

University of Alabama in Huntsville

LOUIS

Open Educational Resources (OER)

2024

Introduction to Computer Architecture and Operating Systems

Kevin Preston

Follow this and additional works at: <https://louis.uah.edu/oer>



Part of the [Systems Architecture Commons](#)



Introduction to Computer Architecture and Operating Systems



Computer architecture is a set of rules and methods that describe the functionality, organization, and implementation of computer systems. The architecture of a system refers to its structure in terms of separately specified components of that system and their interrelationships.

In a similar manner to other uses of the word architecture, computer architecture is focused on determining the needs of the user/system/technology and creating a logical design and standards based on those requirements.

The goals for this course include:

- To learn how to write advanced ARM Assembly Language programs for the Raspberry Pi and the relationship of these instructions to the hardware that implements them.
- To learn how to calculate the different performance metrics with CPUs so that different CPUs performance can be compared.
- To learn how pipelines work and calculate the effects on CPU performance associated with branches.

- To learn the different cache organizations and calculate the performance of cache and effects on overall CPU performance.
- To learn the different Input/Output strategies used for modern computers and the advantages and disadvantages of each.
- To learn the organization of modern computer main memory.
- To learn security defense mechanism implemented by CPUs.

The content for this textbook was created by Kenvin Preston, instructor at The University of Alabama in Huntsville
2023

This textbook is licensed under a Creative Commons Attribution and Share Alike 4.0 International License, which allows users to freely share, copy, distribute, and adapt the work, provided proper attribution is given to the author(s). If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find more information about the license at: [Attribution-ShareAlike 4.0 International](https://creativecommons.org/licenses/by-sa/4.0/)



Introduction to the Hardware Topics



Basic Gates



Basic Gates - Combinational Logic



Sequential Logic



Addition, Subtraction, Multiplication & Division



Floating Point Numbers



Combinational & Sequential Logic



ARM Tutorial



Register Transfer Language & ARM Addressing Modes



ARM Programming



The Pipeline



Cache



Main and Virtual Memory



Input/Output



Operating System: Introduction



OS Time Management



OS Space Management

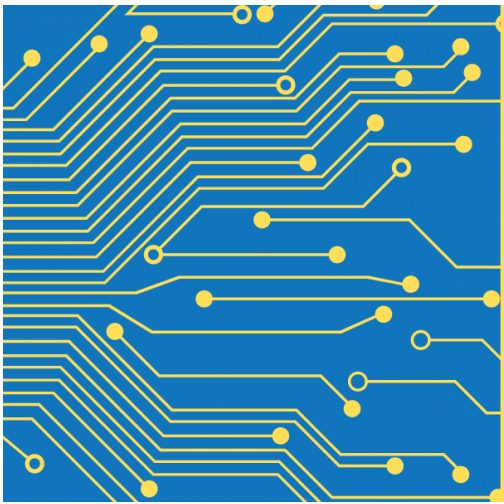


OS Resource Deadlock



OS Concurrency

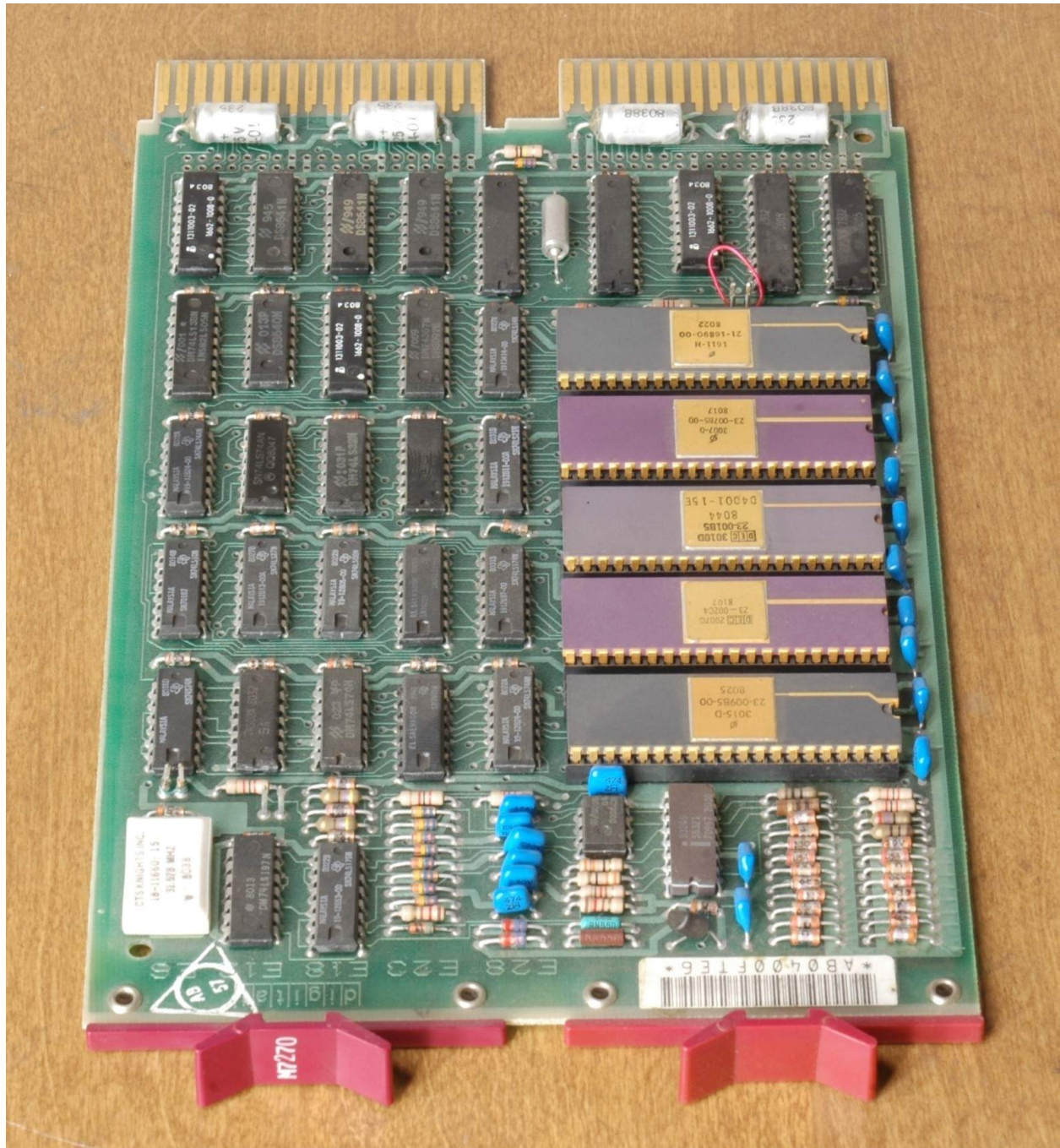
Introduction to the Hardware Topics



Introduction to Hardware Topics

By contrast, software is the set of instructions that can be stored and run by hardware. Hardware is so-termed because it is "hard" or rigid with respect to changes, whereas software is "soft" because it is easy to change.

Hardware is typically directed by the software to execute any command or instruction. A combination of hardware and software forms a usable computing system, although other systems exist with only hardware.



PDP-11 CPU Board

Types of Computer systems

Click on each of the types below to expand for a description:

Personal Computer —

The personal computer is one of the most common types of computer due to its versatility and relatively low price. Desktop personal computers have a monitor, a keyboard, a mouse, and a computer case. The computer case holds the motherboard, fixed or removable disk drives for data storage, and the power supply, and may contain other peripheral devices such as modems or network interfaces. Some models of desktop computers integrated the monitor and keyboard into the same case as the processor and power supply. Separating the elements allows the user to arrange the components in a pleasing, comfortable array, at the cost of managing power and data cables between them.



The Laptop —

Laptops are designed for portability but operate similarly to desktop PCs. They may use lower-power or reduced size components, with lower performance than a similarly priced desktop computer. Laptops

contain the keyboard, display, and processor in one case. The monitor in the folding upper cover of the case can be closed for transportation, to protect the screen and keyboard. Instead of a mouse, laptops may have a trackpad or pointing stick.



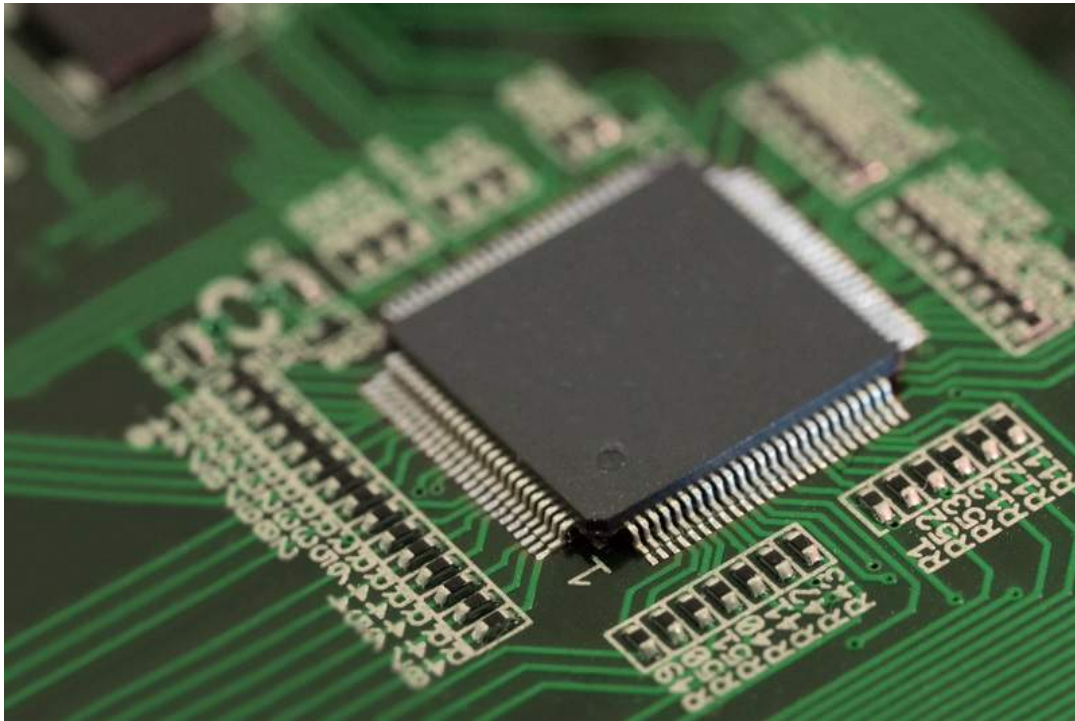
Tablets/Smartphones

Tablets and Smartphones are portable computers that use a touchscreen as the primary input device. Tablets generally weigh less and are smaller than laptops. Some tablets include fold out keyboards, or offer connections to separate external keyboards. Neither of these devices can be used for software development but solely depend on downloaded "Applications" for their function.



IOT and Embedded Processors

The Internet of Things (IOT) and embedded processors are items like microwaves, refrigerators, automobiles, gaming systems, security cameras, missile systems, toys. Almost anything you can think of does or will have a CPU also known as microprocessor in these particular applications.



Components of a Computer

Click the hotspots on the image below for a basic explanation of the parts of a computer:





Motherboard

The motherboard is the main component of a computer. It is a board with integrated circuitry that connects the other parts of the computer including the CPU, the RAM, the disk drives (CD, DVD, hard disk, or any others) as well as any peripherals connected via the ports or the expansion slots. The integrated circuit (IC) chips in a computer typically contain billions of tiny metal–oxide–semiconductor field–effect transistors (MOSFETs).

***See the list below this interaction for more information about the components of the motherboard.**



Power Supply

A power supply unit (PSU) converts alternating current (AC) electric power to low-voltage direct current (DC) power for the computer. Laptops can run on a built-in rechargeable battery. The PSU typically uses a switched-mode power supply (SMPS), with power MOSFETs (power metal–oxide–semiconductor field–effect transistors) used in the converters and regulator circuits of the SMPS.



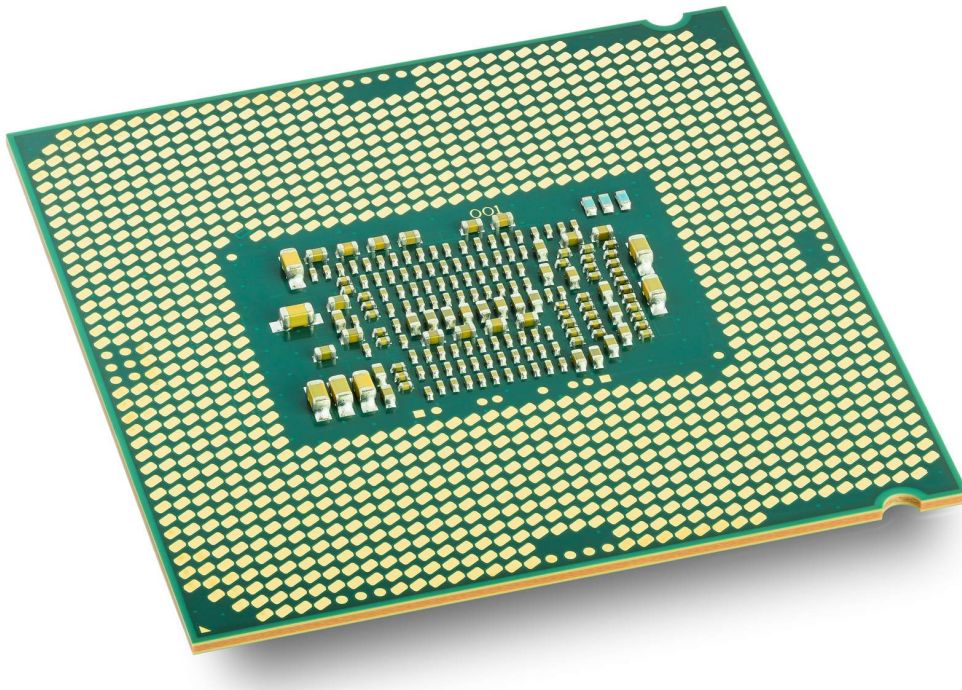
Case

The computer case encloses most of the components of the system. It provides mechanical support and protection for internal elements such as the motherboard, disk drives, and power supplies, and controls and directs the flow of cooling air over internal components. The case is also part of the system to control electromagnetic interference radiated by the computer and protects internal parts from electrostatic discharge. Large tower cases provide space for multiple disk drives or other peripherals and usually stand on the floor, while desktop cases provide less expansion room. All-in-one style designs include a video display built into the same case. Portable and laptop computers require cases that provide impact protection for the unit. A current development in laptop computers is a detachable keyboard, which allows the system to be configured as a touch-screen tablet. Hobbyists may decorate the cases with colored lights, paint, or other features, in an activity called case modding.

Components of the motherboard:

Step 1

The CPU

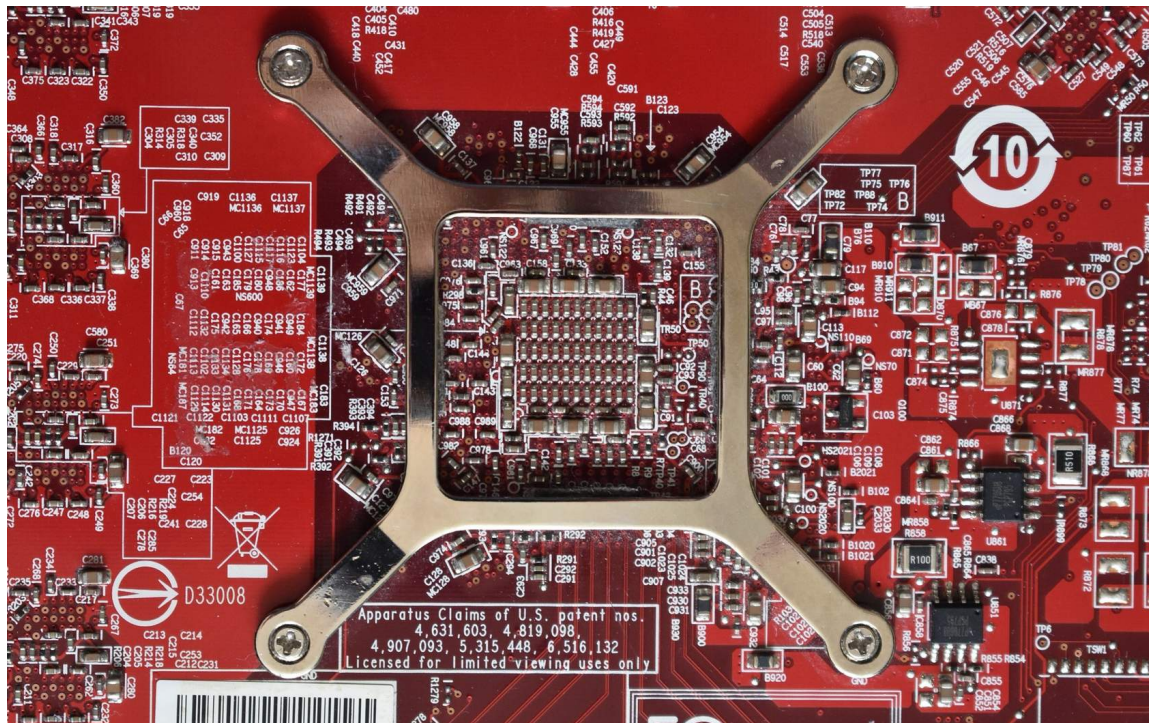


The CPU (central processing unit), which performs most of the calculations which enable a computer to function is commonly referred to as the "brain" of the computer. The CPU retrieves a program instruction from random-access memory (RAM), interprets and processes it and then sends back the result so that the relevant components can carry out the instructions.. The CPU is a microprocessor, which is fabricated on a metal-oxide-semiconductor (MOS) integrated circuit (IC) chip. It is usually cooled by a heat sink and fan, or water-cooling system. Most newer CPUs include an on-die graphics processing unit (GPU). The clock speed of a CPU governs how fast it executes instructions and is measured in GHz; typical values lie between 1 GHz and 5 GHz. Many modern computers have the option to overclock the CPU which enhances performance at the expense of greater thermal output and thus a need for improved cooling. (Details on clocks are provided later in this module).Image from Wikipedia Commons:

https://commons.wikimedia.org/wiki/File:Intel_CPU_Core_i7_6700K_Skylake_perspective.jpg

Step 2

The chipset



The chipset, which includes the north bridge, mediates communication between the CPU and the other components of the system, including main memory; as well as south bridge, which is connected to the north bridge, and supports auxiliary interfaces and buses; and, finally, a Super I/O chip, connected through the south bridge, which supports the slowest and most legacy components like serial ports, hardware monitoring and fan control.

Image from Pixabay: <https://pixabay.com/photos/hardware-graphic-card-chipset-3499505/>

Step 3

Random-access memory (RAM)



Random-access memory (RAM) stores the code and data that are being actively accessed by the CPU. For example, when a web browser is opened on the computer it takes up memory; this is stored in the RAM until the web browser is closed. It is typically a type of dynamic RAM (DRAM), such as synchronous DRAM (SDRAM), where MOS memory chips store data on memory cells consisting of MOSFETs and MOS capacitors. RAM usually comes on dual in-line memory modules (DIMMs) in the sizes of 2GB, 4GB, and 8GB, but can be much larger. The latest memory components are referred to as Double Data Rate (DDR) generations 2, 3, 4, 5, etc. or more commonly as DDR2, DDR3, DDR4, DDR5, etc.

Image from Wikipedia Commons:

https://commons.wikimedia.org/wiki/File:Swissbit_2GB_PC2-5300U-555.jpg

Step 4

Read-only memory (ROM)



Read-only memory (ROM), which stores the BIOS that runs when the computer is powered on or otherwise begins execution, a process known as Bootstrapping, or "booting" or "booting up". The ROM is typically a nonvolatile BIOS memory chip, which stores data on floating-gate MOSFET memory cells. (Image from: <https://www.explainthatstuff.com/how-computer-memory-works.html>)

- The BIOS (Basic Input Output System) includes boot firmware and power management firmware. Newer motherboards use Unified Extensible Firmware Interface (UEFI) instead of BIOS.

Step 5

The Buses

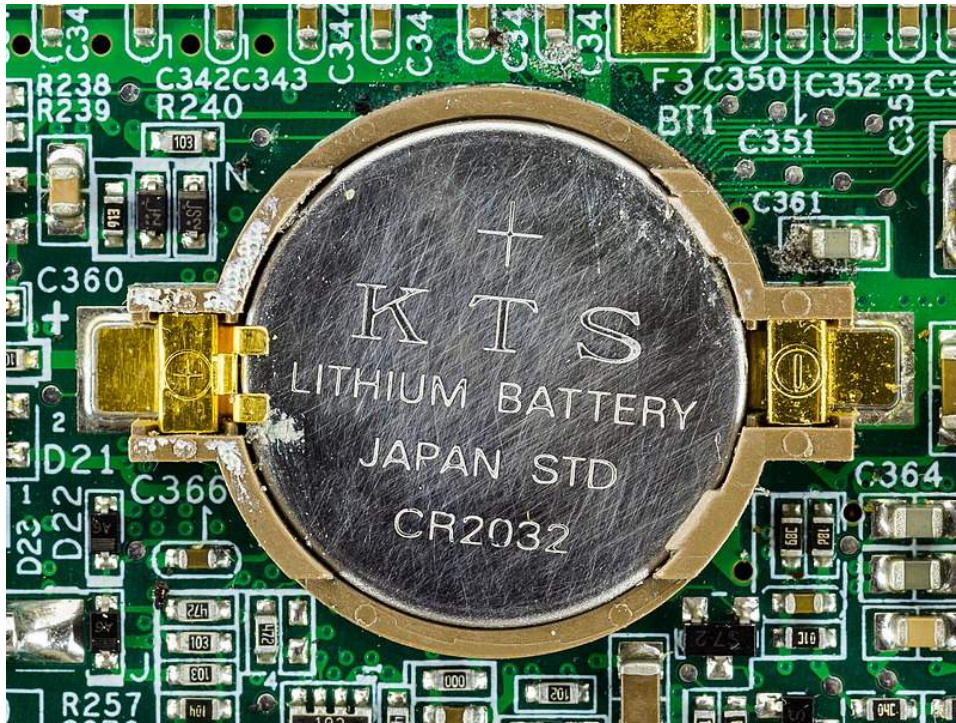


Buses connect the CPU to various internal components and to expand cards for graphics and sound.

Image from Wikimedia Commons: [https://commons.wikimedia.org/wiki/File:PC-FXGA\(C-BUS\)_01.jpg](https://commons.wikimedia.org/wiki/File:PC-FXGA(C-BUS)_01.jpg)

Step 6

The CMOS



The CMOS (complementary MOS) battery, which powers the CMOS memory for date and time in the BIOS chip. This battery is generally a watch battery.

Image from Wikimedia Commons:

https://commons.wikimedia.org/wiki/File:Yakumo_Notebook_536S_-_CR2032_backup_battery_on_motherboard-4667.jpg

Step 7

The video care/graphics card

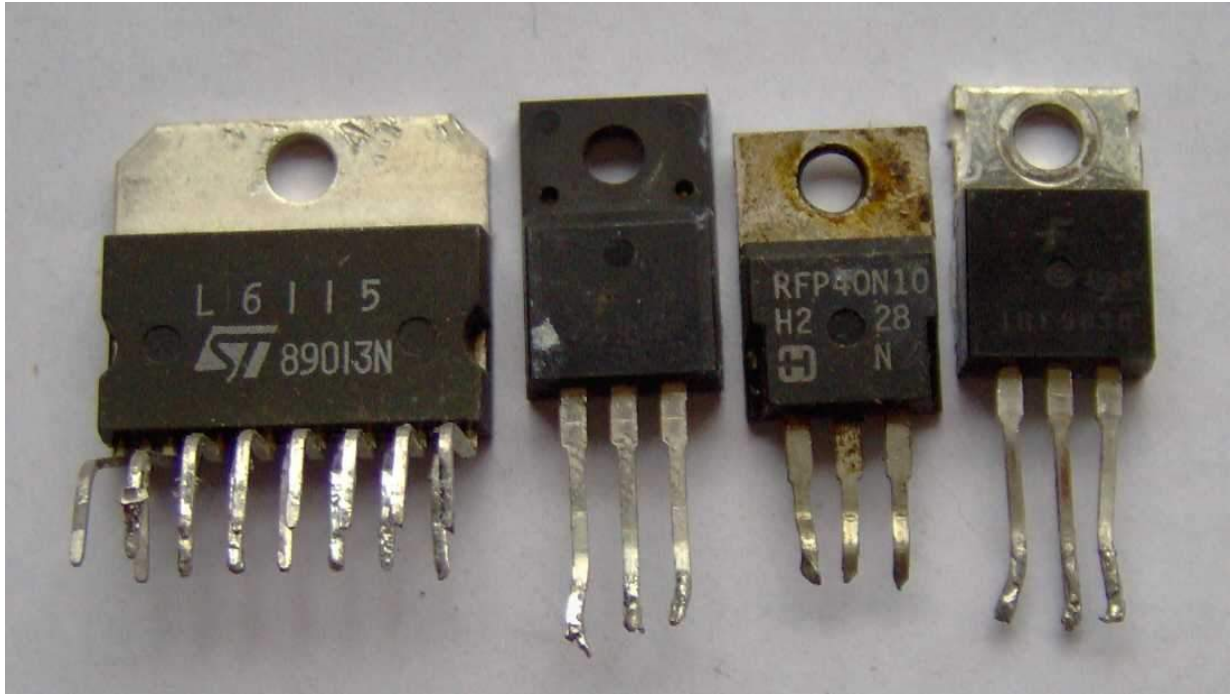


The video card (also known as the graphics card) processes computer graphics. More powerful graphics cards are better suited to handle strenuous tasks, such as playing intensive video games or running computer graphics software. A video card contains a graphics processing unit (GPU) and video memory (typically a type of SDRAM), both fabricated on MOS integrated circuit (MOS IC) chips.

Image from Wikipedia: https://en.wikipedia.org/wiki/Nvidia_RTX

Step 8

The power MOSFETs



Power MOSFETs make up the voltage regulator module (VRM), which controls how much voltage other hardware components receive.

Image from Wikimedia Commons:

https://commons.wikimedia.org/wiki/File:Electronic_component_mosfets.jpg

Additional components:

- Expansion cards: An expansion card in computing is a printed circuit board that can be inserted into

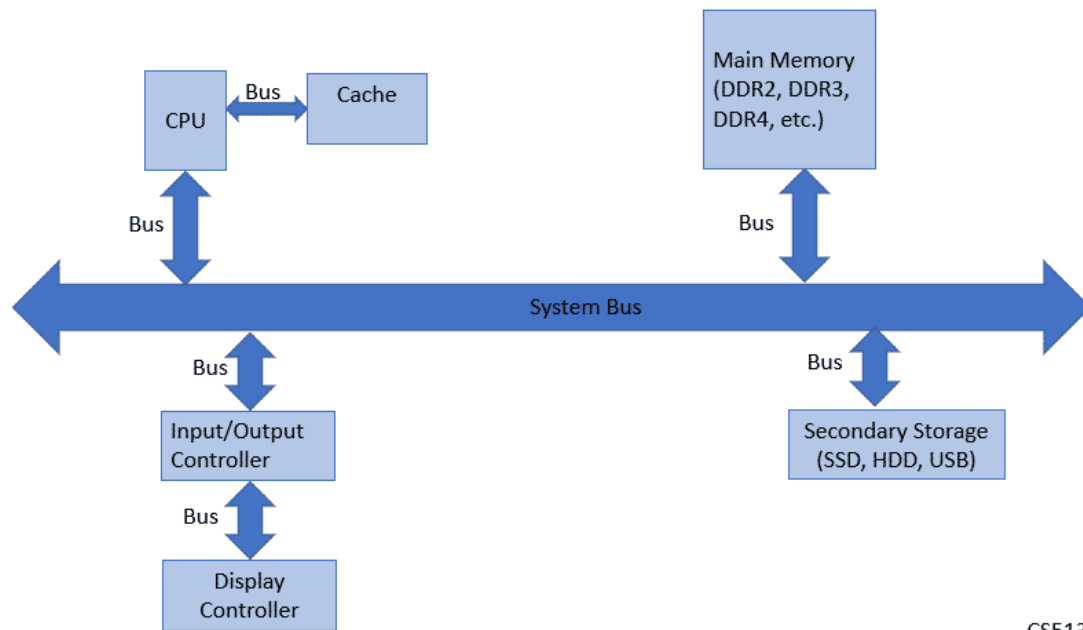
an expansion slot of a computer motherboard or backplane to add functionality to a computer system via the expansion bus. Expansion cards can be used to obtain or expand on features not offered by the motherboard.

- **Storage devices:** A storage device is any computing hardware and digital media that is used for storing, porting, and extracting data files and objects. It can hold and store information both temporarily and permanently and can be internal or external to a computer, server, or any similar computing device. Data storage is a core function and fundamental component of computers.
- **Fixed media:** Data is stored by a computer using a variety of media. Hard disk drives (HDDs) are found in virtually all older computers, due to their high capacity and low cost, but solid-state drives (SSDs) are faster and more power-efficient, although currently more expensive than hard drives in terms of dollar per gigabyte, so are often found in personal computers built post-2007. SSDs use flash memory, which stores data on MOS memory chips consisting of floating-gate MOSFET memory cells. Some systems may use a disk array controller for greater performance or reliability.
- **Removable media:** To transfer data between computers, an external flash memory device (such as a memory card or USB flash drive) or

optical disc (such as a CD-ROM, DVD-ROM, or BD-ROM) may be used. Their usefulness depends on being readable by other systems; the majority of machines have an optical disk drive (ODD), and virtually all have at least one Universal Serial Bus (USB) port.

- **Input and output peripherals:** Input and output devices are typically housed externally to the main computer chassis. The following are either standard or very common to many computer systems.
- **Input device:** Input devices allow the user to enter information into the system, or control its operation. Most personal computers have a mouse and keyboard, but laptop systems typically use a touchpad instead of a mouse. Other input devices include webcams, microphones, joysticks, and image scanners.
- **Output device:** Output devices are designed around the senses of human beings. For example, monitors display text that can be read, speakers produce sound that can be heard. Such devices also could include printers or a Braille embosser.
- **One more term – Bus.** The definition of a Bus can get very complicated but for this course just consider a bus to be a common connection point or to make it simpler a Bus is a collection of wires.

The following figure provides an overall block diagram of how these various components of the computer are connected together.



CS513 Module 1

Diagram illustrating how various components are connected

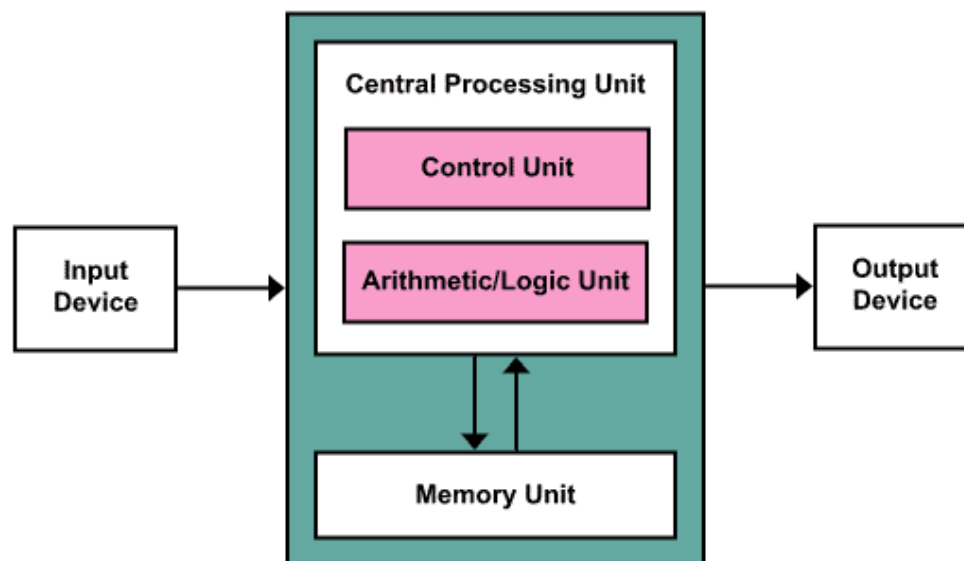


Diagram of Von Neumann Architecture

Von Neumann architecture

Drilling down a level of detail we see that the CPU is composed of two major components. The Control Unit and the Arithmetic/Logic Unit (ALU). This organization is officially known as the Stored Program Digital Computer or more commonly as the Von Neumann Architecture.

Von Neumann architecture scheme

The template for all modern computers is the Von Neumann architecture, detailed in a 1945 paper by Hungarian mathematician John von Neumann. This describes a design architecture for an electronic digital computer with subdivisions of a processing unit consisting of an arithmetic logic unit and processor registers, a control unit containing an instruction register and program counter, a memory to store both data and instructions, external mass storage, and input and output mechanisms. The meaning of the term has evolved to mean a stored-program computer in which an instruction fetch and a data operation cannot occur at the same time because they share a common bus. This is referred to as the Von Neumann bottleneck and often limits the performance of the system.

The next several modules will introduce the basic logic gates that are used to make these various components of the CPU.

More terms:

- **Algorithm** - a finite list of well-defined instructions that carries out an operation.
- **Program** - a set of computer instructions that implements an algorithm. Has to compile, link and run.
- **Pseudocode** - falls between an algorithm and a program. Some parts compile and some will not.
- **Instruction Set Architecture (ISA)** - Assembly language level or what the programmer can see or use. Includes the data types (bits per word), the registers that the programmer has direct control over, instruction types, formatting, and addressing modes. An

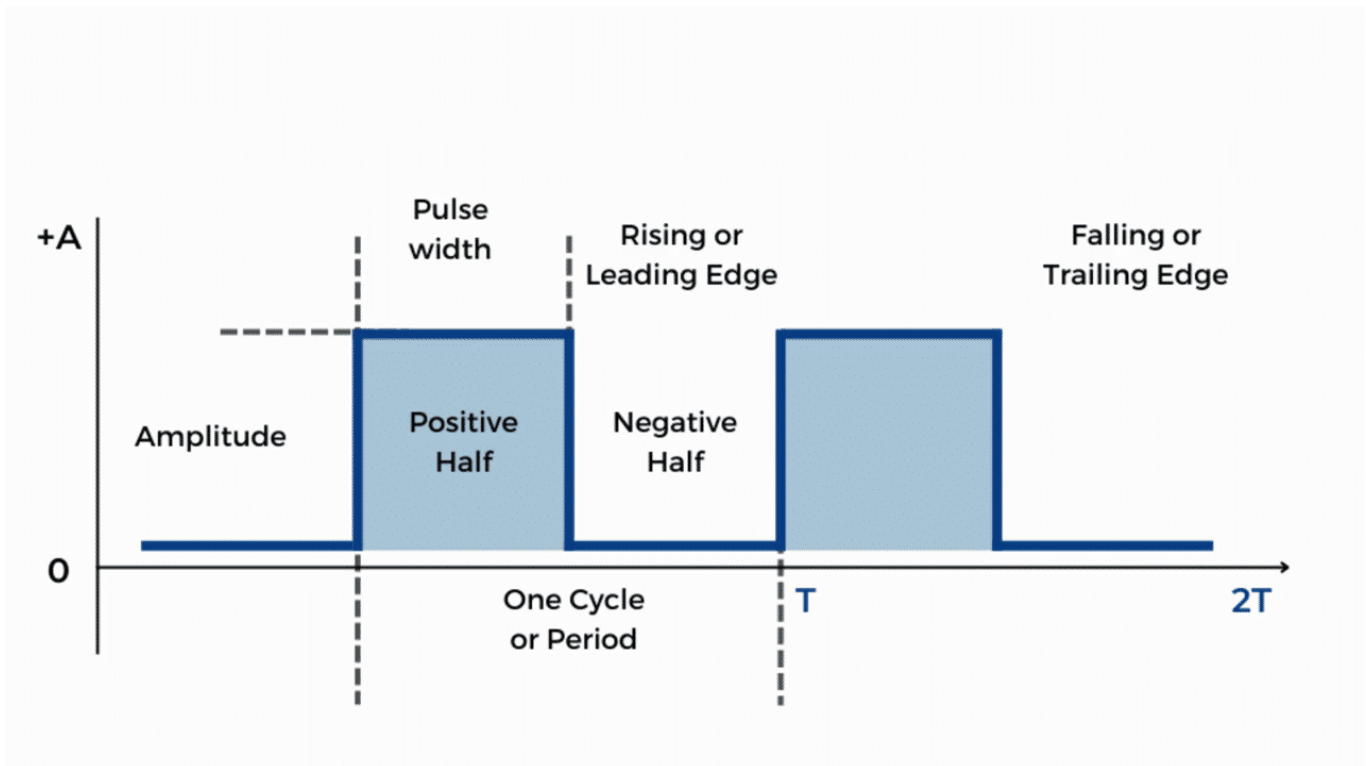
abstract model of a computer that describes what it does not how it does it.

- **Computer Organization** - the hardware used to implement the instruction set. Registers, adders, pipelines, cache, shift registers, floating point processor, etc.
- **Microarchitecture** - how the hardware is implemented on the chip. Layers, masks, buses, connections between the various hardware elements.
- **Machine Code** - this is what is run on the CPU. It is the 1's and 0's that are translated from your C++, Python, or Java program.
- **Assembly language code** - is the human readable version of the Machine Code. Example instructions are: ADD, LDA, STA, TRX, etc.
- **Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC)** - The RISC developers worked to have a smaller instruction set so that the instructions could run faster. This course will mostly deal with RISC and more specifically with the ARM. CISC worked on having more instructions and addressing modes. Examples of the CISC are the Intel Processors line like the iCore Processors.

The lines between the two have blurred modern times in that both sides have used technology from the other.

Clock Speeds

A Clock or CLK makes Logic 0 to Logic 1 transitions at regular intervals. Also called a Square wave.



CLK Clock.pdf
166.1 KB



Clock Speed Terms

- **Amplitude** - is the height of the wave. For modern digital circuits the 0 (or logic 0) is considered ground or zero Volts Direct Current (VDC). The 1 (or logic 1) can vary depending on the technology being used but in most cases ranges from 3.3 to 5 VDC.
- **Cycle or Period** - is the amount of time it takes for one wavelength which is the distance between identical points along the waveform. In this example, the cycle starts when the

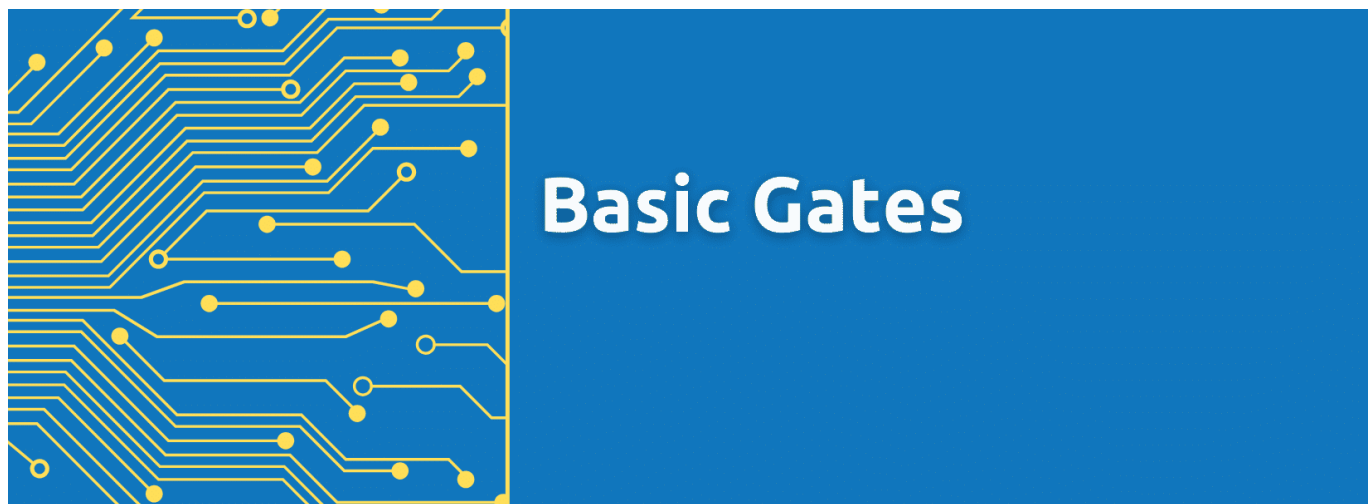
waveform starts changing from the logic 0 to the logic 1 and ends when that same transition is made.

- **Pulse Width** - is the amount of time the waveform is at a logic 1.
- **Rising Edge or Positive Edge or Leading Edge** - is when the waveform transitions from a logic 0 to a logic 1.
- **Trailing Edge or Negative Edge or Falling Edge** - is when the waveform transitions from a logic 1 to a logic 0.
- **Frequency = $1/t$** Where t is the amount of time in seconds it takes to complete one Cycle or Period. The unit of measure is Hertz aka Hz. Cycles per second. If it takes one second to complete one Cycle then the frequency of the waveform is 1 Hertz. If the cycle takes 2 nanoseconds (a billionth of a second) then the frequency is 500 Megahertz (MHz) or 0.5 Gigahertz (GHz). Most modern CPUs run in the 2 GHz to 8 GHz range.

$1/t$ gives the Frequency and $1/\text{frequency}$ gives the time for one cycle.

Given that a clock runs at 800 MHz then the time for one cycle is 1.25 nanoseconds (Calculated by $1/(800 \times 10^6)$)

Basic Gates

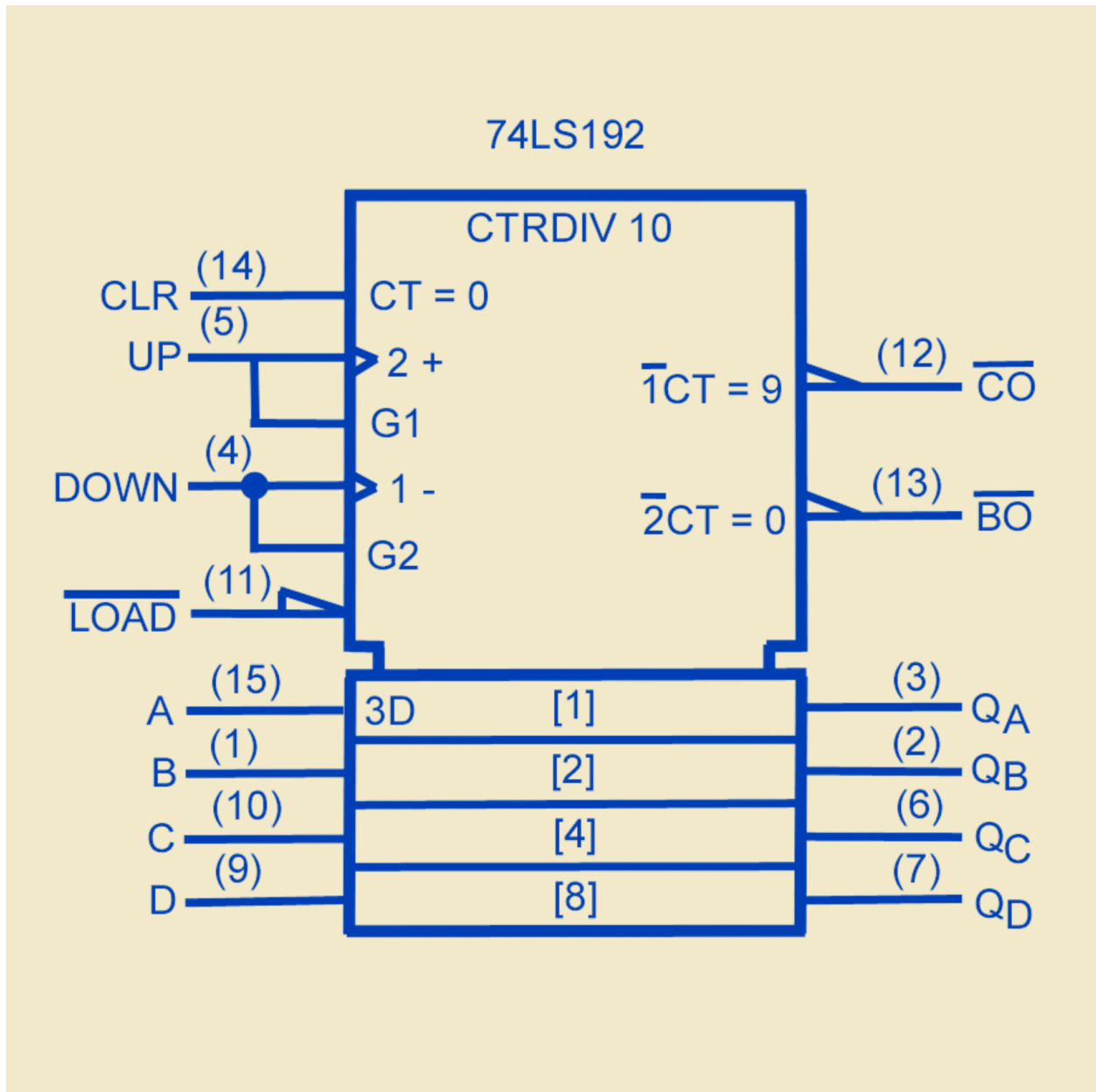


Introduction

A logic gate is an idealized or physical electronic device implementing a Boolean function, a logical operation performed on one or more binary inputs that produces a single binary output. Logic gates are primarily implemented using diodes or transistors acting as electronic switches, but can also be constructed using vacuum tubes, electromagnetic relays (relay logic), fluidic logic, pneumatic logic, optics, molecules, or even mechanical elements. With amplification, logic gates can be cascaded in the same way that Boolean functions can be composed, allowing the construction of a physical model of all of Boolean logic, and therefore, all of the algorithms and mathematics that can be described with Boolean logic.

NOTE: Before proceeding into this material students should download Logisim to their computer. Logisim is an easy to learn and use Java application to create and test digital circuits. No

install in required. Just open the executable to run it. Logisim will be used to demonstrate the circuits in this course and will be used for student lab submissions.



Objectives

From this module students should learn:

- The functions of basic logic gates

- Read the following material
- Download the Logisim Java application
- Watch the associated video lecture
- Take the Canvas quiz on this topic
- Design, implement and test a combinational logic circuit. Submit the resulting circuit to Canvas

Students should:

- Read the following material
- Download the Logisim Java application
- Watch the associated video lecture
- Take the Canvas quiz on this topic
- Design, implement and test a combinational logic circuit. Submit the resulting circuit to Canvas



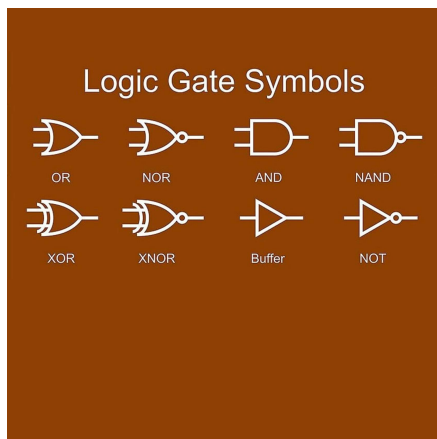
Before proceeding, please download [LOGISM-EVOLUTION](#)

Logic Gates

A logic gate is an idealized or physical electronic device implementing a Boolean function, a logical operation performed on one or more binary inputs that produces a single binary output. Depending on the context, the term may refer to an [ideal logic gate](#), one that has for instance zero rise time and unlimited fan-out, or it may refer to a non-ideal physical device (see Ideal and real op-amps for comparison).

Logic gates are primarily implemented using diodes or transistors acting as electronic switches, but can also be constructed using vacuum tubes, electromagnetic relays (relay logic), fluidic logic, pneumatic logic, optics, molecules, or even mechanical elements. With amplification, logic gates can be cascaded in the same way that Boolean functions can be composed, allowing the construction of a physical model of all of Boolean logic, and therefore, all of the algorithms and mathematics that can be described with Boolean logic.

Logic circuits include such devices as multiplexers, registers, arithmetic logic units (ALUs), and computer memory, all the way up through complete microprocessors, which may contain more than 100 million gates. In modern practice, most gates are made from MOSFETs (metal–oxide–semiconductor field–effect transistors).

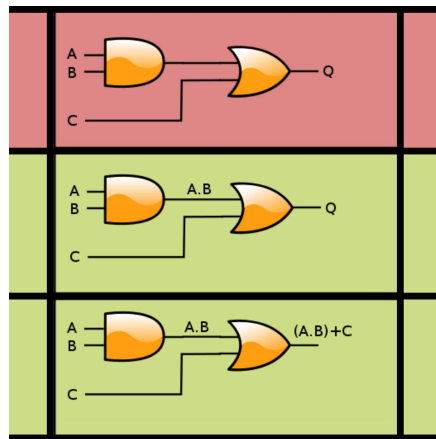


Logic Gate Symbols

Image from:

[https://commons.wikimedia.org/wiki/](https://commons.wikimedia.org/wiki/File:NV_0501_Byers_Social4.jpg)

File:NV_0501_Byers_Social4.jpg



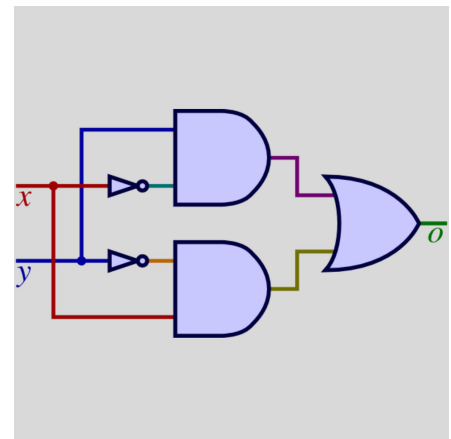
Ideal Logic Gate:

Image created on [Canva](https://www.canva.com/)

From:

[https://commons.wikimedia.org/wiki/](https://commons.wikimedia.org/wiki/File:CPT-logic-gate_example.svg)

File:CPT-logic-gate_example.svg



Logic Circuit

Image from:

[https://www.pngkey.com/png/full/92-](https://www.pngkey.com/png/full/92-920785_a-simple-logic-circuit-x-y-circuit.png)

920785_a-simple-logic-circuit-x-

y-circuit.png

Electronic Gates

A functionally complete logic system may be composed of relays, valves (vacuum tubes), or transistors. The simplest family of logic gates uses bipolar transistors, and is called resistor–transistor logic (RTL). Unlike simple diode logic gates (which do not have a gain element), RTL gates can be cascaded indefinitely to produce more complex logic functions. RTL gates were used in early integrated circuits.

For higher speed and better density, the resistors used in RTL were replaced by diodes resulting in diode–transistor logic (DTL).

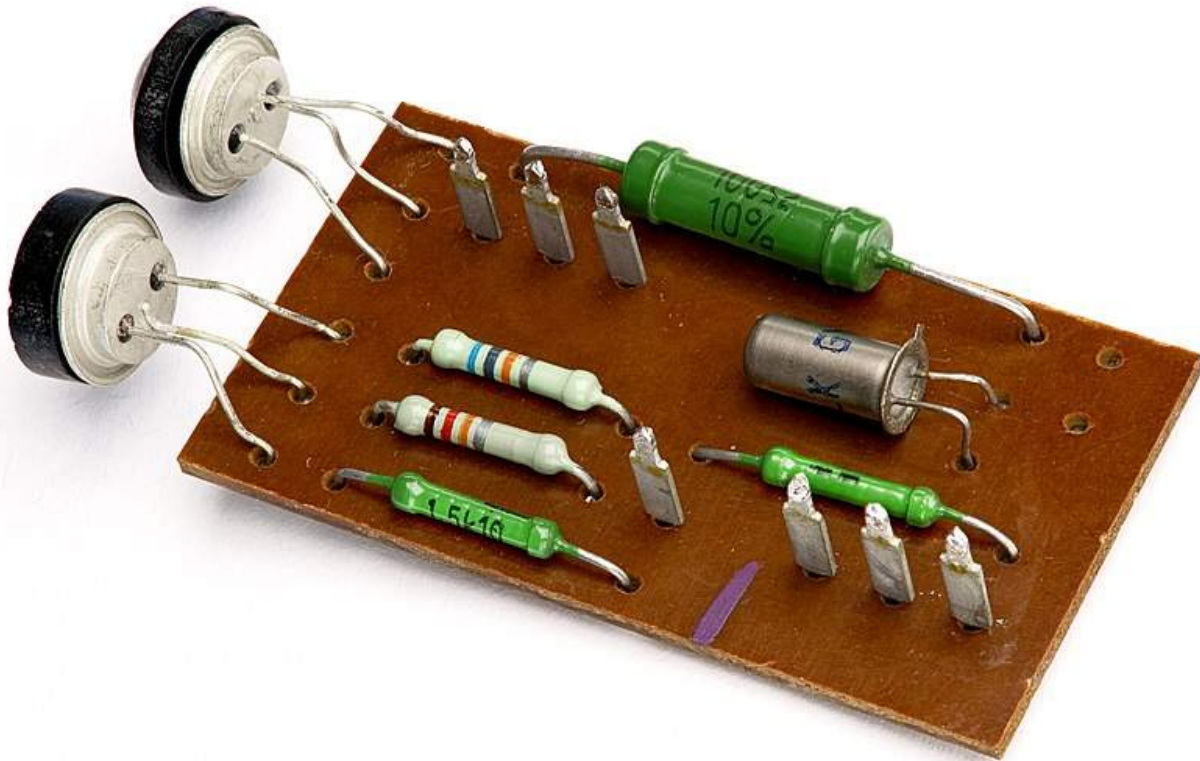


Image from: https://commons.wikimedia.org/wiki/File:PCB_4029-66_1.jpg

Transistor–transistor logic (TTL) then supplanted DTL. As integrated circuits became more complex, bipolar transistors were replaced with smaller field-effect transistors (MOSFETs); see PMOS and NMOS. To reduce power consumption still further, most contemporary chip implementations of digital systems now use CMOS logic. CMOS uses complementary (both n-channel and p-channel) MOSFET devices to achieve a high speed with low power dissipation.

For small-scale logic, designers now use prefabricated logic gates from families of devices such as the TTL 7400 series by Texas Instruments, the CMOS 4000 series by RCA, and their more recent descendants. Increasingly, these fixed-function logic gates are being replaced by programmable logic devices, which allow designers to pack many mixed logic gates into a single integrated circuit. The field-programmable nature of programmable logic devices such as FPGAs has reduced the 'hard' property of hardware; it is now possible to change the logic design of a hardware system by reprogramming some of its components, thus allowing the features or function of a hardware implementation of a logic system to be changed.



