigned int	16-bits	0 to 65,535
long	32-bits	-2,147,483,648 to 2,147,483,647
unsigned long	32-bits	0 to 4,294,967,295
Floating-Point Datatypes		
float	32-bits	-3.4×10^{38} to 3.4×10^{38}
double	32-bits	-3.4×10^{38} to 3.4×10^{38}

Table 18.1: C datatypes used on the Arduino Uno using the Arduino IDE.

Integer datatypes are used to represent whole numbers. **Floating-point** datatypes are capable of representing fractional numbers as

well as numbers which are too large or too small to fit into the constraints of the integer datatypes. The two floating-point datatypes defined by the Arduino IDE, `float` and `double`, are both single-precision floating-point numbers. These numbers have a single sign bit, 8 exponent bits, and 23 fractional bits. The number they represent is

$$(-1^s) \times (1.f) \times 2^{(e-127)},$$

where s is the sign bit, f is the fractional number, and e is the exponent component. There are downsides to working with floating-point numbers. Floating-point arithmetic is very slow. This is because floating-point math requires custom subroutines to carry out mathematical operations using an instruction set built off of integer-based operations. This can yield strange or unexpected results (for example: $6.0/3.0$ may not be exactly equal to 2.0 using a floating-point operation) and limits the amount of useful precision of each floating-point value. In addition to this, each floating-point variable requires 4 bytes of data memory, regardless of the value it's representing.

Practice Problems

1. Indicate the datatype you would use for the following variables...

- | | |
|--|---------------|
| (a) ...temperature (in either degrees C or F). | int |
| (b) ...the number of days in a week. | unsigned char |
| (c) ...the number of days in a year. | unsigned int |
| (d) ...the number of months in a year. | unsigned char |
| (e) ...an address of 64K RAM space. | unsigned int |
| (f) ...the age of a person in years. | unsigned char |

18.2 Variable Scope and Keywords

The scope of a variable refers to what functions can access the variable. A variable defined within a function can only be accessed by that function. The following code

```

int main(void) {
    unsigned char j = 15;
    return 1;
}

void externalFunction(void) {
    // this function cannot access j
}

```

shows an example of variable scope. The variable `j` can only be accessed within the `main()` function in which it is defined.

A variable that is defined outside of all functions can be accessed by every function in the code, and is referred to as a **global variable**. Global variables should only be used in situations where it is necessary; otherwise, it is recommended that the scope of variables be limited by keeping them within the function in which they are to be used.

volatile Variables

When a compiler takes C code and translates it into assembly language, it attempts to optimize that code by leaving out any unused variables and converting unchanging variables to constants, among other optimizations. At times, it may appear that a global variable is unused by functions, especially in the case of an interrupt service routine (ISR). An ISR is never formally invoked (or called) by the `main()` function in C, and it may appear to a compiler as if any variables that are used within the ISR are unused (in which case it does not save them in memory) or unchanging (in which case it saves it in program memory as a constant value). By creating a `volatile` variable, the compiler knows not to discard the variable or to treat it as a constant. All datatypes can be saved as `volatile` variables.

static Variables

The keyword `static` in front of a variable refers to how long the variable is active in memory. Variables without this keyword are known as automatic, meaning that they come into existence when they are declared, and then expire whenever the function or control flow in which they reside has finished running. A `static` variable exists in memory for as long as the program is running. This means that, even if they are declared with a certain value inside of a function or control flow block, they will not be re-declared subsequent times that function or control flow block executes. This allows for the use of non-global variables that can change within a function or control

flow. Consider the following examples, which show the difference between automatic and static variables. Figure 18.1 depicts the difference between static and automatic variables.

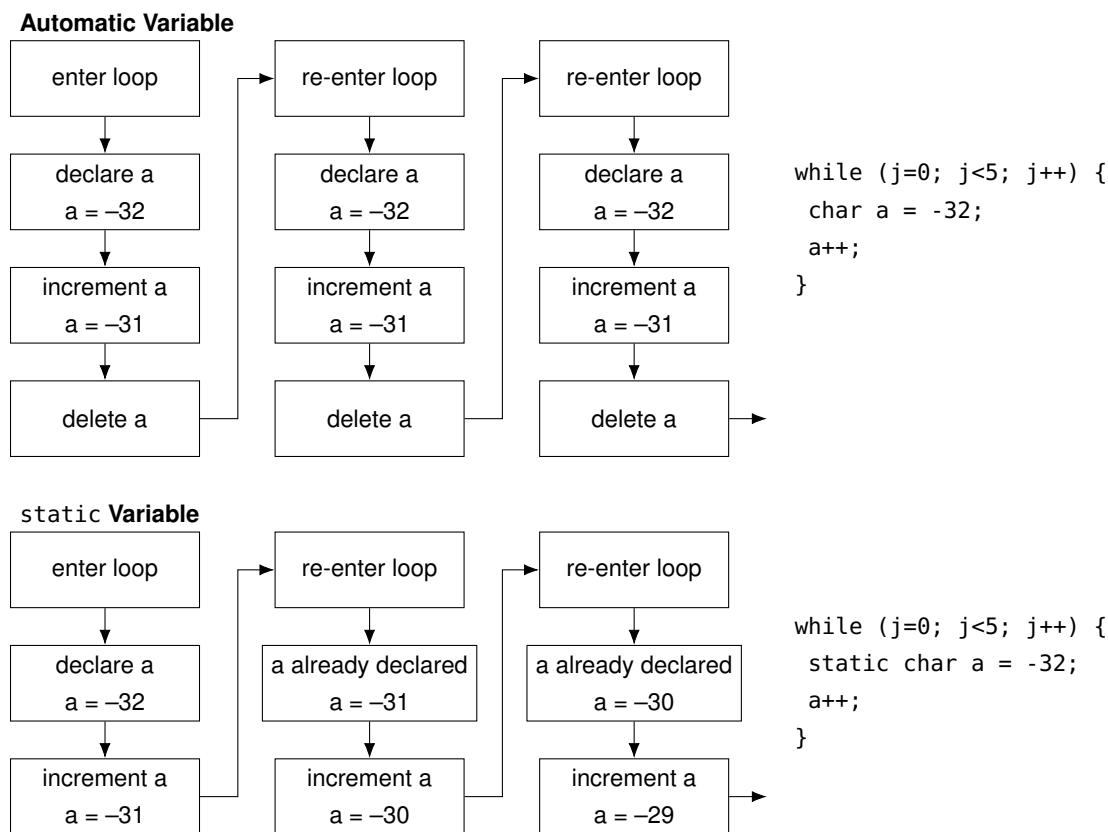


Figure 18.1: A flowchart representation of the difference between automatic and static variables.

const Variables

The keyword **const** is used to denote a variable whose value will not change. The keyword can be used with all datatypes. This keyword is convenient to use with data arrays to define the number of elements to be stored into the array. Using a variable allows the number to be stored in a single location which can easily be changed while debugging the software code. Additionally, if an array is defined using a variable to denote the number of elements, that variable **must** be a **const** variable. Most compilers will give a warning if attempting to assign a new value to a **const** variable inside of C code.

18.3 Arrays

When variables are related to each other, it may be prudent to store the data in an array. An array, which must be initialized with the

number of values to be given in the array, can store any type of variable (`char`, `int`, `long`, `float`). Each element in the array is defined by its index. Index numbers always go from 0 to $(n-1)$, where n is the number of values in the array. The syntax for defining an array is

```
datatype arrayName[sizeofArray];
```

Consider the array `myArray[6]` consisting of six `unsigned char` variables shown in Table 18.2.

<code>unsigned char myArray[6] = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6}</code>					
index:	0	1	2	3	4
value:	0xFC	0x60	0xDA	0xF2	0x66

Table 18.2: An array of six `unsigned char` variables.

To access an element of this array, a new variable can be assigned the value of one of the elements. For example, `unsigned char a = myArray[5]` will save the 6th element of the array into variable a. Note that a and `myArray[]` are of the same datatype. To save a new value into the array, an element inside of the array can be assigned a new value. For example, `myArray[1] = 13` saves the value of 13 into the 2nd element of the array `myArray[]`.

While arrays can be populated with any type of variable, **character arrays** using ASCII formatting (not to be confused with `char` arrays!) require a special precaution in that they must contain a null character (`\0`) at the end. Therefore, if they are assigned with the number of elements, that number must be $n+1$. Optionally, the number in square brackets can be left out.

Multi-Dimensional Arrays

Data can be stored into a multi-dimensional array. An array of integers, for example, `int a[n][m]` has n rows and m columns of data. This means that $n \times m$ elements can be stored into the array. Memory considerations must be taken into account before defining large arrays, which is especially true in multi-dimensional arrays, as memory can expand non-linearly with the addition of new elements. An example two-dimensional array is given in Table 18.3.

<code>unsigned char wholeNums[2][5] = {</code>					
<code> {1, 2, 3, 4, 5},</code>					
<code> {6, 7, 8, 9, 10} };</code>					
n:	0	1	2	3	4
m = 0:	1	2	3	4	5
m = 1:	6	7	8	9	10

Table 18.3: A two-dimensional array consisting of ten `unsigned char` variables.

18.4 Arithmetic and Assignment Operations

The assignment and arithmetic operators perform core mathematical functions in C. The assignment operator defines a value or variable by storing it in either program or data memory. The assignment operator should not be when changing values in a PORTx register! In this case, bitwise operators (defined below) should be used instead.

There are five arithmetic operations.

Addition and Subtraction

The addition operation (+) calculates the sum of two numbers and subtraction (-) calculates the difference of two numbers. Note that while addition is commutative ($a + b = b + a$), subtraction is not ($a - b \neq b - a$), so care must be taken in ensuring the correct position of each operand.

Because subtraction in particular can lead to negative results, it is important to ensure that the correct datatype (signed or unsigned) is used to store the result of any operations.

Multiplication and Division

The multiplication operation (*) calculates the product of two numbers and division (/) calculates the quotient of two numbers. Note that while multiplication is commutative ($a * b = b * a$), division is not ($a/b \neq b/a$), so care must be taken in ensuring the correct position of each operand.

When using integer datatypes, keep in mind that the microcontroller will truncate all values as integers. It is extremely important to keep this in mind when doing a sequence of math. Say a variable x must be multiplied by 50 and divided by 3. The multiplication step must occur first, and then division can occur second. This is because if division occurs first, the value will be truncated to an integer before multiplying by 50. This will lead to strange results. If $x = 10$, for example, the expected integer result of the operation will be 166. Dividing first leads to the following (unexpected) result.

```
(x / 3) * 50 =
10 / 3 // result is 3
3 * 50 // result is 150
```

Multiplying first leads to the following (expected) result.

```
(x * 50) / 3 =
10 * 50 // result is 500
500 / 3 // result is 166
```

A second consideration in multiplication is that the intermediate result when using the char or int datatypes is that the intermediate

result will be stored as an `int`. If the intermediate result may overflow that datatype, then the answer will be incorrect. To overcome this limitation, the letter `L` can be used to promote the intermediate result to a `long`. For example, say an integer variable `x` takes on values between 0 and 1023. This variable must be multiplied by 125 and then divided by 2 to obtain a desired result. The following example demonstrates what can happen if the `L` promotion is not used.

```
// if x = 1000, the result should be 62500
(125 * x) / 2 =
(125 * 1000) / 2 // intermediate result overflows
(59464) / 2 // result is 29732
```

The following example demonstrates what can happen if the `L` promotion is used.

```
// if x = 1000, the result should be 62500
(125L * x) / 2 =
(125L * 1000) / 2 // intermediate result does not overflow
(125000) / 2 // result is 62500
```

When the final result of a value will be a `long`, using `L` may be insufficient to prevent overflow. In this case, append a literal with `LL` to promote the intermediate result to a `long long` and avoid overflow.

Modulo

The modulo operation (%) calculates the remainder of a variable or value after performing division with some number. Note that modulo is not defined for negative numbers! Therefore, care should be taken in code to ensure that the modulo of a negative number is not calculated.

Practice Problems

- Calculate the result of the following operations.

- | | |
|---|-----------------------|
| (a) <code>unsigned char a = 0x05 + 0xF3;</code> | <code>a = 0xF8</code> |
| (b) <code>unsigned char a = 38 - 14;</code> | <code>a = 24</code> |
| (c) <code>unsigned char a = 52 - 100;</code> | <code>a = 208</code> |
| (d) <code>unsigned char a = 5 * 10;</code> | <code>a = 50</code> |
| (e) <code>unsigned char a = (25 * 10) / 2;</code> | <code>a = 125</code> |

```
(f) unsigned char a = (25 / 2) * 10;           a = 120
```

```
(g) unsigned char a = 36 % 8;                  a = 4
```

18.5 Bitwise Operations

While there are many types of operators used in programming (the assignment operator, arithmetic operators, logical operators, etc.), bitwise operations are used frequently in microcontroller programming. Bitwise means each bit in one variable is compared individually with the corresponding bit in the other variable. They are used to manipulate binary values, especially when only some values in a data byte need to be changed while leaving others alone. For this reason, they are used frequently when changing particular bits in PORTxn registers or when accessing the values stored in PINxn registers.

Bitwise AND: &

The bitwise AND operator, `&`, acts as a series of AND gates operating between the two operands, as shown in Figure 18.2. Because taking a logical AND with the number 0 results in a 0, the bitwise AND operation is used to selectively clear bits while leaving others alone.

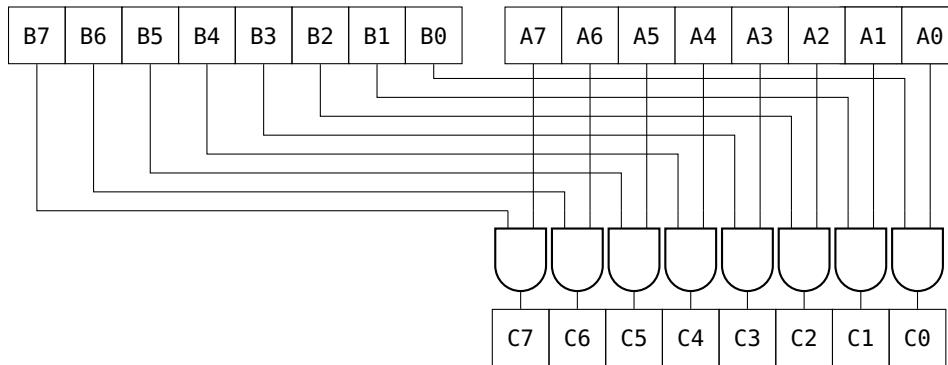


Figure 18.2: A bitwise AND operation takes the logical AND of each corresponding bit of the two operands.

Bitwise OR: |

The bitwise OR operator, `|`, acts as a series of OR gates operating between the two operands, as shown in Figure 18.3. Because taking a logical OR with the number 1 results in a 1, the bitwise OR operation is used to selectively set bits while leaving others alone.

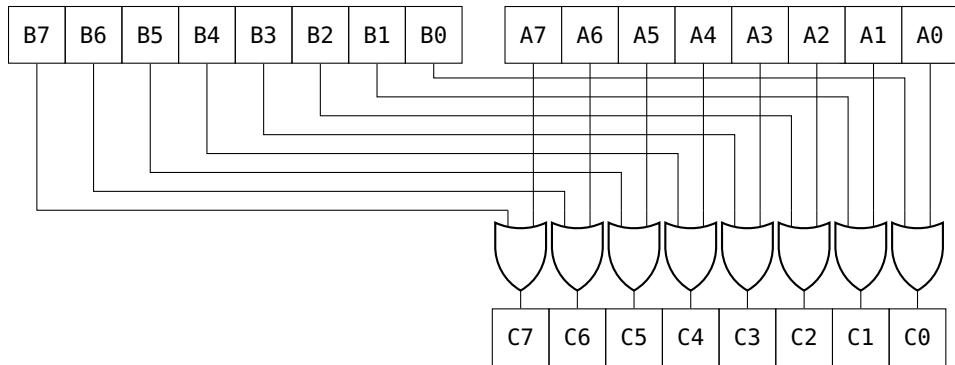


Figure 18.3: A bitwise OR operation takes the logical OR of each corresponding bit of the two operands.

Bitwise XOR: ^

The bitwise XOR operator, ^, acts as a series of XOR gates operating between the two operands, as shown in Figure 18.4. Because taking a logical XOR with the number 1 results in a toggle, the bitwise XOR operation is used to selectively toggle bits while leaving others alone.

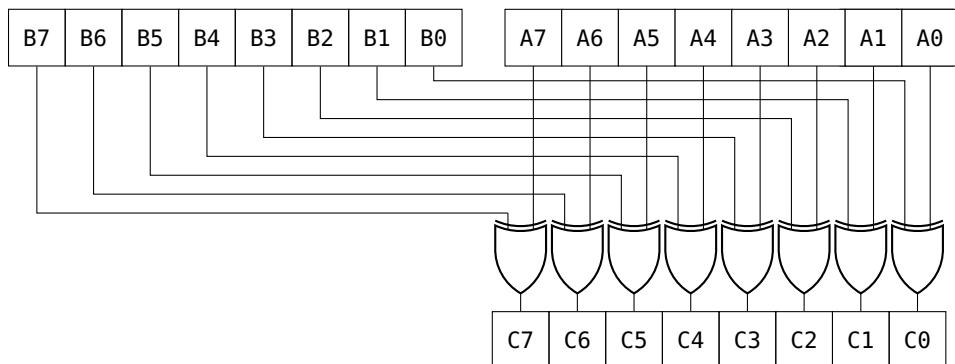


Figure 18.4: A bitwise XOR operation takes the logical OR of each corresponding bit of the two operands.

Bitwise NOT: ~

The bitwise NOT operator, ~, acts to invert each individual bit of the operand, as shown in Figure 18.5. The bitwise NOT operation is used to toggle all bits simultaneously.

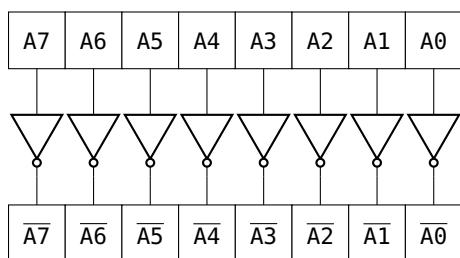


Figure 18.5: A bitwise NOT operation takes the logical NOT of each bit of the operand.

Bitshift Right: »

The bitshift right operator, `»`, shifts the operand to the right a specific number of places. When being bitshifted to the right, the number 0 will be shifted in to the most significant bit. Figure 18.6 shows an example bitshift right operation.

A	
A7	A6
A5	A4
A3	A2
A1	A0
A » 3	
0	0
0	A7
A6	A5
A4	A3

Figure 18.6: An example bitshift right operation.

Bitshift Left: «

The bitshift left operator, `«`, shifts the operand to the left a specific number of places. When being bitshifted to the left, the number 0 will be shifted in to the least significant bit. Figure 18.7 shows an example bitshift left operation.

A	
A7	A6
A5	A4
A3	A2
A1	A0
A « 5	
A2	A1
A0	0
0	0
0	0
0	0

Figure 18.7: An example bitshift left operation.

Practice Problems

1. Find the content of a after the following operations:

(a) $a = 0x37 \& 0xCA;$

$a = 0x02$

(b) $a = 0x37 | 0xCA;$

$a = 0xFF$

(c) $a = 0x37 ^ 0xCA;$

$a = 0xFD$

2. To selectively clear certain bits, which bitwise operator should be used?

bitwise AND

3. To selectively set certain bits, which bitwise operator should be used?

bitwise OR

18.6 Comparison and Boolean Operators

Comparison operators are used to compare two variables or values. They are useful when decisions need to be made based on how one variable compares to another. Comparison operators return Boolean values (TRUE, which is equal to 1, and FALSE, which is equal to 0) based on the result of the operation. The comparison operators are shown in Table 18.4, with $a = 50$ and $b = -10$ used to show examples.

Operator	Description	Example	Result
<code>==</code>	Equal To	<code>a == b</code>	FALSE
<code>!=</code>	Not Equal To	<code>a != b</code>	TRUE
<code>></code>	Greater Than	<code>a > b</code>	TRUE
<code>>=</code>	Greater Than or Equal To	<code>a >= b</code>	TRUE
<code><</code>	Less Than	<code>a < b</code>	FALSE
<code><=</code>	Less Than or Equal To	<code>a <= b</code>	FALSE

Table 18.4: Comparison operators with examples.

Boolean operators are used to make logical decisions based on the results of two or more comparison operations. Table 18.5 lists all of the Boolean operators. It is important to note that Boolean operators use two symbols (with the exception of NOT), whereas bitwise operators only use one.

Operator	Name	Description
<code>&&</code>	AND	$a \&\& b = \text{TRUE}$ if both a and b are TRUE $a \&\& b = \text{FALSE}$ if either a or b is FALSE
<code> </code>	OR	$a b = \text{TRUE}$ if either a or b is TRUE $a b = \text{FALSE}$ if both a and b are FALSE
<code>!</code>	NOT	$!a = \text{TRUE}$ if a is FALSE $!a = \text{FALSE}$ if a is TRUE

Table 18.5: Boolean operators.

Practice Problems

- Find result of the following operations:

$$(a) (-5 > 0) \mid\mid (3 <= 8) \quad \text{TRUE}$$

$$(b) (1 >= 2) \&\& (5 >= 0) \quad \text{FALSE}$$

(c) <code>(14 != 38) && (20 >= -10)</code>	TRUE
(d) <code>(0 == 3) !(5 != 8)</code>	FALSE

18.7 Compound Operators

While not necessary to use in a microcontroller program, compound operators provide a convenient shorthand for arithmetic and bitwise operations. Using `x = x + 1` as an example, the instructions are to find the number stored in the variable `x`, add the number `1` to it, and then store the result into the variable `x`. This code may have an inefficiency in that the location of variable `x` has to be looked up twice if the compiler does not recognize that two parts of the expression are identical. It can be replaced with a compound operation instead: `x += 1`.

Operator	Description	Example	Longhand Syntax
<code>++</code>	Increment	<code>a++</code>	<code>a = a + 1</code>
<code>--</code>	Decrement	<code>a--</code>	<code>a = a - 1</code>
<code>+=</code>	Compound Addition	<code>a+=4</code>	<code>a = a + 4</code>
<code>-=</code>	Compound Subtraction	<code>a-=12</code>	<code>a = a - 12</code>
<code>*=</code>	Compound Multiplication	<code>a*=10</code>	<code>a = a * 10</code>
<code>/=</code>	Compound Division	<code>a/=5</code>	<code>a = a / 5</code>
<code>%=</code>	Compound Modulo	<code>a%=3</code>	<code>a = a % 3</code>
<code>&=</code>	Compound Bitwise AND	<code>a&=0xFC</code>	<code>a = a & 0xFC</code>
<code> =</code>	Compound Bitwise OR	<code>a =0xFC</code>	<code>a = a 0xFC</code>
<code>^=</code>	Compound Bitwise XOR	<code>a^=0xFC</code>	<code>a = a ^ 0xFC</code>
<code>>>=</code>	Compound Bitshift Right	<code>a>>=3</code>	<code>a = a >> 3</code>
<code><<=</code>	Compound Bitshift Left	<code>a<<=6</code>	<code>a = a << 6</code>

Table 18.6: Compound operators with examples.

Practice Problems

- Find the final value of each variable after the compound operation has been executed.

(a) `unsigned char a = 38;`
`a/=9;` `a = 4`

(b) `char n = -9;`
`n++;` `n = -8`

- (c) `unsigned char j = 12;`
`j%7;` `j = 5`
- (d) `unsigned char x = 0xAC;`
`x^=0xE9;` `x = 0x45`
- (e) `unsigned char num = 22;`
`num&=199;` `num = 6`

18.8 Control Flow: Conditional

A microcontroller can execute instructions using three different paradigms: sequential, conditional, and iterative. The **sequential flow** of code is what normally occurs in C programs. The program execution starts at the first line of code and carries out every subsequent line in order.

At times, however, it is necessary to execute code in a different fashion depending on the value of one or more variables (for example, in [chapter 17](#) the need for turning up the heat based on the temperature of the room is discussed). This type of control flow is known as **conditional flow**. The two types of conditional flow functions are `if`, `if/else`, `if/else if/else`, and `switch case`.

if, if/else, and if/else if/else Statements

The `if` statement is a conditional statement that executes a different set of code based on the result of a Boolean and/or comparison operation. For example, when continuously polling a pushbutton to determine its state, the code may need to turn on an LED if the button was pushed. A flowchart showing how an `if` statement works is given in Figure 18.8.

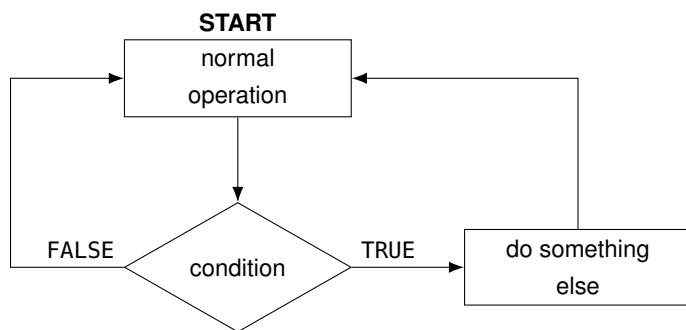


Figure 18.8: A flowchart depicting the operation of microcontroller during an `if` statement.

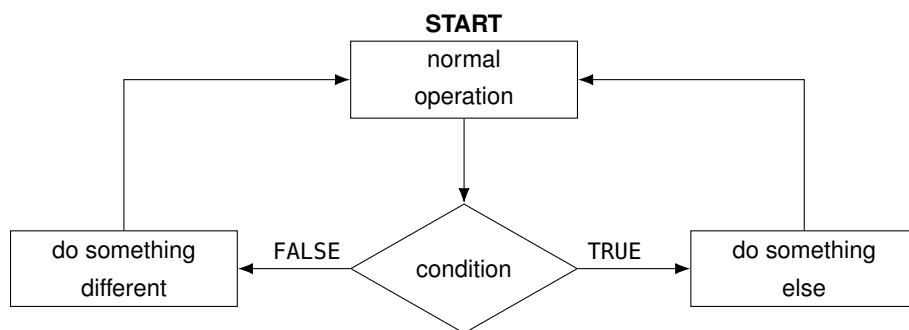
The syntax of this example is:

```

if (condition) {
    //do something else
}
// normal operation

```

It is possible that different blocks of code should be implemented based on the result of a condition. If these two different blocks of code should execute under mutually exclusive conditions, then `if/else` should be used. If the conditional logic of the `if` block is TRUE, then the `else` logic will not execute, and vice versa. A flowchart showing the operation an `if/else` flow is given in Figure 18.9.



The syntax of this example is:

```

if (condition) {
    //do something else
}
else {
    //do something different
}
// normal operation

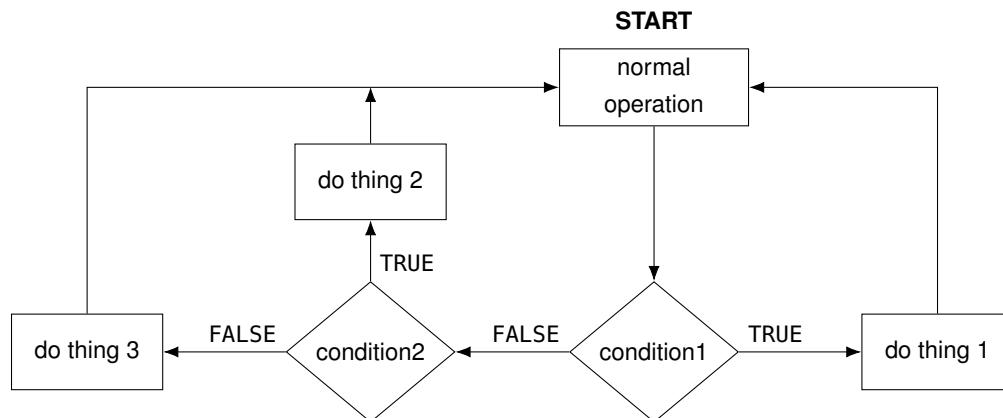
```

Once an `if` statement conditional is TRUE, no subsequent elements of that control flow sequence will be checked or executed. Therefore, if it's possible that two (or more) separate conditions can be TRUE, and both need some type of code to execute, then separate `if` statements should be used. However, if all conditions to be checked are mutually exclusive, then it is more efficient to code them as `if/else`, or `if/else if/else` (described below).

If more than two mutually exclusive results are possible based on the conditional statement, one or more `else if` block can be included in the conditional flow. An optional `else` can be included at the

Figure 18.9: A flowchart depicting the operation of microcontroller during an `if/else` statement.

conclusion to execute if the `if` and all `else if` conditions are FALSE. A flowchart showing the operation an `if/else if/else` flow is given in Figure 18.10.



The syntax of this example is:

```

if (condition1) {
    //do thing 1
}
else if (condition2) {
    //do thing 2
}
else {
    //do thing 3
}
// normal operation
  
```

Figure 18.10: A flowchart depicting the operation of microcontroller during an `if/else if/else` statement.

Switch Case

If a single variable needs to be compared to multiple different values, it can be more efficient to use a switch case instead of multiple repeated `if/else if` blocks. The flowchart for a switch case is given in Figure 18.11.

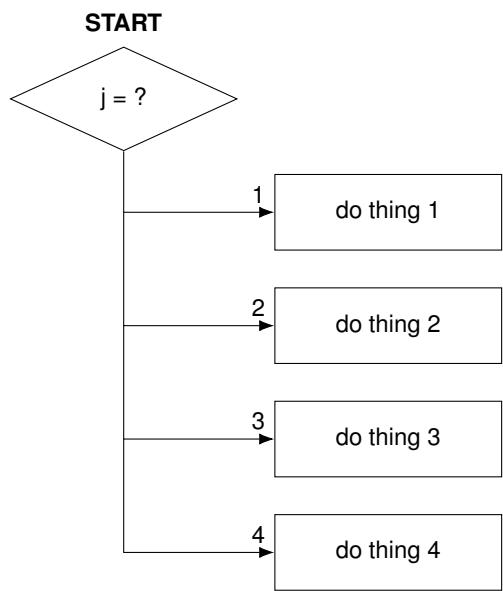


Figure 18.11: Flowchart and associated code for a switch case statement.

The syntax for the switch case shown in the flowchart in Figure 18.11 is:

```

switch(j) {
    case 1:
        // do thing 1
        break;
    case 2:
        // do thing 2
        break;
    case 3:
        // do thing 3
        break;
    case 4:
        // do thing 4
        break;
}
  
```

The `break` command instructs the compiler to exit the control flow at that point. If it is missing from the switch case, then any and all subsequent cases will execute in sequence until either the switch case ends or a `break` command is reached.

If different code should be executed in case all of the defined cases are FALSE, then a default case can be included at the end of the switch case. This does not require a `break` command. An example is:

```

switch(j) {
    case 1:
        // do thing 1
        break;
    case 2:
        // do thing 2
        break;
    default:
        // do something if 1 and 2 are FALSE
}

```

18.9 Control Flow: Iterative

Iterative flow of code is used to repeat identical (or nearly identical) functions a certain number of times (or infinitely). In [chapter 12](#), the concept of a circular buffer was discussed for taking rolling averages. An average requires all of the elements of an array to be summed together before being divided by the number of array entries. Rather than having one line of code for each array element (which would be inefficient on many levels), a loop can be used to repeat code a certain number of times before returning to sequential flow. Three types of iterative flow are the `for` loop, `while` loop, and `do/while` loop.

for Loop

`for` loops are best used when a given segment of code needs to be iterated a defined number of times (such as the example of summing up the circular buffer, which must be iterated a number of times equal to the number of elements in the array). The flowchart of a `for` loop is shown in [Figure 18.12](#).

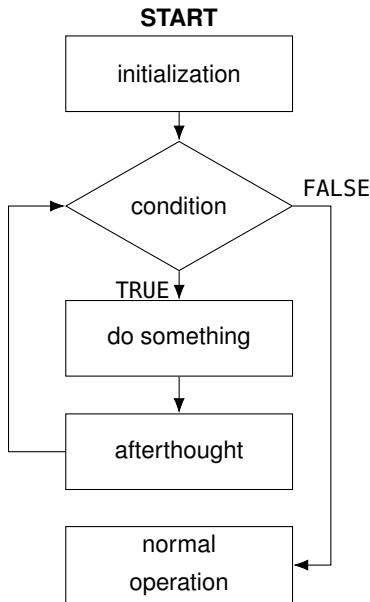


Figure 18.12: Flowchart indicating the usage of a `for` loop.

The syntax of a `for` loop,

```
for (initialization; condition; afterthought) {
    // code goes here
}
```

indicates that after an optional initialization, in which any variables required within the loop that aren't already declared are declared, a conditional statement is checked. If the conditional is TRUE, the contents of the loop will be executed. If the conditional is FALSE, the software will exit the loop. An optional afterthought is executed at the end of the loop every time the loop executes. The initialization, condition, and afterthought are used to control how many times the conditional logic will execute.

An infinite `for` loop is used when code needs to be repeated indefinitely, and can be written by leaving the initialization, condition, and afterthought blank as follows.

```
for ( ; ; ) {
    // this code will be repeated an infinite number of times
}
```

while Loop

A second type of loop is the `while` loop. It is recommended to use a `while` loop when code needs to be repeated an unknown number of

times until a particular condition is met.

The syntax of a while loop is:

```
// optional initialization
while (condition) {
    // code goes here
    // optional afterthought
}
```

An example flowchart of a while loop is given in Figure 18.13.

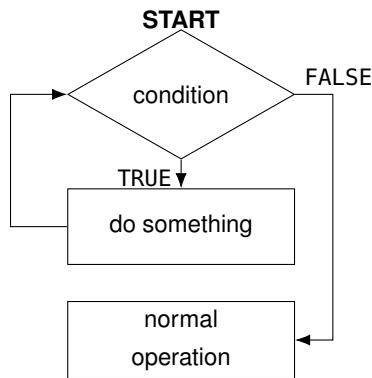


Figure 18.13: Flowchart indicating the usage of a while loop.

An infinite while loop can be written by making the condition equal to 1 as follows

```
while (1) {
    // this code will be repeated an infinite number of times
}
```

do/while Loop

A third type of loop is the do/while, which is very similar to a while loop, except that the body of the code is executed once before the condition is checked, as shown in the flowchart in Figure 18.14.

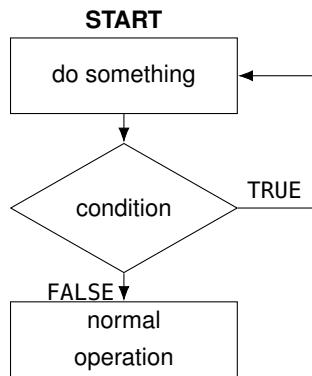


Figure 18.14: Flowchart indicating the usage of a do/while loop.

The syntax of a do/while loop follows.

```
// optional initialization
do {
  // code goes here
  // optional afterthought
} while (condition);
```

As with the while loop, to create an infinite do/while loop, the condition should be set to 1 as follows

```
do {
  // this code will be repeated an infinite number of times
} while (1);
```

Practice Problems

- Find the final value of the variable x after the loop has been executed.

(a) `for (unsigned char x = 0; x > 6; x++) { }` $x = 0$

(b) `unsigned int x = 1;
while (x <= 5) { x*=2; }` $x = 8$

(c) `unsigned int x = 0;
do { x++; } while(x <= 20);` $x = 21$

19

Assembly

ASSEMBLY CAN BE USED TO PROGRAM THE ATMEGA328P; it requires a detailed understanding of the AVR instruction set.¹ In order to access all of the information stored in general purpose (GP) registers, data memory, and program memory, an understanding of memory addressing (discussed in [chapter 7](#)) is fundamental. The instructions allowed on the microcontroller can be divided into several categories.

19.1 Data Transfer Instructions

Before any meaningful instructions can take place, data must be loaded into the GP registers. This data can come from program memory, data memory (including the I/O registers), another GP register, or it can be an immediate data value. Instructions that load data into GP registers can:

- move – copy a register or pair of registers into a GP register,
- load – load immediate data into a GP register; load data directly or indirectly (with or without displacement, post-increment, or pre-decrement) from data memory into a GP register; load data indirectly from program memory into a GP register,
- in – input data from an I/O register into a GP register, and
- pop – take data from the stack and save it into a GP register.

Data can also be stored from a GP register into memory. These instructions can:

- store – store data directly or indirectly (with or without displacement, post-increment, or pre-decrement) from a GP register into data memory; store data indirectly from a GP register into program memory,

¹ Atmel, "AVR Instruction Set Manual," November 2016.

- out – output data from a GP register into an I/O register, and
- push – take data from a GP register and save it in the stack.

Practice Problems

1. Which general purpose registers can be used with instruction LD? r0 – r31
2. Which general purpose registers can be used with instruction LDI? r16 – r31

19.2 Arithmetic and Logic Instructions

Arithmetic and logic instructions allow the microcontroller to add, subtract, and multiply. Data can be complemented using either one's or two's complement. Logical operations include AND, OR, and XOR. Some of these instructions have the option to be used with or without a carry and using signed, unsigned, or fractional values. All of these instructions are carried out within one or two clock cycles. Most all of these instructions affect flags in SREG.

Of the arithmetic and logic instructions, many of them either use direct, two register addressing or immediate addressing with direct, one register addressing. For example, the AND instruction is used to perform the logical AND operation on the contents of two GP registers (a source and destination) and save the results in the destination register. The ANDI instruction is used to perform the logical AND operation on the contents of one GP register with an immediate and save the results back into the GP register.

The logic instructions are essentially equivalent to bitwise operators. Table 19.1 shows the logical assembly instructions and their equivalent C counterparts.

Operation	Instruction(s)
AND	AND, ANDI, CBR
OR	OR, ORI, SBR
XOR	EOR
NOT	COM

Table 19.1: Logical instructions can be used to emulate bitwise operators.

Notice that not all assembly instructions allow for immediate addressing. Shifting bits (as accomplished using bitshift operators in C) is discussed below, as bitshifting is accomplished using bit manipulation instructions. It is also possible to set an individual bit

in a GP register using SBR, which is similar to using a bitwise OR with a single high bit. Similarly, an individual bit in a GP register can be cleared using CBR, which is similar to using a bitwise AND with a single low bit.

Addition and subtraction operations can take place with or without a carry. For example, ADD simply adds the contents of two GP registers (a source and a destination) and saves the result into the destination register. ADC adds the contents of two GP registers (a source and a destination) and the contents of the C flag in SREG, and then saves the result in the destination register.

The arithmetic and logic instructions also contain instructions that can:

- increment a register – $Rd \leftarrow Rd + 1$,
- decrement a register – $Rd \leftarrow Rd - 1$,
- clear a register – $Rd \leftarrow 0x00$, and
- set a register – $Rd \leftarrow 0xFF$.

Practice Problems

1. Determine the contents of the destination register (in decimal) after each subsequent operation.

- | | |
|-------------------|-----------|
| (a) LDI r17, 0x80 | r17 = 128 |
| (b) LDI r18, 0x15 | r18 = 21 |
| (c) COM r18 | r18 = 234 |
| (d) AND r18, r17 | r18 = 128 |
| (e) ORI r17, 0x03 | r17 = 131 |
| (f) EOR r17, r18 | r17 = 3 |
| (g) SER r18 | r18 = 255 |
| (h) SUB r18, r17 | r18 = 252 |
| (i) NEG r18 | r18 = 4 |
| (j) MUL r17, r18 | r17 = 12 |

19.3 Branch (Control Flow) Instructions

Just as there are conditional and iterative control flow operations in C, there are instructions that allow for these same processes in AVR assembly. It is important to have a solid understanding of branch instructions and memory addressing, as these are fundamental to carrying out conditional logic in assembly.

One subset of branch instructions are jump instructions, which tell the microcontroller to move to a different part of memory. Essentially, these instructions change the value of the program counter so that a different part of the program memory will be addressed on the next clock cycle. The two most important jump instructions in AVR assembly are

- **JMP** – (absolute jump) jumps to a location in program memory, this instruction can address up to 4 M of memory using a 22-bit operand; and
- **RJMP** – (relative jump) jumps to a relative location in program memory, this instruction moves forward or backward in memory space using a relative change in the program counter (note that RJMP is not always capable of addressing the entire memory space).

There are also call instructions, which are used for executing subroutines. Locations in program memory are saved into the stack to ensure that the code is able to return to the same point once it has finished executing the subroutine.

Whereas in C a conditional statement is made in syntax and curly brackets are used to denote the start and end of the conditionally or iteratively executed subroutine, in assembly this is done using logical instructions, compare instructions, and branch instructions.

Two important compare instructions are

- **CP** – compare, which compares the contents of two GP registers Rd and Rr, and
- **CPI** – compare with immediate, which compares the contents of a GP register Rd with a constant.

Branch instructions cause the code to jump to a different address in memory if a certain condition is met. There many possible branch conditions, including

- branch if a flag is set or clear – these instructions will jump to a different address in memory if a flag in SREG is either 0 or 1,
- branch if two numbers are equal (BREQ) – this instruction will jump to a different address in memory if two numbers are the same,

- branch if two numbers are not equal (BRNE) – this instruction will jump to a different address in memory if two numbers are not the same,
- branch if one number is greater than or equal to another – there are different instructions to use for signed (BRGE) and unsigned (BRSH) values, and
- branch if one number is less than another – there are different instructions to use for signed (BRLT) and unsigned (BRL0) values.

The argument of a branch instruction is the name of a subroutine in the code, which is defined using a colon, as follows:

```
;define a subroutine known as function1
function1:
    ;compare two registers, repeat function1 if they are equal
    CP r1, r2
    BREQ function1
```

Conditional Control Flow

As seen in [chapter 18](#), conditional control flow in C took on the form of **if statements** and **switch cases**. There isn't really a switch case equivalent in assembly; a series of mutually exclusive if statements can be used instead.

A simple if statement can be executed by using a compare instruction and a branch instruction. This works for checking to see if two values are equal, unequal, greater than, or less than, and making a decision based on the result.

The following example branches to the `conditiontrue` memory address if the value of register `r1` \geq `r2`, as shown in the flowchart in [Figure 19.1](#).

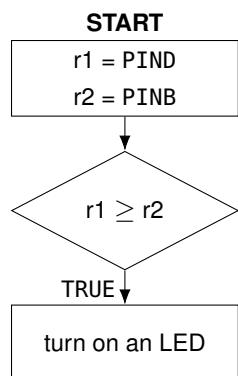


Figure 19.1: An example of conditional control flow (equivalent to a simple `if` statement) in assembly.

The values stored in the two GP registers come from the data stored in the PINB and PIND registers. If the condition is satisfied, an LED connected to pin 5 in port C is turned on. The associated assembly code is as follows:

```
;configure PORTC pin 5 as output
SBI DDRC, 5
;read contents of PINB and PIND
IN r1, PINB
IN r2, PIND
;compare r1 and r2
CP r1, r2
BRSH conditiontrue
;condition false
JMP end
;condition true subroutine
conditiontrue:
    SBI PORTC, 5
end:
```

This code is using unsigned values in registers r1 and r2, which is why the BRSH instruction is used. Had signed values been necessary, the BRGE instruction would have been used instead.

This code can easily be tailored to act as an **if / else** statement using the lines of code between the BRSH instruction and the conditiontrue subroutine. Series of nested if / else statements can be used to generate an **if / elseif / else** situation.

More complicated if statements can be written by cleverly configuring compares and jumps. The following example uses GP registers r16 and r17 (which both obtain their data from I/O registers). In C, the conditional statement inside of the if would have looked like `((r16 < 100) && (r17 >= 50))`. If the condition is satisfied, an LED connected to pin 5 in port C is turned on. Otherwise, the LED is turned off. The flowchart is shown in Figure 19.2.

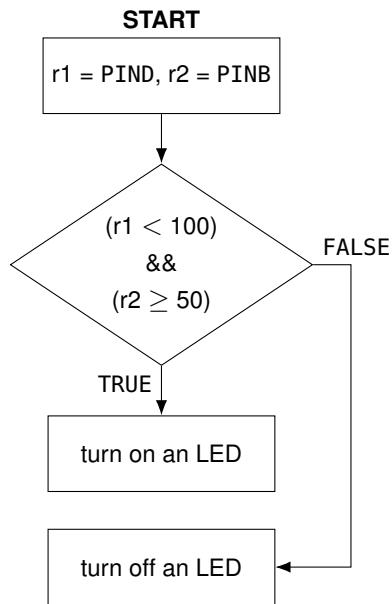


Figure 19.2: An example of conditional control flow (equivalent to a more complicated `if/else` statement) in assembly.

The associated assembly code is as follows:

```

;configure PORTC pin 5 as output
SBI DDRC, 5
;read contents of PINB and PIND
IN r16, PINB
IN r17, PIND
;compare r16 with 100
CPI r16, 100
;if r16 >=100, condition is FALSE
BRSH conditionfalse
;compare r17 with 50
CPI r17, 50
;if r17 < 50, condition is FALSE
BRL0 conditionfalse
conditiontrue:
    SBI PORTC, 5
    JMP end
conditionfalse:
    CBI PORTC, 5
end:
  
```

Note that GP registers must be at least 16 due to the use of instructions using immediate addressing. Note also that unsigned values were used in both GP registers.

Iterative Control Flow

The equivalent of a **for loop** can be accomplished in assembly by initializing a GP register to take on a particular value, executing the iterative code, executing the afterthought to the GP register, and then comparing the value of the GP register to the conditional statement.

The following example demonstrates how to increment the PORTB register from 0–10; perhaps it is connected to a 7-segment display via a BCD to 7-segment decoder. This code will enable the decoder to increment from 0–10. The flowchart is shown in Figure 19.3

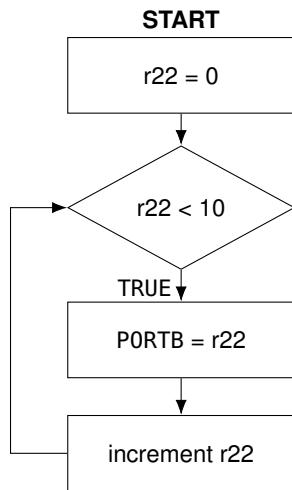


Figure 19.3: An example of iterative control flow (equivalent to a **for** loop) in assembly.

The associated assembly code is as follows:

```

;configure PORTB pins as outputs
LDI r23, 0x0F
OUT DDRB, r23
;initialization: clear register r22
CLR r22
loop1:
  OUT PORTB, r22
  ;afterthought: increment r22
  INC r22
  ;conditional: compare r22 with 10, repeat if it's lower
  CPI r22, 10
  BRL0 loop1
  
```

Note the use of GP registers greater than 16 for the instructions using immediate addressing (LDI and CPI).

As with C code, the for configuration is used when a segment code should be iterated a given number of times before returning

to normal operation. When a condition needs to be met before executing a segment of code, a **while** or **do/while** configuration should be used. The assembly code will still use compare and branch instructions, the only difference from the for configuration will be the ordering of each exact instruction.

The following example turns on an LED (connected to port C pin 0) if a toggle switch (connected to port B pin 0) is turned on. Otherwise, if the toggle switch is turned off, the LED turns off. The flowchart is shown in Figure 19.4.