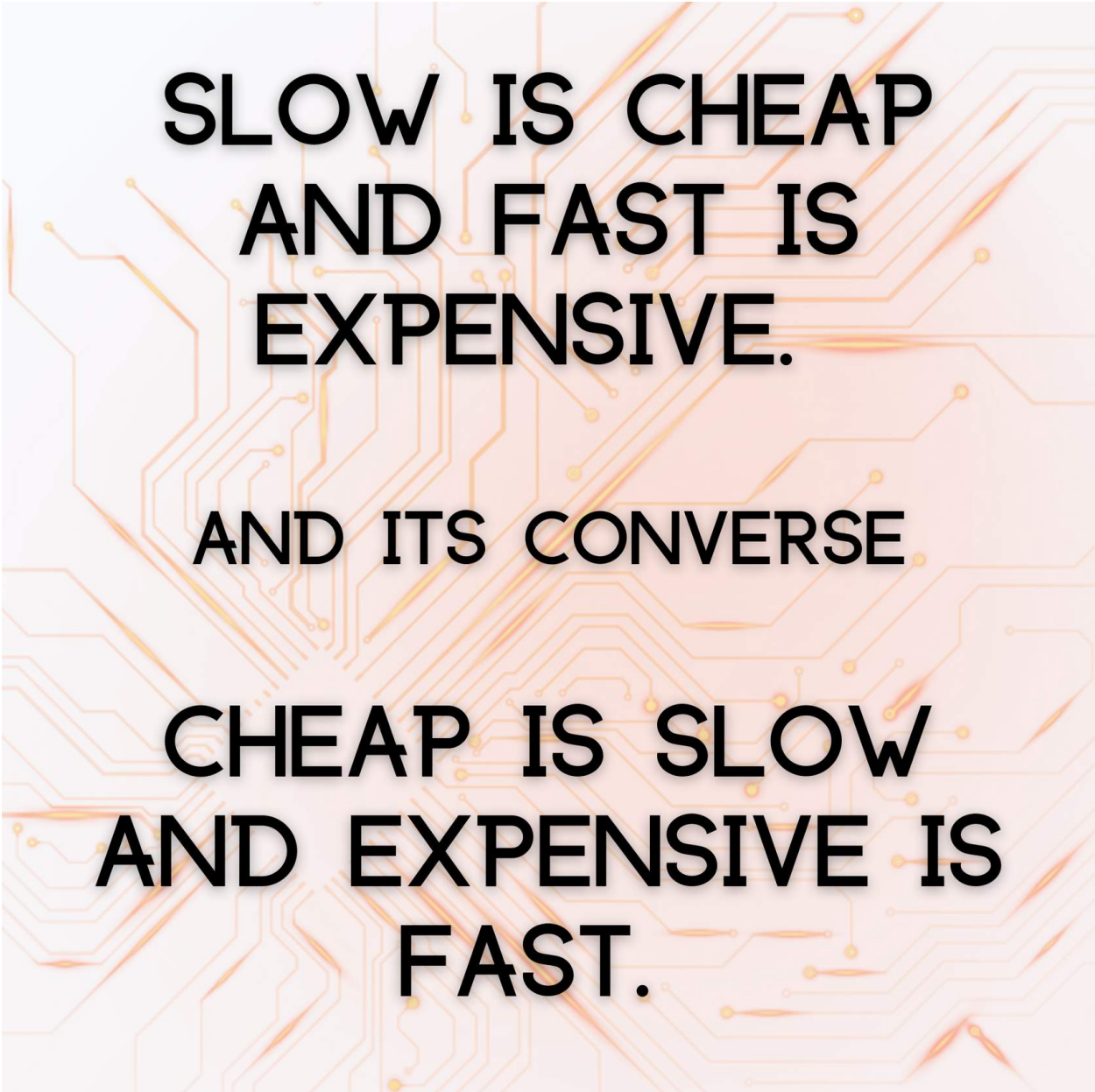
The details and design of these devices are beyond the scope of this course. They are mentioned here because the type of technology used to implement digital circuits greatly affects the speed, size and power consumption of the devices.

An advantage of standardized integrated circuit logic families, such as the 7400 and 4000 families, is that they can be cascaded. This means that the output of one gate can be wired to the inputs of one or several other gates, and so on. Systems with varying degrees of complexity can be built without great concern of the designer for the internal workings of the gates, provided the limitations of each integrated circuit are considered.

The output of one gate can only drive a finite number of inputs to other gates, a number called the 'fan-out limit'. Also, there is always a delay, called the 'propagation delay', from a change in input of a gate to the corresponding change in its output. When gates are cascaded, the total propagation delay is approximately the sum of the individual delays, an effect which can become a problem in high-speed circuits. Additional delay can be caused when many inputs are connected to an output, due to the distributed capacitance of all the inputs and wiring and the finite amount of current that each output can provide.

Propagation Delay is dependent on the technology used to implement the gate (TTL, RTL, DTL, CMOS, etc.). The delay runs as quickly as one nanosecond (one billionth of a second) to 30 or more nanoseconds. One of the important things related to computer hardware: Slow is cheap and fast is expensive – and its converse – cheap is slow and expensive is fast.

The following provides definitions of the basic logic gates that are used in modern digital circuits. Remember each of these gates consist of several transistors, resistors and capacitors. The design of these logic gates is beyond the scope of this course.



**SLOW IS CHEAP
AND FAST IS
EXPENSIVE.**

AND ITS CONVERSE

**CHEAP IS SLOW
AND EXPENSIVE IS
FAST.**

Gate Name	Symbol	Boolean Algebraic Notation (Q)	Description	Truth Table						
Buffer		A	The output is always that same as the input.	<table><tr><td>A</td><td>Q</td></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	A	Q	0	0	1	1
A	Q									
0	0									
1	1									
NOT or Invertor		A' or $\neg A$	The output is always the inverse or complement of the input.	<table><tr><td>A</td><td>Q</td></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Q	0	1	1	0
A	Q									
0	1									
1	0									

Definitions of Basic Logic Gates

Image created using Canva. PDF version available below



Definitions of Basic Logic Gates.pdf

48.5 KB



In electronics, a NOT gate is more commonly called an inverter. The circle on the symbol is called a **bubble** and is used in logic diagrams to indicate a logic negation between the external logic state and the internal logic state (1 to 0 or vice versa). On a circuit diagram, it must be accompanied by a statement asserting that the **positive logic convention** or

negative logic convention is being used (high voltage level = 1 or low voltage level = 1, respectively). For the purpose of this course, we will use the positive logic convention.

Gate Name	Symbol	Boolean Algebraic Notation	Description	Truth Table															
AND		$A * B$ $A \wedge B$ $AB \leftarrow$ Most common	The output is only a 1 when all the inputs are 1. If any one of the inputs is a 0 it does not matter what the other inputs are the output will be 0. This is referred to as the gate is disabled. You can think of this as regular multiplication	<table><tr><td>A</td><td>B</td><td>Q</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Q	0	0	0	0	1	0	1	0	0	1	1	1
A	B	Q																	
0	0	0																	
0	1	0																	
1	0	0																	
1	1	1																	
OR		$A + B$ $A \vee B$	The output is only a 0 when all the inputs are 0. If any one of the inputs is a 1 it does not matter what the other inputs are the output will be 1. This is referred to as the gate disabled. You can think of this as kinda like regular addition.	<table><tr><td>A</td><td>B</td><td>Q</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Q	0	0	0	0	1	1	1	0	1	1	1	1
A	B	Q																	
0	0	0																	
0	1	1																	
1	0	1																	
1	1	1																	
NAND		$(AB)'$	The circle on the output negates the gate output.	<table><tr><td>A</td><td>B</td><td>Q</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Q	0	0	1	0	1	1	1	0	1	1	1	0
A	B	Q																	
0	0	1																	
0	1	1																	
1	0	1																	
1	1	0																	
NOR		$(A+B)'$	The circle on the output negates the gate output.	<table><tr><td>A</td><td>B</td><td>Q</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Q	0	0	1	0	1	0	1	0	0	1	1	0
A	B	Q																	
0	0	1																	
0	1	0																	
1	0	0																	
1	1	0																	
XOR		$A \oplus B$	The output of a two input exclusive-OR is true only when the two input values are different, and false if they are equal, regardless of the value	<table><tr><td>A</td><td>B</td><td>Q</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Q	0	0	0	0	1	1	1	0	1	1	1	0
A	B	Q																	
0	0	0																	
0	1	1																	
1	0	1																	
1	1	0																	
XNOR		$(A \oplus B)'$	The circle on the output negates the gate output.	<table><tr><td>A</td><td>B</td><td>Q</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Q	0	0	1	0	1	0	1	0	0	1	1	1
A	B	Q																	
0	0	1																	
0	1	0																	
1	0	0																	
1	1	1																	

Click Image to Zoom. PDF Available Below.



Gate symbol table chapter 2.pdf

166.5 KB



All of the gates will always have one output. The XOR and XNOR gates will almost always have exactly two inputs. There are exceptions but they are rare. The inverter will only have a single input.

As for the rest of the gates they will have from two to any number of inputs. The number of inputs to a gate is referred to as fan-in and is determined at design time. Logisim provides the ability to select the number of inputs to a gate. This is an example of what an 8-input AND gate looks like in Logisim.

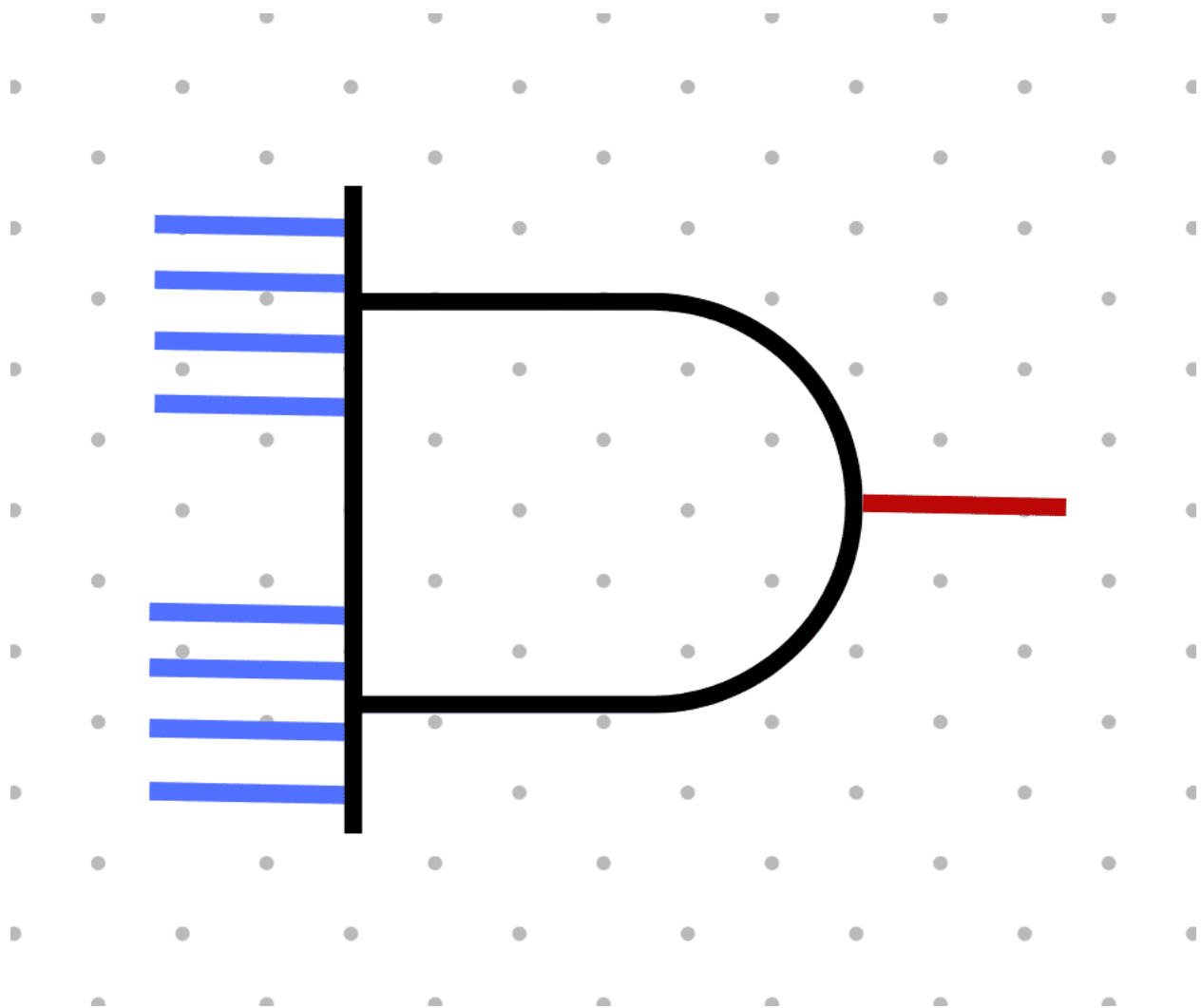


Image created using [Canva](#)

Regardless of the number of inputs the function of the gate is not changed. With the AND gate above if any of the eight inputs is ever a 0 the output of the gate will be a 0. The only time the output of the gate is a 1 is when all the inputs are a 1.

Universal logic gates

Charles Sanders Peirce (during 1880–81) showed that NOR gates alone (or alternatively NAND gates alone) can be used to reproduce the functions of all the other logic gates, but his work on it was unpublished until 1933. The first published proof was by Henry M. Sheffer in 1913, so

the NAND logical operation is sometimes called Sheffer stroke; the logical NOR is sometimes called Peirce's arrow. Consequently, these gates are sometimes called universal logic gates.

De Morgan equivalent symbols

By use of De Morgan's laws, an AND function is identical to an OR function with negated inputs and outputs. Likewise, an OR function is identical to an AND function with negated inputs and outputs. A NAND gate is equivalent to an OR gate with negated inputs, and a NOR gate is equivalent to an AND gate with negated inputs.

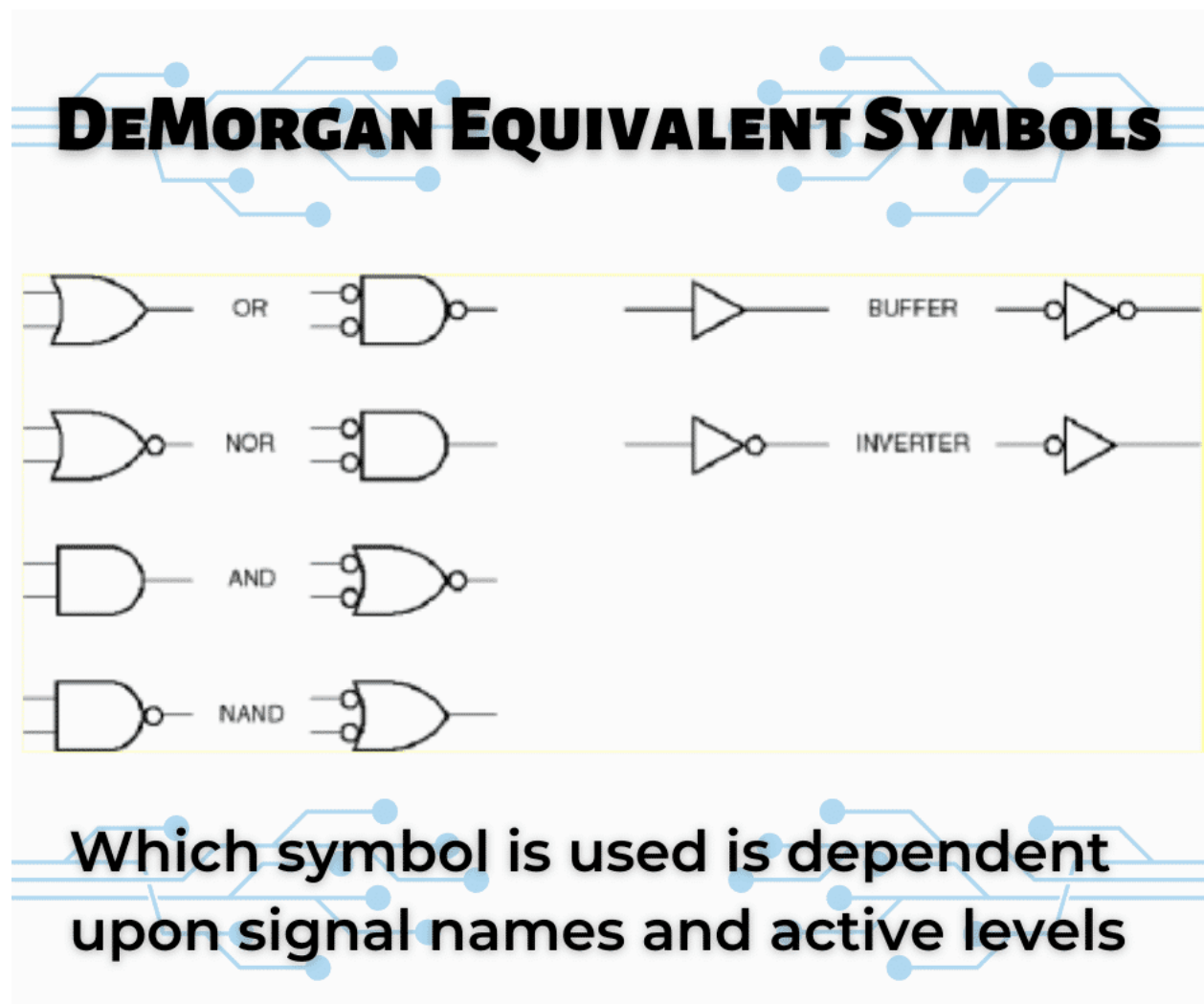


Image created using [Canva](#). PDF available below.

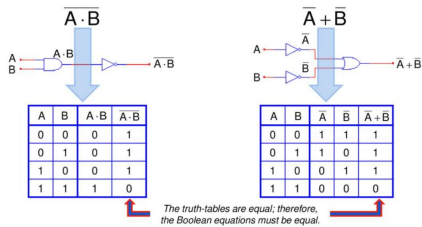
This leads to an alternative set of symbols for basic gates that use the opposite core symbol (AND or OR) but with the inputs and outputs negated. Use of these alternative symbols can

make logic circuit diagrams much clearer and help to show accidental connection of an active high output to an active low input or vice versa. Any connection that has logic negations at both ends can be replaced by a negation-less connection and a suitable change of gate or vice versa. Any connection that has a negation at one end and no negation at the other can be made easier to interpret by instead using the De Morgan equivalent symbol at either of the two ends. When negation or polarity indicators on both ends of a connection match, there is no logic negation in that path (effectively, bubbles "cancel"), making it easier to follow logic states from one symbol to the next. This is commonly seen in real logic diagrams – thus the reader must not get into the habit of associating the shapes exclusively as OR or AND shapes, but also take into account the bubbles at both inputs and outputs in order to determine the "true" logic function indicated.

DeMorgan's Theorem #1

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$

Proof



DeMorgan's Theorems

DeMorgan's Theorems are two additional simplification techniques that can be used to simplify Boolean expressions. Again, the simpler the Boolean expression, the simpler the resulting logic.

$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$

DeMorgan's Theorem #2

$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

Proof

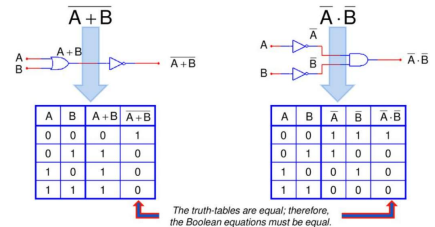


Image created using [Canva](#)

Image created using [Canva](#)

Image created using [Canva](#)

De Morgan's theorem is most commonly used to implement logic gates as combinations of only NAND gates, or as combinations of only NOR gates, for economic reasons.

A De Morgan symbol can show more clearly a gate's primary logical purpose and the polarity of its nodes that are considered in the "signaled" (active, on) state. Consider the simplified case where a two-input NAND gate is used to drive a motor when either of its inputs are brought low by a switch. The "signaled" state (motor on) occurs when either one OR the other switch is on. Unlike a regular NAND symbol, which suggests AND logic, the De Morgan version, a two negative-input OR gate, correctly shows that OR is of interest. The regular NAND symbol has a bubble at the output and none at the inputs (the opposite of the states that

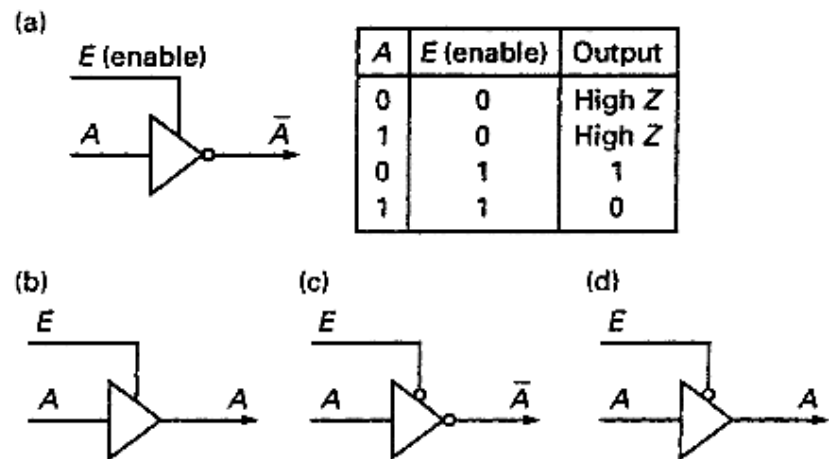
will turn the motor on), but the De Morgan symbol shows both inputs and output in the polarity that will drive the motor.



Gate symbols for table ch 2.pdf
240.6 KB



Three-State Logic Gates



Tristate buffers (a) Inverting, active high enable with truth table (b) Non-inverting, active high enable (c) Inverting, active low enable (d) Non-inverting, active low enable

Tristate Buffer

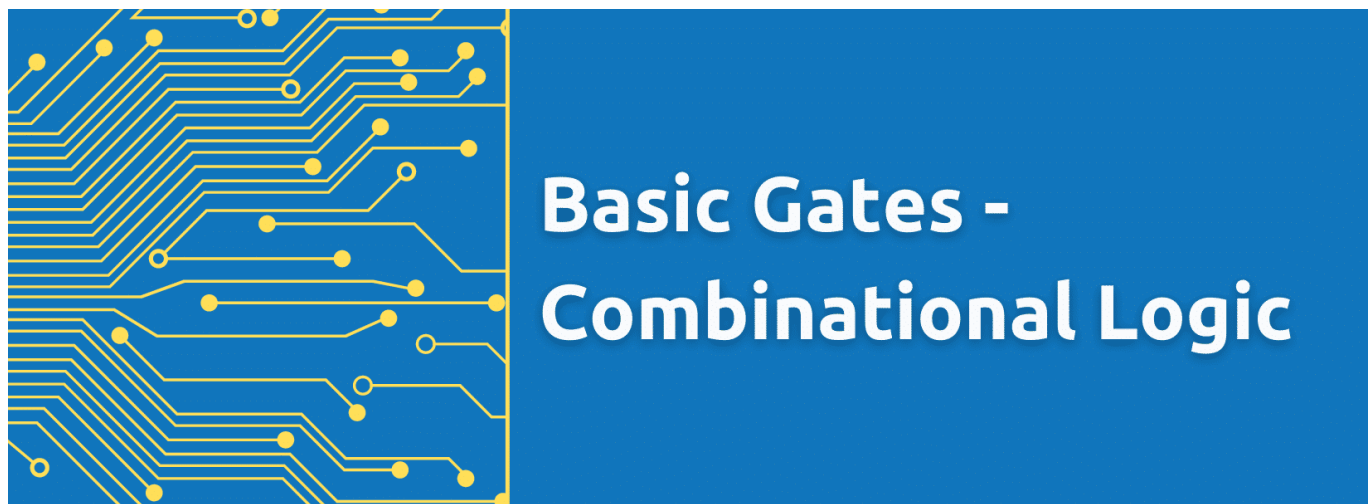
A tristate buffer can be thought of as a switch. If E is on, the switch is closed. If E is off, the switch is open. This figure shows the various types of tristate buffers that vary in if they are inverting or non-inverting buffers or enabled with a logic 1 or logic 0. Tristate buffers are very useful to controlling access to and reading from a bus. They are so useful that basic register or memory chips have tristate outputs.

A three-state logic gate is a type of logic gate that can have three different outputs: high (H), low (L) and high-impedance (Z). The high-impedance state plays no role in the logic, which

is strictly binary. These devices are used on buses of the CPU to allow multiple chips to send data. A group of three-states driving a line with a suitable control circuit is basically equivalent to a multiplexer, which may be physically distributed over separate devices or plug-in cards.

In electronics, a high output would mean the output is sourcing current from the positive power terminal (positive voltage). A low output would mean the output is sinking current to the negative power terminal (zero voltage). High impedance would mean that the output is effectively disconnected from the circuit.

Basic Gates - Combinational Logic



Introduction

A logic gate is an idealized or physical electronic device implementing a Boolean function, a logical operation performed on one or more binary inputs that produces a single binary output. Logic gates are primarily implemented using diodes or transistors acting as electronic switches, but can also be constructed using vacuum tubes, electromagnetic relays (relay logic), fluidic logic, pneumatic logic, optics, molecules, or even mechanical elements. With amplification, logic gates can be cascaded in the same way that Boolean functions can be composed, allowing the construction of a physical model of all of Boolean logic, and therefore, all of the algorithms and mathematics that can be described with Boolean logic.

NOTE: Before proceeding into this material students should download Logisim to their computer. Logisim is an easy to learn and use Java application to create and test digital circuits. No

install in required. Just open the executable to run it. Logisim will be used to demonstrate the circuits in this course and will be used for student lab submissions.

Combinational Logic

The term “combinational” comes to us from mathematics. In mathematics a combination is an unordered set, which is a formal way to say that nobody cares which order the items came in. Most games work this way, if you rolled dice one at a time and get a 2 followed by a 3 it is the same as if you had rolled a 3 followed by a 2. With combinational logic, the circuit produces the same output regardless of the order the inputs are changed.

Half-Adder

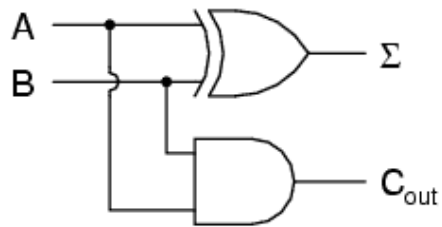
As a first example of useful combinational logic, let’s build a device that can add two binary digits together. We can quickly calculate what the answers should be:

$$0 + 0 = 0 \quad 0 + 1 = 1 \quad 1 + 0 = 1 \quad 1 + 1 = 10_2$$

Gif created using [Canva](#)

So we will need two inputs (a and b) and two outputs. The low order output will be called Σ because it represents the sum, and the high order output will be called C_{out} because it represents the carry out. The truth table is

A	B	Σ	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Simplifying boolean equations or making some Karnaugh map will produce the same circuit shown below, but start by looking at the results. The Σ column is our familiar XOR gate, while the C_{out} column is the AND gate. This device is called a half-adder for reasons that will make sense in the next section.

Full-Adder

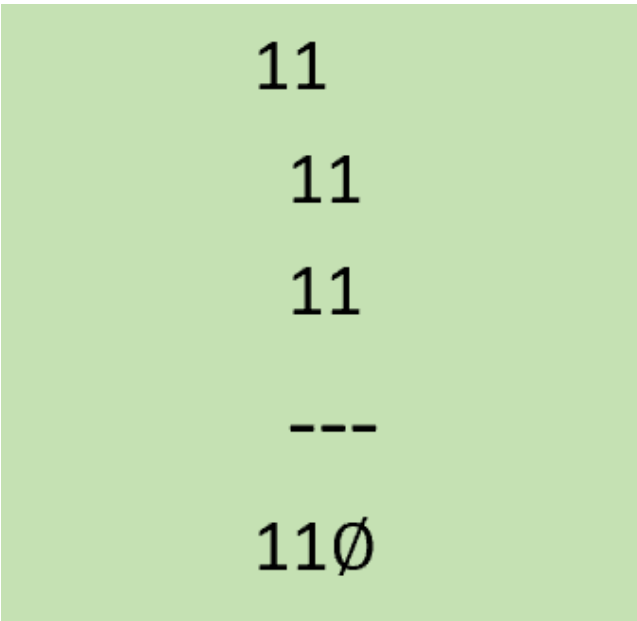
Click through the slides below to learn more about the usefulness of the full-adder. You may click each image to zoom in and out.

Full-Adder

The half-adder is extremely useful until you want to add more than one binary digit quantities. The slow way to develop a two binary digit adders would be to make a truth table and reduce it. Then when you decide to make a three binary digit adder, do it again. Then when you decide to make a four digit adder, do it again. Then when ...

The circuits would be fast, but development time would be slow.

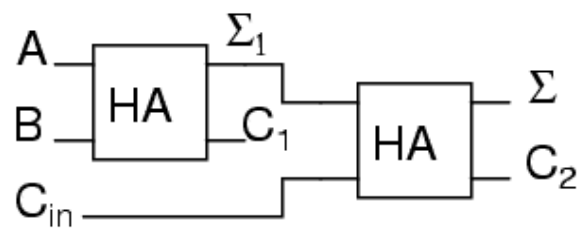
Step 1


$$\begin{array}{r} 11 \\ 11 \\ 11 \\ \hline 110 \end{array}$$

Looking at a two binary digit sum shows what we need to extend addition to multiple binary digits.

Look at how many inputs the middle column uses. Our adder needs three inputs; a, b, and the carry from the previous sum, and we can use our two-input adder to build a three input adder.

Step 2



Σ is the easy part. Normal arithmetic tells us that if $\Sigma = a + b + C_{in}$ and $\Sigma_1 = a + b$, then $\Sigma = \Sigma_1 + C_{in}$.

Step 3

Cin + a + b = ?

$$\emptyset + \emptyset + \emptyset = \emptyset$$

$$\emptyset + \emptyset + 1 = 1$$

$$\emptyset + 1 + 1 = 1\emptyset$$

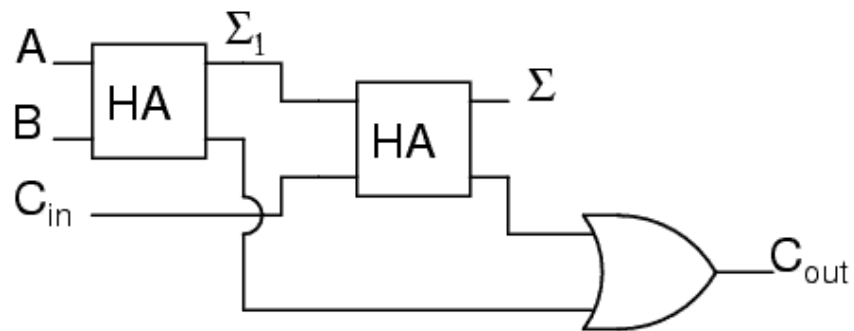
$$1 + \emptyset + \emptyset = 1$$

$$1 + \emptyset + 1 = 1\emptyset$$

$$1 + 1 + 1 = 11$$

What do we do with C1 and C2? Let's look at three input sums and quickly calculate:

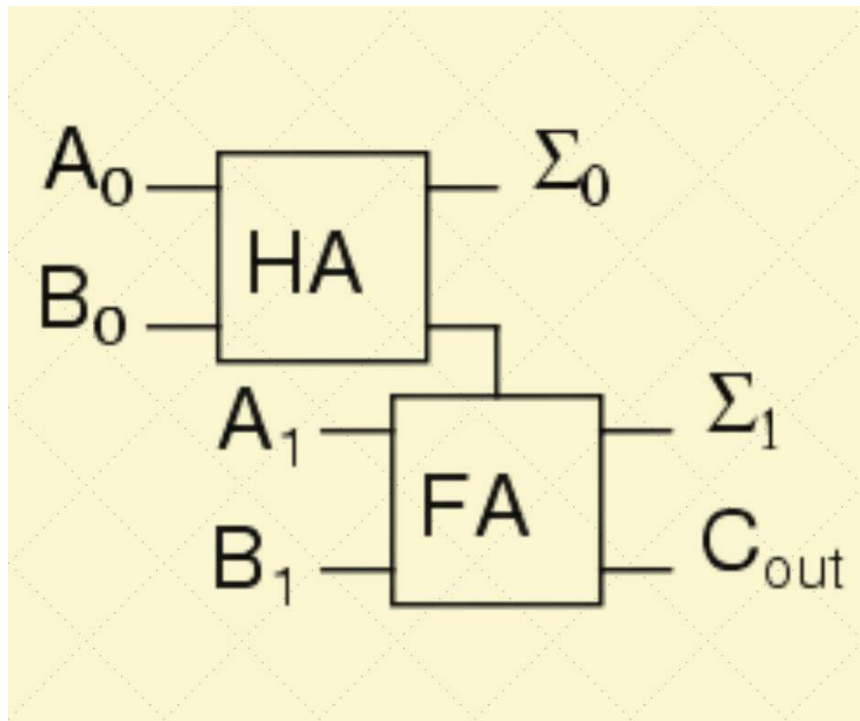
Step 4



If you have any concern about the low order bit, please confirm that the circuit calculates it correctly.

In order to calculate the high order bit, notice that it is 1 in both cases when $a + b$ produces a C_1 . Also, the high order bit is 1 when $a + b$ produces a Σ_1 and C_{in} is a 1. So We will have a carry when C_1 OR $(\Sigma_1 \text{ AND } C_{in})$. Our complete three input adder is: (See image at top of Step 4)

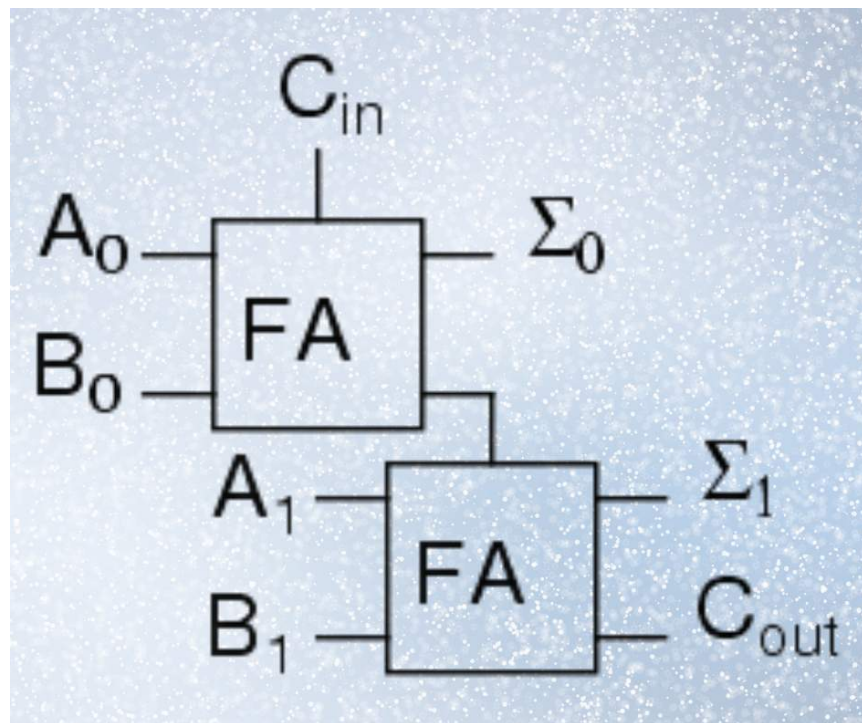
Step 5



For some designs, being able to eliminate one or more types of gates can be important, and you can replace the final OR gate with an XOR gate without changing the results.

We can now connect two adders to add 2 bit quantities.

Step 6



A_0 is the low order bit of A, A_1 is the high order bit of A, B_0 is the low order bit of B, B_1 is the high order bit of B, Σ_0 is the low order bit of the sum, Σ_1 is the high order bit of the sum, and C_{out} is the Carry.

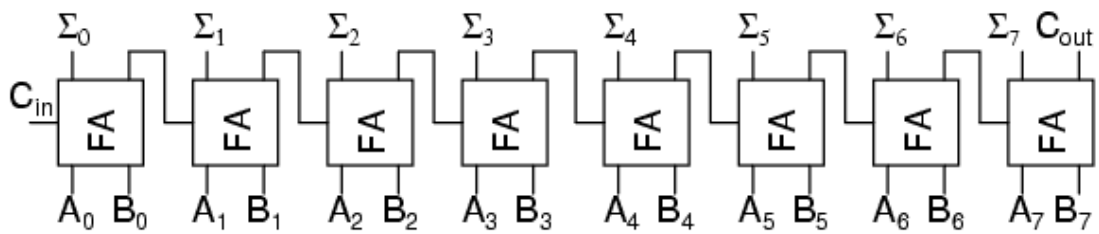
A two binary digit adder would never be made this way. Instead the lowest order bits would also go through a full adder.

Step 7

111	1< -	+	11< +	-	
0110			01		10
1011			10		11
-----	-		-----		---
10 0 01	1	+	-100	+	-101

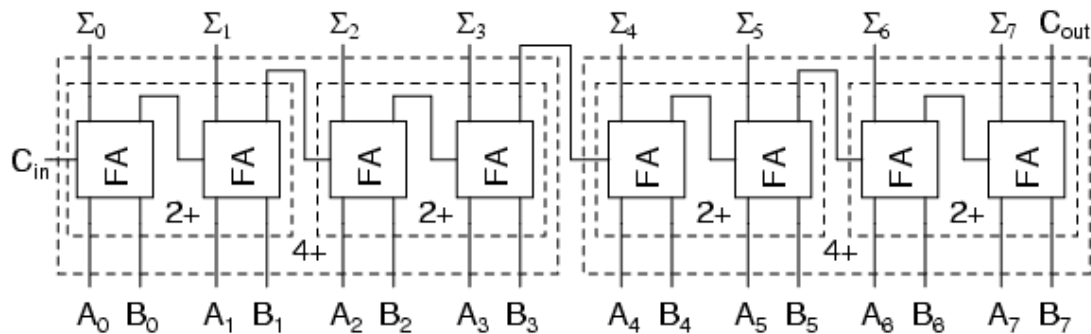
There are several reasons for this, one being that we can then allow a circuit to determine whether the lowest order carry should be included in the sum. This allows for the chaining of even larger sums. Consider two different ways to look at a four bit sum.

Step 8



If we allow the program to add a two bit number and remember the carry for later, then use that carry in the next sum the program can add any number of bits the user wants even though we have only provided a two-bit adder. Small PLCs can also be chained together for larger numbers. These full adders can also can be expanded to any number of bits space allows. As an example, here's how to do an 8 bit adder.

Step 9



This is the same result as using the two 2-bit adders to make a 4-bit adder and then using two 4-bit adders to make an 8-bit adder or re-duplicating ladder logic and updating the numbers.

Each “2+” is a 2-bit adder and made of two full adders. Each “4+” is a 4-bit adder and made of two 2-bit adders. And the result of two 4-bit adders is the same 8-bit adder we used full adders to build. For any large combinational circuit there are generally two approaches to design: you can take simpler circuits and replicate them; or you can design the complex circuit as a complete device. Using simpler circuits to build complex circuits allows a you to spend less time designing but then requires more time for signals to propagate through the transistors. The 8-bit adder design above has to wait for all the Cxout signals to move from $A_0 + B_0$ up to the inputs of Σ_7 . If a designer builds an 8-bit adder as a complete device simplified to a sum of products, then each signal just travels through one NOT gate, one AND gate and one OR gate. A seventeen input device has a truth table with 131,072 entries (2^{17}), and reducing 131,072 entries to a sum of products will take some time. When designing for systems that have a maximum allowed response time to provide the final result, you can begin by using simpler circuits and then attempt to replace portions of the circuit that are too slow. That way you spend most of your time on the portions of a circuit that matter.

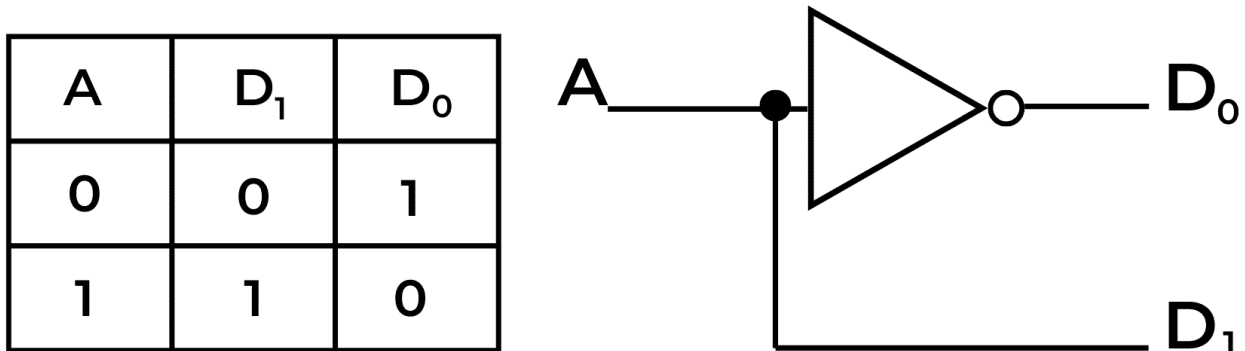
Decoder

A decoder is a circuit that changes a code into a set of signals. It is called a decoder because it does the reverse of encoding, but we will begin our study of encoders and decoders with decoders because they are simpler to design. Expand each section of the accordion interaction to learn more about each type:

Line Decoder —

A common type of decoder is the line decoder which takes an n -digit binary number and decodes it into 2^n data lines. The simplest is the 1-to-2 line decoder. The truth table is (see table image)

A is the address and D is the data line. D_0 is NOT A and D_1 is A. The circuit looks like (see circuit image)

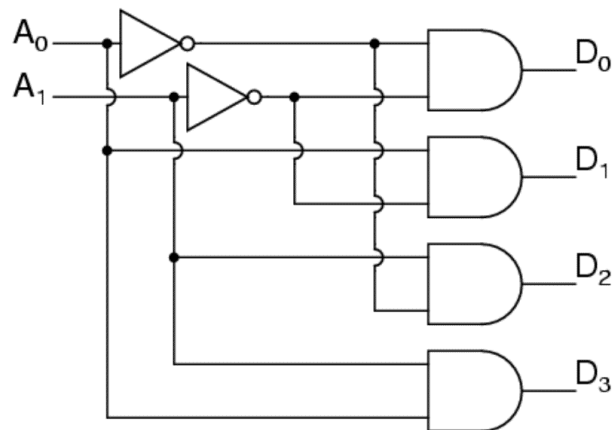


2-to-4 Line Decoder —

Only slightly more complex is the 2-to-4 line decoder. The truth table is (see table image)

Developed into a circuit it looks like (see circuit image)

A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



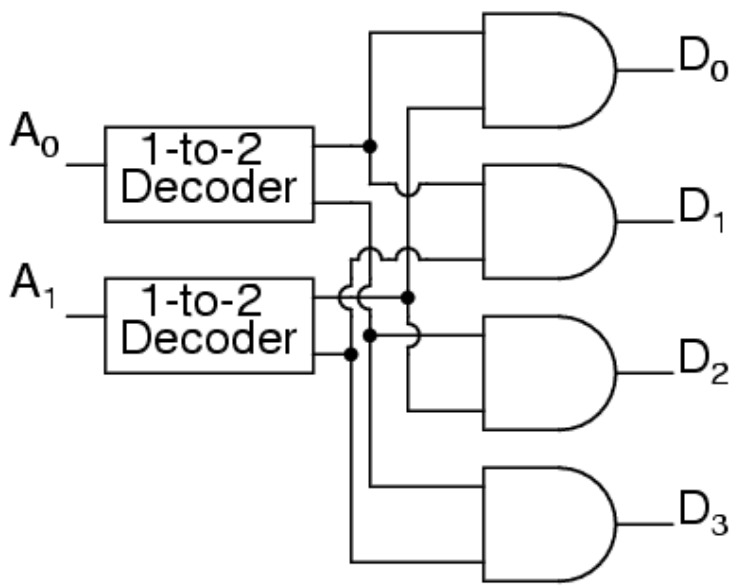
Larger Line Decoders

Larger line decoders can be designed in a similar fashion, but just like with the binary adder there is a way to make larger decoders by combining smaller decoders. An alternate circuit for the 2-to-4 line decoder is

Replacing the 1-to-2 Decoders with their circuits will show that both circuits are equivalent. In a similar fashion a 3-to-8 line decoder can be made from a 1-to-2 line decoder and a 2-to-4 line decoder, and a 4-to-16 line decoder can be made from two 2-to-4 line decoders.

For an eight-bit adder we only know how to sum eight bits by summing one bit at a time. Usually it is easier to design ladder logic from boolean equations or truth tables rather than design logic gates and then “translate” that into ladder logic.

A typical application of a line decoder circuit is to select among multiple devices. A circuit needing to select among sixteen devices could have sixteen control lines to select which device should “listen”. With a decoder only four control lines are needed.



Encoder

D_1	D_0	A
0	1	0
1	0	1

An encoder is a circuit that changes a set of signals into a code. Let's begin making a 2-to-1 line encoder truth table by reversing the 1-to-2 decoder truth table.

This truth table is a little short. A complete truth table would be

D_1	D_0	A
0	0	
0	1	0
1	0	1
1	1	

One question we need to answer is what to do with those other inputs? Do we ignore them? Do we have them generate an additional error output? In many circuits, this problem is solved by adding sequential logic in order to know not just what input is active but also which order the inputs became active.

Demultiplexers

A demultiplexer, sometimes abbreviated dmux, is a circuit that has one input and more than one output. It is used when a circuit wishes to send a signal to one of many devices. This description sounds similar to the description given for a decoder, but a decoder is used to

select among many devices while a demultiplexer is used to send a signal among many devices.

A demultiplexer is used often enough that it has its own schematic symbol

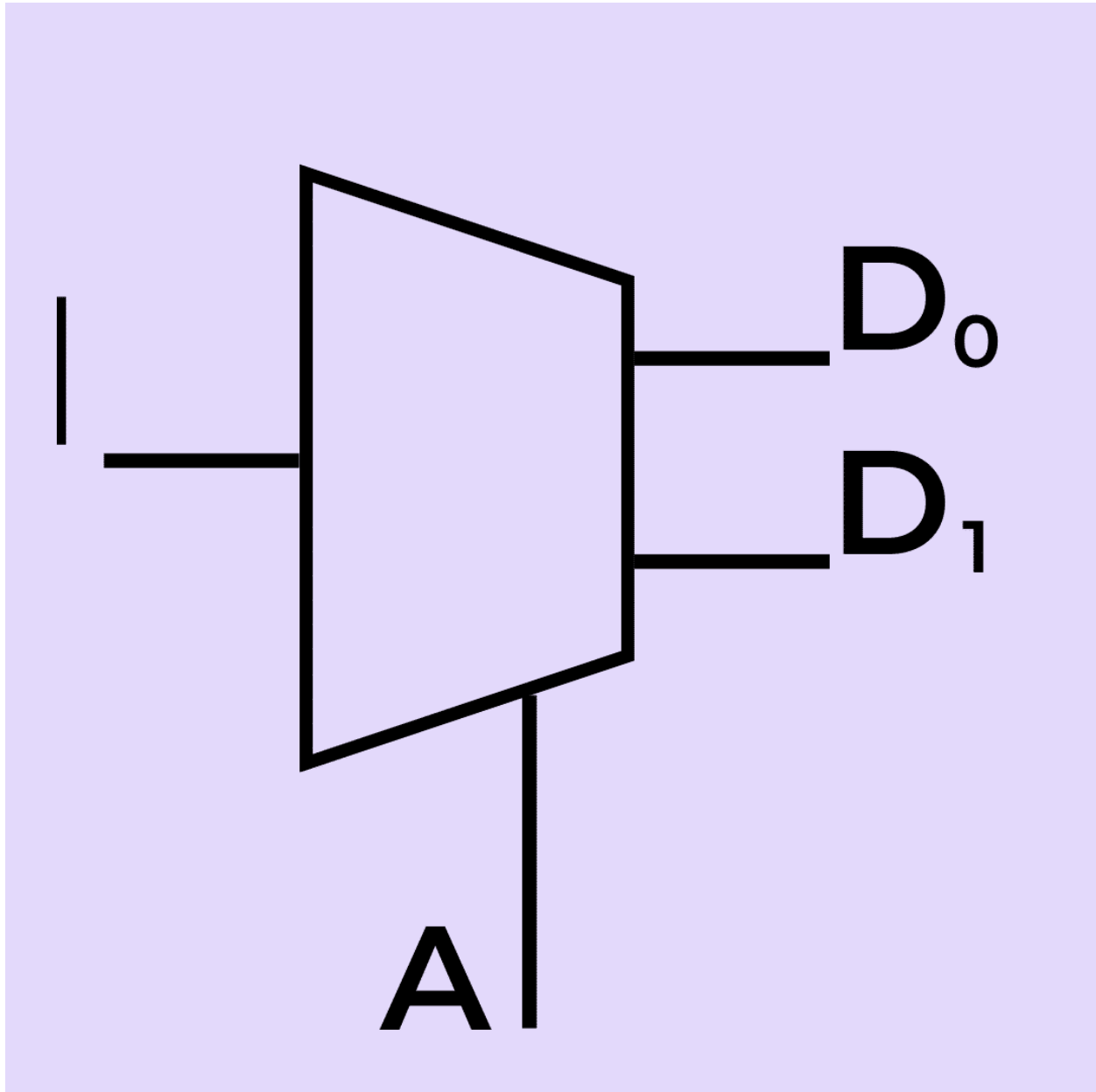


Image created with [Canva](#)

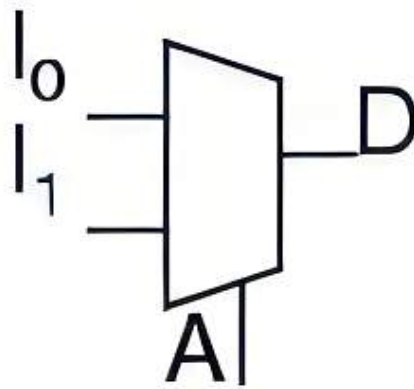
The truth table for a 1-to-2 demultiplexer is (see below). Using our 1-to-2 decoder as part of the circuit, we can express this circuit easily

	A	D ₀	D ₁
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

Multiplexers

Multiplexers

A multiplexer, abbreviated mux, is a device that has multiple inputs and one output. The schematic symbol for multiplexers is

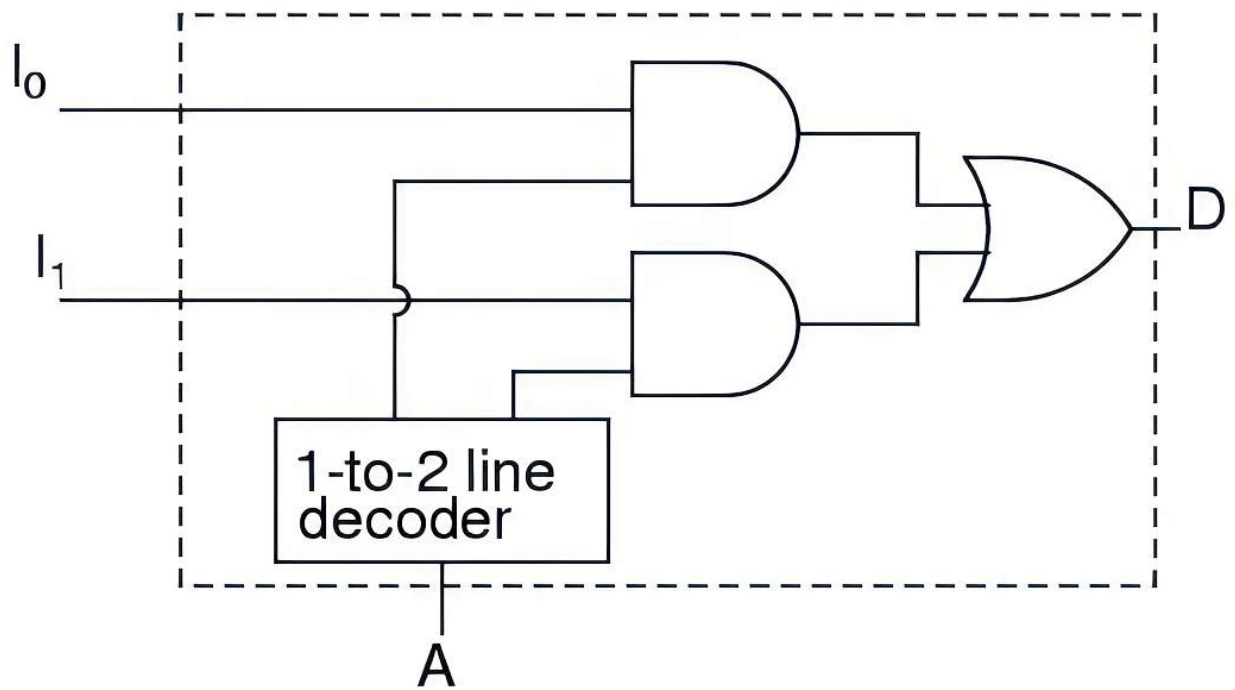


The truth table for a 2-to-1 multiplexer is

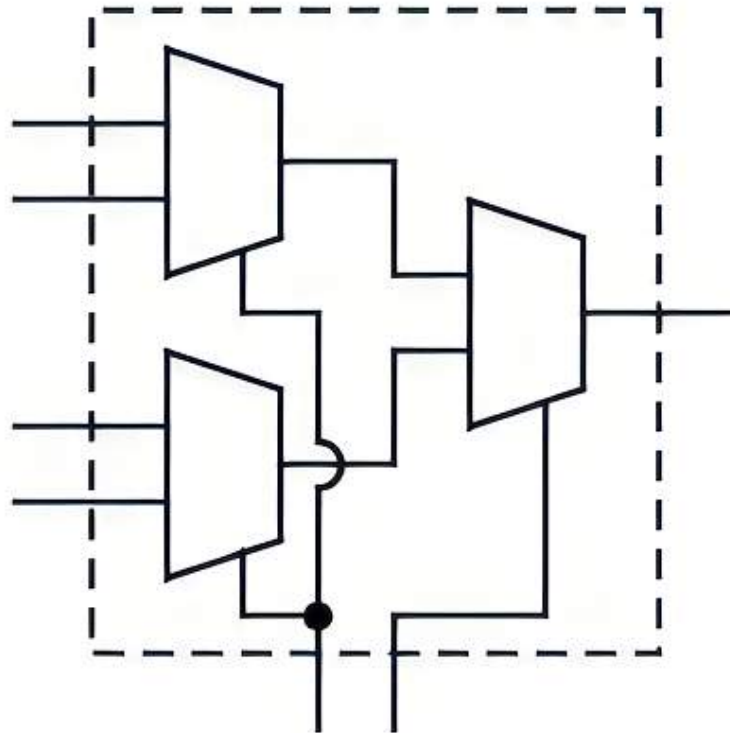
When we show up to the present moment with all of our senses, we invite the world to fill us with joy. The pains of the past are behind us. The future has yet to unfold. But the now is full of beauty simply waiting for our attention.

I_1	I_0	A	D
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Using a 1-to-2 decoder as part of the circuit, we can express this circuit easily.



Multiplexers can also be expanded with the same naming conventions as demultiplexers. A 4-to-1 multiplexer circuit is



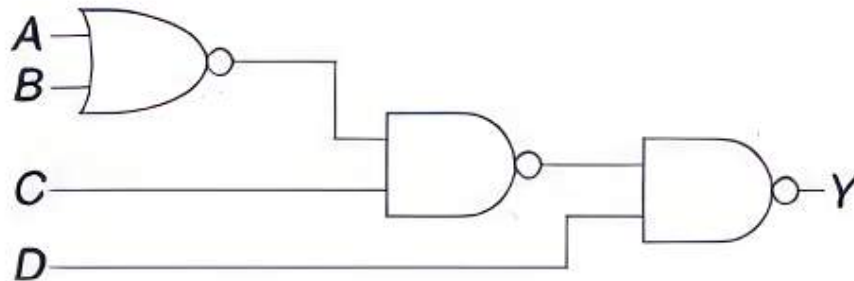
That is the formal definition of a multiplexer. Informally, there is a lot of confusion. Both demultiplexers and multiplexers have similar names, abbreviations, schematic symbols and circuits, so confusion is easy. The term multiplexer, and the abbreviation mux, are often used

to also mean a demultiplexer, or a multiplexer and a demultiplexer working together. So when you hear about a multiplexer, it may mean something quite different.

Analysis of a Combinational Circuit

One of the issues you may have to deal with is trying to determine what a combinational circuit is doing. This process starts with a circuit diagram and will typically end with a truth-table. Under some conditions it is possible to provide a short description of the function of the circuit like "Grey code convertor".

In this example we want to analyze this following circuit.



1

Step 1: Label all the gates - $U_1, U_2, U_3, \dots U_n$. Note: Historically labeling logic gates is of the form U followed by an integer. I'm not sure why but if I had to guess it is much like all resistor labels start with R, capacitors start with C and inductors start with L.

2

Step 2: Starting with the gates closest to the inputs express each gate's output as a column in the truth table. This process works well with a small number of inputs (five or less). If there are too many inputs the Boolean function equation will have to be used. For this course we are going to stick with analysis of circuits that have five or less inputs.

It really helps to remember some of the short cuts related to specific gates. For example if you know that one of the inputs to a four-input AND gate is a 0 there is no need to evaluate the other inputs since the AND gate will output a 0 regardless of the other input values.

3

Step 3: Continue moving from the inputs to all the other gates until all outputs are expressed.