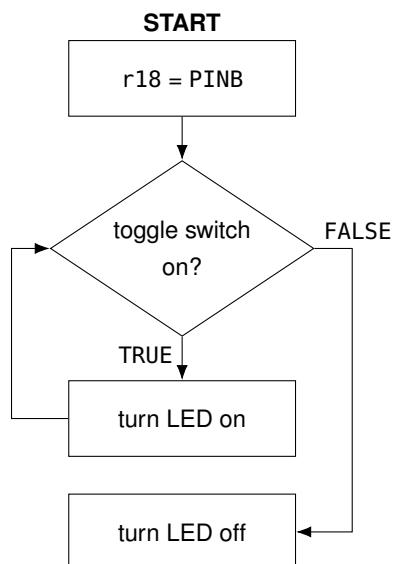


Figure 19.4: Flowchart indicating the usage of a while loop.

The assembly code follows

```

SBI DDRC, 0
main:
JMP checktrue
conditiontrue:
  SBI PORTC, 0
checktrue:
  IN r18, PINB
  ANDI r18, 0x01
  CPI r18, 0x01
  BREQ conditiontrue
CBI PORTC, 0
JMP main
  
```

A do/while loop executes the contents of the loop first before checking the conditional statement. In this case, the **JMP checktrue** instruction can be removed to simulate a do/while loop.

Control Flow and SREG

Compare and branch instructions operate by determining whether or not a particular flag was set in SREG. For example, conditional logic that would have been represented as `if (a == b)` in C code is carried out using a compare instruction on two GP registers, and then a branch instruction if the two registers are equal (BREQ), as follows:

```
CP r1, r2
BREQ subroutine1
```

The compare instruction executes the operation $r1 - r2$. If the result is zero (indicating that the two values are equal), the Z flag in SREG will be set. Therefore, the BREQ instruction checks to see if the Z flag in SREG had been set as a result of the previous instruction and uses that to decide whether or not to change the value of the program counter. The flags that are checked for several logical comparisons are shown in Table 19.2.

Comparison	Instruction	Flag
$a == b$	BREQ	$Z = 1$
$a != b$	BRNE	$Z = 0$
$a >= b$	BRSH (unsigned)	$C = 0$
$a >= b$	BRGE (signed)	$N \oplus V = 0$
$a < b$	BRL0 (unsigned)	$C = 1$
$a < b$	BRLT (signed)	$N \oplus V = 1$

Table 19.2: Branch instructions check the status of flags in SREG.

Because flag checks are how the branch instructions know whether or not to change the value of the program counter, it can be seen why there are no "greater than" or "less than or equal to" instructions. With unsigned values, a greater than situation would occur if $Z = 0$ AND $C = 0$, so both flags would need to be logically compared. It is possible to do this in assembly manually, but it is much easier to change the value to compare with. Similarly, with unsigned values, a less than or equal to situation would occur if $C \text{ XOR } Z$ is TRUE.

Practice Problems

- Determine the contents of SREG after each subsequent operation.

(a) LDI r18, 20 0x00

(b) LDI r19, 13 0x00

(c) CP r18, r19	0x20
(d) CPI r19, 200	0x01
(e) NEG r18	0x35
(f) LDI r18, 220	0x00
(g) LDI r19, 57	0x00
(h) ADD r18, r19	0x21
2. Which instruction(s) will branch if Z = 1?	BREQ
3. Which instruction(s) will branch if Z = 0?	BRNE
4. Which instruction(s) will branch if C = 1?	BRCS, BRL0
5. Which instruction(s) will branch if C = 0?	BRCC, BRSH

19.4 Bit Manipulation Instructions

Bit manipulation instructions consist of instructions that allow registers to be manipulated on the bit level. This consists of shifting, swapping, setting, and clearing instructions.

Shift instructions come in a few flavors: logical shift, arithmetic shift, and rotate through carry. A **logical shift** is equivalent to bitshift operations in C. When a register is logically shifted right (using the LSR instruction), the MSB is replaced with a 0. Assuming that the result does not overflow, this operation is equivalent to multiplying a binary value (signed or unsigned) by two. When a register is logically shifted left (using the LSL instruction), the LSB is replaced by a 0. A logical shift right is equivalent to dividing an unsigned binary value by two. While C code is able to perform multiple bitshifts at a time (for example, to bitshift a value 3 times to the right, `>>3` is used), each shift instruction in assembly executes a single time. Iterative control flow can be used to execute multiple bit shifts.

In an **arithmetic shift** right (which is the ASR instruction, no such arithmetic shift left exists as it would be equivalent to logical shift

left), the sign bit is shifted in to the MSB of the register. This is equivalent to dividing a signed or unsigned binary value by two by preserving the sign bit.

Rotate through carry instructions can occur to the left (using the ROL instruction) or right (using the ROR instruction), and execute a rotation (where the register wraps around so the MSB is moved to the LSB, or vice versa) through the previous value of the carry bit. A rotate left through carry is depicted graphically in Figure 19.5.

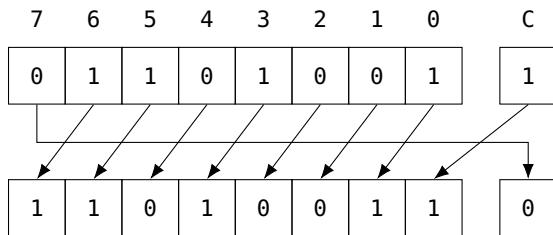


Figure 19.5: Rotate left through carry.

A rotate right through carry is depicted graphically in Figure 19.6. These instructions are particularly useful when dealing with values that are stored in two registers (i.e., the numbers are greater than 8 bits). When rotating, the value that is rotated out is saved into the carry flag of SREG and is therefore ready to be rotated in to the next byte without the need for additional processing.

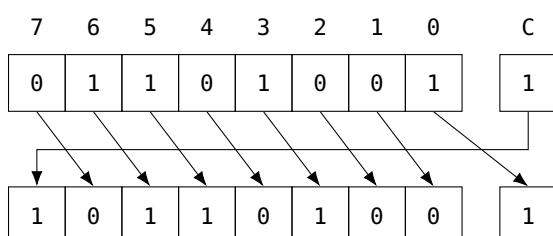


Figure 19.6: Rotate right through carry.

The SWAP instruction swaps the high and low nibbles on the operand register.

The T flag in SREG is a bit used to store a single bit of data. Storing data to this flag is accomplished using the BST instruction, while retrieving data from this flag is accomplished using the BLD instruction.

The last set of bit manipulation pertain to setting and clearing bits. Bits can be set and cleared individually in an I/O register using SBI and CBI, individual bits in SREG can be set and cleared using BSET and BCLR, and each individual flag in SREG can be set or cleared using one of the many instructions that accomplish those tasks. Individual bits can be cleared or set in a GP register by using ANDI or ORI similar to bitwise AND and bitwise OR operations in C.

19.5 *Miscellaneous Instructions*

The miscellaneous instructions on the ATmega328P are **BREAK** (break), which is used by the on-chip debug system; **NOP** (no operation), which spends one clock cycle doing nothing; **SLEEP** (sleep), which sets the device into sleep mode; and **WDR** (watchdog reset), which resets the watchdog timer.

Index

- alternate pin functions, 74
- analog comparator, 81, 87
- analog to digital conversion, 77
- analog to digital converter, 80
 - flash, 80
 - pipelined, 83
 - resolution, 86
 - successive approximation register, 82
 - trigger source, 86, 112
- Arduino Uno, 17
- arithmetic and logic unit, 14, 31, 59
- arithmetic operator, 164
- array, 162
- assembly language, 19, 62, 181
 - arithmetic and logic instructions, 182
 - bit manipulation instructions, 191
 - branch instructions, 184
 - call instructions, 184
 - data transfer instructions, 181
 - jump instructions, 184
 - miscellaneous instructions, 193
- assignment operator, 164
- ATmega328P, 17, 65
- automatic variable, 161
 - bit addressing, 55
 - bitwise operator, 166, 182
 - Boolean operator, 169
 - bus, 14
- C programming language, 159
 - arithmetic operator, 164
 - array, 162
 - assignment operator, 164
- automatic variable, 161
- bitwise operator, 166
- Boolean operator, 169
- comparison operator, 169
- compound operator, 170
- const variable, 162
- datatype, 159
- do/while loop, 178
- for loop, 175
- if/else statement, 172
- operator
 - arithmetic, 164
 - assignment, 164
 - bitwise, 166
 - Boolean, 169
 - comparison, 169
 - compound, 170
- static variable, 161
- switch case, 174
- variable
 - automatic, 161
 - const, 162
 - scope, 160
 - static, 161
 - volatile, 161
- variable scope, 160
- volatile variable, 161
- while loop, 177
- calibration
 - datasheet, 95
 - multiple-point, 96
 - one-point, 96
 - sensor, 95
- central processing unit, 14, 27
- circular buffer, 98
- clock, 107
- ceramic, 107
- crystal, 107
- multiplier, 108
- prescaler, 108
- RC circuit, 108
- closed-loop negative feedback control, 153
- comparison operator, 169
- compiler, 20
- compound operator, 170
- computer, 13
- conditional control flow, 171, 185
- const variable, 162
- control flow, 184
 - conditional, 171, 185
 - iterative, 175, 188
- control systems, 153
- control unit, 14
- data memory, 34, 46, 181
- data memory addressing, 51
- datasheet calibration, 95
- datatype, 159
- debugging, 24
- design tools, 23
- digital to analog conversion, 88
- do/while loop, 178
- duty cycle, 116
- embedded system, 16
- encoding, 80
- expanding opcodes, 30
- extended fuse byte, 68
- external interrupts, 105
- fast pulse-width modulation, 118

- feedback control, 153
 - closed-loop, 153
 - proportional, 154
 - proportional-integral, 155
 - proportional-integral-derivative, 156
- firmware, 14
- flash analog to digital converter, 80
- flash memory, 21, 45
- for loop, 175
- fuse byte, 68
 - extended, 68
 - high, 69
 - low, 69
- general purpose register, 31, 46, 181
 - addressing, 47
- Harvard architecture, 27
- high fuse byte, 69
- high-level language, 19, 159
- I/O pin, 71
- I/O register, 31, 46, 71
 - addressing, 50
- if/else statement, 172
- immediate addressing, 49, 181
- input capture unit, 126
- input device, 14, 59, 91
- instruction decoder, 31
- integrated circuit, 15
- interrupt, 101
 - external interrupts, 105
 - interrupt flags, 105
 - ISR, 102
 - masking, 102
 - pin change interrupts, 105
 - reset, 106
 - timed, 110
- interrupt service routine, 102
- iterative control flow, 175, 188
- low fuse byte, 69
- machine instructions, 28, 59, 181
- machine language, 19, 62
- memory, 13, 34, 45
 - addressing, 47
 - bit, 55
 - data memory, 51
 - general purpose register, 47
- I/O register, 50
 - immediate, 49
 - program memory, 54
- flash, 21, 45
- non-volatile, 36, 45
 - random-access, 21, 34
 - read-only, 21, 37
 - volatile, 34, 46
- memory address register, 31
- memory addressing, 47
- microcomputer, 15
- microcontroller, 13, 27, 59, 65
- microprocessor, 15
- multiple-point calibration, 96
- non-volatile memory, 36, 45
- one-point calibration, 96
- opcode, 29, 59
- output compare unit, 112
- output device, 14, 59
- parallel in / serial out register, 33
- parallel in/parallel out register, 32
- parity detection, 135
- phase- and frequency-correct pulse-width modulation, 125
- phase-correct pulse-width modulation, 121
- pin change interrupts, 105
- pipelined analog to digital converter, 83
- polling, 101
- power management, 149
- prescaler, 108
- program counter, 14, 33
- program memory, 36, 45, 181
- program memory addressing, 54
- programming, 18, 159
- proportional feedback control, 154
- proportional-integral feedback control, 155
- proportional-integral-derivative feedback control, 156
- pull-up resistor
 - internal, 74
- pulse-width modulation, 116
 - average voltage, 117
 - duty cycle, 116
 - fast, 118
 - frequency, 117
- phase- and frequency-correct, 125
- phase-correct, 121
- quantization, 78
 - error, 79
- random-access memory, 21, 34
- read-only memory, 21, 37
- reduced instruction set computing, 28
- register, 31
 - general purpose, 31, 46
 - I/O, 31, 46, 71
 - memory address, 31
 - parallel in / serial out, 33
 - parallel in/parallel out, 32
 - serial in/parallel out, 32
 - serial in/serial out, 32
 - status, 31, 41
- register transfer language, 62
- registers, 59
- reset, 106
- resolution
 - analog to digital converter, 79
- sampling, 78
- sensor, 77, 91
 - calibration, 95
 - noise, 97
- serial communication, 133
 - asynchronous, 134
 - full-duplex, 134
 - half-duplex, 134
 - parity detection, 135
 - serial peripheral interface, 136
 - simplex, 134
 - synchronous, 134
 - two-wire interface, 145
 - universal synchronous/asynchronous receiver/transmitter, 140
- serial in/parallel out register, 32
- serial in/serial out register, 32
- serial peripheral interface, 136
- sleep mode, 149
- software, 14
- SREG, 41
- stack, 55, 181
 - collision, 57
 - overflow, 56
 - underflow, 56
- static variable, 161

- status register, 31, 41, 190
- successive approximation register
 - analog to digital converter, 82
- switch case, 174
- timer/counter, 110
 - clear timer on compare match mode, 114
 - input capture unit, 126
- normal mode, 113
- output compare unit, 112
- timer/counter 0, 110
- timer/counter 1, 110
- timer/counter 2, 110
- two-wire interface, 145
- universal synchronous/asynchronous receiver/transmitter, 140
- variable scope, 160
- volatile memory, 34, 46
- volatile variable, 161
- von Neumann architecture, 27
- watchdog timer, 129
- while loop, 177



MRCET CAMPUS

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY (AUTONOMOUS INSTITUTION - UGC, GOVT.OF INDIA)

Affiliated to JNTUH; Approved by AICTE, NBA-Tier 1 & NAAC with A-GRADE | ISO 9001:2015
Maisammaguda, Dhulapally, Komapally, Secunderabad - 500100, Telangana State, India

LECTURE NOTES

ANALOG & DIGITAL ELECTRONICS

2021-22 (R20)



MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

I Year B. TECH

L/T/P/C

3/-/-3

(R20A0401) ANALOG & DIGITAL ELECTRONICS

COURSE OBJECTIVES:

The main objectives of the course are:

1. To familiarize with the principle of operation, analysis and design of pn junction diode.
2. To study the construction of BJT and its characteristics in different configurations.
3. To study the construction and characteristics of JFET and MOSFET.
4. To study basic number systems codes and logical gates.
5. To introduce the methods for simplifying Boolean expressions and design of combinational circuits.

UNIT-I

P-N Junction diode: Qualitative Theory of P-N Junction, P-N Junction as a diode, diode equation, volt-ampere characteristics, temperature dependence of V-I characteristics, ideal versus practical, diode equivalent circuits, Zener diode characteristics.

UNIT-II

Bipolar Junction Transistor: The Junction transistor, Transistor construction, Transistor current components, Transistor as an amplifier, Input and Output characteristics of transistor in Common Base, Common Emitter, and Common collector configurations. α and β Parameters and the relation between them, BJT Specifications.

UNIT-III

FIELD EFFECT TRANSISTOR: JFET-Construction, principle of Operation, Volt–Ampere characteristics, Pinch- off voltage. Small signal model of JFET. FET as Voltage Variable Resistor, Comparison of BJT and FET. MOSFET- Construction, Principle of Operation and symbol, MOSFET characteristics in Enhancement and Depletion modes.

UNIT IV:

Number System and Boolean Algebra: Number Systems, Base Conversion Methods, Complements of Numbers, Codes- Binary Codes, Binary Coded Decimal, Unit Distance Code, Digital Logic Gates (AND, NAND, OR, NOR, EX-OR, EX-NOR), Properties of XOR Gates, Universal Gates, Basic Theorems and Properties, Switching Functions, Canonical and Standard Form.

UNIT-V

Minimization Techniques: The Karnaugh Map Method, Three, Four and Five Variable Maps, Prime and Essential Implications, Don't Care Map Entries, Using the Maps for Simplifying, Multilevel NAND/NOR realizations.

Combinational Circuits: Design procedure – Half adder, Full Adder, Half subtractor, Full subtractor, Multiplexer/Demultiplexer, decoder, encoder, Code converters, Magnitude Comparator.

TEXT BOOKS:

1. Integrated Electronics Analog Digital Circuits, Jacob Millman and D. Halkias, McGrawHill.
2. Electronic Devices and Circuits, S.Salivahanan, N.Sureshkumar, McGrawHill.
3. M. Morris Mano, Digital Design, 3rd Edition, Prentice Hall of India Pvt. Ltd., 2003

REFERENCE BOOKS:

1. Electronic Devices and Circuits,K.Lal Kishore B.SPublications
2. Electronic Devices and Circuits, G.S.N. Raju, I.K. International Publications, New Delhi, 2006.
3. John F.Wakerly, Digital Design, Fourth Edition, Pearson/PHI, 2006
4. John.M Yarbrough, Digital Logic Applications and Design, Thomson Learning, 2002.
5. Charles H.Roth. Fundamentals of Logic Design, Thomson Learning, 2003.

COURSE OUTCOMES:

After completion of the course, the student will be able to:

1. Understand the principal of operation, analysis and design of pn junction diode.
2. Understand the construction of BJT and its characteristics in different configurations.
3. Understand the construction and characteristics of JFET and MOSFET.
4. Understand basic number systems codes and logical gates.
5. Understand the methods for simplifying Boolean expressions and design of combinational circuits

PROGRAM OUTCOMES (POs)

Engineering Graduates will be able to:

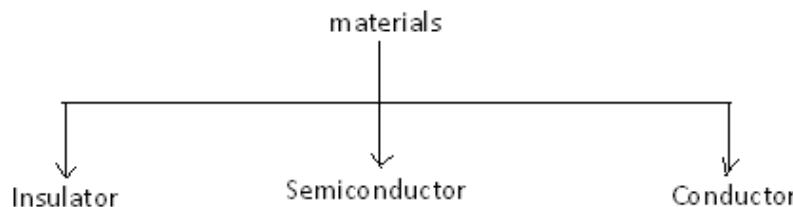
1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design / development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multi disciplinary environments.
12. **Life- long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

UNIT-I

PN JUNCTION DIODE

1.0 INTRODUCTION

Based on the electrical conductivity all the materials in nature are classified as insulators, semiconductors, and conductors.



Insulator: An insulator is a material that offers a very low level (or negligible) of conductivity when voltage is applied. Eg: Paper, Mica, glass, quartz. Typical resistivity level of an insulator is of the order of 10^{10} to $10^{12} \Omega\text{-cm}$. The energy band structure of an insulator is shown in the fig.1.1. Band structure of a material defines the band of energy levels that an electron can occupy. Valence band is the range of electron energy where the electron remain bended too the atom and do not contribute to the electric current. Conduction bend is the range of electron energies higher than valance band where electrons are free to accelerate under the influence of external voltage source resulting in the flow of charge.

The energy band between the valance band and conduction band is called as forbidden band gap. It is the energy required by an electron to move from balance band to conduction band i.e. the energy required for a valance electron to become a free electron.

$$1 \text{ eV} = 1.6 \times 10^{-19} \text{ J}$$

For an insulator, as shown in the fig.1.1 there is a large forbidden band gap of greater than 5eV. Because of this large gap there a very few electrons in the CB and hence the conductivity of insulator is poor. Even an increase in temperature or applied electric field is insufficient to transfer electrons from VB to CB.

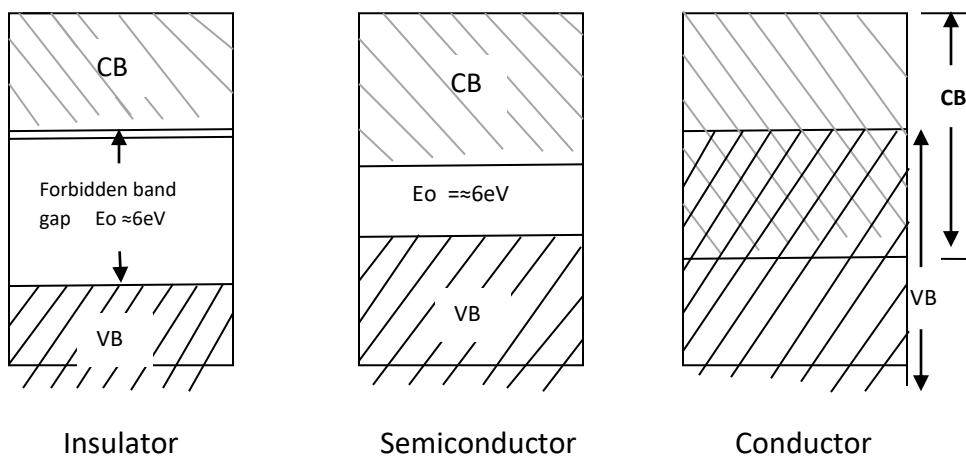


Fig:1.1 Energy band diagrams insulator, semiconductor and conductor

Conductors: A conductor is a material which supports a generous flow of charge when a voltage is applied across its terminals. i.e. it has very high conductivity. Eg: Copper, Aluminum, Silver, Gold. The resistivity of a conductor is in the order of 10^{-4} and $10^{-6} \Omega\text{-cm}$. The Valance and conduction bands overlap (fig1.1) and there is no energy gap for the electrons to move from valance band to conduction band. This implies that there are free electrons in CB even at absolute zero temperature (0K). Therefore at room temperature when electric field is applied large current flows through the conductor.

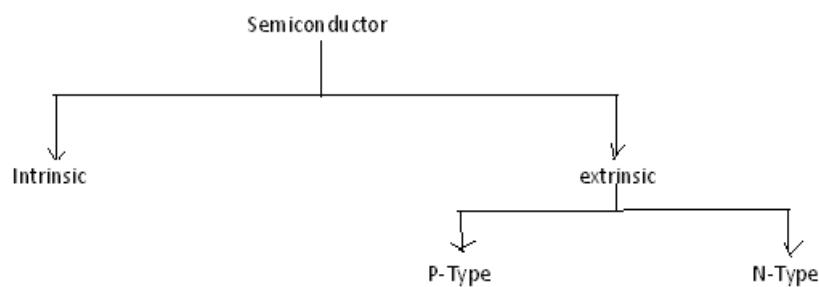
Semiconductor: A semiconductor is a material that has its conductivity somewhere between the insulator and conductor. The resistivity level is in the range of 10 and $10^4 \Omega\text{-cm}$. Two of the most commonly used are Silicon (Si=14 atomic no.) and germanium (Ge=32 atomic no.). Both have 4 valance electrons. The forbidden band gap is in the order of 1eV. For eg., the band gap energy for Si, Ge and GaAs is 1.21, 0.785 and 1.42 eV, respectively at absolute zero temperature (0K). At 0K and at low temperatures, the valance band electrons do not have sufficient energy to move from V to CB. Thus semiconductors act a insulators at 0K. as the temperature increases, a large number of valance electrons acquire sufficient energy to leave the VB, cross the forbidden band gap and reach CB. These are now free electrons as they can move freely under the influence of electric field. At room temperature there are sufficient electrons in the CB and hence the semiconductor is capable of conducting some current at room temperature.

Inversely related to the conductivity of a material is its resistance to the flow of charge or current. Typical resistivity values for various materials' are given as follows.

Insulator	Semiconductor	Conductor
$10^{-6} \Omega\text{-cm}$ (Cu)	$50\Omega\text{-cm}$ (Ge)	$10^{12} \Omega\text{-cm}$ (mica)
	$50\times 10^3 \Omega\text{-cm}$ (Si)	

Typical resistivity values

1.0.1 Semiconductor Types



A pure form of semiconductors is called as intrinsic semiconductor. Conduction in intrinsic sc is either due to thermal excitation or crystal defects. Si and Ge are the two most important semiconductors used. Other examples include Gallium arsenide GaAs, Indium Antimonide (InSb) etc.

Let us consider the structure of Si. A Si atomic no. is 14 and it has 4 valence electrons. These 4 electrons are shared by four neighboring atoms in the crystal structure by means of covalent bond. Fig. 1.2a shows the crystal structure of Si at absolute zero temperature (0K). Hence a pure SC acts has poor conductivity (due to lack of free electrons) at low or absolute zero temperature.

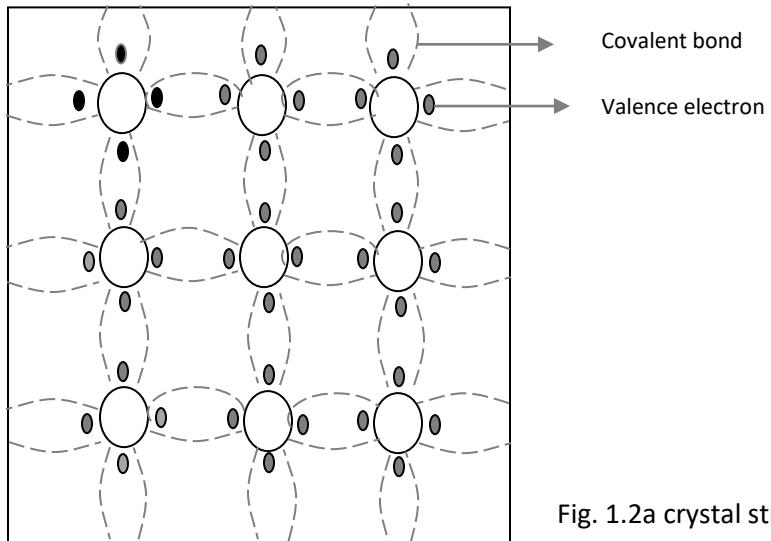


Fig. 1.2a crystal structure of Si at 0K

At room temperature some of the covalent bonds break up to thermal energy as shown in fig 1.2b. The valance electrons that jump into conduction band are called as free electrons that are available for conduction.

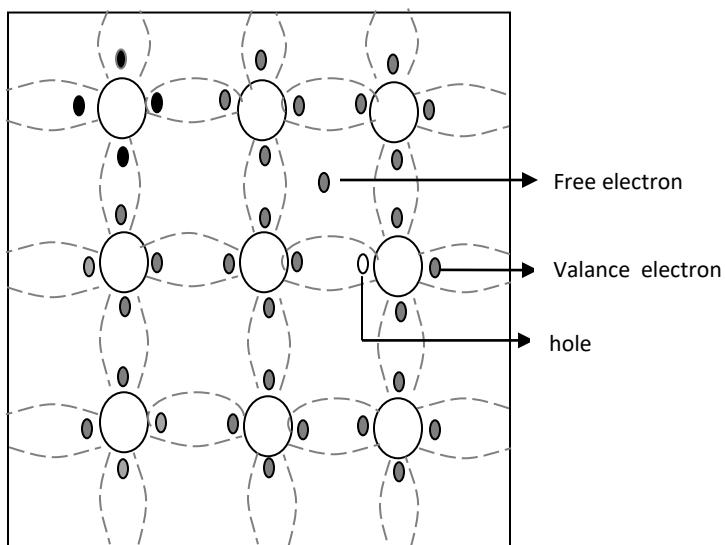
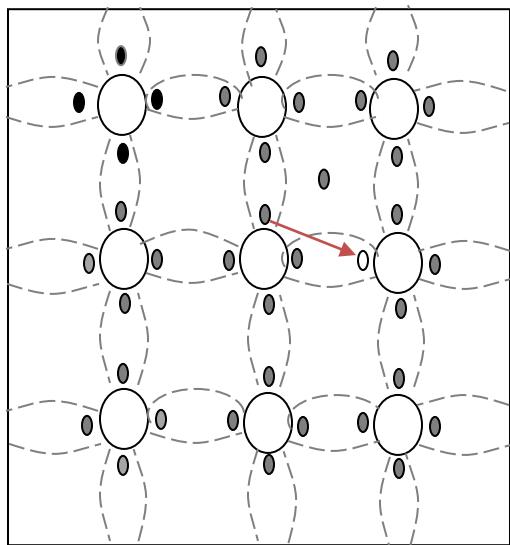


Fig. 1.2b crystal structure of Si at room temperature0K

The absence of electrons in covalent bond is represented by a small circle usually referred to as hole which is of positive charge. Even a hole serves as carrier of electricity in a manner similar to that of free electron.

The mechanism by which a hole contributes to conductivity is explained as follows:

When a bond is incomplete so that a hole exists, it is relatively easy for a valence electron in the neighboring atom to leave its covalent bond to fill this hole. An electron moving from a bond to fill a hole moves in a direction opposite to that of the electron. This hole, in its new position may now be filled by an electron from another covalent bond and the hole will correspondingly move one more step in the direction opposite to the motion of electron. Here we have a mechanism for conduction of electricity which does not involve free electrons. This phenomenon is illustrated in fig1.3



→ Electron movement
← Hole movement

Fig. 1.3a

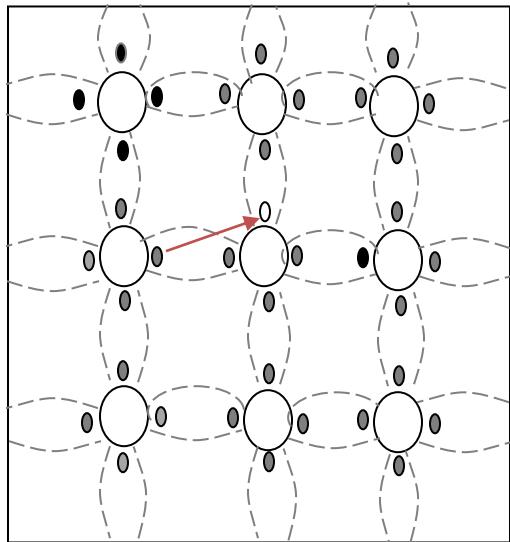


Fig. 1.3b

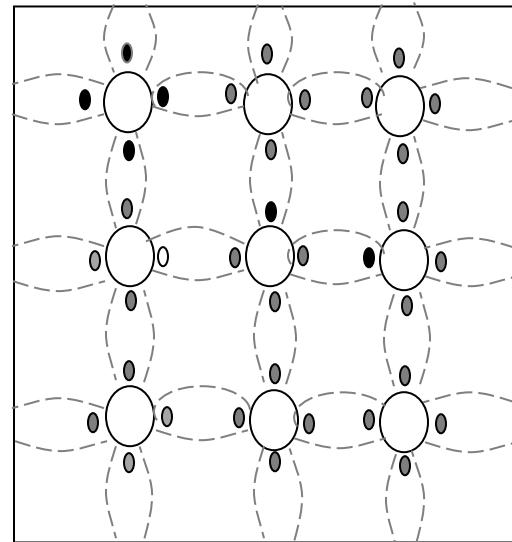


Fig. 1.3c

Fig 1.3a show that there is a hole at ion 6. Imagine that an electron from ion 5 moves into the hole at ion 6 so that the configuration of 1.3b results. If we compare both fig 1.3a & fig 1.3b, it appears as if the hole has moved towards the left from ion 6 to ion 5. Further if we compare fig 1.3b and fig 1.3c, the hole moves from ion 5 to ion 4. This discussion indicates the motion of hole is in a direction opposite to that of motion of electron. Hence we consider holes as physical entities whose movement constitutes flow of current.

In a pure semiconductor, the number of holes is equal to the number of free electrons.

1.0.2 EXTRINSIC SEMICONDUCTOR

Intrinsic semiconductor has very limited applications as they conduct very small amounts of current at room temperature. The current conduction capability of intrinsic semiconductor can be increased significantly by adding a small amounts impurity to the intrinsic semiconductor. By adding impurities it becomes impure or extrinsic semiconductor. This process of adding impurities is called as doping. The amount of impurity added is 1 part in 10^6 atoms.

N type semiconductor: If the added impurity is a pentavalent atom then the resultant semiconductor is called N-type semiconductor. Examples of pentavalent impurities are Phosphorus, Arsenic, Bismuth, Antimony etc.

A pentavalent impurity has five valence electrons. Fig 1.4a shows the crystal structure of N-type semiconductor material where four out of five valence electrons of the impurity atom(antimony) forms covalent bond with the four intrinsic semiconductor atoms. The fifth electron is loosely bound to the impurity atom. This loosely bound electron can be easily

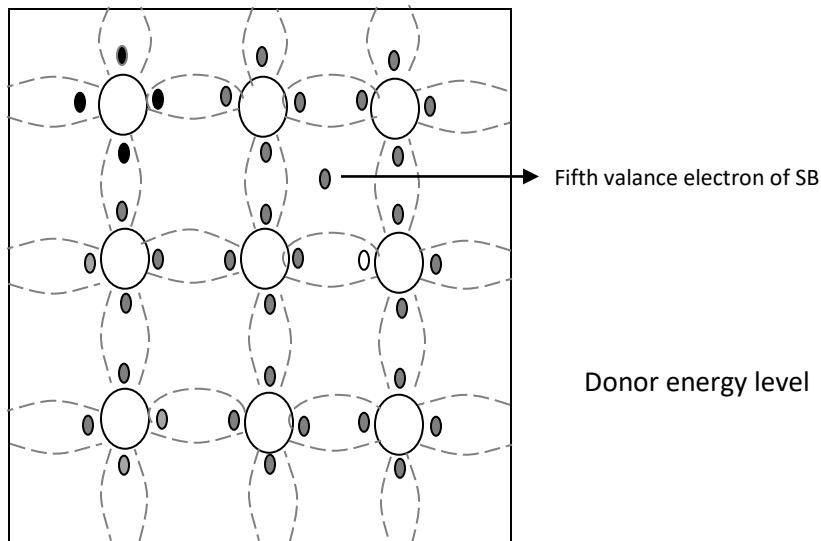


Fig. 1.4a crystal structure of N type SC

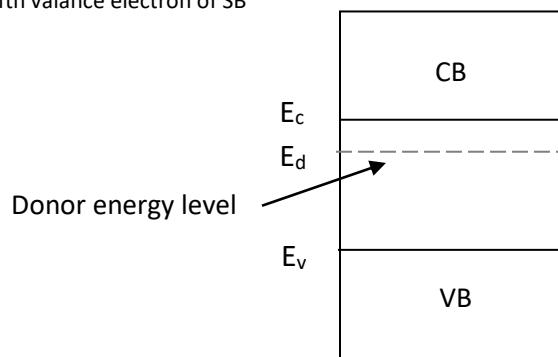


Fig. 1.4b Energy band diagram of N type

Excited from the valence band to the conduction band by the application of electric field or increasing the thermal energy. The energy required to detach the fifth electron from the impurity atom is very small of the order of 0.01 eV for Ge and 0.05 eV for Si.

The effect of doping creates a discrete energy level called donor energy level in the forbidden band gap with energy level E_d slightly less than the conduction band (fig 1.4b). The difference between the energy levels of the conducting band and the donor energy level is the energy required to free the fifth valence electron (0.01 eV for Ge and 0.05 eV for Si). At room temperature almost all the fifth electrons from the donor impurity atom are raised to conduction band and hence the number of electrons in the conduction band increases significantly. Thus every antimony atom contributes to one conduction electron without creating a hole.

In the N-type sc the no. of electrons increases and the no. of holes decreases compared to those available in an intrinsic sc. The reason for decrease in the no. of holes is that the larger no. of electrons present increases the recombination of electrons with holes. Thus current in N type sc is dominated by electrons which are referred to as majority carriers. Holes are the minority carriers in N type sc.

P type semiconductor: If the added impurity is a trivalent atom then the resultant semiconductor is called P-type semiconductor. Examples of trivalent impurities are Boron, Gallium , indium etc.

The crystal structure of p type sc is shown in the fig1.5a. The three valence electrons of the impurity (boon) forms three covalent bonds with the neighboring atoms and a vacancy exists in the fourth bond giving rise to the holes. The hole is ready to accept an electron from the neighboring atoms. Each trivalent atom contributes to one hole generation and thus introduces a large no. of holes in the valance band. At the same time the no. electrons are decreased compared to those available in intrinsic sc because of increased recombination due to creation of additional holes.

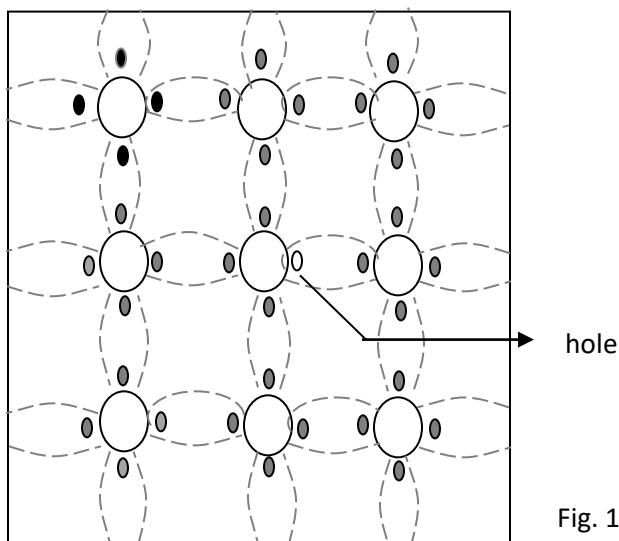


Fig. 1.5a crystal structure of P type sc

Thus in P type sc , holes are majority carriers and electrons are minority carriers. Since each trivalent impurity atoms are capable accepting an electron, these are called as acceptor atoms. The following fig 1.5b shows the pictorial representation of P type sc

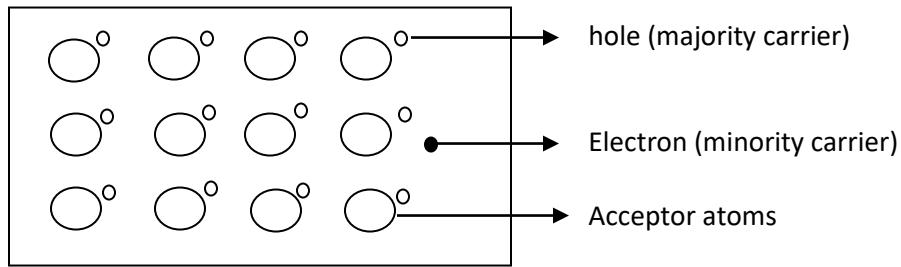


Fig. 1.5b crystal structure of P type sc

- The conductivity of N type sc is greater than that of P type sc as the mobility of electron is greater than that of hole.
- For the same level of doping in N type sc and P type sc, the conductivity of an N type sc is around twice that of a P type sc

1.0.3 CONDUCTIVITY OF SEMICONDUCTOR

In a pure sc, the no. of holes is equal to the no. of electrons. Thermal agitation continue to produce new electron- hole pairs and the electron hole pairs disappear because of recombination. with each electron hole pair created , two charge carrying particles are formed . One is negative which is a free electron with mobility μ_n . The other is a positive i.e., hole with mobility μ_p . The electrons and hole move in opppsitte direction in a an electric field E, but since they are of opposite sign, the current due to each is in the same direction. Hence the total current density J within the intrinsic sc is given by

$$J = J_n + J_p$$

$$=q n \mu_n E + q p \mu_p E$$

$$= (n \mu_n + p \mu_p) q E$$

$$=\sigma E$$

Where n =no. of electrons / unit volume i.e., concentration of free electrons

P = no. of holes / unit volume i.e., concentration of holes

E =applied electric field strength, V/m

q = charge of electron or hole I n Coulombs

Hence, σ is the conductivity of sc which is equal to $(n \mu_n + p \mu_p) q$. he resistivity of sc is reciprocal of conductivity.

$$\rho = 1/\sigma$$

It is evident from the above equation that current density within a sc is directly proportional to applied electric field E.

For pure sc, $n=p=n_i$ where n_i = intrinsic concentration. The value of n_i is given by

$$n_i^2 = AT^3 \exp(-E_{GO}/KT)$$

$$\text{therefore, } J = n_i (\mu_n + \mu_p) q E$$

$$\text{Hence conductivity in intrinsic sc is } \sigma_i = n_i (\mu_n + \mu_p) q$$

Intrinsic conductivity increases at the rate of 5% per $^{\circ}\text{C}$ for Ge and 7% per $^{\circ}\text{C}$ for Si.

Conductivity in extrinsic sc (N Type and P Type):

The conductivity of intrinsic sc is given by $\sigma_i = n_i (\mu_n + \mu_p) q = (n \mu_n + p \mu_p) q$

For N type, $n \gg p$

$$\text{Therefore } \sigma = q n \mu_n$$

For P type, $p \gg n$

$$\text{Therefore } \sigma = q p \mu_p$$

1.0.4 CHARGE DENSITIES IN P TYPE AND N TYPE SEMICONDUCTOR:

Mass Action Law:

Under thermal equilibrium for any semiconductor, the product of the no. of holes and the concentration of electrons is constant and is independent of amount of donor and acceptor impurity doping.

$$n.p = n_i^2$$

where n = electron concentration

p = hole concentration

$$n_i^2 = \text{intrinsic concentration}$$

Hence in N type sc, as the no. of electrons increase the no. of holes decreases. Similarly in P type as the no. of holes increases the no. of electrons decreases. Thus the product is constant and is equal to n_i^2 in case of intrinsic as well as extrinsic sc.

The law of mass action has given the relationship between free electrons concentration and hole concentration. These concentrations are further related by the law of electrical neutrality as explained below.

Law of electrical neutrality:

Sc materials are electrically neutral. According to the law of electrical neutrality, in an electrically neutral material, the magnitude of positive charge concentration is equal to that of negative charge concentration. Let us consider a sc that has N_D donor atoms per cubic centimeter and N_A acceptor atoms per cubic centimeter i.e., the concentration of donor and acceptor atoms are N_D and N_A respectively. Therefore N_D positively charged ions per cubic centimeter are contributed by donor atoms and N_A negatively charged ions per cubic centimeter are contributed by the acceptor atoms. Let n, p is concentration of free electrons and holes respectively. Then according to the law of neutrality

$$N_D + p = N_A + n \quad \dots \dots \dots \text{eq 1.1}$$

For N type sc, $N_A = 0$ and $n \gg p$. Therefore $N_D \approx n$ eq 1.2

Hence for N type sc the free electron concentration is approximately equal to the concentration of donor atoms. In later applications since some confusion may arise as to which type of sc is under consideration at the given moment, the subscript n or p is added for N type or P type respectively. Hence eq 1.2 becomes $N_D \approx n_n$

Therefore current density in N type sc is $J = N_D \mu_n q E$

And conductivity $\sigma = N_D \mu_n q$

For P type sc, $N_D = 0$ and $p \gg n$. Therefore $N_A \approx p$

$$\text{Or } N_A \approx p_p$$

Hence for P type sc the hole concentration is approximately equal to the concentration of acceptor atoms.

Therefore current density in N type sc is $J = N_A \mu_p q E$

And conductivity $\sigma = N_A \mu_p q$

Mass action law for N type, $n_n p_n = n_i^2$

$$p_n = n_i^2 / N_D \quad \text{since } (n_n \approx N_D)$$

Mass action law for P type, $n_p p_p = n_i^2$

$$n_p = n_i^2 / N_A \quad \text{since } (p_p \approx N_A)$$

1.1 QUANTITATIVE THEORY OF PN JUNCTION DIODE

1.1.1 PN JUNCTION WITH NO APPLIED VOLTAGE OR OPEN CIRCUIT CONDITION:

In a piece of sc, if one half is doped by p type impurity and the other half is doped by n type impurity, a PN junction is formed. The plane dividing the two halves or zones is called PN junction. As

shown in the fig the n type material has high concentration of free electrons, while p type material has high concentration of holes. Therefore at the junction there is a tendency of free electrons to diffuse over to the P side and the holes to the N side. This process is called diffusion. As the free electrons move across the junction from N type to P type, the donor atoms become positively charged. Hence a positive charge is built on the N-side of the junction. The free electrons that cross the junction uncover the negative acceptor ions by filling the holes. Therefore a negative charge is developed on the p –side of the junction..This net negative charge on the p side prevents further diffusion of electrons into the p side. Similarly the net positive charge on the N side repels the hole crossing from p side to N side. Thus a barrier is set up near the junction which prevents the further movement of charge carriers i.e. electrons and holes. As a consequence of induced electric field across the depletion layer, an electrostatic potential difference is established between P and N regions, which are called the potential barrier, junction barrier, diffusion potential or contact potential, V_0 . The magnitude of the contact potential V_0 varies with doping levels and temperature. V_0 is 0.3V for Ge and 0.72 V for Si.

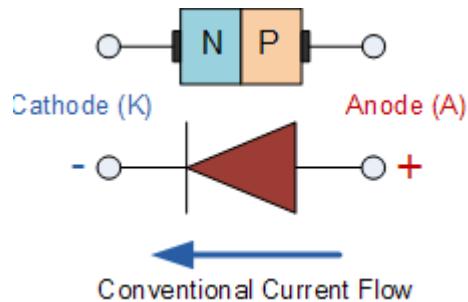


Fig 1.6: Symbol of PN Junction Diode

The electrostatic field across the junction caused by the positively charged N-Type region tends to drive the holes away from the junction and negatively charged p type regions tend to drive the electrons away from the junction. The majority holes diffusing out of the P region leave behind negatively charged acceptor atoms bound to the lattice, thus exposing a negative space charge in a previously neutral region. Similarly electrons diffusing from the N region expose positively ionized donor atoms and a double space charge builds up at the junction as shown in the fig. 1.7a

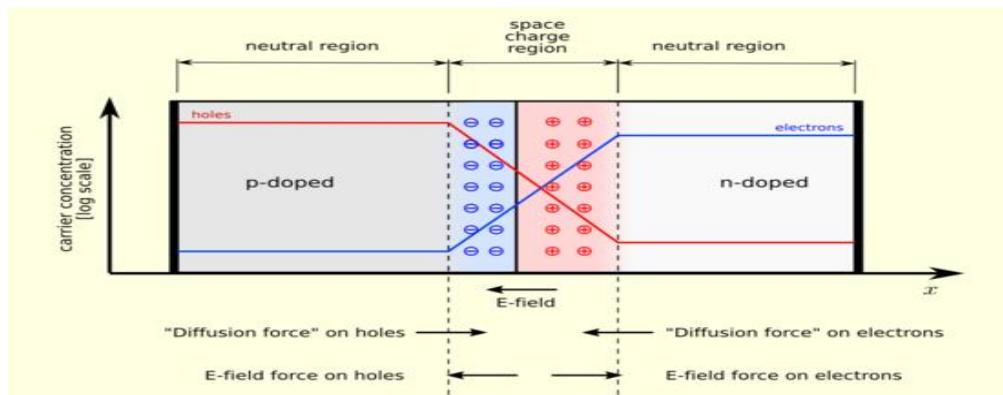


Fig 1.7a

It is noticed that the space charge layers are of opposite sign to the majority carriers diffusing into them, which tends to reduce the diffusion rate. Thus the double space of the layer causes an electric field to be set up across the junction directed from N to P regions, which is in such a direction to inhibit the diffusion of majority electrons and holes as illustrated in fig 1.7b. The shape of the charge density, ρ , depends upon how diode is doped. Thus the junction region is depleted of mobile charge carriers. Hence it is called depletion layer, space region, and transition region. The depletion region is of the order of $0.5\mu\text{m}$ thick. There are no mobile carriers in this narrow depletion region. Hence no current flows across the junction and the system is in equilibrium. To the left of this depletion layer, the carrier concentration is $p = N_A$ and to its right it is $n = N_D$.