The completed true table looks like this:

Dec	A	B	C	D	U1-NOR	U2-NAND	U3 (Y)
0	0	0	0	0	1	1	1
1	0	0	0	1	1	1	0
2	0	0	1	0	1	0	1
3	0	0	1	1	1	0	1
4	0	1	0	0	0	1	1
5	0	1	0	1	0	1	0
6	0	1	1	0	0	1	1
7	0	1	1	1	0	1	0
8	1	0	0	0	0	1	1
9	1	0	0	1	0	1	0
10	1	0	1	0	0	1	1
11	1	0	1	1	0	1	0
12	1	1	0	0	0	1	1
13	1	1	0	1	0	1	0
14	1	1	1	0	0	1	1
15	1	1	1	1	0	1	0

You will notice the first column of the truth table is the decimal number that corresponds to the row number which is also the binary number of the input values (ABCD). This provides an

easier way to refer to a specific row and ensure the truth table contains all the possible values for the inputs and no values are missing.

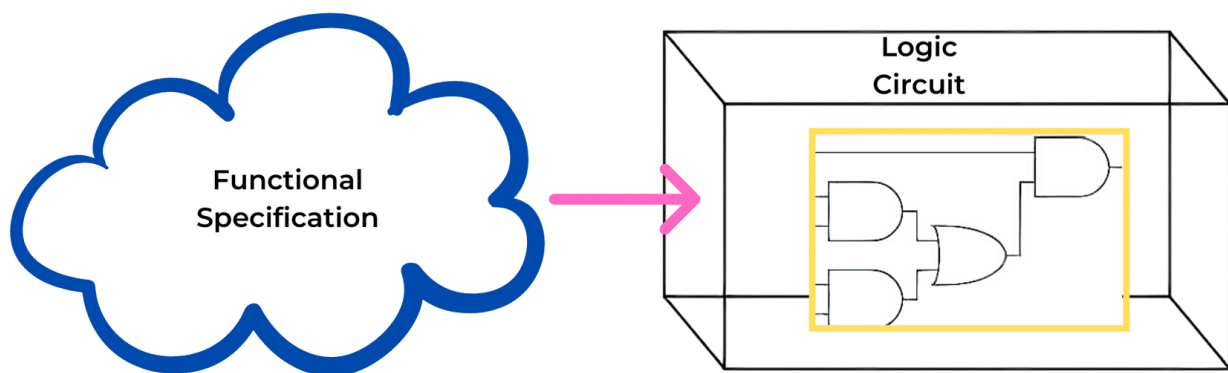
Expect the students:

- To be able to analyze a circuit and provide the truth table.
- Given the boolean expression (equation) provide the truth table or logic diagram for the circuit.

You can verify your analysis is correct by using Logisim and running through all possible inputs to verify your final output is correct. This is left as an exercise for the students.

Designing of a Combinational Circuit

What if one of the "standard" circuits (as given above) does not work for your particular application? That is where this design process comes into play. In this process you start with a functional specification which is usually given in terms that maybe somewhat vague and incomplete. The final product is a simplified circuit (minimum number of gates) that implements the requirements.



This is how you design a Combinational circuit.

Determine the number of inputs and outputs and assign unique labels to each one.

- Express the outputs solely as a function of the inputs
- Minimize the output function(s)
- Implement the simplified circuit.

Depending on the requirements or your preference the output function(s) can be represented in one of the following forms:

- The minimized Sum Of Products (SOP) form (minterms) produces an AND-OR implementation.
- The minimized Product of Sums (POS) form (maxterms) produces a OR-AND implementation.

This design process leads to two levels of gates. Circuit inputs connect directly to AND gates and the output of the AND gates connect to the inputs of the OR gates which provide the final output. Always assume that every input is available in its normal form (uncomplemented) and complemented form.

Regardless of the minimized function the basic gates can be replaced with Universal Gates (NAND or NOR).

If NAND-NAND is the final implementation:

- Draw the AND-OR circuit (Sum of Products)
- Replace the ANDs, and ORs, with the target gates then simplify by removing all NOT gates that feed only into another NOT gate.

If NOR-NOR is the final implementation:

- Draw the OR-AND circuit (Product of Sums)
- Replace the ANDs, and ORs with the target gate then simplify by removing all NOT gates that feed only into another NOT gate.

Design Example

Let's now go through a complete design example.

Functional specification: Design a circuit which takes 4-binary inputs and turn the output on (Logic 1) if the input value is Odd or greater than a decimal 10 (1010 in binary).

Step 1

Step 1

Step 1 - Identify the inputs 4-bits. A, B, C, D and the output Y. A is the most significant bit and D is the least significant bit.

Step 2

Step 2

Truth Table					
	A	B	C	D	Y
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	1

Step 2 - Present as a truth-table

Step 3

Step 3

Of this entire design process simplification of the circuit is the most difficult for students to get correct. This is especially true if there are four or more inputs to the circuit. In previous generations of this course several lectures dealt with covering Karnaugh maps and the Quine–McClusky method. Even with all the time spent on the topic students typically were not able to do the process correctly. In most of the modern textbooks these processes are mentioned but are not discussed in detail.

This course will use a Website that does the simplifications. There are much more important topics that need to be covered in this course than covering how to perform boolean circuit simplification.

<http://www.32x8.com/>

Another website but it is not as robust:

http://www.ee.calpoly.edu/media/uploads/resources/KarnaughExplorer_1.html

Results:

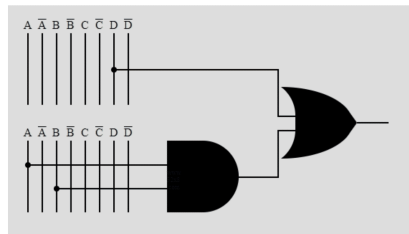
$Y = D + AB$ for Sum of Products AND-OR implementation

$Y = (A + D)(B + D)$ for Product of Sums OR-AND implementation

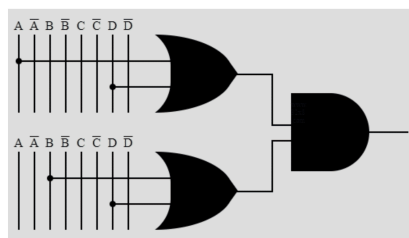
Step 4

Step 4

This is the AND-OR implementation



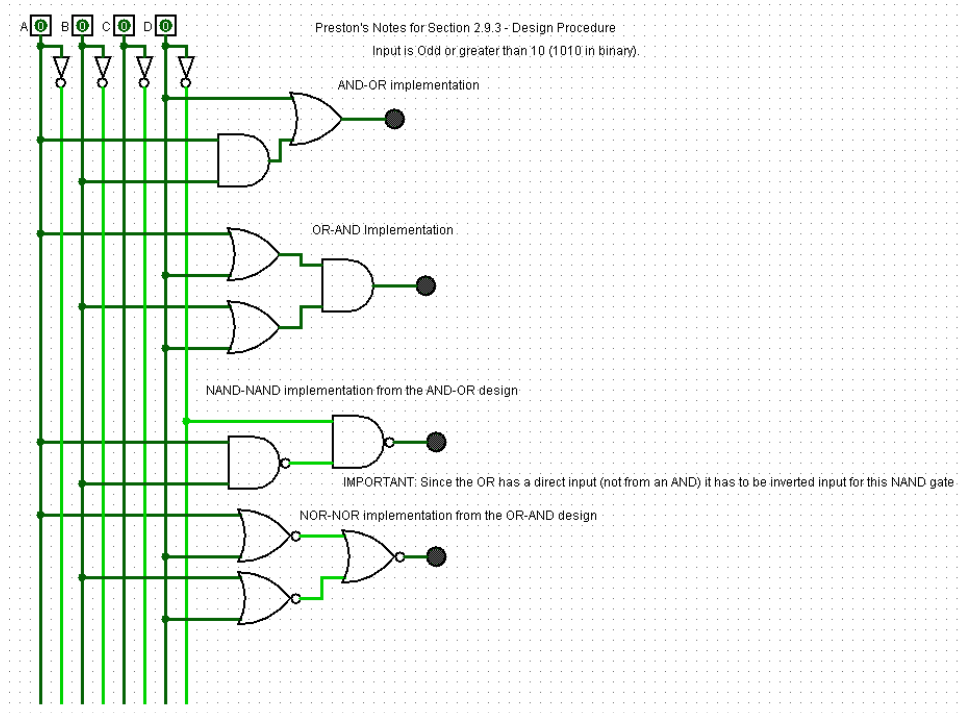
This is the OR-AND implementation



Step 4 - Draw the Circuit. (See images)

Step 5

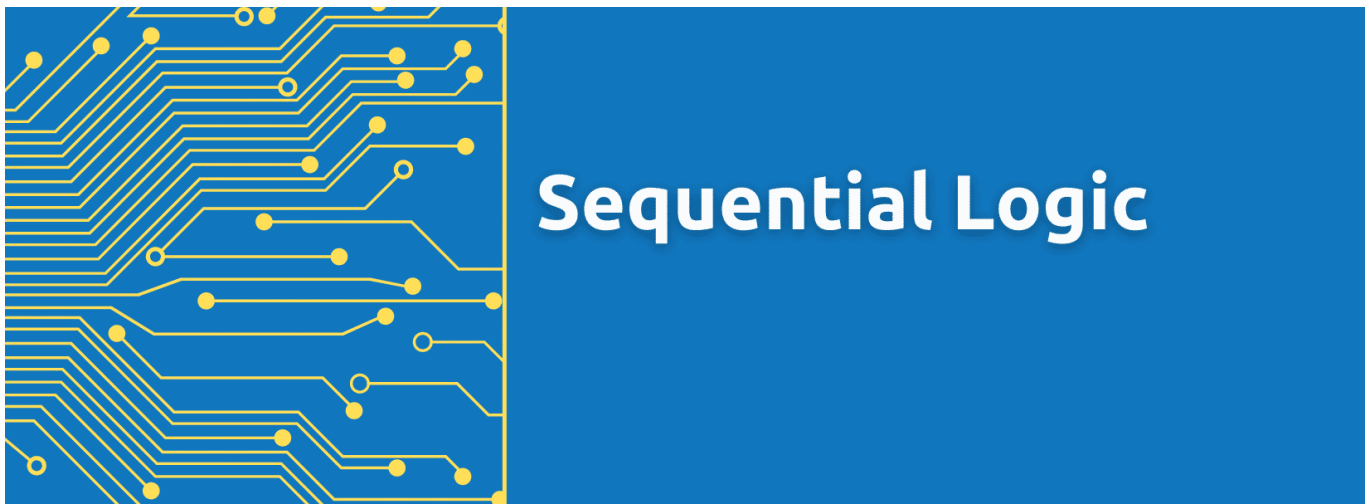
Step 4b



Always assume that every input is available in its normal form (uncomplemented) and complemented form.

This is the screen shot of the four different versions of the circuit.

Sequential Logic



Introduction

sequential logic is a type of logic circuit whose output depends on the present value of its input signals and on the sequence of past inputs, the input history. This is in contrast to combinational logic, whose output is a function of only the present input. That is, sequential logic has state (memory) while combinational logic does not.

Sequential logic is used to construct finite-state machines, a basic building block in all digital circuitry. Virtually all circuits in practical digital devices are a mixture of combinational and sequential logic.

From: [Wikipedia](#)



© Raimond Spekking / CC BY-SA 4.0 (via [Wikimedia Commons](#))

Objectives

From this module students should learn:

- The difference between a latch and flip-flop
- How flip-flops are used to form a register
- How registers can transfer data via a bus

Students should:

- Read the following material
- Watch the associated video lecture
- Take the Canvas quiz on this topic
- Design, implement and test a sequential logic circuit. Submit the resulting circuit to Canvas

Sequential Logic

Sequential logic consists of Combinational logic paired with memory elements. These memory elements are capable of storing information as long as needed.

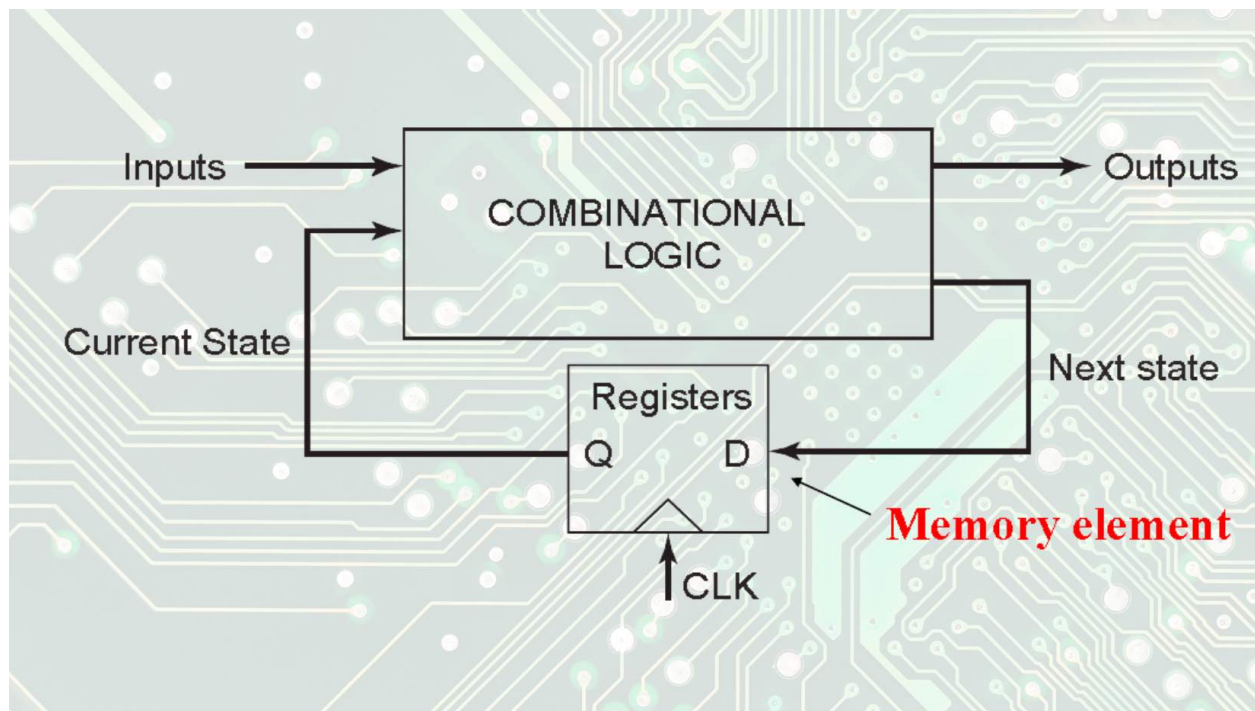


Image created using [Canva](#)

The memory elements can be Latches (no clock) and/or Flip-Flops (clock) as shown above.

Synchronous - Has some sort of clock or trigger signal. Flip-Flops

Asynchronous - No clock. Dependent on gate delays for everything to settle. Latches

The output of the circuit is a function of the inputs and its current state.

Latches

A latch is one of the basic memory elements that store information in a digital system. One latch can store 1-bit of information. The outputs of a latch are constantly changing according

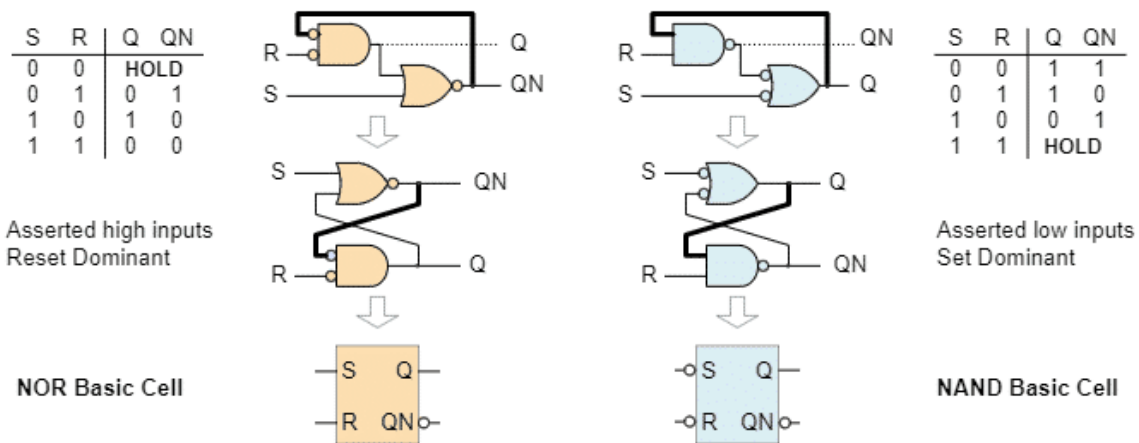
to the inputs as long as the latch is enabled. In other words, the content of a latch changes immediately when the inputs change when it is enabled. When a latch is not enabled, the latch holds the value that is stored.

Here we will talk about two different kinds of latches: an SR-Latch and a D-Latch. The differences between them are the number of inputs and how they change the state.

SR-Latch

Click each tab below for information concerning the SR-Latch.

SR-LATCH	NAND AND NOR CIRCUITS	FIGURE 2 EXPLAINED	EXAMINING TRUTH TABLES AND FIGURES	
SR-Latches use two inputs named S (for set) and R (for reset), and an output named Q (by convention, Q is nearly always used to label the output signal from a memory device). The S input, when asserted, sets the output to a '1', and the R input resets the output to a '0'. Figure 1 shows two basic implementations of an SR-Latch. These two cells are commonly referred to as basic cells. In the NAND circuit on the left, S must be driven to '0' to drive Q to '1', and R must be driven to '0' to drive Q to '0' (so both S and R are active low). The output Q is set by the positive pulse on S at time 2, and Q remains set until it is reset at time 3. Thus, Q exhibits memory by remaining at '1' after the input S is de-asserted, and during the time between point 2 and point 3 the circuit memorized a logic '1'. Likewise, when R is asserted (as a negative pulse), Q is reset to logic '0' and it remains there until it is set sometime in the future, and the circuit memorized a logic '0'.				



SR-LATCH

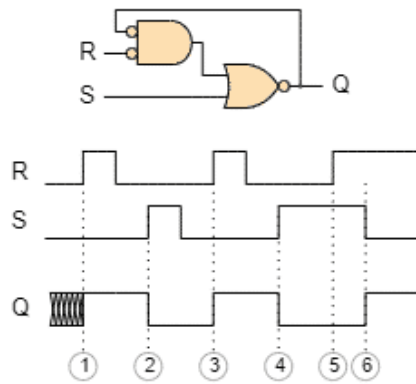
NAND AND NOR
CIRCUITS

FIGURE 2 EXPLAINED

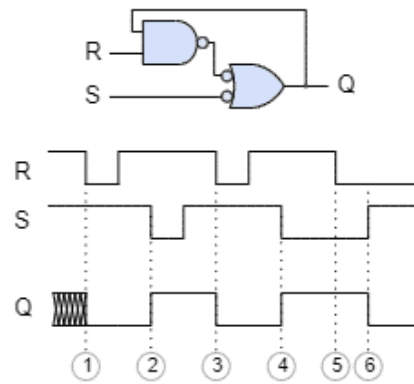
EXAMINING TRUTH
TABLES AND FIGURES

In the NOR circuit on the right, S must be driven to '1' to drive Q to '0', and R must also be driven to '1' to drive Q to '1' (so both S and R are active high). The reader is highly encouraged to examine the circuits and timing diagrams below, and ensure that the behaviors shown are well understood.

The NAND and NOR circuits are symmetric, so either input can be labeled S or R. By convention, the output that S drives to '1' is called Q, and the output that S drives to '0' is called QN (and thus the NOR-based circuit above is mislabeled, while the one below is correctly labeled). In the NOR circuit, a '1' on S drives Q to '1' (provided R is at '0'), and so the NOR circuit inputs are active high. In the NAND circuit, a '0' on S drives Q to '1', and so the NAND inputs are active low.



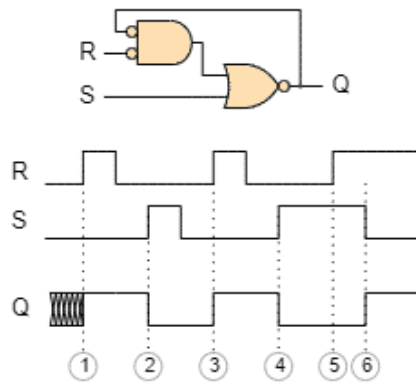
1. Q is undefined until R is asserted
2. Q → '0' when S is asserted
3. Q → '1' when R is asserted
4. Q → '0' when S is asserted
5. Q stays at '0' when S & R asserted
6. Q → '1' R when asserted, S de-asserted



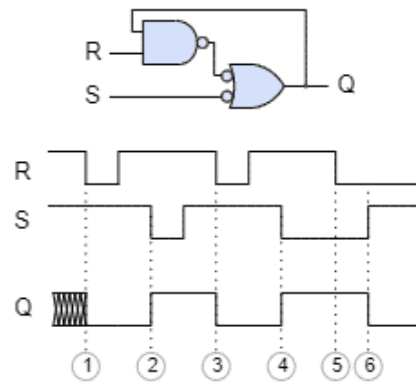
1. Q is undefined until R is asserted
2. Q → '1' when S is asserted
3. Q → '0' when R is asserted
4. Q → '1' when S is asserted
5. Q stays at '1' when S & R asserted
6. Q → '0' when R asserted, S de-asserted

SR-LATCH	NAND AND NOR CIRCUITS	FIGURE 2 EXPLAINED	EXAMINING TRUTH TABLES AND FIGURES
----------	--------------------------	--------------------	---------------------------------------

In Fig. 2 below, the basic cells have been redrawn as a cross-coupled circuit, with the feedback path emboldened for emphasis. In the NOR basic cell, the Q output is derived from the gate driven directly by R, and so R can determine the output Q regardless of S: if R is driven to a '1', Q will be a '0' regardless of S. Thus, a NOR basic cell is said to be reset dominant. In the NAND basic cell, the input S can determine the output regardless of R: if S is driven to a '0', Q will be '1' regardless of R. The NAND basic cell is said to be set-dominant. The difference between set and reset dominance are evident in the truth table rows where both inputs are asserted. In the reset-dominant NOR cell, Q is forced to '0' when R is asserted (last row), and in the set-dominant NAND cell Q is forced to '1' when S is asserted (first row).



1. Q is undefined until R is asserted
2. $Q \rightarrow '0'$ when S is asserted
3. $Q \rightarrow '1'$ when R is asserted
4. $Q \rightarrow '0'$ when S is asserted
5. Q stays at '0' when S & R asserted
6. $Q \rightarrow '1'$ R when asserted, S de-asserted



1. Q is undefined until R is asserted
2. $Q \rightarrow '1'$ when S is asserted
3. $Q \rightarrow '0'$ when R is asserted
4. $Q \rightarrow '1'$ when S is asserted
5. Q stays at '1' when S & R asserted
6. $Q \rightarrow '0'$ when R asserted, S de-asserted

SR-LATCH	NAND AND NOR CIRCUITS	FIGURE 2 EXPLAINED	EXAMINING TRUTH TABLES AND FIGURES
----------	--------------------------	--------------------	---------------------------------------

Examining the truth tables and Fig. 2 above yields the following observations:

- The middle two rows of the truth tables are similar for both circuits (i.e., both Q and QN are driven opposite from one another when either just S is asserted or just R is asserted).
- When both inputs are asserted, Q and QN are driven to the same logic level (i.e., they are no longer inverses of one another).
- When neither input is asserted, the logic level present on the feedback loop determines the circuit output.

Based on these observations, we can state the following behavioral rules for a basic cell (remembering that SET and RESET are active high for the NOR cell and low for the NAND cell):

- When just SET is active, Q is driven to '1' and QN is driven to '0';
- When just RESET is active, Q is driven to '0' and QN is driven to '1';
- When both SET and RESET are active, Q and QN are both driven to '0' (NOR cell) or '1' (NAND cell);
- When neither SET or RESET are active, the output is determined by the logic value stored in the feedback loop.

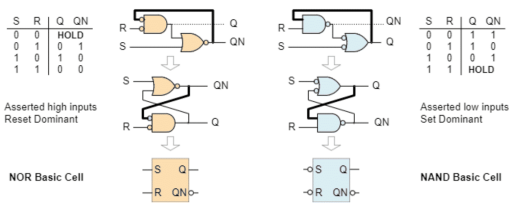


Figure 1

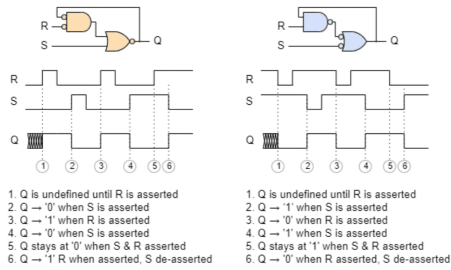


Figure 2

SR-LATCH	NAND AND NOR CIRCUITS	FIGURE 2 EXPLAINED	EXAMINING TRUTH TABLES AND FIGURES
----------	-----------------------	--------------------	------------------------------------

If both inputs to a basic cell are de-asserted at exactly the same time, the feedback loop can become astable, and the memory device can get temporarily stuck in the astable region. This results from the fact that two different logic levels are introduced into the feedback loop at the same time, and these values chase each other around the loop creating an oscillation. The oscillation shown in the simulator results from the fact that gate delays can be set to exactly the same value, and inputs can be changed at exactly the same time. In a real circuit, gate delays are not identical and input values cannot change (to the picosecond) at exactly the same time. Thus, oscillations may be seen, but only for a short while. Equally likely is an output that floats temporarily between '1' and '0'. Either behavior represents metastability, where the output from the memory circuit is temporarily not in one of the two stable operating states. Metastable states are highly unlikely in a real circuit, and if they are entered, they are quickly resolved to a stable state. But it is important to note that the possibility of a memory device entering a metastable state can never be eliminated.

We will use the NAND cell in the following discussion, but similar circuits could be built with the NOR cell.

D-Latch

The basic cell is the most rudimentary memory device, and it is useful in certain situations. But by adding only two logic gates to a basic cell, a much more useful memory device called a D-latch can be created. Latches are a type of temporary storage device that are normally placed in a different category from flip-flops. The main difference between latches and flip-flops is the method used for changing their state. A D-latch uses a basic cell for a memory element, but it only allows the value stored in memory to be changed (or programmed) when a timing control input is asserted. Thus, a D-latch has two inputs the timing control input and a data input. The timing control input, commonly called gate, or clock, or latch enable is used to coordinate when new data can be written into the memory element, and conversely, when data cannot be written. In Fig. 3 below, observe that when the Gate input is not asserted, S and R are driven to “1” and the output Q is determined by the value stored in the basic cell feedback loop (and so Q is showing the stored logic value). In the figure on the right, observe that when the Gate input is asserted, the D (for Data) input drives S and R to opposite levels, forcing a SET or RESET operation on the basic cell. By combining a timing control input and a data input that forces the basic cell to either SET or RESET, an useful memory device is created. The D-latch is widely used in all sorts of modern digital circuits.

Figure 3. D-Latch circuit

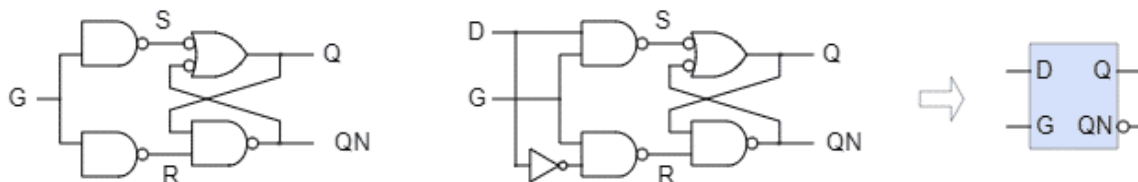


Figure 3. D-Latch circuit

A timing diagram for the D latch is shown below in Fig. 4. Note that when the Gate input is asserted, the output Q simply follows the input. But when the Gate input is not asserted, the output remembers the value present at D at the time the Gate signal was de-asserted.

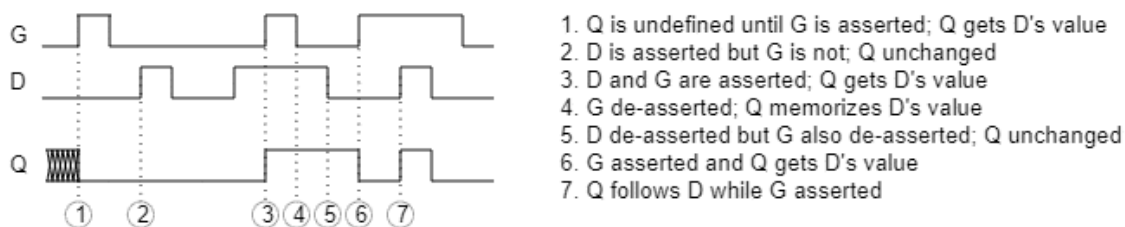
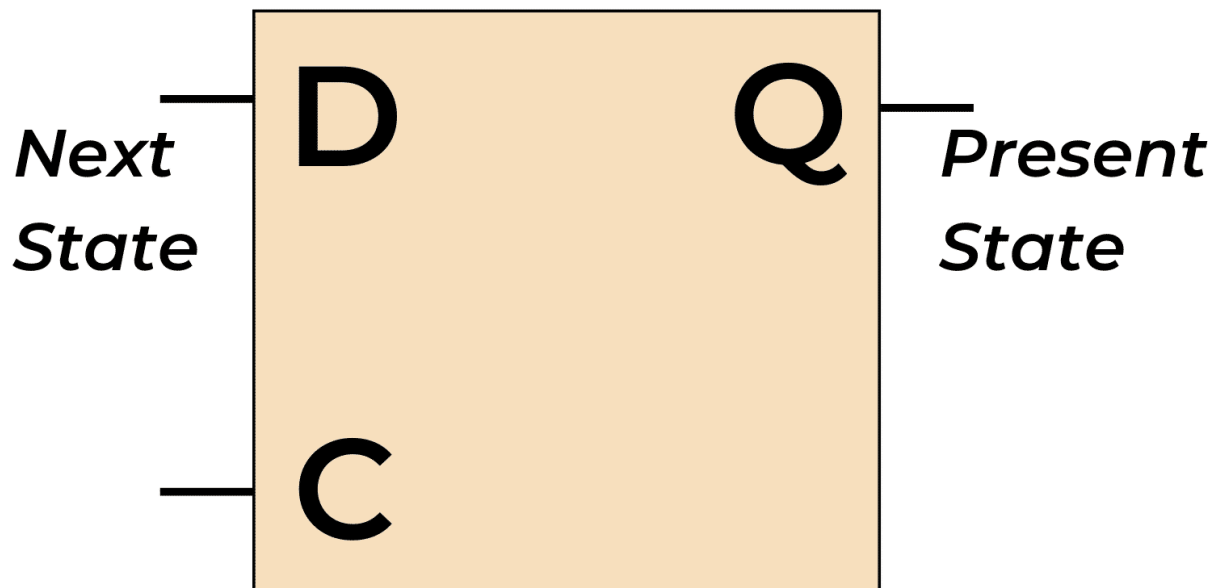


Figure 4. D-Latch timing



Data input to a memory device is called the *Next State*

Output from a memory device is called the *Present State*

Flip-Flops

All useful memory devices need at least two inputs – a data input signal and a timing signal to control when the data signal is memorized. The output of a memory device is often referred to as the present state, and the input is called the next state. At each new assertion of the timing signal, the next state becomes the present state. In a D-latch, the next state input is tied directly to the present state output whenever the timing control input is asserted. But when the timing control is de-asserted, the present state becomes fixed and holds the value that was present on the next state when the timing signal was de-asserted. A D flip-flop modifies the function of a D-latch in a fundamental and important way: the next state (or D input) can only be written into the memory on the edge (or transition) of the timing signal.

A D flip-flop (DFF) is one of the most fundamental memory devices. It typically has at least three inputs, including a data input (usually called D) that defines the next state, a timing control signal (usually called clock) that controls when the data is memorized, and a reset input that forces the memory to be reset to '0', regardless of the other two inputs. A DFF records (or registers) new data whenever an active clock edge occurs; the active edge can be either the rising edge or the falling edge. The clock signal is typically a regular, repeating square wave that continuously causes the DFF to memorize its data input signal at some frequency.

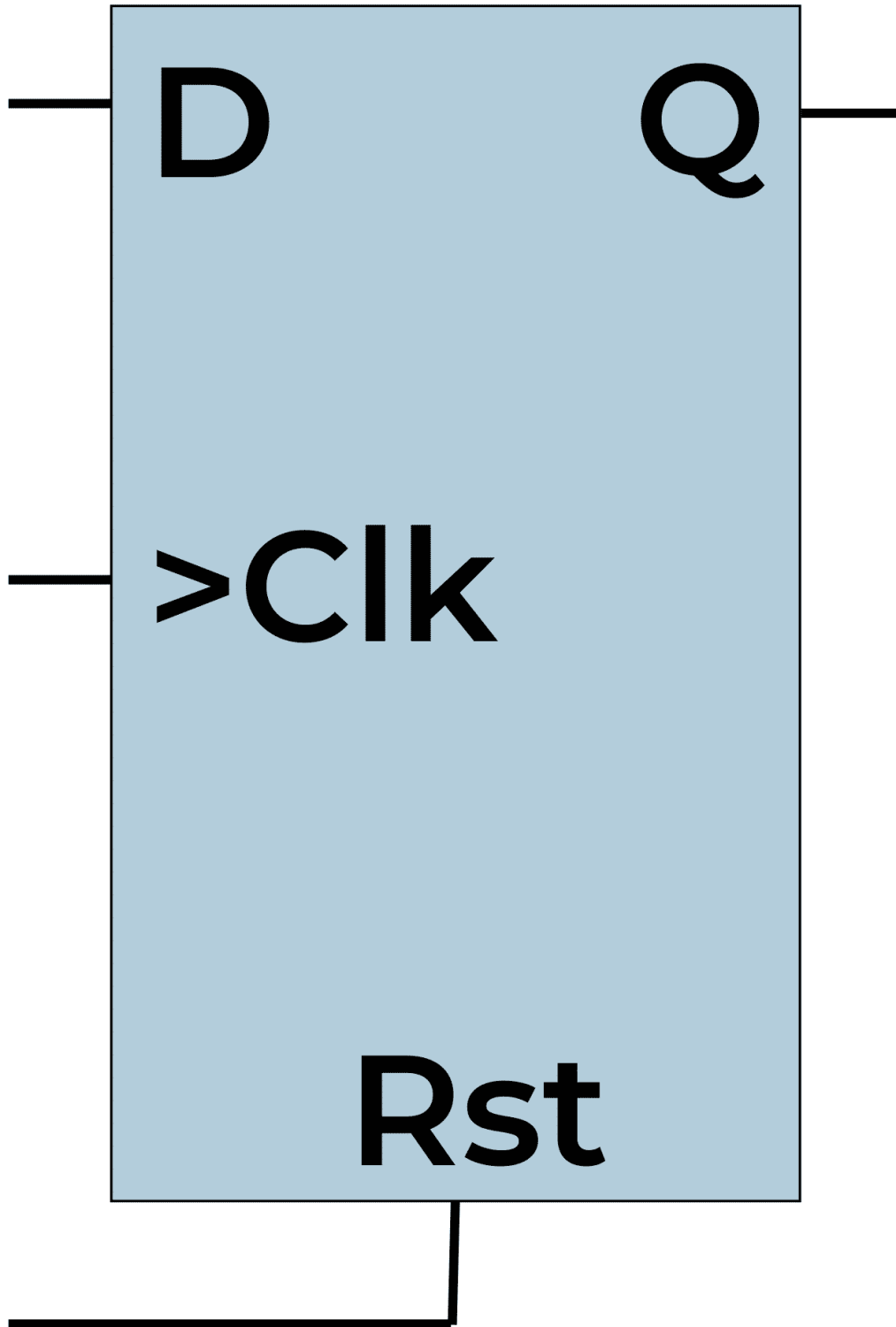


Image created using [Canva](#)

A rising-edge triggered (RET) DFF symbol uses a triangle to show that the flip-flop is edge-triggered; a falling-edge triggered (FET) DFF symbol uses the same triangle, but with a bubble

on the outside of the bounding box (just like any other asserted-low input). The timing diagram below in Fig. 3 illustrates RET DFF behavior. Note that the Q output changes only on the active edge of the clock, and the reset signal forces the output to '0' regardless of the other inputs.

Figure 3. Clock for a Flip-Flop.

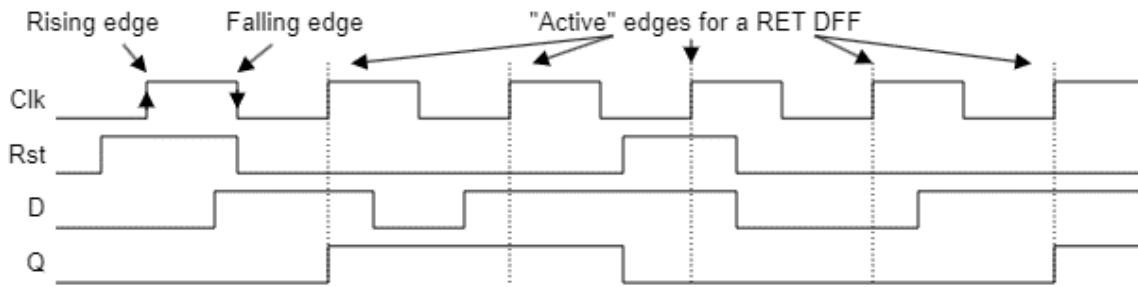


Figure 3. Clock for a Flip-Flop

As with the basic cells, a D flip-flop or D-latch can enter a metastable state if the data and control inputs are changed at exactly the same time. In a D-latch, the data must be stable when the control input is de-asserted. In a DFF, the data input must be stable for a time immediately before and immediately after the clock edge. If the data is not stable at the clock edge, a metastable state may be clocked into the memory element. If this happens, the memory element may not be able to immediately resolve to either low voltage or high voltage, and it may oscillate for a time. Thus, when designing circuits using edge-triggered flip-flops, it is important to ensure the data input is stable for adequate time prior to the clock edge (known as the setup time), and for a time after the clock edge (known as the hold time). Setup and hold times vary between several tens of picoseconds (for designs inside single IC's) to several nanoseconds (for designs using discrete logic chips).

Figure 4. Window of change.

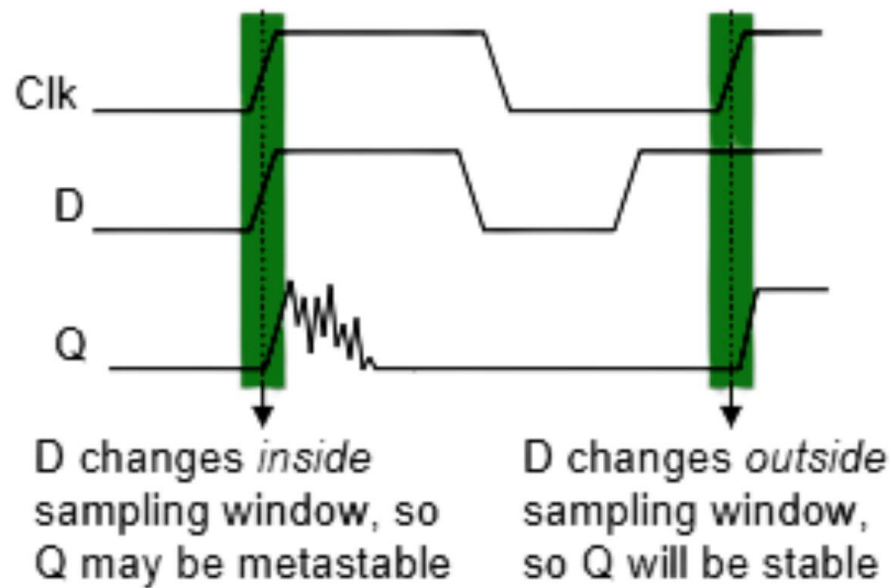
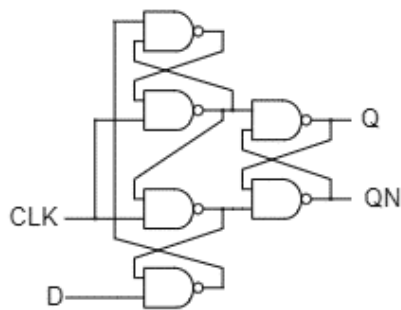


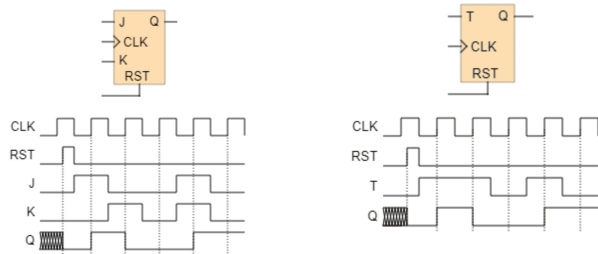
Figure 4. Window of Change



A schematic for a basic D flip-flop is shown in Fig. 5. Note the flip-flop is constructed from three basic cells. Two of the basic cells check for a rising edge, and the third memorizes the D input if a rising edge was detected. Several slightly different schematics can be found in various references, but any circuit called a DFF will exhibit the same behavior.

The DFF is the simplest and most useful edge-triggered memory device, and it can be used in any application that requires a flip-flop. Over the years, other types of flip-flops were designed and used as well, but they were really just D flip-flops with some added features. For example, the JK flip-flop used two inputs to direct state changes: the J input set the output to a '1'; the K input reset the output to a '0'; and if both J and K were asserted simultaneously, the output toggled between '1' and '0'. Another device, the T flip-flop, simply toggled between '1' and '0' on each successive clock edge so long as the T input is asserted. These other flip-flops

were commonly used in older digital systems (especially those built of discrete 7400 logic ICs) because their added features meant that fewer chips might be needed to construct a given circuit. But they offer no advantages to designs built using programmable chips (like the FPGA used on the Blackboard). Figure 6 below illustrates JK and T flip-flops.



JK Flip-flop - [click to enlarge](#)

T Flip-flop - [click to enlarge](#)

Registers

Registers are collections of one or more D-flip-flops that share a common clock and reset signals, but that have separate data in and data out signals. The most common register form, parallel-in parallel-out (or PIPO), are used to store data busses transporting binary numbers. They are commonly used to hold input and/or output data for data-path circuits, or to hold working state (or status) information about a circuit.

Serial-in, serial out (SISO) registers are commonly called “shift registers”. They store a single input at every clock edge, and then move the data down the chain of flip-flops until the data is presented at the output signal n clocks later. The SIPO register is a more general variation of the SISO register (all the bits are brought to output ports). It is also generally called a shift register, and it can be used in place of a SISO register. Shift registers are commonly used on the receiving end of serial communications systems (like USB, Ethernet, WIFI, etc.) to record serial-in data, and then to make that data available as a parallel data element.

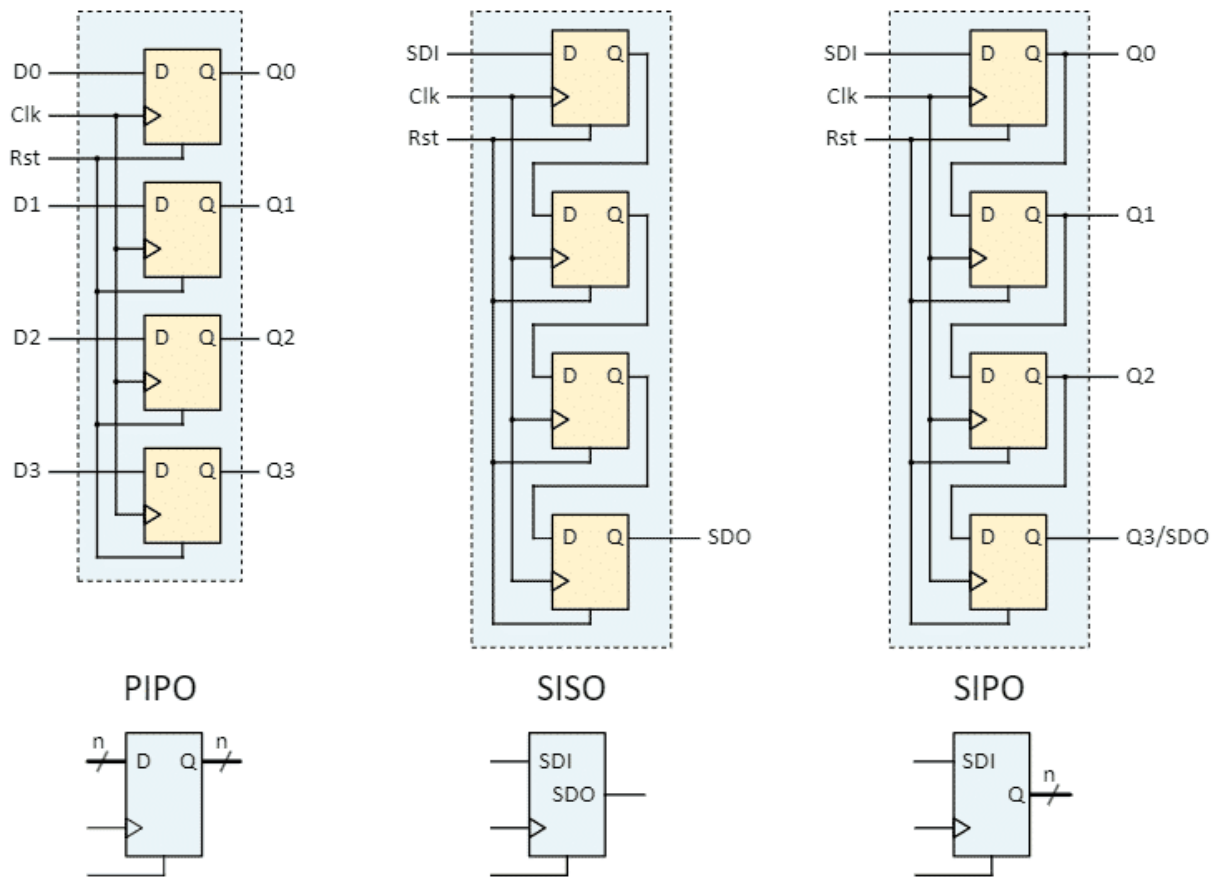


Figure 1. PIPO, SISO, and SIPO registers

The PSIO register includes a 2:1 mux on flip-flop inputs so that either serial or parallel data can be loaded. The parallel load option is commonly used on the transmitting end of serial communications systems, so parallel data can be loaded by the processor, and then sent out one bit at a time.

The universal register, as the name implies, can be used as a PIPO, SISO, SIPO, or PISO register.

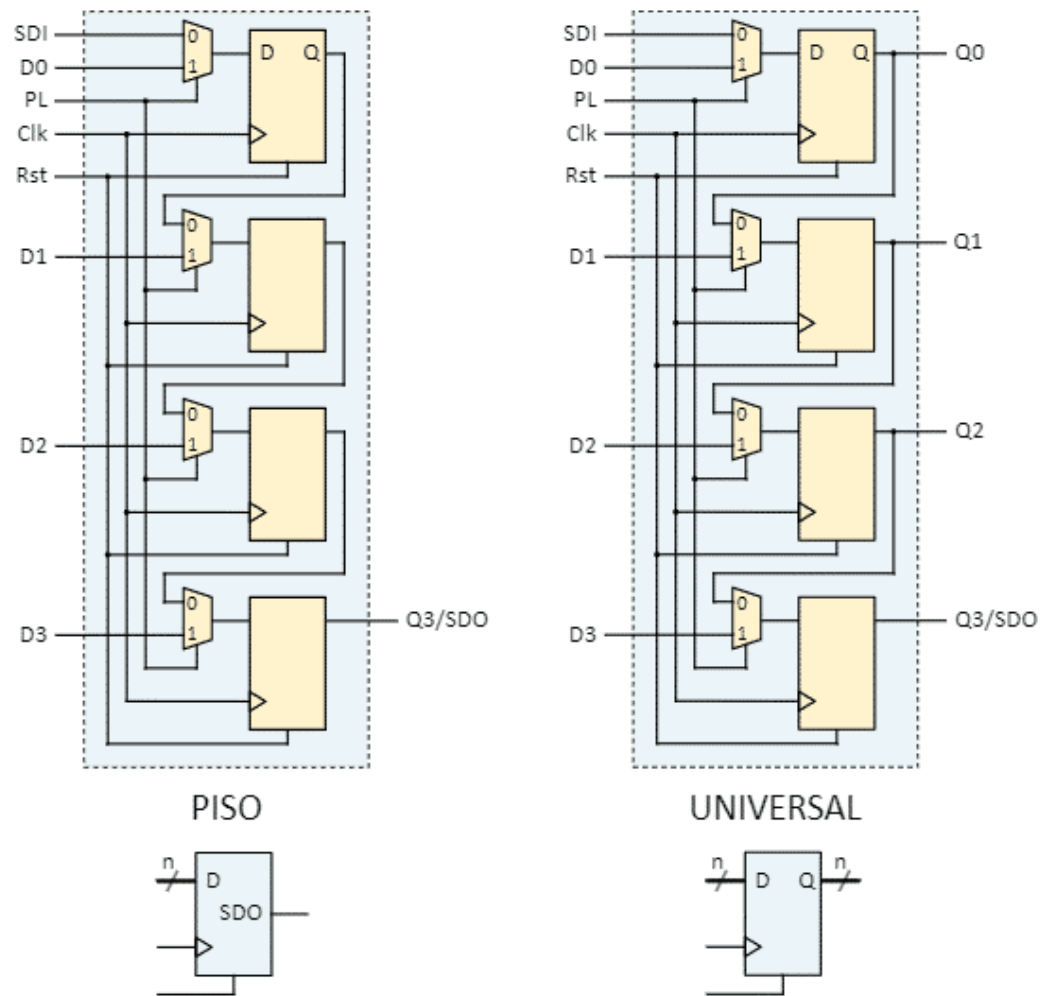
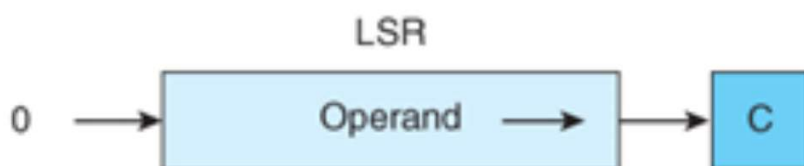
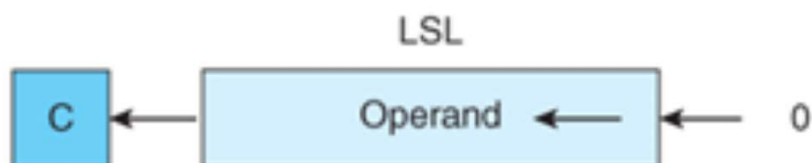


Figure 2. PISO and universal registers



Logical shift



Arithmetic shift



Rotate

Shift Registers move the bits stored in the register one or more positions to the right or left.

When considering the register contains a binary number:

- Moving to the right divides the binary number by 2.
- Moving to the left multiplies the binary number by 2.

Just like moving the decimal point in base 10 numbers multiplies or divides the number by 10, shifting provides a quick way to multiply or divide by multiples of 2. This is typically much faster than the normal multiply instruction that will be covered later in this course.

There are different types of shifts that can be used for various purposes. Almost all CPUs support these types of shifts. At this point we introduce a bit that is part of the Condition Code Register (CCR) or Current Processor Status Register (CPSR). Every CPU has a register of this type and it contains various status bits related to arithmetic operations or process status. The specific bit we are using related to shifts is the Carry bit.

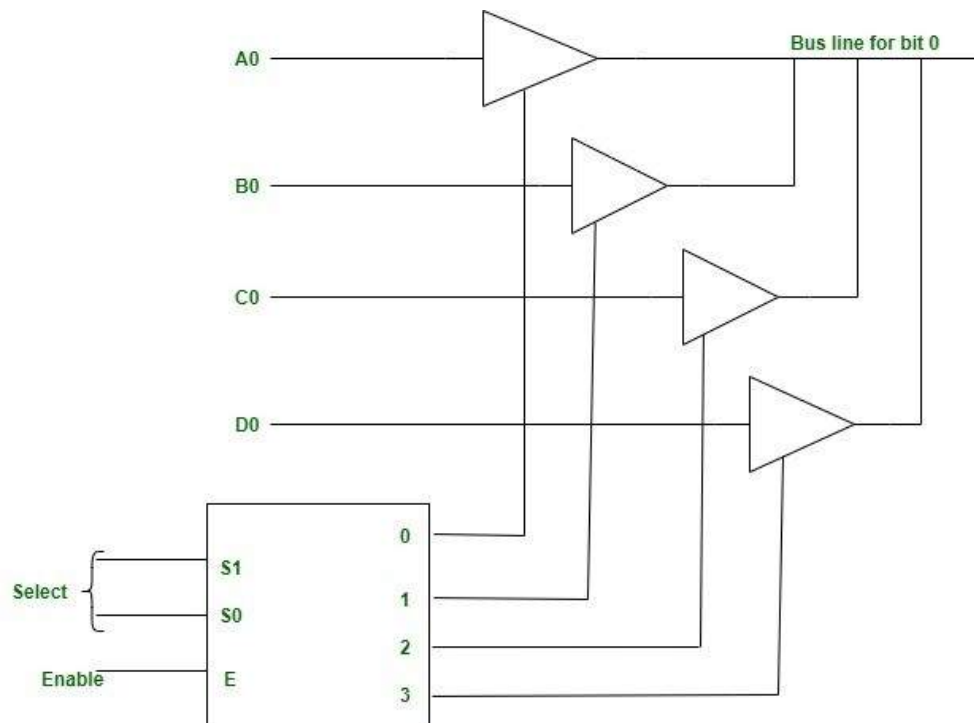
Depending on the shift type either the **most significant bit (MSB)** or the **least significant bit (LSB)** will be moved into the C-bit. To clarify the MSB is the leftmost bit and the LSB is the rightmost bit.

Transferring Data Between Registers

Introduction

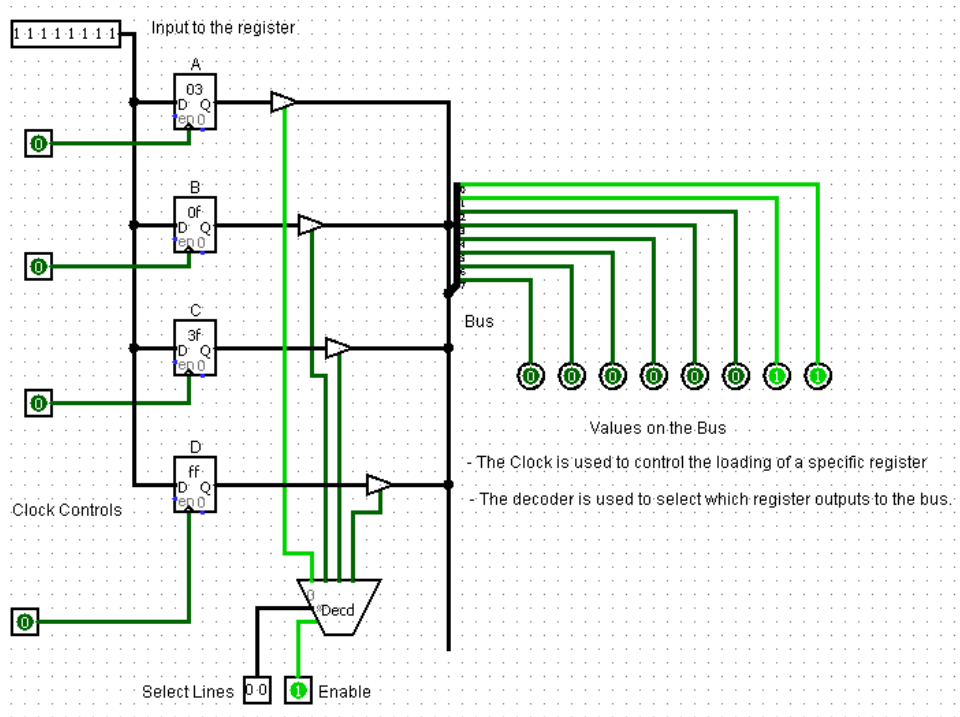
A large part of the CPU consists of connections of various registers via a bus or buses to transfer data. To make these connections you need:

- A bus (common connection points)
- Tri-state buffers
- Decoder
- Registers

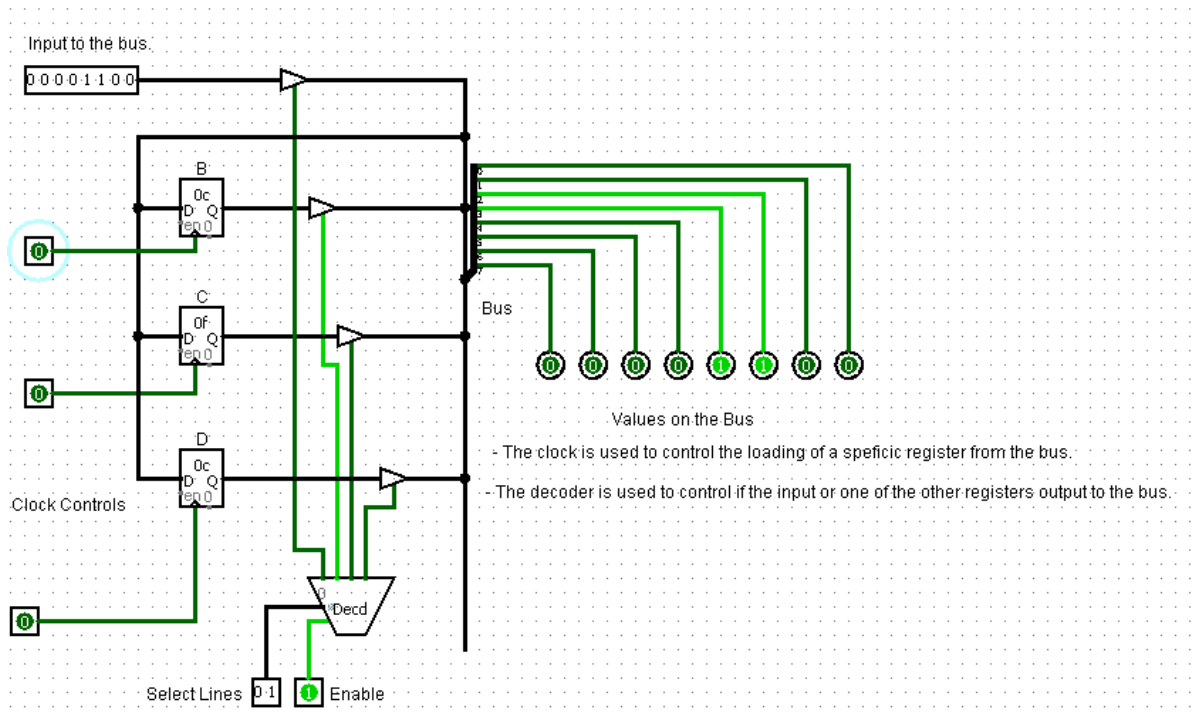


This Figure shows how the least significant bit (bit 0) of four different registers (A, B, C, D) are connected to tri-state buffers and a 2-to-4 decoder to control output of the bit to the bus.