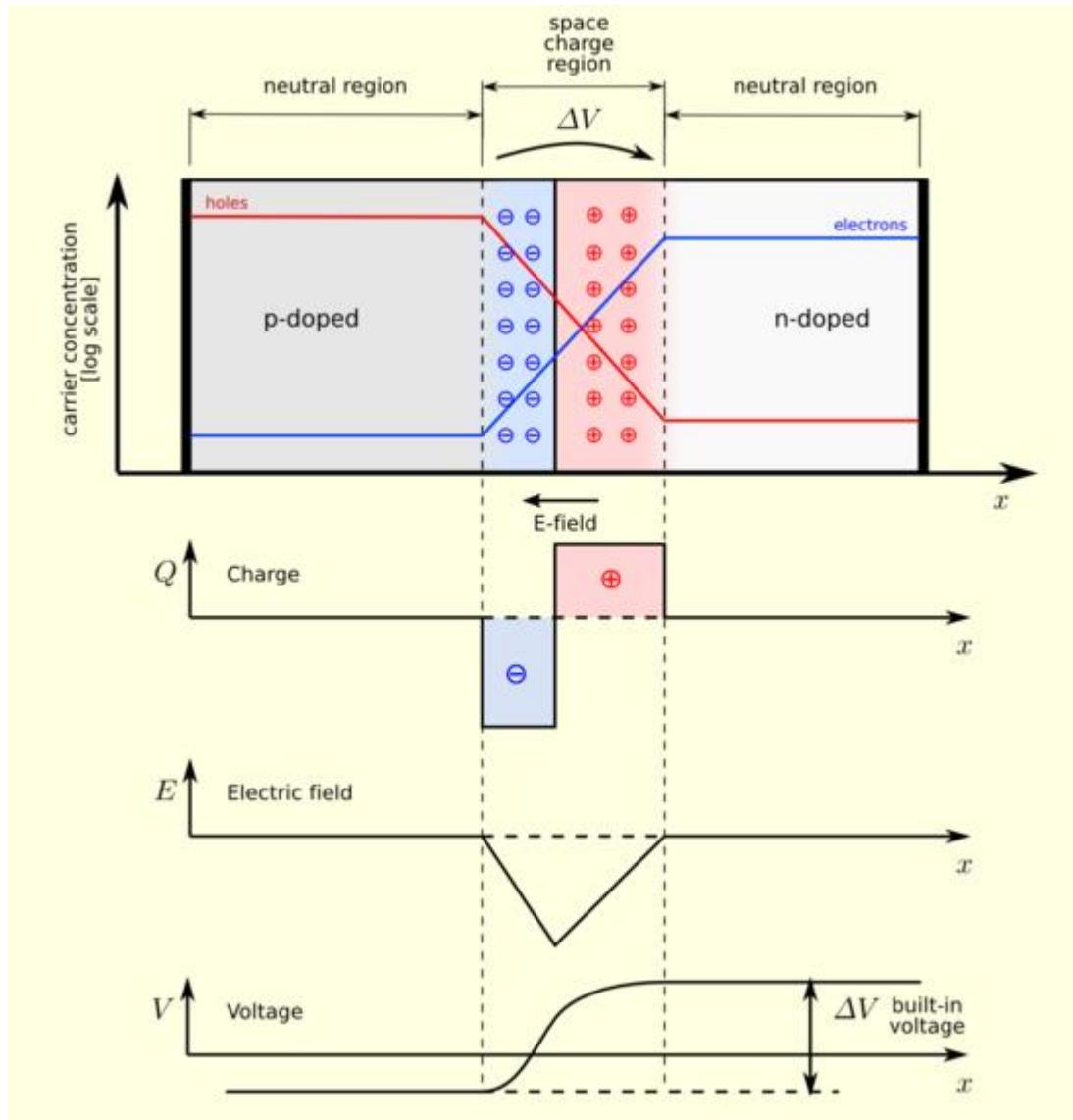


Fig 1.7b

1.1.2 FORWARD BIASED JUNCTION DIODE

When a diode is connected in a **Forward Bias** condition, a negative voltage is applied to the N-type material and a positive voltage is applied to the P-type material. If this external voltage becomes greater than the value of the potential barrier, approx. 0.7 volts for silicon and 0.3 volts for germanium, the potential barriers opposition will be overcome and current will start to flow. This is because the negative voltage pushes or repels electrons towards the junction giving them the energy to cross over and combine with the holes being pushed in the opposite direction towards the junction by the positive voltage. This results in a characteristics curve of zero current flowing up to this voltage point, called the "knee" on the static curves and then a high current flow through the diode with little increase in the external voltage as shown below.

Forward Characteristics Curve for a Junction Diode

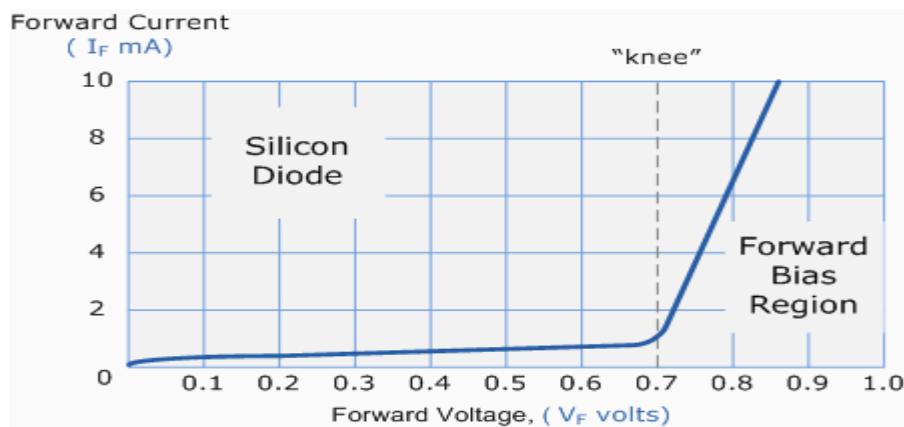


Fig 1.8a: Diode Forward Characteristics

The application of a forward biasing voltage on the junction diode results in the depletion layer becoming very thin and narrow which represents a low impedance path through the junction thereby allowing high currents to flow. The point at which this sudden increase in current takes place is represented on the static I-V characteristics curve above as the "knee" point.

Forward Biased Junction Diode showing a Reduction in the Depletion Layer

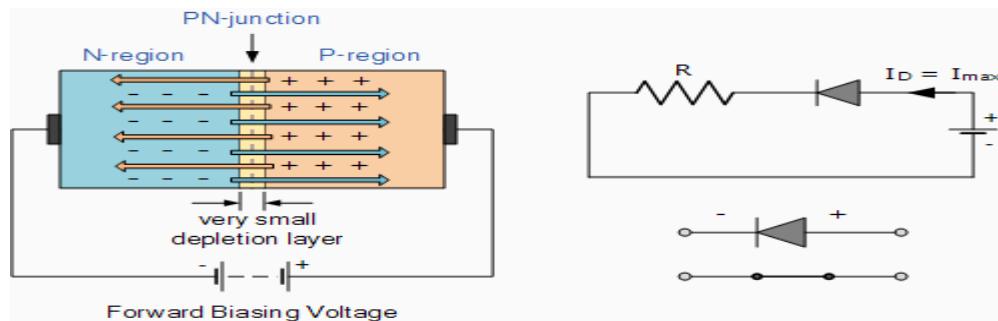


Fig 1.8b: Diode Forward Bias

This condition represents the low resistance path through the PN junction allowing very large currents to flow through the diode with only a small increase in bias voltage. The actual potential difference across the junction or diode is kept constant by the action of the depletion layer at approximately 0.3v for germanium and approximately 0.7v for silicon junction diodes. Since the diode can conduct "infinite" current above this knee point as it effectively becomes a short circuit, therefore resistors are used in series with the diode to limit its current flow. Exceeding its maximum forward current specification causes the device to dissipate more power in the form of heat than it was designed for resulting in a very quick failure of the device.

1.1.2 PN JUNCTION UNDER REVERSE BIAS CONDITION:

Reverse Biased Junction Diode

When a diode is connected in a **Reverse Bias** condition, a positive voltage is applied to the N-type material and a negative voltage is applied to the P-type material. The positive voltage applied to the N-type material attracts electrons towards the positive electrode and away from the junction, while the holes in the P-type end are also attracted away from the junction towards the negative electrode. The net result is that the depletion layer grows wider due to a lack of electrons and holes and presents a high impedance path, almost an insulator. The result is that a high potential barrier is created thus preventing current from flowing through the semiconductor material.

Reverse Biased Junction Diode showing an Increase in the Depletion

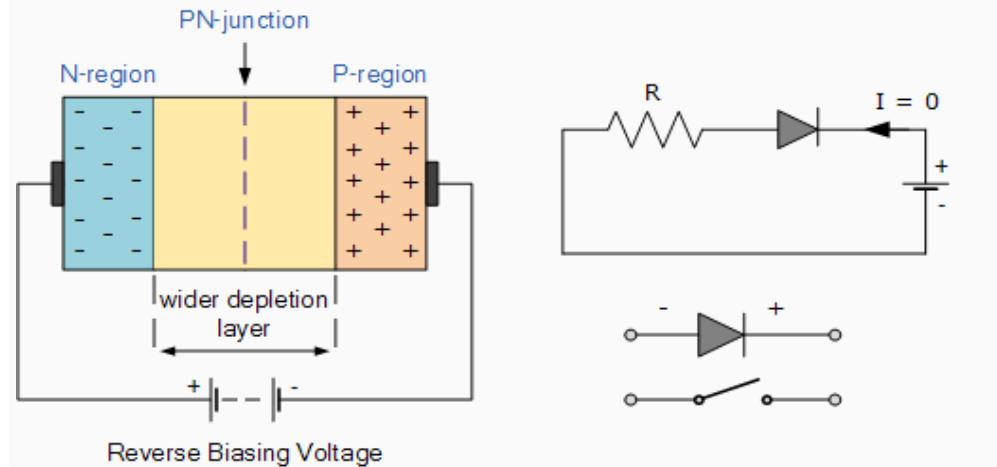


Fig 1.9a: Diode Reverse Bias

This condition represents a high resistance value to the PN junction and practically zero current flows through the junction diode with an increase in bias voltage. However, a very small **leakage current** does flow through the junction which can be measured in microamperes, (μA). One final point, if the reverse bias voltage V_r applied to the diode is increased to a sufficiently high enough value, it will cause the PN junction to overheat and fail due to the avalanche effect around the junction. This may

cause the diode to become shorted and will result in the flow of maximum circuit current, and this is shown as a step downward slope in the reverse static characteristics curve below.

Reverse Characteristics Curve for a Junction Diode

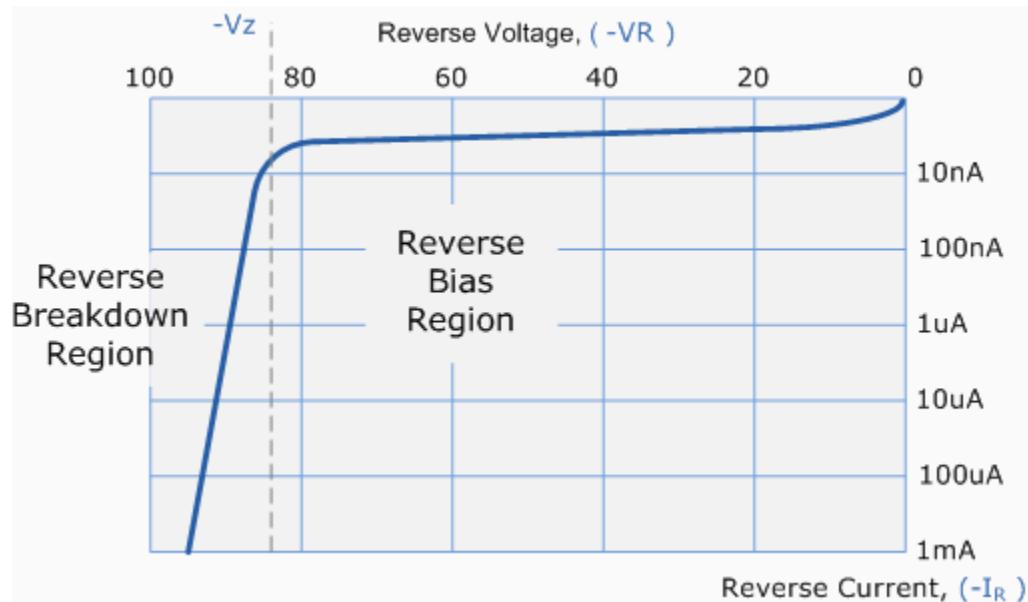


Fig 1.9b: Diode Reverse Characteristics

Sometimes this avalanche effect has practical applications in voltage stabilizing circuits where a series limiting resistor is used with the diode to limit this reverse breakdown current to a preset maximum value thereby producing a fixed voltage output across the diode. These types of diodes are commonly known as **Zener Diodes**

1.2 VI CHARACTERISTICS AND THEIR TEMPERATURE DEPENDENCE

Diode terminal characteristics equation for diode junction current:

$$I_D = I_0 (e^{\frac{V}{\eta V_T}} - 1)$$

Where $V_T = KT/q$;

V_D _ diode terminal voltage, Volts

I_0 _ temperature-dependent saturation current, μA

T _ absolute temperature of p-n junction, K

K _ Boltzmann's constant $1.38 \times 10^{-23} J/K$

q _ electron charge $1.6 \times 10^{-19} C$

η = empirical constant, 1 for Ge and 2 for Si

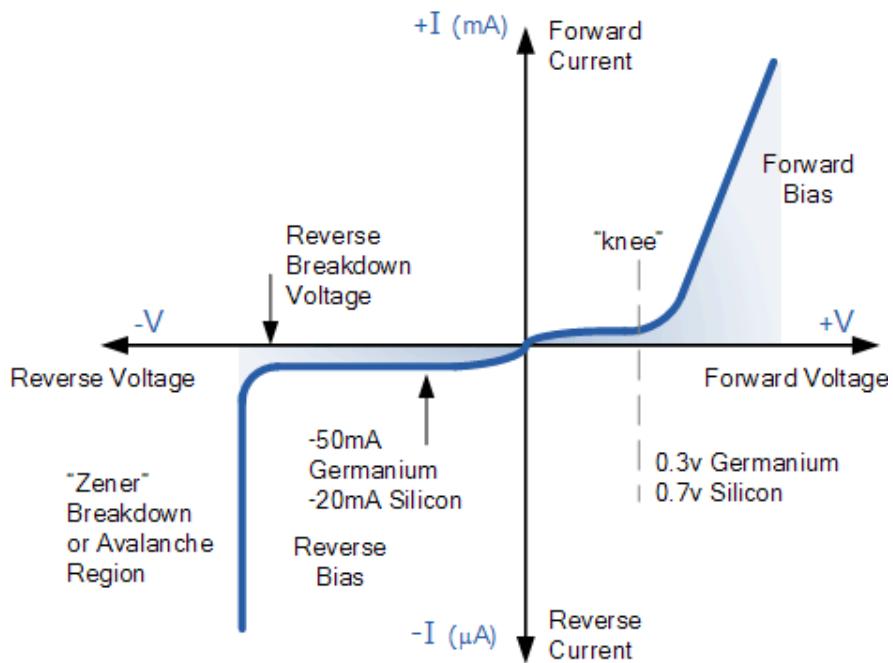


Fig 1.10: Diode Characteristics

Temperature Effects on Diode

Temperature can have a marked effect on the characteristics of a silicon semiconductor diode as shown in Fig. 11. It has been found experimentally that the reverse saturation current I_o will just about double in magnitude for every 10°C increase in temperature.

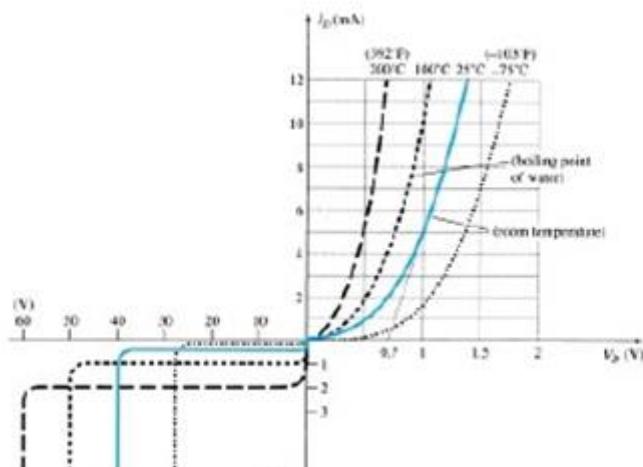


Fig 1.11 Variation in Diode Characteristics with temperature change

It is not uncommon for a germanium diode with an I_o in the order of 1 or 2 A at 25°C to have a leakage current of 100 A - 0.1 mA at a temperature of 100°C . Typical values of I_o for silicon are much lower than

that of germanium for similar power and current levels. The result is that even at high temperatures the levels of I_o for silicon diodes do not reach the same high levels obtained. For germanium—a very important reason that silicon devices enjoy a significantly higher level of development and utilization in design. Fundamentally, the open-circuit equivalent in the reverse bias region is better realized at any temperature with silicon than with germanium. The increasing levels of I_o with temperature account for the lower levels of threshold voltage, as shown in Fig. 1.11. Simply increase the level of I_o in and not rise in diode current. Of course, the level of TK also will be increase, but the increasing level of I_o will overpower the smaller percent change in TK. As the temperature increases the forward characteristics are actually becoming more “ideal,”

1.3 IDEAL VERSUS PRACTICAL RESISTANCE LEVELS

DC or Static Resistance

The application of a dc voltage to a circuit containing a semiconductor diode will result in an operating point on the characteristic curve that will not change with time. The resistance of the diode at the operating point can be found simply by finding the corresponding levels of V_D and I_D as shown in Fig. 1.12 and applying the following Equation:

$$R_D = \frac{V_D}{I_D}$$

The dc resistance levels at the knee and below will be greater than the resistance levels obtained for the vertical rise section of the characteristics. The resistance levels in the reverse-bias region will naturally be quite high. Since ohmmeters typically employ a relatively constant-current source, the resistance determined will be at a preset current level (typically, a few mill amperes).

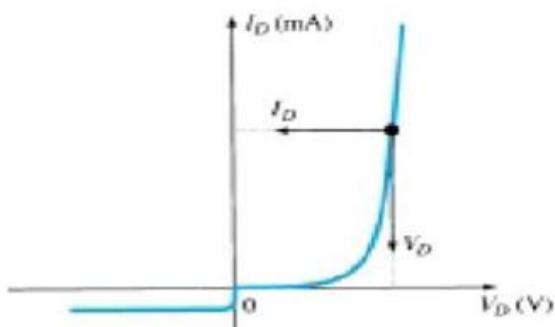


Fig 1.12 Determining the dc resistance of a diode at a particular operating point.

AC or Dynamic Resistance

It is obvious from Eq. 1.3 that the dc resistance of a diode is independent of the shape of the characteristic in the region surrounding the point of interest. If a sinusoidal rather than dc input is applied, the situation will change completely. The varying input will move the instantaneous operating point up and down a region of the characteristics and thus defines a specific change in current and voltage as shown in Fig. 1.13. With no applied varying signal, the point of operation would be the Q-point appearing on Fig. 1.13 determined by the applied dc levels. The designation Q-point is derived from the word quiescent, which means “still or unvarying.” A straight-line drawn tangent to the curve through the Q-point as shown in Fig. 1.13 will define a particular change in voltage and current that can be used to determine the ac or dynamic resistance for this region of the diode characteristics. In equation form,

$$r_d = \frac{\Delta V_d}{\Delta I_d}$$

Where Δ Signifies a finite change in the quantity

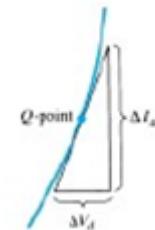


Fig 1.13: Determining the ac resistance of a diode at a particular operating point.

1.4 DIODE EQUIVALENT CIRCUITS

An equivalent circuit is a combination of elements properly chosen to best represent the actual terminal characteristics of a device, system, or such in a particular operating region. In other words, once the equivalent circuit is defined, the device symbol can be removed from a schematic and the equivalent circuit inserted in its place without severely affecting the actual behavior of the system. The result is often a network that can be solved using traditional circuit analysis techniques.

Piecewise-Linear Equivalent Circuit

One technique for obtaining an equivalent circuit for a diode is to approximate the characteristics of the device by straight-line segments, as shown in Fig. 1.31. The resulting equivalent

circuit is naturally called the piecewise-linear equivalent circuit. It should be obvious from Fig. 1.31 that the straight-line segments do not result in an exact duplication of the actual characteristics, especially in the knee region. However, the resulting segments are sufficiently close to the actual curve to establish an equivalent circuit that will provide an excellent first approximation to the actual behaviour of the device. The ideal diode is included to establish that there is only one direction of conduction through the device, and a reverse-bias condition will result in the open-circuit state for the device. Since a silicon semiconductor diode does not reach the conduction state until V_D reaches 0.7 V with a forward bias (as shown in Fig. 1.14a), a battery V_T opposing the conduction direction must appear in the equivalent circuit as shown in Fig. 1.14b. The battery simply specifies that the voltage across the device must be greater than the threshold battery voltage before conduction through the device in the direction dictated by the ideal diode can be established. When conduction is established, the resistance of the diode will be the specified value of r_{av} .

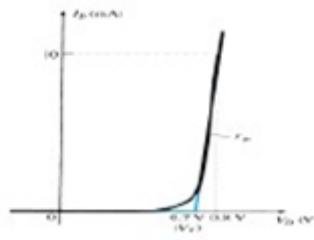


Fig: 1.14a Diode piecewise-linear model characteristics

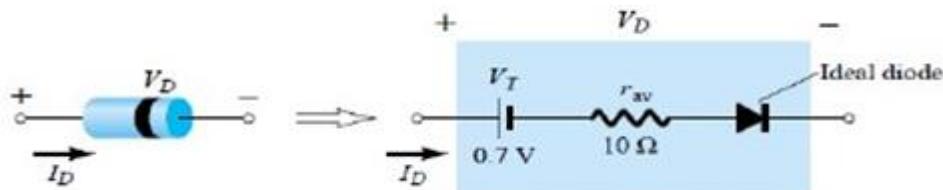


Fig: 1.14b Diode piecewise-linear model equivalent circuit

The approximate level of r_{av} can usually be determined from a specified operating point on the specification sheet. For instance, for a silicon semiconductor diode, if $I_F = 10 \text{ mA}$ (a forward conduction current for the diode) at $V_D = 0.8 \text{ V}$, we know for silicon that a shift of 0.7 V is required before the characteristics rise.

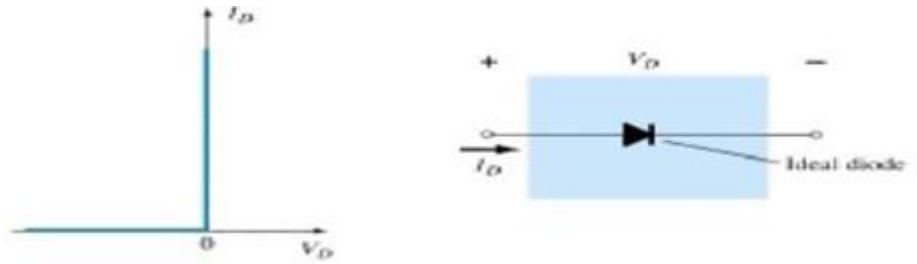


Fig 1.15 Ideal Diode and its characteristics

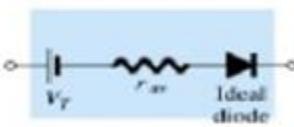
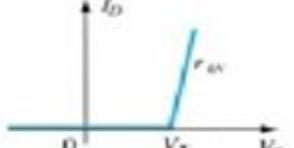
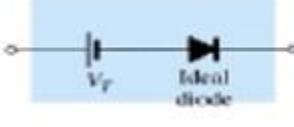
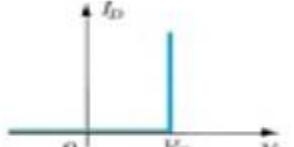
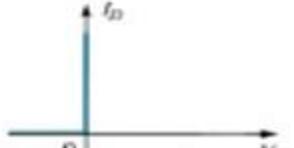
Type	Conditions	Model	Characteristics
Piecewise-linear model			
Simplified model	$R_{\text{network}} \gg r_{av}$		
Ideal device	$R_{\text{network}} \gg r_{av}$ $E_{\text{network}} \gg V_T$		

Fig 1.16: Diode equivalent circuits(models)

1.5 TRANSITION AND DIFFUSION CAPACITANCE

Electronic devices are inherently sensitive to very high frequencies. Most shunt capacitive effects that can be ignored at lower frequencies because the reactance $XC=1/2\pi fC$ is very large (open-circuit equivalent). This, however, cannot be ignored at very high frequencies. XC will become sufficiently small due to the high value of f to introduce a low-reactance “shorting” path. In the p-n semiconductor diode, there are two capacitive effects to be considered. In the reverse-bias region we have the transition- or depletion region capacitance (CT), while in the forward-bias region we have the diffusion (CD) or storage capacitance. Recall that the basic equation for the capacitance of a parallel-plate capacitor is defined by $C=\epsilon A/d$, where ϵ is the permittivity of the dielectric (insulator) between the plates of area A separated by a distance d . In the reverse-, bias region there is a depletion region (free of carriers) that behaves essentially like an insulator between the layers of opposite charge. Since

the depletion width (d) will increase with increased reverse-bias potential, the resulting transition capacitance will decrease. The fact that the capacitance is dependent on the applied reverse-bias potential has application in a number of electronic systems. Although the effect described above will also be present in the forward-bias region, it is over shadowed by a capacitance effect directly dependent on the rate at which charge is injected into the regions just outside the depletion region. The capacitive effects described above are represented by a capacitor in parallel with the ideal diode, as shown in Fig. 1.38. For low- or mid-frequency applications (except in the power area), however, the capacitor is normally not included in the diode symbol.

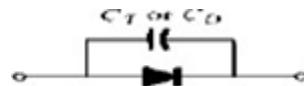


Fig 1.17: Including the effect of the transition or diffusion capacitance on the semiconductor diode

Diode capacitances: The diode exhibits two types of capacitances transition capacitance and diffusion capacitance.

- Transition capacitance: The capacitance which appears between positive ion layer in n-region and negative ion layer in p-region.
- Diffusion capacitance: This capacitance originates due to diffusion of charge carriers in the opposite regions.

The transition capacitance is very small as compared to the diffusion capacitance.

In reverse bias transition, the capacitance is the dominant and is given by:

$$C_T = \epsilon A/W$$

where C_T - transition capacitance

A - diode cross sectional area

W - depletion region width

In forward bias, the diffusion capacitance is the dominant and is given by:

$$C_D = dQ/dV = \tau^* dI/dV = \tau^* g = V/r \text{ (general)}$$

where C_D - diffusion capacitance

dQ - change in charge stored in depletion region

V - change in applied voltage

τ - time interval for change in voltage

g - diode conductance

r - diode resistance

The diffusion capacitance at low frequencies is given by the formula:

$$C_D = \tau^* g / 2 \text{ (low frequency)}$$

The diffusion capacitance at high frequencies is inversely proportional to the frequency and is given by the formula:

$$C_D = g(\tau/2\omega)^{1/2}$$

Note: The variation of diffusion capacitance with applied voltage is used in the design of varactor.

1.6 BREAK DOWN MECHANISMS

When an ordinary **P-N junction diode** is reverse biased, normally only very small reverse saturation current flows. This current is due to movement of minority carriers. It is almost independent of the voltage applied. However, if the reverse bias is increased, a point is reached when the junction breaks down and the reverse current increases abruptly. This current could be large enough to destroy the junction. If the reverse current is limited by means of a suitable series resistor, the power dissipation at the junction will not be excessive, and the device may be operated continuously in its breakdown region to its normal (reverse saturation) level. It is found that for a suitably designed diode, the breakdown voltage is very stable over a wide range of reverse currents. This quality gives the breakdown diode many useful applications as a voltage reference source.

The critical value of the voltage, at which the breakdown of a P-N junction diode occurs, is called the *breakdown voltage*. The breakdown voltage depends on the width of the depletion region, which, in turn, depends on the doping level. The junction offers almost zero resistance at the breakdown point.

There are two mechanisms by which breakdown can occur at a reverse biased P-N junction:

1. **avalanche breakdown and**
2. **Zener breakdown.**

Avalanche breakdown

The minority carriers, under reverse biased conditions, flowing through the junction acquire a kinetic energy which increases with the increase in reverse voltage. At a sufficiently high reverse voltage (say 5 V or more), the kinetic energy of minority carriers becomes so large that they knock out electrons from the covalent bonds of the semiconductor material. As a result of collision, the liberated

electrons in turn liberate more electrons and the current becomes very large leading to the breakdown of the crystal structure itself. This phenomenon is called the avalanche breakdown. The breakdown region is the knee of the characteristic curve. Now the current is not controlled by the junction voltage but rather by the external circuit.

Zener breakdown

Under a very high reverse voltage, the depletion region expands and the potential barrier increases leading to a very high electric field across the junction. The electric field will break some of the covalent bonds of the semiconductor atoms leading to a large number of free minority carriers, which suddenly increase the reverse current. This is called the Zener effect. The breakdown occurs at a particular and constant value of reverse voltage called the breakdown voltage, it is found that Zener breakdown occurs at electric field intensity of about 3×10^7 V/m.



Fig 1.18: Diode characteristics with breakdown

Either of the two (Zener breakdown or avalanche breakdown) may occur independently, or both of these may occur simultaneously. Diode junctions that breakdown below 5 V are caused by Zener effect. Junctions that experience breakdown above 5 V are caused by avalanche effect. Junctions that breakdown around 5 V are usually caused by combination of two effects. The Zener breakdown occurs in heavily doped junctions (P-type semiconductor moderately doped and N-type heavily doped), which produce narrow depletion layers. The avalanche breakdown occurs in lightly doped junctions, which produce wide depletion layers. With the increase in junction temperature Zener breakdown voltage is reduced while the avalanche breakdown voltage increases. The Zener diodes have a negative temperature coefficient while avalanche diodes have a positive temperature coefficient. Diodes that have breakdown voltages around 5 V have zero temperature coefficient. The breakdown phenomenon is reversible and harmless so long as the safe operating temperature is maintained.

1.7 ZENER DIODES

The **Zener diode** is like a general-purpose signal diode consisting of a silicon PN junction. When biased in the forward direction it behaves just like a normal signal diode passing the rated current, but as soon as a reverse voltage applied across the zener diode exceeds the rated voltage of the device, the diodes breakdown voltage V_B is reached at which point a process called *Avalanche Breakdown* occurs in the semiconductor depletion layer and a current starts to flow through the diode to limit this increase in voltage.

The current now flowing through the zener diode increases dramatically to the maximum circuit value (which is usually limited by a series resistor) and once achieved this reverse saturation current remains fairly constant over a wide range of applied voltages. This breakdown voltage point, V_B is called the "zener voltage" for zener diodes and can range from less than one volt to hundreds of volts.

The point at which the zener voltage triggers the current to flow through the diode can be very accurately controlled (to less than 1% tolerance) in the doping stage of the diodes semiconductor construction giving the diode a specific *zener breakdown voltage*, (V_z) for example, 4.3V or 7.5V. This zener breakdown voltage on the I-V curve is almost a vertical straight line.

Zener Diode I-V Characteristics

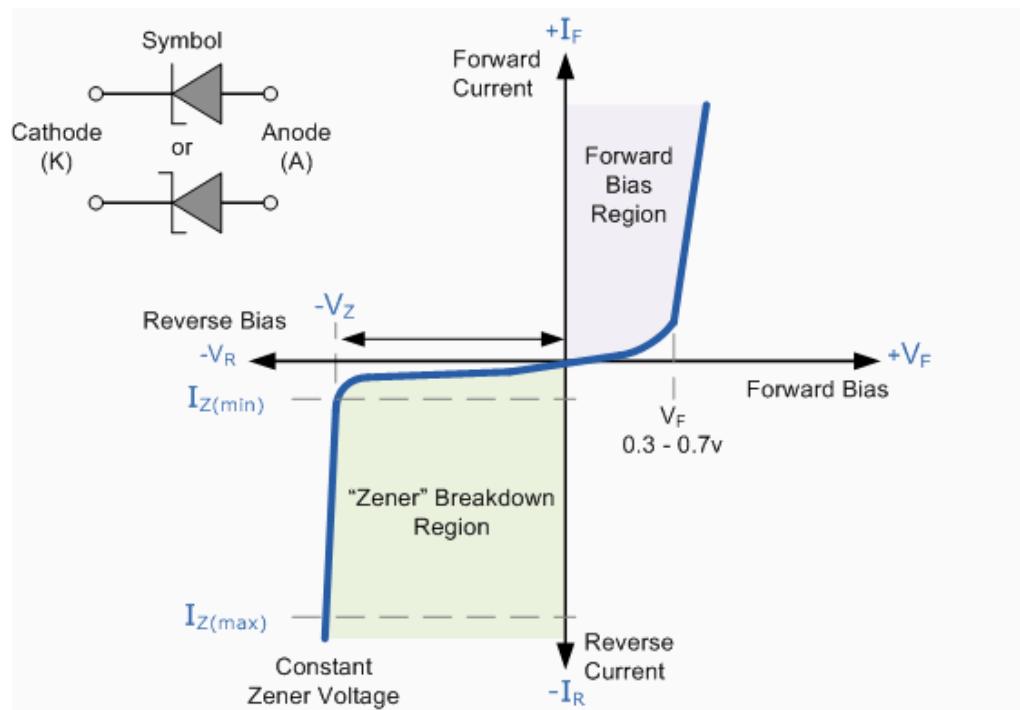


Fig 1.19 : Zener diode characteristics

The **Zener Diode** is used in its "reverse bias" or reverse breakdown mode, i.e. the diodes anode connects to the negative supply. From the I-V characteristics curve above, we can see that the zener

diode has a region in its reverse bias characteristics of almost a constant negative voltage regardless of the value of the current flowing through the diode and remains nearly constant even with large changes in current as long as the zener diodes current remains between the breakdown current $I_{Z(\min)}$ and the maximum current rating $I_{Z(\max)}$.

This ability to control itself can be used to great effect to regulate or stabilize a voltage source against supply or load variations. The fact that the voltage across the diode in the breakdown region is almost constant turns out to be an important application of the zener diode as a voltage regulator. The function of a regulator is to provide a constant output voltage to a load connected in parallel with it in spite of the ripples in the supply voltage or the variation in the load current and the zener diode will continue to regulate the voltage until the diodes current falls below the minimum $I_{Z(\min)}$ value in the reverse breakdown region.

UNIT II

BIPOLAR JUNCTION TRANSISTOR

2.1 INTRODUCTION

A bipolar junction transistor (BJT) is a three terminal device in which operation depends on the interaction of both majority and minority carriers and hence the name bipolar. The BJT is analogous to vacuum triode and is comparatively smaller in size. It is used as amplifier and oscillator circuits, and as a switch in digital circuits. It has wide applications in computers, satellites and other modern communication systems.

A **Transistor** is a three terminal semiconductor device that regulates current or voltage flow and acts as a switch or gate for signals.

Why Do We Need Transistors?

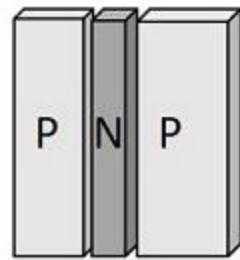
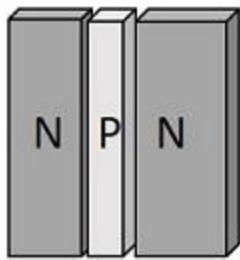
Suppose that you have a FM receiver which grabs the signal you want. The received signal will obviously be weak due to the disturbances it would face during its journey. Now if this signal is read as it is, you cannot get a fair output. Hence we need to amplify the signal. **Amplification** means increasing the signal strength.

This is just an instance. Amplification is needed wherever the signal strength has to be increased. This is done by a transistor. A transistor also acts as a **switch** to choose between available options. It also **regulates** the incoming **current and voltage** of the signals.

Constructional Details of a Transistor

The Transistor is a three terminal solid state device which is formed by connecting two diodes back to back. Hence it has got **two PN junctions**. Three terminals are drawn out of the three semiconductor materials present in it. This type of connection offers two types of transistors. They are **PNP** and **NPN** which means an N-type material between two P-types and the other is a P-type material between two N-types respectively.

The construction of transistors is as shown in the following figure which explains the idea discussed above.



Construction of PNP & NPN Transistors

The three terminals drawn from the transistor indicate Emitter, Base and Collector terminals. They have their functionality as discussed below.

Emitter

- The left hand side of the above shown structure can be understood as **Emitter**.
- This has a **moderate size** and is **heavily doped** as its main function is to **supply** a number of **majority carriers**, i.e. either electrons or holes.
- As this emits electrons, it is called as an Emitter.
- This is simply indicated with the letter **E**.

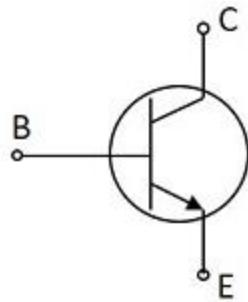
Base

- The middle material in the above figure is the **Base**.
- This is **thin** and **lightly doped**.
- Its main function is to **pass** the majority carriers from the emitter to the collector.
- This is indicated by the letter **B**.

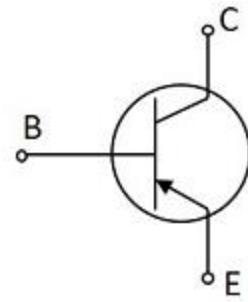
Collector

- The right side material in the above figure can be understood as a **Collector**.
- Its name implies its function of **collecting the carriers**.
- This is **a bit larger** in size than emitter and base. It is **moderately doped**.
- This is indicated by the letter **C**.

The symbols of PNP and NPN transistors are as shown below.



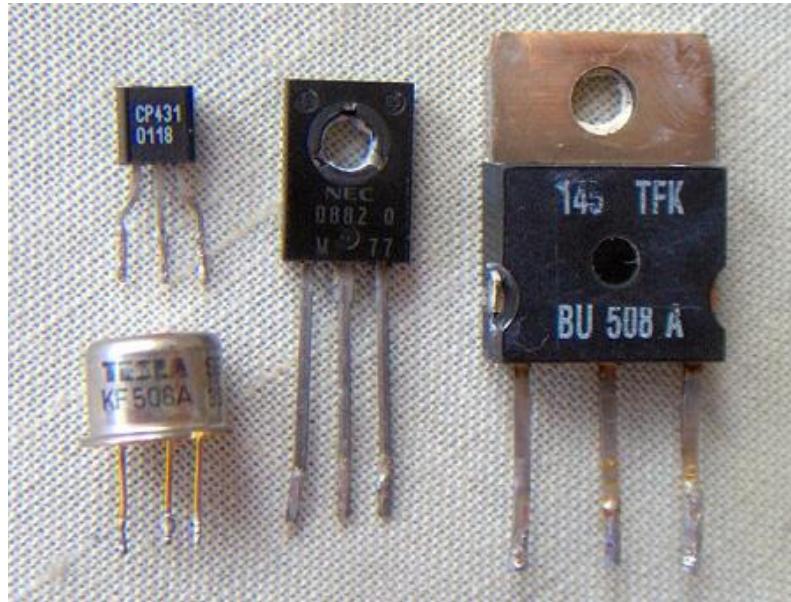
Symbol of
NPN transistor



Symbol of
PNP transistor

The **arrow-head** in the above figures indicated the **emitter** of a transistor. As the collector of a transistor has to dissipate much greater power, it is made large. Due to the specific functions of emitter and collector, they are **not interchangeable**. Hence the terminals are always to be kept in mind while using a transistor.

In a Practical transistor, there is a notch present near the emitter lead for identification. The PNP and NPN transistors can be differentiated using a Multimeter. The following figure shows how different practical transistors look like.

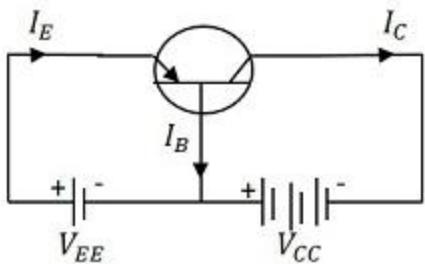
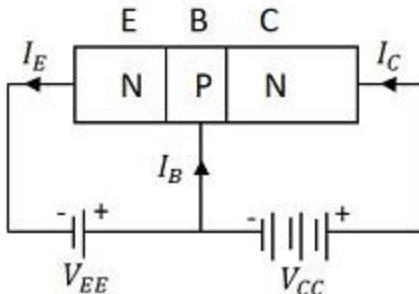
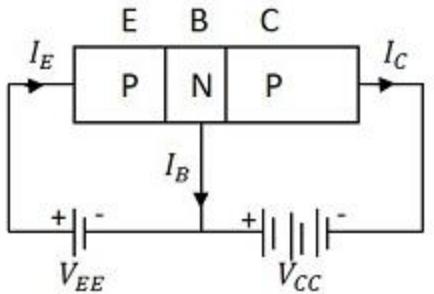


We have so far discussed the constructional details of a transistor, but to understand the operation of a transistor, first we need to know about the biasing.

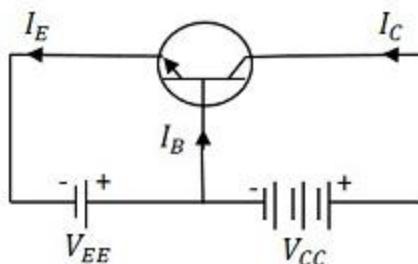
Transistor Biasing

As we know that a transistor is a combination of two diodes, we have two junctions here. As one junction is between the emitter and base, that is called as **Emitter-Base junction** and likewise, the other is **Collector-Base junction**.

Biassing is controlling the operation of the circuit by providing power supply. The function of both the PN junctions is controlled by providing bias to the circuit through some dc supply. The figure below shows how a transistor is biased.



P-N-P Transistor biasing



N-P-N Transistor biasing

By having a look at the above figure, it is understood that

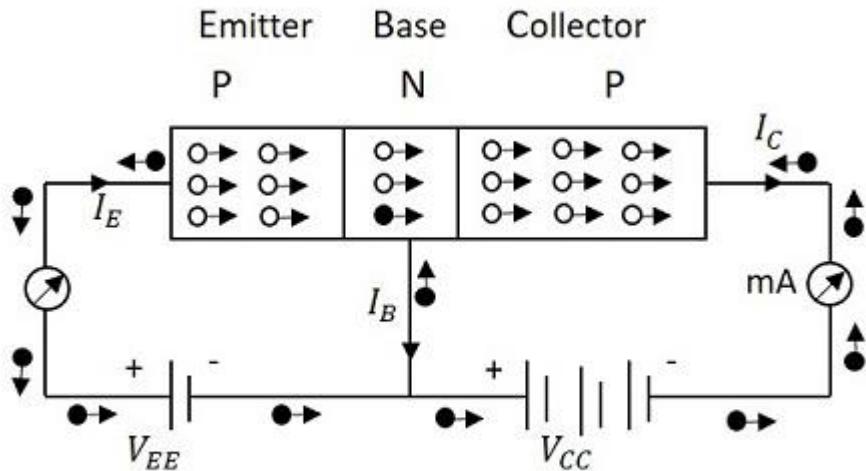
- The N-type material is provided negative supply and P-type material is given positive supply to make the circuit **Forward bias**.
- The N-type material is provided positive supply and P-type material is given negative supply to make the circuit **Reverse bias**.

By applying the power, the **emitter base junction** is always **forward biased** as the emitter resistance is very small. The **collector base junction** is **reverse biased** and its resistance is a bit higher. A small forward bias is sufficient at the emitter junction whereas a high reverse bias has to be applied at the collector junction.

The direction of current indicated in the circuits above, also called as the **Conventional Current**, is the movement of hole current which is **opposite to the electron current**.

Operation PNP Transistor

The operation of a PNP transistor can be explained by having a look at the following figure, in which emitter-base junction is forward biased and collector-base junction is reverse biased.



Operation of a PNP transistor

The voltage V_{EE} provides a positive potential at the emitter which repels the holes in the P-type material and these holes cross the emitter-base junction, to reach the base region. There a very low percent of holes recombine with free electrons of N-region. This provides very low current which constitutes the base current I_B . The remaining holes cross the collector-base junction, to constitute collector current I_C , which is the hole current.

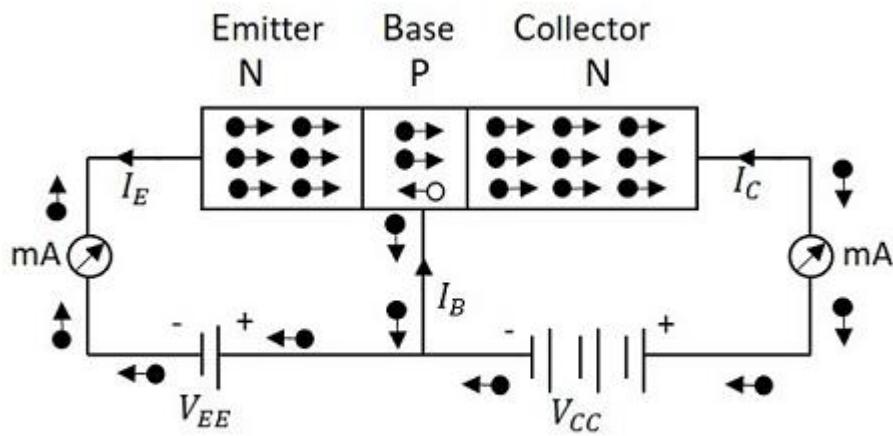
As a hole reaches the collector terminal, an electron from the battery negative terminal fills the space in the collector. This flow slowly increases and the electron minority current flows through the emitter, where each electron entering the positive terminal of V_{EE} , is replaced by a hole by moving towards the emitter junction. This constitutes emitter current I_E .

Hence we can understand that –

- The conduction in a PNP transistor takes place through holes.
- The collector current is slightly less than the emitter current.
- The increase or decrease in the emitter current affects the collector current.

Operation NPN Transistor

The operation of an NPN transistor can be explained by having a look at the following figure, in which emitter-base junction is forward biased and collector-base junction is reverse biased.



Operation of a NPN transistor

The voltage V_{EE} provides a negative potential at the emitter which repels the electrons in the N-type material and these electrons cross the emitter-base junction, to reach the base region. There a very low percent of electrons recombine with free holes of P-region. This provides very low current which constitutes the base current I_B . The remaining holes cross the collector-base junction, to constitute the collector current I_C .

As an electron reaches out of the collector terminal, and enters the positive terminal of the battery, an electron from the negative terminal of the battery V_{EE} enters the emitter region. This flow slowly increases and the electron current flows through the transistor.

Hence we can understand that –

- The conduction in a NPN transistor takes place through electrons.
- The collector current is higher than the emitter current.
- The increase or decrease in the emitter current affects the collector current.

Advantages

There are many advantages of a transistor such as –

- High voltage gain.
- Lower supply voltage is sufficient.
- Most suitable for low power applications.
- Smaller and lighter in weight.
- Mechanically stronger than vacuum tubes.
- No external heating required like vacuum tubes.
- Very suitable to integrate with resistors and diodes to produce ICs.

There are few disadvantages such as they cannot be used for high power applications due to lower power dissipation. They have lower input impedance and they are temperature dependent.