The decoder is used to ensure that only one tri-state buffer outputs to bit 0 of the bus at one time. Disabling the 2-to-4 decoder turns off all four of the tri-state buffers.

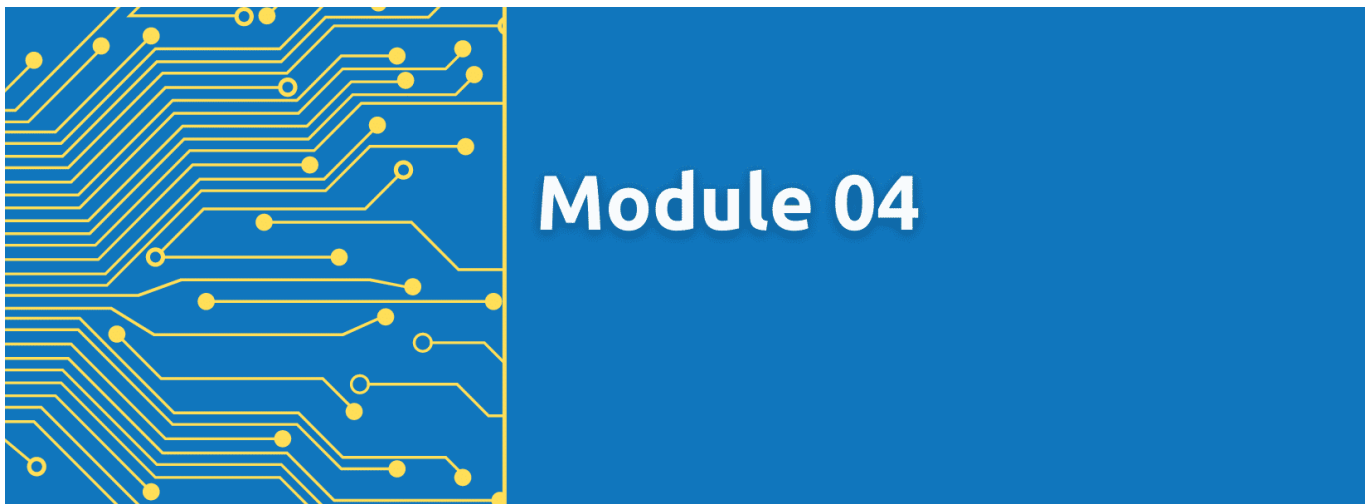


The expansion of the figure above to 8-bit registers is shown in this figure.



How do registers transfer data? With a minor change to the circuit above registers can read from the bus and transfer (copy) data from registers.

Addition, Subtraction, Multiplication & Division



Introduction

From the material we have covered in this course so far it should be obvious that the computer only has the ability to store and manipulate 0's and 1's. This module will explain how just using these 1's and 0's that integer arithmetic can be performed.

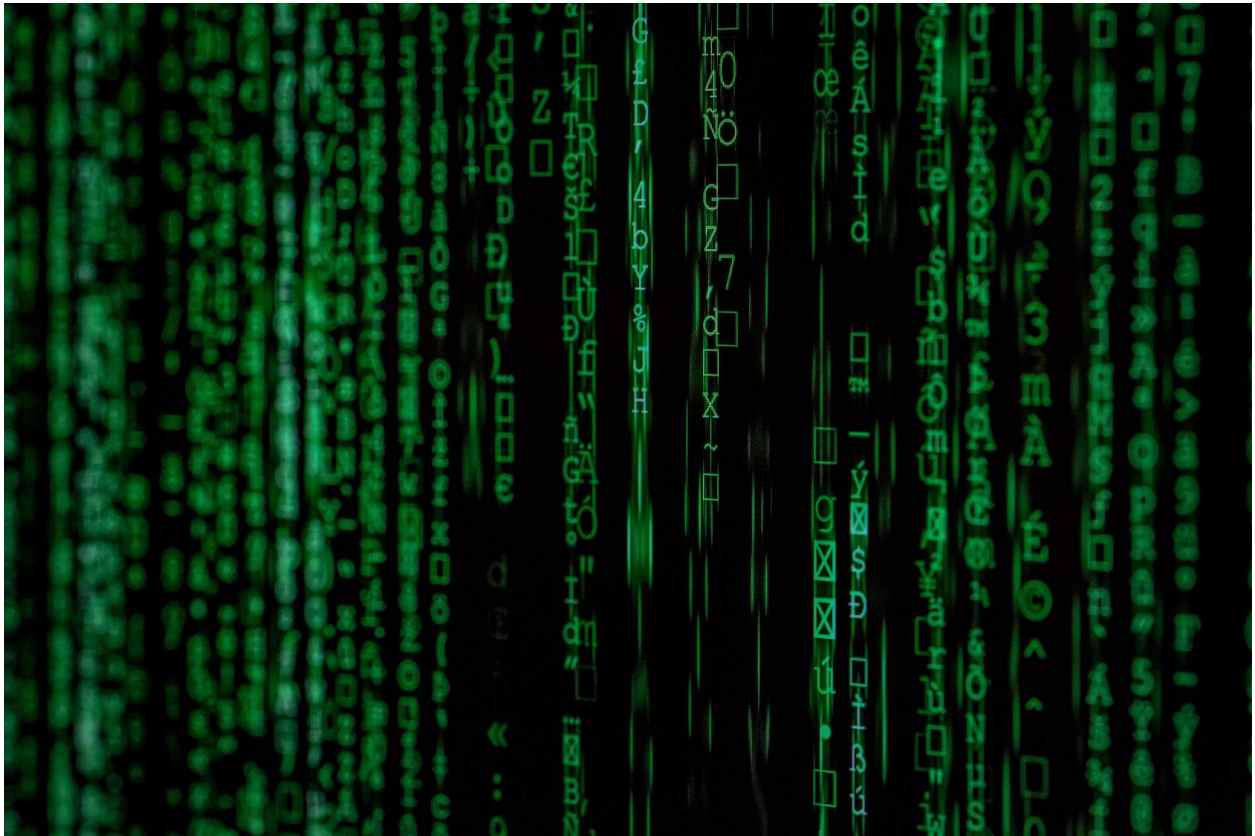


Image from Articulate Content Library

Objectives

From this module students should learn:

- How to convert integers to and from two's complement representation
- Details on the hardware design to perform binary addition and subtraction
- Overview of the hardware needed to perform binary multiplication and division

Students should:

- Read the following material
- Watch the associated video lecture
- Take the Canvas quiz on this topic

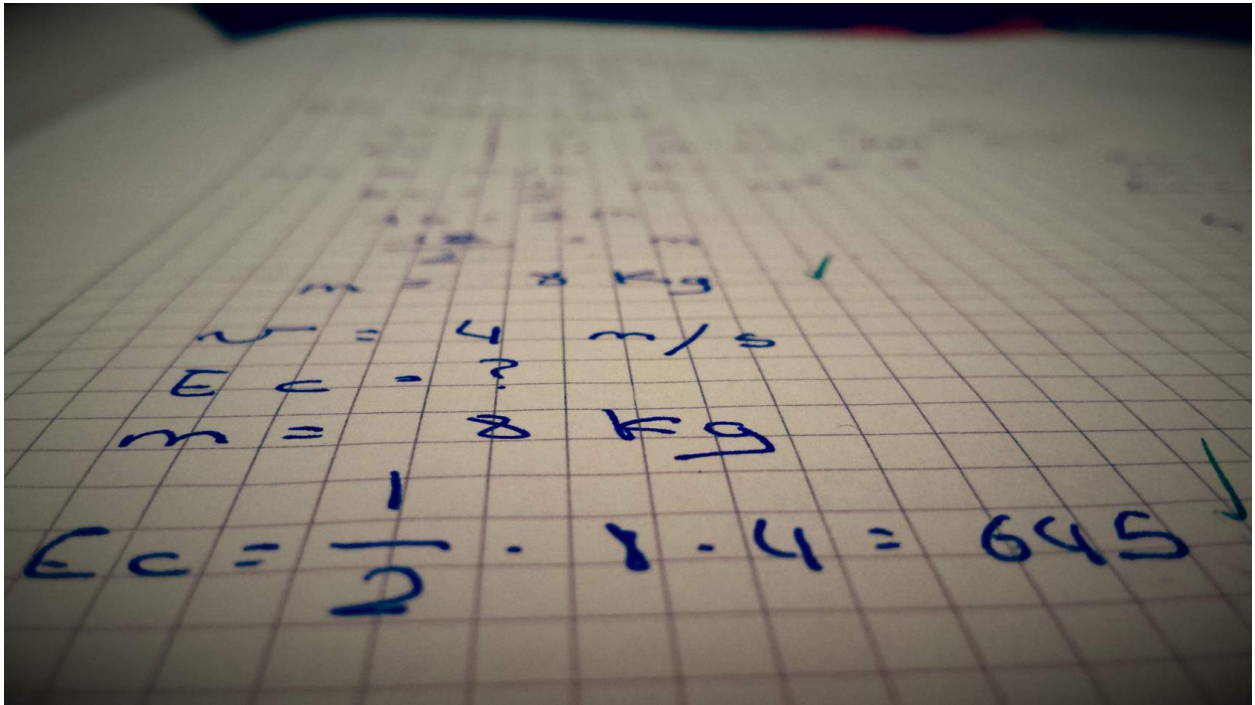
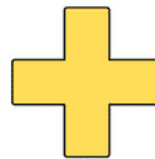


Image from Articulate Content Library

Students should have already learned about Integer representation in CS524 and how to convert from Decimal (base 10) to Binary (base 2) to Hexadecimal (base 16) and Octal (base 8).

Here are several sites that show how to perform these base conversions:

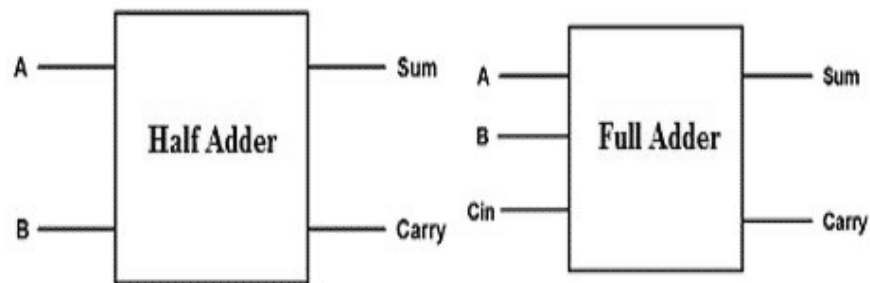
- http://www.pages.drexel.edu/~stn38/natter_lab6.html
- <https://www.csee.umbc.edu/courses/104/fall05/ratsimor/lectures/L4/sbinary.html>
- <https://courses.cs.vt.edu/~csonline/NumberSystems/Lessons/BinaryNumbers/index.html>



Addition

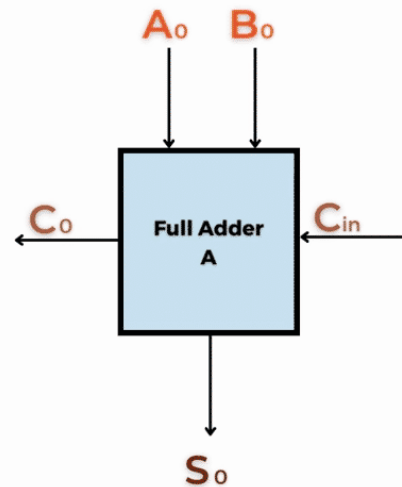
In Module 02 we showed the hardware needed to perform addition of two binary bits, which produces a Sum and Carry outputs (Half-adder), and the addition of three binary bits, which produces a Sum and Carry outputs (Full-adder).

At this point (like a lot of other circuits) we are no longer concerned about the gate level details so we will just be using a block diagram for the adders with only show the inputs and outputs.



Addition of binary numbers of any length can be performed by having a Full Adder for each bit or a Half Adder for the least significant bit and the rest is made up of Full Adders.

The following video diagram shows using four full adders to add two 4-bit inputs producing a 4-bit sum and a Carry out. To work properly the Cin input has to be set to 0 or to ground.



4-bit Ripple Carry Adder

Keep in mind the **Propagation Delay** discussed in Module 02. For simplicity we will assume the propagation delay (or gate delay) for each full adder is 10 nanoseconds. This means that once the inputs for the full adder is set then it takes 10 nanoseconds for the correct sum and carry to be produced.

So it takes 10 nanoseconds to get C_0 and S_0 correct then another 10 nanoseconds to get C_1 and S_1 correct then another 10 nanoseconds to get C_2 and S_2 then another 10 nanoseconds to get C_3 and S_3 correct. So this circuit takes a total of 40 nanoseconds to produce the correct final sum and carry bit.

This is the reason this adder is called a **Ripple Carry Adder**. The carry out from the lower bits has to ripple through all the other adders until the correct sum and carry is produced.

Most modern CPUs will add two 32-bit or 64-bit binary numbers. If they are using this adder circuit it will take 320 to 640 nanoseconds to perform the sum. This is FOREVER in CPU speeds. As mentioned in Module 02 this hardware is very slow but it is cheap. Modern CPUs use what is called a **carry look ahead adder** which changes the design so that there is no propagation delay. There is a constant delay regardless of the number of bits that are being added. The design of the carry look ahead adder is beyond the scope of this course but students should be aware of its existence and use in modern CPUs.

01100111 11111100 01111101 01111101 11011001 11001010 11101000 10011110 11101111 10100000 10010111 00100001 00010111 01000011 00011100
11100010 10011100 01100010 01011111 11010011 10001100 10001101 01110101 10010000 01011011 01110000 10111110 10110010 10110101 10011001
11001011 01001001 11100010 01011011 11000101 10001011 01000011 01000111 00011110 01001101 00000010 11100100 00011010 10010010 10000010
00100011 00011111 00001000 01011011 01001111 10100101 01111001 00010110 00110101 01101010 010001101101111 10010011 10000000 01100110
10111001 01010010 01100011 00000100 00001011 10011101 01001000 10011010 11010100 10001101 01100111 00000101
00100011 11010010 10000111 00011000 10011101 01111001 01001000 10011010 11010000 10011001 11010000 10011000 11100001 10100100 10001101 01100001
01000101 01010001 00100101 11110001 00110111 00000001 10011100 01001000 01001000 01100001 10011100 01100111 11111100 01111101 01111101 11011001
11001010 11101000 10011110 11011111 10100000 01011011 10011100 01111101 01111101 10011100 01100010 01011111 11010011
10001100 10001101 01110101 10010000 01011011 11111100 01100010 01100010 01011101 10011100 01001001 11100010 01011101 11000101
10001011 01000011 01000111 00011110 01001011 11111100 01100010 01011101 10011100 01001001 11100010 01011101 11000101
10100101 01111001 00111001 11001011 01001011 11111100 01100010 01011101 10011100 01001001 11100010 01011101 11000101
10011100 00101000 00010110 00111010 10001010 01001001 11100010 01011101 10011100 01001001 11100010 01011101 11000101
01111011 01011010 10001100 10101010 01011111 00011111 00001000 01011011 11000101 11100010 01011101 11000101
00100100 11010110 10101000 01011011 01001011 00011111 00001000 01011011 11000101 11100010 01011101 11000101
10010111 00100001 00010110 10010010 10000011 11010010 10000111 00011000 10011100 00001001 01011101 11000101
0100011011101111 10010011 10000000 00000011 11010010 10000111 00011000 10011100 00001001 01011101 11000101
00011001 11010100 10001101 01101011 01001011 00000001 00010110 00011000 10011100 00001001 01011101 11000101
11100001 10100100 01001000 00110111 00000001 01010001 01010001 00100101 11110001 00110111 11000101 10101100 01011101 00011001
01100111 11111100 01111101 01111100 11001010 11101000 10011110 11101111 10100000 00000001 00010111 01000011 00011100
11100010 10011100 01100010 01011111 10001100 10001101 01110101 10010000 01011011 11000101 10101100 10110101 10011001
11001011 01001001 11100010 01011011 10001100 10001101 01110101 10010000 01011011 11000101 10101100 10010010 10000010
00100011 00011111 00001000 01011011 10001101 01110101 10010000 01011011 11000101 10000000 01100110
10111001 01010010 01100011 00000100 00010111 01000011 01000011 01000111 00011100 01011010 10001101 01100111 00000101
00100011 11010010 10000111 00011000 00000100 00010111 01000011 01000111 00011100 01011010 10001101 01100111 00000101
01000101 01010001 00100101 11110001 11110001 01110001 01110001 01110001 01110001 01110001 01110001 01110001 01110001
11001010 11101000 10011110 11101111 00011100 00111001 11001010 10000110 10001100 01011111 11010011
10001100 10001101 01110101 10010000 01011011 01010000 00010110 00111001 11001010 10001100 11101110 11010011
10001011 01000011 01000111 00011110 01010000 00010110 00111001 11001010 10001100 10011011 00000100 00001011
10100101 01111001 00111001 11001011 01011010 10001100 10101111 00011100 01010001 01010001 00000100 00001011
10011100 00101000 00010110 00111010 10000100 00000100 00000100 00000100 00000100 00000100 00000100 00000100
01111011 01011010 10001100 10101010 11101100 10000001 00010111 01010000 10011100 11101100 11101111 11010000
00000010 11100100 00011010 10010010 10000010 00100011 00011111 11000101 01110001 10100101 01110001 11001011 01101010
010001011011111 10010011 10000000 01100110 10111001 01010010 01100011 00000100 00001011 10011100 00101000 00010110 00111010 10000100
10011001 11010100 10001101 01100111 00000101 00100011 11010010 10000111 00011001 01111011 01011010 10001100 10101010 11101100
11100001 10100100 01001000 00110111 00100001 01000101 01010001 00100101 11110001 00110111 00100100 11010110 10101100 01011111 00111001

[Image](#) licensed under [CC-0/Public Domain](#)

Negatives

Unlike the decimal system, binary does not have a negation sign "-". Instead, there are several different ways of expressing negative binary numbers. What you have to keep in mind is one more bit is needed beyond what is required to store the magnitude of the number in order to identify if the number is a positive number (0) or a negative number (1). The most significant bit (MSB) is used to identify if the number is positive or negative.

One other point - Looking at the contents of a register or memory tells you nothing about the type of data that is being stored there. All you know is that it is a collection of 1's and 0's. You have to know the type of data that is being stored there in order to then determine what is being stored there. Once you are told that then you can determine what those collections of 1's and 0's really represent.

Sign Magnitude, One's Complement & Two's Complement

Expand the accordion for more information.

Sign-magnitude —

Sign-magnitude

Sign-magnitude revolves around the most significant bit (MSB).

This bit (the leftmost one) is made either 0 for positive numbers or 1 for negative numbers. The rest of the bits are used to determine the magnitude of the number.

For example:

- $1001 = (-)(001) = -1$
- $0001 = (+)(001) = 1$
- $11001 = (-)(1001) = -9$
- $111 = (-)(11) = -3$
- $010101 = (+)(10101) = 21$

One's Complement —

One's Complement

To represent a negative binary number using One's Complement, flip all the bits.

That's basically it.

For example:

For -7, begin with the binary of +7: 0111, and change the 1s to 0s and the 0s to 1s.

- $0111 \rightarrow 1000$. (-7 in One's Complement)
- $1010 \rightarrow 0101$. (5 in One's Complement)
- $0011 \rightarrow 1100$. (-4 in One's Complement)
- $01101 \rightarrow 10010$. (-13 in One's Complement)

Another way that I like to think of this is as "absence of". So for -7, the MSB will be 1 followed by an absence of 7, e.g. "000".

For -13, the MSB is 1 followed by "absence of" 13, e.g. "0010".

Two's Complement —

Two's Complement

Similar steps to One's Complement:

- Flip the bits

- Add 1 to the Least significant bit

For example:

- $0111 \rightarrow 1001$. (-7 in Two's Complement)
- $1010 \rightarrow 0110$. (5 in Two's Complement)
- $0011 \rightarrow 1101$. (-4 in Two's Complement)
- $01101 \rightarrow 10010$. (-13 in Two's Complement)

Here is a helpful link to learn more about negative notations in binary:



Back to the hardware for the subtractor.



Take a look back the XOR basis gate. You will notice that if one of the inputs is a constant 0 the output is whatever the other input is. If one of the inputs is a constant 1 the output is the complement of the other input. The XOR allows you to complement or not complement on command. Making the

complement or not complement input bit to the XOR to also be the Cin turns the 1's complement of Input B into 2's complement.

Multiplication

Hardware multipliers, based directly on adder architectures, have become indispensable in modern computers. Multiplier circuits are modeled after the shift and add algorithm as shown below. In this algorithm, one partial product is created for each bit in the multiplier; the first partial product is created by the LSB of the multiplier, the second partial product is created by the second bit in the multiplier, etc. Partial products are a copy of the multiplicand if the corresponding multiplier bit is a '1', and all 0's if the corresponding multiplier bit is '0'. Each successive partial product is shifted one bit position to the left.

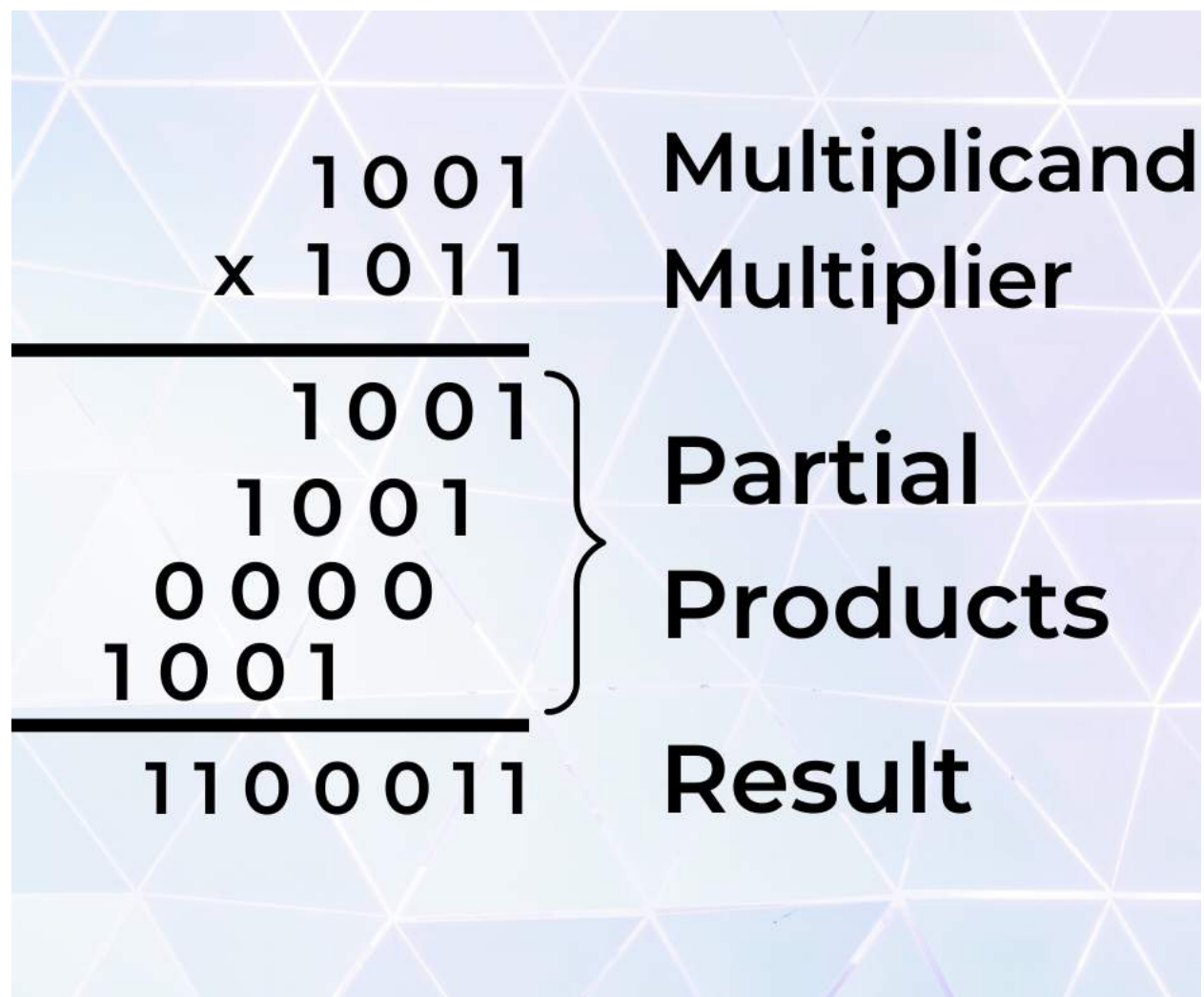


Figure 1. Binary multiplication

This specific multiplication example is recast in a generalized example on the left below. Each input, partial product digit, and result have been given a logical name, and these same names are used as signal names in the circuit schematics. By comparing the signal names in the multiplication example with the schematics, the behavior of the multiply circuit can be confirmed.

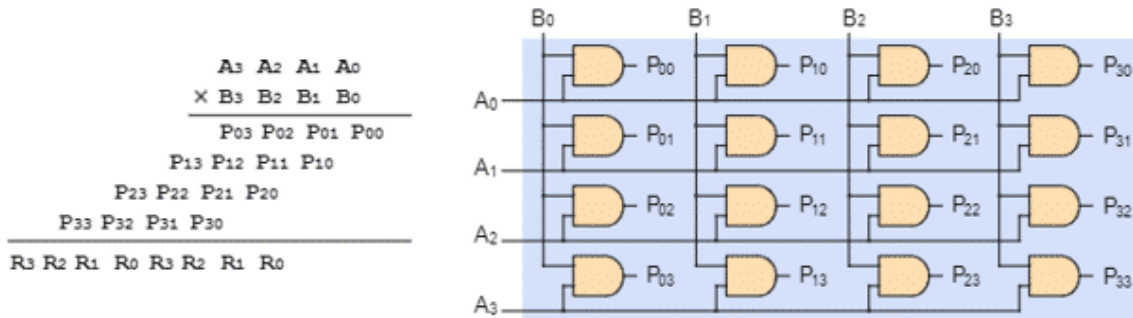


Figure 2a. Hardware multiplier

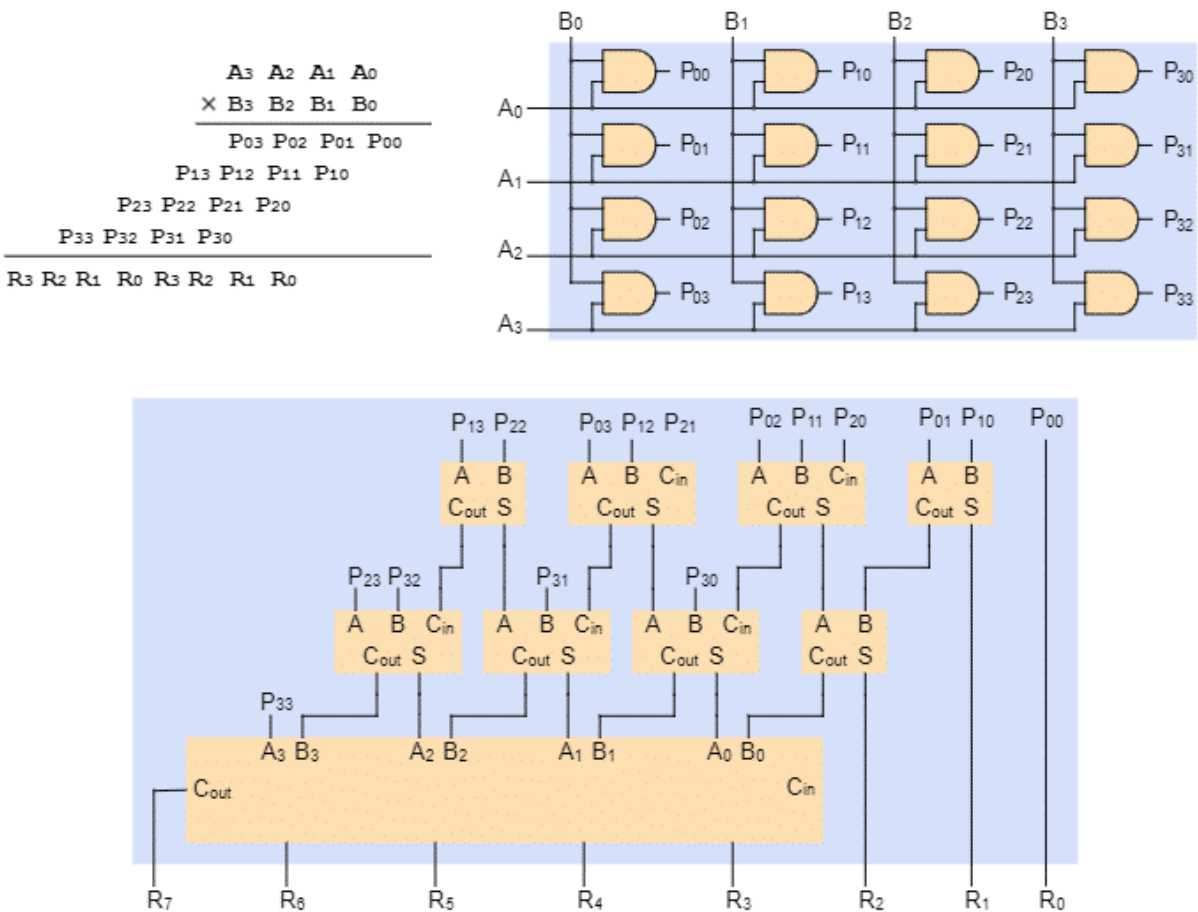


Figure 2b. Hardware multiplier

In this circuit above, each bit in the multiplier is AND'ed with each bit in the multiplicand to form the corresponding partial product bits. The partial product bits are fed to an array of full adders (and half adders where appropriate), with the adders shifted to the left as indicated by the multiplication example. The final partial products are added with a **carry look ahead adder** circuit. Note that some full-adder circuits bring signal values into the carry-in inputs (instead of carries from the neighboring stage). This is a valid use of the full-adder circuit; the full adder simply adds any three bits applied to its inputs. You are encouraged to work through a few examples on your own to confirm the adder array and **carry look ahead adder** work together to properly sum the partial products. In the lab project, you are asked to implement a multiplier circuit.

As the number of multiplier and multiplicand bits increase, so does the number of adder stages required in the multiplier circuit. It is possible to develop a faster adding array for use in

a multiplier by follow a similar line of reasoning as was used in the development of the **carry look ahead adder** circuit.

Important Ideas:

1

Multiplier circuits are modeled after the shift and add algorithm. In this algorithm, one partial product is created for each bit in the multiplier the first partial product is created by the LSB of the multiplier, the second partial product is created by the second bit in the multiplier, etc.

2

Each input, partial product digit, and result have been given a logical name, and these same names are used as signal names in the circuit schematics.

3

As the number of multiplier and multiplicand bits increase, so does the number of adder stages required in the multiplier circuit.

Information

from: <https://www.realdigital.org/doc/42f4of5d5d502c3672477a2bbd78b697>

Faster Multiplication

As discussed in the Register section shifting a register to the left multiplies the register by two (2). This can be used to perform multiplication much faster than using the multiplication circuit above. The drawback is you have to know one of the values that you are multiplying so you can write the code for the multiplication.

For example with this calculation:

It takes the unknown value in X and multiplies it by 16.

Shifting X four (4) positions to the left multiplies X by 16. Why? $2^4 = 16$. Shifting in a modern CPU only takes one-clock cycle because of what is called a barrel shifter. The input value can be shifted any number of positions to the right or left using only one-clock cycle. The basis for this hardware is a

decoder and a bunch of tristate buffers. This takes a lot of hardware but is very fast. Showing once again that fast is expensive.



GIF created using [Canva](#)

This method works very well for powers of 2 but what about other values like 17 or 25?



Image created using [Canva](#)

Image created using [Canva](#)

Shift X 4 positions to the left $X*16$ Shift X 4 positions to the left $X*16$
 Shift X 3 positions to the left $X*8$
 Add X back in $X*16 + X = X*17$ Add $X*16 + X * 8 + X = X*25$

Table Look-up Multiplication

Another method for performing multiplication is looking up the value in a table. Using a simple example of multiplication of two four-bit numbers (or two single hexadecimal digits) produces a multiplication table that looks like:

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

The trade-off for this method is the cost of memory vs speed improvement.

To do multiplications with larger numbers the table can be expanded to two hexadecimal digits, which requires much more memory for storage.

There is a different method of performing larger multiplications by using smaller tables that involve breaking up the two numbers and then performing shifts and adds.

Here is an 8-bit example and the multiplication table above.

$X * Y = X_1X_0 * Y_1Y_0$ where X_n is a four-bit value.

Image created using [Canva](#)

Doing this multiplication we get

$X_1 * Y_1 * 4 * 4 + X_1 * Y_0 * 4 + X_0 * Y_1 * 4 + X_0 * Y_0$

Image created using [Canva](#) - PDF of each image is available below

The $X_n * Y_n$ can be looked-up in the table, and then shifts can be used to get the magnitude of the multiplication correct. Then add all the values together.

This method of multiplication is not a common practice but does provide the ability to perform the multiplication for CPUs that did not have multiplication instruction.



$X Y = X_1X_0 Y_1Y_0$ where X_n is a four-bit value..pdf

6 KB



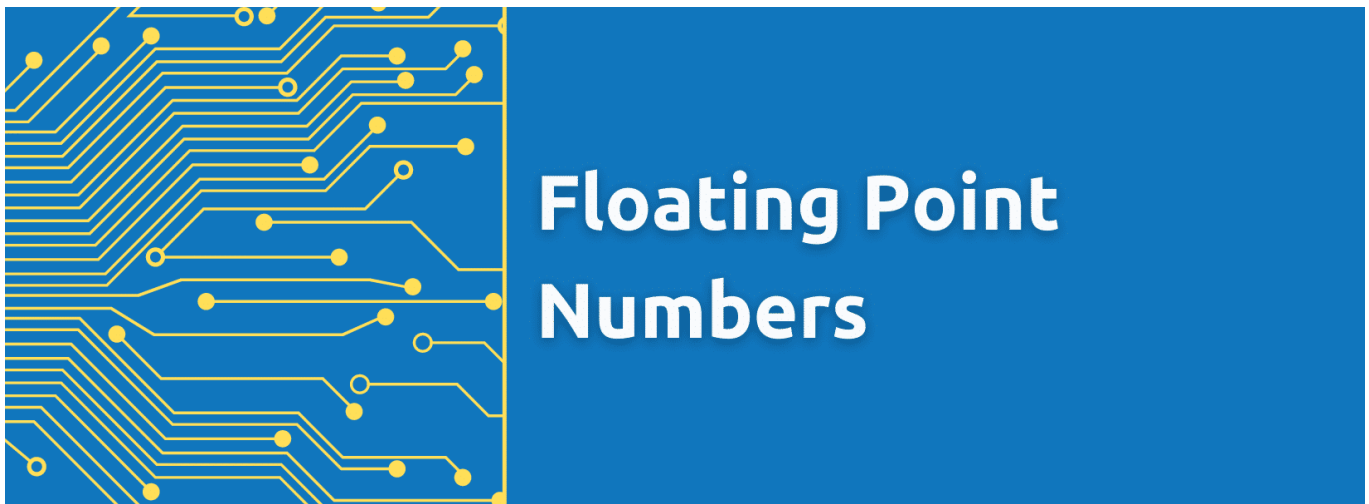
Division

The gate level details for division are beyond the scope of this course. This document can provide some additional details on possible algorithms.

PLACEHOLDER FOR STANFORDCHAP5DIVISION

The gate level details for division are beyond the scope of this course. This document can provide some additional details on possible algorithms.

Floating Point Numbers



Introduction

There are times when the numbers we are attempting to represent are too large or small to be represented with fixed point notation. However, scientific notation can be applied to any base, including binary. In this chapter, we'll discuss the three parts of scientific notation: sign, mantissa, and exponent. We'll also discuss floating point binary and IEEE 754 notation.

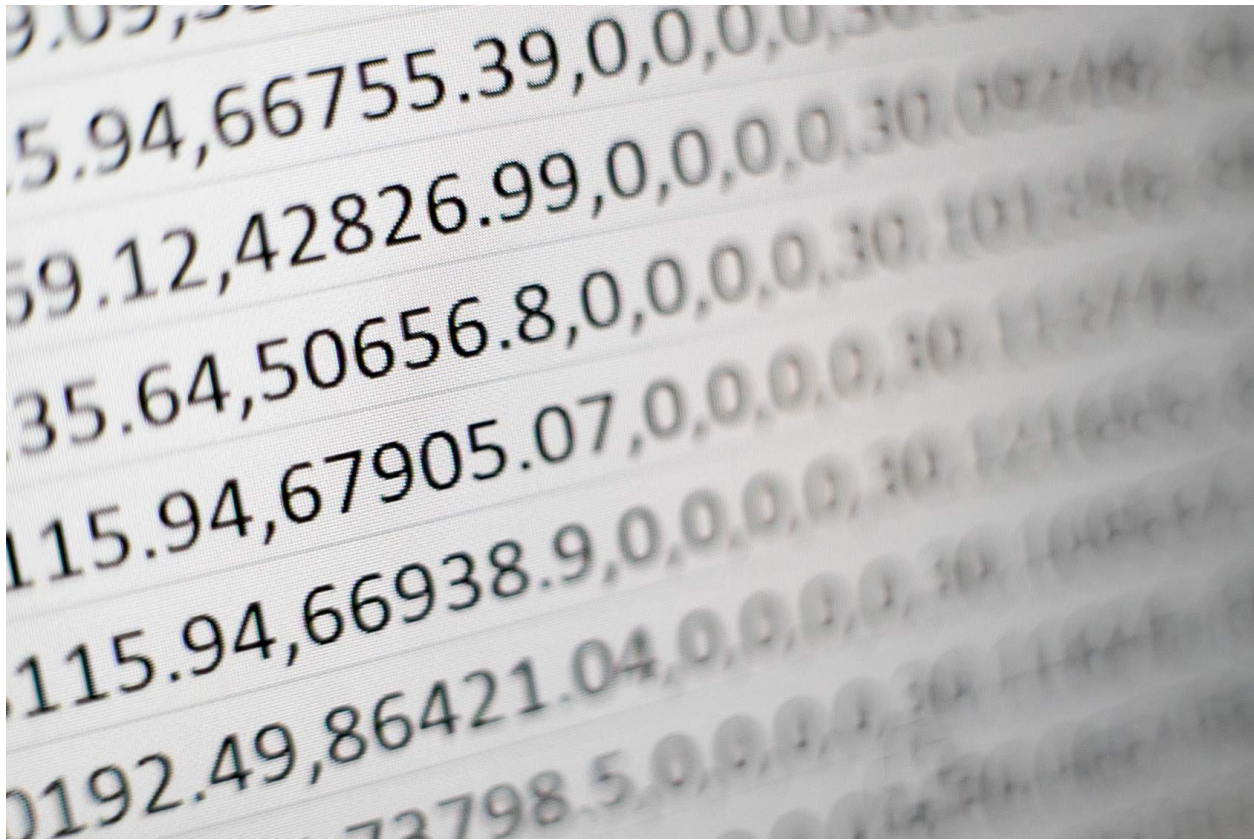


Image from Articulate Content Library

Objectives

From this module students should learn:

- How to represent floating point numbers in binary
- Issues with floating point representation and calculations
- Determining the amount of error in calculations
- Other functions calculation methods
- How to avoid using floating point

Students should:

- Read the following material
- Watch the associated video lecture
- Take the Canvas quiz on this topic



Image from Articulate Content Library

Overview

Before digging into this topic there are some things that need to be made clear.

1. A certain range of Integers can be represented exactly in binary. The range is $-2^{(n-1)}$ to $+2^{(n-1)} - 1$. Where n is the number of bits used for the representation. For example: if $n=8$ then range is -128 to $+127$.
2. At some point the computer will run out of storage for representation of just the Integers. Last time I checked there are still an infinite number of Integers. Even though modern computers have petabytes worth of storage (2^{50}) that is still way smaller than infinity.
3. Some of our scientific or engineering calculations are not really concerned about keeping track of 100 or more significant digits. Ten or twenty significant digits may only really be needed to get the results we are looking for.

4. Some calculations must be exact. For example, currency calculations need to be exact to the penny.
5. I now have an idea about the hardware needed for addition, subtraction, multiplication and division but what about things like $\sin$, $\cos$, $\arctan$, e^x , etc?

We will try to answer this question in the module: What does a computer programmer really need to know about these things as they go about doing their job?

Introduction to Floating Point Binary and IEEE 754 Notation

Sometimes numbers we're trying to represent are too large or too small to be represented with fixed-point notation. For example, it isn't practical to write decimal numbers such as the Avogadro number, six zero two followed by twenty-one zeros, or the width of a hydrogen atom in meters, 0.000000000106. That's why we have scientific notation.

In scientific notation, the decimal value -342,000,000 is represented as -3.42×10^8 . Note that there are three components to this representation: the sign, the mantissa (the part that represents the significant digits of the number), and an integer exponent. The exponent represents the number of times we multiply or divide the mantissa by the base. In proper scientific notation, the integer portion of the mantissa must be greater than or equal to one and less than ten, i.e., a single non-zero integer digit.

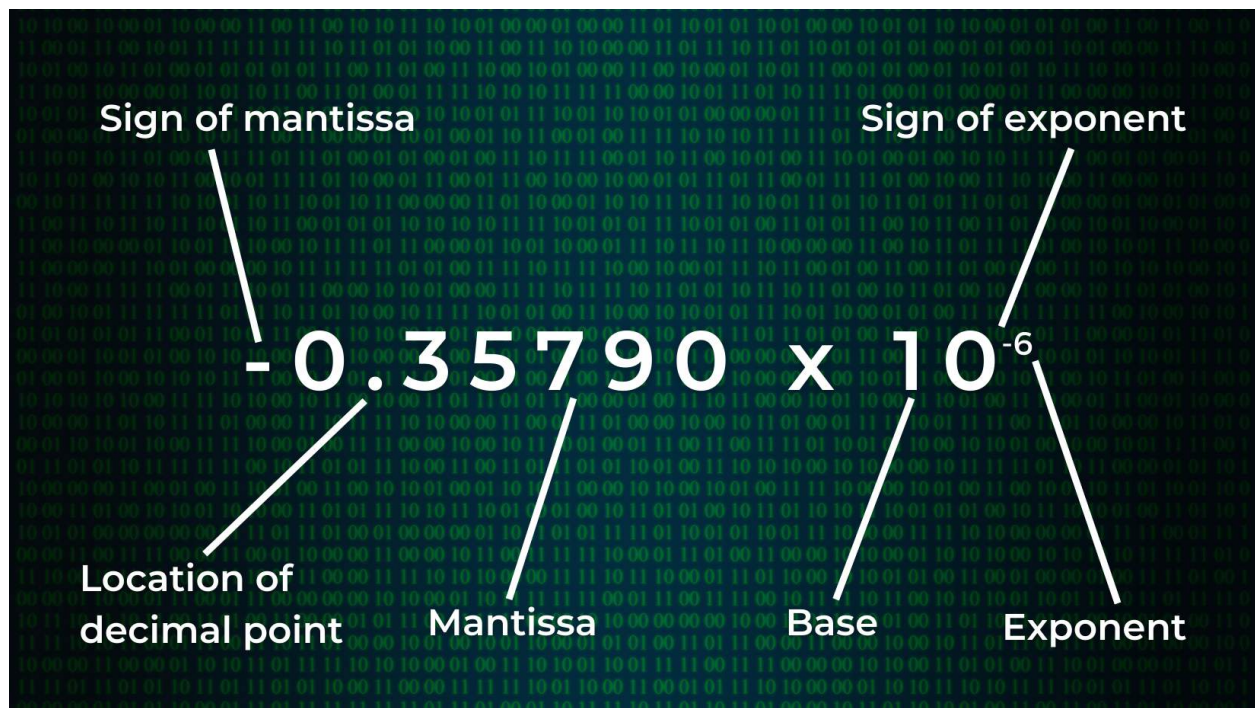


Image created using [Canva](#)

Scientific notation can be applied to any base, including binary. Changing the base not only changes the numerals available to represent the mantissa, it also changes the base. Remember that when you multiply or divide a number by its base, it has the effect of moving the point right or left. That means the exponent simply represents the number of positions right or left that the point must be moved. An exponent of five means that the point must be moved five positions to the right. An exponent of negative three means that the point must be moved three positions to the left.

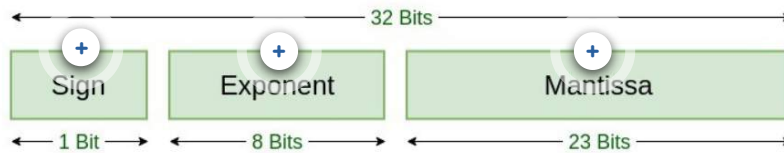
Most programming languages require variables to be declared with a type before using them. In Java, for example, a variable declared as an int is stored in 32-bit twos complement representation. For a variable meant to accommodate a real number, programming languages such as Java declare them using keywords like float or double. For these formats, computers rely on floating point notation in base-2.

NOTE: *Scientific notation* is NOT the same thing as ***floating point notation*** (hence the different names).

Just like decimal, floating point notation in binary has a sign identifying the number as positive or negative, the mantissa with a single non-zero digit to the left of the binary point and the fraction to the right, and an exponent that in this case uses two as the multiplier to move the binary point right or left. For example, the binary number 10001.011 would be represented in base-2 floating point notation as 1.0001011×2^4 . Note that in binary, there is only one option for the single non-zero digit to the left of the binary point. It must be a one.

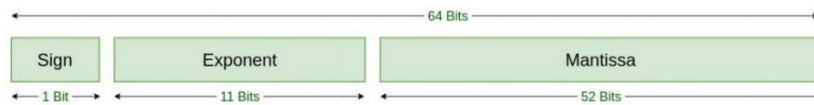
In order to reliably share floating-point binary data between devices, a standard was created, the IEEE 754 Standard for Floating-Point Arithmetic. This standard is used to represent real numbers on the majority of contemporary computer systems. The two most common formats are a 32-bit pattern to represent single-precision numbers and a 64-bit pattern to represent double-precision numbers. Both of these formats partition the binary pattern into three parts with each part representing a different scientific notation component. The order and use of each partition are the same in both formats. Only the number of bits used to represent each part varies.

Click each hotspot below for more information about each group of bits:

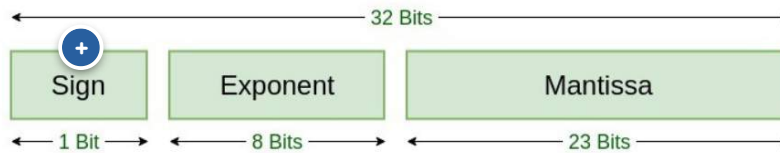


Single Precision IEEE 754 Floating-Point Standard

NOTE: Both of these formats partition the binary pattern into three parts

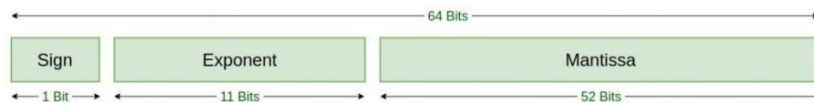


Double Precision IEEE 754 Floating-Point Standard



Single Precision IEEE 754 Floating-Point Standard

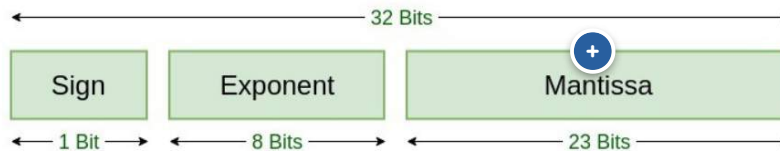
NOTE: Both of these formats partition the binary pattern into three parts



Double Precision IEEE 754 Floating-Point Standard

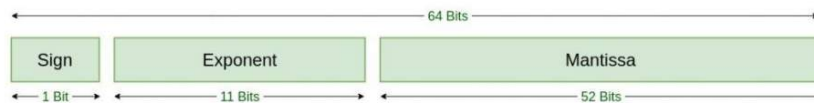
Sign

The first or left-most bit of both formats is used to represent the sign of the number. A zero in this position means the number is positive while a one means the number is negative.



Single Precision IEEE 754 Floating-Point Standard

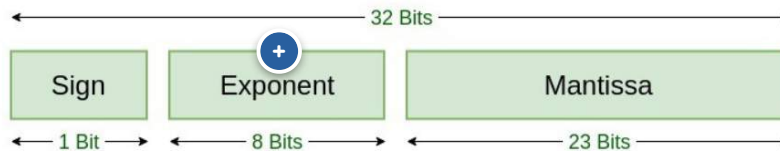
NOTE: Both of these formats partition the binary pattern into three parts



Double Precision IEEE 754 Floating-Point Standard

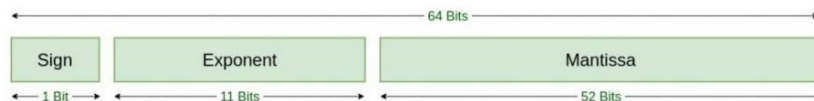
Mantissa

The third and right-most group of bits represents the fraction from which the mantissa is generated. Single-precision uses twenty-three bits here while double-precision uses fifty-two. Here we will represent the fraction with the letter F. There can be some confusion here. Remember that the integer part of any mantissa has a lower limit of one and an upper limit of one less than the base. In binary, this integer portion must always be one. Because of this, the integer portion of the mantissa in binary is not stored as part of the binary pattern, only the fraction is.



Single Precision IEEE 754 Floating-Point Standard

NOTE: Both of these formats partition the binary pattern into three parts



Double Precision IEEE 754 Floating-Point Standard

Exponent

The next group of bits, which we will call E, represents the exponent in offset notation. For 32-bit single-precision notation, this exponent is eight bits long with a bias of 127 while 64-bit double-precision notation uses an eleven-bit exponent with a bias of 1023.

To convert from IEEE 754 notation to binary scientific notation, we first partition the 32- or 64-bit number into its three components, then fill in the blanks of the corresponding scientific notation format. For single-precision numbers, the equation including the operation to take care of the offset or bias is $\pm 1.F \times 2^{(E - 127)}$. For double-precision numbers, the equation including the operation to take care of the offset or bias is $\pm 1.F \times 2^{(E - 1023)}$. In both cases, F is preceded with the implied '1' followed by a binary point.

Let's convert the 32-bit single-precision IEEE Standard 754 pattern below to scientific notation:

1 10000101 01000000000000000000000000000000

1

First, we partition the 32-bit number into its components. The most significant bit is a one, which means that this is a negative number. The eight bits following the sign bit represent the exponent, E. In this example, E equals 10001001, which in decimal equals $128 + 4 + 1$ or 133. Remember that this is in offset notation, so we subtract 127 from 133 to get the exponent six.

2

The remaining 23 bits are the fraction of the mantissa. They are 01 followed by twenty-one trailing zeros. That gives us our value in scientific notation: -1.01×2^6 . Note that the trailing zeros have been discarded. To determine the value this represents in fixed-point notation, we move the binary point six positions to the right to get -1010000. In decimal, -1010000 equals -80.0.

3

Now let's go the other way. Create the 32-bit single-precision IEEE 754 representation of the binary number +0.0001011. The first step is to put this number into scientific notation with a single 1 to the left of the binary point. Note that this is done by moving the binary point four positions to the right to get $+1.011 \times 2^{-4}$. This is a positive number, so the sign bit is zero. Next, the exponent is -4. Since we're storing this in offset notation with a bias of 127, add 127 to -4 to get 123. One hundred and twenty-three in 8-bit binary is 01111011.

4

For the fraction, we only store the digits following the binary point. In this case, the digits to the right of the binary point are 011. We add twenty trailing zeros to the right of 011 since the portion of IEEE 754 single-precision notation that stores the fraction requires exactly twenty-three bits. Putting these patterns in the order of sign bit followed by 8-bit exponent followed by 23-bit fraction gives us the IEEE 754 single-precision representation:

0 01111011 011000000000000000000000.

Image created using [Canva](#)

The following table summarizes information about the different precisions for IEEE 754 floating point representations.

Precision	Bits	Exponent (E)	Significand (F)	Bias	Decimal Range
Half	16	5	10	15	65,504
Single	32	8	23	127	6.8×10^{38}
Double	64	11	52	1023	3.6×10^{308}
Quad	128	15	112	16383	2.4×10^{4932}

Image created using [Canva](#). PDF available below.



ieee precision table.pdf
17.1 KB



IEEE 754 defines a few special cases. First, you may have noticed that it is impossible to represent zero since the integer part of the mantissa is assumed always to be one. Zero is represented by setting both E and F to zero. The sign bit is ignored. IEEE 754 also allows us to represent **infinite**. Infinite is represented by setting E to all ones (255), and clearing F to all zeros. The sign bit distinguishes **positive infinite** from **negative infinite**. Lastly, if E is set to all ones and F is set to a non-zero value, we have what is called NaN or **Not a Number**. This means that the value we've attempted to store is not representable. For example, an attempt to take the square root of a negative floating-point number should result in an NaN value.

You can use these two websites to practice and check your IEEE-754 Floating Point conversions.

<http://babbage.cs.qc.cuny.edu/IEEE-754.old/Decimal.html>

<http://babbage.cs.qc.cuny.edu/IEEE-754.old/32bit.html>

Issue with Floating Point

Let's move away from computers for a moment and back to the real world (pun intended). With the Real numbers (those that involve a decimal point) there are an infinite number of reals between any two real numbers (Yes, I can prove this if I need to!). But look at the floating point representations. With 32 bits we can only represent $2^{32}=4,294,967,296$ different numbers which is still way too small just to represent the values between 2.14123 and 2.14124.



Image from Articulate Content Library

This of course leads to gaps in the representation or values that cannot be stored in IEEE 754 format.

These gaps are referred to as the **forbidden zones**. This presents a problem in that this notation cannot exactly represent the values that we want. So, values for π , e , square root of 2, etc. are approximate values in the computer. Another problem is the difference between the exact value and its

IEEE 754 representation varies from number to number. There is not an exact error value that can be used for all the numbers. Sometimes the values may be very close or far away. To complicate this even more the actual number may be larger or smaller than the represented number.

Calculations using Floating Point

FLOATING POINT ADDITION

To understand floating point addition, first, we see the addition of real numbers in decimal as the same logic is applied in both cases. Click through the process interaction for examples.

For example, we have to add $1.1 * 10^3$ and 50.

We cannot add these numbers directly. First, we need to align the exponent and then, we can add significant.

After aligning exponent, we get $50 = 0.05 * 10^3$

Now adding significant, $0.05 + 1.1 = 1.15$

So, finally we get $(1.1 * 10^3 + 50) = 1.15 * 10^3$

Here, notice that we shifted 50 and made it 0.05 to add these numbers.

Now let us take an example of floating point number addition

We follow these steps to add two numbers:

1. Align the significant
2. Add the significant
3. Normalize the result

Let the two numbers be

$$x = 9.75$$

$$y = 0.5625$$

Converting them into 32-bit floating point representation,

9.75's representation in 32-bit format =

0 10000010 001110000000000000000000

0.5625's representation in 32-bit format =

0 01111110 001000000000000000000000

Now, we shift the mantissa of lesser number right side by 4 units.

Mantissa of **0.5625** =

1.001000000000000000000000

(note that 1 before decimal point is understood in 32-bit representation)

Shifting right by 4 units, we get **0.000100100000000000000000**

Mantissa of **9.75** = **1.**

001110000000000000000000

Adding mantissa of both

0. 000100100000000000000000

+ 1. 001110000000000000000000

1. 010010100000000000000000

In the final answer, we take the exponent of the bigger number

Step 6 Title

So, the final answer consists of :

Sign bit = 0

Exponent of bigger number = 10000010

Mantissa = 010010100000000000000000

32-bit representation of answer =

x + y = 0 10000010 010010100000000000000000

FLOATING POINT SUBTRACTION

Subtraction is similar to addition with some differences like we subtract mantissa unlike addition and in sign bit we put the sign of greater number.

Let the two numbers be

$x = 9.75$

$y = -0.5625$

Converting them into 32-bit floating point representation

9.75's representation in 32-bit format =

0 10000010 001110000000000000000000

– **0.5625**'s representation in 32-bit format =
1 01111110 001000000000000000000000

Now, we find the difference of exponents to know how much shifting is required.

$$(10000010 - 01111110)_2 = (4)_{10}$$

Now, we shift the mantissa of lesser number right side by 4 units.

Mantissa of – **0.5625** = **1.001000000000000000000000**

(note that 1 before the decimal point is understood in 32-bit representation)

Shifting right by 4 units, **0.000100100000000000000000**

Mantissa of **9.75** = **1. 001110000000000000000000**

Subtracting mantissa of both

0. 000100100000000000000000

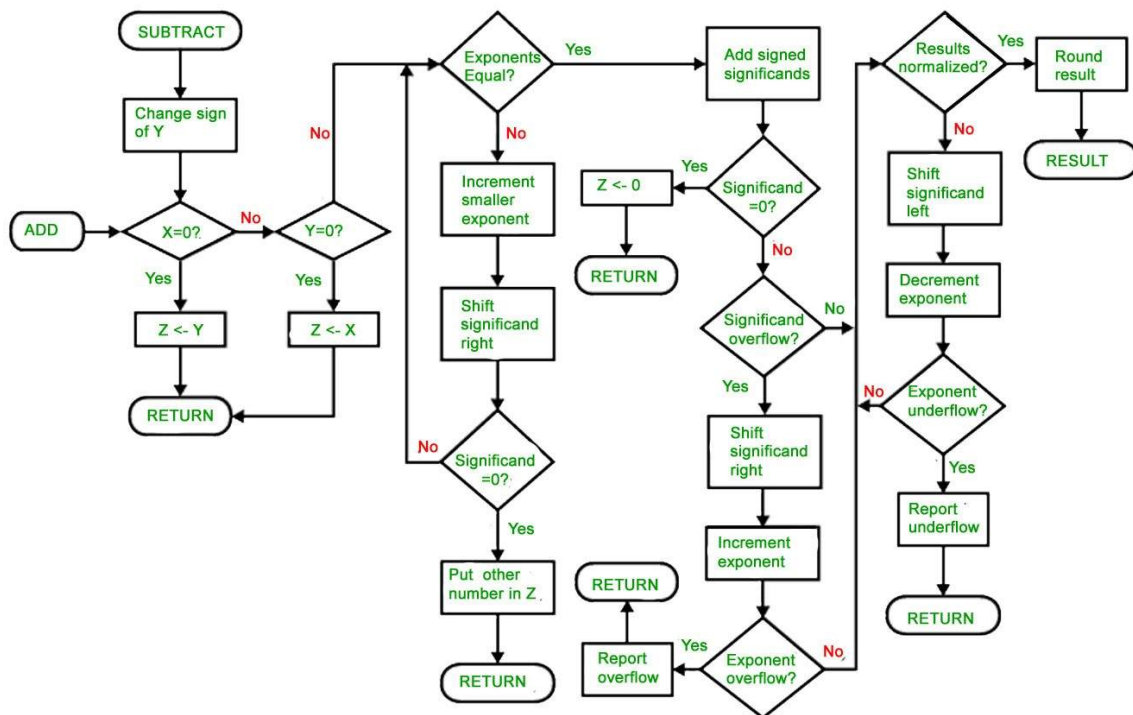
– 1. 001110000000000000000000

1. 001001100000000000000000

Sign bit of bigger number = 0

So, finally the answer =

$x - y = 0\ 10000010\ 001001100000000000000000$

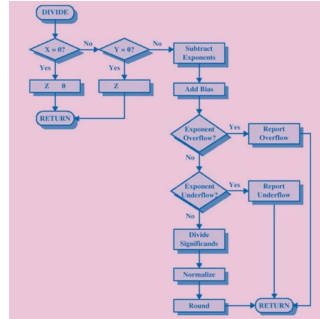
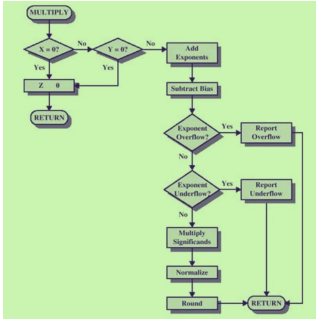


Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

This article has been contributed by Anuj Batham.

Multiplication and Division

We are not going into examples on floating point multiplication and division. The flowcharts for these processes are given below.



Images created using [Canva](#)

Errors with Floating Point

1

Rounding and Truncation

When doing these floating point calculations there are more digits used in the calculation than what can be stored in the final representation. What happens with the extra digits?

The choices are:

- Rounding
- Truncating

The simplest rounding mechanism is truncation or "rounding towards zero".

In rounding to the nearest, the closest floating-point representation to the actual number is used.

In rounding to positive infinity or negative infinity, the nearest valid floating-point number in the direction of positive infinity or negative infinity respectively is chosen.

When the number to be rounded is midway between two points on the floating-point continuum, IEEE rounding specifies the point whose least-significant digit is zero (i.e., round to even).

These techniques all produce errors (induced errors) in the final calculations. Rounding to the nearest is the preferred method because it is more accurate and gives an unbiased error. Truncation always produces a smaller result than the actual value. Rounding can produce a smaller OR larger result than the actual value.

Errors with Calculations

Floating point addition is not associative. The order the numbers are added makes a difference!

Here is an example:

$$(-1.5 * 10^{38} + 1.5 * 10^{38}) + 1.0 = 1$$

$$-1.5 * 10^{38} + (1.5 * 10^{38} + 1.0) = 0$$

These are the actual calculations produced by OneNote and can be reproduced in Excel or in your favorite programming language.

The moral of the story - If you are doing floating point calculations check your input numbers and your answers. Especially if you are using a parallel machine that dynamically assigns calculations to processors.

Calculating the amount of error

The calculations used for error propagation with computer calculations are the same formula as for errors you may have learned in a physics or chemistry lab.

X and Y are the real values.

X' and Y' are the values represented in the computer.

R_x and R_y are the differences between the real value and the represented values. Error values.

So: $X' = X + R_x$ and $Y' = Y + R_y$

Relative error calculations are R_x/X and R_y/Y . This gives a percentage of error.

If we do this calculation: $X * Y$ in reality what is calculated is:

$X' * Y'$ where $X' = X + R_x$ and $Y' = Y + R_y$

$(X + R_x)(Y + R_y)$

$XY + X*R_y + Y*R_x + R_x * R_y$ since R_x and R_y are "small" values we can ignore $R_x * R_y$.

The error for the calculation is now: $X*R_y + Y*R_x$

And the relative error calculation is now:

$$(X * Ry + Y * Rx) / (X * Y)$$

Addition error propagations are much smaller than the multiplication error propagations.

Other Calculations Methods

As we have covered so far there are issues with using the basic arithmetic calculations

(+, -, * and /) with floating point numbers.

What about other functions like sin, arctan, square root, etc.?

You might remember **Taylor series expansion** from a calculus class. It can be used to calculate functions such as sqrt(x) and sin(x) and the following figure shows how the Taylor series can be used to do several other functions.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots \text{for } |x| < 1$$

$$\tan^{-1}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots \text{for } |x| < 1$$

PDF available below

The longer the series is calculated the more accurate the calculation.



other calculations Taylor Series.pdf

85.8 KB



You may remember another series the Chebyshev series. It turns out to be better behaved than the Taylor series and the errors are distributed more evenly. One last advantage is the Chebyshev series requires fewer terms to converge the series or arrive at an accurate solution.

One last method to perform these calculations is called Liner interpretation. It uses an abbreviated table of stored values of a function then assumes the values between the stored ones can be assumed to be a straight line to calculate the value of the specific value wanted.

Here is a simple example. We want to be able to calculate values for Cosine. To save space only every five degrees of values are stored ($\cos(0)$, $\cos(5)$, $\cos(10)$, etc.).

- What happens when we want a value that is not specifically in the table? For example: $\cos(32)$?

The values stored in the table that bound $\cos(32)$ are:

$$\cos(30) = 0.866025403784439$$

$$\cos(35) = 0.819152044288992$$

- Cosine (and a lot of other functions) is not liner (not a straight line) but we use a straight line to get a value for $\cos(32)$.
- Here is the calculation:

$$(0.8660 - 0.8192) / 5 = 0.0094$$

Value for every degree over this range from 30 to 35.

$$0.8660 - (0.0094 * 2) = 0.8472$$



While the actual value is:

$$\text{Cos}(32) = 0.848048096156426$$

Not bad if all you really need is two significant digits and speed is more important than accuracy.

Look-up tables stored in Read Only Memory (ROM) have been and continue to be used to look up functions. Instead of doing a calculation all that is required is a memory access.

What can you do to keep from using Floating Point?

As you have been able to figure out by now there are lots of issues with floating point representations and calculations. What can be done to keep from using floating point? Here are some examples where integers are used instead of floating point.

1

US Currency - change the units from dollars to cents. So, \$123.67 would be defined as an integer instead of a floating point. Just have to remember to change the format before it is displayed.

2

Distance - again change the units so that integer calculations can be used. For example:

~ Tenths of miles - 35.4 miles store as 354.

~ Thousands of miles - Assuming the least 3 significant digits are not important and would still fit in a 32-bit unsigned number. So, 45698734 would become 45698.

~ A little easier with the metric system since everything is

already on the power of 10. So instead of keeping the units in kilometers, you can do it in meters.

If there are still not enough digits then you can build your own Quad² or Quad⁴ integers as a class and create your own overloaded operators for those new classes/data types.

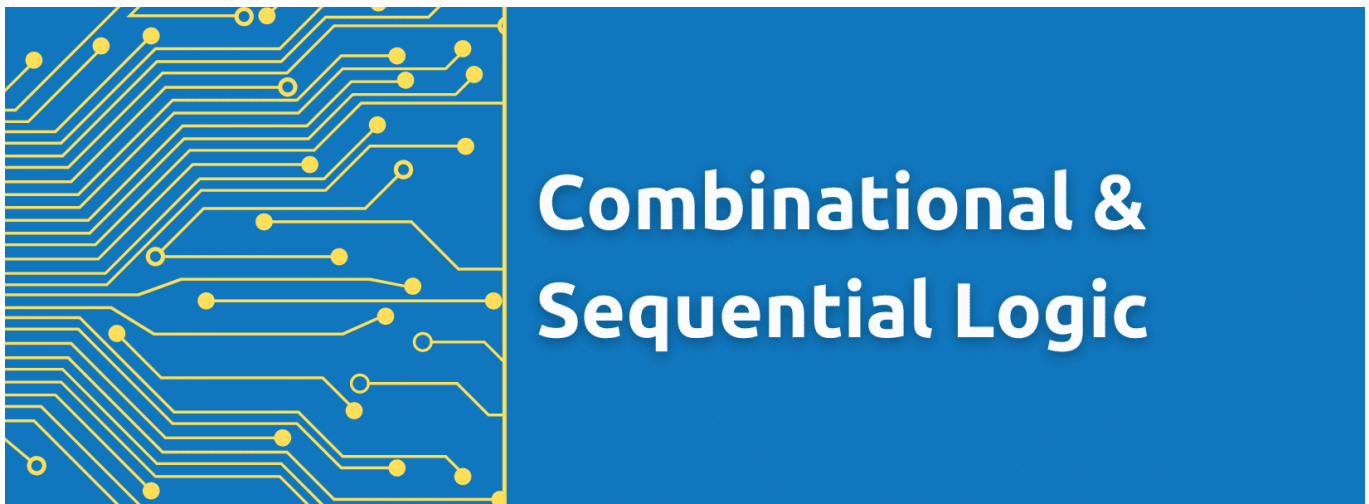
If calculation speed and accurate representation are needed then this type of approach works well. The drawback is keeping track of the units, especially if there are calculations that take place. Make sure you are using compatible. For example, you are keeping a distance in tenths of feet but to use 32 feet/second² to calculate the acceleration due to gravity the calculations will be off by a factor of 10.

Summary

The important parts of this chapter to remember:

1. All of these calculations can be performed using the basic gates that have already been covered in this course. Yes, it is really complicated but it can be done.
2. Inexact representations (errors) are compounded when calculations are performed.
3. Floating point calculations do not work the same way as you have been taught in mathematics classes.

Combinational & Sequential Logic



Introduction

So far we have covered the basic building blocks of the central process unit (CPU), how the basic gates can be used to create complex circuits as well as the internal representation of Integers and floating point numbers. At this point, we are now turning back to Combinational and Sequential logic and learning how to build a CPU.

After building the CPU we will go into how this design works to fetch and execute instructions.

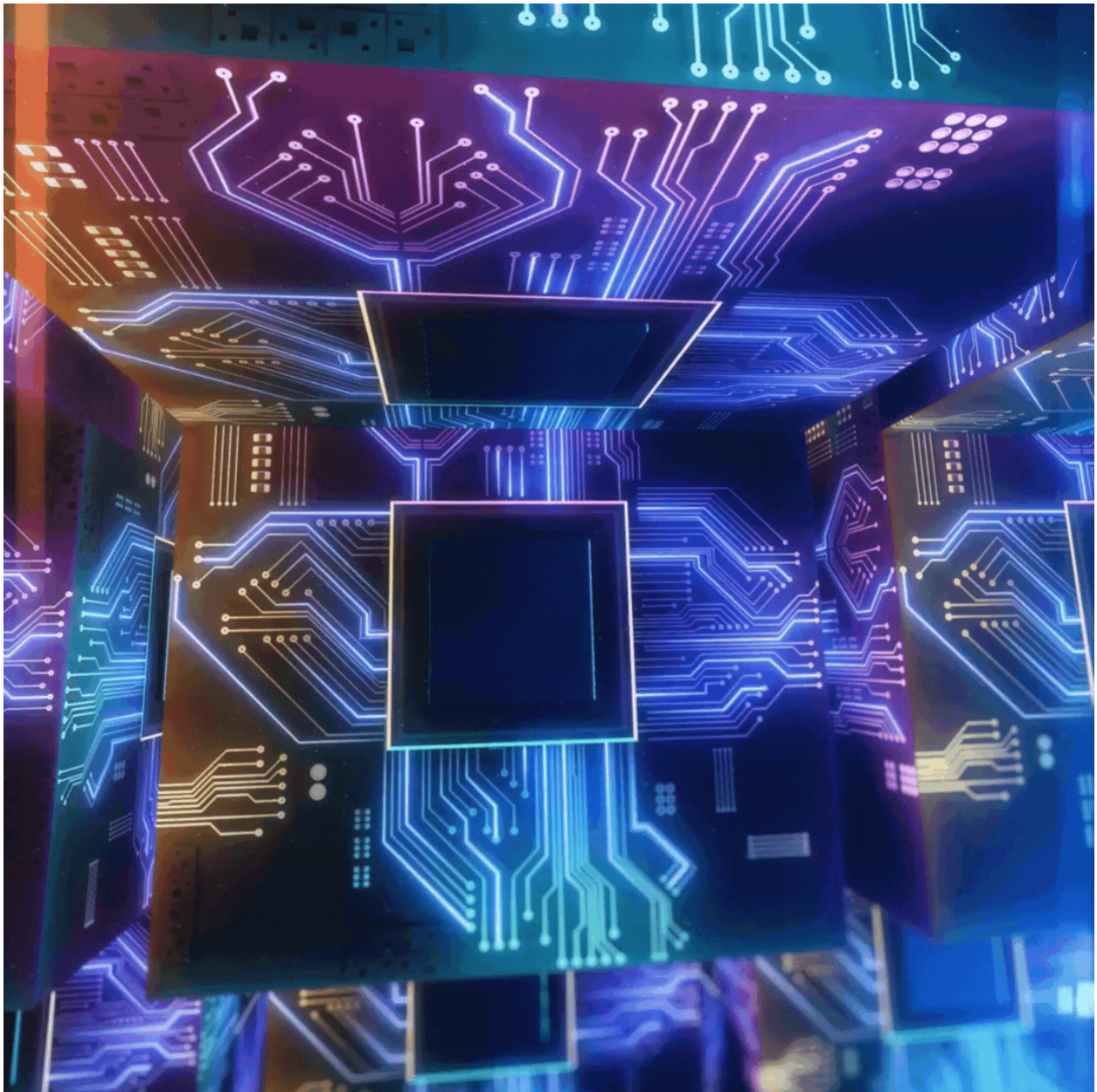


Image created using [Canva](#)

Objectives:

From this module, students should learn

- Classifications of CPU register types
- The components of a CPU
- Instruction Fetch process
- Instruction Execution process
- Main memory organization
- Details of a hardwired control unit

Students should:

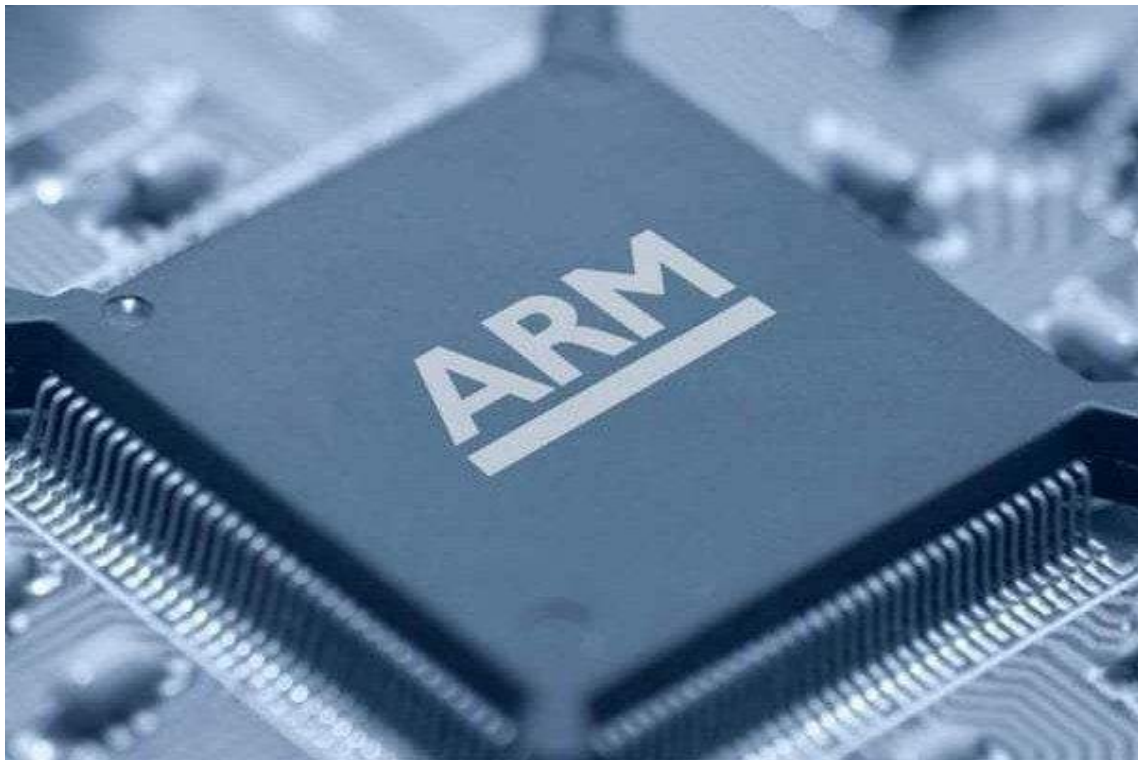
- Read the following material
- Watch the associated video lecture
- Take the Canvas quiz on this topic

Something to keep in mind is the ARM CPU is extremely complex. What will be shown is just enough of the design to illustrate a few parts of the topic that is being described. Later in this course, different designs will be covered to get a better understanding of the relationship of the design to the execution speed of the instruction.

The major parts of a modern CPU are

1. Registers
2. Buses
3. Arithmetic and Logic Unit (ALU)
4. Control Unit

Click each tab below to learn more about each part.



REGISTERS

BUSES

ARITHMETIC AND LOGIC UNIT (ALU)

CONTROL UNIT

Registers were described in the Sequential Logic module. CPU Registers are classified according to their visibility to the assembler programmer.

They are:

- **General Purpose Registers** that can be accessed/used by the programmer.
- **Special Purpose Registers** that the programmer can effect but not directly accessed
- **Invisible Registers** are needed by the CPU but cannot be directly controlled by the programmer.

The "General Purpose" Registers for the ARM are labeled R0, R1, . . . R15. The reason this is in quotes is as an assembly programmer all of these registers can be used and accessed BUT just because you can use any of these registers does not mean you should. This is just another example of the difference between theory and practice.

ARM has r0-r11 are completely interchangeable and can be used by the assembler programmer for any purpose and clearly fall under the **General Purpose Register** classification. As for r12-r15:

- R12 used for Intra-Procedure-call scratch register (SR)
- R13 used as the stack pointer (SP)
- R14 subroutine return address aka Link Register (LR)
- R15 program counter (PC)

Even though the assembler and the CPU allow access to these registers you as a new assembler programmer should not use these registers UNLESS you really know what you are doing.

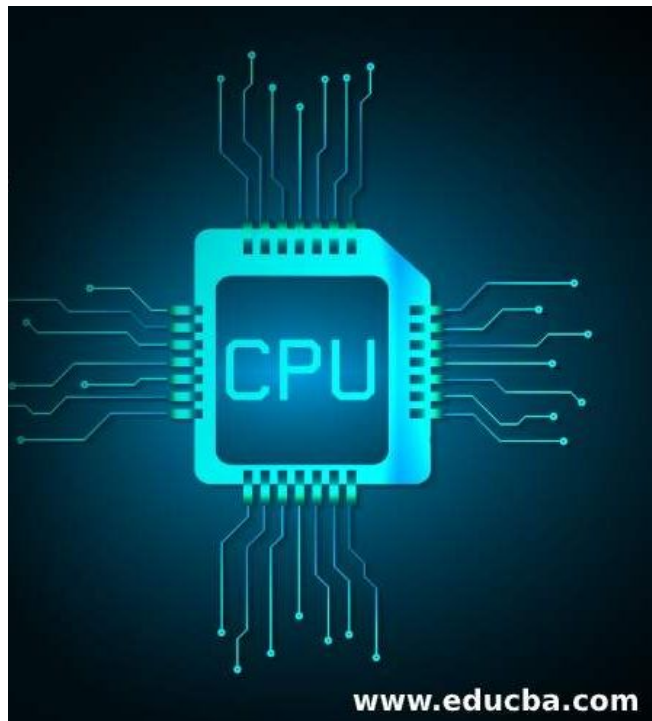
On other CPUs (non-ARM) these registers, these registers (r12-r15) would be classified as special purpose registers. Another special purpose register is the Condition Code Register (CCR) or Process Status Register (PSR). The name can vary from CPU to CPU but they serve the function.

Other **Special Purpose Registers** are typically:

- IR - Instruction Register
- MAR - Memory Address Register
- MBR - Memory Buffer Register

There may be many **Invisible Registers** depending on the design of the CPU. There will be some architectures that will be covered in another module that make use of **Invisible Registers**.

Most modern CPUs have all their registers exactly the same size. The ARM's registers are 32-bits. Other CPUs may be larger or smaller and typically are powers of 2 in size. 16, 32, 64, 128 bits are typical register sizes.



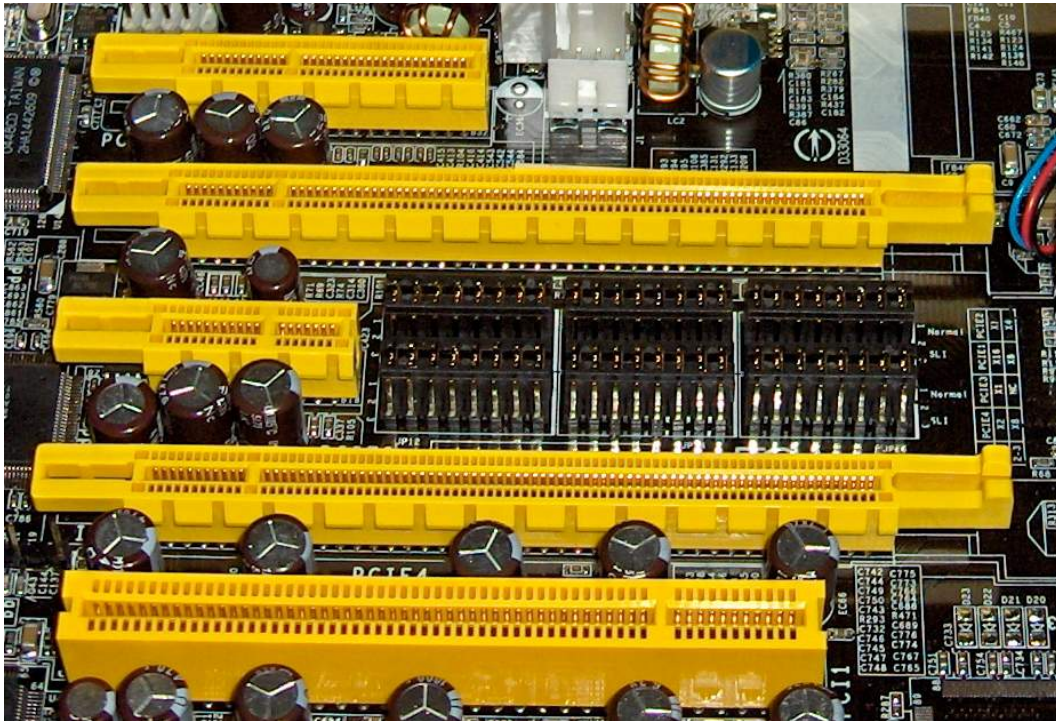
REGISTERS

BUSES

ARITHMETIC AND
LOGIC UNIT (ALU)

CONTROL UNIT

Once again, a Bus is a collection of wires used to transfer data. The number and size (width or number of bits) of the bus varies from CPU to CPU. The ARM's bus is 32-bit which matches the width of the registers. In most cases, the width of the bus and registers are the same. There are other CPUs in which the bus width is not the same as the CPU registers. For example, there can be a bus width of 16 bits but the internal registers are 32-bits in width.



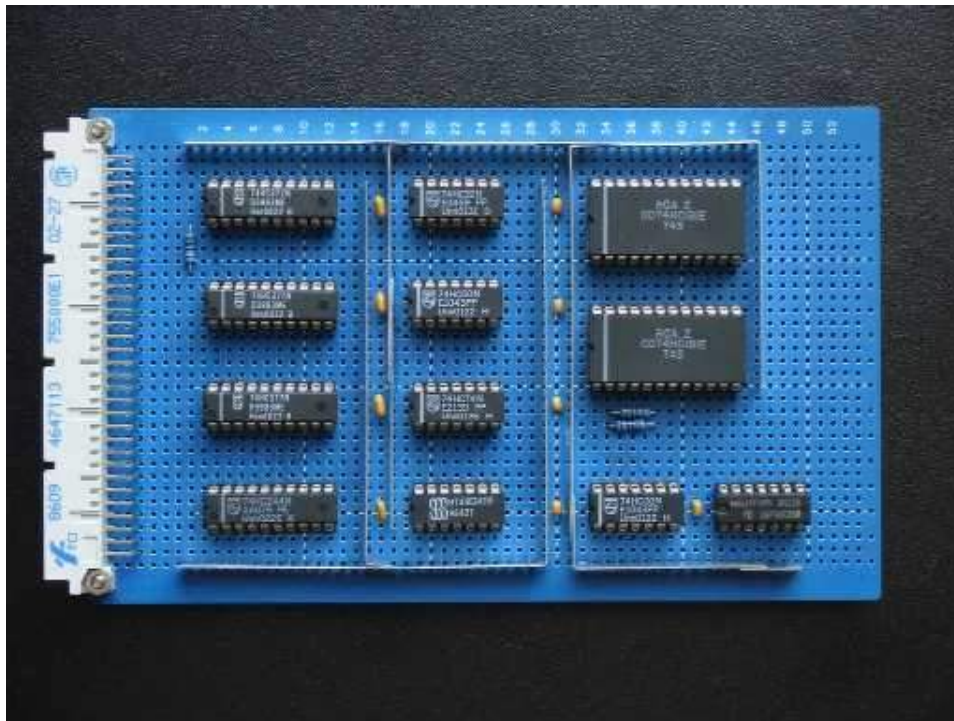
REGISTERS

BUSES

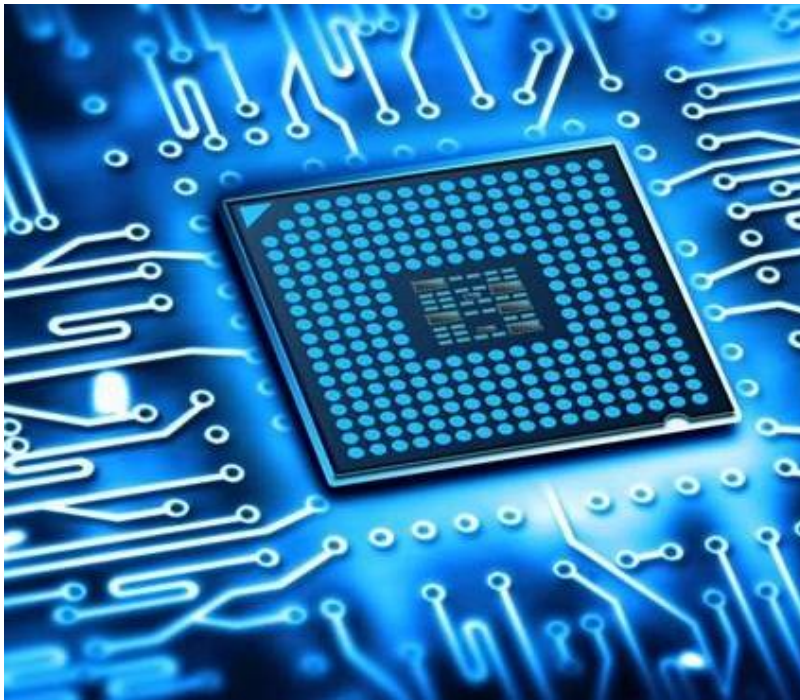
ARITHMETIC AND
LOGIC UNIT (ALU)

CONTROL UNIT

The ALU is a piece of Combinational logic that performs all the arithmetic functions, comparisons, and calculations performed by the CPU. The ARM's ALU does addition, subtraction, multiplication, and various different shifts. The latest version of the ARM now supports division.



REGISTERS	BUSES	ARITHMETIC AND LOGIC UNIT (ALU)	CONTROL UNIT
The CPU's control unit is by far the most complicated part of the CPU. It is responsible to issue commands to the registers and ALU to fetch and execute all the CPU instructions.			

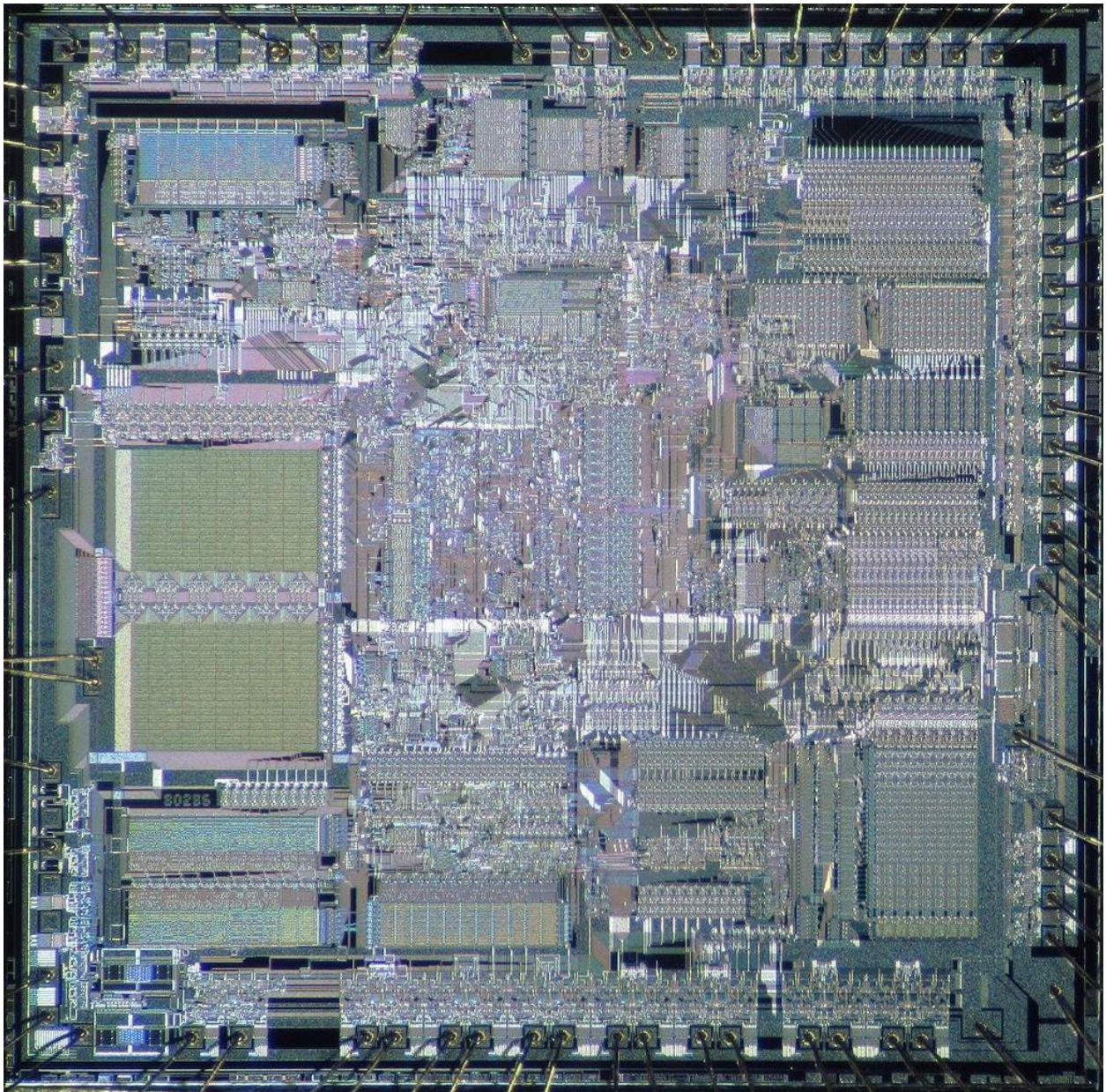


Simplified CPU

Just so you understand how simplified the CPU we are going to start out with is going to be let's take a look at a the Intel 80286 CPU that was released in 1985. Summary of its capabilities:

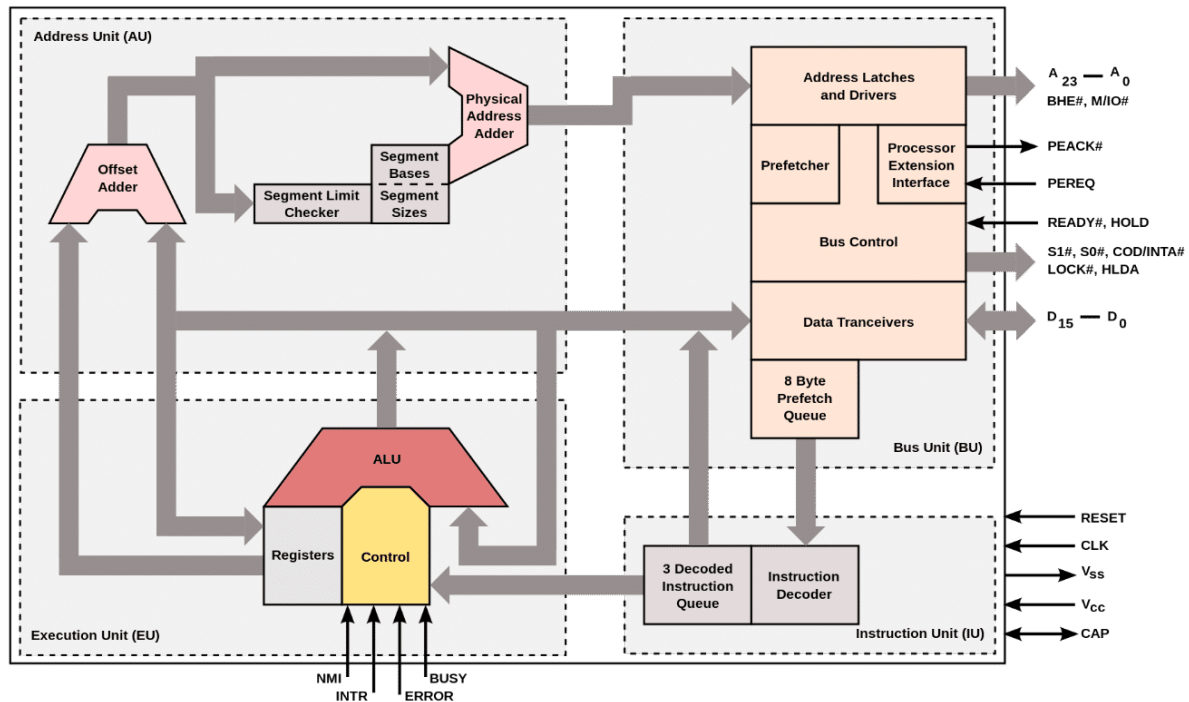
- Clock speed 12.5 MHz
- 134,000 transistors
- 16-bit architecture
- 68 pins Integrated Circuit
- 2.67 Millions of Instructions Per Second (MIPS)
- Only integer calculations. Performs no floating points calculations.

You can check out more details of the 80286 here: https://en.wikipedia.org/wiki/Intel_80286



Picture of the 80286 CPU Die

Intel 80286 architecture

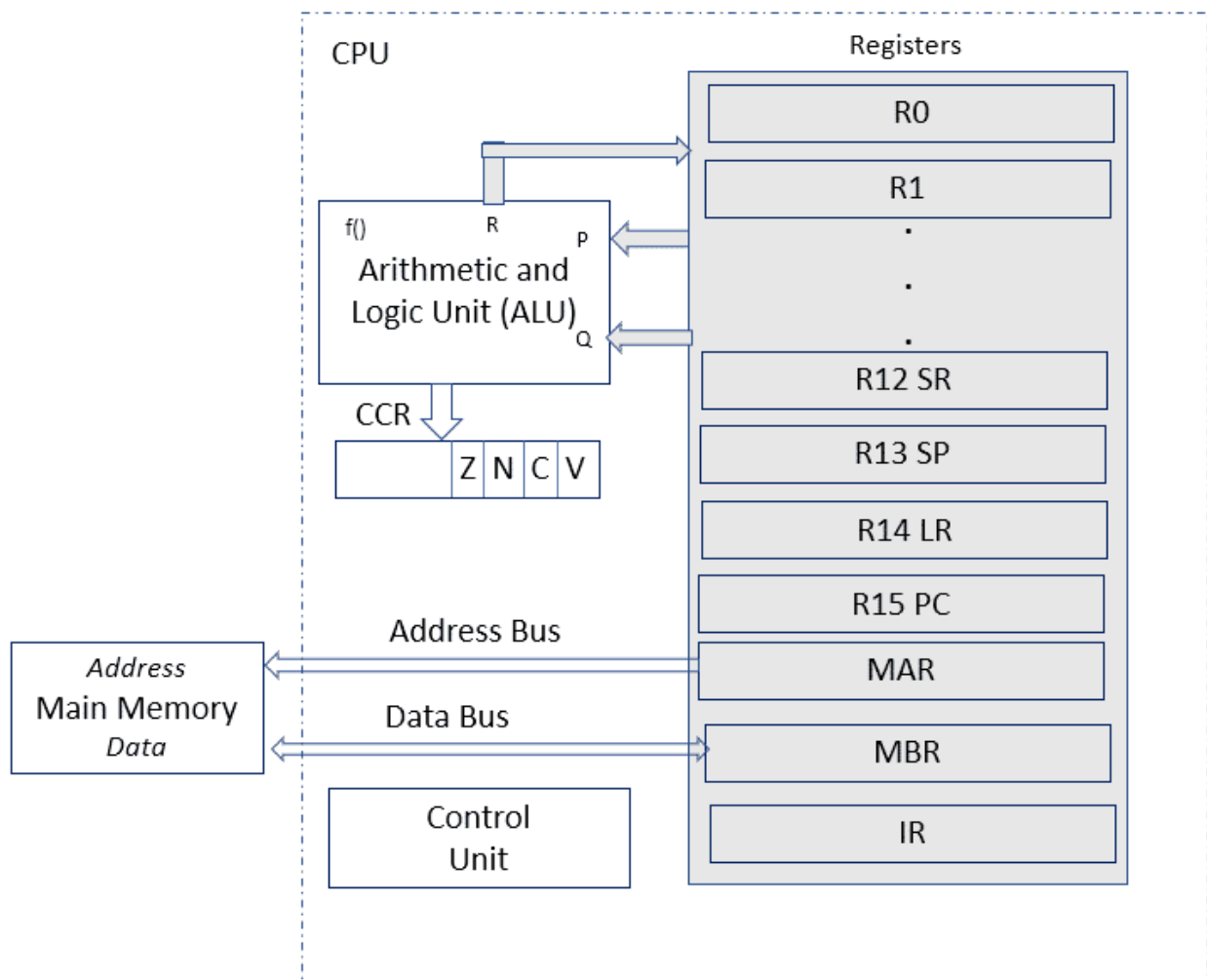


Block diagram of the 80286 CPU

Even this really old CPU's design details are beyond the scope of this course. One of the goals of this course is to give you enough of an understanding of the various parts of the CPU so that if you really needed to figure out the details of a specific modern CPU you should be able to.

The following figure is the one that we will spend some time on in understanding the instruction fetch and execution process. It does reflect (somewhat) the design of the ARM CPU. Just so we do not confuse this diagram with other CPU diagrams we will refer to this one as "Mr. Preston's ARM CPU".

What do we have here and how does this work?



Mr. Preston's ARM CPU



Registers

In this design all the ARM registers are grouped together. This includes the general purpose registers and the special purpose registers. ARM does not have any hidden registers. Any one of these registers can output to bus P or Q and can read data from bus R.

The CCR is the one register that is not part of this bus P or Q arrangements. The calculations made by the ALU cause the flags in the CCR to be

updated. This is not always the case and will be discussed in the ARM Assembly module. The specific flags are:

- Z - Zero
- N - Negative
- C - Carry
- V - Overflow

The details on when these flags get set or cleared will be covered in another module.

There are three new registers that are special purpose registers that need to be described.

- **MAR - Memory Address Register.** This contains the address of the data that is being read from or written to memory.
- **MBR - Memory Buffer Register.** This contains the data that is being read from or written to memory.
- **IR - Instruction Register.** This contains the current instruction being executed. The instruction is decoded and control signals are generated to execute the instruction.



Buses

This design has three buses (P, Q and R) that are connected to the ALU and the ARM registers. This

arrangement forces all the data that is transferred between the registers to be routed through the ALU.

There are two buses in this design that are needed to access memory. There is the address bus and the data bus. To read memory an address needs to be on the address bus and when the read is complete the data shows up on the data bus. Writing to memory requires both the address on the address bus and the data to write on the data bus.

There is also a bus that connects the ALU to the CCR.



Arithmetic and Logic Unit (ALU)

The ALU functions will operate on one or both of the input busses (P or Q) and provide the output onto bus R. $f()$ is another input to the ALU that selects the operation to be performed with the input busses. It should be obvious what some of the ALU functions should be and a few not so obvious.

The general categories of these functions fall under:

- Arithmetic
- Shifts
- Transfers

General	Function
Arithmetic	Add Subtract Multiply Complement Two's Complement
Shifts	ASL ASR ROL ROR LSL LSR
Transfer	P --> R Q --> R



Control Unit

The control unit just shows up on the block on "Mr. Preston's ARM CPU" not connected to anything. Actually, the Control Unit has connections to every other component on the CPU. If all the real connections were shown lines would be all over the place and it would be difficult to see or understand what all the other structures do and how they interact with each other.

We will go over the details on what the Control Unit looks like after covering how "Mr. Preston's ARM CPU" works to fetch and execute some instructions.

To keep everything from happening at once the Control Unit has two timing features.

- **Major Clock cycles**
- **Minor Clock cycles**

The **Major Clock** cycles identify if the control unit is doing an instruction fetch or instruction execution. In some cases, the instruction execution may be broken down into additional cycles. More on this when pipelines are covered. For now, the **Major Clock** cycles will only be Fetch or Execute.

The **Minor Clock** cycles subdivide the **Major Clock** cycles into states that are labeled T0, T1, T2, and T3.

At the end of the clock cycles (Major or Minor) ALL the signals from the control unit are turned OFF. There are no "leftover" signals. If a signal from the previous cycle needs to be on again it must be reasserted.

Instruction Fetch

Since Von Neumann came up with the stored program concept of digital computers the instruction fetch process has not changed. The program that is being executed is located in the main memory. The Program Counter (PC) contains the address of the NEXT instruction to be executed. Once the instruction is read from the main memory it then must be put into the Instruction Register (IR) where the instruction is then decoded and executed.

Using "Mr. Preston's ARM CPU" this is what needs to happen to perform the Instruction Fetch:

Instruction Fetch

Since Von Neumann came up with the stored program concept of digital computers the instruction fetch process has not changed. The program that is being executed is located in the main memory. The Program Counter (PC) contains the address of the NEXT instruction to be executed. Once the instruction is read from the main memory it then must be put into the Instruction Register (IR) where the instruction is then decoded and executed.

Using "Mr. Preston's ARM CPU" this is what needs to happen to perform the Instruction Fetch:

Major Clock Cycle = FETCH

Minor Clock Cycles:

T0: $PC \rightarrow \text{Bus P}$, $f(P \rightarrow R)$, $R \rightarrow \text{MAR}$, Read

T1: $PC \rightarrow \text{Bus P}$, $4 \rightarrow Q$, $f(\text{ADD})$, $R \rightarrow PC$

**T2: $\text{MAR} \rightarrow \text{Bus P}$, $f(P \rightarrow R)$, $R \rightarrow \text{IR}$,
(Decode Instruction)**



Major Clock Cycle = FETCH Minor Clock Cycles.pdf
14.4 KB



Translating this into English. The goal of the FETCH is to read memory and get the next instruction to be executed into the Instruction Register where it can be decoded and the Instruction Execution process can then begin.

How does this happen with "Mr. Preston's ARM CPU"?

The PC needs to get into the MAR to read the next instruction to be executed. That is what happens in T₀. While the read is taking place the PC needs to be updated to point to the next instruction to be executed. So in T₁ the PC is updated by adding 4.

Why 4? In the ARM the word size is 32-bits but memory is byte addressable. To get to the next word 4 needs to be added to the PC.

After T₁ is completed that should have given enough time for the read to complete. The CPU runs much faster than main memory access. For this module, we will assume it always takes one minor clock cycle for the main memory access to complete. In T₂ the instruction we want to execute is now in the MBR. Since this is an instruction it must go into the IR.

When trying to understand what control signals need to happen it is best to start where the data is located and where it needs to go. Actually, the order in which the control signals in a minor clock cycle does not matter. All the control signals activate at the same time.

After the three minor clock cycles are completed for the FETCH the Major clock cycle is set to EXECUTE.

Instruction Execution

The instruction execution will vary depending on the instruction being executed, registers being accessed and the addressing modes.

The ARM addressing modes will not be covered until later in this course so the examples of Instruction Execution will deal specifically with register operations.

ADD r1, r2, r3 -- This instruction adds the contents of r2 and r3 and puts the results in r1.

Execution:

To: r2 --> P, r3 --> Q, f(ADD), R--> r1

SUB r1, r2, r3

Execution:

To: r2 --> P, r3 --> Q, f(SUB), R--> r1

MOV r1, r2 -- Copies the contents of r2 and overwrites what is in r1.

Execution:

To: r2 --> P, f(P --> R), P --> r1

LDR r2, 0x0F -- This is a load register instruction. It will read the contents of memory location 0x0F and put the result into register r2.

Execution:

T0: IR --> P, f(P --> R), R --> MAR, Read

T1: Wait (Have to wait until the read is complete.)

T2: MBR --> P, f(P --> R), R --> r2

STR r3, 0xA8 -- This is a store register instruction. It will take the contents of r3 and stores it at memory location 0xA8

Execution:

T0: IR --> P, f(P --> R), R --> MAR

T1: r3 --> P, f(P --> R), R --> MBR, Write

T2: Wait (optional)

ADDRESS	CONTENTS
00000000	FFFFFFFF
00000004	12345678
FFFFFFFF	00000000

PDF available below

Main Memory Organization

The modern design of main memory will be covered in detail in a later module. For our purposes, at this point of the course, it is best to think of main memory as a large array. The subscript for the array can be thought of as the address of the memory location.

This table illustrates how main memory would look on an ARM computer. ARM memory is Byte (16-bits) addressable and each memory location contains 32-bits or 4-bytes. Each address goes up by four

(4) for consecutive memory locations. The range of the addresses goes from 00000000 to FFFFFFFF in hexadecimal. The contents column in the table are just random values.

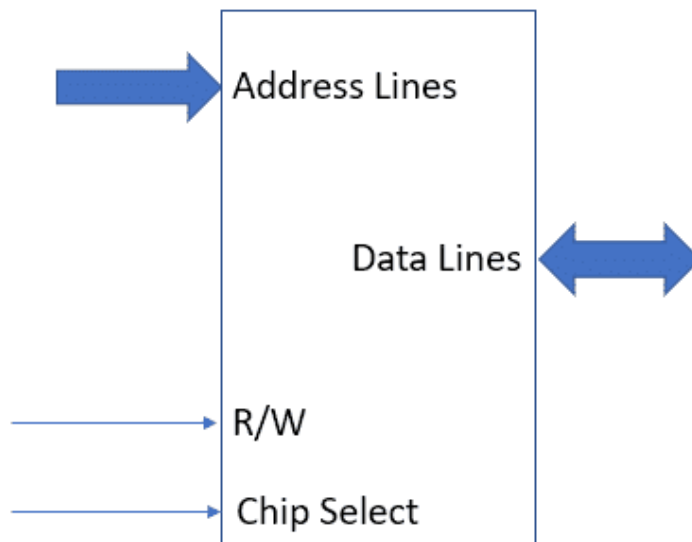


main memory organization.pdf

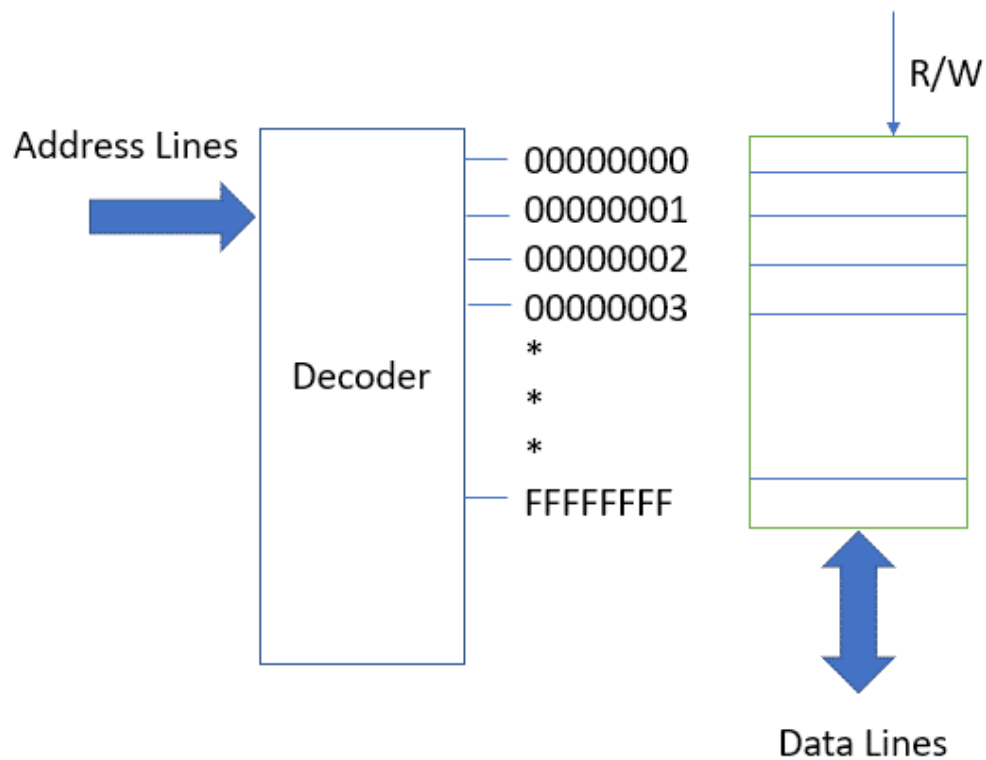
11.8 KB



Main memory falls under the classification of Random Access Memory (RAM). Any memory location can be accessed directly as long as the address is provided. From a hardware standpoint the memory location can be considered as a register that consists of D-flip flops.



Block Diagram of a memory Chip



Internal details of a memory chip
Both images available as PDF

Each row in this table for the diagram to the left can be thought of as being made up of 32-bit registers made using D-Flip Flops.

The address lines are the inputs to a very large decoder. When the address matches the memory location (register) its contents will either be put out on the data lines on a Read or overwritten by what is on the data lines on a Write.