2.3 TRANSISTOR CURRENT COMPONENTS:

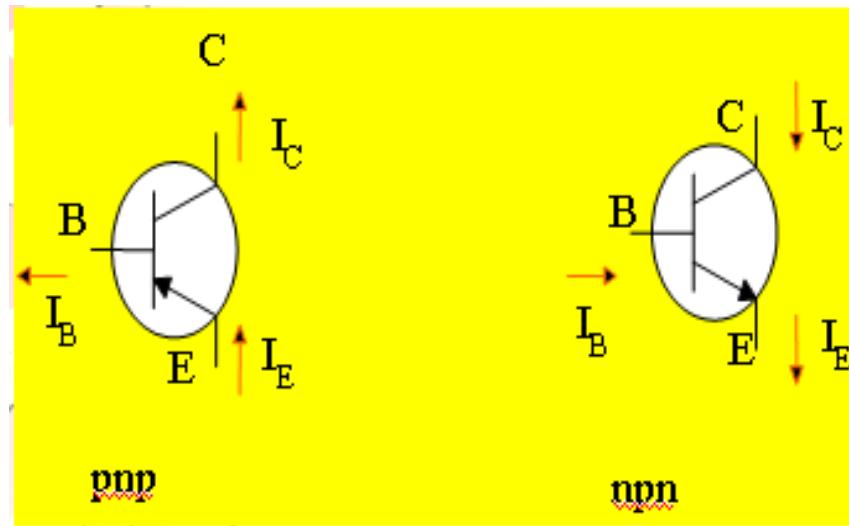


Fig 2.2 Bipolar Junction Transistor Current Components

The above fig 3.2 shows the various current components, which flow across the forward biased emitter junction and reverse- biased collector junction. The emitter current I_E consists of hole current I_{pE} (holes crossing from emitter into base) and electron current I_{nE} (electrons crossing from base into emitter). The ratio of hole to electron currents, I_{pE} / I_{nE} , crossing the emitter junction is proportional to the ratio of the conductivity of the p material to that of the n material. In a transistor, the doping of that of the emitter is made much larger than the doping of the base. This feature ensures (in p-n-p transistor) that the emitter current consists an almost entirely of holes. Such a situation is desired since the current which results from electrons crossing the emitter junction from base to emitter do not contribute carriers, which can reach the collector.

Not all the holes crossing the emitter junction J_E reach the the collector junction J_C

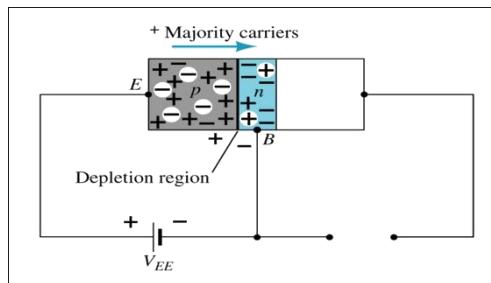
Because some of them combine with the electrons in n-type base. If I_{pC} is hole current at junction J_C there must be a bulk recombination current ($I_{pE} - I_{pC}$) leaving the base.

Actually, electrons enter the base region through the base lead to supply those charges, which have been lost by recombination with the holes injected in to the base across J_E . If the emitter were open circuited so that $I_E=0$ then I_{pC} would be zero. Under these circumstances, the base and collector current I_C would equal the reverse saturation current I_{CO} . If $I_E \neq 0$ then

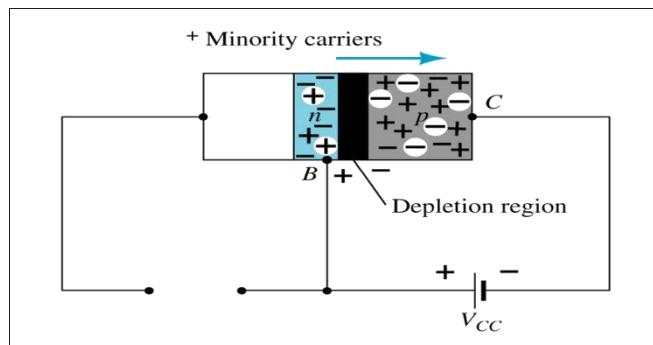
$$I_C = I_{CO} - I_{pC}$$

For a p-n-p transistor, I_{CO} consists of holes moving across J_C from left to right (base to collector) and electrons crossing J_C in opposite direction. Assumed referenced direction for I_{CO} i.e. from right to left, then for a p-n-p transistor, I_{CO} is negative. For an n-p-n transistor, I_{CO} is positive. The basic operation will be described using the pnp transistor. The operation of the pnp transistor is exactly the same if the roles played by the electron and hole are interchanged.

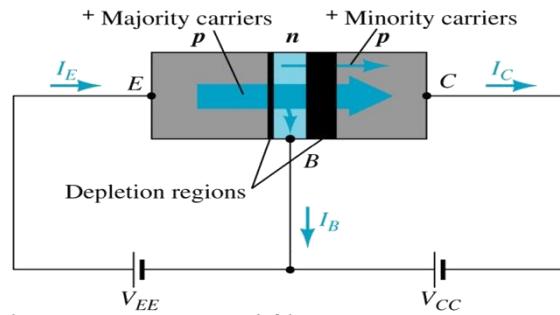
One p-n junction of a transistor is reverse-biased, whereas the other is forward-biased.



3.3a Forward-biased junction of a pnp transistor



2.3b Reverse-biased junction of a pnp transistor



2.3c Both biasing potentials have been applied to a pnp transistor and resulting majority and minority carrier flows indicated.

Majority carriers (+) will diffuse across the forward-biased p-n junction into the n-type material.

A very small number of carriers (+) will through n-type material to the base terminal. Resulting I_B is typically in order of microamperes.

The large number of majority carriers will diffuse across the reverse-biased junction into the p-type material connected to the collector terminal

Applying KCL to the transistor :

$$I_E = I_C + I_B$$

The comprises of two components – the majority and minority carriers

$$I_C = I_{C\text{majority}} + I_{C\text{Ominority}}$$

I_{CO} – I_C current with emitter terminal open and is called leakage current

Various parameters which relate the current components is given below

Emitter efficiency:

$$\gamma = \frac{\text{current of injected carriers at } J_E}{\text{total emitter current}}$$

$$\gamma = \frac{I_{pE}}{I_{pE} + I_{nE}} = \frac{I_{pE}}{I_{nE}}$$

Transport Factor:

$$\beta^* = \frac{\text{injected carrier current reaching } J_C}{\text{injected carrier current at } J_E}$$

$$\beta^* = \frac{I_{pC}}{I_{nE}}$$

Large signal current gain:

The ratio of the negative of collector current increment to the emitter current change from zero (cut-off) to I_E the large signal current gain of a common base transistor.

$$\alpha = \frac{-(I_C - I_{CO})}{I_E}$$

Since I_C and I_E have opposite signs, then α , as defined, is always positive. Typically numerical values of α lies in the range of 0.90 to 0.995

$$\alpha = \frac{I_{pC}}{I_E} = \frac{I_{pC}}{I_{nE}} * \frac{I_{pE}}{I_E} \quad \alpha = \beta * \gamma$$

The transistor alpha is the product of the transport factor and the emitter efficiency. This statement assumes that the collector multiplication ratio α^* is unity. α^* is the ratio of total current crossing J_C to hole arriving at the junction.

2.4 Bipolar Transistor Configurations

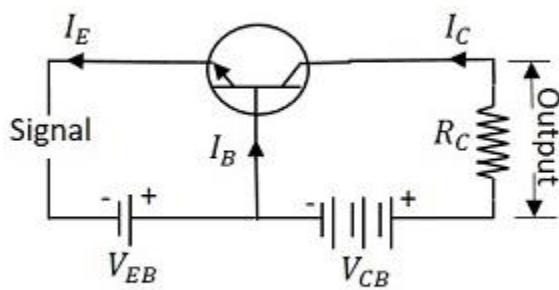
A Transistor has 3 terminals, the emitter, the base and the collector. Using these 3 terminals the transistor can be connected in a circuit with one terminal common to both input and output in a 3 different possible configurations.

The three types of configurations are **Common Base**, **Common Emitter** and **Common Collector** configurations. In every configuration, the emitter junction is forward biased and the collector junction is reverse biased.

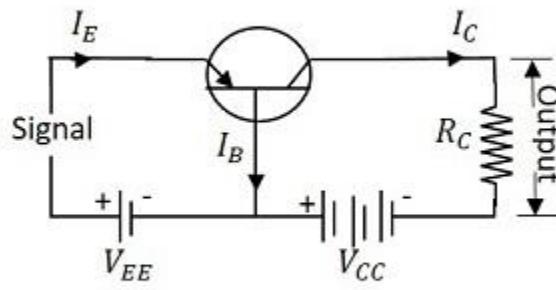
Common Base (CB) Configuration

The name itself implies that the Base terminal is taken as common terminal for both input and output of the transistor. The common base connection for both NPN and PNP transistors is as shown in the following figure.

Common Base Connection



Using NPN transistor



Using PNP transistor

For the sake of understanding, let us consider NPN transistor in CB configuration. When the emitter voltage is applied, as it is forward biased, the electrons from the negative terminal repel the emitter electrons and current flows through the emitter and base to the collector to contribute collector current. The collector voltage V_{CB} is kept constant throughout this.

In the CB configuration, the input current is the emitter current I_E and the output current is the collector current I_C .

Current Amplification Factor (α)

The ratio of change in collector current ($\Delta I_C / I_C$) to the change in emitter current ($\Delta I_E / I_E$) when collector voltage V_{CB} is kept constant, is called as **Current amplification factor**. It is denoted by α .

$$\alpha = \frac{\Delta I_C}{\Delta I_E} \text{ at constant } V_{CB}$$

Expression for Collector current

With the idea above, let us try to draw some expression for collector current. Along with the emitter current flowing, there is some amount of base current I_B which flows through the base terminal due to electron hole recombination. As collector-base junction is reverse biased, there is another current which is flown due to minority charge carriers. This is the leakage current which can be understood as $I_{leakage}$. This is due to minority charge carriers and hence very small.

The emitter current that reaches the collector terminal is

$$\alpha I_E$$

Total collector current

$$I_C = \alpha I_E + I_{leakage}$$

If the emitter-base voltage $V_{EB} = 0$, even then, there flows a small leakage current, which can be termed as I_{CBO} (collector-base current with output open).

The collector current therefore can be expressed as

$$I_C = \alpha I_E + I_{CBO}$$

$$I_E = I_C + I_B$$

$$I_C = \alpha(I_C + I_B) + I_{CBO}$$

$$I_C(1 - \alpha) = \alpha I_B + I_{CBO}$$

$$I_C = \left(\frac{\alpha}{1 - \alpha}\right) I_B + \left(\frac{I_{CBO}}{1 - \alpha}\right)$$

$$I_C = \left(\frac{\alpha}{1 - \alpha}\right) I_B + \left(\frac{1}{1 - \alpha}\right) I_{CBO}$$

Hence the above derived is the expression for collector current. The value of collector current depends on base current and leakage current along with the current amplification factor of that transistor in use.

Characteristics of CB configuration

- This configuration provides voltage gain but no current gain.
- Being V_{CB} constant, with a small increase in the Emitter-base voltage V_{EB} , Emitter current I_E gets increased.
- Emitter Current I_E is independent of Collector voltage V_{CB} .
- Collector Voltage V_{CB} can affect the collector current I_C only at low voltages, when V_{EB} is kept constant.
- The input resistance r_i is the ratio of change in emitter-base voltage (ΔV_{EB}) to the change in emitter current (ΔI_E) at constant collector base voltage V_{CB} .

$$= \frac{\Delta V_{EB}}{\Delta I_E} \text{ at constant } V_{CB}$$

- As the input resistance is of very low value, a small value of V_{EB} is enough to produce a large current flow of emitter current I_E .

- The output resistance r_o is the ratio of change in the collector base voltage (ΔV_{CB}) to the change in collector current (ΔI_C) at constant emitter current I_E .

$$r_o = \frac{\Delta V_{CB}}{\Delta I_C} \text{ at constant } I_E$$

- As the output resistance is of very high value, a large change in V_{CB} produces a very little change in collector current I_C .
- This Configuration provides good stability against increase in temperature.
- The CB configuration is used for high frequency applications.

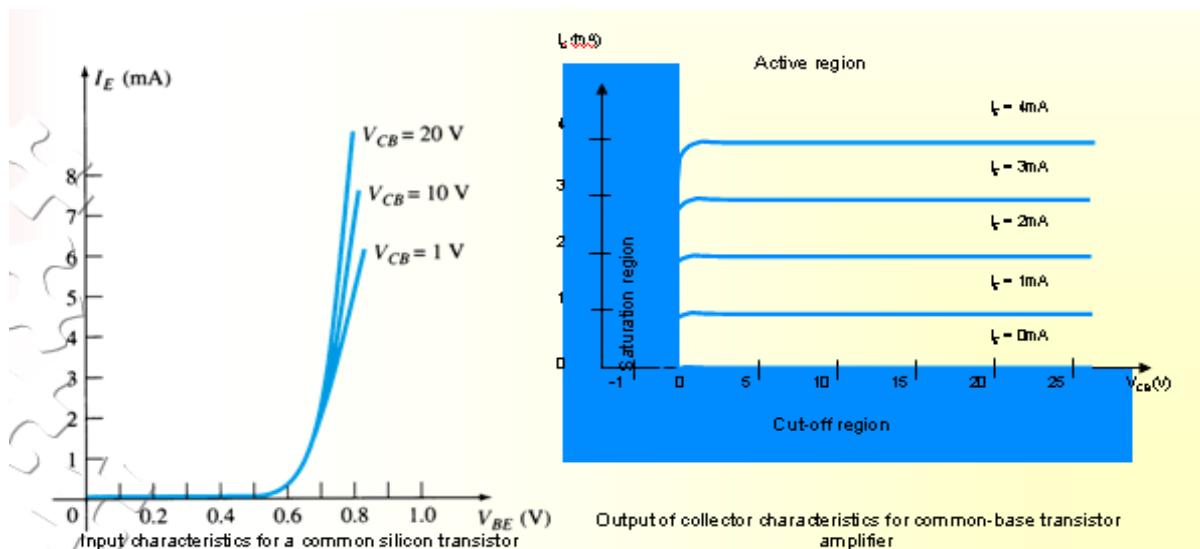
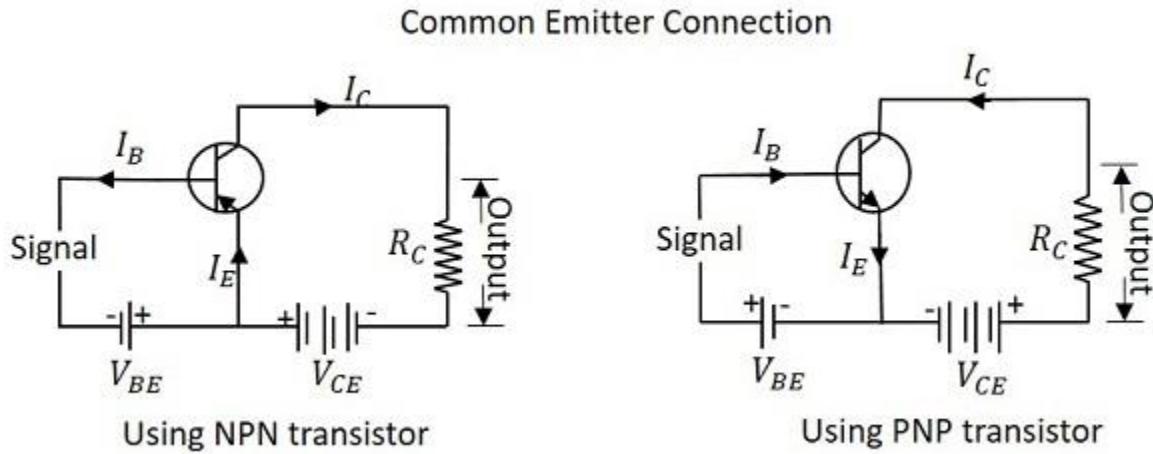


Fig 2.5 CB Input-Output Characteristics

Active region	Saturation region	Cut-off region
<ul style="list-style-type: none"> I_E increased, I_C increased BE junction forward bias and CB junction reverse bias Refer to the graf, $I_C \approx I_E$ I_C not depends on V_{CB} Suitable region for the transistor working as amplifier 	<ul style="list-style-type: none"> BE and CB junction is forward bias Small changes in V_{CB} will cause big different to I_C The allocation for this region is to the left of $V_{CB} = 0$ V. 	<ul style="list-style-type: none"> Region below the line of $I_E = 0$ A BE and CB is reverse bias no current flow at collector, only leakage current

Common Emitter (CE) Configuration

The name itself implies that the **Emitter** terminal is taken as common terminal for both input and output of the transistor. The common emitter connection for both NPN and PNP transistors is as shown in the following figure.



Just as in CB configuration, the emitter junction is forward biased and the collector junction is reverse biased. The flow of electrons is controlled in the same manner. The input current is the base current I_B and the output current is the collector current I_C here.

Base Current Amplification factor (β)

The ratio of change in collector current (ΔI_C) to the change in base current (ΔI_B) is known as **Base Current Amplification Factor**. It is denoted by β

$$\beta = \frac{\Delta I_C}{\Delta I_B}$$

Relation between β and α

Let us try to derive the relation between base current amplification factor and emitter current amplification factor.

$$\beta = \frac{\Delta I_C}{\Delta I_B}$$

$$\alpha = \frac{\Delta I_C}{\Delta I_E}$$

$$I_E = I_B + I_C$$

$$\Delta I_E = \Delta I_B + \Delta I_C$$

$$\Delta I_B = \Delta I_E - \Delta I_C$$

We can write

$$\beta = \frac{\Delta I_C}{\Delta I_E - \Delta I_C}$$

$$\beta = \frac{\frac{\Delta I_C}{\Delta I_E}}{\frac{\Delta I_E}{\Delta I_E} - \frac{\Delta I_C}{\Delta I_E}}$$

$$\alpha = \frac{\Delta I_C}{\Delta I_E}$$

We have

$$\alpha = \frac{\Delta I_C}{\Delta I_E}$$

Therefore,

$$\beta = \frac{\alpha}{1 - \alpha}$$

From the above equation, it is evident that, as α approaches 1, β reaches infinity.

Hence, **the current gain in Common Emitter connection is very high**. This is the reason this circuit connection is mostly used in all transistor applications.

Expression for Collector Current

In the Common Emitter configuration, I_B is the input current and I_C is the output current.

We know

$$I_E = I_B + I_C$$

And

$$\begin{aligned} I_C &= \alpha I_E + I_{CBO} \\ &= \alpha(I_B + I_C) + I_{CBO} \\ I_C(1 - \alpha) &= \alpha I_B + I_{CBO} \\ I_C &= \frac{\alpha}{1 - \alpha} I_B + \frac{1}{1 - \alpha} I_{CBO} \end{aligned}$$

If base circuit is open, i.e. if $I_B = 0$,

The collector emitter current with base open is I_{CEO}

$$I_{CEO} = \frac{1}{1 - \alpha} I_{CBO}$$

Substituting the value of this in the previous equation, we get

$$I_C = \frac{\alpha}{1 - \alpha} I_B + I_{CEO}$$

$$I_C = \beta I_B + I_{CEO}$$

Hence the equation for collector current is obtained.

Knee Voltage

In CE configuration, by keeping the base current I_B constant, if V_{CE} is varied, I_C increases nearly to 1v of V_{CE} and stays constant thereafter. This value of V_{CE} up to which collector current I_C changes with V_{CE} is called the **Knee Voltage**. The transistors while operating in CE configuration, they are operated above this knee voltage.

Characteristics of CE Configuration

- This configuration provides good current gain and voltage gain.
- Keeping V_{CE} constant, with a small increase in V_{BE} the base current I_B increases rapidly than in CB configurations.
- For any value of V_{CE} above knee voltage, I_C is approximately equal to βI_B .
- The input resistance r_i is the ratio of change in base emitter voltage (ΔV_{BE}) to the change in base current (ΔI_B) at constant collector emitter voltage V_{CE} .

$$r_i = \frac{\Delta V_{BE}}{\Delta I_B} \text{ at constant } V_{CE}$$

- As the input resistance is of very low value, a small value of V_{BE} is enough to produce a large current flow of base current I_B .
- The output resistance r_o is the ratio of change in collector emitter voltage (ΔV_{CE}) to the change in collector current (ΔI_C) at constant I_B .

$$r_o = \frac{\Delta V_{CE}}{\Delta I_C} \text{ at constant } I_B$$

- As the output resistance of CE circuit is less than that of CB circuit.

- This configuration is usually used for bias stabilization methods and audio frequency applications.

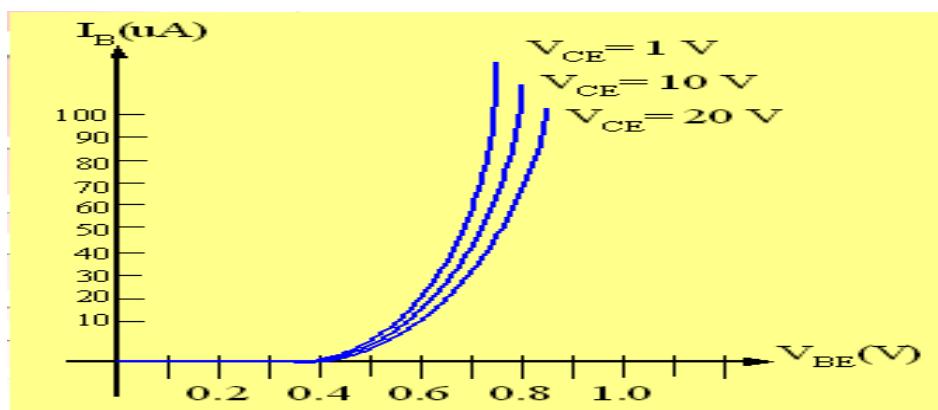


Fig 3.9a Input characteristics for common-emitter npn transistor

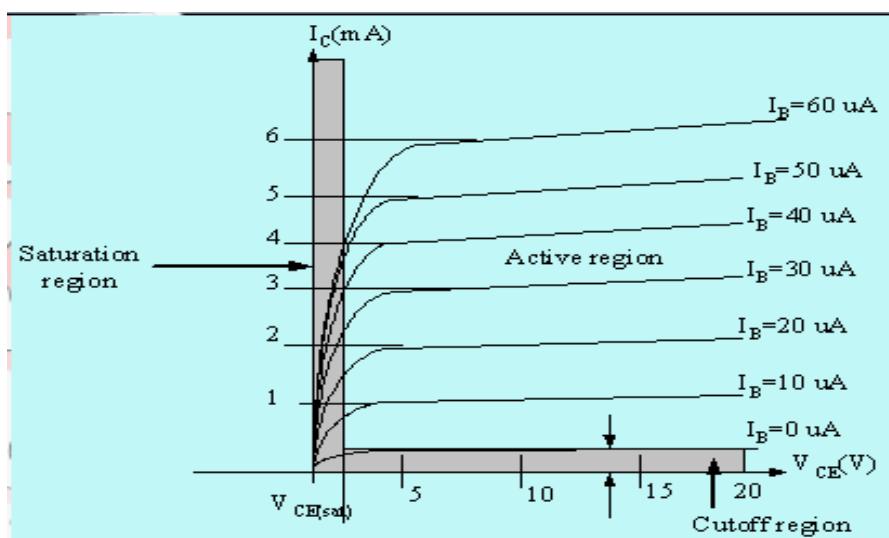


Fig 2.9b Output characteristics for common-emitter npn transistor

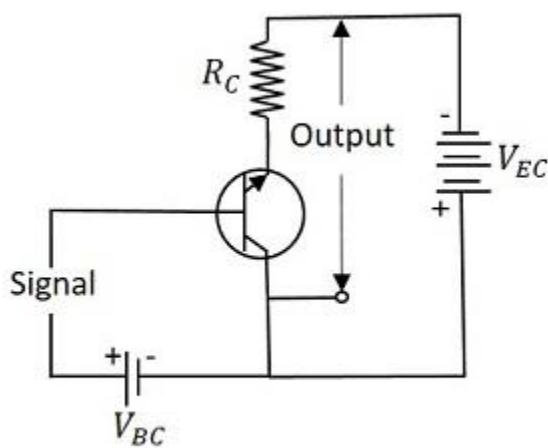
Active region	Saturation region	Cut-off region
<ul style="list-style-type: none"> B-E junction is forward bias C-B junction is reverse bias can be employed for voltage, current and power amplification 	<ul style="list-style-type: none"> B-E and C-B junction is forward bias, thus the values of I_B and I_C is too big. The value of V_{CE} is so small. Suitable region when the transistor as a logic switch. NOT and avoid this region when the transistor as an amplifier. 	<ul style="list-style-type: none"> region below $I_B=0\mu A$ is to be avoided if an undistorted o/p signal is required B-E junction and C-B junction is reverse bias $I_B=0$, I_C not zero, during this condition $I_C=I_{CEO}$ where is this current flow when B-E is reverse bias.

2.7 COMMON – COLLECTOR CONFIGURATION

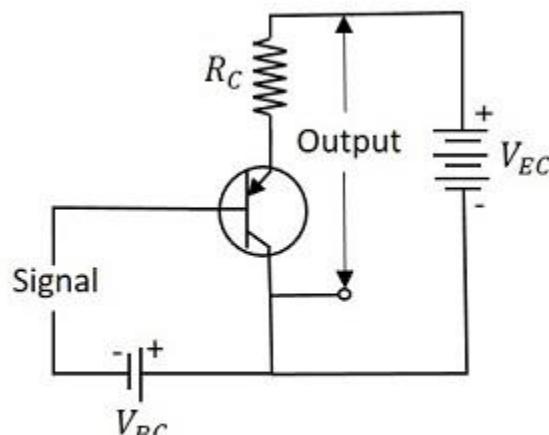
Common Collector (CC) Configuration

The name itself implies that the **Collector** terminal is taken as common terminal for both input and output of the transistor. The common collector connection for both NPN and PNP transistors is as shown in the following figure.

Common Collector Connection



Using NPN transistor



Using PNP transistor

Just as in CB and CE configurations, the emitter junction is forward biased and the collector junction is reverse biased. The flow of electrons is controlled in the same manner. The input current is the base current I_B and the output current is the emitter current I_E here.

Current Amplification Factor (γ)

The ratio of change in emitter current ($\Delta I_E \Delta I_E$) to the change in base current ($\Delta I_B \Delta I_B$) is known as **Current Amplification factor** in common collector (CC) configuration. It is denoted by γ .

$$\gamma = \frac{\Delta I_E}{\Delta I_B}$$

- The current gain in CC configuration is same as in CE configuration.
- The voltage gain in CC configuration is always less than 1.

Relation between γ and α

Let us try to draw some relation between γ and α

$$\gamma = \frac{\Delta I_E}{\Delta I_B}$$

$$\alpha = \frac{\Delta I_C}{\Delta I_E}$$

$$I_E = I_B + I_C$$

$$\Delta I_E = \Delta I_B + \Delta I_C$$

$$\Delta I_B = \Delta I_E - \Delta I_C$$

Substituting the value of I_B , we get

$$\gamma = \frac{\Delta I_E}{\Delta I_E - \Delta I_C}$$

Dividing by ΔI_E

$$\gamma = \frac{\frac{\Delta I_E}{\Delta I_B}}{\frac{\Delta I_E}{\Delta I_E} - \frac{\Delta I_C}{\Delta I_E}}$$

$$\frac{1}{1 - \alpha}$$

$$\gamma = \frac{1}{1 - \alpha}$$

Expression for collector current

We know

$$I_C = \alpha I_E + I_{CBO}$$

$$I_E = I_B + I_C = I_B + (\alpha I_E + I_{CBO})$$

$$I_E(1 - \alpha) = I_B + I_{CBO}$$

$$I_E = \frac{I_B}{1 - \alpha} + \frac{I_{CBO}}{1 - \alpha}$$

$$I_C \cong I_E = (\beta + 1)I_B + (\beta + 1)I_{CBO}$$

The above is the expression for collector current.

Characteristics of CC Configuration

- This configuration provides current gain but no voltage gain.
- In CC configuration, the input resistance is high and the output resistance is low.
- The voltage gain provided by this circuit is less than 1.
- The sum of collector current and base current equals emitter current.
- The input and output signals are in phase.
- This configuration works as non-inverting amplifier output.
- This circuit is mostly used for impedance matching. That means, to drive a low impedance load from a high impedance source.

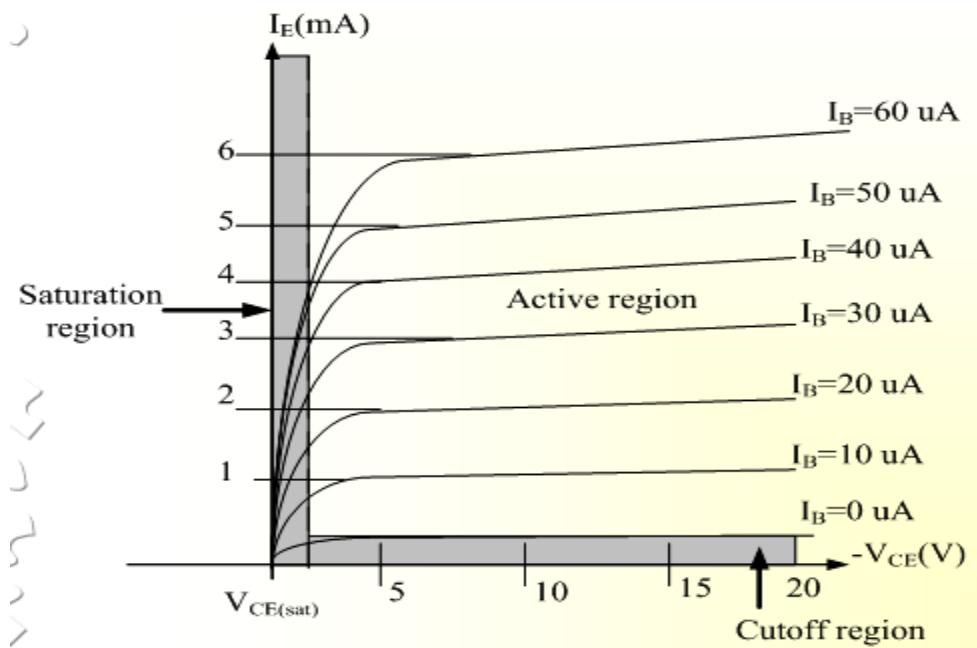


Fig 2.11 Output Characteristics of CC Configuration for npn Transistor

UNIT III

FIELD EFFECT TRANSISTOR

INTRODUCTION

1. The Field effect transistor is abbreviated as FET , it is an another semiconductor device like a BJT which can be used as an amplifier or switch.
2. The Field effect transistor is a voltage operated device. Whereas Bipolar junction transistor is a current controlled device. Unlike BJT a FET requires virtually no input current.
3. This gives it an extremely high input resistance , which is its most important advantage over a bipolar transistor.
4. FET is also a three terminal device, labeled as source, drain and gate.
5. The source can be viewed as BJT's emitter, the drain as collector, and the gate as the counter part of the base.
6. The material that connects the source to drain is referred to as the channel.
7. FET operation depends only on the flow of majority carriers ,therefore they are called uni polar devices. BJT operation depends on both minority and majority carriers.
8. As FET has conduction through only majority carriers it is less noisy than BJT.
9. FETs are much easier to fabricate and are particularly suitable for ICs because they occupy less space than BJTs.
10. FET amplifiers have low gain bandwidth product due to the junction capacitive effects and produce more signal distortion except for small signal operation.
11. The performance of FET is relatively unaffected by ambient temperature changes. As it has a negative temperature coefficient at high current levels, it prevents the FET from thermal breakdown. The BJT has a positive temperature coefficient at high current levels which leads to thermal breakdown.

CLASSIFICATION OF FET:

There are two major categories of field effect transistors:

1. Junction Field Effect Transistors
2. MOSFETs

These are further sub divided in to P- channel and N-channel devices.

MOSFETs are further classified in to two types Depletion MOSFETs and Enhancement .

MOSFETs

The schematic symbols for the P-channel and N-channel JFETs are shown in the figure.



Fig 4.1 schematic symbols for the P-channel and N-channel JFET

CONSTRUCTION AND OPERATION OF N- CHANNEL FET

If the gate is an N-type material, the channel must be a P-type material.

CONSTRUCTION OF N-CHANNEL JFET

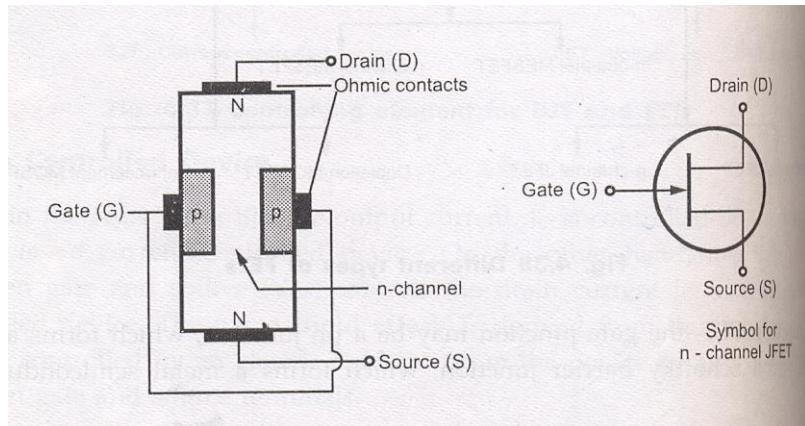


Fig 4.2 Construction of N-Channel JFET

A piece of N-type material, referred to as channel has two smaller pieces of P-type material attached to its sides, forming PN junctions. The channel ends are designated as the drain and source. And the two pieces of P-type material are connected together and their terminal is called the gate. Since this channel is in the N-type bar, the FET is known as N-channel JFET.

OPERATION OF N-CHANNEL JFET:-

The overall operation of the JFET is based on varying the width of the channel to control the drain current.

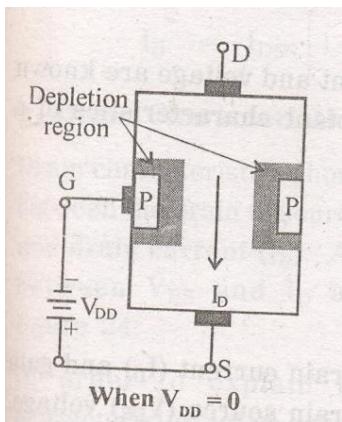
A piece of N type material referred to as the channel, has two smaller pieces of P type material attached to its sites, forming PN –Junctions. The channel's ends are designated the drain and the source. And the two pieces of P type material are connected together and their terminal is called the gate. With the gate terminal not connected and the potential applied positive at the drain negative at the source a drain current I_D flows. When the gate is biased negative with respective to the source the PN junctions are reverse biased and depletion regions are formed. The channel is more lightly doped than the P type gate blocks, so the depletion regions penetrate deeply into the channel. Since depletion region is a region depleted of charge carriers it behaves as an Insulator. The result is that the channel is narrowed. Its resistance is increased and I_D is reduced. When the negative gate bias voltage is further increased, the depletion regions meet at the center and I_D is cut off completely.

There are two ways to control the channel width

1. By varying the value of V_{GS}
2. And by Varying the value of V_{DS} holding V_{GS} constant

1 By varying the value of V_{GS} :-

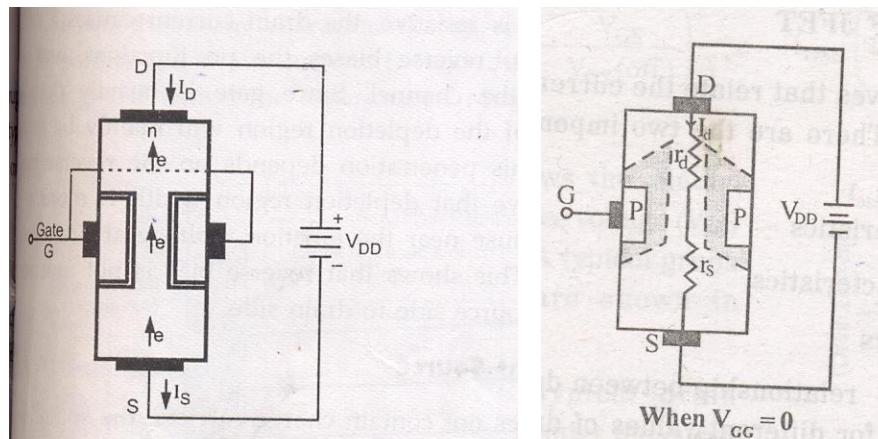
We can vary the width of the channel and in turn vary the amount of drain current. This can be done by varying the value of V_{GS} . This point is illustrated in the fig below. Here we are dealing with N channel FET. So channel is of N type and gate is of P type that constitutes a PN junction. This PN junction is always reverse biased in JFET operation .The reverse bias is applied by a battery voltage V_{GS} connected between the gate and the source terminal i.e positive terminal of the battery is connected to the source and negative terminal to gate.



- 1) When a PN junction is reverse biased the electrons and holes diffuse across junction by leaving immobile ions on the N and P sides , the region containing these immobile ions is known as depletion regions.
- 2) If both P and N regions are heavily doped then the depletion region extends symmetrically on both sides.
- 3) But in N channel FET P region is heavily doped than N type thus depletion region extends more in N region than P region.
- 4) So when no V_{ds} is applied the depletion region is symmetrical and the conductivity becomes Zero. Since there are no mobile carriers in the junction.
- 5) As the reverse bias voltage is increases the thickness of the depletion region also increases. i.e. the effective channel width decreases .
- 6) By varying the value of V_{gs} we can vary the width of the channel.

2 Varying the value of V_{ds} holding V_{gs} constant :-

- 1) When no voltage is applied to the gate i.e. $V_{gs}=0$, V_{ds} is applied between source and drain the electrons will flow from source to drain through the channel constituting drain current I_d .
- 2) With $V_{gs}= 0$ for $I_d= 0$ the channel between the gate junctions is entirely open .In response to a small applied voltage V_{ds} , the entire bar acts as a simple semi conductor resistor and the current I_d increases linearly with V_{ds} .
- 3) The channel resistances are represented as r_d and r_s as shown in the fig.



In-channel JFET with gate open and V_{DD} applied between drain and source

- 4) This increasing drain current I_d produces a voltage drop across r_d which reverse biases the gate to source junction,($r_d > r_s$) .Thus the depletion region is formed which is not symmetrical .

- 5) The depletion region i.e. developed penetrates deeper in to the channel near drain and less towards source because $V_{rd} \gg V_{rs}$. So reverse bias is higher near drain than at source.
- 6) As a result growing depletion region reduces the effective width of the channel. Eventually a voltage V_{ds} is reached at which the channel is pinched off. This is the voltage where the current I_d begins to level off and approach a constant value.
- 7) So, by varying the value of V_{ds} we can vary the width of the channel holding V_{gs} constant.

When both V_{gs} and V_{ds} is applied:-

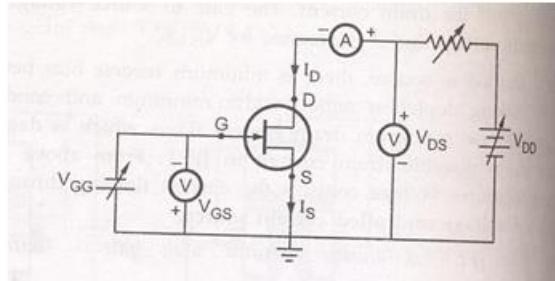
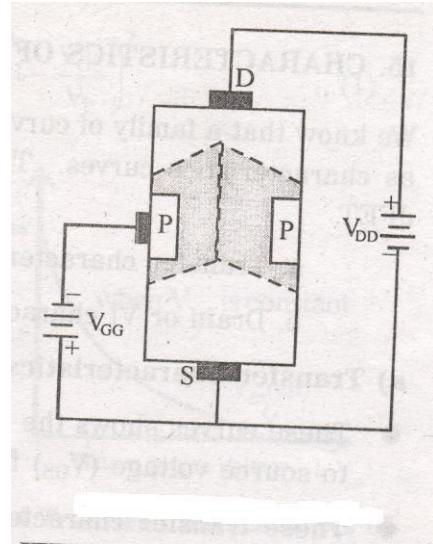


Fig Experimental setup to plot JFET characteristics

It is of course in principle not possible for the channel to close Completely and there by reduce the current I_d to Zero for, if such indeed, could be the case the gate voltage V_{gs} is applied in the direction to provide additional reverse bias

- 1) When voltage is applied between the drain and source with a battery V_{dd} , the electrons flow from source to drain through the narrow channel existing between the depletion regions. This constitutes the drain current I_d , its conventional direction is from drain to source.
- 2) The value of drain current is maximum when no external voltage is applied between gate and source and is designated by I_{dss} .



- 3) When V_{GS} is increased beyond Zero the depletion regions are widened. This reduces the effective width of the channel and therefore controls the flow of drain current through the channel.
- 4) When V_{GS} is further increased a stage is reached at which to depletion regions touch each other that means the entire channel is closed with depletion region. This reduces the drain current to Zero.

CHARACTERISTICS OF N-CHANNEL JFET

The family of curves that shows the relation between current and voltage are known as characteristic curves.

There are two important characteristics of a JFET.

- 1) Drain or VI Characteristics
- 2) Transfer characteristics

1. Drain Characteristics:-

2. Drain characteristics shows the relation between the drain to source voltage V_{DS} and drain current I_D . In order to explain typical drain characteristics let us consider the curve with $V_{GS} = 0\text{V}$.

- 1) When V_{DS} is applied and it is increasing the drain current I_D also increases linearly up to knee point.
- 2) This shows that FET behaves like an ordinary resistor. This region is called as ohmic region.
- 3) I_D increases with increase in drain to source voltage. Here the drain current is increased slowly as compared to ohmic region.

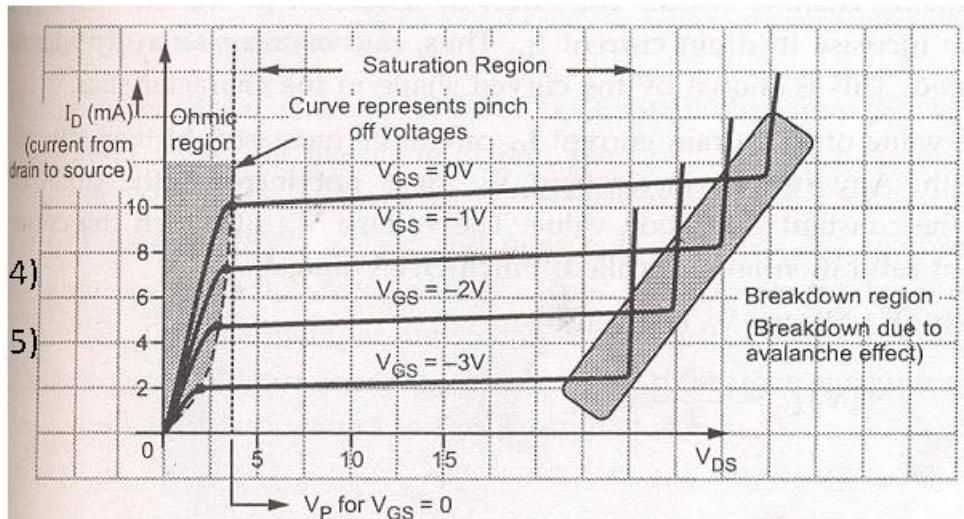


Fig. Drain V-I characteristics of n-channel JFET

Fig 4.4: V-I characteristics of n-channel JFET

- 4) It is because of the fact that there is an increase in V_{DS} . This in turn increases the reverse bias voltage across the gate source junction. As a result of this depletion region grows in size thereby reducing the effective width of the channel.
- 5) All the drain to source voltage corresponding to point the channel width is reduced to a minimum value and is known as pinch off.
- 5) The drain to source voltage at which channel pinch off occurs is called pinch off voltage(V_p).

PINCH OFF Region:-

- 1) This is the region shown by the curve as saturation region.
- 2) It is also called as saturation region or constant current region. Because of the channel is occupied with depletion region , the depletion region is more towards the drain and less towards the source, so the channel is limited, with this only limited number of carriers are only allowed to cross this channel from source drain causing a current that is constant in this region. To use FET as an amplifier it is operated in this saturation region.
- 3) In this drain current remains constant at its maximum value I_{DSS} .
- 4) The drain current in the pinch off region depends upon the gate to source voltage and is given by the relation

$$I_d = I_{DSS} [1 - V_{GS}/V_p]^2$$

This is known as shokley's relation.

BREAKDOWN REGION:-

- 1) The region is shown by the curve .In this region, the drain current increases rapidly as the drain to source voltage is increased.
- 2) It is because of the gate to source junction due to avalanche effect.

- 3) The avalanche break down occurs at progressively lower value of VDS because the reverse bias gate voltage adds to the drain voltage thereby increasing effective voltage across the gate junction

This causes

1. The maximum saturation drain current is smaller
2. The ohmic region portion decreased.

- 4) It is important to note that the maximum voltage VDS which can be applied to FET is the lowest voltage which causes available break down.

3. TRANSFER CHARACTERISTICS:-

These curves shows the relationship between drain current ID and gate to source voltage VGS for different values of VDS.

- 1) First adjust the drain to source voltage to some suitable value , then increase the gate to source voltage in small suitable value.
- 2) Plot the graph between gate to source voltage along the horizontal axis and current ID on the vertical axis. We shall obtain a curve like this.

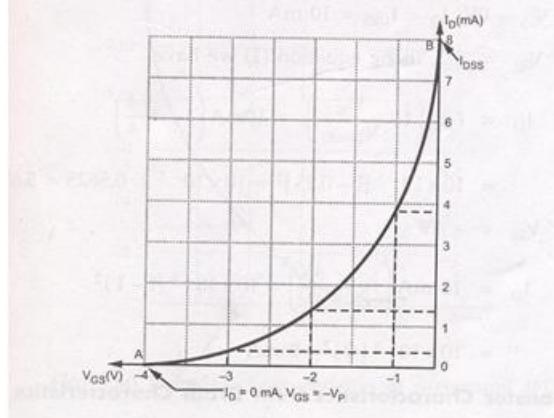


Fig 4.5: Transfer characteristics of n-channel JFET

- 3) As we know that if V_{GS} is more negative curves drain current to reduce . where V_{GS} is made sufficiently negative, I_d is reduced to zero. This is caused by the widening of the depletion region to a point where it is completely closes the channel. The value of V_{GS} at the cutoff point is designed as V_{GSOFF}

- 4) The upper end of the curve as shown by the drain current value is equal to I_{dss} that is when $V_{gs} = 0$ the drain current is maximum.
- 5) While the lower end is indicated by a voltage equal to V_{gsoff}
- 6) If V_{gs} continuously increasing, the channel width is reduced, then $I_d = 0$
- 7) It may be noted that curve is part of the parabola; it may be expressed as

$$I_d = I_{dss} [1 - V_{gs}/V_{gsoff}]^2$$

DIFFERENCE BETWEEN V_p AND V_{gsoff} –

V_p is the value of V_{gs} that causes the JFET to become constant current component, It is measured at $V_{gs} = 0V$ and has a constant drain current of $I_d = I_{dss}$. Where V_{gsoff} is the value of V_{gs} that reduces I_d to approximately zero.

Why the gate to source junction of a JFET be always reverse biased ?

The gate to source junction of a JFET is never allowed to become forward biased because the gate material is not designed to handle any significant amount of current. If the junction is allowed to become forward biased, current is generated through the gate material. This current may destroy the component.

There is one more important characteristic of JFET reverse biasing i.e. J FET's have extremely high characteristic gate input impedance. This impedance is typically in the high mega ohm range. With the advantage of extremely high input impedance it draws no current from the source. The high input impedance of the JFET has led to its extensive use in integrated circuits. The low current requirements of the component makes it perfect for use in ICs. Where thousands of transistors must be etched on to a single piece of silicon. The low current draw helps the IC to remain relatively cool, thus allowing more components to be placed in a smaller physical area.

JFET PARAMETERS

The electrical behavior of JFET may be described in terms of certain parameters. Such parameters are obtained from the characteristic curves.

A C Drain resistance(r_d):

It is also called dynamic drain resistance and is the a.c.resistance between the drain and source terminal,when the JFET is operating in the pinch off or saturation region. It is given by the ratio of small change in drain to source voltage ΔV_{ds} to the corresponding change in drain current ΔI_d for a constant gate to source voltage V_{gs} .

Mathematically it is expressed as $r_d = \Delta V_{ds} / \Delta I_d$ where V_{gs} is held constant.

TRANSE CONDUCTANCE (g_m):

It is also called forward transconductance . It is given by the ratio of small change in drain current (ΔI_d) to the corresponding change in gate to source voltage (ΔV_{ds})

Mathematically the transconductance can be written as

$$g_m = \Delta I_d / \Delta V_{ds}$$

AMPLIFICATION FACTOR (μ)

It is given by the ratio of small change in drain to source voltage (ΔV_{ds}) to the corresponding change in gate to source voltage (ΔV_{gs})for a constant drain current (I_d).

Thus $\mu = \Delta V_{ds} / \Delta V_{gs}$ when I_d held constant

The amplification factor μ may be expressed as a product of transconductance (g_m)and ac drain resistance (r_d)

$$\mu = \Delta V_{ds} / \Delta V_{gs} = g_m r_d$$

THE FET SMALL SIGNAL MODEL

The linear small signal equivalent circuit for the FET can be obtained in a manner similar to that used to derive the corresponding model for a transistor.

We can express the drain current i_D as a function f of the gate voltage and drain voltage V_{ds} .

$$I_d = f(V_{gs}, V_{ds}) \quad \dots \quad (1)$$

The transconductance g_m and drain resistance r_d :-

If both gate voltage and drain voltage are varied, the change in the drain current is approximated by using taylors series considering only the first two terms in the expansion

$$\Delta I_d = \frac{\partial i_d}{\partial V_{gs}} |_{V_{ds}=\text{constant}} \cdot \Delta V_{gs} + \frac{\partial i_d}{\partial V_{ds}} |_{V_{gs}=\text{constant}} \Delta V_{ds}$$

we can write $\Delta I_d = i_d$

$$\Delta V_{gs} = V_{gs}$$

$$\Delta V_{ds} = V_{ds}$$

$$I_d = g_m V_{gs} + \frac{1}{r_d} V_{ds} \rightarrow (1)$$

$$\text{Where } g_m = \frac{\partial i_d}{\partial V_{gs}} |_{V_{ds}} \approx \frac{\Delta i_d}{\Delta V_{gs}} |_{V_{ds}}$$

$$g_m = \frac{i_d}{V_{gs}} | V_{ds}$$

Is the mutual conductance or transconductance .It is also called as gfs or yfs common source forward conductance .

The second parameter r_d is the drain resistance or output resistance is defined as

$$r_d = \frac{\partial V_{ds}}{\partial i_d} | V_{gs} \cong \frac{\Delta V_{ds}}{\Delta i_d} | V_{gs} = \frac{V_{ds}}{i_d} | V_{gs}$$

$$r_d = \frac{V_{ds}}{i_d} | V_{gs}$$

The reciprocal of the r_d is the drain conductance g_d .It is also designated by Yos and Gos and called the common source output conductance . So the small signal equivalent circuit for FET can be drawn in two different ways.

- 1.small signal current –source model
- 2.small signal voltage-source model.

A small signal current –source model for FET in common source configuration can be drawn satisfying Eq→(1) as shown in the figure(a)

This low frequency model for FET has a Norton's output circuit with a dependent current generator whose magnitude is proportional to the gate-to –source voltage. The proportionality factor is the transconductance 'g_m'. The output resistance is 'r_d'. The input resistance between the gate and source is infinite, since it is assumed that the reverse biased gate draws no current. For the same reason the resistance between gate and drain is assumed to be infinite.

The small signal voltage-source model is shown in the figure(b).

This can be derived by finding the Thevenin's equivalent for the output part of fig(a) .

These small signal models for FET can be used for analyzing the three basic FET amplifier configurations:

- 1.common source (CS) 2.common drain (CD) or source follower
3. common gate(CG).

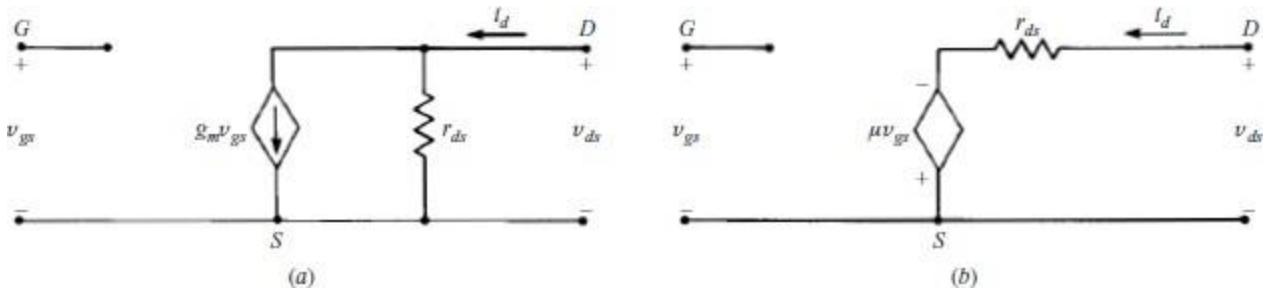


Fig 4.6 (a)Small Signal Current source model for FET (b)Small Signal voltage source model for FET

Here the input circuit is kept open because of having high input impedance and the output circuit satisfies the equation for ID

MOSFET

We now turn our attention to the insulated gate FET or metal oxide semi conductor FET which is having the greater commercial importance than the junction FET.

Most MOSFETS however are triodes, with the substrate internally connected to the source. The circuit symbols used by several manufacturers are indicated in the Fig below.



Fig 4.7(a)Depletion type MOSFET (b) Enhancement type MOSFET

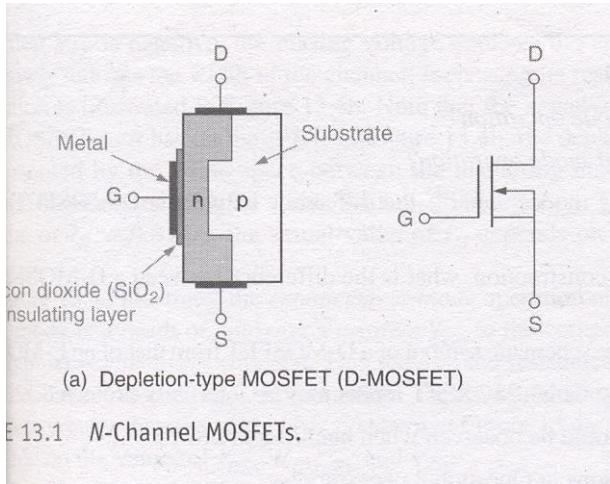
Both of them are P- channel

Here are two basic types of MOSFETS

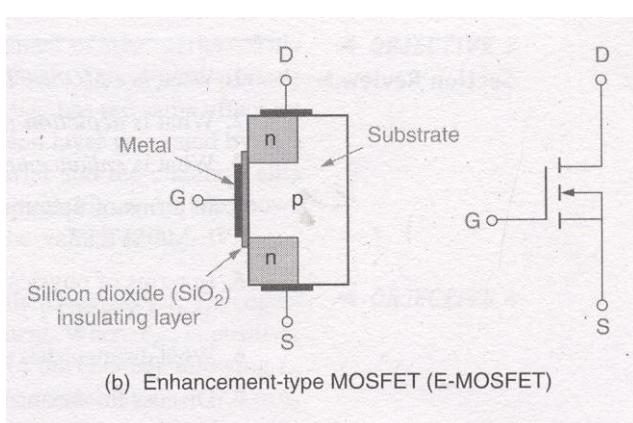
- (1) Depletion type (2) Enhancement type MOSFET.

D-MOSFETS can be operated in both the depletion mode and the enhancement mode. E MOSFETS are restricted to operate in enhancement mode. The primary difference between them is their physical construction.

The construction difference between the two is shown in the fig given below.



E 13.1 N-Channel MOSFETs.



The E MOSFET on the other hand has no such channel physically. It depends on the gate voltage to form a channel between the source and the drain terminals.

Both MOSFETS have an insulating layer between the gate and the rest of the component. This insulating layer is made up of SiO_2 a glass like insulating material. The gate material is made up of

metal conductor .Thus going from gate to substrate, we can have metal oxide semi conductor which is where the term MOSFET comes from.

Since the gate is insulated from the rest of the component, the MOSFET is sometimes referred to as an insulated gate FET or IGFET.

The foundation of the MOSFET is called the substrate. This material is represented in the schematic symbol by the center line that is connected to the source.

In the symbol for the MOSFET, the arrow is placed on the substrate. As with JFET an arrow pointing in represents an N-channel device, while an arrow pointing out represents p-channel device.

CONSTRUCTION OF AN N-CHANNEL MOSFET:-

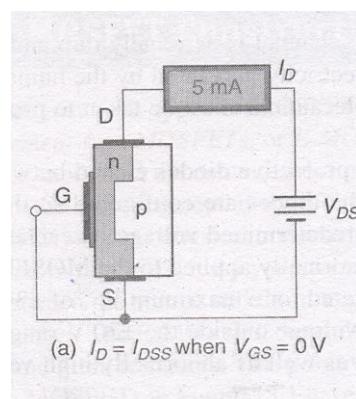
The N- channel MOSFET consists of a lightly doped p type substance into which two heavily doped n+ regions are diffused as shown in the Fig. These n+ sections , which will act as source and drain.

A thin layer of insulation silicon dioxide (SiO_2) is grown over the surface of the structure, and holes are cut into oxide layer, allowing contact with the source and drain. Then the gate metal area is overlaid on the oxide, covering the entire channel region.Metal contacts are made to drain and source and the contact to the metal over the channel area is the gate terminal.The metal area of the gate, in conjunction with the insulating dielectric oxide layer and the semiconductor channel, forms a parallel plate capacitor. The insulating layer of SiO_2

Is the reason why this device is called the insulated gate field effect transistor. This layer results in an extremely high input resistance (10^{10} to 10^{15} ohms) for MOSFET.

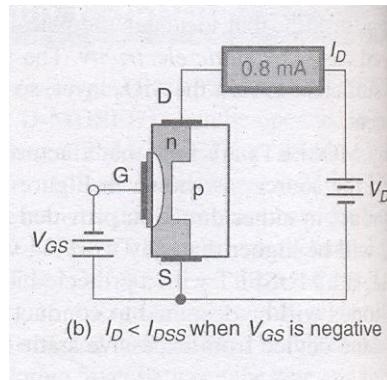
DEPLETION MOSFET

The basic structure of D –MOSFET is shown in the fig. An N-channel is diffused between source and drain with the device an appreciable drain current I_{DSS} flows for zero gate to source voltage, $V_{GS}=0$.



Depletion mode operation:-

- 1) The above fig shows the D-MOSFET operating conditions with gate and source terminals shorted together($V_{GS}=0V$)
- 2) At this stage $I_D = I_{DSS}$ where $V_{GS}=0V$, with this voltage V_{DS} , an appreciable drain current I_{DSS} flows.
- 3) If the gate to source voltage is made negative i.e. V_{GS} is negative .Positive charges are induced in the channel through the SiO_2 of the gate capacitor.
- 4) Since the current in a FET is due to majority carriers(electrons for an N-type material) , the induced positive charges make the channel less conductive and the drain current drops as V_{GS} is made more negative.
- 5) The re distribution of charge in the channel causes an effective depletion of majority carriers , which accounts for the designation depletion MOSFET.
- 6) That means biasing voltage V_{GS} depletes the channel of free carriers This effectively reduces the width of the channel , increasing its resistance.
- 7) Note that negative V_{GS} has the same effect on the MOSFET as it has on the JFET.



- 8) As shown in the fig above, the depletion layer generated by V_{GS} (represented by the white space between the insulating material and the channel) cuts into the channel, reducing its width. As a result , $I_D < I_{DSS}$.The actual value of I_D depends on the value of I_{DSS} , $V_{GS}(\text{off})$ and V_{GS} .

Enhancement mode operation of the D-MOSFET:-

- 1) This operating mode is a result of applying a positive gate to source voltage V_{GS} to the device.
- 2) When V_{GS} is positive the channel is effectively widened. This reduces the resistance of the channel allowing I_D to exceed the value of I_{DSS}
- 3) When V_{GS} is given positive the majority carriers in the p-type are holes. The holes in the p type substrate are repelled by the +ve gate voltage.

- 4) At the same time, the conduction band electrons (minority carriers) in the p type material are attracted towards the channel by the +gate voltage.
- 5) With the build up of electrons near the channel , the area to the right of the physical channel effectively becomes an N type material.
- 6) The extended n type channel now allows more current, $I_D > I_{DSS}$

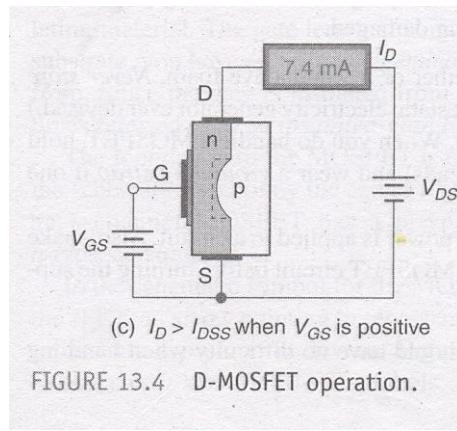
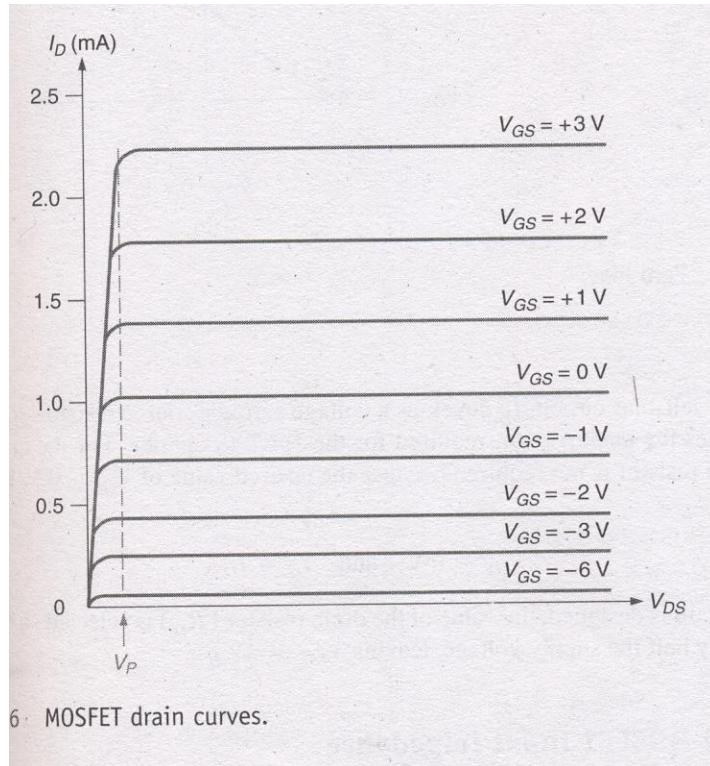


FIGURE 13.4 D-MOSFET operation.

Characteristics of Depletion MOSFET:-

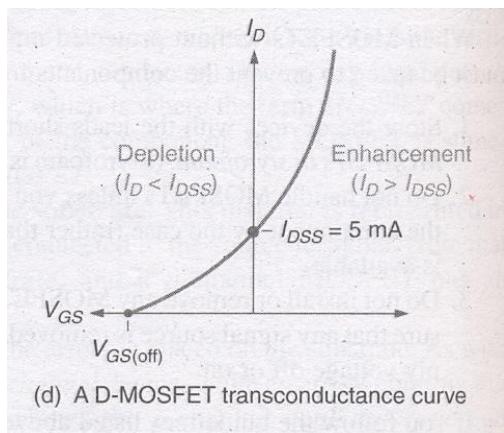
The fig. shows the drain characteristics for the N channel depletion type MOSFET

- 1) The curves are plotted for both V_{GS} positive and V_{GS} negative voltages
- 2) When $V_{GS}=0$ and negative the MOSFET operates in depletion mode when V_{GS} is positive ,the MOSFET operates in the enhancement mode.
- 3) The difference between JFET and D MOSFET is that JFET does not operate for positive values of V_{GS} .
- 4) When $V_{DS}=0$, there is no conduction takes place between source to drain, if $V_{GS}<0$ and $V_{DS}>0$ then I_D increases linearly.
- 5) But as $V_{GS},0$ induces positive charges holes in the channel, and controls the channel width. Thus the conduction between source to drain is maintained as constant, i.e. I_D is constant.
- 6) If $V_{GS}>0$ the gate induces more electrons in channel side, it is added with the free electrons generated by source. again the potential applied to gate determines the channel width and maintains constant current flow through it as shown in Fig



TRANSFER CHARACTERISTICS:-

The combination of 3 operating states i.e. $V_{GS}=0\text{V}$, $V_{GS}<0\text{V}$, $V_{GS}>0\text{V}$ is represented by the D MOSFET transconductance curve shown in Fig.

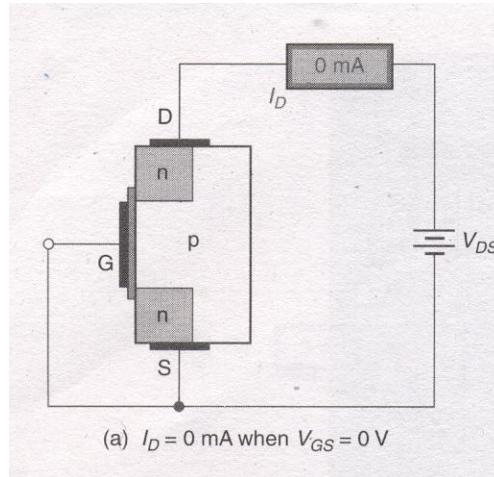


- 1) Here in this curve it may be noted that the region AB of the characteristics similar to that of JFET.
- 2) This curve extends for the positive values of V_{GS}

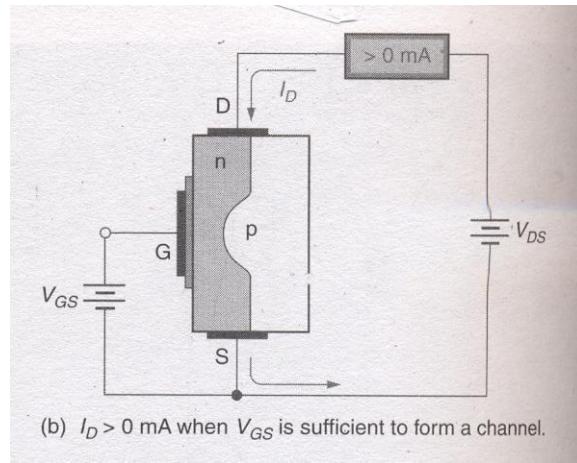
- 3) Note that $I_d = I_{dss}$ for $V_{gs} = 0V$ when V_{gs} is negative, $I_d < I_{dss}$ when $V_{gs} = V_{gs(off)}$, I_d is reduced to approximately $0mA$. Where V_{gs} is positive $I_d > I_{dss}$. So obviously I_{dss} is not the maximum possible value of I_d for a MOSFET.
- 4) The curves are similar to JFET so that the D MOSFET have the same transconductance equation.

E-MOSFETS

The E MOSFET is capable of operating only in the enhancement mode. The gate potential must be positive w.r.t to source.



- 1) when the value of $V_{gs}=0V$, there is no channel connecting the source and drain materials.
- 2) As a result, there can be no significant amount of drain current.
- 3) When $V_{gs}=0$, the V_{dd} supply tries to force free electrons from source to drain but the presence of p-region does not permit the electrons to pass through it. Thus there is no drain current at $V_{gs}=0$,
- 4) If V_{gs} is positive, it induces a negative charge in the p type substrate just adjacent to the SiO_2 layer.
- 5) As the holes are repelled by the positive gate voltage, the minority carrier electrons attracted toward this voltage. This forms an effective N type bridge between source and drain providing a path for drain current.
- 6) This +ve gate voltage forms a channel between the source and drain.
- 7) This produces a thin layer of N type channel in the P type substrate. This layer of free electrons is called N type inversion layer.



- 8) The minimum V_{GS} which produces this inversion layer is called threshold voltage and is designated by $V_{GS(\text{th})}$. This is the point at which the device turns on is called the threshold voltage $V_{GS(\text{th})}$
- 9) When the voltage V_{GS} is $< V_{GS(\text{th})}$ no current flows from drain to source.
- 10) However when the voltage $V_{GS} > V_{GS(\text{th})}$ the inversion layer connects the drain to source and we get significant values of current.

CHARACTERISTICS OF E MOSFET:-

1. DRAIN CHARACTERISTICS

The volt ampere drain characteristics of an N-channel enhancement mode MOSFET are given in the

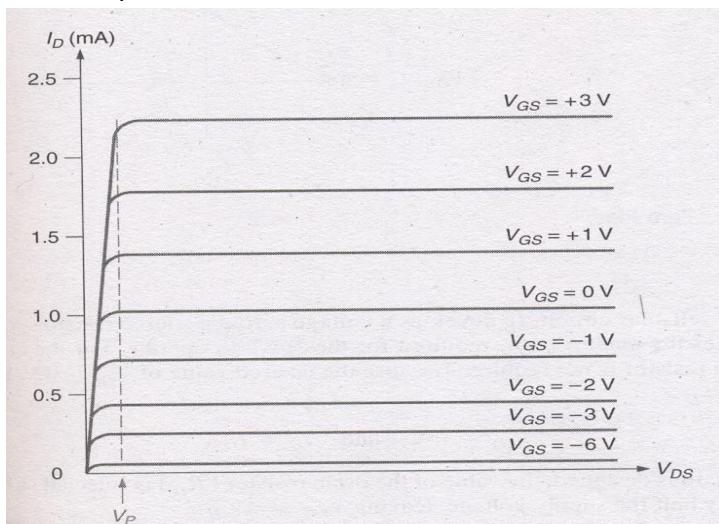


fig. 6. MOSFET drain curves.

2. TRANSFER CHARACTERISTICS:-

- 1) The current I_{DSS} at $V_{GS} \leq 0$ is very small being of the order of a few nano amps.
- 2) As V_{GS} is made +ve , the current I_D increases slowly at first, and then much more rapidly with an increase in V_{GS} .
- 3) The standard transconductance formula will not work for the E MOSFET.
- 4) To determine the value of I_D at a given value of V_{GS} we must use the following relation

$$I_D = K[V_{GS} - V_{GS(Th)}]^2$$

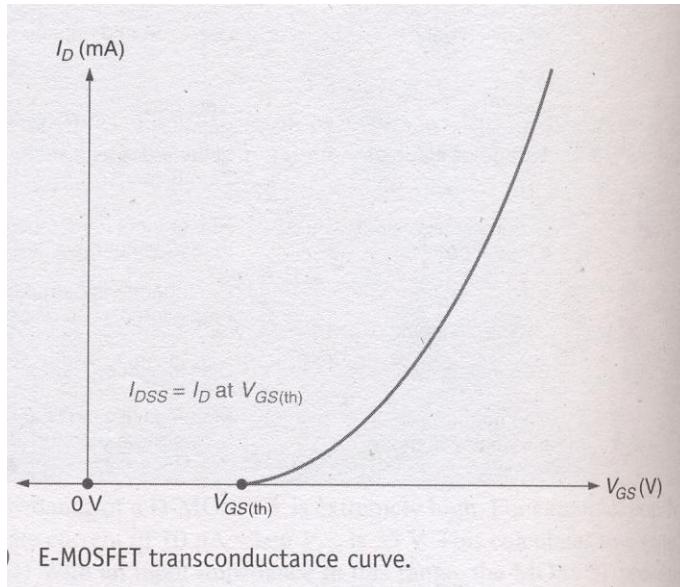
Where K is constant for the MOSFET . found as

$$K = \frac{I_D(\text{on})}{[V_{GS(\text{on})} - V_{GS(\text{Th})}]^2}$$

From the data specification sheets, the 2N7000 has the following ratings.

$I_D(\text{on})= 75\text{mA}(\text{minimum})$.

And $V_{GS(\text{th})}=0.8\text{(minimum)}$



APPLICATION OF MOSFET

One of the primary contributions to electronics made by MOSFETs can be found in the area of digital (computer electronics). The signals in digital circuits are made up of rapidly switching dc levels. This signal is called as a rectangular wave ,made up of two dc levels (or logic levels). These logic levels are 0V and +5V.

A group of circuits with similar circuitry and operating characteristics is referred to as a logic family. All the circuits in a given logic family respond to the same logic levels, have similar speed and power-handling capabilities , and can be directly connected together. One such logic family is complementary MOS (or CMOS) logic. This logic family is made up entirely of MOSFETs.

BIASING FET:-

For the proper functioning of a linear FET amplifier, it is necessary to maintain the operating point Q stable in the central portion of the pinch off region The Q point should be independent of device parameter variations and ambient temperature variations

This can be achieved by suitably selecting the gate to source voltage VGS and drain current ID which is referred to as biasing

JFET biasing circuits are very similar to BJT biasing circuitsThe main difference between JFET circuits and BJT circuits is the operation of the active components themselves

There are mainly two types of Biasing circuits

- 1) Self bias
- 2) Voltage divider bias.

SELF BIAS

Self bias is a JFET biasing circuit that uses a source resistor to help reverse bias the JFET gate. A self bias circuit is shown in the fig. Self bias is the most common type of JFET bias. This JFET must be operated such that gate source junction is always reverse biased. This condition requires a negative VGS for an N channel JFET and a positive VGS for P channel JFET. This can be achieved using the self bias arrangement as shown in Fig. The gate resistor RG doesn't affect the bias because it has essentially no voltage drop across it, and : the gate remains at 0V .RG is necessary only to isolate an ac signal from ground in amplifier applications. The voltage drop across resistor RS makes gate source junction reverse biased.

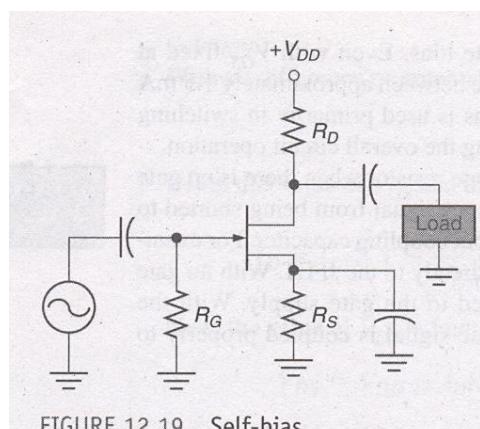


FIGURE 12.19 Self-bias.

For the dc analysis coupling capacitors are open circuits.

For the N channel FET in Fig (a)

IS produces a voltage drop across RS and makes the source positive w.r.t ground. In any JFET circuit all the source current passes through the device to the drain circuit .This is due to the fact that there is no significant gate current.

We can define source current as $I_S = I_D$

($V_G = 0$ because there is no gate current flowing in R_G So V_G across R_G is zero)

$$V_G = 0 \text{ then } V_S = I_S R_S = I_D R_S$$

$$V_{GS} = V_G - V_S = 0 - I_D R_S = -I_D R_S$$

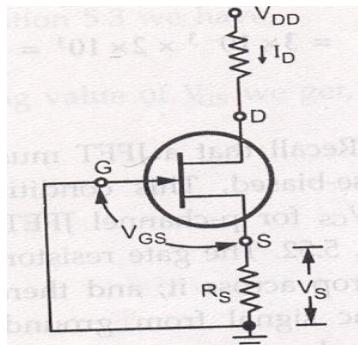
DC analysis of self Bias:-

In the following DC analysis, the N channel J FET shown in the fig. is used for illustration.

For DC analysis we can replace coupling capacitors by open circuits and we can also replace the resistor R_G by a short circuit equivalent.: $I_G = 0$.The relation between I_D and V_{GS} is given by

$$I_D = I_{DSS} \left[1 - \frac{V_{GS}}{V_P} \right]^2$$

V_{GS} for N channel JFET is $= -I_D R_S$



Substituting this value in the above equation

$$I_D = I_{DSS} \left[1 - \frac{(-I_D R_S)}{V_P} \right]^2$$

$$I_D = I_{DSS} \left[1 + \frac{(I_D R_S)}{V_P} \right]^2$$

For the N-channel FET in the above figure

I_s produces a voltage drop across R_s and makes the source positive w.r.t ground in any JFET circuit all the source current passes through the device to drain circuit this is due to the fact that there is no significant gate current. Therefore we can define source current as $I_s=I_d$ and $V_g=0$ then

$$V_s = I_s R_s = I_d R_s$$

$$V_{gs} = V_g - V_s = 0 - I_d R_s = -I_d R_s$$

Drawing the self bias line:-

Typical transfer characteristics for a self biased JFET are shown in the fig.

The maximum drain current is 5mA and the gate source cut off voltage is -3V. This means the gate voltage has to be between 0 and -3V.

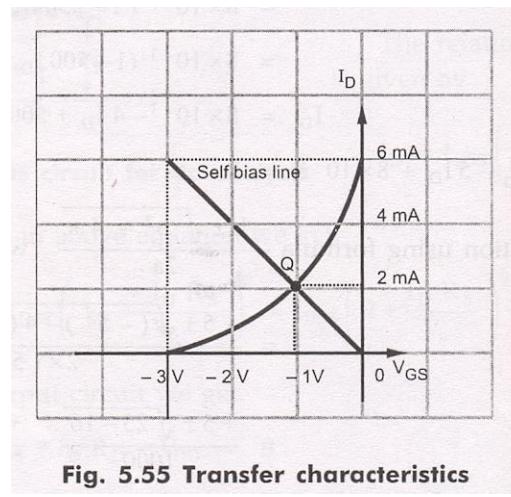


Fig. 5.55 Transfer characteristics

Now using the equation $V_{GS} = -I_d R_s$ and assuming R_s of any suitable value we can draw the self bias line.

Let us assume $R_s = 500\Omega$

With this R_s , we can plot two points corresponding to $I_d = 0$ and $I_d = I_{DSS}$

for $I_d = 0$

$$V_{GS} = -I_d R_s$$

$$V_{GS} = 0 \times (500\Omega) = 0V$$

So the first point is $(0, 0)$

$$(I_d, V_{GS})$$

For ID= IDSS=5mA

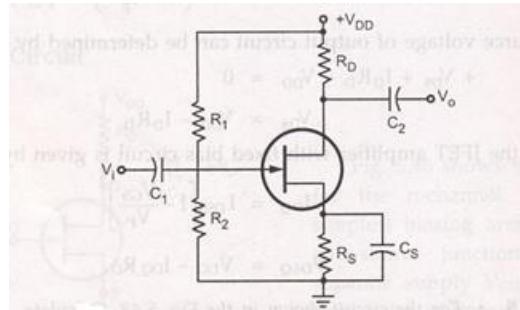
$$VGS = (-5mA) (500 \Omega) = -3V$$

So the 2nd Point will be (5mA,-3V)

By plotting these two points, we can draw the straight line through the points. This line will intersect the transconductance curve and it is known as self bias line. The intersection point gives the operating point of the self bias JFET for the circuit.

At Q point , the ID is slightly > than 2mA and VGS is slightly > -1V. The Q point for the self bias JFET depends on the value of Rs. If Rs is large, Q point far down on the transconductance curve ,ID is small, when Rs is small Q point is far up on the curve , ID is large.

VOLTAGE DIVIDER BIAS:-



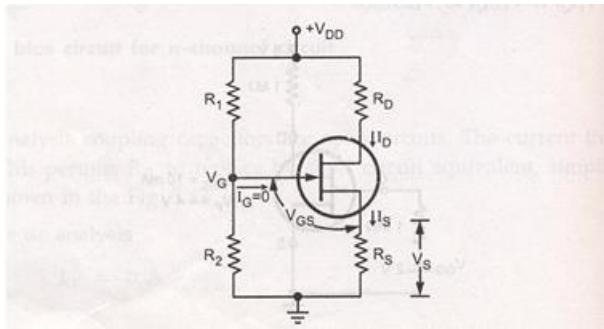
The fig. shows N channel JFET with voltage divider bias. The voltage at the source of JFET must be more positive than the voltage at the gate in order to keep the gate to source junction reverse biased. The source voltage is

$$VS = IDR_S$$

The gate voltage is set by resistors R1 and R2 as expressed by the following equation using the voltage divider formula.

$$Vg = \left(\frac{R_2}{(R_1+R_2)} \right) V_{dd}$$

For dc analysis



Applying KVL to the input circuit

$$VG - VGS - VS = 0$$

$$\therefore VGS = VG - Vs = VG - ISRS$$

$$VGS = VG - IDRS \quad \therefore IS = ID$$

Applying KVL to the input circuit we get

$$VDS + IDR_D + VS - VDD = 0$$

$$\therefore VDS = VDD - IDR_D - IDRS$$

$$VDS = VDD - ID (RD + RS)$$

The Q point of a JFET amplifier , using the voltage divider bias is

$$IDQ = IDSS [1 - VGS/VP]^2$$

$$VDSQ = VDD - ID (RD + RS)$$

COMPARISON OF MOSFET WITH JFET

- a. In enhancement and depletion types of MOSFET, the transverse electric field induced across an insulating layer deposited on the semiconductor material controls the conductivity of the channel.
- b. In the JFET the transverse electric field across the reverse biased PN junction controls the conductivity of the channel.

- c. The gate leakage current in a MOSFET is of the order of 10^{-12}A . Hence the input resistance of a MOSFET is very high in the order of 10^{10} to $10^{15}\Omega$. The gate leakage current of a JFET is of the order of 10^{-9}A , and its input resistance is of the order of $10^8\Omega$.
- d. The output characteristics of the JFET are flatter than those of the MOSFET, and hence the drain resistance of a JFET (0.1 to $1\text{M}\Omega$) is much higher than that of a MOSFET (1 to $50\text{k}\Omega$).
- e. JFETs are operated only in the depletion mode. The depletion type MOSFET may be operated in both depletion and enhancement mode.
- f. Comparing to JFET, MOSFETs are easier to fabricate.
- g. Special digital CMOS circuits are available which involve near zero power dissipation and very low voltage and current requirements. This makes them suitable for portable systems.

UNIT IV

Number System and Boolean Algebra

If base or radix of a number system is 'r', then the numbers present in that number system are ranging from zero to $r-1$. The total numbers present in that number system is ' r '. So, we will get various number systems, by choosing the values of radix as greater than or equal to two.

In this chapter, let us discuss about the **popular number systems** and how to represent a number in the respective number system. The following number systems are the most commonly used.

- Decimal Number system
- Binary Number system
- Octal Number system
- Hexadecimal Number system

Decimal Number System

The **base** or radix of Decimal number system is **10**. So, the numbers ranging from 0 to 9 are used in this number system. The part of the number that lies to the left of the **decimal point** is known as integer part. Similarly, the part of the number that lies to the right of the decimal point is known as fractional part.

In this number system, the successive positions to the left of the decimal point having weights of 10^0 , 10^1 , 10^2 , 10^3 and so on. Similarly, the successive positions to the right of the decimal point having weights of 10^{-1} , 10^{-2} , 10^{-3} and so on. That means, each position has specific weight, which is **power of base 10**

Example

Consider the **decimal number 1358.246**. Integer part of this number is 1358 and fractional part of this number is 0.246. The digits 8, 5, 3 and 1 have weights of 100 , 101 , 10^2 and 10^3 respectively. Similarly, the digits 2, 4 and 6 have weights of 10^{-1} , 10^{-2} and 10^{-3} respectively.

Mathematically, we can write it as

$$1358.246 = (1 \times 10^3) + (3 \times 10^2) + (5 \times 10^1) + (8 \times 10^0) + (2 \times 10^{-1}) + (4 \times 10^{-2}) + (6 \times 10^{-3})$$

After simplifying the right hand side terms, we will get the decimal number, which is on left hand side.

Binary Number System

All digital circuits and systems use this binary number system. The **base** or radix of this number system is **2**. So, the numbers 0 and 1 are used in this number system.

The part of the number, which lies to the left of the **binary point** is known as integer part. Similarly, the part of the number, which lies to the right of the binary point is known as fractional part.

In this number system, the successive positions to the left of the binary point having weights of $2^0, 2^1, 2^2, 2^3$ and so on. Similarly, the successive positions to the right of the binary point having weights of $2^{-1}, 2^{-2}, 2^{-3}$ and so on. That means, each position has specific weight, which is **power of base 2**.

Example

Consider the **binary number 1101.011**. Integer part of this number is 1101 and fractional part of this number is 0.011. The digits 1, 0, 1 and 1 of integer part have weights of $2^0, 2^1, 2^2, 2^3$ respectively. Similarly, the digits 0, 1 and 1 of fractional part have weights of $2^{-1}, 2^{-2}, 2^{-3}$ respectively.

Mathematically, we can write it as

$$1101.011 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$$

After simplifying the right hand side terms, we will get a decimal number, which is an equivalent of binary number on left hand side.

Octal Number System

The **base** or radix of octal number system is **8**. So, the numbers ranging from 0 to 7 are used in this number system. The part of the number that lies to the left of the **octal point** is known as integer part. Similarly, the part of the number that lies to the right of the octal point is known as fractional part.

In this number system, the successive positions to the left of the octal point having weights of $8^0, 8^1, 8^2, 8^3$ and so on. Similarly, the successive positions to the right of the octal point having weights of $8^{-1}, 8^{-2}, 8^{-3}$ and so on. That means, each position has specific weight, which is **power of base 8**.

Example

Consider the **octal number 1457.236**. Integer part of this number is 1457 and fractional part of this number is 0.236. The digits 7, 5, 4 and 1 have weights of $8^0, 8^1, 8^2$ and 8^3 respectively. Similarly, the digits 2, 3 and 6 have weights of $8^{-1}, 8^{-2}, 8^{-3}$ respectively.

Mathematically, we can write it as

$$1457.236 = (1 \times 8^3) + (4 \times 8^2) + (5 \times 8^1) + (7 \times 8^0) + (2 \times 8^{-1}) + (3 \times 8^{-2}) + (6 \times 8^{-3})$$

After simplifying the right hand side terms, we will get a decimal number, which is an equivalent of octal number on left hand side.

Hexadecimal Number System

The **base** or radix of Hexa-decimal number system is **16**. So, the numbers ranging from 0 to 9 and the letters from A to F are used in this number system. The decimal equivalent of Hexa-decimal digits from A to F are 10 to 15.

The part of the number, which lies to the left of the **hexadecimal point** is known as integer part. Similarly, the part of the number, which lies to the right of the Hexa-decimal point is known as fractional part.

In this number system, the successive positions to the left of the Hexa-decimal point having weights of 16^0 , 16^1 , 16^2 , 16^3 and so on. Similarly, the successive positions to the right of the Hexa-decimal point having weights of 16^{-1} , 16^{-2} , 16^{-3} and so on. That means, each position has specific weight, which is **power of base 16**.

Example

Consider the **Hexa-decimal number 1A05.2C4**. Integer part of this number is 1A05 and fractional part of this number is 0.2C4. The digits 5, 0, A and 1 have weights of 16^0 , 16^1 , 16^2 and 16^3 respectively. Similarly, the digits 2, C and 4 have weights of 16^{-1} , 16^{-2} and 16^{-3} respectively.

Mathematically, we can write it as

$$1A05.2C4 = (1 \times 16^3) + (10 \times 16^2) + (0 \times 16^1) + (5 \times 16^0) + (2 \times 16^{-1}) + (12 \times 16^{-2}) + (4 \times 16^{-3})$$

After simplifying the right hand side terms, we will get a decimal number, which is an equivalent of Hexa-decimal number on left hand side.

In previous chapter, we have seen the four prominent number systems. In this chapter, let us convert the numbers from one number system to the other in order to find the equivalent value.

Decimal Number to other Bases Conversion

If the decimal number contains both integer part and fractional part, then convert both the parts of decimal number into other base individually. Follow these steps for converting the decimal number into its equivalent number of any base 'r'.

- Do **division** of integer part of decimal number and **successive quotients** with base 'r' and note down the remainders till the quotient is zero. Consider the remainders in reverse order to get the integer part of equivalent number of base 'r'. That means, first and last remainders denote the least significant digit and most significant digit respectively.
- Do **multiplication** of fractional part of decimal number and **successive fractions** with base 'r' and note down the carry till the result is zero or the desired number of equivalent digits is obtained. Consider the normal sequence of carry in order to get the fractional part of equivalent number of base 'r'.

Decimal to Binary Conversion

The following two types of operations take place, while converting decimal number into its equivalent binary number.

- Division of integer part and successive quotients with base 2.
- Multiplication of fractional part and successive fractions with base 2.

Example

Consider the **decimal number 58.25**. Here, the integer part is 58 and fractional part is 0.25.

Step 1 – Division of 58 and successive quotients with base 2.

Operation	Quotient	Remainder
58/2	29	0 (LSB)
29/2	14	1
14/2	7	0
7/2	3	1
3/2	1	1
1/2	0	1(MSB)

$$\Rightarrow (58)_{10} = (111010)_2$$

Therefore, the **integer part** of equivalent binary number is **111010**.

Step 2 – Multiplication of 0.25 and successive fractions with base 2.

Operation	Result	Carry
0.25 x 2	0.5	0
0.5 x 2	1.0	1
-	0.0	-

$$\Rightarrow (.25)_{10} = (.01)_2$$

Therefore, the **fractional part** of equivalent binary number is **.01**

$$\Rightarrow (58.25)_{10} = (111010.01)_2$$

Therefore, the **binary equivalent** of decimal number 58.25 is 111010.01.

Decimal to Octal Conversion

The following two types of operations take place, while converting decimal number into its equivalent octal number.

- Division of integer part and successive quotients with base 8.
- Multiplication of fractional part and successive fractions with base 8.

Example

Consider the **decimal number 58.25**. Here, the integer part is 58 and fractional part is 0.25.

Step 1 – Division of 58 and successive quotients with base 8.

Operation	Quotient	Remainder
58/8	7	2
7/8	0	7

$$\Rightarrow (58)_{10} = (72)_8$$

Therefore, the **integer part** of equivalent octal number is **72**.

Step 2 – Multiplication of 0.25 and successive fractions with base 8.

Operation	Result	Carry
0.25 x 8	2.00	2
-	0.00	-

$$\Rightarrow (.25)_{10} = (.2)_8$$

Therefore, the **fractional part** of equivalent octal number is **.2**

$$\Rightarrow (58.25)_{10} = (72.2)_8$$

Therefore, the **octal equivalent** of decimal number 58.25 is 72.2.

Decimal to Hexa-Decimal Conversion

The following two types of operations take place, while converting decimal number into its equivalent hexa-decimal number.

- Division of integer part and successive quotients with base 16.

- Multiplication of fractional part and successive fractions with base 16.

Example

Consider the **decimal number 58.25**. Here, the integer part is 58 and decimal part is 0.25.

Step 1 – Division of 58 and successive quotients with base 16.

Operation	Quotient	Remainder
58/16	3	10=A
3/16	0	3

$$\Rightarrow (58)_{10} = (3A)_{16}$$

Therefore, the **integer part** of equivalent Hexa-decimal number is 3A.

Step 2 – Multiplication of 0.25 and successive fractions with base 16.

Operation	Result	Carry
0.25 x 16	4.00	4
-	0.00	-

$$\Rightarrow (.25)_{10} = (.4)_{16}$$

Therefore, the **fractional part** of equivalent Hexa-decimal number is .4.

$$\Rightarrow (58.25)_{10} = (3A.4)_{16}$$

Therefore, the **Hexa-decimal equivalent** of decimal number 58.25 is 3A.4.

Binary Number to other Bases Conversion

The process of converting a number from binary to decimal is different to the process of converting a binary number to other bases. Now, let us discuss about the conversion of a binary number to decimal, octal and Hexa-decimal number systems one by one.

Binary to Decimal Conversion

For converting a binary number into its equivalent decimal number, first multiply the bits of binary number with the respective positional weights and then add all those products.

Example

Consider the **binary number 1101.11**.

Mathematically, we can write it as

$$(1101.11)_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2})$$

$$\Rightarrow (1101.11)_2 = 8 + 4 + 0 + 1 + 0.5 + 0.25 = 13.75$$

$$\Rightarrow (1101.11)_2 = (13.75)_{10}$$

Therefore, the **decimal equivalent** of binary number 1101.11 is 13.75.

Binary to Octal Conversion

We know that the bases of binary and octal number systems are 2 and 8 respectively. Three bits of binary number is equivalent to one octal digit, since $2^3 = 8$.

Follow these two steps for converting a binary number into its equivalent octal number.

- Start from the binary point and make the groups of 3 bits on both sides of binary point. If one or two bits are less while making the group of 3 bits, then include required number of zeros on extreme sides.
- Write the octal digits corresponding to each group of 3 bits.

Example

Consider the **binary number 101110.01101**.

Step 1 – Make the groups of 3 bits on both sides of binary point.

101 110.011 01

Here, on right side of binary point, the last group is having only 2 bits. So, include one zero on extreme side in order to make it as group of 3 bits.

$\Rightarrow 101\ 110.011\ 010$

Step 2 – Write the octal digits corresponding to each group of 3 bits.

$\Rightarrow (101\ 110.011\ 010)_2 = (56.32)_8$

Therefore, the **octal equivalent** of binary number 101110.01101 is 56.32.

Binary to Hexa-Decimal Conversion

We know that the bases of binary and Hexa-decimal number systems are 2 and 16 respectively. Four bits of binary number is equivalent to one Hexa-decimal digit, since $2^4 = 16$.

Follow these two steps for converting a binary number into its equivalent Hexa-decimal number.

- Start from the binary point and make the groups of 4 bits on both sides of binary point. If some bits are less while making the group of 4 bits, then include required number of zeros on extreme sides.
- Write the Hexa-decimal digits corresponding to each group of 4 bits.

Example

Consider the **binary number 101110.01101**

Step 1 – Make the groups of 4 bits on both sides of binary point.

10 1110.0110 1

Here, the first group is having only 2 bits. So, include two zeros on extreme side in order to make it as group of 4 bits. Similarly, include three zeros on extreme side in order to make the last group also as group of 4 bits.

$\Rightarrow 0010\ 1110.0110\ 1000$

Step 2 – Write the Hexa-decimal digits corresponding to each group of 4 bits.

$\Rightarrow (0010\ 1110.0110\ 1000)_2 = (2E.68)_{16}$

Therefore, the **Hexa-decimal equivalent** of binary number 101110.01101 is (2E.68).

Octal Number to other Bases Conversion

The process of converting a number from octal to decimal is different to the process of converting an octal number to other bases. Now, let us discuss about the conversion of an octal number to decimal, binary and Hexa-decimal number systems one by one.

Octal to Decimal Conversion

For converting an octal number into its equivalent decimal number, first multiply the digits of octal number with the respective positional weights and then add all those products.