Memory chip diagrams.pdf
46.9 KB

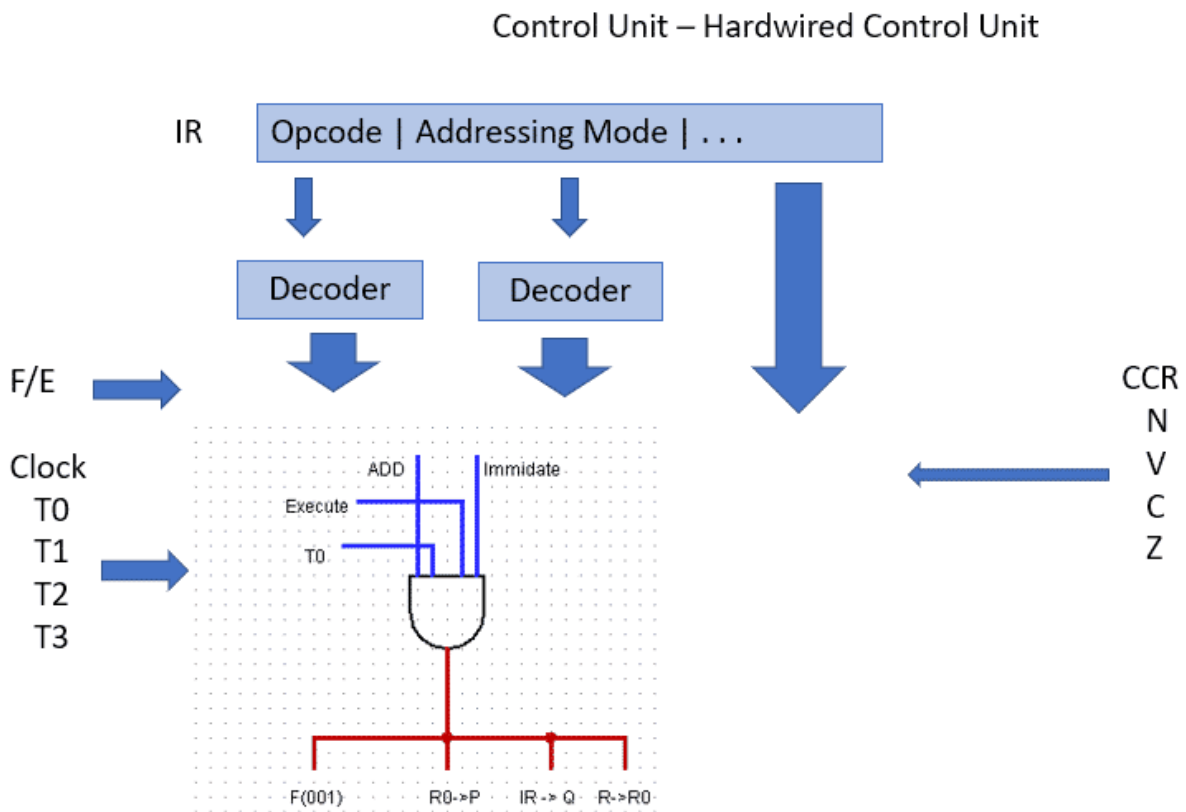


Hardwired Control Unit Details

Here is what the hardwired Control Unit looks like in a little more detail. The inputs to the control unit are:

1. Major Clock Cycle (Fetch/Execute)
2. Minor Clock Cycles (T0 to T3)
3. Instruction Register
4. CCR with its flags

The outputs of the Control Unit go to the ALU and the controls for all the buses. There are also commands to perform memory read and write operations.



Control Unit - hardwired Control Unit - PDF Available Below



Hardwired Control Unit.pdf

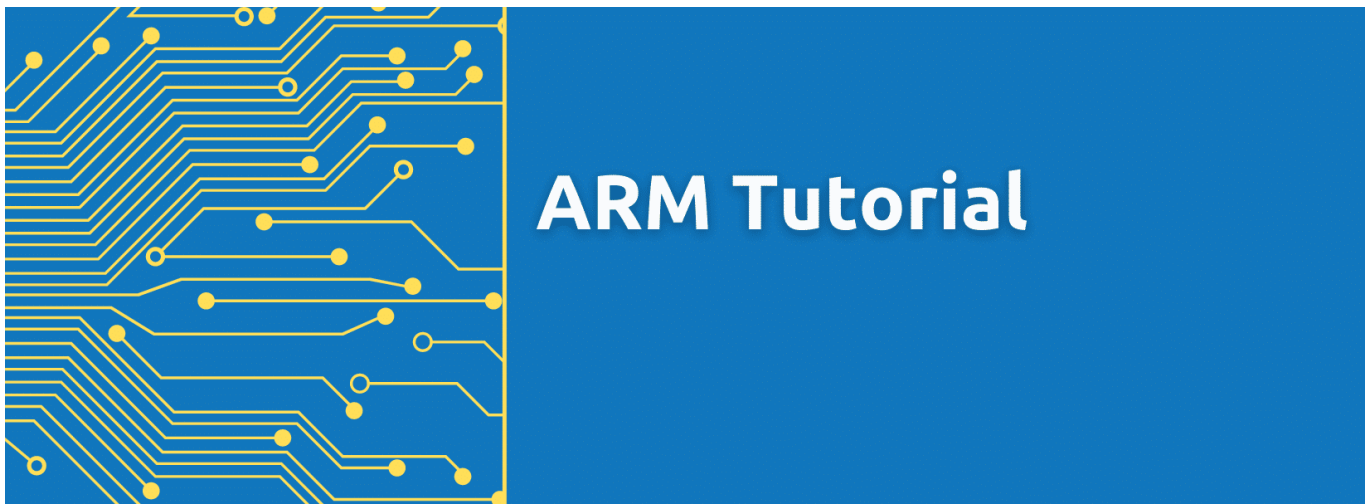
59.3 KB



The Instruction Register (IR) consists of 32-bits. In general, the IR will contain the:

- Opcode - The instruction
- Addressing Mode - More details on this in the ARM Assembly module
- Registers used (if applicable)
- Literal - More details in the ARM Assembly module

ARM Tutorial



Introduction

This reader introduces the basics of ARM v7 assembly. The primary goal is to help students who already know how to program in a high-level language learn about the form their programs take after compiling – the kinds of instructions a computer really processes information. These materials provide examples of basic ARM instructions, assembly programming constructs, and how high-level languages constructs are represented in assembly.

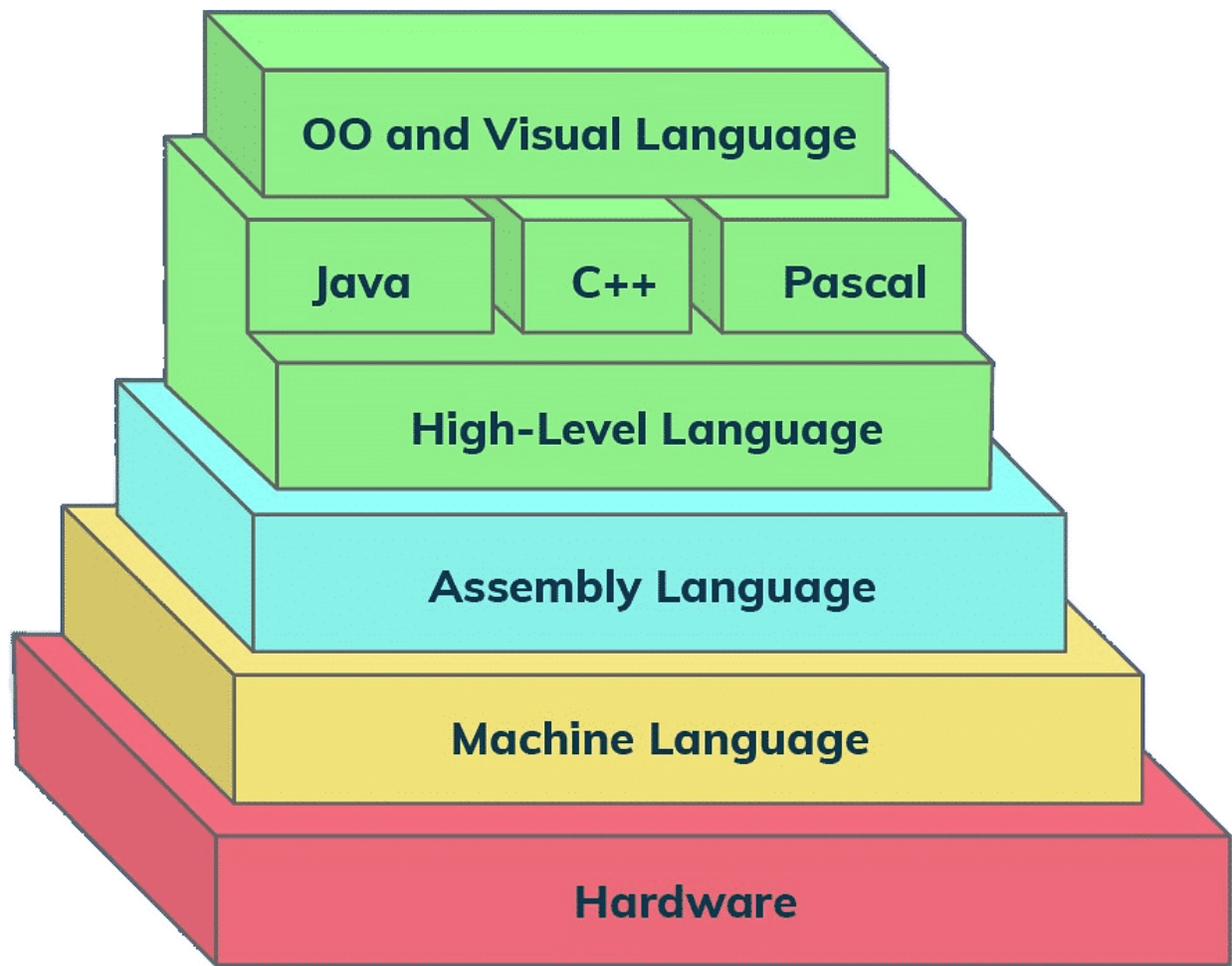


Image from: <https://www.freecodecamp.org/news/learn-assembly-language-programming-with-arm/>

Objectives

By the end of this chapter, students should be able to apply the information concerning:

- Basics of ARM assembly
- Memory
- Bitwise operations
- Branches and control structures
- Arrays and strings

- Functions and the stack

BASICS

Instructional Format

ARM instructions generally take the form:

ADD **rd**, **rn**, **#**

Things like **rd** or **rn** are shorthand for **register**. They may normally be any of the general purpose registers like r1, r2, r3,... **rd** stands for “destination” register – it is typically the register that gets modified by the instruction. The other letters, like n or m in rn or rm are meaningless placeholders. It is usually possible for the destination register to be the same as one (or multiple) of the source registers.

The **#** stands for an **immediate** value – a literal number. Literals always start with a **#** and may be written as:

- Signed integers : #10, #-5
- Hexadecimal values : #0xFF
- Binary values : #0b110
- ascii chars : #'A'

Samples:

```

ADD    r1, r1, #10
ADD    r3, r4, #0xFF
ADD    r5, r5, #0b11111111
ADD    r6, r6, #'A'

```

[Try Sample](#)

Comments

Multiline comments are surrounded by `/* */` and single line comments start with `@`:

```

/* This is a
   multiline comment
*/

ADD    r1, r2, #10 @Line comment
@Line comment
ADD    r1, r2, #10

```

[Try Sample](#)

Data Movement

The MOV and MVN instruction copy (move) data from into registers.

MOV *rd*, *rn*

MOVE. Copy data from register *rn* to register *rd*. Does not modify *rn*.

MOV *rd*, #

Copy the immediate value # into register *rd*.

```

MOV    r1, #100      @Place 100 base 10 in r1
MOV    r2, #0xff     @Place hex ff (255) in r2
MOV    r3, #0b1100   @Place 1100 binary in r3 (12 in decimal, 0xC in Hex)
MOV    r4, #'A'      @Place char 'A' or 65 in r4
MOV    r5, r3        @Copy value in r3 into r5

```

[Try Sample](#)

MVN *rd*, *rn* /

MoVe Negated. Places the bitwise negation of the source register or immediate *rn* / # into register *rd*.

```
@Load r1, then load its negation into r2
MOV  r1, #1          @ r1 = 0000 0000 ... 0001
MVN  r2, r1          @ r2 = 1111 1111 ... 1110

@Load r3, then load its negation into r4
MOV  r3, #0xA3       @ r3 = 0000 0000 ... 1010 0011
MVN  r4, r3          @ r4 = 1111 1111 ... 0101 1100

@Load the negation of 0000 0000 0000 ... 1111 into r5
MVN  r5, #0xF        @ r5 = 1111 1111 ... 0000
```

[Try Sample](#)

Arithmetic

Arithmetic instructions combine two values via addition or subtraction, placing the results into a destination register.

Tip: All of the arithmetic instructions only support an immediate value as the last parameter.

ADD *rd*, *rn*, *rm* /

Add *m* to either register *rm* or immediate #, place the results in *rd*.

```
MOV  r1, #100
MOV  r2, #200
ADD  r3, r1, r2      @r3 = r1 + r2
ADD  r4, r1, #1       @r4 = r1 + 1
ADD  r4, r4, #1       @r4++
```

[Try Sample](#)

SUB *rd, rn, rm / #*

Subtracts either register *rm* or immediate *#* from *rn* and places answer in *rd*.

```
@Load some starting values
MOV    r1, #50
MOV    r2, #200
SUB    r2, r2, r1    @r2 = r2 - r1
SUB    r3, r1, r2    @r3 = r1 - r2
SUB    r4, r1, #10   @r4 = r1 - 10
@SUB   r4, #10, r1   @Invalid! Immediate value must be last
```

[Try Sample](#)

Because only the final operand for SUB can be an immediate, we cannot write SUB r2, #10, r1 to subtract r1 from 10. Instead, there is a reverse subtract instruction RSB that subtracts the first parameter from the second.

RSB *rd, rn, rm / #*

Reverse SuBtract. Subtracts *rn* from either register *rm* or immediate *#* and places answer in *rd*.

```
@Load some starting values
MOV    r1, #50
MOV    r2, #200
RSB    r3, r2, r1    @r3 = r1 - r2 using RSB
RSB    r5, r1, #10   @r5 = 10 - r1
```

[Try Sample](#)

Rotation

The rotate instruction shifts all the bits in a register to the right by a given amount. If we take this pattern:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0
0				0				0				0				6				F				0				0			

And shift it right 1 bit, we get:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0	0	0
0				0				0				0				3				7				8				0			

Notice that shifting one bit completely changes the hex version of the pattern. If we want to shift by one hex digit, we should shift the original pattern 4 bits.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	0	0	0	0
0				0				0				0				0				6				F				0			

Tip: Each 4-bits of rotation shifts the hex digits by 1

Bits that shift past position 0 wrap around to bit 31. Here is the pattern shifted 6 more bits:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	
C				0				0				0				0				0				1				B			

ROR *rd, rn, rn / #*

ROTate Right. Rotate *rn* to the right the number of bits in *rn* or *#*. Result in *rd*.

```
MOV    r1, #0x12    @ r1 = 0000 0000 ... 0000 0001 0010
@Rotate r1 right by 1 bit
ROR    r2, r1, #1    @ r2 = 0010 0000 ... 0000 0000 1001
@Rotate r1 right by 4 bits - one hex char
ROR    r3, r1, #4    @ r3 = 0010 0000 ... 0000 0000 0001
```

[Try Sample](#)

There is no rotate left instruction, because we do not need one. A 31-bit rotate is the same as rotating left 1 bit, a 30-bit rotate the same as rotating left 2 bits, If we want to rotate left 12 bits, we could instead rotate 20 to the right:

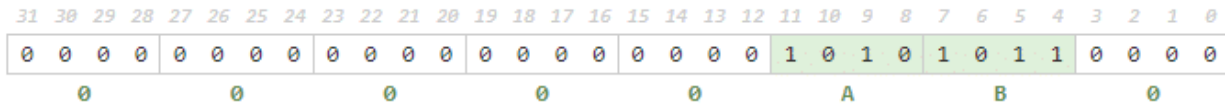
```
MOV    r1, #0x12    @ r1 = 0x00000012
@Rotate r1 left 12 bits by ROR 20 bits (32 - 20 = 12); answer in r6
ROR    r6, r1, #20    @ r6 = 0x00012000
```

[Try Sample](#)

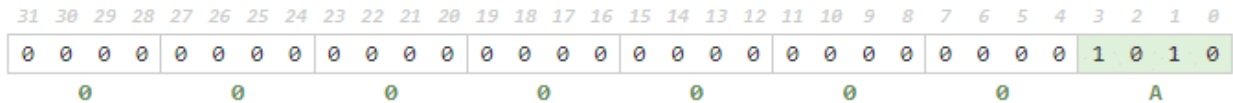
Tip: To rotate left X bits, we rotate 32 - X to the right

Shifts

The shift instructions shift all the bits in a register to by a given amount. However, unlike with rotate, when bits move past position 0 or 31, they disappear. Shifting this pattern:



8 bits to the right would end up “dropping” the 8 rightmost bits:



Since the bits do not wrap, we can’t fake a left shift with a right one.

LSL *rd, rn, rn / #*

Logical Shift Left. Shift *rn* to the left the number of bits in *rn* or *#*. Result in *rd*.

LSR *rd, rn, rn / #*

Logical Shift Right. Shift *rn* to the right the number of bits in *rn* or *#*. Result in *rd*.

```
MOV r1, #0x12    @ r1 = 0000 ... 0000 0001 0010
@Shift 4 BITS or 1 hex digit to the right
LSL r2, r1, #4    @ r2 = 0000 ... 0001 0010 0000
@Shift 2 BITS to the left
LSR r3, r1, #2    @ r3 = 0000 ... 0000 0000 0100
@Shift r1 by number of bits in r3 (4), result in r5
LSL r5, r1, r3
```

[Try Sample](#)

Multiplying and Dividing via Shift

Shifting bits that represent a binary number to the left one place is equivalent to multiplying it by 2. Here, binary for 5 gets shifted and becomes 10 (A):

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	
0				0				0				0				0				0				5							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	
0				0				0				0				0				0				A							

Shifting 2 bits is equivalent to multiplying by 4. Each additional bit shifted represents an additional power of 2.

```
MOV    r1, #5
LSL    r2, r1, #1 @ r6 = 10 or 0xA
LSL    r3, r1, #2 @ r6 = 20 or 0x14
LSL    r4, r1, #8 @ r6 = 5 * 256 = 1280 or 0x500
```

[Try Sample](#)

Conversely, each bit shifted to the right divides a binary pattern by 2. Here is 24 (0x18) shifted right to produce 12 (0xC):

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
0				0				0				0				0				0				1				8			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
0				0				0				0				0				0				0				C			

However, when we shift to the right, and the pattern represents a **two's complement** value, shifting in a 0 for the leftmost bit can cause an error. Watch what happens if we shift -2

(0xFFFFFFFF) to the right one bit - we get 0x7FFFFFFF or 2,147,483,647:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	
F				F				F				F				F				F				E							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
7				F				F				F				F				F				F							

To get the right answer, we need to copy the leftmost bit as we shift the pattern. Here is a correct version of -2 / 2 to produce -1 (0xFFFFFFFF):

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	
F				F				F				F				F				F				E							

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
F				F				F				F				F				F				F							

This is the point of the Arithmetic Shift Right instruction - it does a shift that preserves the sign of a two's complement number:

ASR *rd, m, rm / #*

Arithmetic Shift Right. Shift *m* to the right the number of bits in *rm* or *#*, copying the leftmost bit to fill the void. Result in *rd*.

```
MOV    r5, #50
ASR     r6, r5, #1      @ r6 = r5 / 2
ASR     r7, r5, #3      @ r7 = r5 / 8
MOV     r8, #-200
ASR     r9, r8, #6      @ r9 = r8 / 64 = 3
```

[Try Sample](#)

Tip: Shifting *n* bits to the left multiplies a value by 2^n . Shifting to the right divides by 2^n as long as you preserve the leftmost bit.

Operand Shifts

The architecture of the ARM processor provides a shift unit in the data path between the registers and the ALU. Because of its location, it is possible to shift one of the operands (always the second one) as it is brought to the ALU from the register file.

We do this by using ROR, ASR, LSL, or LSR

Since the bits do not wrap, we can't fake a left shift with a right one.

INSTRUCTION *rd, m, rm, LSL/LSR/ASR/ROR #*

Perform a basic INSTRUCTION like MOV or ADD. However, do the given shift to the value from *rm* before using it. *rm* itself is not affected.

```

MOV    r1, #0x5
@Take r1, left shift value by 1 bit (multiplies by 2) and move into r2
MOV    r2, r1, LSL #1    @r2 = r1 * 2
LSL    r2, r1, #1        @Same!!!

@Use ASR 2 bits to divide by 4 while moving value
MOV    r3, #-100
MOV    r4, r3, ASR #2    @r4 = r3 / 4

@Use use LSL 2 bits to multiply value of r8 by 4 while adding to itself
@ to produce 5 * r8
MOV    r8, #3
ADD    r9, r8, r8, LSL #2 @r9 = r8 + r8 * 4
                        @ = 5 * r8

@Calculate (2 * r5) - r6 in one line using LSL and ReverseSubtract
MOV    r5, #6
MOV    r6, #9
RSB    r7, r6, r5, LSL #1 @r7 = r5 * 2 - r6

```

[Try Sample](#)

Multiply

Shifts are efficient ways to multiply and divide by powers of 2, but sometimes we want to multiply by other values. Fortunately, there is a multiply instruction:

MUL *rd, m, rm*

Multiply *m* by *rm*. Result in *rd*. An immediate value cannot be used – all operands must be registers.

```

MOV    r1, #5
MOV    r2, #2

MUL    r3, r1, r2    @r3 = r1 * r2
MUL    r3, r3, r3    @r3 = r3 * r3

```

[Try Sample](#)

The design of the ARM processor (at least historically) allowed for the data path to efficiently add a value to a product that had just been calculated. Being able to multiply two values and immediately sum them with something can be useful for accumulating a total while performing a loop over a series of calculations. This operation is handled by the MLA instruction:

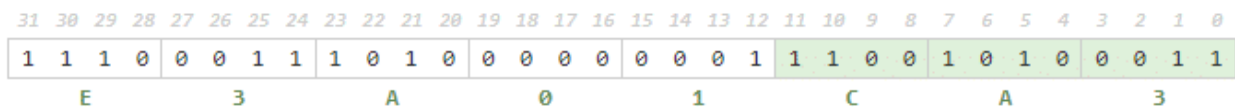
The last 12 bits are the immediate bits. Normally, 12 bits would allow us to store a value up to 4096. However, it is quite possible that we would like to have a larger value as an immediate, so the value is not stored as a plain 12-bit number. Instead, it is stored as an 8-bit number with a 4-bit shift amount:



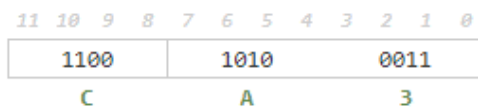
To calculate the actual value an immediate represents, we:

- Take the shift amount as a binary value and double it
- Rotate the value bits to the right the doubled shift, wrapping as needed
- Read the resulting binary value

For example, `MOV r1, #41728` looks like this:



CA3 is not 41,728. Instead, those three hex chars should be interpreted as a rotation of 24 (2 * C) done to 10100011 (A3):



This scheme means not all binary patterns can be represented. We can only represent patterns that require 8-bits of significant digits (the first to last 1 must span over no more than 8 bits). And because we double the rotate bits, the final rotation amount must always be an even number of bits.

In practical terms, we can use a wide range of values as immediate, including some large values. But there will be values that can't be represented and result in an error if we try to use them. To work with values that do not fit as an immediate, we will have to build them up in multiple steps or load them into a register from memory.

```

@Legal - requires a rotate
MOV    r1, #41728

@Legal - requires rotating FF
MOV    r2, #0xFF0000

@Legal - no rotate required for values < 256
MOV    r2, #0xA0

@Illegal - can't be represented - comment out to fix
MOV    r2, #10000

```

[Try Sample](#)

Tip: The error message *Error: invalid constant after fixup* tells you you have an illegal immediate value.

MEMORY

Directives and Identifiers

Directives

Directives are instructions for the assembler about what to do with the information that comes after them. Directives always start with a `.` like: `.word` or `.section`. We can use directives to say things like “here is a word (32-bit pattern) to place into memory”:

```
.word 0xABCDABCD
```

[Try Sample](#)

Identifiers

Identifiers are how we name lines of code. They do not become code, they are just labels we can use later on to refer to a particular piece of data or instruction. Identifiers always need to

end with a : and follow normal rules for naming things in programming – no spaces or odd characters, first character must be alphabetical – like X: or milesPerHour: or input1:. The identifier may be on the same line as the thing it names or on the preceding line:

```
@This word of memory can be referred to as X
X: .word    0xABCDABCD

@This word of memory can be referred to as pattern
pattern:
.word    0x12345678
```

[Try Sample](#)

Sections

Assembled code is broken into logical sections that are used in different ways. The bits that represent instructions must be executable but should generally not be writable. The bits that represent data should be readable and writable but may not make sense to execute.

Some common sections:

- **text** stores the code of the program
- **data** stores global data that can be read or written
- **rodata** stores global information that is Read Only (constants)
- **bss** has an uninitialized blank space that we have reserved for use by the program to store data

Directives are used to specify what section various parts of an assembly program belong to:

```

.section      .data
myGlobal:    .word    0xC

.section      .rodata
MY_CONSTANT: .word    0x64

.section      .bss
uninitializedGlobal: .space 4

.text
.global _start
_start:
@Do nothing...
MOV    r1, #0

end:    b end        @stop program

```

[Try Sample](#)

That code gets build into what is shown below. Some things to note:

- The sections appear in a specific order: text, rodata, data, then bss.
- Sections may be **padded** with extra 0's to make them occupy a number of bytes that is some specific power of 2. We can see this below at the end of the .data, .bss. and .rodata sections: they have all been padded so that the next section starts at an address that is a multiple of 8.

00000000	e3a01000	_start: 14 mov r1, #0 ; 0x0 end:
00000004	ea000000	b 0x4 (0x4: end)
00000008	00000000	MY_CONSTANT: andeq r0, r0, r4, RRX andeq r0, r0, r0
0000000c	00000000	myGlobal: andeq r0, r0, r12 andeq r0, r0, r0
00000010	00000000	uninitializedGlobal: andeq r0, r0, r0 andeq r0, r0, r0
00000014	00000000	
00000018	00000000	
0000001c	00000000	

Note: Most of the time, this tutorial will stick to using just using .data for all data and .text for the code. We will generally not use .rodata or .bss.

Data & Alignment

There are different directives for storing values in different sized chunks of memory:

- **.word** stores a machine word - a 32-bit (4 byte) chunk
- **.hword** stores a half word value in a 16-bit (2 byte) chunk
- **.byte** stores an 8-bit (1 byte) chunk

In addition, we can use **.space** to allocate a specified number of bytes of empty space.

Chunks of memory must be aligned in order for instructions to access them. What this means is that words must always start at an address that is a multiple of 4, and hwords must start at an address that is a multiple of 2. We use the .align directive to say “fill out this word of memory” with 0’s so the next data starts at a multiple of 4.

Warning

Failure to align memory when switching between bytes and words is a common mistake. The program will assemble just fine, but when a load instruction tries to access miss-aligned memory, there will be an error.

When in doubt, add .align any time you have allocated some bytes and need to switch to allocating words. If you use an .align when the next chunk of memory would already be aligned, the directive will be safely ignored.

Here are some samples of using the data and align directives:

```

.data
@a and b take up 1 byte each
a: .byte 4
b: .byte 0xBB

.align @adds 0's to fill until word boundary

@Word is 4 bytes
@Should align to 4 bytes before placing word
c: .word 0xCCCCCCCC

@d takes up 2 bytes (half-word)
d: .hword 0xDDDD

@e/f can go in same word as d
e: .byte 0xEE
f: .byte -1 @0xFF

@g: starts a new word
g: .byte 0x12

.text
MOV r1, r1 @do nothing

```

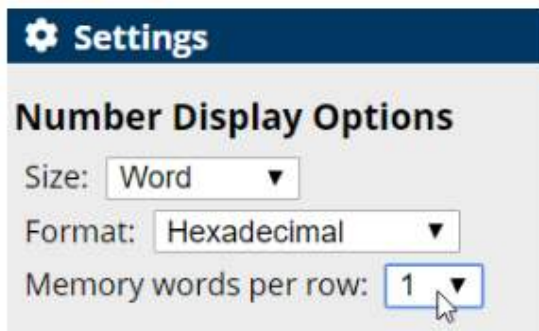
[Try Sample](#)

Although you can view the memory in the Disassembly view, it is a little confusing to see “instructions” listed for the data. The Memory tab of cpulculator allows you to see a plain view of memory:

Address	Memory contents and ASCII			
00000000	e1a01001	00000000	0000bb04	cccccccc
00000010	ffeedddd	00000012	aaaaaaaa	aaaaaaaa
00000020	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa
00000030				

 Editor (Ctrl-E)
 Disassembly (Ctrl-D)
 Memory (Ctrl-M)

By default, 4 words are shown per row. You may find that makes it harder to figure out individual addresses. If so, you can force the Memory view to display just one word per row:



Address	Memory contents and ASCII
00000000	e1a01001
00000004	00000000
00000008	0000bb04
0000000c	cccccccc
00000010	ffeedddd
00000014	00000012
00000018	aaaaaaaa

In memory, we can see the MOV instruction (e1a01001) and the 00000000 that the .text is padded with. After that, the .data section begins. First are the bytes a and b (04 and BB), then the word c, etc...

To find a piece of memory, you can use the symbols panel:

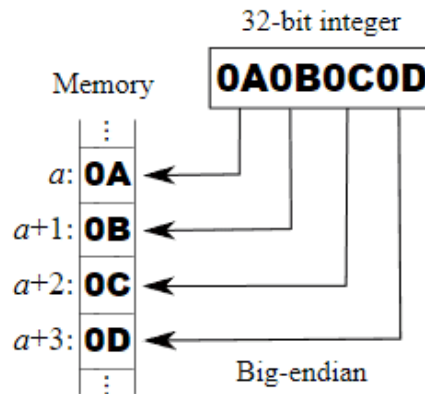
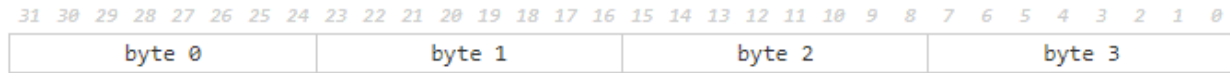
Symbols		
Symbol	Address	
a	00000008	</> Q
b	00000009	</> Q
c	0000000c	</> Q
d	00000010	</> Q
e	00000012	</> Q
f	00000013	</> Q
g	00000014	</> Q
_end	00000018	</> Q

Registers Call stack Trace
Breakpoints Watchpoints
Symbols Counters

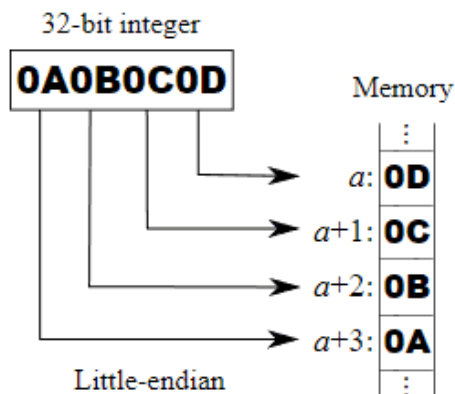
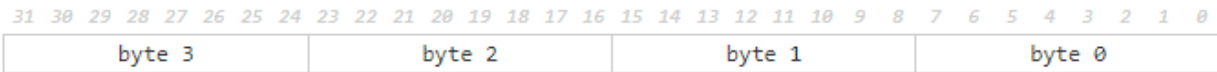
Endianness and Address Numbering

Main memory is addressable at the byte level - we can specify the address of any 8-bit chunk. That means that a machine word, 32-bits in ARMv7, consists of 4 bytes of memory. The order those bytes are numbered in is called **endianness**.

In a **big-endian** processor, the bytes are numbered from left to right with the leftmost byte being the lowest address:



In a little-endian processor (like ARM or Intel), the bytes are numbered from right to left.



Notice that the view in memory looks reversed - 0D is the first value that appears in memory, followed by 0C, then 0B, and finally 0A.

If you go back to the Data & Alignment page and look closely at how the data was stored into memory, you will notice that some of it looks out of order. This is due to endianness.

Note: ARM (and intel) processors are Little Endian - which means that sometimes the 4 bytes of a word in memory will look like they are backwards. Little endian english thgim kool ekil siht (might look like this) - the words are in order, but the letters are backwards.

This interesting numbering system can be seen in the memory view. Here are the first 48 bytes of memory loaded with values corresponding to their address (address 0 has 0x00, address 1 has 0x01...):

Address	Memory contents and ASCII			
00000000	03020100	07060504	0b0a0908	0f0e0d0c
00000010	13121110	17161514	1b1a1918	1f1e1d1c
00000020	23222120	27262524	2b2a2928	2f2e2d2c

Notice that each row of the memory display contains four words, or 16 bytes, of memory. Addresses are always shown in - the address 10 is not ten, but is 0x10 or 16. Row 1 has addresses 0x0-0xF (0-15), row 2 has 0x10-0x1F (16-31)... Words are in the expected order: addresses 0-3 come before 4-7, but the bytes within each word read backwards: 3, then 2, then 1, then 0.

When working with a full word of data, you do not have to worry about endianness - the 32-bit chunk will take up 4 bytes and you don't really care how they are ordered. You will need to worry about endianness when:

1. You take/send data from a machine that is one endianness to a machine that is the opposite form.
2. You work with individual bytes or half words.

This code sample shows the difference between loading a pattern as a word and loading it as a series of bytes and half words.

```

@allocate 2 words with given hex values
word1:    .word    0xAABBCCDD
word2:    .word    0xEEFF0011

.space 8   @padding between items

@allocate 4 bytes with given values
bytes1:    .byte    0xAA, 0xBB, 0xCC, 0xDD
chunk:     .hword    0xEEEE
bytes2:    .byte    0x00, 0x11

```

[Try Sample](#)

It results in this memory:

Address	Memory contents and ASCII		
00000000	aabbccdd	eeff0011	0000
00000010	ddccbbaa	1100eeee	aaaa

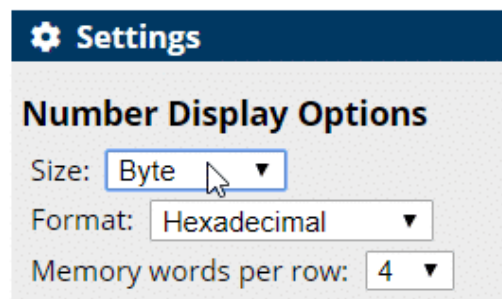
The first row in the picture shows hex addresses 0x0-0x7 (0x8-0xF are cut out of the picture). The two words were loaded into them:

address	0x03	0x02	0x01	0x00	0x07	0x06	0x05	0x04
data	AABBCCDD				EEFF0011			

The second row shows hex addresses 0x10-0x17. The first byte loaded - 0xAA is in address 0x10, but that is not the leftmost address! Address 0x11 has the next byte, etc... as shown below:

address	0x13	0x12	0x11	0x10	0x17	0x16	0x15	0x14
data	DD	CC	BB	AA	11	00	EEEE	

However, if you ask the simulator to display numbers in bytes:



It displays bytes in numerical order: 0x00, 0x01, 0x02, ... Notice how in byte mode the memory allocated as words (starting at address 0x00) looks jumbled up, while the memory allocated as bytes (starting at 0x10) looks correct.

Address	Memory contents and ASCII
00000000	dd cc bb aa 11 00 ff ee
00000010	aa bb cc dd ee ee 00 11

The same data displayed in byte mode.

Loading & Storing

ARM is a load/store architecture, meaning that most of the instructions can only work on registers. To work with data we must first load it into a register. When we are done working on it, we store it back to memory.

To load data we use the load register instruction. It typically is a two step process. First we load the address of the data that we want, then we load the data itself. We need this two step process because the memory address we want to load from is a 32-bit value – it cannot be included inside the instruction as it is too big. The address must be loaded into a register that can hold the full 32 bit value.

LDR *rd*, =LABEL

Load ***rd*** with the address that corresponds to the thing labeled with LABEL. This gets the address of the data, not the data itself. Similar to the & operator in C/C++

LDR *rd*, [*rn*]

Load ***rd*** with the word of memory at the location stored in *m*. This loads the actual data pointed at by the register. This is similar to the * operator in C/C++

```
.data
x:  .word  0x12345678

.text
LDR  r1, =x      @r1 <- address of x
                        @int* r1 = &x; in c++

LDR  r2, [r1]     @r2 <- value at address stored in r1
                        @int r2 = *r1; in C++

end:  b end       @stop program
```

[Try Sample](#)

Note: If you look closely at the assembled version of that code, you will notice that in the assembled version, the `=x` has been replaced by `[pc, #4]`.

00000000	e59f1004	5 LDR r1, =x @r1 <- address of x
		ldr r1, [pc, #4] ; 0xc
		6 @int* r1 = &x; in c++
00000004	e5912000	8 LDR r2, [r1] @r2 <- value at address
		ldr r2, [r1]
		9 @int r2 = *r1; in C++
00000008	eaaffffe	end:
		b 0x8 (0x8: end)
0000000c	00000010	5 LDR r1, =x @r1 <- address of x
		andeq r0, r0, r0, LSL r0
00000010	12345678	x:
		eornes r5, r4, #125829120 ; 0x78000000

`[pc, #4]` means 4 bytes past where the Program Counter currently is. When the first LDR runs, the program counter will be 00000000 (the address of that instruction). In early version of the ARM architecture, the PC would actually be 8 bytes past that when the instruction executed. So the final address would be $0x00000000 + 4 + 8 = 12 = 0xc$. If you look at address 0000000c, it contains the data 0x00000010. 0x00000010 is in fact the address of x. The assembler has: 1) calculated the address of x; 2) stored that value in memory at the end of the .text; 3) set the first LDR to load that value. That is all fairly complex, but most of the time, we do not have to worry about the mechanics. We just use `LDR rd, =LABEL` and count on the assembler to manage the details.

Loading an Immediate

We can use the assembler's tricks to have it load a value that is too large/complex to fit as the immediate value in a MOV instruction. A load can specify a numeric value to load, the assembler will place it into memory and automatically calculate the address for it.

LDR *rd*, =value

Load *rd* with the value specified. The value is not an immediate - it does not get # in front of it.

```
LDR    r1, =0xABCDABCD
LDR    r2, =45000

end:   b end      @stop program
```

[Try Sample](#)

Note: A load instruction may be less efficient than MOV instructions that do not rely on a trip to memory. ARMv7 provides `MOVT` and `MOVW` instruction that each can move a 16-bit chunk of data into a register. `MOVW` loads the lower 16-bits of a register and wipes out the top 16-

bits. MOVT places 16-bits in the top 16-bits of the register, leaving the lower 16 bits unmodified. So a MOVW followed by a MOVT is likely a more efficient way to load a 32-bit value than loading it.

Storing

To store a piece of data back to memory after working on it in registers, we use the store register instruction:

STR *rs*, [*rn*]

Take the value that is stored in *rs* and store it into memory at the location in *rn*. Note that the destination is the second register!

The store register instruction needs to have the address it will store to loaded into a register, so it will usually be preceded by a LDR *rn*, =identifier instruction that loads the memory address we are going to store to. This program loads data from a variable *x*, multiplies it by 2 and stores the answer back to *y*:

```
.data
x:  .word  5
y:  .word  0

.text
_start:
LDR  r1, =x      @r1 <- address of x (&x)
LDR  r2, [r1]    @r2 <- x

LSL  r2, r2, #1  @r2 <- r2 * 2

LDR  r3, =y      @r3 <- address of y (&y)
STR  r2, [r3]    @store from r2 to location given by r3
                        @y <- r2

end:  b end      @stop program
```

[Try Sample](#)

Loading & Storing Bytes

Sometimes, we don't want a whole word of memory – we just want to access a single byte. To do this, we use the LDRB instruction:

LDRB *rd*, [*rn*]

Load Register Byte. Load *rd* with the byte of memory at the location stored in *rn*. The upper 3 bytes of the register will be zeroed out.

Say we load 0xFF into a register – we would end up with:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0				0				0				0				0				0				F				F			

That would be the correct value if that 0xFF is just a bit pattern or if it represented an unsigned number (255). However, if that 8-bit pattern is being used to represent a signed value, we have a problem. 0xFF is -1. But 0x000000FF is 255. To get the correct 32-bit value, we need to sign extend the byte as it is loaded. We need to copy the most significant bit from the byte (bit 7) into bits 8-31:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
F				F				F				F				F				F				F							

Any positive 8-bit signed value will start with a 0. So when it gets sign extended, bits 8-31 get 0's. Here is 100, hex 0x64, getting sign extended:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0
0				0				0				0				0				0				6				4			

To handle loading with sign extension, there is an alternative load byte instruction:

LDRSB, *rd* [*rn*]

Load Register Signed Byte. Load *rd* with the byte of memory at the location stored in *rn*. All the digits in the upper three bytes will be filled with the most significant bit from the byte loaded. (It will be sign extended.)

Tip: We still use LDR to load the byte's address. Then we use LDRB to load an 8-bit value that is either unsigned or not a number, or LDRSB to load an 8-bit signed number.

```
.data
b: .byte 34 @ 0x22
c: .byte -1 @ 0xFF
d: .byte 0 @

.align

.text
LDR r3, =b @r3 <- address of b
LDRB r3, [r3] @r3 <- b

LDR r4, =c @r4 <- address of c
@INCORRECT - Load negative value as unsigned:
LDRB r4, [r4] @r4 <- 0x000000FF = 255!!!

LDR r4, =c @r4 <- address of c
@CORRECT - Load negative value as signed:
LDRSB r4, [r4] @r4 <- 0xFFFFFFFF = -1
```

[Try Sample](#)

Storing Bytes

To store just a single byte, we use the STRB instruction. It writes 8-bits of memory, leaving any neighboring bytes unchanged. Because we are going from a 32-bit value (the register), to an 8-bit block of memory, there is no need for a separate signed version - bits 8-31 are just chopped off as the value is written. (Which does mean that if the register held a value too large to represent with 8 bits, the wrong value will be written to memory.)

STRB *rs*, [*rn*]

Take the lower byte from *rs* (bits 0-7) and store it to address *rn*.

This code sample adds one to the value *a* in memory by loading it, incrementing the value, then writing it back:

```

.data
a: .byte 6 @ 0x05

.align

.text
LDR    r1, =a      @r1 <- address of a
LDRSB  r2, [r1]     @r2 <- a

ADD     r2, r2, #1   @r2 <- a + 1
STRB    r2, [r1]     @a <- r2

```

[Try Sample](#)

BITWISE OPERATIONS

Bitwise Logic 1

The instructions AND, ORR, EOR, and MVN function as the bitwise logical operations and, or, exclusive or, and not. These bitwise operations combine either two registers or a register and an immediate by comparing bit 0 of the first to bit 0 of the second, bit 1 of the first to bit 1 of the second, etc...

AND

The AND instruction compares the individual bits of two patterns using logical AND: produce a 1 if both inputs are 1, 0 otherwise. In these two patterns:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0
0				0				0				0				0				0				3		C					

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	1
0				0				0				0				0				0				6		D					

Bits 2, 3, and 5 are the only ones that are 1 in both - thus this output is produced:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0
0				0				0				0				0				0				2		C					

AND $rd, rn, rm / \#$

Combine the bits of rn and the bits of rm or $\#$ with AND. Result in rd .

ORR

ORR compares the individual bits of two patterns using logical **OR**: produce a 1 if either or both inputs are 1, 0 otherwise. In these two patterns:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1	0	0	
0				0				0				0				0				0				A		C					

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	1
0				0				0				0				0				0				F		1					

Bits 0, 2, 3, and 4-7 are 1 in at least one of the two inputs - thus this output is produced:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	1
0				0				0				0				0				0				F		D					

ORR $rd, rn, rm / \#$

Combine the bits of rn and the bits of rm or $\#$ with OR. Result in rd .

```
@AND keeps 1 if both inputs are 1
MOV    r1,    #0x3C          @ r1 = 0000 ... 0011 1100
@      #0x6D          0000 ... 0110 1101
AND     r2, r1, #0x6D          @ r2 = 0000 ... 0010 1100 or 0x2C

@ORR keeps 1 from either pattern:
MOV     r4,    #0xAC          @ r4 = 0000 ... 1010 1100
MOV     r5,    #0xF1          @ r5 = 0000 ... 1111 0001
ORR     r6, r4, r5            @ r6 = 0000 ... 1111 1101 or 0xFD
```

[Try Sample](#)

Bitwise Logic 2

EOR

The **EOR** instruction compares the individual bits of two patterns using logical **exclusive or**: produce a 1 if one of the two inputs is 1, produce 0 if the input bits are the same. In these two patterns:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1	0	0
0				0				0				0				0				0				A				C			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1	0
0				0				0				0				0				0				F				6			

Bits number 1, 3, 4, and 6 are 1 in exactly one of the inputs:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0
0				0				0				0				0				0				5				A			

EOR *rd, rn, rm / #*

Combine the bits of *rn* and the bits of *rm* or *#* with exclusive or. Result in *rd*.

NOT

There is no instruction named **NOT**, but the **MVN** instruction does the job of logical **NOT**. It takes a pattern and flips all the bits so 1's become 0's and vice versa. This pattern:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	1	0	0
0				0				0				0				0				0				A				C			

Produces this when negated:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	0	1	1	
F				F				F				F				F				F				5				3			

MVN *rd, rn / #*

Place the bitwise negation of *rm* or *#* in *rd*. Does logical NOT.


```

@EOR gives 1 if bits are different
MOV  r7,    #0xac          @ r7 = 0000 ... 1010 1100
@
EOR   r8, r7, #0xFF        @ r8 = 0000 ... 0101 1010

MOV   r9,    #0xAC          @ r9 = 0000 ... 1010 1100
MVN   r10, r9               @ r10 = 1111 ... 0101 0011
MVN   r11, r10              @ r11 = 0000 ... 1010 1100

```

[Try Sample](#)

3.3. Masking

Masking is the process of using bitwise operations to isolate part of a binary pattern. As an example, here are the bits that represent the ARM machine code for ADD r1, r2, r3. Bits 12-15 have the destination register:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	1
E				0				8				2				1				0				0				3			

To isolate that register from the pattern, we could **AND** with 0xF000 (hex for 1111 0000 0000 0000):

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0				0				0				0				F				0				0				0			

The bits that were ANDed with 1, will retain their value. The bits that were ANDed with 0, are forced to 0:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0				0				0				0				1				0				0				0			

Recall that immediates can generally be no more than two consecutive hex chars padded with os (like 0xFF or 0xA4000 but not 0xF00F where the two non-zero chars are spaced out). Those patterns can be masked directly with an AND. Larger patterns must be built up in a register before being used as a mask:

```

LDR    r5, =0x1BADDEED

@isolate bits 0-7 (last byte: 0xED)
@ & means bitwise OR
AND    r6, r5, #0xFF      @r6 = 0x1BADDEED & 0x000000FF = 0x000000ED

@isolate bits 8-15 (next to last byte: 0xDE)
AND    r7, r5, #0xFF00    @r7 = 0x1BADDEED & 0x0000FF00 = 0x0000DE00

@isolate bits 16-27 (0xBAD from 0x1BADDEED)
@ want pattern 0xFFFF0000, but that takes too many bits to represent, so build it
MOV    r8, #0x00FF0000    @Start with 00FF0000
ORR    r8, r8, #0x0F000000 @r8 = 00FF0000 | 0F000000 = 0FFF0000
AND    r8, r5, r8          @r8 = 0x1BADDEED & 0x0FFF0000 = 0x0BAD0000

@isolate bits 2-5 :
@Use mask      0000 ... 0011 1100 = 3C
@r5 has        ??? ... 1110 1101 (...ED)
@ result ==    0000 ... 0010 1100 = 0x2C
AND    r8, r5, #0x3C      @r8 = 0x1BADDEED & 0x0000003C = 0x0000002C

end:    b end            @stop program

```

[Try Sample](#)

A mask leaves the bits in their original location. If we want to use the masked value as a number, we would have to shift its bits so the leftmost one is aligned with the 0 bit.

If our goal is to end with the desired bits aligned to the right of the register, it may be simpler to just use a pair of shifts to clear extra bits and align the ones we want with bit 0. As an example, we will isolate bits 12-15 in the pattern below and align them with bit 0:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0
E				0				8				2				1				0				0				3			

First we need to clear out everything past bit 15. We can do this by left shifting by 16 bits:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1				0				0				3				0				0				0				0			

Then, we shift back to the right. There are 32 total bits, and we want to preserve just 4. So shift right 28 bits:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0				0				0				0				0				0				0				1			

```

LDR    r5, =0xE0821003

@isolate bits 12-15 using just shifts
LSL    r9, r5, #16      @r9 = 0x10030000
LSR    r9, r9, #28      @r9 = 0x00000001

LDR    r6, =0x1BADDEED
@isolate bits 16-27 (0xBAD from 0x1BADDEED) using just shifts
LSL    r10, r6, #4       @r10 = 0xBADDEED0
LSR    r10, r10, #20     @r10 = 0x00000BAD

```

[Try Sample](#)

Clearing

Clearing bits is the opposite of masking. Instead of wanting to keep certain bits, we want to clear out specific bits while keeping other ones. For instance, we might want to clear out bits 25-29 from this pattern:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1
			1			B				A					D				D				E			E				D	

To do so, we could mask by ANDing with this pattern:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
			C				1				F				F				F				F				F			F	

Bits 25-29 would become 0 and everything else would remain as is. However, that pattern is too complex to represent as an immediate. Any time we are trying to clear a small number of bits, we will have a similar problem of needing a large mask to AND with.

Instead, we can use the bitwise clear instruction - it uses a pattern to specify which bits to clear (ANDing with a pattern specifies which bits to keep).

BIC *rd, rn, rm / #*

Bit Clear. Take the pattern from *rm*, clear out the bits that are 1 in either register *rm* or immediate *#*, place the results in *rd*.

```

LDR    r5, =0x1BADDEED

@Clear bits 25-29
BIC    r6, r5, #0x3E000000

@Clear bits 16-31 (first two bytes)
BIC    r7, r5, #0xFF000000 @clear first byte
BIC    r7, r7, #0x00FF0000 @now clear second

```

[Try Sample](#)

BRANCHES AND CONTROL STRUCTURES

Branch

The unconditional branch instruction allows us to skip to a new location to continue executing code from. When the branch executes, the program counter is not increased by 4 bytes as normal, but instead, its value is replaced with the address of a new instruction.

B label

Branch. Change the program counter to the address of the line identified by *label*.

In this code sample, line number 3 has a branch that will cause execution to skip over the code on lines 4 and 5 and execute the next line after the label target, which is line 7.

```

1      MOV    r1, #1    @random work
2      MOV    r2, #2
3      B      target    @branch to target
4      MOV    r3, #3    @skipped
5      MOV    r4, #4    @skipped
6      target:
7      MOV    r5, #5

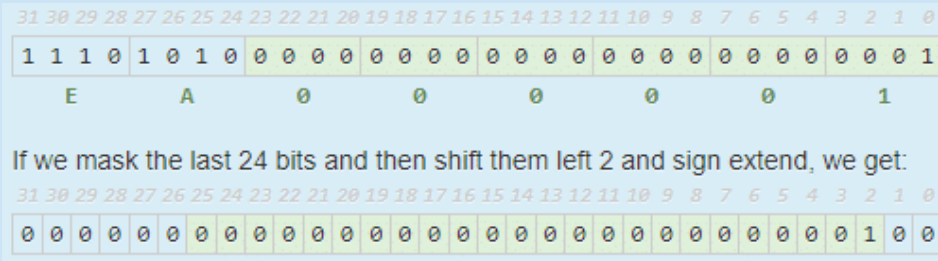
```

[Try Sample](#)

Note: The assembler calculates the destination address and stores it in the instruction as +/- value to add to the PC register when the branch executes. This offset is stored as 24-bits in the instruction:



When we show up to the present moment with all of our senses, we invite the world to fill us with joy. The pains of the past are behind us. The future has yet to unfold. But the now is full of beauty simply waiting for our attention.



The final offset value is 4. To that value, we have to add 8. (This add 8 is due to the historical design of the ARM architecture where the PC will already have advanced 2 instructions or 8 bytes before the branch address is actually used). So the final amount for the branch is 12 bytes, or at 4 bytes per instructions, 3 instructions ahead. Sure enough, the shown branch skips over 2 instructions and runs the third one after the B.

Condition Codes

The ARM processor maintains a special register known as the Current Program Status Register or CPSR.

cpsr | 800001d3 |

The CPSR is used to track information about what mode the processor is in and the result of recent instructions. In particular, the upper 4 bits of the register are used to track the following information:



- N Was the result negative?
- Z Was the result zero?

- C Was there a carry out of the leftmost bit? (Though this bit is used in complex ways that do not always mean carry.)
- V Was there mathematical overflow? (For signed math this is not the same as having a carry out of the leftmost bit.)

Tip: The exact rules for setting the carry bit are complex and depend on the type of comparison instruction being executed. You are not responsible for the details, just recognize that the carry bit does not always mean a literal carry.

These values are only set by specific instructions. The most straightforward of these instructions is **compare**:

CMP *rn*, *rm* /

CoMPare. Subtract *rm* or # by from *rn* and update the status registers based on the results. The result of the subtraction is ignored.

In this code sample below the value 10 is compared with something larger, something smaller, and itself. The only result of running the CMP instructions is to change the CPSR.

```

1  MOV    r1, #10      @load x's address
2
3  CMP    r1, #20      @do r1 - 20, set result bits
4  @result bits are now 0x8 or 0b1000 : only Negative is set
5
6  CMP    r1, #2       @do r1 - 2, set result bits
7  @result bits are now 0x2 or 0b0010 : only carry bit was set
8
9  CMP    r1, #10      @do r1 - 10, set result bits
10 @result bits are now 0x6 or 0b0110 : carry and zero bits are set
11
12 end:
13 B      end          @stop here

```

[Try Sample](#)

Conditional Branches

To use the condition bits in the CPSR, we can use **condition codes** that correspond to various logical combinations of the bits. Here are the most common codes:

Code	Meaning	Bits
EQ	Equal	Z set
NE	Not Equal	Z clear
GE	Greater or Equal	N == V
LT	Less Than	N != V
GT	Greater Than	(Z == 0) && (N == V)
LE	Less or Equal	(Z == 1) (N != V)

Tip: You are not responsible for memorizing the bits associated with each code. Focus on making sure you recognize the codes.

There are also some less common codes:

Code	Meaning	Bits
MI	Negative (MInus)	N set
PL	Positive or zero (PLus)	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
CS	Unsigned higher or same	C set
CC	Unsigned lower	C clear

These codes can be added to a B instruction to modify the circumstances under which it branches. Like BEQ for “branch if equal” or BGT for “branch if greater than”.

The ARM sample below uses a conditional branch to approximate this C++ code:

```
if(x == 2)
    y = 5;
```

On line 10, the assembly does a comparison between r1, where x has been loaded, and 2. Then, on line 11, the branch checks the condition bits set by that operation. If they indicate that the

values were not equal to each other, it skips down to the label on line 18. If the values were equal, it continues on after line 11 to the next instruction at 14.

```
1      .data
2      x:      .word    2
3      y:      .word    0
4
5      .text
6      _start:
7      LDR     r1, =x      @Load x's address
8      LDR     r1, [r1]    @r1 = x
9
10     CMP     r1, #2      @test r1 vs 2 (calculate r1 - 2)
11     BNE     endIf       @branch past "if body" if r1 Not Equal to 2
12
13     @These instructions only executed if r1 was == 2
14     MOV     r2, #5
15     LDR     r3, =y      @Load y's address
16     STR     r2, [r3]    @y = 5
17
18     endIf:
19     MOV     r0, r0      @do nothing... rest of program here
```

[Try Sample](#)

Warning: Before using the comparison codes, you need to do a comparison to set the condition bits!