Example

Consider the **octal number 145.23**.

Mathematically, we can write it as

$$(145.23)_8 = (1 \times 8^2) + (4 \times 8^1) + (5 \times 8^0) + (2 \times 8^{-1}) + (3 \times 8^{-2})$$

$$\Rightarrow (145.23)_8 = 64 + 32 + 5 + 0.25 + 0.05 = 101.3$$

$$\Rightarrow (145.23)_8 = (101.3)_{10}$$

Therefore, the **decimal equivalent** of octal number 145.23 is 101.3.

Octal to Binary Conversion

The process of converting an octal number to an equivalent binary number is just opposite to that of binary to octal conversion. By representing each octal digit with 3 bits, we will get the equivalent binary number.

Example

Consider the **octal number 145.23**.

Represent each octal digit with 3 bits.

$$(145.23)_8 = (001\ 100\ 101.010\ 011)_2$$

The value doesn't change by removing the zeros, which are on the extreme side.

$$\Rightarrow (145.23)_8 = (1100101.010011)_2$$

Therefore, the **binary equivalent** of octal number 145.23 is 1100101.010011.

Octal to Hexa-Decimal Conversion

Follow these two steps for converting an octal number into its equivalent Hexa-decimal number.

- Convert octal number into its equivalent binary number.
- Convert the above binary number into its equivalent Hexa-decimal number.

Example

Consider the **octal number 145.23**

In previous example, we got the binary equivalent of octal number 145.23 as 1100101.010011.

By following the procedure of binary to Hexa-decimal conversion, we will get

$$(1100101.010011)_2 = (65.4C)_{16}$$

$$\Rightarrow (145.23)_8 = (65.4C)_{16}$$

Therefore, the **Hexa-decimal equivalent** of octal number 145.23 is 65.4C.

Hexa-Decimal Number to other Bases Conversion

The process of converting a number from Hexa-decimal to decimal is different to the process of converting Hexa-decimal number into other bases. Now, let us discuss about the conversion of Hexa-decimal number to decimal, binary and octal number systems one by one.

Hexa-Decimal to Decimal Conversion

For converting Hexa-decimal number into its equivalent decimal number, first multiply the digits of Hexa-decimal number with the respective positional weights and then add all those products.

Example

Consider the **Hexa-decimal number 1A5.2**

Mathematically, we can write it as

$$(1A5.2)_{16} = (1 \times 16^2) + (10 \times 16^1) + (5 \times 16^0) + (2 \times 16^{-1})$$

$$\Rightarrow (1A5.2)_{16} = 256 + 160 + 5 + 0.125 = 421.125$$

$$\Rightarrow (1A5.2)_{16} = (421.125)_{10}$$

Therefore, the **decimal equivalent** of Hexa-decimal number 1A5.2 is 421.125.

Hexa-Decimal to Binary Conversion

The process of converting Hexa-decimal number into its equivalent binary number is just opposite to that of binary to Hexa-decimal conversion. By representing each Hexa-decimal digit with 4 bits, we will get the equivalent binary number.

Example

Consider the **Hexa-decimal number 65.4C**

Represent each Hexa-decimal digit with 4 bits.

$$(65.4C)_6 = (0110\ 0101.0100\ 1100)_2$$

The value doesn't change by removing the zeros, which are at two extreme sides.

$$\Rightarrow (65.4C)_{16} = (1100101.010011)_2$$

Therefore, the **binary equivalent** of Hexa-decimal number 65.4C is 1100101.010011.

Hexa-Decimal to Octal Conversion

Follow these two steps for converting Hexa-decimal number into its equivalent octal number.

- Convert Hexa-decimal number into its equivalent binary number.
- Convert the above binary number into its equivalent octal number.

Example

Consider the **Hexa-decimal number 65.4C**

In previous example, we got the binary equivalent of Hexa-decimal number 65.4C as 1100101.010011.

By following the procedure of binary to octal conversion, we will get

$$(1100101.010011)_2 = (145.23)_8$$

$$\Rightarrow (65.4C)_{16} = (145.23)_8$$

Therefore, the **octal equivalent** of Hexa-decimal number 65.4C is 145.23.

We can make the binary numbers into the following two groups – **Unsigned numbers** and **Signed numbers**.

Unsigned Numbers

Unsigned numbers contain only magnitude of the number. They don't have any sign. That means all unsigned binary numbers are positive. As in decimal number system, the placing of positive sign in front of the number is optional for representing positive numbers. Therefore, all positive numbers including zero can be treated as unsigned numbers if positive sign is not assigned in front of the number.

Signed Numbers

Signed numbers contain both sign and magnitude of the number. Generally, the sign is placed in front of number. So, we have to consider the positive sign for positive numbers and negative sign for negative numbers. Therefore, all numbers can be treated as signed numbers if the corresponding sign is assigned in front of the number.

If sign bit is zero, which indicates the binary number is positive. Similarly, if sign bit is one, which indicates the binary number is negative.

Representation of Un-Signed Binary Numbers

The bits present in the un-signed binary number holds the **magnitude** of a number. That means, if the un-signed binary number contains ‘N’ bits, then all N bits represent the magnitude of the number, since it doesn’t have any sign bit.

Example

Consider the **decimal number 108**. The binary equivalent of this number is **1101100**. This is the representation of unsigned binary number.

$$(108)_{10} = (1101100)_2$$

It is having 7 bits. These 7 bits represent the magnitude of the number 108.

Representation of Signed Binary Numbers

The Most Significant Bit (MSB) of signed binary numbers is used to indicate the sign of the numbers. Hence, it is also called as **sign bit**. The positive sign is represented by placing ‘0’ in the sign bit. Similarly, the negative sign is represented by placing ‘1’ in the sign bit.

If the signed binary number contains ‘N’ bits, then (N-1) bits only represent the magnitude of the number since one bit (MSB) is reserved for representing sign of the number.

There are three **types of representations** for signed binary numbers

- Sign-Magnitude form
- 1’s complement form
- 2’s complement form

Representation of a positive number in all these 3 forms is same. But, only the representation of negative number will differ in each form.

Example

Consider the **positive decimal number +108**. The binary equivalent of magnitude of this number is 1101100. These 7 bits represent the magnitude of the number 108. Since it is positive number, consider the sign bit as zero, which is placed on left most side of magnitude.

$$(+108)_{10} = (01101100)_2$$

Therefore, the **signed binary representation** of positive decimal number +108 is **01101100**. So, the same representation is valid in sign-magnitude form, 1's complement form and 2's complement form for positive decimal number +108.

Sign-Magnitude form

In sign-magnitude form, the MSB is used for representing **sign** of the number and the remaining bits represent the **magnitude** of the number. So, just include sign bit at the left most side of unsigned binary number. This representation is similar to the signed decimal numbers representation.

Example

Consider the **negative decimal number -108**. The magnitude of this number is 108. We know the unsigned binary representation of 108 is 1101100. It is having 7 bits. All these bits represent the magnitude.

Since the given number is negative, consider the sign bit as one, which is placed on left most side of magnitude.

$$(-108)_{10} = (11101100)_2$$

Therefore, the sign-magnitude representation of -108 is **11101100**.

1's complement form

The 1's complement of a number is obtained by **complementing all the bits** of signed binary number. So, 1's complement of positive number gives a negative number. Similarly, 1's complement of negative number gives a positive number.

That means, if you perform two times 1's complement of a binary number including sign bit, then you will get the original signed binary number.

Example

Consider the **negative decimal number -108**. The magnitude of this number is 108. We know the signed binary representation of 108 is 01101100.

It is having 8 bits. The MSB of this number is zero, which indicates positive number. Complement of zero is one and vice-versa. So, replace zeros by ones and ones by zeros in order to get the negative number.

$$(-108)_{10} = (10010011)_2$$

Therefore, the **1's complement of (108)₁₀** is **(10010011)₂**.

2's complement form

The 2's complement of a binary number is obtained by **adding one to the 1's complement** of signed binary number. So, 2's complement of positive number gives a negative number. Similarly, 2's complement of negative number gives a positive number.

That means, if you perform two times 2's complement of a binary number including sign bit, then you will get the original signed binary number.

Example

Consider the **negative decimal number -108**.

We know the 1's complement of $(108)_{10}$ is $(10010011)_2$

$2's\ compliment\ of\ (108)_{10} = 1's\ compliment\ of\ (108)_{10} + 1.$

$$= 10010011 + 1$$

$$= 10010100$$

Therefore, the **2's complement of $(108)_{10}$** is $(10010100)_2$.

In this chapter, let us discuss about the basic arithmetic operations, which can be performed on any two signed binary numbers using 2's complement method. The **basic arithmetic operations** are addition and subtraction.

Addition of two Signed Binary Numbers

Consider the two signed binary numbers A & B, which are represented in 2's complement form. We can perform the **addition** of these two numbers, which is similar to the addition of two unsigned binary numbers. But, if the resultant sum contains carry out from sign bit, then discard (ignore) it in order to get the correct value.

If resultant sum is positive, you can find the magnitude of it directly. But, if the resultant sum is negative, then take 2's complement of it in order to get the magnitude.

Example 1

Let us perform the **addition** of two decimal numbers **+7 and +4** using 2's complement method.

The **2's complement** representations of +7 and +4 with 5 bits each are shown below.

$$(+7)_{10} = (00111)_2$$

$$(+4)_{10} = (00100)_2$$

The addition of these two numbers is

$$(+7)_{10} + (+4)_{10} = (00111)_2 + (00100)_2$$

$$\Rightarrow (+7)_{10} + (+4)_{10} = (01011)_2.$$

The resultant sum contains 5 bits. So, there is no carry out from sign bit. The sign bit '0' indicates that the resultant sum is **positive**. So, the magnitude of sum is 11 in decimal number system. Therefore, addition of two positive numbers will give another positive number.

Example 2

Let us perform the **addition** of two decimal numbers -7 and -4 using 2's complement method.

The **2's complement** representation of -7 and -4 with 5 bits each are shown below.

$$(-7)_{10} = (11001)_2$$

$$(-4)_{10} = (11100)_2$$

The addition of these two numbers is

$$(-7)_{10} + (-4)_{10} = (11001)_2 + (11100)_2$$

$$\Rightarrow (-7)_{10} + (-4)_{10} = (110101)_2.$$

The resultant sum contains 6 bits. In this case, carry is obtained from sign bit. So, we can remove it

Resultant sum after removing carry is $(-7)_{10} + (-4)_{10} = (10101)_2$.

The sign bit '1' indicates that the resultant sum is **negative**. So, by taking 2's complement of it we will get the magnitude of resultant sum as 11 in decimal number system. Therefore, addition of two negative numbers will give another negative number.

Subtraction of two Signed Binary Numbers

Consider the two signed binary numbers A & B, which are represented in 2's complement form. We know that 2's complement of positive number gives a negative number. So, whenever we have to subtract a number B from number A, then take 2's complement of B and add it to A. So, **mathematically** we can write it as

$$A - B = A + (2's \text{ complement of } B)$$

Similarly, if we have to subtract the number A from number B, then take 2's complement of A and add it to B. So, **mathematically** we can write it as

$$B - A = B + (2's \text{ complement of } A)$$

So, the subtraction of two signed binary numbers is similar to the addition of two signed binary numbers. But, we have to take 2's complement of the number, which is supposed to be subtracted. This is the **advantage** of 2's complement technique. Follow, the same rules of addition of two signed binary numbers.

Example 3

Let us perform the **subtraction** of two decimal numbers **+7 and +4** using 2's complement method.

The subtraction of these two numbers is

$$(+7)_{10} - (+4)_{10} = (+7)_{10} + (-4)_{10}.$$

The **2's complement** representation of +7 and -4 with 5 bits each are shown below.

$$(+7)_{10} = (00111)_2$$

$$(+4)_{10} = (11100)_2$$

$$\Rightarrow (+7)_{10} + (+4)_{10} = (00111)_2 + (11100)_2 = (00011)_2$$

Here, the carry obtained from sign bit. So, we can remove it. The resultant sum after removing carry is

$$(+7)_{10} + (+4)_{10} = \mathbf{(00011)}_2$$

The sign bit '0' indicates that the resultant sum is **positive**. So, the magnitude of it is 3 in decimal number system. Therefore, subtraction of two decimal numbers +7 and +4 is +3.

Example 4

Let us perform the **subtraction** of two decimal numbers **+4 and +7** using 2's complement method.

The subtraction of these two numbers is

$$(+4)_{10} - (+7)_{10} = (+4)_{10} + (-7)_{10}.$$

The **2's complement** representation of +4 and -7 with 5 bits each are shown below.

$$(+4)_{10} = (00100)_2$$

$$(-7)_{10} = (11001)_2$$

$$\Rightarrow (+4)_{10} + (-7)_{10} = (00100)_2 + (11001)_2 = (11101)_2$$

Here, carry is not obtained from sign bit. The sign bit '1' indicates that the resultant sum is **negative**. So, by taking 2's complement of it we will get the magnitude of resultant sum as 3 in decimal number system. Therefore, subtraction of two decimal numbers +4 and +7 is -3.

In the coding, when numbers or letters are represented by a specific group of symbols, it is said to be that number or letter is being encoded. The group of symbols is called as **code**. The digital data is represented, stored and transmitted as group of bits. This group of bits is also called as **binary code**.

Binary codes can be classified into two types.

- Weighted codes
- Unweighted codes

If the code has positional weights, then it is said to be **weighted code**. Otherwise, it is an unweighted code. Weighted codes can be further classified as positively weighted codes and negatively weighted codes.

Binary Codes for Decimal digits

The following table shows the various binary codes for decimal digits 0 to 9.

Decimal Digit	8421 Code	2421 Code	84-2-1 Code	Excess 3 Code
0	0000	0000	0000	0011
1	0001	0001	0111	0100
2	0010	0010	0110	0101
3	0011	0011	0101	0110
4	0100	0100	0100	0111
5	0101	1011	1011	1000
6	0110	1100	1010	1001
7	0111	1101	1001	1010
8	1000	1110	1000	1011
9	1001	1111	1111	1100

We have 10 digits in decimal number system. To represent these 10 digits in binary, we require minimum of 4 bits. But, with 4 bits there will be 16 unique combinations of zeros and ones. Since, we have only 10 decimal digits, the other 6 combinations of zeros and ones are not required.

8 4 2 1 code

- The weights of this code are 8, 4, 2 and 1.
- This code has all positive weights. So, it is a **positively weighted code**.
- This code is also called as **natural BCD** (Binary Coded Decimal) **code**.

Example

Let us find the BCD equivalent of the decimal number 786. This number has 3 decimal digits 7, 8 and 6. From the table, we can write the BCD (8421) codes of 7, 8 and 6 are 0111, 1000 and 0110 respectively.

$$\therefore (786)_{10} = (011110000110)_{BCD}$$

There are 12 bits in BCD representation, since each BCD code of decimal digit has 4 bits.

2 4 2 1 code

- The weights of this code are 2, 4, 2 and 1.
- This code has all positive weights. So, it is a **positively weighted code**.
- It is an **unnatural BCD** code. Sum of weights of unnatural BCD codes is equal to 9.
- It is a **self-complementing** code. Self-complementing codes provide the 9's complement of a decimal number, just by interchanging 1's and 0's in its equivalent 2421 representation.

Example

Let us find the 2421 equivalent of the decimal number 786. This number has 3 decimal digits 7, 8 and 6. From the table, we can write the 2421 codes of 7, 8 and 6 are 1101, 1110 and 1100 respectively.

Therefore, the 2421 equivalent of the decimal number 786 is **110111101100**.

8 4 -2 -1 code

- The weights of this code are 8, 4, -2 and -1.
- This code has negative weights along with positive weights. So, it is a **negatively weighted code**.
- It is an **unnatural BCD** code.
- It is a **self-complementing** code.

Example

Let us find the 8 4-2-1 equivalent of the decimal number 786. This number has 3 decimal digits 7, 8 and 6. From the table, we can write the 8 4 -2 -1 codes of 7, 8 and 6 are 1001, 1000 and 1010 respectively.

Therefore, the 8 4 -2 -1 equivalent of the decimal number 786 is **100110001010**.

Excess 3 code

- This code doesn't have any weights. So, it is an **un-weighted code**.
- We will get the Excess 3 code of a decimal number by adding three (0011) to the binary equivalent of that decimal number. Hence, it is called as Excess 3 code.
- It is a **self-complementing** code.

Example

Let us find the Excess 3 equivalent of the decimal number 786. This number has 3 decimal digits 7, 8 and 6. From the table, we can write the Excess 3 codes of 7, 8 and 6 are 1010, 1011 and 1001 respectively.

Therefore, the Excess 3 equivalent of the decimal number 786 is **101010111001**

Gray Code

The following table shows the 4-bit Gray codes corresponding to each 4-bit binary code.

Decimal Number	Binary Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101

7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

- This code doesn't have any weights. So, it is an **un-weighted code**.
- In the above table, the successive Gray codes are differed in one bit position only. Hence, this code is called as **unit distance** code.

Binary code to Gray Code Conversion

Follow these steps for converting a binary code into its equivalent Gray code.

- Consider the given binary code and place a zero to the left of MSB.
- Compare the successive two bits starting from zero. If the 2 bits are same, then the output is zero. Otherwise, output is one.
- Repeat the above step till the LSB of Gray code is obtained.

Example

From the table, we know that the Gray code corresponding to binary code 1000 is 1100. Now, let us verify it by using the above procedure.

Given, binary code is 1000.

Step 1 – By placing zero to the left of MSB, the binary code will be 01000.

Step 2 – By comparing successive two bits of new binary code, we will get the gray code as **1100**.

Boolean Algebra is an algebra, which deals with binary numbers & binary variables. Hence, it is also called as Binary Algebra or logical Algebra. A mathematician, named George Boole had developed this algebra in 1854. The variables used in this algebra are also called as Boolean variables.

The range of voltages corresponding to Logic ‘High’ is represented with ‘1’ and the range of voltages corresponding to logic ‘Low’ is represented with ‘0’.

Postulates and Basic Laws of Boolean Algebra

In this section, let us discuss about the Boolean postulates and basic laws that are used in Boolean algebra. These are useful in minimizing Boolean functions.

Boolean Postulates

Consider the binary numbers 0 and 1, Boolean variable (x) and its complement (x'). Either the Boolean variable or complement of it is known as **literal**. The four possible **logical OR** operations among these literals and binary numbers are shown below.

$$x + 0 = x$$

$$x + 1 = 1$$

$$x + x = x$$

$$x + x' = 1$$

Similarly, the four possible **logical AND** operations among those literals and binary numbers are shown below.

$$x \cdot 1 = x$$

$$x \cdot 0 = 0$$

$$x \cdot x = x$$

$$x \cdot x' = 0$$

These are the simple Boolean postulates. We can verify these postulates easily, by substituting the Boolean variable with ‘0’ or ‘1’.

Note– The complement of complement of any Boolean variable is equal to the variable itself. i.e., $(x')' = x$.

Basic Laws of Boolean Algebra

Following are the three basic laws of Boolean Algebra.

- Commutative law
- Associative law
- Distributive law

Commutative Law

If any logical operation of two Boolean variables give the same result irrespective of the order of those two variables, then that logical operation is said to be **Commutative**. The logical OR & logical AND operations of two Boolean variables x & y are shown below

$$x + y = y + x$$

$$x \cdot y = y \cdot x$$

The symbol ‘+’ indicates logical OR operation. Similarly, the symbol ‘.’ indicates logical AND operation and it is optional to represent. Commutative law obeys for logical OR & logical AND operations.

Associative Law

If a logical operation of any two Boolean variables is performed first and then the same operation is performed with the remaining variable gives the same result, then that logical operation is said to be **Associative**. The logical OR & logical AND operations of three Boolean variables x , y & z are shown below.

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

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

Associative law obeys for logical OR & logical AND operations.

Distributive Law

If any logical operation can be distributed to all the terms present in the Boolean function, then that logical operation is said to be **Distributive**. The distribution of logical OR & logical AND operations of three Boolean variables x , y & z are shown below.

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

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

Distributive law obeys for logical OR and logical AND operations.

These are the Basic laws of Boolean algebra. We can verify these laws easily, by substituting the Boolean variables with ‘0’ or ‘1’.

Theorems of Boolean Algebra

The following two theorems are used in Boolean algebra.

- Duality theorem
- DeMorgan's theorem

Duality Theorem

This theorem states that the **dual** of the Boolean function is obtained by interchanging the logical AND operator with logical OR operator and zeros with ones. For every Boolean function, there will be a corresponding Dual function.

Let us make the Boolean equations (relations) that we discussed in the section of Boolean postulates and basic laws into two groups. The following table shows these two groups.

Group1	Group2
$x + 0 = x$	$x \cdot 1 = x$
$x + 1 = 1$	$x \cdot 0 = 0$
$x + x = x$	$x \cdot x = x$
$x + x' = 1$	$x \cdot x' = 0$
$x + y = y + x$	$x \cdot y = y \cdot x$
$x + (y + z) = (x + y) + z$	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$
$x \cdot (y + z) = x \cdot y + x \cdot z$	$x + (y \cdot z) = (x + y) \cdot (x + z)$

In each row, there are two Boolean equations and they are dual to each other. We can verify all these Boolean equations of Group1 and Group2 by using duality theorem.

DeMorgan's Theorem

This theorem is useful in finding the **complement of Boolean function**. It states that the complement of logical OR of at least two Boolean variables is equal to the logical AND of each complemented variable.

DeMorgan's theorem with 2 Boolean variables x and y can be represented as

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

The dual of the above Boolean function is

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

Therefore, the complement of logical AND of two Boolean variables is equal to the logical OR of each complemented variable. Similarly, we can apply DeMorgan's theorem for more than 2 Boolean variables also.

Simplification of Boolean Functions

Till now, we discussed the postulates, basic laws and theorems of Boolean algebra. Now, let us simplify some Boolean functions.

Example 1

Let us **simplify** the Boolean function, $f = p'qr + pq'r + pqr' + pqr$

We can simplify this function in two methods.

Method 1

Given Boolean function, $f = p'qr + pq'r + pqr' + pqr$.

Step 1 – In first and second terms r is common and in third and fourth terms pq is common. So, take the common terms by using **Distributive law**.

$$\Rightarrow f = (p'q + pq')r + pq(r' + r)$$

Step 2 – The terms present in first parenthesis can be simplified to Ex-OR operation. The terms present in second parenthesis can be simplified to '1' using **Boolean postulate**

$$\Rightarrow f = (p \oplus q)r + pq(1)$$

Step 3 – The first term can't be simplified further. But, the second term can be simplified to pq using **Boolean postulate**.

$$\Rightarrow f = (p \oplus q)r + pq$$

Therefore, the simplified Boolean function is $f = (p \oplus q)r + pq$

Method 2

Given Boolean function, $f = p'qr + pq'r + pqr' + pqr$.

Step 1 – Use the **Boolean postulate**, $x + x = x$. That means, the Logical OR operation with any Boolean variable 'n' times will be equal to the same variable. So, we can write the last term pqr two more times.

$$\Rightarrow f = p'qr + pq'r + pqr' + pqr + pqr + pqr$$

Step 2 – Use **Distributive law** for 1st and 4th terms, 2nd and 5th terms, 3rd and 6th terms.

$$\Rightarrow f = qr(p' + p) + pr(q' + q) + pq(r' + r)$$

Step 3 – Use **Boolean postulate**, $x + x' = 1$ for simplifying the terms present in each parenthesis.

$$\Rightarrow f = qr(1) + pr(1) + pq(1)$$

Step 4 – Use **Boolean postulate**, $x \cdot 1 = x$ for simplifying the above three terms.

$$\Rightarrow f = qr + pr + pq$$

$$\Rightarrow f = pq + qr + pr$$

Therefore, the simplified Boolean function is $f = pq + qr + pr$.

So, we got two different Boolean functions after simplifying the given Boolean function in each method. Functionally, those two Boolean functions are same. So, based on the requirement, we can choose one of those two Boolean functions.

Example 2

Let us find the **complement** of the Boolean function, $f = p'q + pq'$.

The complement of Boolean function is $f' = (p'q + pq')'$.

Step 1 – Use DeMorgan's theorem, $(x + y)' = x' \cdot y'$.

$$\Rightarrow f' = (p'q)' \cdot (pq')'$$

Step 2 – Use DeMorgan's theorem, $(x \cdot y)' = x' + y'$

$$\Rightarrow f' = \{(p')' + q'\} \cdot \{p' + (q')'\}$$

Step3 – Use the Boolean postulate, $(x')' = x$.

$$\Rightarrow f' = \{p + q'\} \cdot \{p' + q\}$$

$$\Rightarrow f' = pp' + pq + p'q' + qq'$$

Step 4 – Use the Boolean postulate, $xx' = 0$.

$$\Rightarrow f = 0 + pq + p'q' + 0$$

$$\Rightarrow f = pq + p'q'$$

Therefore, the **complement** of Boolean function, $p'q + pq'$ is $pq + p'q'$.

We will get four Boolean product terms by combining two variables x and y with logical AND operation. These Boolean product terms are called as **min terms** or **standard product terms**. The min terms are $x'y'$, $x'y$, xy' and xy .

Similarly, we will get four Boolean sum terms by combining two variables x and y with logical OR operation. These Boolean sum terms are called as **Max terms** or **standard sum terms**. The Max terms are $x + y$, $x + y'$, $x' + y$ and $x' + y'$.

The following table shows the representation of min terms and MAX terms for 2 variables.

x	y	Min terms	Max terms
0	0	$m_0=x'y'$	$M_0=x + y$
0	1	$m_1=x'y$	$M_1=x + y'$
1	0	$m_2=xy'$	$M_2=x' + y$
1	1	$m_3=xy$	$M_3=x' + y'$

If the binary variable is '0', then it is represented as complement of variable in min term and as the variable itself in Max term. Similarly, if the binary variable is '1', then it is represented as complement of variable in Max term and as the variable itself in min term.

From the above table, we can easily notice that min terms and Max terms are complement of each other. If there are ' n ' Boolean variables, then there will be 2^n min terms and 2^n Max terms.

Canonical SoP and PoS forms

A truth table consists of a set of inputs and output(s). If there are ' n ' input variables, then there will be 2^n possible combinations with zeros and ones. So the value of each output variable depends on the combination of input variables. So, each output variable will have '1' for some combination of input variables and '0' for some other combination of input variables.

Therefore, we can express each output variable in following two ways.

- Canonical SoP form
- Canonical PoS form

Canonical SoP form

Canonical SoP form means Canonical Sum of Products form. In this form, each product term contains all literals. So, these product terms are nothing but the min terms. Hence, canonical SoP form is also called as **sum of min terms** form.

First, identify the min terms for which, the output variable is one and then do the logical OR of those min terms in order to get the Boolean expression (function) corresponding to that output variable. This Boolean function will be in the form of sum of min terms.

Follow the same procedure for other output variables also, if there is more than one output variable.

Example

Consider the following **truth table**.

Inputs		Output	
p	q	r	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Here, the output (f) is '1' for four combinations of inputs. The corresponding min terms are $p'qr$, $pq'r$, pqr' , pqr . By doing logical OR of these four min terms, we will get the Boolean function of output (f).

Therefore, the Boolean function of output is, $f = p'qr + pq'r + pqr' + pqr$. This is the **canonical SoP form** of output, f. We can also represent this function in following two notations.

$$f = m_3 + m_5 + m_6 + m_7 \quad f = m_3 + m_5 + m_6 + m_7$$

$$f = \sum m(3, 5, 6, 7) f = \sum m(3, 5, 6, 7)$$

In one equation, we represented the function as sum of respective min terms. In other equation, we used the symbol for summation of those min terms.

Canonical PoS form

Canonical PoS form means Canonical Product of Sums form. In this form, each sum term contains all literals. So, these sum terms are nothing but the Max terms. Hence, canonical PoS form is also called as **product of Max terms** form.

First, identify the Max terms for which, the output variable is zero and then do the logical AND of those Max terms in order to get the Boolean expression (function) corresponding to that output variable. This Boolean function will be in the form of product of Max terms.

Follow the same procedure for other output variables also, if there is more than one output variable.

Example

Consider the same truth table of previous example. Here, the output (f) is '0' for four combinations of inputs. The corresponding Max terms are $p + q + r$, $p + q + r'$, $p + q' + r$, $p' + q + r$. By doing logical AND of these four Max terms, we will get the Boolean function of output (f).

Therefore, the Boolean function of output is, $f = (p + q + r).(p + q + r').(p + q' + r).(p' + q + r)$. This is the **canonical PoS form** of output, f . We can also represent this function in following two notations.

$$f = M_0.M_1.M_2.M_4 f = M_0.M_1.M_2.M_4$$

$$f = \prod M(0, 1, 2, 4) f = \prod M(0, 1, 2, 4)$$

In one equation, we represented the function as product of respective Max terms. In other equation, we used the symbol for multiplication of those Max terms.

The Boolean function, $f = (p + q + r).(p + q + r').(p + q' + r).(p' + q + r)$ is the dual of the Boolean function, $f = p'qr + pq'r + pqr' + pqr$.

Therefore, both canonical SoP and canonical PoS forms are **Dual** to each other. Functionally, these two forms are same. Based on the requirement, we can use one of these two forms.

Standard SoP and PoS forms

We discussed two canonical forms of representing the Boolean output(s). Similarly, there are two standard forms of representing the Boolean output(s). These are the simplified version of canonical forms.

- Standard SoP form
- Standard PoS form

We will discuss about Logic gates in later chapters. The main **advantage** of standard forms is that the number of inputs applied to logic gates can be minimized. Sometimes, there will be reduction in the total number of logic gates required.

Standard SoP form

Standard SoP form means **Standard Sum of Products** form. In this form, each product term need not contain all literals. So, the product terms may or may not be the min terms. Therefore, the Standard SoP form is the simplified form of canonical SoP form.

We will get Standard SoP form of output variable in two steps.

- Get the canonical SoP form of output variable
- Simplify the above Boolean function, which is in canonical SoP form.

Follow the same procedure for other output variables also, if there is more than one output variable. Sometimes, it may not possible to simplify the canonical SoP form. In that case, both canonical and standard SoP forms are same.

Example

Convert the following Boolean function into Standard SoP form.

$$f = p'qr + pq'r + pqr' + pqr$$

The given Boolean function is in canonical SoP form. Now, we have to simplify this Boolean function in order to get standard SoP form.

Step 1 – Use the **Boolean postulate**, $x + x = x$. That means, the Logical OR operation with any Boolean variable ‘n’ times will be equal to the same variable. So, we can write the last term pqr two more times.

$$\Rightarrow f = p'qr + pq'r + pqr' + pqr + pqr + pqr$$

Step 2 – Use **Distributive law** for 1st and 4th terms, 2nd and 5th terms, 3rd and 6th terms.

$$\Rightarrow f = qr(p' + p) + pr(q' + q) + pq(r' + r)$$

Step 3 – Use **Boolean postulate**, $x + x' = 1$ for simplifying the terms present in each parenthesis.

$$\Rightarrow f = qr(1) + pr(1) + pq(1)$$

Step 4 – Use **Boolean postulate**, $x \cdot 1 = x$ for simplifying above three terms.

$$\Rightarrow f = qr + pr + pq$$

$$\Rightarrow f = pq + qr + pr$$

This is the simplified Boolean function. Therefore, the **standard SoP form** corresponding to given canonical SoP form is $f = pq + qr + pr$

Standard PoS form

Standard PoS form means **Standard Product of Sums** form. In this form, each sum term need not contain all literals. So, the sum terms may or may not be the Max terms. Therefore, the Standard PoS form is the simplified form of canonical PoS form.

We will get Standard PoS form of output variable in two steps.

- Get the canonical PoS form of output variable
- Simplify the above Boolean function, which is in canonical PoS form.

Follow the same procedure for other output variables also, if there is more than one output variable. Sometimes, it may not possible to simplify the canonical PoS form. In that case, both canonical and standard PoS forms are same.

Example

Convert the following Boolean function into Standard PoS form.

$$f = (p + q + r).(p + q + r').(p + q' + r).(p' + q + r)$$

The given Boolean function is in canonical PoS form. Now, we have to simplify this Boolean function in order to get standard PoS form.

Step 1 – Use the **Boolean postulate**, $x \cdot x = x$. That means, the Logical AND operation with any Boolean variable ‘n’ times will be equal to the same variable. So, we can write the first term $p+q+r$ two more times.

$$\Rightarrow f = (p + q + r).(p + q + r).(p + q + r).(p + q + r').(p + q' + r).(p' + q + r)$$

Step 2 – Use **Distributive law**, $x + (y.z) = (x + y).(x + z)$ for 1st and 4th parenthesis, 2nd and 5th parenthesis, 3rd and 6th parenthesis.

$$\Rightarrow f = (p + q + rr').(p + r + qq').(q + r + pp')$$

Step 3 – Use **Boolean postulate**, $x \cdot x' = 0$ for simplifying the terms present in each parenthesis.

$$\Rightarrow f = (p + q + 0).(p + r + 0).(q + r + 0)$$

Step 4 – Use **Boolean postulate**, $x + 0 = x$ for simplifying the terms present in each parenthesis

$$\Rightarrow f = (p + q).(p + r).(q + r)$$

$$\Rightarrow f = (p + q).(q + r).(p + r)$$

This is the simplified Boolean function. Therefore, the **standard PoS form** corresponding to given canonical PoS form is $f = (p + q).(q + r).(p + r)$. This is the **dual** of the Boolean function, $f = pq + qr + pr$. Therefore, both Standard SoP and Standard PoS forms are Dual to each other.

Digital electronic circuits operate with voltages of **two logic levels** namely Logic Low and Logic High. The range of voltages corresponding to Logic Low is represented with '0'. Similarly, the range of voltages corresponding to Logic High is represented with '1'.

The basic digital electronic circuit that has one or more inputs and single output is known as **Logic gate**. Hence, the Logic gates are the building blocks of any digital system. We can classify these Logic gates into the following three categories.

- Basic gates
- Universal gates
- Special gates

Now, let us discuss about the Logic gates come under each category one by one.

Basic Gates

In earlier chapters, we learnt that the Boolean functions can be represented either in sum of products form or in product of sums form based on the requirement. So, we can implement these Boolean functions by using basic gates. The basic gates are AND, OR & NOT gates.

AND gate

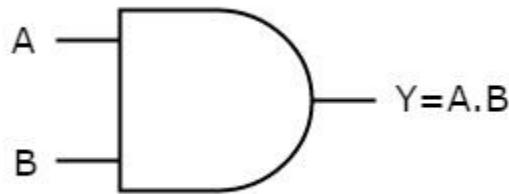
An AND gate is a digital circuit that has two or more inputs and produces an output, which is the **logical AND** of all those inputs. It is optional to represent the **Logical AND** with the symbol '.'.

The following table shows the **truth table** of 2-input AND gate.

A	B	$Y = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

Here A, B are the inputs and Y is the output of two input AND gate. If both inputs are '1', then only the output, Y is '1'. For remaining combinations of inputs, the output, Y is '0'.

The following figure shows the **symbol** of an AND gate, which is having two inputs A, B and one output, Y.



This AND gate produces an output (Y), which is the **logical AND** of two inputs A, B. Similarly, if there are 'n' inputs, then the AND gate produces an output, which is the logical AND of all those inputs. That means, the output of AND gate will be '1', when all the inputs are '1'.

OR gate

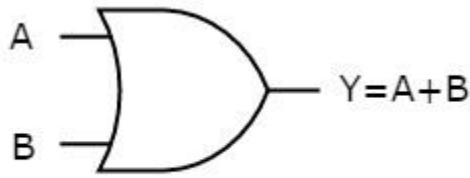
An OR gate is a digital circuit that has two or more inputs and produces an output, which is the logical OR of all those inputs. This **logical OR** is represented with the symbol '+'.

The following table shows the **truth table** of 2-input OR gate.

A	B	$Y = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Here A, B are the inputs and Y is the output of two input OR gate. If both inputs are '0', then only the output, Y is '0'. For remaining combinations of inputs, the output, Y is '1'.

The following figure shows the **symbol** of an OR gate, which is having two inputs A, B and one output, Y.



This OR gate produces an output (Y), which is the **logical OR** of two inputs A, B. Similarly, if there are 'n' inputs, then the OR gate produces an output, which is the logical OR of all those inputs. That means, the output of an OR gate will be '1', when at least one of those inputs is '1'.

NOT gate

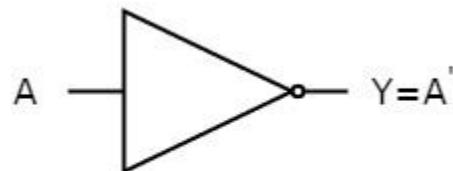
A NOT gate is a digital circuit that has single input and single output. The output of NOT gate is the **logical inversion** of input. Hence, the NOT gate is also called as inverter.

The following table shows the **truth table** of NOT gate.

A	$Y = A'$
0	1
1	0

Here A and Y are the input and output of NOT gate respectively. If the input, A is '0', then the output, Y is '1'. Similarly, if the input, A is '1', then the output, Y is '0'.

The following figure shows the **symbol** of NOT gate, which is having one input, A and one output, Y.



This NOT gate produces an output (Y), which is the **complement** of input, A.

Universal gates

NAND & NOR gates are called as **universal gates**. Because we can implement any Boolean function, which is in sum of products form by using NAND gates alone. Similarly, we can implement any Boolean function, which is in product of sums form by using NOR gates alone.

NAND gate

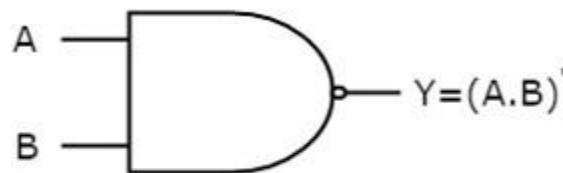
NAND gate is a digital circuit that has two or more inputs and produces an output, which is the **inversion of logical AND** of all those inputs.

The following table shows the **truth table** of 2-input NAND gate.

A	B	$Y = (A \cdot B)'$
0	0	1
0	1	1
1	0	1
1	1	0

Here A, B are the inputs and Y is the output of two input NAND gate. When both inputs are '1', the output, Y is '0'. If at least one of the input is zero, then the output, Y is '1'. This is just opposite to that of two input AND gate operation.

The following image shows the **symbol** of NAND gate, which is having two inputs A, B and one output, Y.



NAND gate operation is same as that of AND gate followed by an inverter. That's why the NAND gate symbol is represented like that.

NOR gate

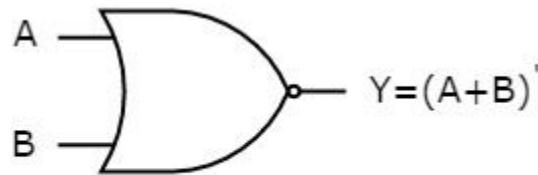
NOR gate is a digital circuit that has two or more inputs and produces an output, which is the **inversion of logical OR** of all those inputs.

The following table shows the **truth table** of 2-input NOR gate

A	B	$Y = (A+B)'$
0	0	1
0	1	0
1	0	0
1	1	0

Here A, B are the inputs and Y is the output. If both inputs are '0', then the output, Y is '1'. If at least one of the input is '1', then the output, Y is '0'. This is just opposite to that of two input OR gate operation.

The following figure shows the **symbol** of NOR gate, which is having two inputs A, B and one output, Y.



NOR gate operation is same as that of OR gate followed by an inverter. That's why the NOR gate symbol is represented like that.

Special Gates

Ex-OR & Ex-NOR gates are called as special gates. Because, these two gates are special cases of OR & NOR gates.

Ex-OR gate

The full form of Ex-OR gate is **Exclusive-OR** gate. Its function is same as that of OR gate except for some cases, when the inputs having even number of ones.

The following table shows the **truth table** of 2-input Ex-OR gate.

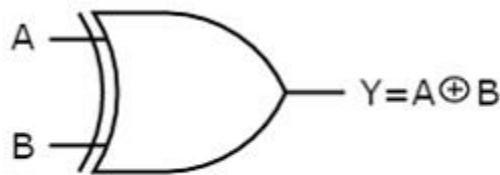
A	B	$Y = A \oplus B$
0	0	0

0	1	1
1	0	1
1	1	0

Here A, B are the inputs and Y is the output of two input Ex-OR gate. The truth table of Ex-OR gate is same as that of OR gate for first three rows. The only modification is in the fourth row. That means, the output (Y) is zero instead of one, when both the inputs are one, since the inputs having even number of ones.

Therefore, the output of Ex-OR gate is '1', when only one of the two inputs is '1'. And it is zero, when both inputs are same.

Below figure shows the **symbol** of Ex-OR gate, which is having two inputs A, B and one output, Y.



Ex-OR gate operation is similar to that of OR gate, except for few combination(s) of inputs. That's why the Ex-OR gate symbol is represented like that. The output of Ex-OR gate is '1', when odd number of ones present at the inputs. Hence, the output of Ex-OR gate is also called as an **odd function**.

Ex-NOR gate

The full form of Ex-NOR gate is **Exclusive-NOR** gate. Its function is same as that of NOR gate except for some cases, when the inputs having even number of ones.

The following table shows the **truth table** of 2-input Ex-NOR gate.

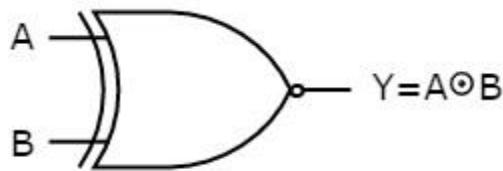
A	B	$Y = A \odot B$
0	0	1
0	1	0
1	0	0

1	1	1
---	---	---

Here A, B are the inputs and Y is the output. The truth table of Ex-NOR gate is same as that of NOR gate for first three rows. The only modification is in the fourth row. That means, the output is one instead of zero, when both the inputs are one.

Therefore, the output of Ex-NOR gate is ‘1’, when both inputs are same. And it is zero, when both the inputs are different.

The following figure shows the **symbol** of Ex-NOR gate, which is having two inputs A, B and one output, Y.



Ex-NOR gate operation is similar to that of NOR gate, except for few combination(s) of inputs. That's why the Ex-NOR gate symbol is represented like that. The output of Ex-NOR gate is ‘1’, when even number of ones present at the inputs. Hence, the output of Ex-NOR gate is also called as an **even function**.

From the above truth tables of Ex-OR & Ex-NOR logic gates, we can easily notice that the Ex-NOR operation is just the logical inversion of Ex-OR operation.

The maximum number of levels that are present between inputs and output is two in **two level logic**. That means, irrespective of total number of logic gates, the maximum number of Logic gates that are present (cascaded) between any input and output is two in two level logic. Here, the outputs of first level Logic gates are connected as inputs of second level Logic gate(s).

Consider the four Logic gates AND, OR, NAND & NOR. Since, there are 4 Logic gates, we will get 16 possible ways of realizing two level logic. Those are AND-AND, AND-OR, AND-NAND, AND-NOR, OR-AND, OR-OR, OR-NAND, OR-NOR, NAND-AND, NAND-OR, NAND-NAND, NAND-NOR, NOR-AND, NOR-OR, NOR-NAND, NOR-NOR.

These two level logic realizations can be classified into the following two categories.

- Degenerative form
- Non-degenerative form

Degenerative Form

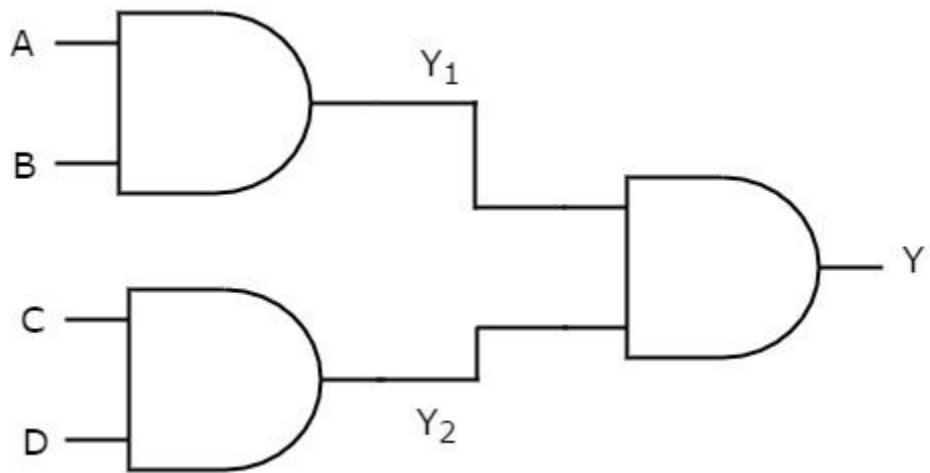
If the output of two level logic realization can be obtained by using single Logic gate, then it is called as **degenerative form**. Obviously, the number of inputs of single Logic gate increases. Due to this, the fan-in of Logic gate increases. This is an advantage of degenerative form.

Only **6 combinations** of two level logic realizations out of 16 combinations come under degenerative form. Those are AND-AND, AND-NAND, OR-OR, OR-NOR, NAND-NOR, NOR-NAND.

In this section, let us discuss some realizations. Assume, A, B, C & D are the inputs and Y is the output in each logic realization.

AND-AND Logic

In this logic realization, AND gates are present in both levels. Below figure shows an example for **AND-AND logic realization**.



We will get the outputs of first level logic gates as $Y_1 = AB$ and $Y_2 = CD$

These outputs, $Y_1 Y_1$ and $Y_2 Y_2$ are applied as inputs of AND gate that is present in second level. So, the output of this AND gate is

$$Y = Y_1 Y_2 = Y_1 Y_2$$

Substitute $Y_1 Y_1$ and $Y_2 Y_2$ values in the above equation.

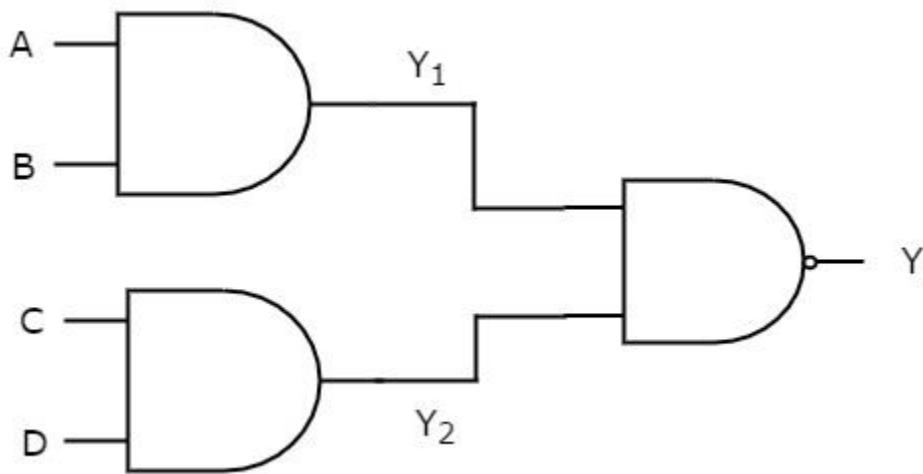
$$Y = (AB)(CD) = (AB)(CD)$$

$$\Rightarrow Y = ABCD \Rightarrow Y = ABCD$$

Therefore, the output of this AND-AND logic realization is **ABCD**. This Boolean function can be implemented by using a 4 input AND gate. Hence, it is **degenerative form**.

AND-NAND Logic

In this logic realization, AND gates are present in first level and NAND gate(s) are present in second level. The following figure shows an example for **AND-NAND logic** realization.