If Else Implementation

You may have noticed that the logic for conditionals in assembly is the reverse of high level languages. In high level languages we use conditionals to specify extra code to run under certain circumstances:

```
if(x < 2)
    y = 5;
//rest of program
```

In assembly, a branch always skips code. So our branch logic has to be written to decide when to skip a group of instructions, not when to run them. Instead of asking “is x less than 2”, the compare and branch statement in the code below effectively say “if the value in r1 is greater than or equal to 2 don’t set y to 5”:


```

1      .data
2      x:      .word  2
3      y:      .word  0
4
5      .text
6      _start:
7      LDR     r1, =x      @Load x's address
8      LDR     r1, [r1]    @r1 = x
9
10     CMP     r1, #2      @test r1 vs 2 (calculate r1 - 2)
11     BGE     endIf      @branch past "if body" if r1 Not Equal to 2
12
13     @These instructions only executed if r1 was == 2
14     MOV     r2, #5
15     LDR     r3, =y      @Load y's address
16     STR     r2, [r3]    @y = 5
17
18     endIf:
19     MOV     r0, r0      @do nothing... rest of program here

```

[Try sample](#)

To write an if else in assembly, we generally use the form:

```

test whether to skip to else
...if code...
skip to end else
else:
...else code...
endElse:

```

So this C++:

```

if(x < 2)
    y = 5;
else
    y = 3;

```

Would become:

```

1      .data
2      x:      .word    1
3      y:      .word    0
4
5      .text
6      _start:
7      LDR     r1, =x      @load x's address
8      LDR     r1, [r1]    @load x
9
10     CMP     r1, #2      @test x vs 2 (x - 2)
11     BGE     else        @branch to else if x Greater or Equal
12
13     @if part
14     MOV     r2, #5
15     B       endElse     @done with if part... get past else
16
17     else:
18     @else part
19     MOV     r2, #3
20
21     endElse:
22     @done with if/else
23     LDR     r3, =y      @load y's address
24     STR     r2, [r3]    @y = value set in if/else

```

[Try Sample](#)

Loops

A branch that targets an earlier instruction sets up an assembly loop. In this sample, instruction 4 branches back to instruction 3 causing an infinite loop that adds 1 to r1.

```

1      MOV     r1, #0      @init r1
2
3      loopStart:
4      ADD     r1, r1, #1   @ r1++
5      B       loopStart

```

[Try Sample](#)

To turn this into a counting loop, we can add a conditional branch that checks to see if we should exit the loop. The assembly below is a literal translation of:

```

r1 = 0
while(r1 < 5)
    r1++

```

Lines 4 and 5 check to see if r1 is now greater than or equal to 5. If so, it is time to exit the loop. Otherwise, we add one to r1, and then restart the loop:

```
1      MOV    r1, #0      @init r1
2
3      loopStart:
4      CMP    r1, #5
5      BGE    endLoop     @branch to endLoop when r1 >= 5
6      ADD    r1, r1, #1   @ r1++ (r1 was < 5)
7      B      loopStart    @go back to start of loop
8
9      endLoop:
10     MOV    r0, r0      @do nothing... rest of program here
```

[Try Sample](#)

Improving the Counting Loop

The sample above requires executing four instructions per loop. Even if we do not take the branch at line 5, we have to execute the instruction.

But, we can rewrite the loop to reduce the amount of work per iteration. The heart of this trick is moving the loop test to the end of the loop and having it decide “do we need to repeat the loop” instead of “do we need to exit the loop”. This is done on lines 8 and 9 of the sample below:

To make sure that the loop test is run before executing the body, we jump to the test to begin executing the loop - this is done at line 3. Now, every iteration of the loop only requires executing three instructions. If the loop is going to iterate thousands of times, cutting the instructions per iteration from four to three could make a significant difference for execution time. So this is the approach compilers take when creating a loop.

Improved counting loop:

```

1      MOV    r1, #0      @init r1
2
3      B      loopTest    @kick start loop by jumping ahead to test
4
5      loopStart:
6      ADD    r1, r1, #1  @ r1++
7      loopTest:
8      CMP    r1, #5      @compare r1 to 5
9      BLT    loopStart   @if r1 Less Than 5, go back to loopStart
10
11     MOV    r0, r0      @do nothing... rest of program here

```

Try Sample

Doing Work in Loops

To do work in a loop, we just have to place the work we want to do in the body of the basic loop structure. Below is an example of implementing ARM assembly roughly equivalent to this C++ code:

```
int sum = 0;
int max = 10;

//r2 will be our counter "variable" - however it will just live in a register and not be stored into memory
for(r2 = 1; r2 <= max; r2++)
    sum += r2;
```

```

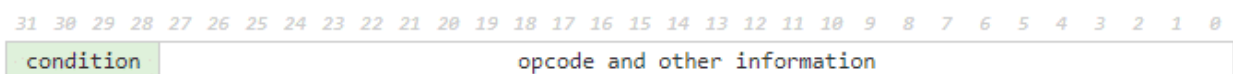
1      .data
2      sum:  .word  0
3      max:  .word  10
4
5      .text
6      _start:
7      LDR    r1, =max      @r1 = &max
8      LDR    r1, [r1]      @r1 = max
9
10     MOV     r2, #0        @r2 is counting variable (i)
11     MOV     r3, #0        @r3 is sum
12
13     @Use test at end loop strategy
14     B       loopTest      @Start loop by jumping to test
15     loopStart:
16     ADD     r3, r3, r2     @sum += r2
17     ADD     r2, r2, #1     @r2++
18     loopTest:
19     CMP     r2, r1         @Compare counter in r2 to max in r1
20     BLE     loopStart      @branch to start if i (r2) <= y (r1)
21
22     LDR     r4, =sum        @r4 = &sum
23     STR     r3, [r4]        @store r3 back to sum to record final answer

```

[Try Sample](#)

ARM Trick - Conditional Execution

In ARM machine code, the first four bits of most instructions are reserved for a condition code:



In this assembled example code, there are a series of branches with different conditions. Notice that the first hex digit of the instruction is different for each one. The second hex digit is always 'a', which corresponds to 'branch'. (The last 24 bits, the addresses, are different as well, as each instruction is one instruction closer to the branch target test.)

ea000005	2	B test	b 0x1c (0x1c: test)
0a000004	3	BEQ test	beq 0x1c (0x1c: test)
1a000003	4	BNE test	bne 0x1c (0x1c: test)
ca000002	5	BGT test	bgt 0x1c (0x1c: test)
aa000001	6	BGE test	bge 0x1c (0x1c: test)
ba000000	7	BLT test	blt 0x1c (0x1c: test)
da000000	8	BLE test	ble 0x1c (0x1c: test)

These same four conditional bits also exist in most basic (movement, bitwise, arithmetic) instructions. Which means that instead of using a compare and branch to skip instructions, we can do a comparison and use the result of it to specify whether or not to actually do the work in following instructions.

In the sample below, lines 13 and 14 are both always executed. But, if the condition code is

Line 13 only executes if the comparison on line 11 indicated equality. Line 14 only executes if it did not.

```

1  .data
2  x:      .word  2
3  y:      .word  0
4
5
6  .text
7  _start:
8  LDR     r1, =x      @Load x's address
9  LDR     r1, [r1]     @r1 = x
10
11  CMP     r1, #2      @test r1 vs 2 (calculate r1 - 2)
12
13  MOVEQ   r2, #5      @only executed if CMP set status indicating values Equal
14  MOVNE   r2, #3      @only executed if CMP set status indicating values Not Equal
15
16  LDR     r3, =y      @Load y's address
17  STR     r2, [r3]     @store result back to y

```

[Try Sample](#)

Why does this trick exist? Those 4 bits are valuable real estate in the machine instruction format. The answer is efficiency.

Using conditional execution in the sample above resulted in 3 lines of code to implement the if/else. An implementation using branches would result in 5 lines. Furthermore, executing branches can be a major source of slow downs in modern pipelined processors. So even in a larger if/else structure - one where the if and else each contained 5 instructions - we might very well be better off always executing all 10 instructions and just ignoring 5 of them than using branches to only execute the correct five instructions.

Note

Fun fact: the condition code for “always” is 1110 or hex E. That is why most normal instructions in ARM assembly start with E.

ARM Trick - Set Bits

Another little trick that ARM assembly allows is for instructions that do arithmetic or logic to set the condition bits while calculating their answer. We do this by adding S to the end of the instruction:

xxxS ??, ??, ??

Set condition bits. Does the instruction xxx but also updates the condition bits depending on the result of the instruction.

Say we want to implement this logic:

```
r1 = r1 - 10
if( r1 < 0 )
    r1 = 0
```

Normally, we would subtract 10, then do a comparison, then do a branch or conditional execution of the `r1 = 0`. Using this trick, we can do the following where only two instructions total are needed for the subtraction and if:

```
MOV    r1, #5        @init r1
SUBS    r1, r1, #10    @r1 -= 10
                        @Update status flags based on result
MOVMI    r1, #0        @set r1 to 0
                        @If status flags show MINus (negative)
MOV     r0, r0        @do nothing... rest of program here
```

[Try Sample](#)

We can do a similar thing to make a counting loop where changing the counter also does the comparison that will help us determine whether the loop is done:

```
MOV     r1, #5        @init r1
B       loopTest      @branch to test
loopStart:
SUBS    r1, r1, #1    @r1--
                        @Set the status flags based on result
loopTest:
BPL     loopStart      @branch to start
                        @If status flags indicate PLUS (zero or positive)
```

[Try Sample](#)

ARRAYS AND STRINGS

Array Basics

An array is simply a block of memory that is being used to store 1 or more same sized elements. In ARM assembly, we can create an array by assigning a label to a list of allocations or to a block of memory reserved with the `.space` directive:

```
.data
@array of 4 words (32-bit integers)
myArray:    .word    5, 6, 8, 9

@array of 6 bytes (8 bit integers)
myBytes:    .byte    10, 11, 12, 13, 14, 15

.align      @make sure we are padded to a full word
@always do this after allocating bytes to make sure any words that come later are word ali
gned

@40 bytes or 10 words (integers) worth of empty space
myEmptyArray: .space  40

@Easy to spot pattern at end of storage
.word       0xFFFFFFFF
```

[Try Sample](#)

There are no instructions in that sample, but it does lay out the arrays in memory. To see, this, switch to the memory view. In it, you should be able to find the words of memory 5, 6, 8, and 9. They are followed by bytes of memory for 10-15 (0a-0f), then some padding to fill out the word that of is in, then 40 bytes of space, then 0xFFFFFFFF.

Address	Memory contents and ASCII			
00000000	00000005	00000006	00000008	00000009
00000010	0d0c0b0a	00000f0e	00000000	00000000
00000020	00000000	00000000	00000000	00000000
00000030	00000000	00000000	00000000	00000000
00000040	ffffffff	00000000	aaaaaaaa	aaaaaaaa

To help find the pieces of memory, you can use the Symbols tab. If you switch to it, all the labels you have declared will be shown along with their address:

🔍 Symbols		
Symbol ▲	Address	🔍
myEmptyArray	00000018	</> 🔍
myBytes	00000010	</> 🔍
myArray	00000000	</> 🔍
_end	00000048	</> 🔍

🔍 Registers	🔍 Call stack	🔍 Trace
🔍 Breakpoints	🔍 Watchpoints	🔍 Symbols
🔍 Counters		

Use the magnifying glass to highlight the location in memory. Note that the highlight shows the word at that address, it does not do anything to show you the size of the array that starts there.

Array Access

We access information in arrays in the same way we do any other piece of memory. First we use a LDR instruction to load the address of the memory, then we use an LDR with that address to load the data itself.

What is different about arrays is that we expect to find multiple units of storage starting at the address we are working with. To access elements other than the first, we need to calculate the correct address by adding a number of bytes to the base address – an **offset**. The **offset** will be the index number of the element we desire times the size of each element in bytes. For example: if we want the 10th item in an array of words, we would add $10 * 4 = 40$ bytes.

As programmers, we have to do the math to calculate the right **offset**. But we don't have to add it to the base address ourselves. Instead, we can specify it as a value to add to the address found in a register using this syntax:

LDR *rd*, [*rn*, #]

Access the memory at the address $rn + \#$, and store its value into *rd*. # should be $i * \text{element_size}$ where *i* is the index of the element desired and *element_size* is the number of bytes per element. Similar [*rn*, #] syntax is available for STR, LDRB, LDRSB, and STRB

This code sample demonstrates setting `myArray[4] = myArray[0] + myArray[1]`:

```

.data
@An array of 4 words
myArray:    .word    5, 10, 15, 20, 0
@myArray will store address of first word
@ each additional one if 4 bytes after the last

.text
LDR    r1, =myArray    @ Load myArray start address

LDR    r2, [r1]         @ r2 = value at address of myArray
                        @      = myArray[0]

LDR    r3, [r1, #4]     @ r2 = value at address of myArray + 4 bytes
                        @      = myArray[1]

ADD    r4, r2, r3       @ r4 = myArray[0] + myArray[1]

STR    r4, [r1, #16]    @ store r4 (sum) to 16 bytes after start of myArray
                        @ myArray[4] = r4

```

[Try Sample](#)

Tip: To see the STR instruction's effects, you will have to have the Memory tab of the simulator showing. You can dock it next to the Disassembly tab so that both views are visible.

Also, when you restart the simulator, the memory does not get reset - make sure to Reload if you want to watch it run again.

```

.data
@an array of 4 SIGNED bytes
myArray:    .byte    -3, 4, 10, 0

.align      @make sure we are padded to a full word
@not needed here, but good to always do it

.text
_start:
    LDR    r1, =myArray    @ Load arrays start address

    LDRSB r2, [r1]         @ r2 = myArray[0] as Signed Byte

    LDRSB r3, [r1, #1]     @ r3 = myArray[1] (1 byte from start)

    ADD    r4, r2, r3       @ r4 = myArray[0] + myArray[1]

    @Store Register r4 back to array as Byte
    STRB   r4, [r1, #3]    @ myArray[2] (2 bytes from start) = r4 (sum)

```

[Try Sample](#)

Looping Through Arrays

To loop through an array, we need to access successive memory addresses: something like x , $x + 4$, $x + 8$, ... To use the `LDR rd, [rn, #]` syntax, we would have to list each memory address by hand. Alternatively, we could load x 's address into a register, and then modify that address each time we were ready to access the next item. But it is often useful to retain the base address of the array, so overwriting our copy of its address may not make sense.

To provide a flexible way to access parts of an array, another form of indicating an address for load and store instructions is provided for in ARM:

LDR *rd, [rn, rm]*

Access the memory at the byte address $rn + rm$, and store its value into rd . Similar `[rn, rm]` syntax is available for STR, LDRB, LDRSB, and STRB

If rn holds our base address, then rm can hold our offset – any time we access memory, those values will be added together.

This code sample demonstrates looping through an array using this technique. Note that we calculate the address of items in the array on line 24 by adding the base address stored in $r2$ with the offset stored in $r5$ and every time through the loop we add 4 to the offset (on line 29).

```

1  .data
2  @an array of 5 ints
3  myArray:  .word  10, 20, 30, 40, 50
4
5  arraySize:  .word  5    @size of myArray
6
7
8  .text
9  _start:
10 LDR  r1, =arraySize    @ r1 = &arraySize
11 LDR  r1, [r1]          @ r1 = arraySize
12
13 LDR  r2, =myArray      @ Load myArray start address
14                                @ r2 will always point at array start
15
16 MOV  r3, #0            @ r3 = loop counter
17 MOV  r5, #0            @ r5 = bytes into array to current element
18 MOV  r4, #0            @ r4 = total
19
20 B    looptest          @ jump ahead to loop test
21
22 looptest:
23 @Calculate address of current element by adding offset r5 to base address
24 LDR  r6, [r2, r5]      @ r6 = element at address r2 + r5
25 ADD  r4, r4, r6        @ total += current value
26
27 @go to next step
28 ADD  r3, r3, #1        @ add one to counter
29 ADD  r5, r5, #4        @ add 4 to offset value for "current element"
30
31 looptest:
32 CMP  r3, r1            @ Compare counter r3 to size of array in r1
33 BLT  looptest          @ If counter Less Than size, go back to start
34
35 end:
36 B    end              @stop here

```

[Try Sample](#)

Looping over byte arrays

If we are working with bytes of memory, the index (loop counter) and offset will always be the same, so we can use one register to represent both. Index 2 will be 2 bytes in, index 5, will be 5 bytes in, etc...

The code sample below totals the values in myArray using a loop. It is just like the sample above except that r3 serves as both the array counter (index) and the memory offset for each element. And as we load elements of the array we have to use LDRSB to load them as bytes.

```
1  .data
2  @an array of 4 SIGNED bytes
3  myArray:    .byte  4, 2, -1, 10
4  .align    @make sure we are padded to a full word
5  myArraySize: .word  4
6
7  .text
8  _start:
9  LDR    r1, =myArraySize    @ r1 = &myArraySize
10 LDR    r1, [r1]            @ r1 = myArraySize - it is a word, use LRB
11
12 LDR    r2, =myArray        @ load myArray start address
13                                     @ r2 will always point at array start
14
15 MOV    r3, #0              @ r3 = loop counter
16 MOV    r4, #0              @ r4 = total
17
18 B      looptest            @ jump ahead to loop test
19
20 looptest:
21 @Load array item - it is a byte so use LDRSB
22 LDRSB  r6, [r2, r3]        @ r6 = element at address r2 + r3
23 ADD    r4, r4, r6          @ total += current value
24
25 @go to next step
26 ADD    r3, r3, #1          @ add one to index
27
28 looptest:
29 CMP    r3, r1              @ Compare counter r3 to size of array in r1
30 BLT    looptest            @ If counter Less Than size, go back to start
31
32 end:
33 B      end                  @stop here
```

[Try Sample](#)

Array Loop Shifted Index

The loop shown on the last page to loop over an array of words works fine, but uses a lot of registers for book keeping. We use one register for the base address of the array, another for the index of the array (to test when we are done), and another to keep track of the memory offset.

But, the index and offset are redundant. For word index 0 the offset will always be 0, for word index 1 the offset will be 4, for index 2 the offset will be 8,... The offset is always 4x the index.

The ARM instruction set provides a trick to avoid storing the index and 4x the index. When we go to use the index of an array to load data, we can shift it 2 bits. This effectively multiplies it by 4, turning the value 0 into 0, 1 into 4, 2 into 8, etc...

LDR *rd*, [*rn*, *rm*, *lsl* #]

Access the memory at the byte address $rn + (rm \text{ shifted left } \# \text{ bits})$, and store its value into *rd*. Typically, we will use a shift of 2 to multiply *rm* by 4. Similar [*rn*, *rm*, *lsl* #] syntax is available for STR.

This code sample below demonstrates looping through an array using this technique. It copies data from **myArray** to the empty space at **doubled** and doubles each element of the array as it copies it. It is roughly equivalent to:

```
int myArraySize = 5;
int myArray[5] = {10, 20, 30, 40, 50};
int doubled[5];

int i = 0;
while(i < myArraySize) {
    doubled[i] = 2 * myArray[i];
    i++;
}
```

In the assembly version, we hold the base address of the original array in r2 and the address of doubled in r3. r4 is the index - to use it to find the offset of an element in an array we shift it left 2 bits.


```

1  .data
2  @An array of 5 ints
3  myArray:  .word  10, 20, 30, 40, 50
4  arraySize: .word  5      @size of myArray
5
6  doubled:  .space  20      @20 bytes = 5 words
7
8
9  .text
10 _start:
11 LDR    r1, =arraySize      @ r1 = &arraySize
12 LDR    r1, [r1]            @ r1 = arraySize
13
14 LDR    r2, =myArray        @ Load myArray start address
15 LDR    r3, =doubled        @ Load newArray start address
16
17 MOV    r4, #0              @ r4 = Loop counter
18
19 B      looptest            @ jump ahead to loop test
20
21 looptest:
22 @myArray[i]'s address is r2 + (4 * r4)
23 LDR    r5, [r2, r4, lsl #2] @ r5 = myArray[i]
24
25 LSL    r5, r5, #1          @double r5
26
27 @doubled[i]'s address is r3 + (4 * r4)
28 STR    r5, [r3, r4, lsl #2] @ doubled[i] = r5
29
30 @go to next step
31 ADD    r4, r4, #1          @ add one to counter
32
33 looptest:
34 CMP    r4, r1              @ Compare counter r4 to size of array in r1
35 BLT    looptest            @ If counter Less Than size, go back to start
36
37
38 end:
39 B      end                 @stop here

```

[Try Sample](#)

Strings

Strings in assembly are simple an array of signed bytes (chars) interpreted by their ASCII code. 'A' is stored as 0x41, 'B' as 0x42, etc... By convention, they usually end with a NULL character - the value 0. This convention allows us to allocate large buffers (chunks of memory), only use the parts we need, and then later identify how much of the buffer is actually in use.

For example, given this chunk of memory:

0x41	0x70	0x70	0x6c	0x65	0x00	0x4f	0x92	0xa0	0x07	0x1c	0x1c	0x3d	0x82	0xbc	0x0f
------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------

We would interpret it to mean 0x41 0x70 0x70 0x6c 0x65 - the 0x00 says “stop reading here”. Anything else in that buffer would be ignored.

The standard way to load a string is via the `.asciz` directive. (The z stands for the 0 at the end). It loads a given string into memory and adds a null byte 0x00 to end it. The sample below loads the chars ‘A’ ‘p’ ‘p’ ‘l’ ‘e’ and places a 0x00 after them:

```
.data
.asciz  "Apple"

@use 0xFF to show where string ends
.byte 0xFF
```

[Try Sample](#)

Displayed in byte mode (so that endianness doesn’t change the order we see the chars in), the result looks like this:

Address	Memory contents and ASCII
00000000	41 70 70 6c 65 00 ff 00 aa aa aa aa aa aa aa aa Appl e...

Note: There also is an `.ascii` which does not add a null byte to the end.

To loop over a string, we use a loop that looks for a null char. The code sample below counts the ‘e’s in a string. It is roughly equivalent to the C++:

```
int eCount = 0;
int i = 0;
while(myString[i] != '\0') {
    char current = myString[i];
    if (current == 'e')
        eCount += 1;
    i++;
}
```

In the assembly version `r` holds the eCount, `r2` the base address of the string, and `r3` the index. Since each char is a single byte, we can use base address + index to calculate the memory address of each char:

```

1  .data
2  myString:  .asciz  "hello there people"
3  .align    @force alignment to word boundary
4
5
6  .text
7  _start:
8  MOV     r1, #0      @ r1 = eCount
9
10 LDR     r2, =myString @ r2 = base address of myString
11 MOV     r3, #0      @ i = 0 - r3 will be index
12
13 B       looptest    @ Jump to loop test
14
15 loopstart:
16     @We know that r4 has current char from doing test
17     CMP     r4, #'e'    @ compare current char to e (0x65)
18     BNE     endIf
19     ADD     r1, #1      @ if equal, add one to counter
20
21     endIf:
22     ADD     r3, r3, #1  @ i++
23
24 looptest:
25 LDRB     r4, [r2, r3]  @ r4 = current char (myString[i])
26     @ Load as a BYTE!!!
27
28 CMP     r4, #0      @ 0 char signifies end of string
29 BNE     loopstart    @ If not null char, go through a loop iteration
30
31 end:
32 B       end          @stop here

```

[Try Sample](#)

FUNCTIONS AND THE STACK

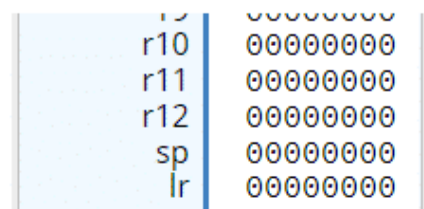
The Stack

The stack is a region of memory typically used by compilers for managing the memory required by functions within a program. It is used by compilers to allocate space for local

variables in functions, other book keeping information (like where a function returns to), and to store values from registers when we need to use that register for some new purpose.

The stack gets its name because of the way memory is allocated within it. Memory is never added or removed from the middle of the stack - it is strictly managed by adding or removing to the top. When memory is required, space is allocated at the top of the stack. When that memory is no longer required, it is removed from the stack.

A register is used to keep track of the memory address for the top of the stack. r13 is usually referred to as the sp or **stack pointer**. Although programs are free to use this register for any purpose, it is assumed that it will be used to track the top of the stack. When a program starts up, the stack pointer will be initialized to a memory address where the program can start building the stack. In the simulator, when we load any program, the sp is set to 0x00000000:



r10	00000000
r11	00000000
r12	00000000
sp	00000000
lr	00000000

This starting location is considered to be already in use (called a full stack). We can't add anything new to that address, we first need to move the stack pointer.

In most implementations, the stack grows down in memory - to add something new we place it at a lower memory address than the previous item. This is called a descending stack. (We will not be looking at the alternative: ascending stack.)

In a descending stack, the bottom of the stack is the highest address and the top of the stack is the lowest address. When we refer to the top of the stack, we will be talking about the lowest memory address.

Tip: Different environments will have different initial addresses for the top of the stack. When you are programming on real Linux hardware, do not assume it will use the same address for the stack.

Working with the stack

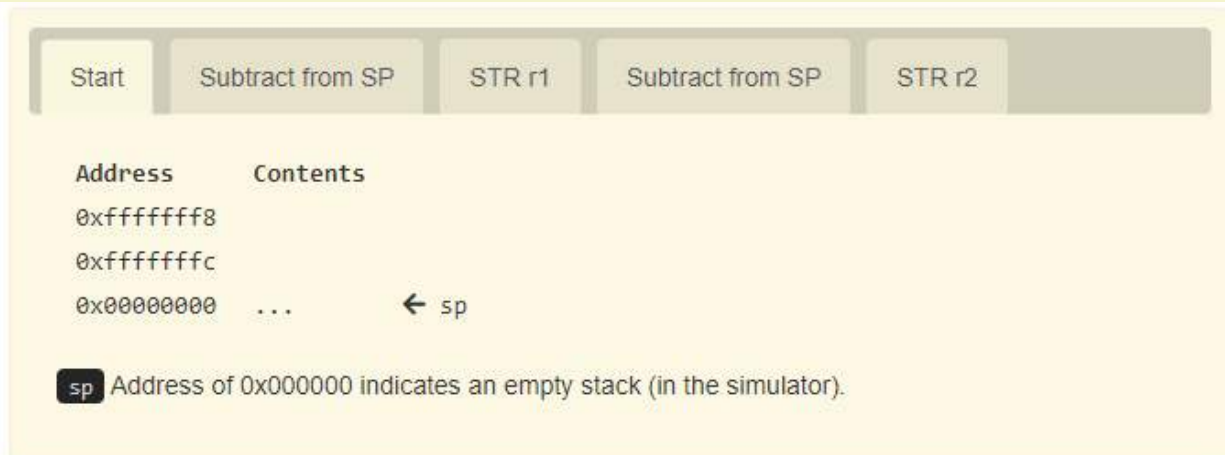
The sp always points to the top byte of memory on the stack (which is the lowest memory address). To add memory to the stack, we first have to subtract the number of bytes desired

from the sp to find the address to use for the new item. When we are done with memory and are ready to remove it from the stack, we add the same number of bytes to the sp to “remove”. The data is not actually removed from memory, but by moving the stack pointer, we are effectively forgetting that piece of data.

The stack is most commonly used to store register values. Since each register is 4 bytes, we typically add or subtract 4 from the sp. To actually load and store the values from memory, we use LDR and STR just like we were accessing any other data in memory.

This sample demonstrates storing the values from r1 and r2 to the stack, then wiping out the registers, before restoring them from memory. Note that we store r1 before r2, meaning that r2 is on the top of the stack (lowest memory address).

Warning: The diagrams and images below all show the stack building up... but note that the memory addresses actually get smaller as we go up in the stack.



When we go to remove value from the stack, we start with the top of the stack (lowest address). In this case, that means we have to remove r2 before r1.

As we restore values from the stack to registers, we add to sp to “remove” items from the stack. The values that were placed in memory remain there, but we no longer consider them to be part of the stack. We will reuse the space next time we add something to the stack.

This sample shows removing the two value from the stack and then adding r3 back to the stack:

Start

LDR r2, add to SP

LDR r1, add to SP

Subtract from SP

STR r3

Address	Contents
0xffffffff8	r2 ← sp
0xffffffffc	r1
0x00000000	...

r2 is the top item on the stack.

Tip: Stacks always work in a Last In, First Out (LIFO) fashion. You must remove things in the opposite order you add them.

```

1  .text
2  .global _start
3  _start:
4  @Some initial values
5  MOV    r1, #0xAA
6  MOV    r2, #0xBB
7  MOV    r3, #0xCC
8
9  @Store r1 to the stack
10 SUB    sp, sp, #4    @Move stack pointer down 4 bytes
11 STR    r1, [sp]      @Copy r1 to that address
12
13 @Store r2 to the stack
14 SUB    sp, sp, #4    @Move stack pointer down 4 bytes
15 STR    r2, [sp]      @Copy r2 to that address
16
17 @Wipe out registers
18 MOV    r1, #0x0
19 MOV    r2, #0x0
20
21 @r2 is at the top of the stack, it must be removed before r1
22 LDR    r2, [sp]      @Restore r2 from stack
23 ADD    sp, sp, #4    @Move stack pointer up 4 bytes
24
25 @restore r1 from stack and remove it
26 LDR    r1, [sp]      @Restore r1 from stack
27 ADD    sp, sp, #4    @Move stack pointer up 4 bytes
28
29 @Store r3 to the stack
30 SUB    sp, sp, #4    @Move stack pointer down 4 bytes
31 STR    r3, [sp]      @Copy r2 to that address
32
33 end:
34 B      end

```

[Try Sample](#)

r8	00000000			Address	Memory contents and ASCII
r9	00000000			ffffff10	aaaaaaaa
r10	00000000			ffffff4	aaaaaaaa
r11	00000000			ffffff8	000000bb
r12	00000000			ffffffc	000000cc
sp	fffffffc			00000000	e3a010aa
lr	00000000				

The final state of memory. Note that the stack pointer does not end up back at 0x00000000 meaning the program left something on the stack. Normally you would not want to do that.

Pushing and Popping

Because storing registers to the stack is so common, ARM has instructions that move the stack pointer and load or store memory at the same time: PUSH and POP

PUSH {*ra, rb, rc, ...*}

Stores the listed registers to the top of the stack and adjusts the stack pointer. The registers must be listed in order: if you want to push r1, r4, and r7 they must be listed as {r1, r4, r7}. Registers are stored in order so that the lowest numbered register is at the lowest address (top of the stack).

The result of pushing r1, r4 and r7 to an empty stack:

Address	Contents	
0xfffffff4	r1	← sp
0xfffffff8	r4	
0xfffffffc	r7	

0x00000000	...	
------------	-----	--

POP {*ra*, *rb*, *rc*, ...}

Loads the listed registers to the stack and adjusts the stack pointer to remove that many words. Registers are popped in order (smallest number first) so if you push a list of registers and then pop the same list, the values end up in the same locations. The registers must be listed in order: if you want to pop *r1*, *r4*, and *r7* they must be listed as {*r1*, *r4*, *r7*}.

The result of popping *r8*, *r9*, and *r10* on the stack shown above:

Address	Contents	
0xffffffff4	<i>r1</i>	Would be stored in <i>r8</i>
0xffffffff8	<i>r4</i>	Would be stored in <i>r9</i>
0xfffffff0c	<i>r7</i>	Would be stored in <i>r10</i>
0x00000000	...	← sp

Note that the registers we popped value back into are different than the ones we pushed. This would have the effect of copying *r1*, *r4*, and *r7* into *r8*, *r9*, and *r10* respectively.

This sample uses the PUSH and POP instructions to store three registers and then restore their values after they are wiped:

```

1  .text
2  .global _start
3  _start:
4  @Some initial values
5  MOV    r1, #0xAA
6  MOV    r2, #0xBB
7  MOV    r3, #0xCC
8
9  @Store to stack using PUSH
10 PUSH   {r1, r2, r3}
11
12 @Wipe out registers
13 MOV    r1, #0x0
14 MOV    r2, #0x0
15 MOV    r3, #0x0
16
17 @POP Removes from stack in back to front order. Sample below
18 @ removes top of stack into r3, next word into r2, next word into r1
19 POP    {r1, r2, r3}
20
21 end:
22 B      end

```

[Try Sample](#)

🏠 Registers		🔍 Memory (Ctrl-M)	
Refresh		Go to address, label, or register:	
		Address	Memory contents and
r0	00000000	ffffffec	aaaaaaaa
r1	000000aa	ffffff0	aaaaaaaa
r2	000000bb	ffffff4	000000aa
r3	000000cc	ffffff8	000000bb
r4	00000000	ffffffc	000000cc
r5	00000000	00000000	e3a010aa
r6	00000000	00000004	e3a020bb
r7	00000000	00000008	e3a030cc
r8	00000000	0000000c	e92d000e
r9	00000000	00000010	e3a01000
r10	00000000	00000014	e3a02000
r11	00000000	00000018	e3a03000
r12	00000000		
sp	00000000		
lr	00000000		

The final state: the stack pointer is back to 0x00000000 and r1-r3 have been restored from the values that were pushed on to the stack.

Branch and Link

In assembly, a function is essentially a set of instructions that we can branch to, run and return from. While the B instruction is sufficient to jump to a new location before running the next instruction, it does not leave us with any information about how to return to where we were.

Normally, the pc register stores the address of the current instruction. So if we want to remember where we were, we could copy the pc into another register or to the stack before changing it. Then, we could use that stored value to return to where we had been. (If the goal was to return after taking a branch, we would actually want to return to the instruction that is just after the branch to avoid immediately branching again.)

Rather than do this process by hand every time we want to use a branch to run a function, we can use the **branch and link** and **branch exchange** instructions that automates the process. BL does a branch and sets register r14, also known as the **link register** lr to the address right after the BL. BX lr branches to the address stored in the lr.

```

5 .text
6 .global _start
7 _start:
8     @do pointless work
9     MOV    r0, r0
10
11     @Branch to myFunction
12     @set Link register to address of next line after this
13     BL     myFunction
14
15     MOV    r1, r0    @copy r0 to r1
16     MOV    r0, #0    @clear r0
17
18     @Branch to myFunction
19     @set Link register to address of next line after this
20     BL     myFunction
21
22     @stop here
23 end:
24     B      end
25
26
27 /*
28 * Minimal function that just needs to return to
29 * where we came from.
30 */
31 myFunction:
32     MOV    r0, #10    @put 10 in r0
33     @return to where we came from
34     BX     LR         @return

```

BL label

Branch and Link: Branches to the memory location identified by **label** and sets the link register, lr, to the address of the instruction after the BL.

BX rd

Branch and eXchange instruction set: Branches to the address stored in register **rd**. To return after a BL instruction, we can use lr as the register. The eXchange refers to the fact that we can use this instruction to switch from 32-bit arm instructions to 16-bit thumb instructions, but we will not be using that feature.

This code sample demonstrates the operation of the BL instruction. It twice jumps to the location labeled myFunction and returns from it. Note that BL sets the lr register as it executes and BXLR jumps back to the stored location.

```
1  /*
2     Demonstrates behavior of BL instruction
3  */
4
5  .text
6  .global _start
7  _start:
8      @do pointless work
9      MOV    r0, r0
10
11     @Branch to myFunction
12     @set Link register to address of next line after this
13     BL     myFunction
14
15     MOV    r1, r0    @copy r0 to r1
16     MOV    r0, #0    @clear r0
17
18     @Branch to myFunction
19     @set Link register to address of next line after this
20     BL     myFunction
21
22     @stop here
23 end:
24     B      end
25
26
27  /*
28  * Minimal function that just needs to return to
29  * where we came from.
30  */
31  myFunction:
32     MOV    r0, #10    @put 10 in r0
33     @return to where we came from
34     BX     lr
```

[Try Sample](#)

Calling Conventions

As control is transferred from one location to another, we also need to transfer information and coordinate register use. We want to be able to pass parameters to functions and have them return values. As those functions do work, they will be using the same registers that the rest

of the program does - we need to make sure that one part of the program does not overwrite an important piece of information stored in a register by another part of the program.

If we were hand coding an entire program in assembly, we could manage registers however we wanted. But keeping track of which ones were in use at every point of a large program would be a complex undertaking. And we generally don't sit down to write an entire program by hand. We use tools to compile code and make use of code that others have written. For arbitrary pieces of code to work together, they can't count on any special knowledge ("I know that register r8 is not being used right now, so I'll put my data there"). Instead, we need an agreed on set of rules for how to pass information from function to function and who gets to use which registers.

Calling conventions are the rules for how functions transmit information and use registers and memory. They will necessarily differ between architectures (ARM and Intel have different registers available) but also may differ between operating systems and even different compilers for the same platform. We will focus on the calling conventions generally used in ARM v7 by the GNU C/C++ compiler - they are a fairly representative set of rules.

In general, register 0-9 are used for program data. r0-r3 are used for temporary storage and information passing. There is never any expectation that other code will preserve values in these registers. r4-r9 are used for longer term storage and functions are expected to not destroy values in them:

Register Use:

Register(s)	Use
r0 - r3	Parameters and return values. Also used for temporary values - any function can change these at will.
r4 - r9	Stored values. Any values stored in

	these must be preserved by functions.
r10 - r15	Special purpose. These have assigned roles (like pc or lr). Not all the roles are always used.

Calling and returning from a function always involves five phases. Control starts at the caller, which has to possibly save information it has in registers r0–r3 and then call the function. The called function needs to gather and save information (the prologue), execute its code, and then clean up and return (the epilogue). Finally, the caller needs to restore any saved values and then work with any returned value. Here is a summary of what happens:

A Simple Function Call

When we show up to the present moment with all of our senses, we invite the world to fill us with joy. The pains of the past are behind us. The future has yet to unfold. But the now is full of beauty simply waiting for our attention.

Calling Procedure (Basic)

Function Call (Done by caller)

Any data in r0–r3 that needs to be stored is pushed to the stack.

Parameters are placed in r0, r1, r2, and r3 (in order).

Use BL to branch to the function.

Function Prologue (Done by called function)

If the function will use any registers in r4–r9, push the current values to the stack.

Function Body (Done by called function)

Do the work of the function. Access passed parameters in r0–r3. Only modify registers r0–r3 and any registers that were saved in the prologue.

Function Epilogue (Done by called function)

Place any return value(s) in r0, r1, r2, r3. (C/C++ generally only return one value, but there is no reason in assembly you can't return more than one.)

Pop any stored registers (r4-r9) from the stack to restore their old values.

Return to the caller with BX lr

Resume Control (Done by caller)

Any returned value(s) are in registers r0-r3.

Pop stored registers (r0-r3) that were preserved before calling.

A Sample Function

The code sample below demonstrates a program that replace r4 and r5 with their absolute values. To do this, it uses an abs function. To call the function, the main part of the program stores r1, which is in use, then places the parameter (value to take absolute value of) in r0. The function preserves, r4 and r5 so it can use them, then does its work. It finishes up by placing the answer in r0, restoring r4 and r5 and branching back. Then the main part of the program moves the returned value out of r0 and restores r1.

Here is how the stack is used to store values along the way:

Start	At line 12	At line 51	At line 62	At line 16
-------	------------	------------	------------	------------

Address	Contents
0xffffffff4	
0xffffffff8	
0xfffffffcc	
0x00000000	... ← sp

```

1  /*
2     Main program is using registers 1, 4 and 5
3     It is responsible for storing/restoring r1 while calling function
4  */
5  .text
6  .global _start
7  _start:
8      MOV    r1, #0xAA
9      MOV    r4, #30
10     MOV    r5, #-12
11
12     PUSH    {r1}    @I want to keep r1 safe, so save it
13     MOV     r0, r4    @set up parameter for abs call
14     BL      abs
15     MOV     r4, r0    @save result from the function back to r4
16     POP     {r1}    @restore r1
17
18     PUSH    {r1}    @I want to keep r1 safe, so save it
19     MOV     r0, r5    @setup param
20     BL      abs
21     MOV     r5, r0    @save result
22     POP     {r1}    @restore r1
23
24 end:
25     B       end      @stop here
26
27
28
29 @-----
30 /*AbsoluteValue
31 Calculates absolute value of a value passed in via r0.
32
33 Demonstrates register responsibility:
34 Uses r0, r1, r4, r5 - must preserve existing values in r4/r5
35
36 Equivalent to:
37 int abs(int x) {
38     if (x < 0) x = -x;
39     return x;
40 }
41
42 Params:
43 r0 = number
44 Return:
45 r0 = |number|
46 */
47 abs:
48     @store any registers 4+ I want to use
49     PUSH    {r4, r5}
50
51     @do work, using
52     MOV     r1, #0
53     CMP     r0, r1    @compare parameter in r0 to 0
54     BGE     end_absIf @if r0 was >= jump ahead
55     MVN     r4, r0    @r0 was negative... copy its bitwise negation to r4
56     ADD     r5, r4, #1 @add 1 to get 2's complement negation into r5
57     MOV     r0, r5    @move negated version into r0
58
59 end_absIf:
60     @r0 has correct answer at this point
61     @restore any registers 4+ I used
62     POP     {r4, r5}
63     BX      lr        @return

```

Tip: In the simulator, make sure to use Step Into if you want the simulator to follow the branch. If you hit Step Over, the simulator will take the branch, run all the code there and not show any updates until it returns to the next instruction after the branch.

A Smarter Function Call

A lot of time is spent in the previous sample pushing and popping values from the stack. This is necessary because the main part of the program is relying on r1 being maintained and the function is making use of r4 and r5 which it is not supposed to modify. We could do the same work with less saving and restoring by using registers in the way the calling convention expects: temporary work should be done in r0-r3 and values we want to preserve are placed in r4-r9.

This version of the absolute value program uses registers in the way they should. Because it uses r0-r3 for temporaries only and r4-r9 for longer term storage, there is no need to save/restore registers to the stack:


```

1  /*
2     Reduces work storing/restoring registers by keeping "local values"
3     in r4+
4  */
5
6  .text
7  .global _start
8  _start:
9      MOV    r6, #0xAA    @set up some numbers to work with
10     MOV    r4, #30
11     MOV    r5, #-12
12
13     MOV    r0, r4        @set up parameter for abs call
14     BL     abs           @get abs
15     MOV    r4, r0        @save result from the function back to r4
16
17     @OK to temporarily use r0-r3 as long as done by next function call
18     @Do some random work...
19     MOV    r1, #3        @r1 = 3
20     MUL    r6, r6, r1    @r6 *= 3
21     @done using r1, we are OK if it gets wiped out
22
23     MOV    r0, r5        @setup param
24     BL     abs           @get abs
25     MOV    r5, r0        @save result back to r5
26
27 end:
28     B      end           @stop here
29
30 @-----
31 /*AbsoluteValue
32
33 Calculates absolute value of a value passed in via r0.
34
35 Avoids taking responsibility for registers by doing all work in r0-r3
36
37 int abs(int x) {
38     if (x < 0) x = -x;
39     return x;
40 }
41
42 Params:
43 r0 = number
44 Return:
45 r0 = |number|
46 */
47 abs:
48     @ONLY use registers r0-r3 to do my work
49     CMP    r0, #0
50     BGE    end_absIf     @check if negative
51     MVN    r1, r0        @negative - get bitwise negation
52     ADD    r0, r1, #1    @add one to get 2's complement negation
53
54 end_absIf:
55     @know that r0 now has abs(r0)
56     BX     lr            @return

```

[Try Sample](#)

Another Function

The sample below has a second demonstration of using the calling conventions - it contains an unsigned division function udiv that is used to figure out the coins to use to make change. The udiv function takes two parameters, the number to divide and the divisor, passed in r0 and r1 and returns two values, the quotient and the remainder, in r0 and r1.

Note: Since parameters are just register numbers, documentation of how the registers are being used is critical!

Like the last one, this program avoids storing and reloading registers by keeping long term values in r4+ and using r0-r3 for all the temporary work in the function.

```

1  /*
2     Demonstrates using unsigned division function to make ideal change for
3     the number of cents loaded in r0.
4
5     Answer provided as:
6         r6: quarters
7         r7: dimes
8         r8: nickles
9         r9: pennies
10  */
11
12  .text
13  .global _start
14  _start:
15
16      @Initial number of cents
17      MOV    r0, #92
18
19      @25 cents per quarter
20      MOV    r1, #25
21      BL     udiv      @ cents / 25
22      @r0 now has quarters and r1 leftover cents
23      MOV    r6, r0      @r6 = num quarters
24      MOV    r0, r1      @put leftover cents back into r0
25
26      @10 cents per dime
27      MOV    r1, #10
28      BL     udiv      @leftover cents / 10
29      @r0 now has dimes needed and r1 leftover cents
30      MOV    r7, r0      @r7 = num dimes
31      MOV    r0, r1      @put leftover cents back into r0
32
33      @5 cents per nickle
34      MOV    r1, #5
35      BL     udiv      @leftover cents / 5
36      @r0 now has nickles needed and r1 leftover cents
37      MOV    r8, r0      @r8 = num nickles
38
39      MOV    r9, r1      @r9 = pennies = leftover cents
40
41  end:
42      B      end        @stop here
43
44  @-----
45  /*Unsigned division - assumes both values are unsigned integers
46  Params:
47      r0 = number
48      r1 = divisor
49  Return:
50      r0 = quotient
51      r1 = remainder
52  */
53  udiv:
54      MOV    r2, #0      @temp_quotient
55      B      udiv_test
56  udiv_loop_start:
57      SUB    r0, r0, r1    @number -= divisor
58      ADD    r2, r2, #1    @quotient++
59  udiv_test:
60      CMP    r0, r1      @is remaining value > divisor?
61      @Can't use GE for unsigned values
62      BCS    udiv_loop_start @if higher or same continue loop
63
64      MOV    r1, r0      @remainder = leftover part of number
65      MOV    r0, r2      @quotient = temp_quotient
66      MOV    pc, lr      @return

```

[Try Sample](#)

Nested Function Calls

The basic calling convention introduced earlier does not provide for a function calling another function. When the first function is called, the lr will be set to point to the return address that function should use. If that function calls a second function using BL, the lr will be replaced with the address to return to within the first function and wipe out where the first function needs to return to!

To solve this problem, we need to store the current value of the lr to the stack as we enter the function, and restore it at the end of the function. That way it does not matter if the lr gets changed during the execution of the function.

Now the calling procedure looks like:

Calling Procedure (With LR storage)

Function Call (Done by caller)

Any data in r0-r3 that needs to be stored is pushed to the stack.

Parameters are placed in r0, r1, r2, and r3 (in order).

Use BL to branch to the function.

Function Prologue (Done by called function)

Push lr to the stack

If the function will use any registers in r4-r9, push the current values to the stack.

Function Body (Done by called function)

Do the work of the function. Access passed parameters in r0-r3. Only modify registers r0-r3 and any registers that were saved in the prologue.

Function Epilogue (Done by called function)

Place any return value(s) in r0, r1, r2, r3. (C/C++ generally only return one value, but there is no reason in assembly you can't return more than one.)

Pop any stored registers (r4-r9) from the stack to restore their old values.

Pop lr from the stack

Return to the caller with BR LR

Resume Control (Done by caller)

Any returned value(s) are in registers r0-r3.

Pop stored registers (r0-r3) that were preserved before calling.

Note: Since storing and restoring the lr is only important if a function will call another function, compilers will generally use this version of the prologue and epilogue while building a function that calls another function, and the more basic version that skips that when building a function that does not make a call to another function.

This simple program demonstrates how saving and restoring the lr allows a function to call another function and still successfully return to its caller. To see what happens without doing so, try commenting out lines 31 and 41. The simulator will realize something is amiss and give an error message when you hit the return in plusTwo. A real processor would happily “return” at the end of plusTwo back to line 42 (address 0x00000018) (where the last call to plusOne returned to) instead of back to line 10 (0x00000008). This would set up an infinite loop of line 42 returning to itself.

```

1  /*
2   Main program calls plusTwo(5) and places result in r6
3  */
4  .text
5  .global _start
6  _start:
7      @save any r0-r3 I need... (none)
8      MOV    r0, #5      @load 5 as
9      BL     plusTwo
10     MOV    r6, r0      @get my return value, store into r6
11     @restore any r0-r3... (none)
12
13     @stop program
14     end: B     end
15     @exit linux style
16     @MOV    r7, #1
17     @SWI    0
18
19
20
21  @-----
22  /* plusTwo - adds two to the value passed in by calling plusOne twice
23
24  Params:
25  r0 = x
26  Return:
27  r0 = x + 2
28  */
29  plusTwo:
30      @@@ Prologue -----
31      PUSH    {lr}      @store my return address
32
33      @@@ Body -----
34      @r0 already has the parameter to pass
35      BL     plusOne
36      @r0 has x + 1
37      BL     plusOne
38      @r0 has x + 2
39
40      @@@ Epilog -----
41      POP     {lr}      @Restore my return address and saved registers
42      BX     lr          @return (return value is in r0)
43  @-----
44
45
46  @-----
47  /* plusOne - adds one to the value passed in
48
49  Params:
50  r0 = x
51  Return:
52  r0 = x + 1
53  */
54  plusOne:
55      @@@ Prologue -----
56      PUSH    {lr}      @store my return address
57
58      @@@ Body -----
59      ADD     r0, r0, #1
60      @r0 has x + 1
61
62      @@@ Epilog -----
63      POP     {lr}      @Restore my return address and saved registers
64      BX     lr          @return (return value is in r0)
65  @-----

```

[Try Sample](#)

Stack Frames

When high level code from a language like C++ is compiled, memory required for local variables needs to be managed as part of the stack. To do this, each function reserves a chunk of memory sufficient to back up necessary registers as well as to store local variables. This chunk of memory is called a **stack frame**.

To keep track of where data is within the stack frame, we need a fixed reference point. The stack pointer can be tricky to use for this as it may change during the function if more memory is allocated. So in ARM, functions set up a **frame pointer** - a register that holds the address where the function's stack frame begins. By convention, this register is r11 - fp is another name for r11.

Note: Sometimes compilers will omit creating a frame pointer and just address relative to the stack pointer. We are going to focus on the full version with a frame pointer.

In the same way that nested function calls need to worry about backing up the link register, each function is going to want its own value in the fp. Unlike the lr, which each function is responsible for backing up on its own, the fp is assumed to not be modified by other functions. So when a function does want to place a new value in fp, it is responsible for backing up the fp of the previous function and restoring it before returning.

As we start a function, we push the current lr (this function's return address) and fp (the caller's frame pointer) values so we can restore them at the end of the function. We then set the fp to hold the address where the start of the stack frame is. Finally, we allocate space for any variables in the function by subtracting from the stack pointer enough bytes to reserve the space required.

Here is what the process looks like:

Start

Push lr, fp

Set fp

Allocate space

Use stack frame

Address	Contents
0xffffffffec	
0xfffffffff0	
0xfffffffff4	
0xfffffffff8	
0xfffffffffc	
0x00000000	... ← sp

To tear down the stack, we reverse the process. We add to the stack pointer to deallocate the space reserved for variables, then pop the lr and fp to restore the values we started with:

Current stack frame

Deallocate space

Pop lr, fp

Address	Contents
0xffffffffec	more??? ← sp
0xfffffffff0	second variable (fp - 12)
0xfffffffff4	first variable (fp - 8)
0xfffffffff8	old fp (fp - 4)
0xfffffffffc	lr ← fp
0x00000000	...

When one function calls another, each new function's stack frame is built on top of the last. As a function exits and its stack frame is removed, the stack pointer and frame pointer are used to restore the previous stack frame:

foo function foo calls bar bar returns to foo		
Address	Contents	
0xffffffffe0		
0xffffffffe4		
0xffffffffe8		
0xffffffffec		
0xfffffffff0	second variable for foo	← sp
0xfffffffff4	first variable for foo	
0xfffffffff8	old fp (for foo's caller)	
0xfffffffffc	lr for foo	← fp
0x00000000	...	

The foo function's stack frame occupies 0xfffffffff0-0xfffffffffc

Using the Stack Frame

After the stack frame is built, we use it to store and load from any time a variable needs to be read or written.

We will demonstrate what that might look like for this c++ function:

```
int bar(int num) {
    int a = num + 1;
    a *= 2;
    return a;
}
```

The stack frame for this function would need space for the parameter num and the local variable a. Since each is an int, they will each require 4 bytes of space. Thus the stack frame will end up looking like this:

Address	Contents	
0xffffffff0	a	(fp - 12) ← sp
0xffffffff4	num	(fp - 8)
0xffffffff8	old fp (for bar's caller)	(fp - 4)
0xfffffff0c	lr	← fp
0x00000000	...	

Tip: All variables are allocated in the stack frame initialization as the function starts. We will assume that they are always placed in the order in which they are declared.

Any time we want to access **num**, we will need to load/store from [fp, #-8]. Any time we want to access the variable **a**, we will use the address [fp, #-12].

The program below has the assembly version of this function.

The prologue sets up this stack frame:

- First the fp and lr are pushed and the frame pointer set to point to the top of the frame on lines 46 and 50
- Line 52 sets up the space for **num** and **a** by moving the stack pointer 8 bytes
- The variable **num** needs to be initialized with the value passed in. The parameter was passed in r0, so on line 55 it is copied to the location in memory reserved for **num**: [fp, #-8]

Then, in the body of the function, we use loads and stores to access and modify variables.

Something like a = num + 1 turns into multiple steps:

- Load the value of **num** from the stack frame into a register
- Add 1 to that register
- Then store the result back to the location in the stack frame that corresponds to **a**.

This process is demonstrated on lines 60–62.

Finally, the epilogue gets the return value ready, cleans up the stack frame, and restores the **fp** and **lr**:

- Line 70 loads the value from the location that corresponds to **a** into **r0** to be returned
- Then line 71 deallocates the space used for **num** and **a** by moving the stack pointer up 8 bytes
- Then line 73 pops the **fp** and **lr**, leaving the stack pointer, **fp** and **lr** all where they were when we entered the function

```

1  .text
2  .global _start
3  _start:
4      @call bar(4)
5      MOV    r0, #4
6      BL     bar
7
8      MOV    r4, r0  @Move returned value to r4
9
10     @stop program
11     end: B     end
12     @exit Linux style
13     @MOV    r7, #1
14     @SWI    0
15
16
17
18     /*
19     Bar function - pointless demonstration function
20
21     int bar(int num) {
22         int a = num + 1;
23         a *= 2;
24         return a;
25     }
26
27     Params:
28     r0 = num
29     Return:
30     r0 = a  (2*num + 1)
31
32     Stack Frame:
33     -----
34     FramePointer
35     Relative
36     Address          Contents
37     -----
38     -12              a
39     -8               num
40     -4              old fp
41     -0              my lr
42     */
43
44     bar:
45         @@@ Prologue -----
46         PUSH {fp, lr}  @push necessary registers to stack
47                        @if we were using r4+ would need to push those too
48                        @sp ends up 4 bytes into stack frame
49
50         ADD    fp, sp, #4  @set frame pointer to top of frame
51
52         SUB    sp, sp, #8  @move stack pointer down 8 bytes
53                        @allocates space for num and a
54
55         STR    r0, [fp, #-8] @store r0 to location reserved for num
56
57         @@@ Body -----
58
59         @@@int a = num + 1
60         LDR    r0, [fp, #-8] @load num into r0
61         ADD    r0, r0, #1
62         STR    r0, [fp, #-12] @store result into a
63
64         @@@a *= 2
65         LDR    r0, [fp, #-12] @load a into r1
66         LSL    r0, r0, #1  @ *= 2
67         STR    r0, [fp, #-12] @store back into a
68
69         @@@ Epilog -----
70         LDR    r0, [fp, #-12] @load a into r0 for return
71         ADD    sp, sp, #8  @move stack pointer up to remove 8 bytes
72                        @releases space for num and a
73         POP    {fp, lr}  @restore all registers
74                        @if we were using r4+ would need to pop those too
75         BX     lr  @return

```

[Try Sample](#)

It might strike you as a lot of wasted work. Why both storing `r0` into memory on line 62 only to load the exact same value from memory back into `r0` on line 65? Indeed, when we ask a compiler to optimize code, one of the things it seeks to do is minimize shuffling things back and forth between registers and the stack. But the unoptimized code shown above is the easiest to generate, as what we do on one line does not depend on what is happening on other lines. Each line of `c++` code does all of its own loads and stores to guarantee that the current values for variables are stored into the expected places regardless of what code comes next.

Final Calling Convention

Our final version of the calling convention, including setting up and tearing down the stack frame, looks like this:

Calling Procedure (With Stack Frame)

Function Call (Done by caller)

Any data in `r0-r3` that needs to be stored is pushed to the stack.

Parameters are placed in `r0`, `r1`, `r2`, and `r3` (in order).

Use `BL` to branch to the function.

Function Prologue (Done by called function)

Push `fp` and `lr` to the stack

Set `fp` to point to top of the stack frame (where the stored `lr` is located)

If the function will use any registers in `r4-r9`, push the current values to the stack.

Allocate space for parameters and locals by subtracting the needed number of bytes from the stack pointer.

Copy parameter values from registers `r0+` into appropriate locations in the stack frame.

Function Body (Done by called function)

Do the work of the function.

Any time we need to read/write a parameter or local variable, load and/or store it from the stack frame.