Previously, we got the outputs of first level logic gates as $Y_1 = AB$ and $Y_2 = CD$

These outputs, $Y_1 Y_1$ and $Y_2 Y_2$ are applied as inputs of NAND gate that is present in second level. So, the output of this NAND gate is

$$Y = (Y_1 Y_2)' = (AB)(CD)'$$

Substitute $Y_1 Y_1$ and $Y_2 Y_2$ values in the above equation.

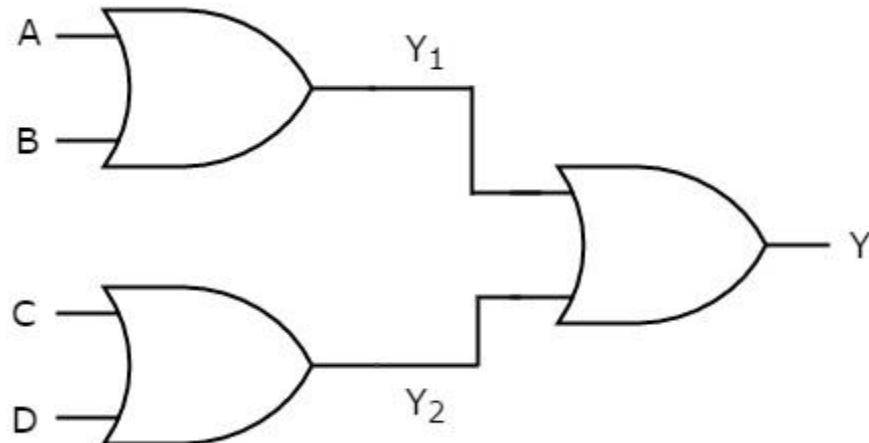
$$Y = ((AB)(CD))' = ((AB)(CD))'$$

$$\Rightarrow Y = (ABCD)' = (ABCD)$$

Therefore, the output of this AND-NAND logic realization is $(ABCD)'(ABCD)$. This Boolean function can be implemented by using a 4 input NAND gate. Hence, it is **degenerative form**.

OR-OR Logic

In this logic realization, OR gates are present in both levels. The following figure shows an example for **OR-OR logic** realization.



We will get the outputs of first level logic gates as $Y_1 = A + B$, $Y_1 = A + B$ and $Y_2 = C + D$, $Y_2 = C + D$.

These outputs, Y_1Y_1 and Y_2Y_2 are applied as inputs of OR gate that is present in second level. So, the output of this OR gate is

$$Y=Y_1+Y_2 Y=Y_1+Y_2$$

Substitute Y_1Y_1 and Y_2Y_2 values in the above equation.

$$Y=(A+B)+(C+D) Y=(A+B)+(C+D)$$

$$\Rightarrow Y=A+B+C+D \Rightarrow Y=A+B+C+D$$

Therefore, the output of this OR-OR logic realization is **$A+B+C+D$** . This Boolean function can be implemented by using a 4 input OR gate. Hence, it is **degenerative form**.

Similarly, you can verify whether the remaining realizations belong to this category or not.

Non-degenerative Form

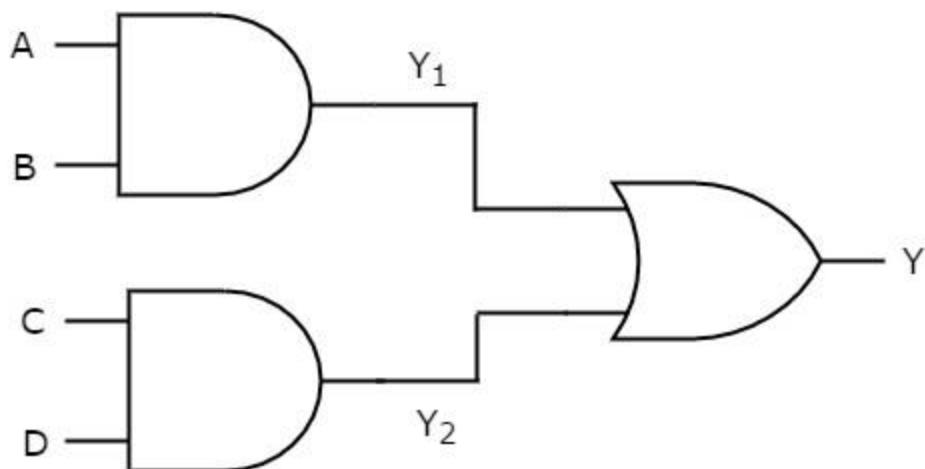
If the output of two level logic realization can't be obtained by using single logic gate, then it is called as **non-degenerative form**.

The remaining **10 combinations** of two level logic realizations come under nondegenerative form. Those are AND-OR, AND-NOR, OR-AND, OR-NAND, NAND-AND, NANDOR, NAND-NAND, NOR-AND, NOR-OR, NOR-NOR.

Now, let us discuss some realizations. Assume, A, B, C & D are the inputs and Y is the output in each logic realization.

AND-OR Logic

In this logic realization, AND gates are present in first level and OR gate(s) are present in second level. Below figure shows an example for **AND-OR logic** realization.



Previously, we got the outputs of first level logic gates as $Y_1=AB$ and $Y_2=CD$.

These outputs, Y_1 and Y_2 are applied as inputs of OR gate that is present in second level. So, the output of this OR gate is

$$Y = Y_1 + Y_2 \quad Y = Y_1 + Y_2$$

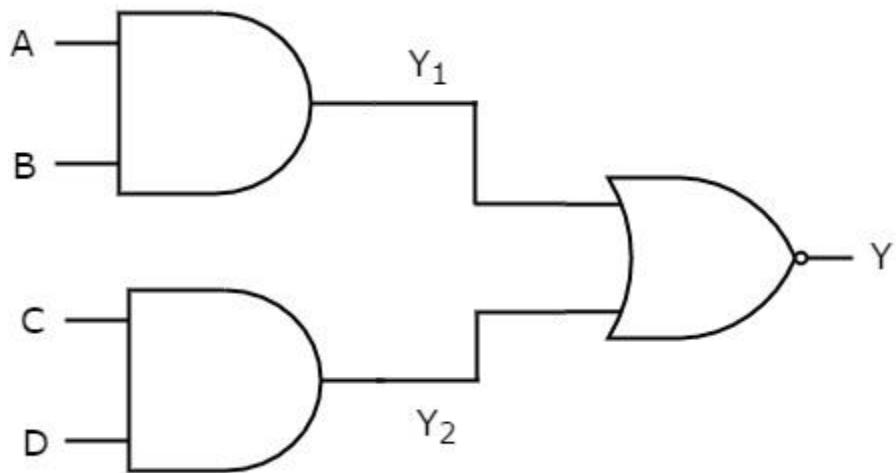
Substitute Y_1Y_1 and Y_2Y_2 values in the above equation

$$Y = AB + CD \quad Y = AB + CD$$

Therefore, the output of this AND-OR logic realization is $AB + CD$. This Boolean function is in **Sum of Products** form. Since, we can't implement it by using single logic gate, this AND-OR logic realization is a **non-degenerative form**.

AND-NOR Logic

In this logic realization, AND gates are present in first level and NOR gate(s) are present in second level. The following figure shows an example for **AND-NOR logic** realization.



We know the outputs of first level logic gates as $Y_1 = AB$ and $Y_2 = CD$

These outputs, Y_1 and Y_2 are applied as inputs of NOR gate that is present in second level. So, the output of this NOR gate is

$$Y = (Y_1 + Y_2)' \quad Y = (AB + CD)'$$

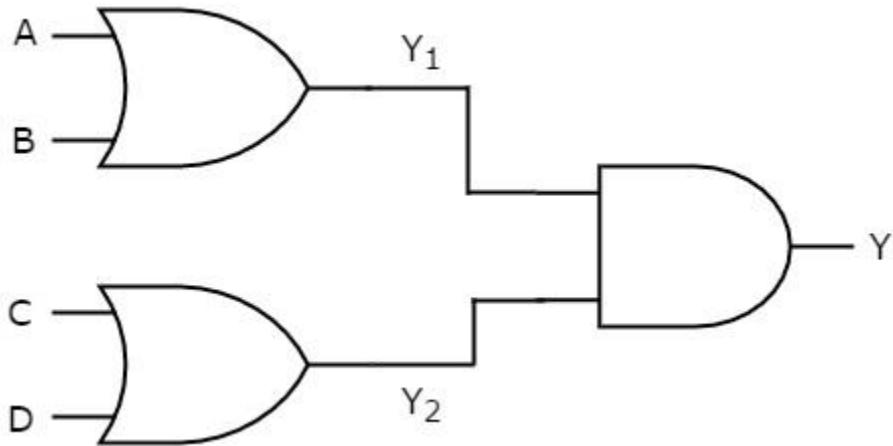
Substitute Y_1Y_1 and Y_2Y_2 values in the above equation.

$$Y = (AB + CD)' \quad Y = (AB + CD)'$$

Therefore, the output of this AND-NOR logic realization is $(AB + CD)'(AB + CD)'$. This Boolean function is in **AND-OR-Invert** form. Since, we can't implement it by using single logic gate, this AND-NOR logic realization is a **non-degenerative form**

OR-AND Logic

In this logic realization, OR gates are present in first level & AND gate(s) are present in second level. The following figure shows an example for **OR-AND logic** realization.



Previously, we got the outputs of first level logic gates as $Y_1 = A + B$ and $Y_2 = C + D$.

These outputs, $Y_1 Y_1$ and $Y_2 Y_2$ are applied as inputs of AND gate that is present in second level. So, the output of this AND gate is

$$Y = Y_1 Y_1 = Y_1 Y_2$$

Substitute $Y_1 Y_1$ and $Y_2 Y_2$ values in the above equation.

$$Y = (A + B)(C + D) = (A + B)(C + D)$$

Therefore, the output of this OR-AND logic realization is $(A + B)(C + D)$. This Boolean function is in **Product of Sums** form. Since, we can't implement it by using single logic gate, this OR-AND logic realization is a **non-degenerative form**.

Similarly, you can verify whether the remaining realizations belong to this category or not.

UNIT-V

Minimization Techniques & Combinational Circuits

In previous chapters, we have simplified the Boolean functions using Boolean postulates and theorems. It is a time consuming process and we have to re-write the simplified expressions after each step.

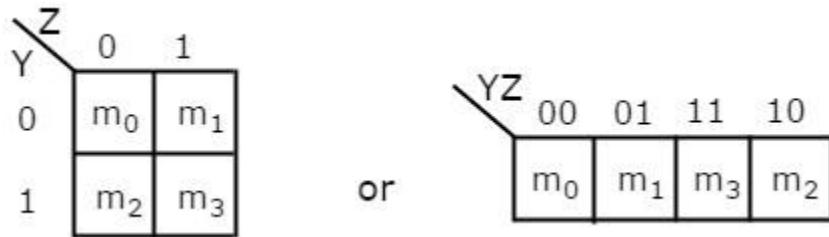
To overcome this difficulty, Karnaugh introduced a method for simplification of Boolean functions in an easy way. This method is known as Karnaugh map method or K-map method. It is a graphical method, which consists of 2^n cells for 'n' variables. The adjacent cells are differed only in single bit position.

K-Maps for 2 to 5 Variables

K-Map method is most suitable for minimizing Boolean functions of 2 variables to 5 variables. Now, let us discuss about the K-Maps for 2 to 5 variables one by one.

2 Variable K-Map

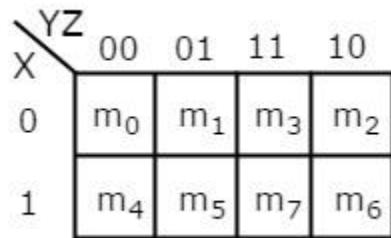
The number of cells in 2 variable K-map is four, since the number of variables is two. The following figure shows **2 variable K-Map**.



- There is only one possibility of grouping 4 adjacent min terms.
- The possible combinations of grouping 2 adjacent min terms are {(m₀, m₁), (m₂, m₃), (m₀, m₂) and (m₁, m₃)}.

3 Variable K-Map

The number of cells in 3 variable K-map is eight, since the number of variables is three. The following figure shows **3 variable K-Map**.



- There is only one possibility of grouping 8 adjacent min terms.
- The possible combinations of grouping 4 adjacent min terms are $\{(m_0, m_1, m_3, m_2), (m_4, m_5, m_7, m_6), (m_0, m_1, m_4, m_5), (m_1, m_3, m_5, m_7), (m_3, m_2, m_7, m_6) \text{ and } (m_2, m_0, m_6, m_4)\}$.
- The possible combinations of grouping 2 adjacent min terms are $\{(m_0, m_1), (m_1, m_3), (m_3, m_2), (m_2, m_0), (m_4, m_5), (m_5, m_7), (m_7, m_6), (m_6, m_4), (m_0, m_4), (m_1, m_5), (m_3, m_7) \text{ and } (m_2, m_6)\}$.
- If $x=0$, then 3 variable K-map becomes 2 variable K-map.

4 Variable K-Map

The number of cells in 4 variable K-map is sixteen, since the number of variables is four. The following figure shows **4 variable K-Map**.

WX		YZ			
		00	01	11	10
WZ	00	m_0	m_1	m_3	m_2
	01	m_4	m_5	m_7	m_6
	11	m_{12}	m_{13}	m_{15}	m_{14}
	10	m_8	m_9	m_{11}	m_{10}

- There is only one possibility of grouping 16 adjacent min terms.
- Let R_1, R_2, R_3 and R_4 represents the min terms of first row, second row, third row and fourth row respectively. Similarly, C_1, C_2, C_3 and C_4 represents the min terms of first column, second column, third column and fourth column respectively. The possible combinations of grouping 8 adjacent min terms are $\{(R_1, R_2), (R_2, R_3), (R_3, R_4), (R_4, R_1), (C_1, C_2), (C_2, C_3), (C_3, C_4), (C_4, C_1)\}$.
- If $w=0$, then 4 variable K-map becomes 3 variable K-map.

5 Variable K-Map

The number of cells in 5 variable K-map is thirty-two, since the number of variables is 5. The following figure shows **5 variable K-Map**.

		V=0						V=1					
		WX	YZ	00	01	11	10	WX	YZ	00	01	11	10
Y	X	00		m ₀	m ₁	m ₃	m ₂	00		m ₁₆	m ₁₇	m ₁₉	m ₁₈
		01		m ₄	m ₅	m ₇	m ₆	01		m ₂₀	m ₂₁	m ₂₃	m ₂₂
		11		m ₁₂	m ₁₃	m ₁₅	m ₁₄	11		m ₂₈	m ₂₉	m ₃₁	m ₃₀
		10		m ₈	m ₉	m ₁₁	m ₁₀	10		m ₂₄	m ₂₅	m ₂₇	m ₂₆

- There is only one possibility of grouping 32 adjacent min terms.
- There are two possibilities of grouping 16 adjacent min terms. i.e., grouping of min terms from m₀ to m₁₅ and m₁₆ to m₃₁.
- If v=0, then 5 variable K-map becomes 4 variable K-map.

In the above all K-maps, we used exclusively the min terms notation. Similarly, you can use exclusively the Max terms notation.

Minimization of Boolean Functions using K-Maps

If we consider the combination of inputs for which the Boolean function is '1', then we will get the Boolean function, which is in **standard sum of products** form after simplifying the K-map.

Similarly, if we consider the combination of inputs for which the Boolean function is '0', then we will get the Boolean function, which is in **standard product of sums** form after simplifying the K-map.

Follow these **rules for simplifying K-maps** in order to get standard sum of products form.

- Select the respective K-map based on the number of variables present in the Boolean function.
- If the Boolean function is given as sum of min terms form, then place the ones at respective min term cells in the K-map. If the Boolean function is given as sum of products form, then place the ones in all possible cells of K-map for which the given product terms are valid.
- Check for the possibilities of grouping maximum number of adjacent ones. It should be powers of two. Start from highest power of two and upto least power of two. Highest power is equal to the number of variables considered in K-map and least power is zero.
- Each grouping will give either a literal or one product term. It is known as **prime implicant**. The prime implicant is said to be **essential prime implicant**, if atleast single '1' is not covered with any other groupings but only that grouping covers.

- Note down all the prime implicants and essential prime implicants. The simplified Boolean function contains all essential prime implicants and only the required prime implicants.

Note 1 – If outputs are not defined for some combination of inputs, then those output values will be represented with **don't care symbol 'x'**. That means, we can consider them as either '0' or '1'.

Note 2 – If don't care terms also present, then place don't cares 'x' in the respective cells of K-map. Consider only the don't cares 'x' that are helpful for grouping maximum number of adjacent ones. In those cases, treat the don't care value as '1'.

Example

Let us **simplify** the following Boolean function, $f(W, X, Y, Z) = WX'Y' + WY + W'YZ'$ using K-map.

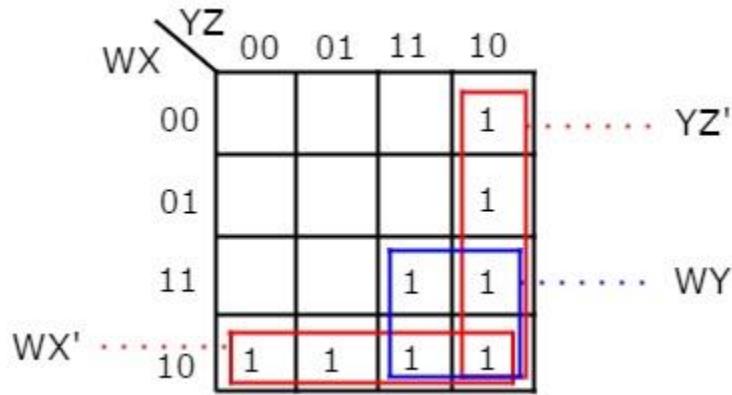
The given Boolean function is in sum of products form. It is having 4 variables W, X, Y & Z. So, we require **4 variable K-map**. The **4 variable K-map** with ones corresponding to the given product terms is shown in the following figure.

		YZ \ WX	00	01	11	10
		00				1
		01				1
		11			1	1
		10	1	1	1	1

Here, 1s are placed in the following cells of K-map.

- The cells, which are common to the intersection of Row 4 and columns 1 & 2 are corresponding to the product term, $WX'Y'$.
- The cells, which are common to the intersection of Rows 3 & 4 and columns 3 & 4 are corresponding to the product term, WY .
- The cells, which are common to the intersection of Rows 1 & 2 and column 4 are corresponding to the product term, $W'YZ'$.

There are no possibilities of grouping either 16 adjacent ones or 8 adjacent ones. There are three possibilities of grouping 4 adjacent ones. After these three groupings, there is no single one left as ungrouped. So, we no need to check for grouping of 2 adjacent ones. The **4 variable K-map** with these three **groupings** is shown in the following figure.



Here, we got three prime implicants WX' , WY & YZ' . All these prime implicants are **essential** because of following reasons.

- Two ones (m_8 & m_9) of fourth row grouping are not covered by any other groupings. Only fourth row grouping covers those two ones.
- Single one (m_{15}) of square shape grouping is not covered by any other groupings. Only the square shape grouping covers that one.
- Two ones (m_2 & m_6) of fourth column grouping are not covered by any other groupings. Only fourth column grouping covers those two ones.

Therefore, the **simplified Boolean function** is

$$f = WX' + WY + YZ'$$

Follow these **rules for simplifying K-maps** in order to get standard product of sums form.

- Select the respective K-map based on the number of variables present in the Boolean function.
- If the Boolean function is given as product of Max terms form, then place the zeroes at respective Max term cells in the K-map. If the Boolean function is given as product of sums form, then place the zeroes in all possible cells of K-map for which the given sum terms are valid.
- Check for the possibilities of grouping maximum number of adjacent zeroes. It should be powers of two. Start from highest power of two and upto least power of two. Highest power is equal to the number of variables considered in K-map and least power is zero.
- Each grouping will give either a literal or one sum term. It is known as **prime implicant**. The prime implicant is said to be **essential prime implicant**, if atleast single '0' is not covered with any other groupings but only that grouping covers.
- Note down all the prime implicants and essential prime implicants. The simplified Boolean function contains all essential prime implicants and only the required prime implicants.

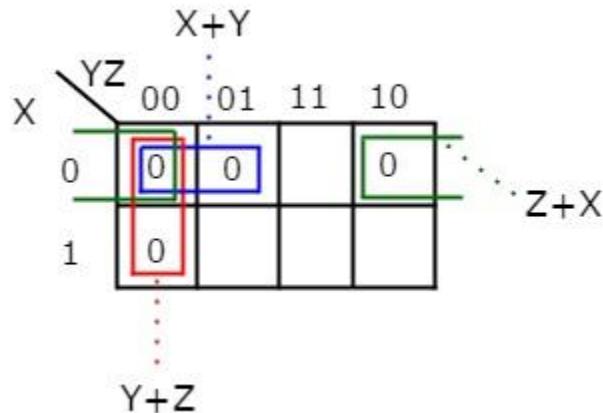
Note – If don't care terms also present, then place don't cares 'x' in the respective cells of K-map. Consider only the don't cares 'x' that are helpful for grouping maximum number of adjacent zeroes. In those cases, treat the don't care value as '0'.

Example

Let us **simplify** the following Boolean function, $f(X,Y,Z) = \prod M(0,1,2,4)$ using K-map. The given Boolean function is in product of Max terms form. It is having 3 variables X, Y & Z. So, we require 3 variable K-map. The given Max terms are M_0, M_1, M_2 & M_4 . The **3 variable K-map** with zeroes corresponding to the given Max terms is shown in the following figure.

		YZ	00	01	11	10
		X	0	0		0
		0	0			
		1	0			

There are no possibilities of grouping either 8 adjacent zeroes or 4 adjacent zeroes. There are three possibilities of grouping 2 adjacent zeroes. After these three groupings, there is no single zero left as ungrouped. The **3 variable K-map** with these three **groupings** is shown in the following figure.



Here, we got three prime implicants $X + Y$, $Y + Z$ & $Z + X$. All these prime implicants are **essential** because one zero in each grouping is not covered by any other groupings except with their individual groupings.

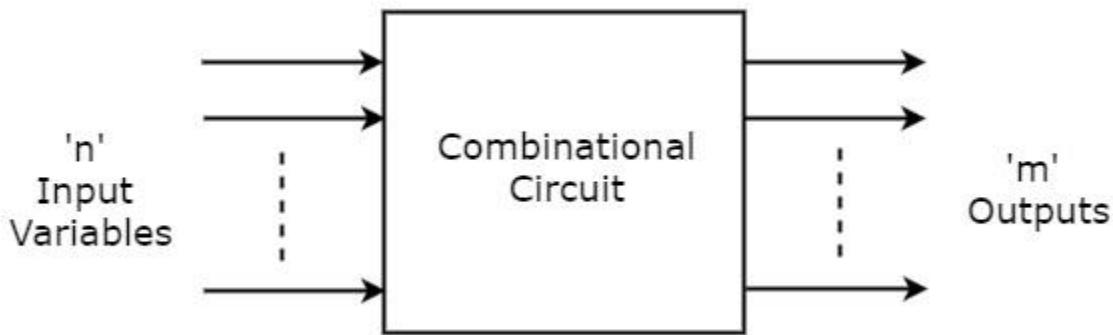
Therefore, the **simplified Boolean function** is

$$f = (X + Y).(Y + Z).(Z + X)$$

In this way, we can easily simplify the Boolean functions up to 5 variables using K-map method. For more than 5 variables, it is difficult to simplify the functions using K-Maps. Because, the number of **cells** in K-map gets **doubled** by including a new variable.

Combinational Circuits

Combinational circuits consist of Logic gates. These circuits operate with binary values. The output(s) of combinational circuit depends on the combination of present inputs. The following figure shows the **block diagram** of combinational circuit.



This combinational circuit has 'n' input variables and 'm' outputs. Each combination of input variables will affect the output(s).

Design procedure of Combinational circuits

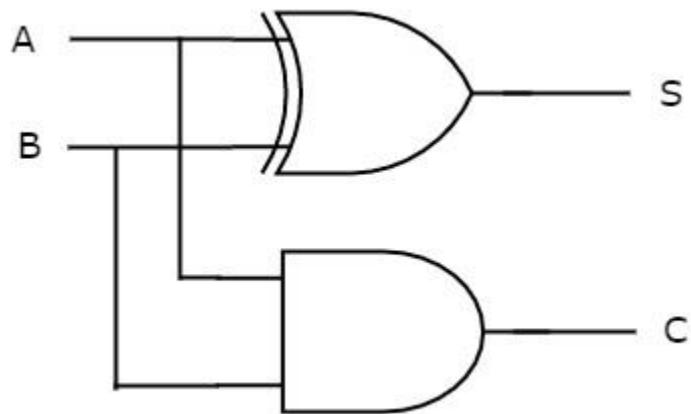
- Find the required number of input variables and outputs from given specifications.
- Formulate the **Truth table**. If there are 'n' input variables, then there will be 2^n possible combinations. For each combination of input, find the output values.
- Find the **Boolean expressions** for each output. If necessary, simplify those expressions.
- Implement the above Boolean expressions corresponding to each output by using **Logic gates**.
- In this chapter, let us discuss about the basic arithmetic circuits like Binary adder and Binary subtractor. These circuits can be operated with binary values 0 and 1.
- Binary Adder
- The most basic arithmetic operation is addition. The circuit, which performs the addition of two binary numbers is known as **Binary adder**. First, let us implement an adder, which performs the addition of two bits.

Half Adder

- Half adder is a combinational circuit, which performs the addition of two binary numbers A and B are of **single bit**. It produces two outputs sum, S & carry, C.
- The **Truth table** of Half adder is shown below.

Inputs		Outputs	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- When we do the addition of two bits, the resultant sum can have the values ranging from 0 to 2 in decimal. We can represent the decimal digits 0 and 1 with single bit in binary. But, we can't represent decimal digit 2 with single bit in binary. So, we require two bits for representing it in binary.
- Let, sum, S is the Least significant bit and carry, C is the Most significant bit of the resultant sum. For first three combinations of inputs, carry, C is zero and the value of S will be either zero or one based on the **number of ones** present at the inputs. But, for last combination of inputs, carry, C is one and sum, S is zero, since the resultant sum is two.
- From Truth table, we can directly write the **Boolean functions** for each output as
 - $S = A \oplus B$
- $C = ABC = AB$
- We can implement the above functions with 2-input Ex-OR gate & 2-input AND gate. The **circuit diagram** of Half adder is shown in the following figure.



- In the above circuit, a two input Ex-OR gate & two input AND gate produces sum, S & carry, C respectively. Therefore, Half-adder performs the addition of two bits.

Full Adder

- Full adder is a combinational circuit, which performs the **addition of three bits A, B and C_{in}**. Where, A & B are the two parallel significant bits and C_{in} is the carry bit, which is generated from previous stage. This Full adder also produces two outputs sum, S & carry, C_{out}, which are similar to Half adder.
- The **Truth table** of Full adder is shown below.

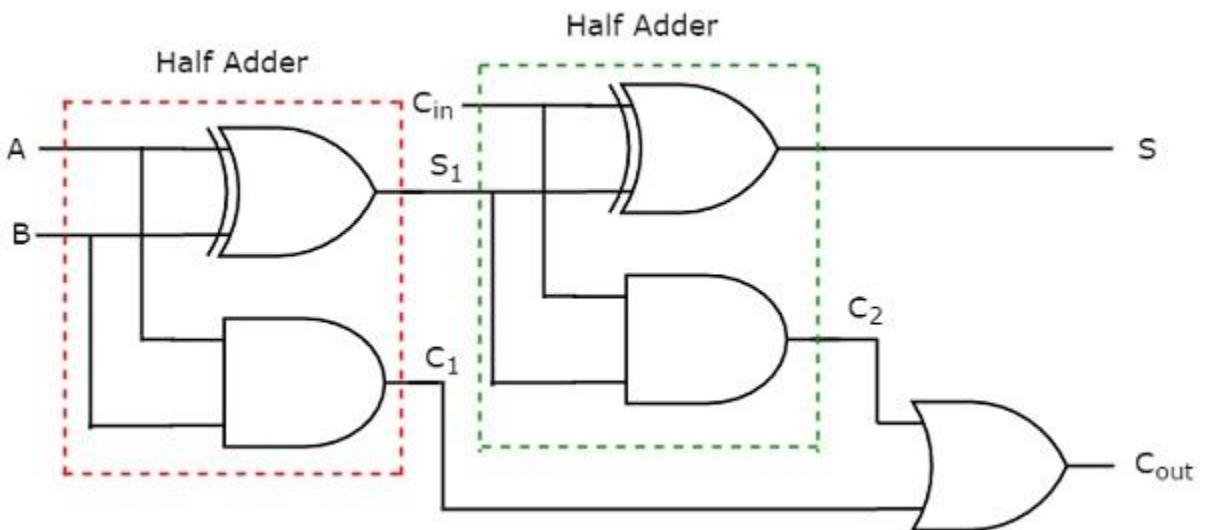
Inputs			Outputs	
A	B	C _{in}	C _{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- When we do the addition of three bits, the resultant sum can have the values ranging from 0 to 3 in decimal. We can represent the decimal digits 0 and 1 with single bit in binary. But, we can't represent the decimal digits 2 and 3 with single bit in binary. So, we require two bits for representing those two decimal digits in binary.
- Let, sum, S is the Least significant bit and carry, C_{out} is the Most significant bit of resultant sum. It is easy to fill the values of outputs for all combinations of inputs in the truth table. Just count

the **number of ones** present at the inputs and write the equivalent binary number at outputs. If C_{in} is equal to zero, then Full adder truth table is same as that of Half adder truth table.

- We will get the following **Boolean functions** for each output after simplification.

- $S = A \oplus B \oplus C_{in}$
- $C_{out} = AB + (A \oplus B)C_{in}$
- The sum, S is equal to one, when odd number of ones present at the inputs. We know that Ex-OR gate produces an output, which is an odd function. So, we can use either two 2-input Ex-OR gates or one 3-input Ex-OR gate in order to produce sum, S . We can implement carry, C_{out} using two 2-input AND gates & one OR gate. The **circuit diagram** of Full adder is shown in the following figure.

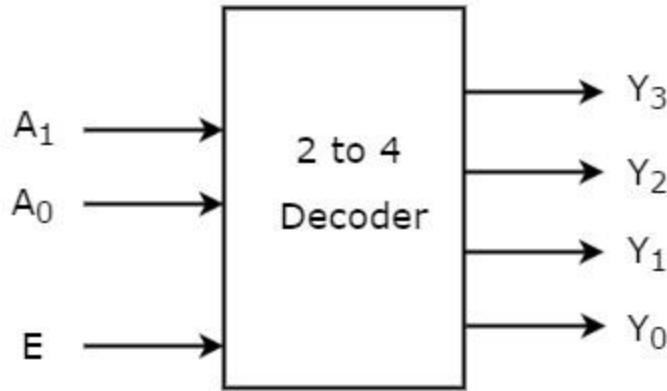


- This adder is called as **Full adder** because for implementing one Full adder, we require two Half adders and one OR gate. If C_{in} is zero, then Full adder becomes Half adder. We can verify it easily from the above circuit diagram or from the Boolean functions of outputs of Full adder.

Decoder is a combinational circuit that has ' n ' input lines and maximum of 2^n output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means decoder detects a particular code. The outputs of the decoder are nothing but the **min terms** of ' n ' input variables (lines), when it is enabled.

2 to 4 Decoder

Let 2 to 4 Decoder has two inputs A_1 & A_0 and four outputs Y_3 , Y_2 , Y_1 & Y_0 . The **block diagram** of 2 to 4 decoder is shown in the following figure.



One of these four outputs will be '1' for each combination of inputs when enable, E is '1'. The **Truth table** of 2 to 4 decoder is shown below.

Enable	Inputs		Outputs			
	E	A_1	A_0	Y_3	Y_2	Y_1
0	x	x	0	0	0	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

From Truth table, we can write the **Boolean functions** for each output as

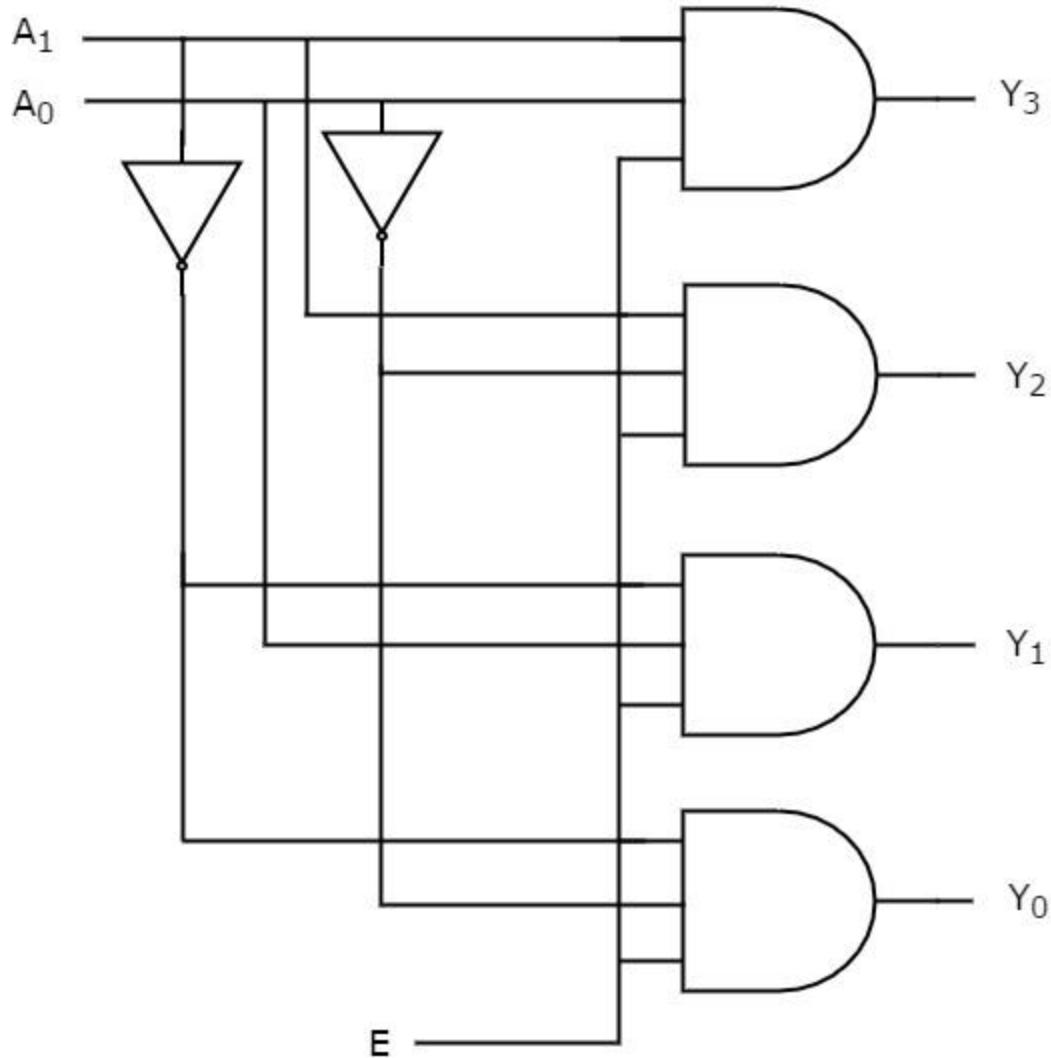
$$Y_3 = E \cdot A_1 \cdot A_0 \\ Y_3 = E \cdot A_1 \cdot A_0$$

$$Y_2 = E \cdot A_1 \cdot A_0' \\ Y_2 = E \cdot A_1 \cdot A_0'$$

$$Y_1 = E \cdot A_1' \cdot A_0 \\ Y_1 = E \cdot A_1' \cdot A_0$$

$$Y_0 = E \cdot A_1' \cdot A_0' \\ Y_0 = E \cdot A_1' \cdot A_0'$$

Each output is having one product term. So, there are four product terms in total. We can implement these four product terms by using four AND gates having three inputs each & two inverters. The **circuit diagram** of 2 to 4 decoder is shown in the following figure.



Therefore, the outputs of 2 to 4 decoder are nothing but the **min terms** of two input variables A_1 & A_0 , when enable, E is equal to one. If enable, E is zero, then all the outputs of decoder will be equal to zero.

Similarly, 3 to 8 decoder produces eight min terms of three input variables A_2 , A_1 & A_0 and 4 to 16 decoder produces sixteen min terms of four input variables A_3 , A_2 , A_1 & A_0 .

Implementation of Higher-order Decoders

Now, let us implement the following two higher-order decoders using lower-order decoders.

- 3 to 8 decoder
- 4 to 16 decoder

3 to 8 Decoder

In this section, let us implement **3 to 8 decoder using 2 to 4 decoders**. We know that 2 to 4 Decoder has two inputs, A_1 & A_0 and four outputs, Y_3 to Y_0 . Whereas, 3 to 8 Decoder has three inputs A_2 , A_1 & A_0 and eight outputs, Y_7 to Y_0 .

We can find the number of lower order decoders required for implementing higher order decoder using the following formula.

$$\text{Required number of lower order decoders} = m_2 m_1 \quad \text{Required number of lower order decoders} = m_2 m_1$$

Where,

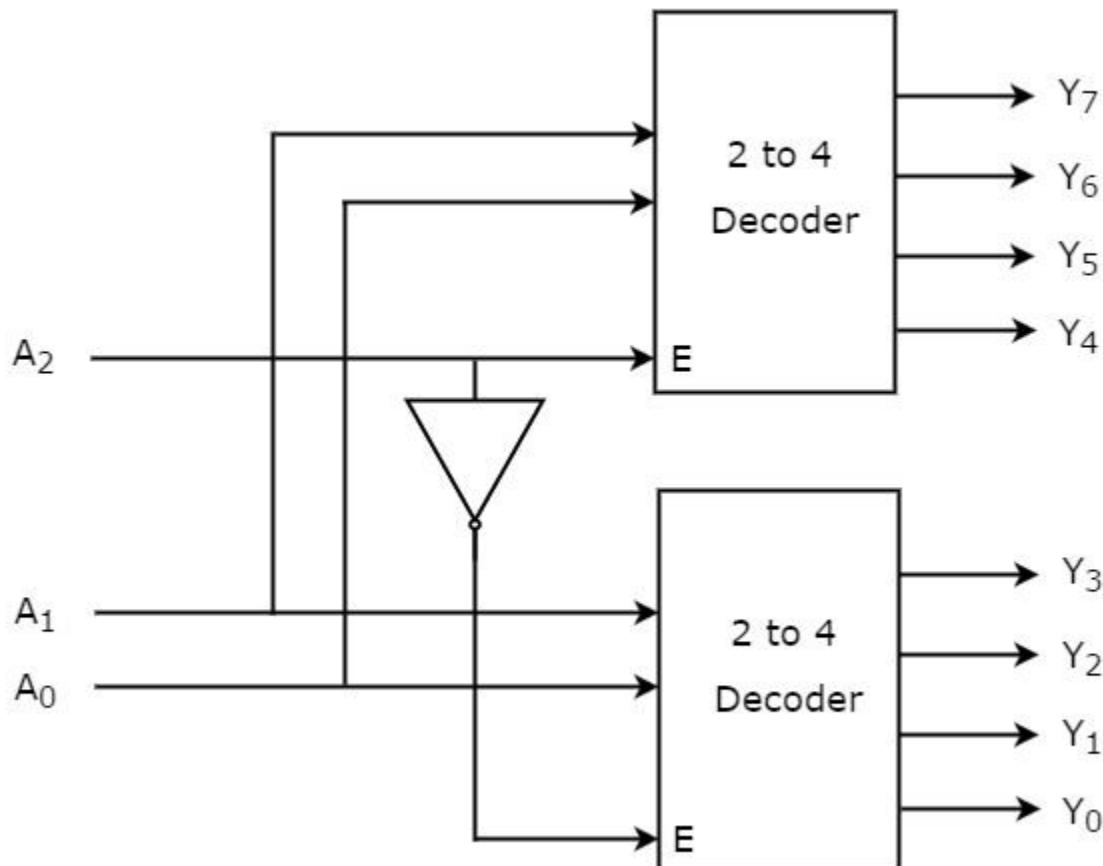
$m_1 m_1$ is the number of outputs of lower order decoder.

$m_2 m_2$ is the number of outputs of higher order decoder.

Here, $m_1 m_1 = 4$ and $m_2 m_2 = 8$. Substitute, these two values in the above formula.

$$\text{Required number of 2 to 4 decoders} = 84 = 2 \quad \text{Required number of 2 to 4 decoders} = 84 = 2$$

Therefore, we require two 2 to 4 decoders for implementing one 3 to 8 decoder. The **block diagram** of 3 to 8 decoder using 2 to 4 decoders is shown in the following figure.



The parallel inputs A_1 & A_0 are applied to each 2 to 4 decoder. The complement of input A_2 is connected to Enable, E of lower 2 to 4 decoder in order to get the outputs, Y_3 to Y_0 . These are

the **lower four min terms**. The input, A_2 is directly connected to Enable, E of upper 2 to 4 decoder in order to get the outputs, Y_7 to Y_4 . These are the **higher four min terms**.

4 to 16 Decoder

In this section, let us implement **4 to 16 decoder using 3 to 8 decoders**. We know that 3 to 8 Decoder has three inputs A_2 , A_1 & A_0 and eight outputs, Y_7 to Y_0 . Whereas, 4 to 16 Decoder has four inputs A_3 , A_2 , A_1 & A_0 and sixteen outputs, Y_{15} to Y_0

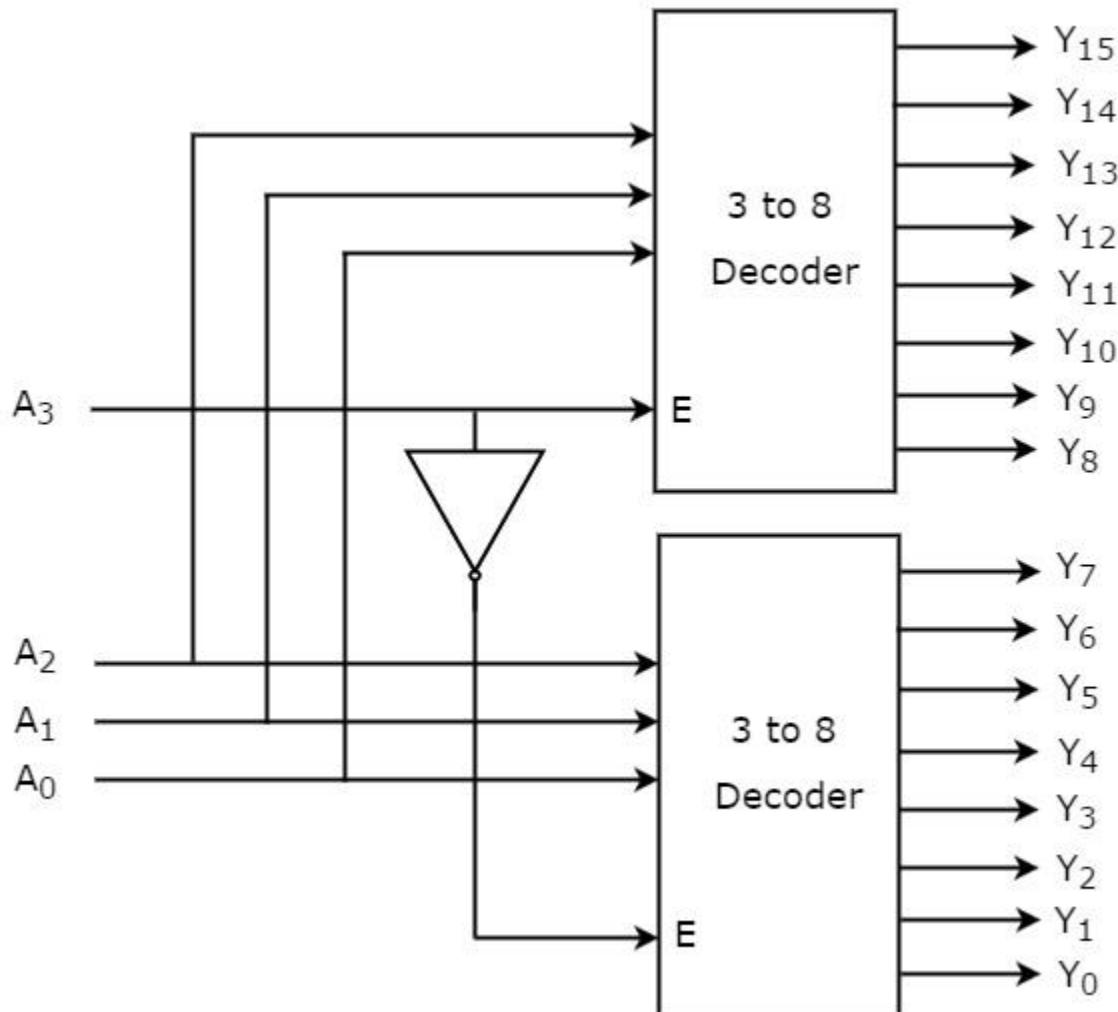
We know the following formula for finding the number of lower order decoders required.

$$\text{Required number of lower order decoders} = m_2 m_1 \quad \text{Required number of higher order decoders} = m_2 m_1$$

Substitute, $m_1 m_1 = 8$ and $m_2 m_2 = 16$ in the above formula.

$$\text{Required number of 3 to 8 decoders} = 16 \cdot 8 = 2 \quad \text{Required number of 3 to 8 decoders} = 16 \cdot 8 = 2$$

Therefore, we require two 3 to 8 decoders for implementing one 4 to 16 decoder. The **block diagram** of 4 to 16 decoder using 3 to 8 decoders is shown in the following figure.

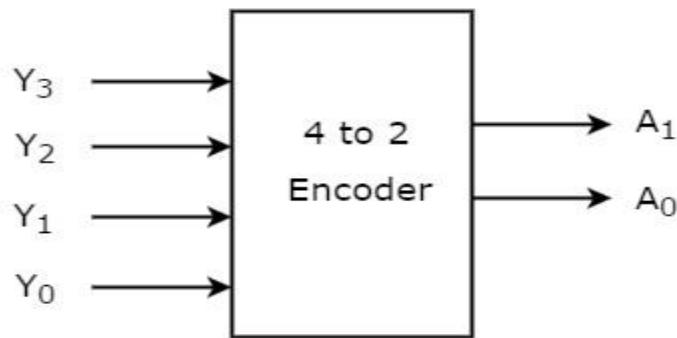


The parallel inputs A_2 , A_1 & A_0 are applied to each 3 to 8 decoder. The complement of input, A_3 is connected to Enable, E of lower 3 to 8 decoder in order to get the outputs, Y_7 to Y_0 . These are the **lower eight min terms**. The input, A_3 is directly connected to Enable, E of upper 3 to 8 decoder in order to get the outputs, Y_{15} to Y_8 . These are the **higher eight min terms**.

An **Encoder** is a combinational circuit that performs the reverse operation of Decoder. It has maximum of 2^n input lines and 'n' output lines. It will produce a binary code equivalent to the input, which is active High. Therefore, the encoder encodes 2^n input lines with 'n' bits. It is optional to represent the enable signal in encoders.

4 to 2 Encoder

Let 4 to 2 Encoder has four inputs Y_3 , Y_2 , Y_1 & Y_0 and two outputs A_1 & A_0 . The **block diagram** of 4 to 2 Encoder is shown in the following figure.



At any time, only one of these 4 inputs can be '1' in order to get the respective binary code at the output. The **Truth table** of 4 to 2 encoder is shown below.

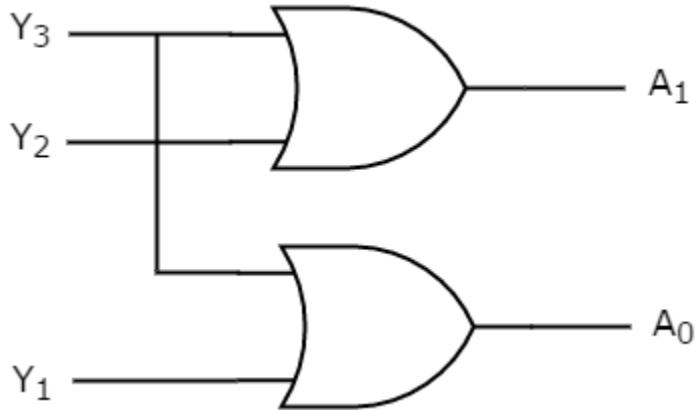
Inputs				Outputs	
Y_3	Y_2	Y_1	Y_0	A_1	A_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

From Truth table, we can write the **Boolean functions** for each output as

$$A_1 = Y_3 + Y_2 \quad A_1 = Y_3 + Y_2$$

$$A_0 = Y_3 + Y_1 \quad A_0 = Y_3 + Y_1$$

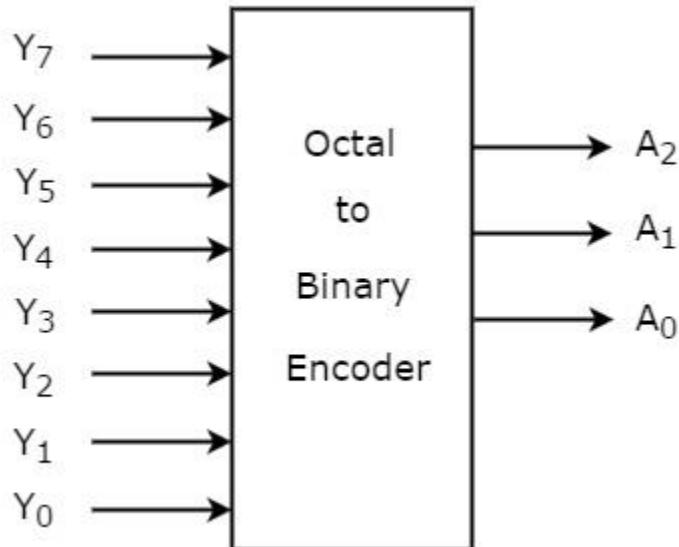
We can implement the above two Boolean functions by using two input OR gates. The **circuit diagram** of 4 to 2 encoder is shown in the following figure.



The above circuit diagram contains two OR gates. These OR gates encode the four inputs with two bits

Octal to Binary Encoder

Octal to binary Encoder has eight inputs, Y_7 to Y_0 and three outputs A_2 , A_1 & A_0 . Octal to binary encoder is nothing but 8 to 3 encoder. The **block diagram** of octal to binary Encoder is shown in the following figure.



At any time, only one of these eight inputs can be '1' in order to get the respective binary code. The **Truth table** of octal to binary encoder is shown below.

Inputs								Outputs		
Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

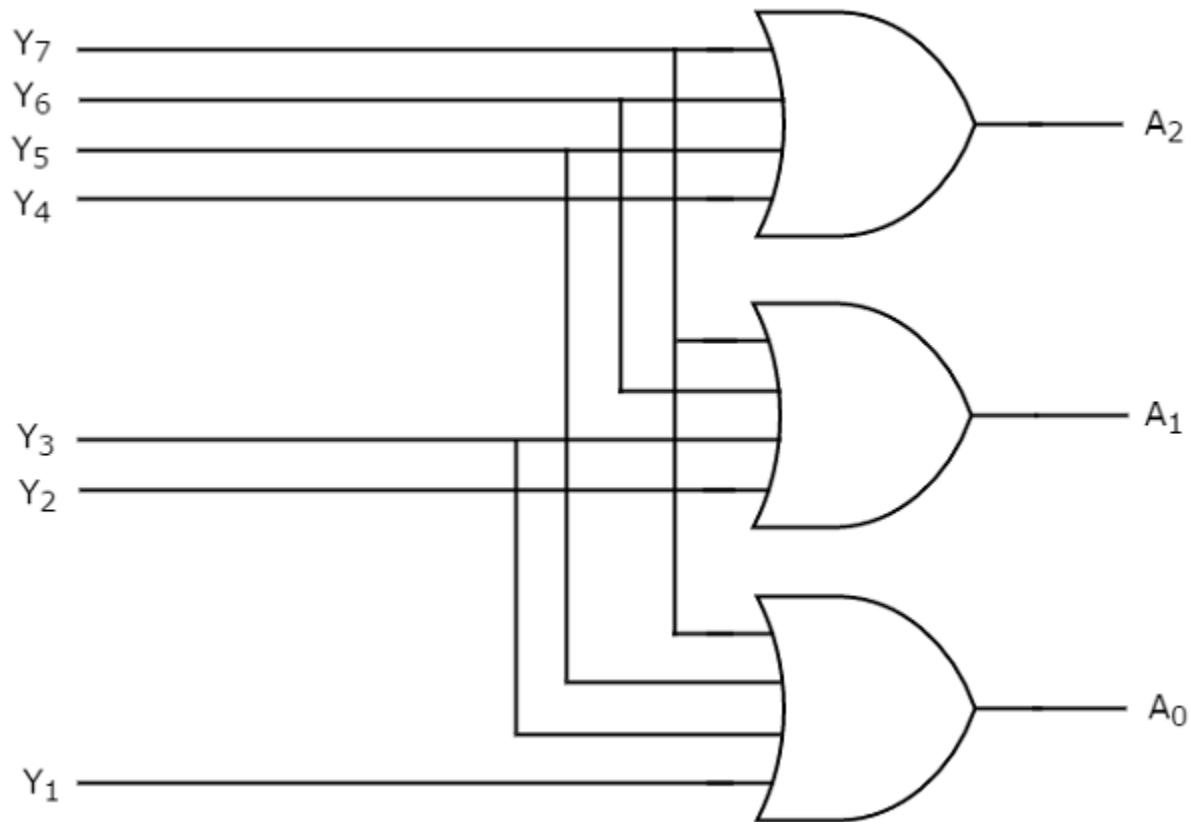
From Truth table, we can write the **Boolean functions** for each output as

$$A_2 = Y_7 + Y_6 + Y_5 + Y_4 \quad A_2 = Y_7 + Y_6 + Y_5 + Y_4$$

$$A_1 = Y_7 + Y_6 + Y_3 + Y_2 \quad A_1 = Y_7 + Y_6 + Y_3 + Y_2$$

$$A_0 = Y_7 + Y_5 + Y_3 + Y_1 \quad A_0 = Y_7 + Y_5 + Y_3 + Y_1$$

We can implement the above Boolean functions by using four input OR gates. The **circuit diagram** of octal to binary encoder is shown in the following figure.



The above circuit diagram contains three 4-input OR gates. These OR gates encode the eight inputs with three bits.

Drawbacks of Encoder

Following are the drawbacks of normal encoder.

- There is an ambiguity, when all outputs of encoder are equal to zero. Because, it could be the code corresponding to the inputs, when only least significant input is one or when all inputs are zero.
- If more than one input is active High, then the encoder produces an output, which may not be the correct code. For **example**, if both Y_3 and Y_6 are '1', then the encoder produces 111 at the output. This is neither equivalent code corresponding to Y_3 , when it is '1' nor the equivalent code corresponding to Y_6 , when it is '1'.

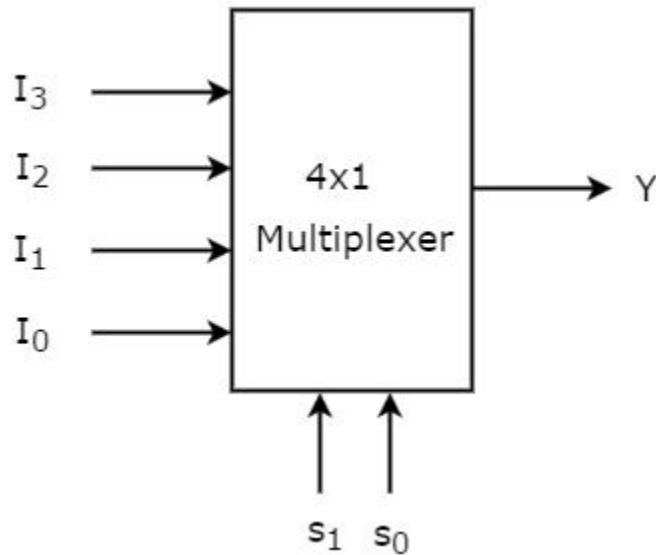
Multiplexer

is a combinational circuit that has maximum of 2^n data inputs, ' n ' selection lines and single output line. One of these data inputs will be connected to the output based on the values of selection lines.

Since there are ' n ' selection lines, there will be 2^n possible combinations of zeros and ones. So, each combination will select only one data input. Multiplexer is also called as **Mux**.

4x1 Multiplexer

4x1 Multiplexer has four data inputs I_3 , I_2 , I_1 & I_0 , two selection lines s_1 & s_0 and one output Y . The **block diagram** of 4x1 Multiplexer is shown in the following figure.



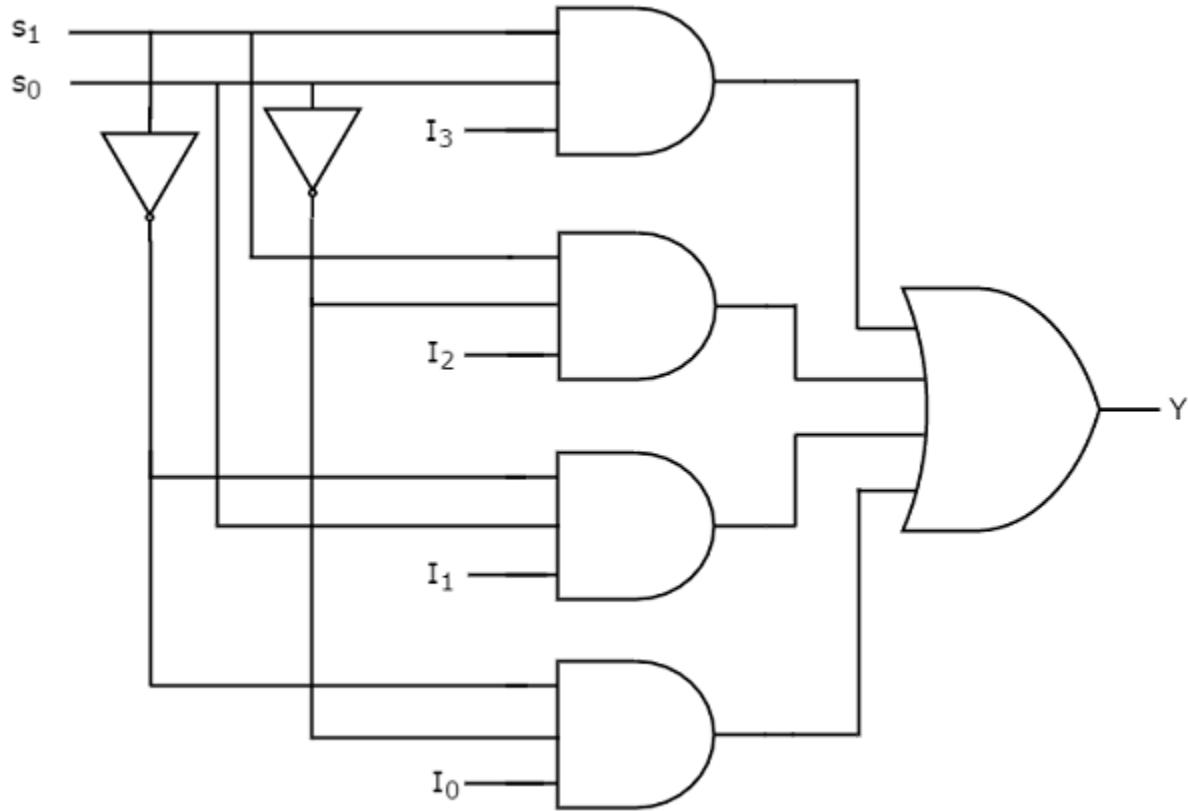
One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines. **Truth table** of 4x1 Multiplexer is shown below.

Selection Lines		Output
s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

From Truth table, we can directly write the **Boolean function** for output, Y as

$$Y = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3 \\ Y = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$$

We can implement this Boolean function using Inverters, AND gates & OR gate. The **circuit diagram** of 4x1 multiplexer is shown in the following figure.



We can easily understand the operation of the above circuit. Similarly, you can implement 8x1 Multiplexer and 16x1 multiplexer by following the same procedure.

Implementation of Higher-order Multiplexers.

Now, let us implement the following two higher-order Multiplexers using lower-order Multiplexers.

- 8x1 Multiplexer
- 16x1 Multiplexer

8x1 Multiplexer

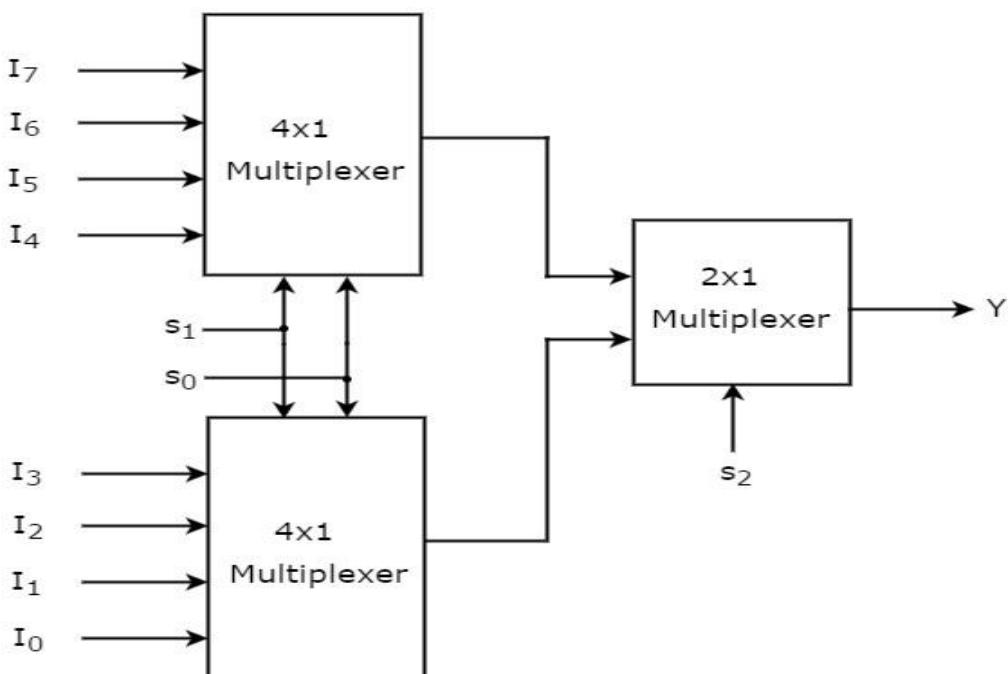
In this section, let us implement 8x1 Multiplexer using 4x1 Multiplexers and 2x1 Multiplexer. We know that 4x1 Multiplexer has 4 data inputs, 2 selection lines and one output. Whereas, 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output.

So, we require two **4x1 Multiplexers** in first stage in order to get the 8 data inputs. Since, each 4x1 Multiplexer produces one output, we require a **2x1 Multiplexer** in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 8x1 Multiplexer has eight data inputs I_7 to I_0 , three selection lines s_2 , s_1 & s_0 and one output Y . The **Truth table** of 8x1 Multiplexer is shown below.

Selection Inputs			Output
s_2	s_1	s_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

We can implement 8×1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 8×1 Multiplexer is shown in the following figure.



The same **selection lines**, s_1 & s_0 are applied to both 4x1 Multiplexers. The data inputs of upper 4x1 Multiplexer are I_7 to I_4 and the data inputs of lower 4x1 Multiplexer are I_3 to I_0 . Therefore, each 4x1 Multiplexer produces an output based on the values of selection lines, s_1 & s_0 .

The outputs of first stage 4x1 Multiplexers are applied as inputs of 2x1 Multiplexer that is present in second stage. The other **selection line**, s_2 is applied to 2x1 Multiplexer.

- If s_2 is zero, then the output of 2x1 Multiplexer will be one of the 4 inputs I_3 to I_0 based on the values of selection lines s_1 & s_0 .
- If s_2 is one, then the output of 2x1 Multiplexer will be one of the 4 inputs I_7 to I_4 based on the values of selection lines s_1 & s_0 .

Therefore, the overall combination of two 4x1 Multiplexers and one 2x1 Multiplexer performs as one 8x1 Multiplexer.

16x1 Multiplexer

In this section, let us implement 16x1 Multiplexer using 8x1 Multiplexers and 2x1 Multiplexer. We know that 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output. Whereas, 16x1 Multiplexer has 16 data inputs, 4 selection lines and one output.

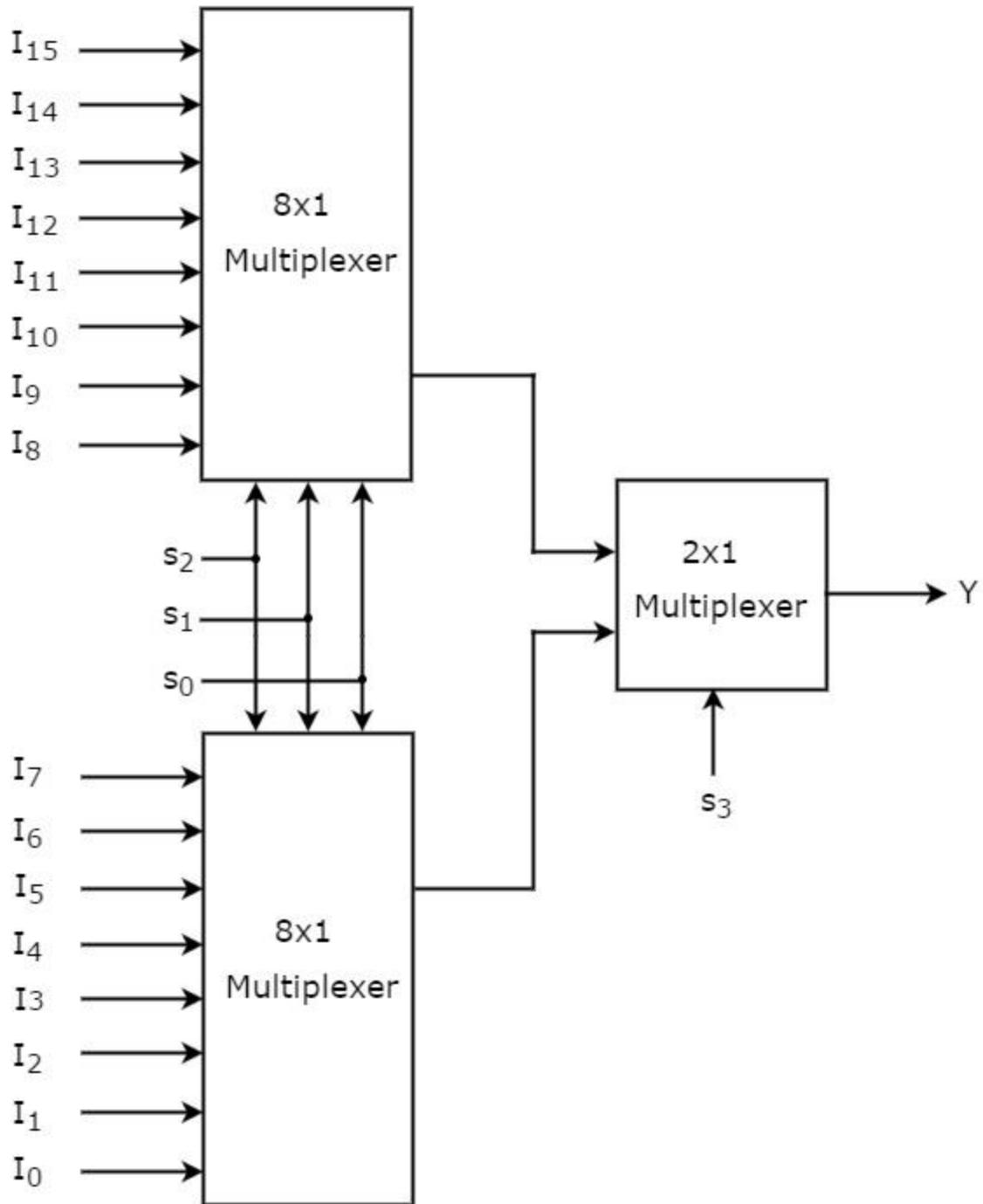
So, we require two **8x1 Multiplexers** in first stage in order to get the 16 data inputs. Since, each 8x1 Multiplexer produces one output, we require a 2x1 Multiplexer in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 16x1 Multiplexer has sixteen data inputs I_{15} to I_0 , four selection lines s_3 to s_0 and one output Y. The **Truth table** of 16x1 Multiplexer is shown below.

Selection Inputs				Output
s_3	s_2	s_1	s_0	Y
0	0	0	0	I_0
0	0	0	1	I_1
0	0	1	0	I_2
0	0	1	1	I_3

0	1	0	0	I_4
0	1	0	1	I_5
0	1	1	0	I_6
0	1	1	1	I_7
1	0	0	0	I_8
1	0	0	1	I_9
1	0	1	0	I_{10}
1	0	1	1	I_{11}
1	1	0	0	I_{12}
1	1	0	1	I_{13}
1	1	1	0	I_{14}
1	1	1	1	I_{15}

We can implement 16x1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 16x1 Multiplexer is shown in the following figure.



The **same selection lines**, s_2 , s_1 & s_0 are applied to both **8x1 Multiplexers**. The data inputs of upper **8x1 Multiplexer** are I_{15} to I_8 and the data inputs of lower **8x1 Multiplexer** are I_7 to I_0 . Therefore, each **8x1 Multiplexer** produces an output based on the values of selection lines, s_2 , s_1 & s_0 .

The outputs of first stage **8x1 Multiplexers** are applied as inputs of **2x1 Multiplexer** that is present in second stage. The other **selection line**, s_3 is applied to **2x1 Multiplexer**.

- If s_3 is zero, then the output of **2x1 Multiplexer** will be one of the 8 inputs I_7 to I_0 based on the values of selection lines s_2 , s_1 & s_0 .

- If s_3 is one, then the output of 2x1 Multiplexer will be one of the 8 inputs I_{15} to I_8 based on the values of selection lines s_2 , s_1 & s_0 .

Therefore, the overall combination of two 8x1 Multiplexers and one 2x1 Multiplexer performs as one 16x1 Multiplexer.

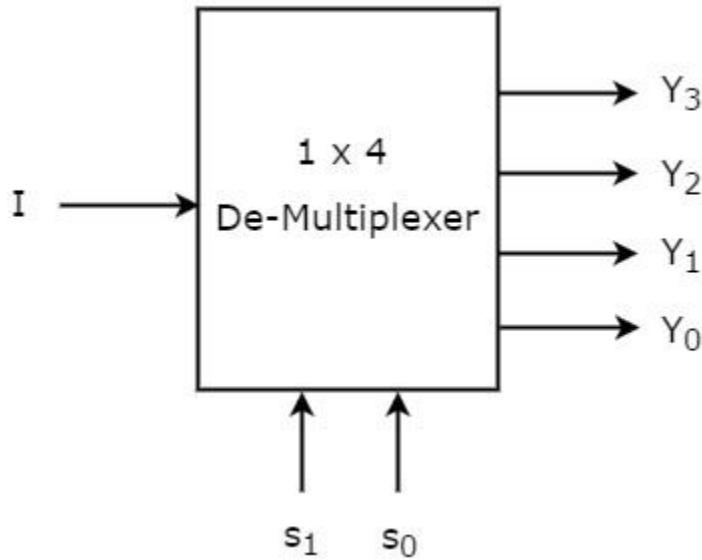
De-Multiplexer

is a combinational circuit that performs the reverse operation of Multiplexer. It has single input, 'n' selection lines and maximum of 2^n outputs. The input will be connected to one of these outputs based on the values of selection lines.

Since there are 'n' selection lines, there will be 2^n possible combinations of zeros and ones. So, each combination can select only one output. De-Multiplexer is also called as **De-Mux**.

1x4 De-Multiplexer

1x4 De-Multiplexer has one input I , two selection lines, s_1 & s_0 and four outputs Y_3 , Y_2 , Y_1 & Y_0 . The **block diagram** of 1x4 De-Multiplexer is shown in the following figure.



The single input 'I' will be connected to one of the four outputs, Y_3 to Y_0 based on the values of selection lines s_1 & s_0 . The **Truth table** of 1x4 De-Multiplexer is shown below.

Selection Inputs		Outputs			
s_1	s_0	Y_3	Y_2	Y_1	Y_0

0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0

From the above Truth table, we can directly write the **Boolean functions** for each output as

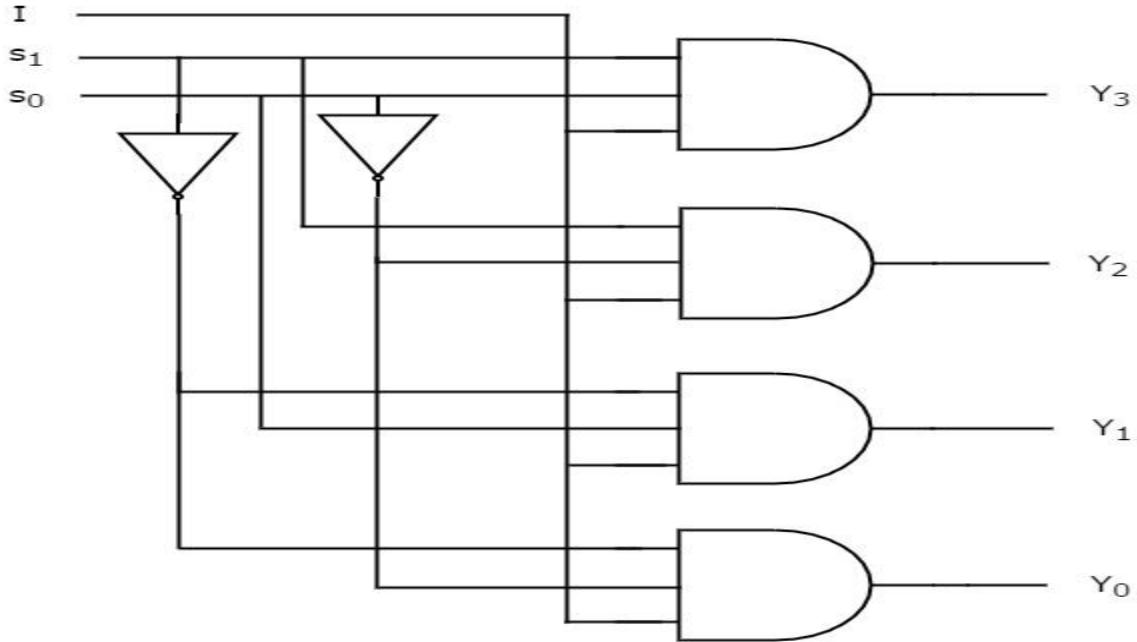
$$Y_3 = s_1 s_0 I \quad Y_3 = s_1 s_0 I$$

$$Y_2 = s_1 s_0' I \quad Y_2 = s_1 s_0' I$$

$$Y_1 = s_1' s_0 I \quad Y_1 = s_1' s_0 I$$

$$Y_0 = s_1' s_0' I \quad Y_0 = s_1' s_0' I$$

We can implement these Boolean functions using Inverters & 3-input AND gates. The **circuit diagram** of 1x4 De-Multiplexer is shown in the following figure.



We can easily understand the operation of the above circuit. Similarly, you can implement 1x8 De-Multiplexer and 1x16 De-Multiplexer by following the same procedure.

Implementation of Higher-order De-Multiplexers

Now, let us implement the following two higher-order De-Multiplexers using lower-order De-Multiplexers.

- 1x8 De-Multiplexer
- 1x16 De-Multiplexer

1x8 De-Multiplexer

In this section, let us implement 1x8 De-Multiplexer using 1x4 De-Multiplexers and 1x2 De-Multiplexer. We know that 1x4 De-Multiplexer has single input, two selection lines and four outputs. Whereas, 1x8 De-Multiplexer has single input, three selection lines and eight outputs.

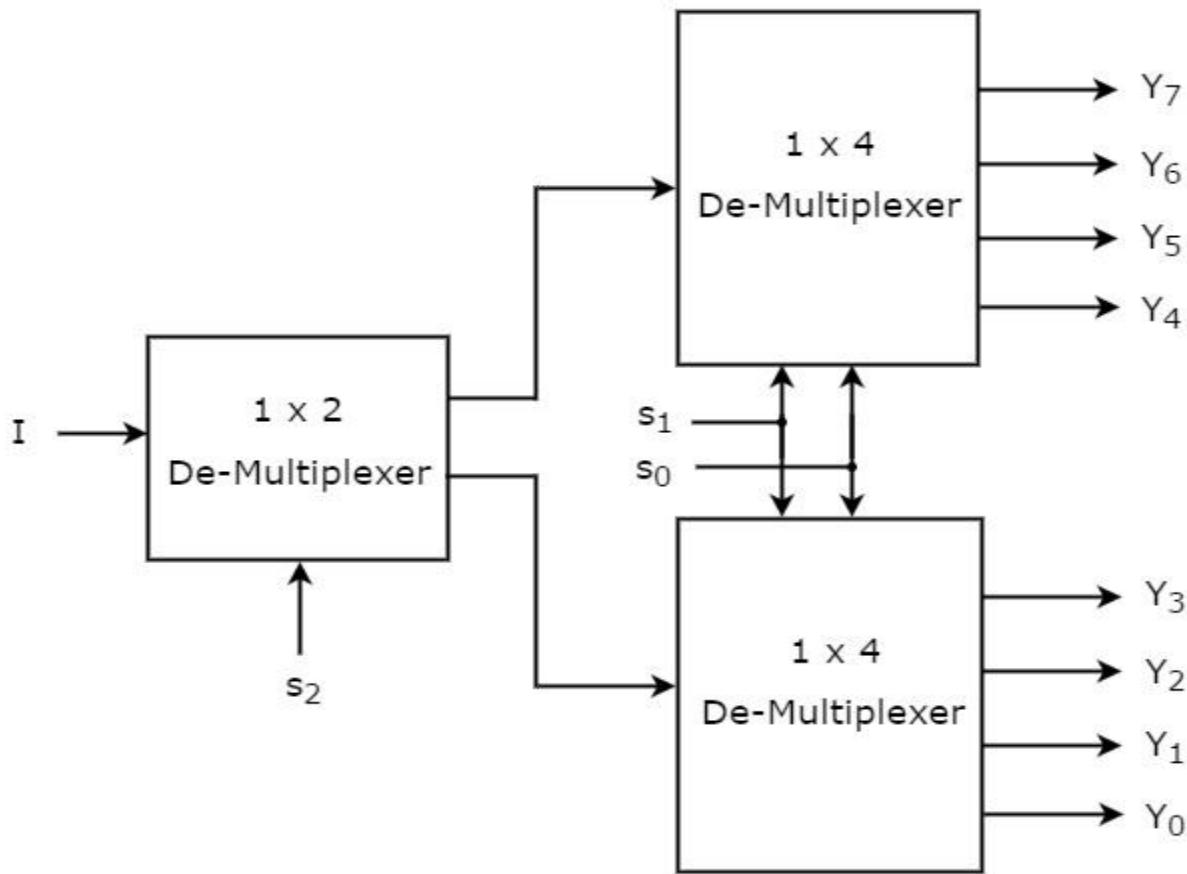
So, we require two **1x4 De-Multiplexers** in second stage in order to get the final eight outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer will be the overall input of 1x8 De-Multiplexer.

Let the 1x8 De-Multiplexer has one input I, three selection lines s_2 , s_1 & s_0 and outputs Y_7 to Y_0 . The **Truth table** of 1x8 De-Multiplexer is shown below.

Selection Inputs			Outputs							
s_2	s_1	s_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	I
0	0	1	0	0	0	0	0	0	I	0
0	1	0	0	0	0	0	0	I	0	0
0	1	1	0	0	0	0	I	0	0	0
1	0	0	0	0	0	I	0	0	0	0
1	0	1	0	0	I	0	0	0	0	0
1	1	0	0	I	0	0	0	0	0	0

1	1	1	1	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---

We can implement 1×8 De-Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 1×8 De-Multiplexer is shown in the following figure.



The common **selection lines**, s_1 & s_0 are applied to both 1×4 De-Multiplexers. The outputs of upper 1×4 De-Multiplexer are Y_7 to Y_4 and the outputs of lower 1×4 De-Multiplexer are Y_3 to Y_0 .

The other **selection line**, s_2 is applied to 1×2 De-Multiplexer. If s_2 is zero, then one of the four outputs of lower 1×4 De-Multiplexer will be equal to input, I based on the values of selection lines s_1 & s_0 . Similarly, if s_2 is one, then one of the four outputs of upper 1×4 DeMultiplexer will be equal to input, I based on the values of selection lines s_1 & s_0 .

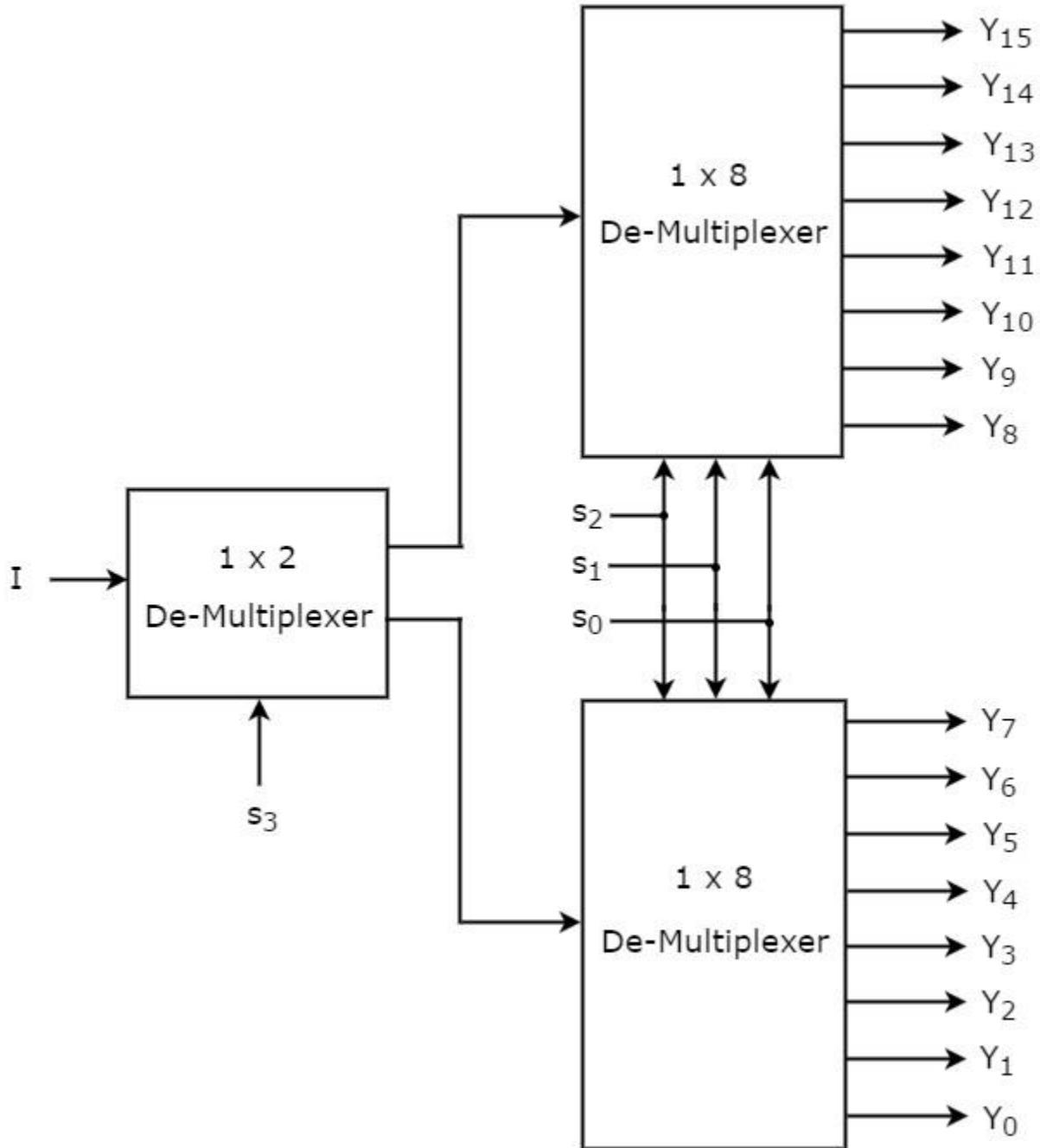
1x16 De-Multiplexer

In this section, let us implement 1×16 De-Multiplexer using 1×8 De-Multiplexers and 1×2 De-Multiplexer. We know that 1×8 De-Multiplexer has single input, three selection lines and eight outputs. Whereas, 1×16 De-Multiplexer has single input, four selection lines and sixteen outputs.

So, we require two **1×8 De-Multiplexers** in second stage in order to get the final sixteen outputs. Since, the number of inputs in second stage is two, we require **1×2 DeMultiplexer** in first stage so that

the outputs of first stage will be the inputs of second stage. Input of this 1×2 De-Multiplexer will be the overall input of 1×16 De-Multiplexer.

Let the 1×16 De-Multiplexer has one input I, four selection lines s_3, s_2, s_1 & s_0 and outputs Y_{15} to Y_0 . The **block diagram** of 1×16 De-Multiplexer using lower order Multiplexers is shown in the following figure.



The common **selection lines** s_2, s_1 & s_0 are applied to both 1×8 De-Multiplexers. The outputs of upper 1×8 De-Multiplexer are Y_{15} to Y_8 and the outputs of lower 1×8 DeMultiplexer are Y_7 to Y_0 .

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

MODEL PAPER

ANALOG & DIGITAL ELECTRONICS

(Common to EEE & ECE)

Roll No

Time: 3 hours

Max. Marks: 70

Note: This question paper Consists of 5 Sections. Answer **FIVE** Questions, Choosing ONE Question from each SECTION and each Question carries 14 marks.

SECTION-I

1 a. Explain the effect of temperature on V-I characteristics of a diode. [7]

b. Explain the V-I characteristics of Zener diode ? [7]

OR

2. a) Draw the equivalent circuits of diode [7]

b) How a PN junction diode works? Draw and explain V-I characteristics of PN diode with neat diagram [7]

SECTION-II

3.(a) Explain different current components in a transistor. [7]

(b) Calculate the values of I_E , α and β for a transistor with $I_B=13\mu A$, $I_C=200mA$, $I_{CBO}=6\mu A$ [7]

OR

4.(a) Draw the circuit diagram of a transistor in CB configuration and explain the output Characteristics with the help of different regions. [14]

SECTION-III

6 With the help of neat sketches and characteristic curves explain the construction & operation of a JFET and mark the regions of operation on the characteristics [14]

OR

7. Explain the construction and principle of operation of Depletion type N-Channel MOSFET [14]

SECTION-IV

8. Find the complement of the following Boolean functions and reduce them to minimum number of literals. a) $(b c' + a' d)(ab' + cd')$ [7]

b) $(b' d + a' b c' + a c d + a' b c)$ [7]

OR

9. (a) Convert the given expression in standard SOP form $f(A,B,C)=AC+BA+BC$ [7]

(b) Convert the given expression in standard POS form $Y=A.(A+B+C)$ [7]

SECTION-V

9.(a) Design full-adder using half adders. [7]

(b) Realize full adder using two half adders and logic gates [7]

OR

10. Simplify the following Boolean functions, using Karnaugh maps:

i. $F(w, x, y, z) = \sum m(11, 12, 13, 14, 15)$ [7]

ii. $F(A, B, C, D) = \sum m(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$ [7]

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

MODEL PAPER

ANALOG & DIGITAL ELECTRONICS

(Common to EEE & ECE)

Roll No

Time: 3 hours

Max. Marks: 70

Note: This question paper Consists of 5 Sections. Answer **FIVE** Questions, Choosing ONE Question from each SECTION and each Question carries 14 marks.

SECTION-I

1(a). Find the value of D.C. resistance and A.C resistance of a Ge junction diode at 25°C with reverse saturation current, $I_o = 25\mu A$ and at an applied voltage of 0.2V across the diode. [7]

(b) Explain about Zener Diode characteristics [7]

OR

2. Explain the operation of PN Junction Diode with neat diagrams? [14]

SECTION-II

3.(a) Draw the circuit diagram of a transistor in CE configuration and explain the output characteristics with the help of different regions. [7]

(b) In a germanium transistor collector current is 51mA, when base current is 0.04mA. If $h_{fe} = \beta_{dc} = 51$, Calculate cut off current, I_{CEO} . [7]

OR

4. (a) Draw the circuit diagram of a transistor in CB configuration and explain the output characteristics with the help of different regions. [7]

(b) Compare CB, CC, and CE configurations. [7]

SECTION-III

5. (a) Compare Depletion MOSFET and enhancement MOSFET

(b) Explain principle of operation JFET and draw the VI Characteristics

OR

6. (a) Compare JFET and MOSFET [7]

(b) Explain how FET act as voltage variable resistor [7]

SECTION-IV

7. (a) Express the following numbers in decimal: [7]

(i) $(26.24)_8$ (ii) $(16.5)_{16}$

(b) Convert the following number to Hexadecimal: [7]

i) $(735.5)_8$ ii) $(1011011)_2$

OR

8.(a) Draw the multiple-level NOR circuit for the following expressions: [7]

$$CD(B + C)A + (BC' + DE')$$

(b) Simplify and implement the following function with two-level NAND gate circuit: [7]

$$F(A, B, C, D) = A'B'C'D + CD + AC'D$$

SECTION-V

9.(a) Define decoder. Construct 3x8 decoder using logic gates and truth table. [7]

(b) Define an encoder. Design octal to binary encoder. [7]

OR

10 Simplify the following Boolean functions, using Karnaugh maps:

i. $F(x, y, z) = \sum(2, 3, 6, 7)$ ii. $F(A, B, C, D) = \sum(2, 3, 6, 7, 12, 13, 14)$ [14]

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

MODEL PAPER

ANALOG & DIGITAL ELECTRONICS

(Common to EEE & ECE)

Roll No											
---------	--	--	--	--	--	--	--	--	--	--	--

Time: 3 hours

Max. Marks: 70

Note: This question paper Consists of 5 Sections. Answer **FIVE** Questions, Choosing ONE Question from each SECTION and each Question carries 14 marks.

SECTION-I

1. Draw the basic structure of PN Junction Diode and explain its operation and V-I Characteristics [14]

OR

2.(a) Explain the V-I characteristics of Zener diode ? [7]

(b) Distinguish between Avalanche and Zener Break downs. [7]

SECTION-II

3. (a) Explain the input and output characteristics of a transistor in CC configuration [7]
(b) Calculate the collector current and emitter current for a transistor with $\alpha_{D.C.} = 0.99$ and $I_{CBO} = 50 \mu A$ when the base current is $20\mu A$ [7]

OR

4. (a) Summarize the salient features of the characteristics of BJT operatives in CE, CB and CC configurations. [7]

(b) Calculate the collector current and emitter current for a transistor with $\alpha_{D.C.} = 0.99$ and $I_{CBO} = 20 \mu A$ when the base current is $50\mu A$. [7]

SECTION-III

5. (a) Explain the construction and principle of operation of Enhancement mode N-channel MOSFET. [7]
(b) Compare BJT & FET [7]

OR

6. (a) Explain V-I characteristics of JFET [7]
(b) Explain how FET act as voltage variable resistor [7]

SECTION-IV

7. (a) Reduce the following Boolean function to four literals and draw the logic diagram:
$$(A'+C)(A'+C')(A+B+C'D)$$
 [7]

(b) Convert the following numbers to Octal:
(i) $(1010.1010)_2$ (ii) $(FAFA)_{16}$ [7]

OR

8. 8(a) Express the following numbers in decimal: [7]
(i) $(13.54)_8$ (ii) $(32.52)_{16}$

(b) Convert the following number to Hexadecimal: [7]
i) $(646.65)_8$ ii) $(101010100011011)_2$

SECTION-V

9. (a) Reduce the following function using k-map technique $F(A,B,C,D)=\pi(0,2,3,8,9,12,13,15)$ [7]
(b) Implement a full adder using 8×1 multiplexer. [7]

OR

10.(a) Design a 1:8 demultiplexer using two 1:4 demultiplexer. [7]
(b) Construct a 5-to-32-line decoder with four 3-to-8-line decoders with enable and a 2-to- 4 line decoder. Use block diagrams for the components. [7]

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

MODEL PAPER

ANALOG & DIGITAL ELECTRONICS

(Common to EEE & ECE)

Roll No										
---------	--	--	--	--	--	--	--	--	--	--

Time: 3 hours

Max. Marks: 70

Note: This question paper Consists of 5 Sections. Answer **FIVE** Questions, Choosing ONE Question from each SECTION and each Question carries 14 marks.

SECTION-I

- 1(a) Explain semi-conductors, insulators and metals classification using energy band diagrams. [7]
(b) Explain in detail the break down mechanisms in a diode. [7]

OR

- 2 (a) Explain the working of pn diode in forward and reverse bias conditions [7]
(b) Draw and explain VI characteristics of Si & Ge diode. [7]

SECTION-II

3. (a) Draw the circuit diagram of a transistor in CB configuration and explain the output characteristics with the help of different regions. [7]
(b) Explain the working of a PNP transistor with a neat diagram [7]

OR

- 4 (a) Explain the working of a NPN transistor. [7]
(b) Derive an expression between transistor parameters (α, β, γ)? [7]

SECTION-III

5. Explain the construction and principle of operation of Depletion type N-Channel MOSFET [14]
OR

6. With the help of neat sketches and characteristic curves explain the construction & operation of a JFET and mark the regions of operation on the characteristics [14]

SECTION-IV

- 7.(a) Obtain the 1's and 2's complements of the following binary numbers: [7]
(i) 00010000 (ii) 00000000 (iii) 11011010 (iv) 10101010
(v) 10000101 (vi) 11111111 [7]
- (b) Implement the following Boolean function with Logic gates: $F=(AB'+D')E+C(A'+B')$
OR
8. (a) Implement all logic gates using NAND gates. [4]
(b) Write the following Boolean expression in product of sums form: $a'b + a'c' + abc$ [4]
(c) Find the dual and complement of the following function: $A'BD'+B'(C'+D')+A'C$ [6]

SECTION-V

10. (a) Simplify the following Boolean function with the don't conditions d using Kmap method:
 $F(A, B, C, D)=\Sigma(1,3,8,10,15); d(A, B, C, D)=\Sigma(0, 2, 9)$ [7+7]
- (b) Implement the following Boolean function with only two input NOR gates: $F=(AB'+CD')E+BC(A+B)$
OR
- 10.(a) Define decoder. Construct 3x8 decoder using logic gates and truth table. [7]
(b) Define an encoder. Design octal to binary encoder. [7]