Function Epilogue (Done by called function)

Place any return value(s) in `r0`, `r1`, `r2`, `r3`. (in `C/C++` we only return one value, but there is no

reason in assembly you can't return more than one.)

Deallocate space for parameters and locals by adding the used number of bytes to the stack pointer.

Pop any stored registers (r4-r9) from the stack to restore their old values.

Pop the fp and lr to restore frame pointer of caller and the return location of this function.

Return to the caller with BX LR

Resume Control (Done by caller)

Any returned value(s) are in registers r0-r3.

Pop stored registers (r0-r3) that were preserved before calling.

Note: Since storing and restoring the lr is only important if a function will call another function, compilers may omit saving the lr for functions that call other functions. They also may omit storing the fp. They also may skip using the stack entirely and just do all the work in registers.

ARM TUTORIAL

CHEMEKETA

ARM Tutorial - Runestone Interactive Overview

This reader introduces the basics of ARM v7 assembly. The primary goal is to help students who already know how to program in a high level language learn about the form their programs take after compiling - the kinds of instructions a computer really processes information.

READ MORE CHEMEKETA >

Register Transfer Language & ARM Addressing Modes



Objectives

From this module students should learn:

- Register Transfer Language notation
- ARM Addressing Modes



Image from Articulate Content Library

Students should:

- Read the following material
- Watch the associated video lecture
- Take the Canvas quiz on this topic

Register Transfer Language

Register Transfer Language (RTL) is a notation that has been around for a very long time. Its purpose is to clearly identify which parts of computer memory are being accessed and distinguish between memory locations (aka Address) and the content of memory.

Brackets $[]$ and the arrow $\leftarrow$ or equal sign $=$ are the major symbols associated with RTL.

The brackets specify the contents of a memory location. For example $[42]$ refers to the contents of memory location 42. If there are no brackets then 42 is just a 42 not a memory location.

The arrow $\leftarrow$ or equal sign $=$ transfers the data on the right into the specified location on the left. For example $[42] \leftarrow 1000$ puts 1000 in memory location 42.

We can do calculations with RTL:

1

$[17] \leftarrow [17] + 1$ Take the contents of memory location 17 add one to it and store it back to memory location 17.

2

Both of these notations do exactly the same thing:

$[33] = 8$

$[33] \leftarrow 8$

The number 8 is put into memory location 33.

3

While

$[33] = [8]$

$[33] \leftarrow [8]$

Still does exactly the same thing but instead of the number 8 being put into memory location 33 it is the contents of memory location 8 is put into memory location 33.

4

What can go inside of the brackets can be a:

Number: [33]

Register: [R5], [MAR]

Symbolic name: [foo]

5

Brackets can also be nested.

[[55]]

The way to think about this is that memory location 55 contains a pointer or an address, not the data.

6

Calculations can also be performed with the pointers and the data.

7

For example: **[5] <- [[5] + 4]**

Which adds 4 to the contents of memory location 5 and puts the results into memory location 5.



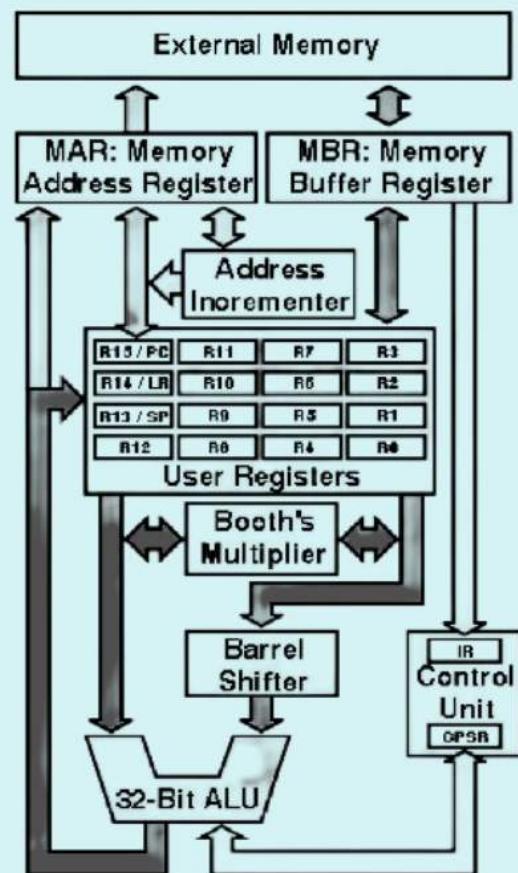
ARM Addressing Modes

The ARM instruction set architecture is a Load/Store architecture, which means that data values must be loaded into CPU registers before arithmetic or logic operations can be performed on them. The instructions that load data values from memory, or store data values in memory cannot alter the value.

In module 07 a few of the addressing modes of ARM Assembly were touched upon. This section will cover all the addressing modes.

ARM Architecture

- Load/store architecture
- A large array of uniform registers
- Fixed-length 32-bit instructions
- 3-address instructions



When it comes to addressing modes there are three things to figure out.

1. What is the effective address?
2. Where is the data also known as the operand?
3. Are there any other effects? i.e., Update of a register for example.

The following table summarizes all the addressing modes of ARM Assembly along with one very common addressing mode that ARM does not support.

Addressing Mode	Effective Address	Where is the data and what are the other effects?	Example
Literal or Immediate	None	Part of the instruction.	MOV r0, #12
Direct or Absolute	Is part of the instruction ARM assembly does not support	Main memory	LDR r0, XYZ
Indirect or Register Indirect	The referenced register contains the effective address.	Main memory	LDR r0, [r1]
Indirect with Offset	Calculated by adding the offset to the value stored in the referenced register. The offset can be a	Main memory. The index register is updated with the offset address BEFORE	LDR r0, [r1, #4] Or LDR r0, [r1, r2]

Addressing Mode	Effective Address	Where is the data and what are the other effects?	Example
	literal or the contents of another register.	the memory access.	
Auto-index pre-indexed	Calculated by adding the offset to the value stored in the referenced register.	Main memory. The index register is updated with the offset address BEFORE the memory access.	LDR r0, [r1, #8]! Or LDR r0, [r1, r2]!
Auto-index post-indexed	The referenced register contains the effective address.	Main memory. The index register is updated AFTER the memory access.	LDR r0, [r1], #8 Or LDR r0, [r1], r2
Program Counter Relative	Calculated by adding an offset to the Program Counter.	Main memory	B [PC, #32] Or B [PC, r0]

Following are more details for each of these addressing modes.

Literal or immediate —

With literal or immediate addressing there is no effective address. The data is part of the instruction. In ARM a literal is identified in the instruction preceded by a # (pound sign or hash tag). This is also called immediate addressing because no additional memory access is necessary to get the data.

Example instruction:

MOV r1, #16 @ Copies 16 into r1.

The RTL representation for this instruction:

[r1] <-- 16

Direct or absolute —

With direct or absolute addressing the effective address is contained in the instruction. The operand or data is in the main memory. ARM does not support this addressing mode. The reason this addressing mode is covered is that most every other CPU in the world supports this addressing mode.

Example instruction:

LDR r2, 25 @ Reads memory location 25 and copies that data into r2.

The RTL representation for this instruction:

[r2] <-- [25]

Indirect addressing or Register Indirect addressing - —

Register Indirect addressing is when the register contains the address of the data that is located in the main memory where the data is stored. One way to think of this is the register contains a pointer (address) to the data. The effective address is the contents of the register.

Example instruction:

LDR r2, [r3] @ Reads memory pointed to by r3 and copies that data into r2.

The RTL representation for this instruction:

[r2] <-- [[r3]]

Register indirect addressing with offset

Register indirect with offset takes the address in the register then adds the offset. This offset can be part of the instruction (looks like a literal) or another register. The effective address is calculated by taking the contents of the register then adding the offset.

Example instruction:

LDR r2, [r3, #8] @ Reads memory pointed by r3 plus 8 and copies that data into r2.

LDR r4, [r0, r5] @ Adds the contents of r0 and r5 to derive the effective address

The RTL representation for these instructions:

```
[r2] <-- [ r3 ] + 8  
[r4] <-- [ r0 ] + [ r5 ]
```

One way to think of this is r3 contains the starting address of an array and 8 is a two word offset from the start of the array.

```
Array[0]  
Array[1] @ Array + 4  
Array[2] @ Array + 8
```

Auto-indexing Pre-indexed Addressing

Auto-indexing pre-indexed takes the address in the register then adds the offset and this is the address to the data. This does the exactly same calculation as the register indirect addressing with offset to this point. The one difference is the base register is updated to this new value BEFORE the memory access is performed. In register indirect addressing with offset the base register is not modified.

This addressing mode is very useful dealing with accessing of arrays inside of loop constructs.

Example instruction:

LDR r0, [r1, #12]!

The RTL representation for this instruction:

```
[r1] <-- [ r1 ] + 12  
[r0] <-- [ [ r1 ] ]
```

Auto-indexing Post-indexed addressing —

Auto-indexing post-indexing takes the address in the register and reads that memory location and copies the contents into the destination register. After the data is put in the destination register the offset is then added to the indexing register.

This addressing mode is very useful dealing with accessing of arrays inside of loop constructs.

Example instruction:

```
LDR r0,[r1],#12
```

The RLT representation for this instruction:

```
[r0] <-- [ [ r1 ] ]  
[r1] <-- [ r1 ] + 12
```

Program Counter relative addressing —

Program counter relative addressing is not that much different than register indirect with offset. Instead of using any of the other ARM registers this addressing mode is restricted to the Program Counter. An additional restriction is the calculated references are within the user's code.

One reason this addressing has a unique classification is if you are using the GNU debugger you will see your code changed to this addressing mode.

You may have written in your assembler code:

```
B exit_loop
```

But the assembler turns what you think is direct addressing into the following:

```
B [PC, #32]
```

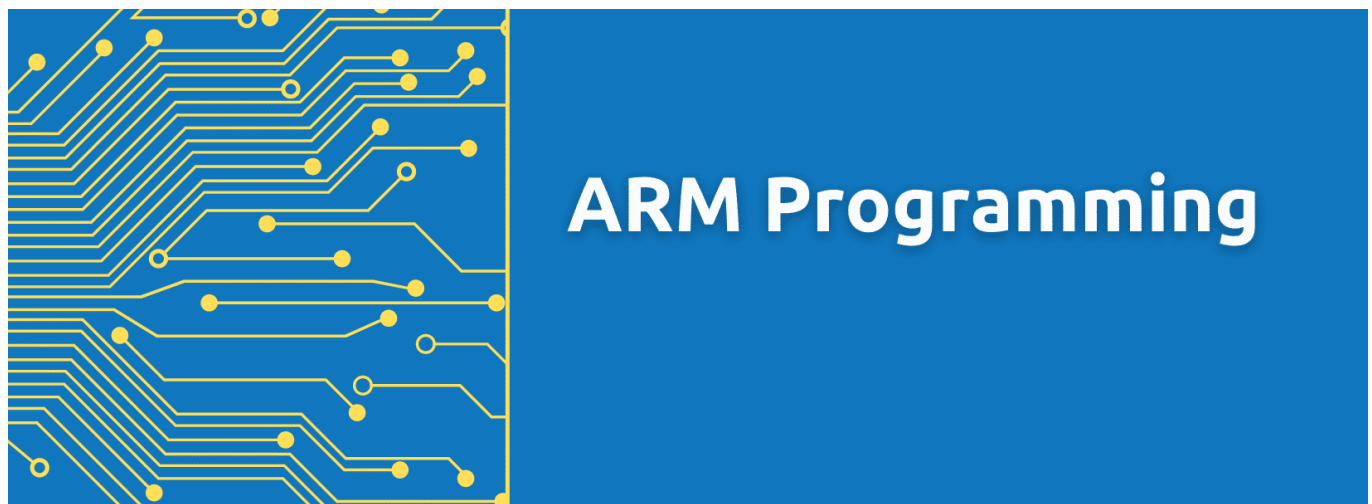
Program counter relative addressing is an important addressing mode that will ensure the code will work properly regardless of where in memory the program is executing from. This will be described in detail why this is important in the virtual memory topic.

Conclusion

The Think in Geek website provides some very good examples of how to use these addressing mode in ARM Assembly.

<http://thinkinggeek.com/arm-assembler-raspberry-pi/>

ARM Programming



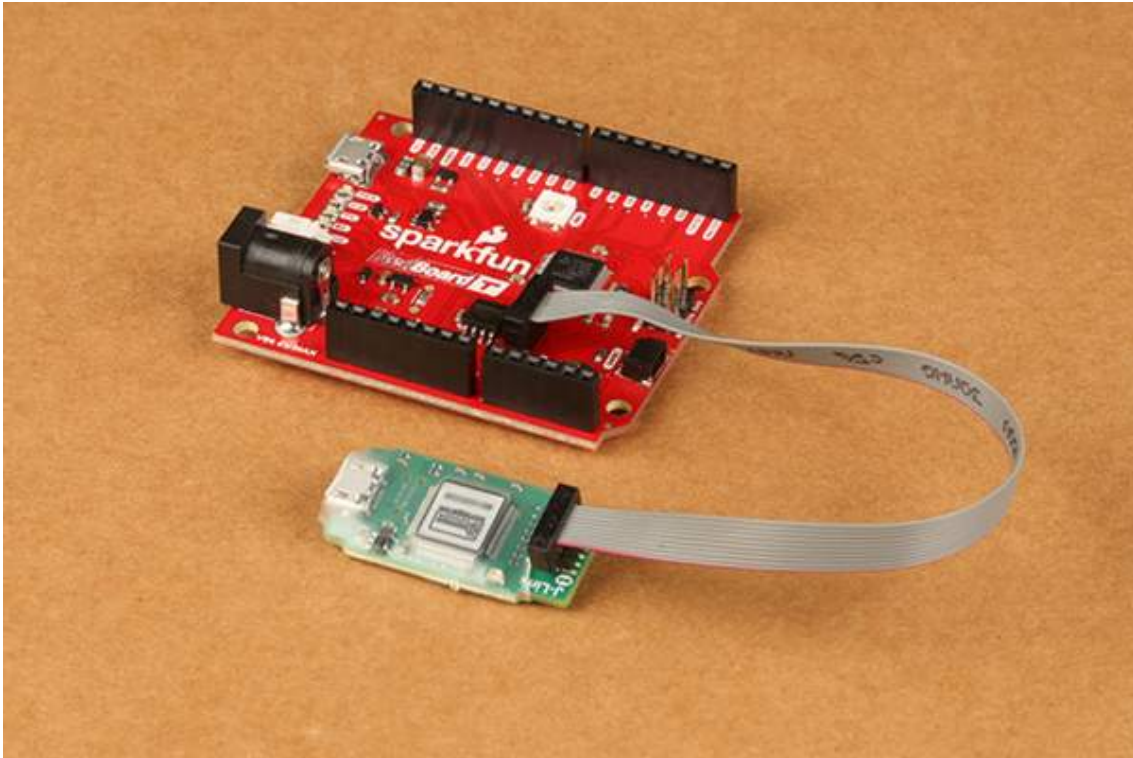
Introduction

At this point of the course, we have covered each of the ARM Assembly instructions, addressing modes and assembly directives. This module will put all those pieces together and show how to make an executable program that runs on a Raspberry Pi. The GNU debugger features are also covered so if you do have a problem getting your code to work you will be able to determine the issue.

Using assembly to do Input/Output is really complicated and to make that process easier we will use the C built-in functions (`printf` and `scanf`) in our programs.

An explanation of how the stack works at the assembly level is provided and the different methods of passing parameters to functions (methods) is also described.

When students finish this section they should be able to write all of the ARM programming assignments for this course.



CC BY-SA 4.0 Image from: <https://learn.sparkfun.com/tutorials/arm-programming/all>

Objectives

From this module, the students should learn:

- How to put the ARM assembly instructions and addressing modes to build a program
- Raspberry Pi assembly, linking, running and debugger.
- How to do user input and output using ARM Assembly
- The different ways a stack can be organized
- How to pass parameters to/from subroutines
- What happens with nested subroutine calls

- How the condition code flags get set

Students should:

- Read the following material
- Watch the associated video lecture
- Take the Canvas quiz on this topic



Image from: <https://commons.wikimedia.org/wiki/File:RaspberryPi.jpg>



Hello World!

It has been a tradition that the first program for most programming classes is the very simple program that prints "Hello World." to the default output device. Nothing comes easy in assembly especially when it comes to performing input/output.

<Put link in Canvas module to download this file>

<<helloworld.s>>

Get Inputs

Most programs require some type of input. It either comes from a file, other program or the user. This example shows how to use the C built in function scanf for performing user input. This part of the course is to teach the students on how to do ARM assembly not the details of the Input/Output processes related to C or C++.

<Put link in Canvas module to download this file>

<<student_inputC.s>>



Image created using Canva

The stack is a data structure. It is a last in first out (LIFO) or first in last out (FILO) queue. Items enter at one end and leave in the reverse order.

Stacks in modern CPUs are implemented by using a special purpose register called the stack pointer (SP). This register is used to keep track of where in memory the stack is located.

With the ARM Assembly there are PUSH and POP instructions that put or remove data to/from the stack. There are two design decisions that have to be made when implementing a stack:

1. Whether the stack grows up toward low memory as items are pushed or whether the stack grows down toward high memory as items are pushed.
2. Stack Pointer points to the top of stack (TOS) and indicates the next item on the stack OR the first free location.

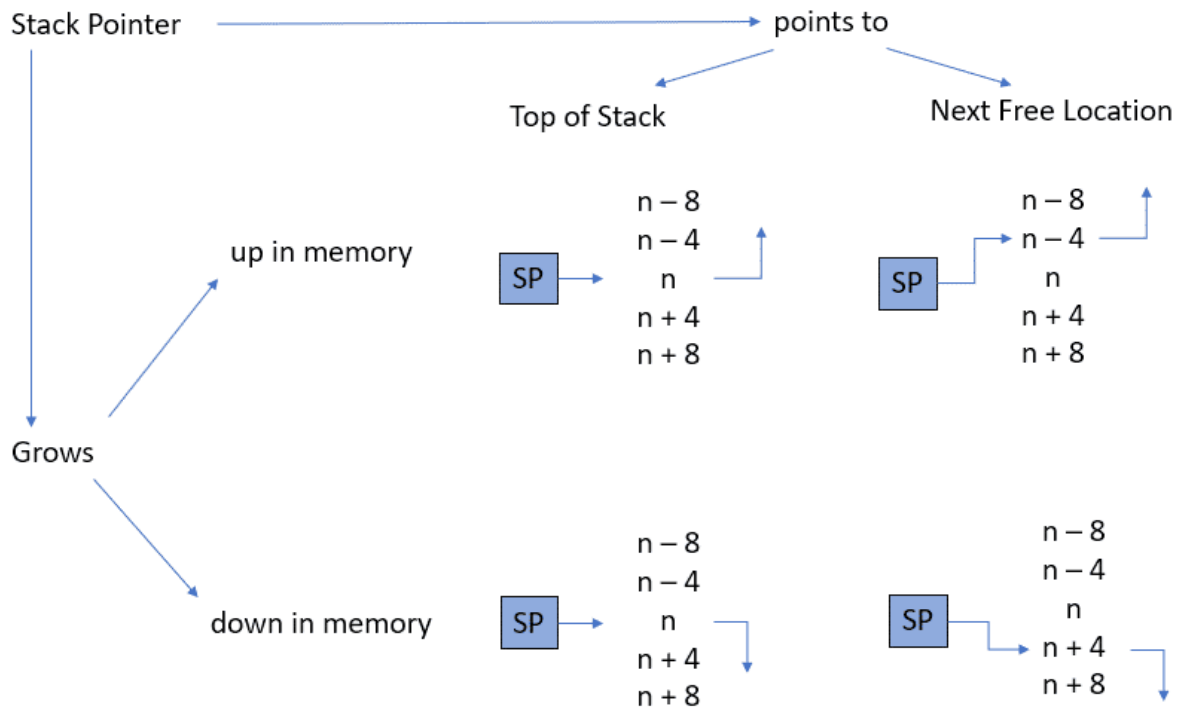


Diagram showing the four options for a Stack Pointer

Passing Parameters

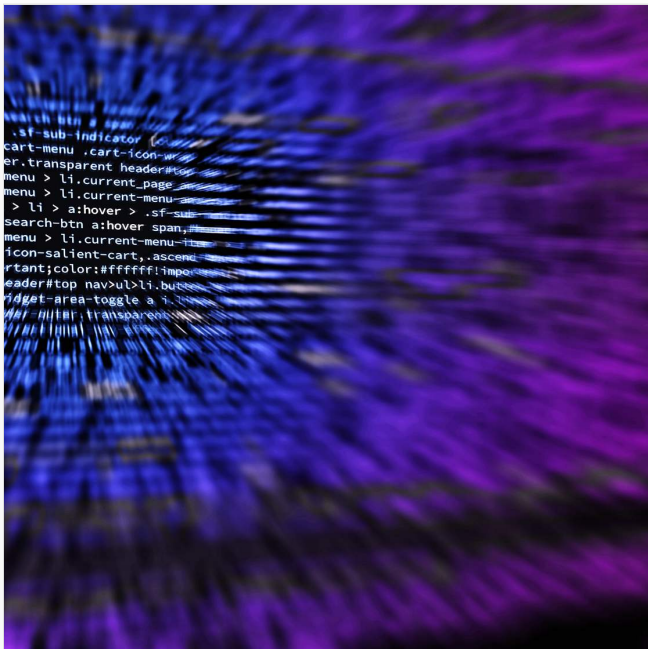
It does not matter if your language of choice calls these subroutines, functions or methods they have to be concerned with the following.

1. How are parameters passed to and from them?
2. How is the return address saved?

An assembler programmer is responsible to figure out how the items above are going to be handled.

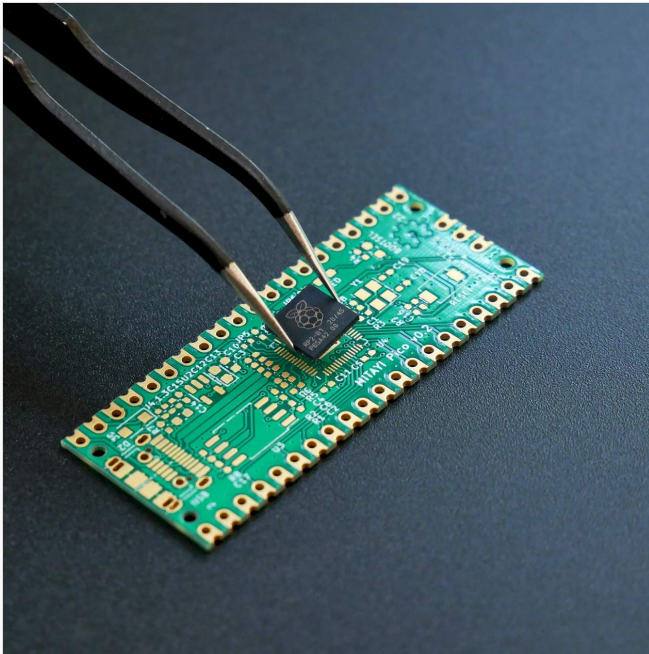
Keep in mind what is being covered here is the low level mechanism to get information to the function and return values from the function. The "methods" for passing parameters in C++ and JAVA, etc. are higher level constructs. Things like pass-by-value and pass-by-reference are defined by the programming language. No matter the High Order Language (HOL) or High Level Language (HLL) passing constructs used by the language they have to be implemented use one of these methods.

Passing of parameters can be done in one of these three ways in assembly. Flip each card to learn more:



1

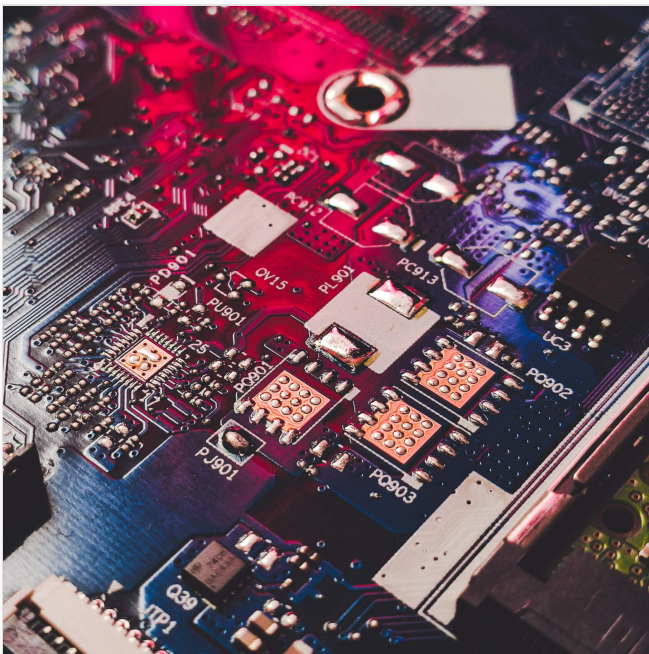
Just have global variables defined in the data section and the subroutine has direct access to them. There is NO information hiding in assembly. Everything is global.



2

The data is put in specific registers in the calling routine and the subroutine knows which registers contain what it needs to work on.

For example with the `printf` `r0` contains the address of the string and `r1` contains any data that may be printed with `%d`, `%s`, etc.



3

The data is pushed onto the stack by the calling routine and the subroutine knows what order they are pushed so they can pop them and then do the operations.

The same type of thing for data being returned from the subroutine:

1. Global variable
2. Register
3. Stack

As an assembler programmer, you need to pick a method and stick with it. With the programming language C, the convention is to use registers r0-r3 to pass values to the function and return values in those registers as well. With C once you go over 4 parameters ALL the parameters are passed via the stack. When the stack is used this slows down the execution time for the call. The stack is in memory and this process takes a LOT of time to perform. The data has to be pushed onto the stack prior to the call and then once the call is performed all that data needs to be retrieved from the stack. On the return, once again the data being returned to the calling routine has to be pushed onto the stack then it all has to be popped back off in the calling routine.

Moral of the story - - - If you want your code to run fast make sure you define all your functions with four or fewer parameters.

Subroutine Calls

ARM Assembly

The BL (Branch and Load) assembly instruction does the following two things:

[LR] <-- [PC] @

Saves the current program counter to the Link Register.

[PC] <-- [PC] + [Address field of the IR] @

Update the program counter to the subroutine location.

The assembly program examples of using the BL are to make calls to printf and scanf. Assembly programmers can define their own subroutines by just using a label for the start of the routine and when the routine is complete take the saved address in the LR and put it back into the PC. This looks like:

MOV PC, LR @Return from this subroutine.

This is a very simple approach for implementing subroutines which is typical of RISC processors. There is a BIG problem with this method for doing subroutines.

The problem happens when there are nested subroutine calls as shown in the example below. When there is a subroutine call from the main and then in that called subroutine (SUB1) there is another subroutine call to SUB2. The BL SUB2 call causes the saved PC in the LR from the main procedure to be overwritten by this BL. The call to and return from SUB2 works with no problem. Execution will return back to the correct place in SUB1 (right after the BL SUB2 instruction). The failure happens with the MOV PC, LR in SUB1. The LR still contains the return address from SUB2 back to SUB1 and the return address back to MAIN has been lost.

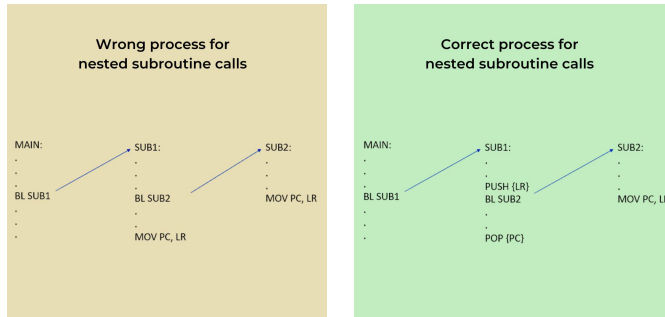


Image created using Canva

The correct way to do these nested subroutine calls is shown in the example below. Prior to making the BL SUB2 call in SUB1 the current LR (the return address back to MAIN) is saved on the stack (PUSH {LR}). SUB2 is sometimes called a leaf routine because it makes no BL calls and there is no danger of LR being overwritten in SUB2. When the return from SUB2 back to SUB1 is made the correct return address back to MAIN is on the top of the stack. The POP {PC} puts the saved LR back into the PC and execution resumes back in MAIN.

How do Complex Instruction Set Computers (CISC) handle subroutines?

They have an instruction that looks something like:

BSR Proc_A @calls subroutine Proc_A.

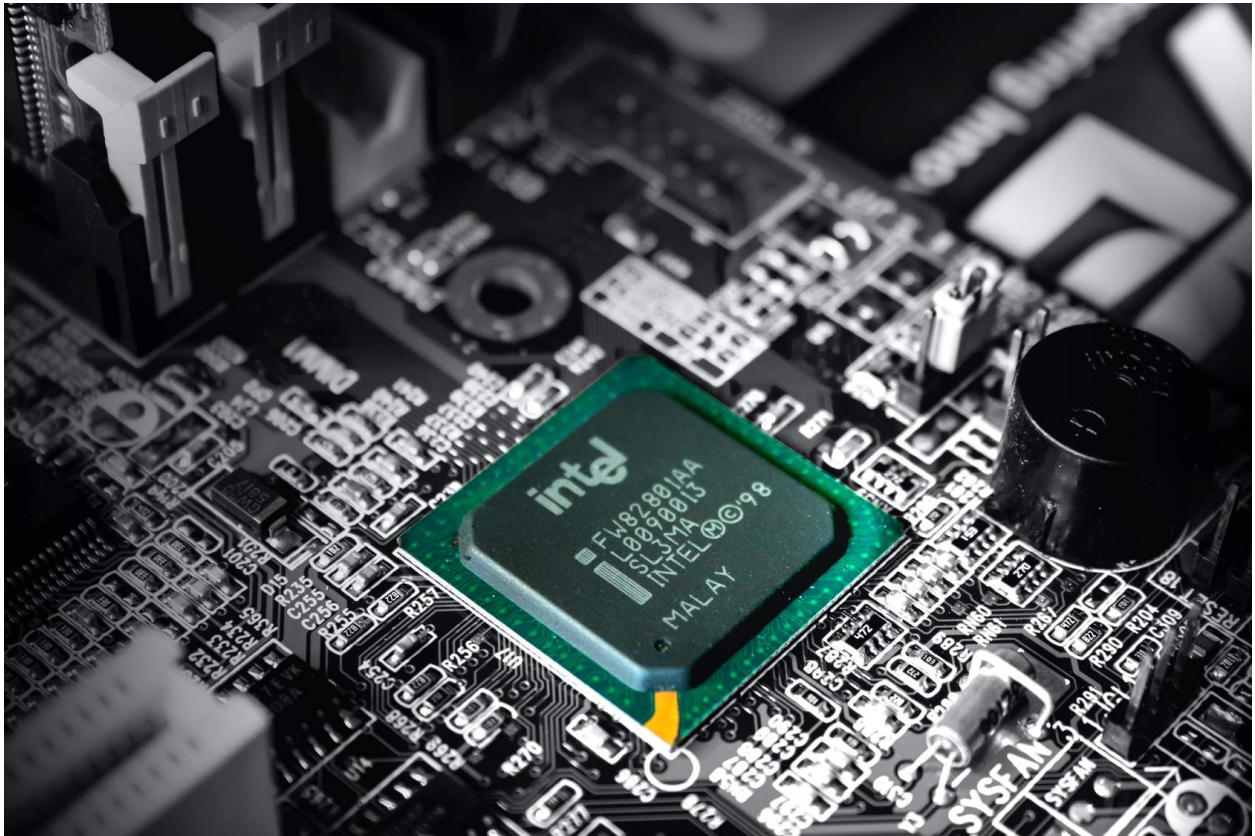


Image from Articulate Content Library

The processor saves the current PC on the stack, and loads the program counter with the address of the first instruction in the subroutine.

At the end of the subroutine a return from subroutine instruction, RTS, causes the processor to return to the point immediately following the subroutine call by popping the saved return address into the program counter.

Always putting the return address on the stack prevents the nested subroutine calls from being a problem like it is with the ARM.

Setting the Flags in the CCR

When the ALU performs an operation, it stores status or condition information in the CCR.

The processor records whether the result is:

1

Zero (Z)

2

Negative in two's complement terms (N), bit 31 is 1

3

Generated a carry (C), carry from bit 31 full adder.

4

Arithmetic overflow (V). -- What is this first? See below.

Decimal example of the problem that computers have to deal with.

$523 + 765 = 1,288$ which is not a problem if there is 4 positions for the result. It is a big problem if there is only 3 positions for the result.

Here are the binary issues using small numbers so you can see the issue.

Binary Example using 8-bits

Example 1	Example 2	Example 3	Example 4
<pre> 0011 0011 +0100 0010 ----- 0111 0101 </pre>	<pre> 1111 1111 +0000 0001 ----- 10000 0000 </pre>	<pre> 0101 1100 +0100 0001 ----- 1001 1101 </pre>	<pre> 1101 1100 +1100 0001 ----- 11001 1101 </pre>
Z=0, N=0, C=0, V=0	Z=1, N=0, C=1, V=0	Z=0, N=1, C=0, V=1	Z=0, N=1, C=1, V=0
$51 + 66 = 117$	$-1 + 1 = 0$	$92 + 65 = -99$	$-36 - 63 = -99$
Correct	Correct Zero Carry	Incorrect Negative Overflow	Correct Negative Carry

What does the hardware look like to implement these flags?

Z

32 input NOR gate

C

Carry out from bit 31 full adder

N

Bit 31 value.

V

Sign bit is the same for both numbers but the sign of the result is different. The boolean function for the overflow looks like this:

$$a(n-1)b(n-1)s(n-1)' + a(n-1)'b(n-1)s(n-1)$$

Where a and b are being added together and s is the resulting sum.

Overflows and how languages handle them:

- C - keeps on going like nothing ever happened
- C++ - has the ability to detect and throw an overflow exception. The program still has to handle it.
- Java does not automatically throw an error or overflow exception. Reference [HERE](#) for more details
- Assembly - no way! You have to check the overflow but and do all the code yourself.

For the ARM an "S" needs to be added to the end of the instruction to get the CCR updated.

For example:

ADDS SUBS

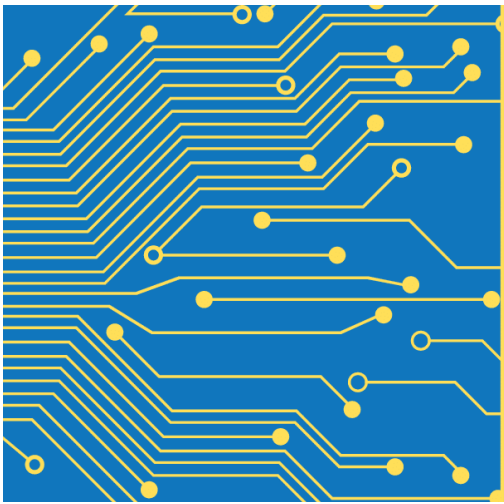
Or just use one of the four testing instructions which always sets all the flags.

- TeQ - tests for equivalent or equal. Only the Z-bit will be changed based on $[r1] - [r2]$ if Zero the Z-bit will be set. If not zero it will be cleared. All the other CCR bits will remain unchanged.
- TST - ANDs two values and sets ALL the CCR bits. This instruction is useful to test individual bits in a register.
- CMP will set ALL the bits in the CCR. It does the subtraction $[r1] - [r2]$ and sets all of the bits of the CCR.
- CMN - complements the second operand before doing the compare. $[r1] - (-[r2])$ which is the same as $[r1] + [r2]$. Sets ALL the bits of the CCR.

CISC processors, like the Intel IA32 update status flags after each operation.

RISC processors, like the ARM, require the programmer to determine when the status flags are updated.

The Pipeline



The Pipeline

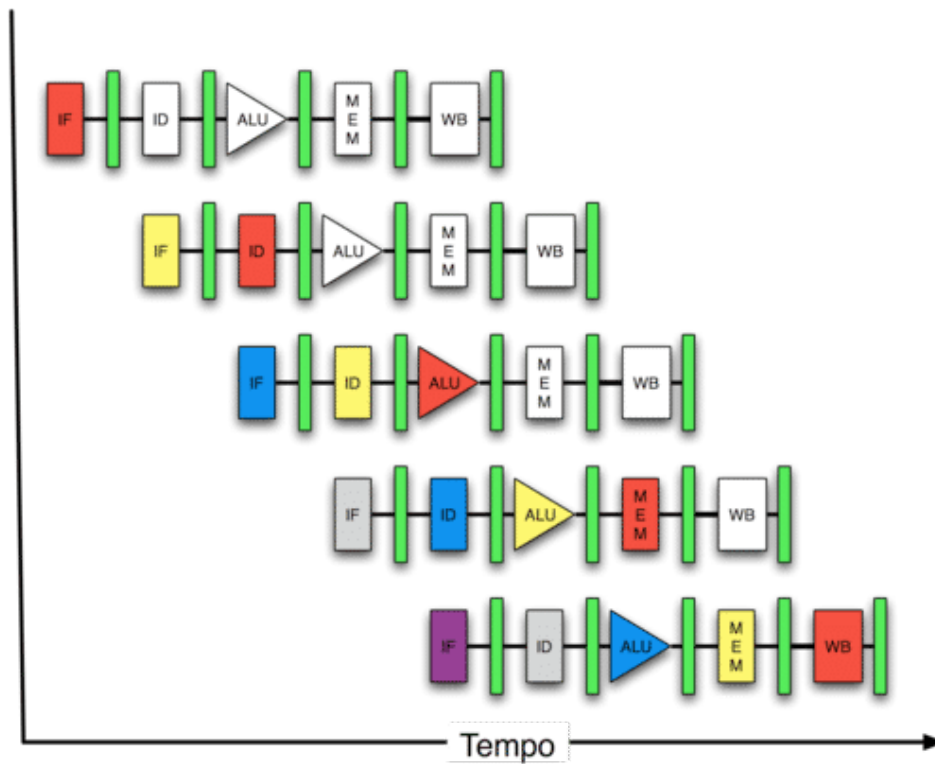


Image from: Hellisp, Public domain, via Wikimedia Commons

Objectives

From this module students should learn:

- How a pipeline improves CPU performance.
- The stages of a typical CPU pipeline
- Hazards of the pipeline
- How to identify hazards
- How pipeline performance can be improved

Students should:

- Read the following material
- Watch the associated video lecture
- Take the Canvas quiz on this topic

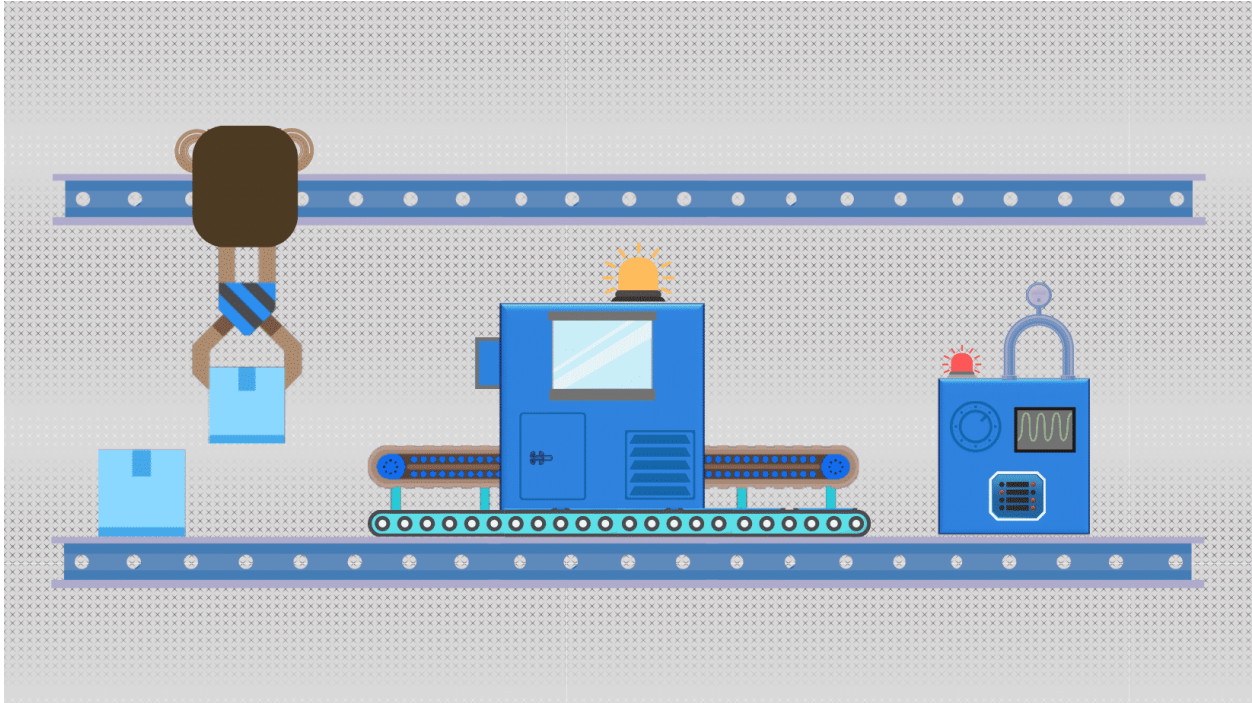


Image created using [Canva](#)

 YOUTUBE



Lecture - 24 Pipelined Processor Design Basic Idea

Lecture Series on Computer Architecture by Prof. Anshul Kumar, Department of Computer Science & Engineering, IIT Delhi. For more details on NPTEL visit <http://nptel.iitm.ac.in>

VIEW ON YOUTUBE >

Introduction to the Pipeline

To help explain how pipelines improve computer performance we will start with a simple example of a factory that uses an assembly line to build a computer. Given all the parts and tools it takes one person 100 minutes to build one computer. This build process can be subdivided into five subtasks that are labeled A to E which take exactly the same amount of time 20 - minutes. Subtask B cannot be started until subtask A has been completed. This is also the case with all the other remaining subtasks. The following table shows how this computer build process works.

0 – Mins	20 – mins	40 – Mins	60– Mins	80 – Mins	100 – Mins	100 – Mins	100 – Mins	100 – Mins
A1	B1	C1	D1	E1	Complete 1	.	.	-
.	A2	B2	C2	D2	E2	Complete 2	.	-
.	.	A3	B3	C3	D3	E3	Complete 3	-
.	.	.	A4	B4	C4	D4	E4	Complete 4

At time 0 – Minutes there is only one computer being built. Computer 1 is in stage A. At time 20 minutes there are two computers being built. Computer 1 is now in stage B and computer 2 is in stage B.

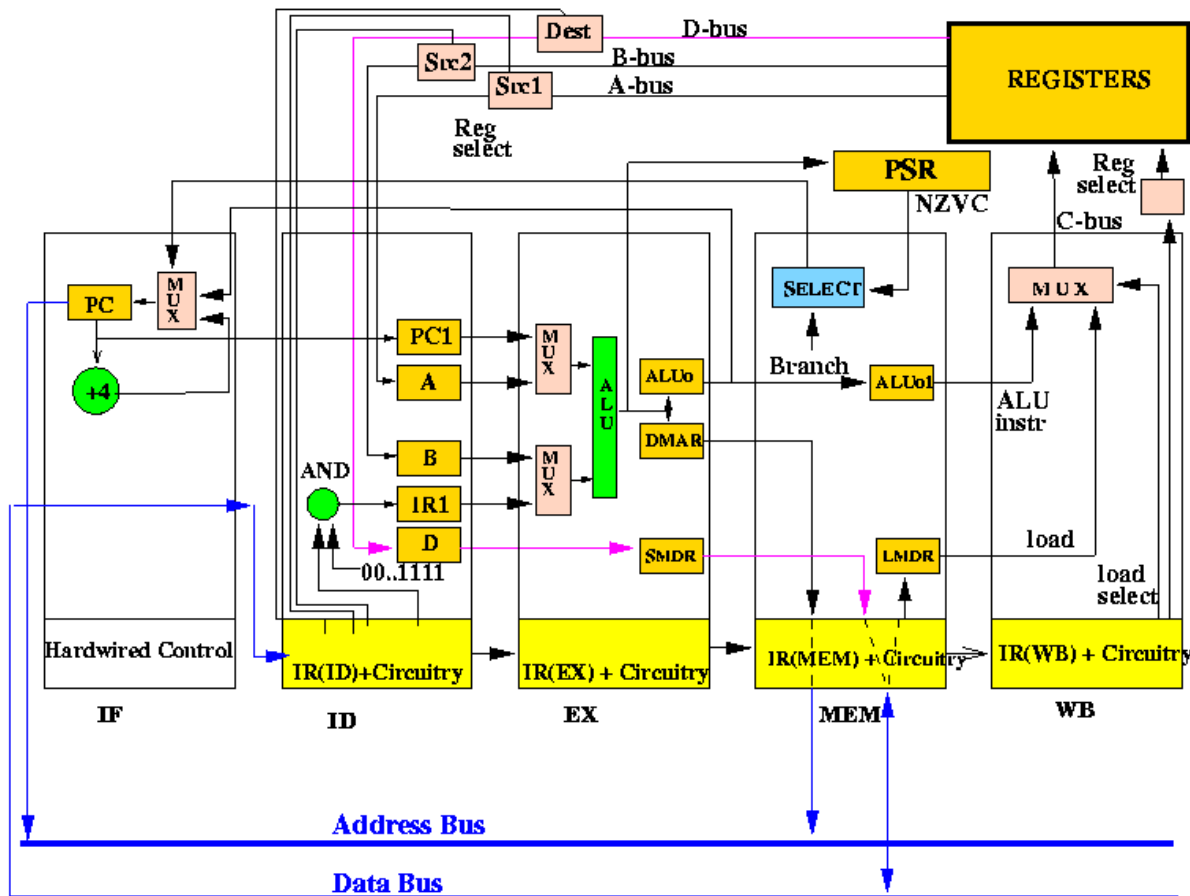
Important points to understand about this process:

- Once the pipeline is full a computer is finished every cycle (every 20 minutes).
- It still takes the same amount of time to build a single computer (100 minutes each)
- More resources are needed (people, workstations, tools) to implement this approach
- The same principles apply to CPUs as well (with some additional complexity).

The Basic Pipelined CPU

The following figure shows a basic CPU that has a five-stage pipeline. It is not important that students know the details of this specific design. You should recognize this is more complex and the CPU design covered in previous modules. This was to be expected because some

resources cannot be shared across pipeline stages. One example is this design has an additional adder (+4) in the IF stage to update the Program Counter while in the previous design the ALU was used to perform this operation. This is not the only example of additional hardware required to implement the pipeline.



The five states of this pipeline are:

1. IF - Instruction Fetch - We have been over the fetch process already and will not cover it again.
2. ID - Instruction Decode - Determines what instruction is being executed and gets all the proper operands for the next stage.
3. EX- Execute - Performs the operation specified by the instruction.
4. MEM - Memory operation - Performs any of the needed memory read or write operations and performs the branch operation.

5. WB - Write Back - Save the data to the specified register.

Each of these stages takes one clock-cycle to execute.

Not all instructions require all 5-stages. For example a CMP does not have the need for the MEM or WB stages but those instructions will still need to go through the stages so as not to complicate the pipeline any more than what is already is.



The **yellow** rectangles are **Instruction Registers** for the various pipeline stages
Because the pipelined CPU is executing **multiple instructions** at once, there are multiple Instruction Registers.



The **gold (orange-color)** rectangles are various **memory elements** (i.e., the special and general purpose **registers**).
The **pinkish** rectangles are **multiplexors** and/or **decoders** (C-bus) that are used to **select** inputs or registers (we have covered this thoroughly in the non-pipelined CPU and the same principles and concepts did not (and will not) change in the discussion of the pipelined CPU).



The **green** rectangles are **computational circuits** (adder, ALU, AND) - again, there is no difference between adder, ALU circuits in a non-pipelined and a pipelined CPU.



The **blue** rectangle is a **branch decision logic circuitry** that make branch decisions.



There is one clock signal and All registers are tied to this one clock signal:
• So **all registers** (consisting of **D-flipflops**) are updated **once per clock period**. (I did **not** draw the clock signal in the figure. You **must** assume there is a **clock signal** to **all** registers (= the **gold (orange-colored)** rectangles)



Also, you should remember that:
• Up to **1 registers** in the set of General Purpose Registers are

updated per instruction. (A decoder will select one register to be updated)



Because every register must be tied to the clock signal, due to its simplicity and space (i.e., neatness) limitation, the **clock** signal is **left out** in the figure.

As already stated it is not important the students understand the details of this design. This is still a simplified design and most modern CPUs use more than a five stage pipeline.

What do modern CPUs have for their pipelines?

- Pentium Pro - 10-stages
- Pentium 4 - 20 -stages
- Pentium (other versions) up to 31-stages
- ARM11 - 8-stages
- ARM7 - 3-stages
- ARM10 - 6-stages
- SPARC T1 - 6-stages

Floating point instructions have their own separate pipeline in most modern CPUs.

Speed-up calculations

Consider a 'k' segment pipeline with clock cycle time as 'Tp'. Let there be 'n' tasks to be completed in the pipelined processor. The first instruction is going to take 'k' cycles to come out of the pipeline but the other 'n - 1' instructions will take only '1' cycle each, i.e, a total of 'n - 1' cycles. The Execution Time (ET) for 'n' instructions in a pipelined processor:

$$\begin{aligned} ET_{\text{pipeline}} &= k + n - 1 \text{ cycles} \\ &= (k + n - 1) T_p \end{aligned}$$

In the same case, for a non-pipelined processor, execution time of 'n' instructions will be:

$$ET_{\text{non-pipeline}} = n * k * T_p$$

So, speedup (S) of the pipelined processor over non-pipelined processor, when 'n' tasks are executed on the same processor is:

$$S = \frac{\text{Performance of pipelined processor}}{\text{Performance of Non-pipelined processor}}$$

As the performance of a processor is inversely proportional to the execution time, we have,

$$S = \frac{ET_{\text{non-pipeline}}}{ET_{\text{pipeline}}}$$

$$\Rightarrow S = \frac{n * k * T_p}{(k + n - 1) * T_p}$$

$$S = \frac{n * k}{k + n - 1}$$

When the number of tasks 'n' are significantly larger than k, that is, $n \gg k$

$$S = \frac{n * k}{n}$$

$$S = k$$

where 'k' are the number of stages in the pipeline.

Also,

$$\text{Efficiency} = \frac{\text{Given speed up}}{\text{Max speed up}} = \frac{S}{S_{\text{max}}}$$

We know that, $S_{\text{max}} = k$

So,

$$\text{Efficiency} = \frac{S}{k}$$

$$\text{Throughput} = \frac{\text{Number of instructions}}{\text{Total time to complete the instructions}}$$

So,

$$\text{Throughput} = n / (k + n - 1) * T_p$$

The cycles per instruction (CPI) value of an ideal pipelined processor is 1.

The Speedup will also reach the number of stages of the pipeline as the number of instructions being executed approaches infinity.

Hazards

Ideally, we expect a CPI value of 1 and a speedup equal to the number of stages in the pipeline. But, there are a number of factors that limit this. The problems that occur in the pipeline are called **hazards**. **Hazards** that arise in the pipeline prevent the next instruction from executing during its designated clock cycle. There are three types of hazards:

- **Structural hazards:** Hardware cannot support certain combinations of instructions (two instructions in the pipeline require the same resource).
- **Data hazards:** Instruction depends on result of prior instruction still in the pipeline
- **Control hazards:** Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

Structural hazards —

Structural hazards arise because there is not enough duplication of resources. For example if the processor has only one ALU then it could not update the Program Counter at the same time that an address calculation takes place or an ADD in a later stage. Another example is when two instructions need to access memory at different stages of the pipeline. With only one memory address bus and data bus it is not possible to have two requests to memory at the same time.

How can structural hazards be resolved?

Solution 1: The CPU just waits until the resource is available. This requires the following:

- o Must detect the hazard

- o Must have mechanism to stall
- o Low cost and simple
- o Used for rare cases

Solution 2: Throw more hardware at the problem. Of course this has to happen during CPU design time. For example instead of using the ALU to update the Program Counter (PC) a specific adding (+4) circuit is added to the Fetch stage. The same type of thing can be done for address calculations in the MEM stage.

Data Hazards —

There are two types of data dependence – true data dependences and name dependences.

An instruction j is data dependent on instruction i if either of the following holds:

- Instruction i produces a result that may be used by instruction j, or
- Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i

A name dependence occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name

Two types of name dependences between an instruction i that precedes instruction j in program order:

An antidependence between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved.

An output dependence occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved.

Since this is not a true dependence, renaming can be more easily done for register operands, where it is called register renaming

Register renaming can be done either statically by a compiler or dynamically by the hardware.

Control Dependences

Control dependences determine the ordering of an instruction with respect to a branch instruction so that an instruction *i* is executed in correct program order. There are two general constraints imposed by control dependences:

- An instruction that is control dependent on its branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
- An instruction that is not control dependent on its branch cannot be moved after the branch so that its execution is controlled by the branch.

Having introduced the various types of data dependences and control dependence, let us discuss how these dependences cause problems in the pipeline. Dependences are properties of programs and whether the dependences turn out to be hazards and cause stalls in the pipeline are properties of the pipeline organization.

Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instructions:

RAW (Read After Write) —

- Corresponds to a true data dependence
- Program order must be preserved
- This hazard results from an actual need for communication
- Considering two instructions *i* and *j*, instruction *j* reads the data before *i* writes it

i: ADD R1, R2, R3

j: SUB R4, R1, R3

Add modifies R1 and then Sub should read it. This causes a RAW hazard in that if R1 is not updated in time by the ADD the SUB instruction will not be accessing the correct data in R1.

WAW (Write After Write) —

The WAW is unlikely to cause a problem with a single processor. It can happen with superscalar processors that perform out-of-order execution. This issue is discussed later in this course and is called a race condition. When the value of a variable is dependent the order of execution of the instructions and the last instruction to execute wins by having its value stored.

- Corresponds to an output dependence
- Occurs when there are multiple writes or a short integer pipeline and a longer floating-point pipeline or when an instruction proceeds when a previous instruction is stalled WAW (write after write)
- This is caused by a name dependence. There is no actual data transfer. It is the same name that causes the problem
- Considering two instructions i and j , instruction j should write after instruction i has written the data

i : SUB R1, R4, R3

j : ADD R1, R2, R3

Instruction i has to modify register $r1$ first, and then j has to modify it. Otherwise, there is a WAW hazard. There is a problem because of R1. If some other register had been used, there will not be a problem

Solution is register renaming, that is, use some other register. The hardware can do the renaming or the compiler can do the renaming

WAR (Write After Read) —

WAR is not an issue with pipelined processors. Consider the following sequence.

ADD R1,R2,R3

SUB R2,R4,R5

R2 is read early in cycle i and written to late in cycle $i+1$. There is no dependency. The read does not hold up the write operation in this case.

IF the instructions are reordered then this presents a problem. Reordering of code and out of sequence execution can happen with modern compilers and multicore processors.

There is no actual data transfer. It is the same name that causes the problem.

To solve this problem the register can be renamed, that is, use some other register. The hardware can do the renaming or the compiler can do the renaming.

Determining the potential hazards

Following is a method that can be used to determining if there are **potential hazards** with code in a pipelined processor. More analysis has to be conducted in order to determine if a **real hazard** actually exists.

Instruction	R1	R2	R3	R4	R5	R6	R7	R8	R9
I1: add r1, r2, r3	W	R	R						
I2: add r4, r2, r3		R	R	W					
I3: add r5, r2, r4		R		R	W				
I4: add r6, r3, r4			R	R		W			
I5: add r7, r2, r1	R	R					W		
I6: add r8, r8, r3			R					R W	
I7: add r9, r7, r1	R						R		W

RAW:

R1 I1 -> I5, I7

R4 I2 -> I3, I4

R7 I5 -> I7

WAR

R8: I6 -> I6

WAW

None

Control Hazard

A control hazard happens when there is a conditional branch (i.e., BNE exit) and the destination of a branch cannot be determined until the instruction gets to a later stage. What

can happen is the pipeline can be filled with instructions that are NOT in the correct execution path.

Unconditional branches do not cause a control hazard. The CPU can detect the unconditional branch in the fetch stage and will fetch the correct instruction for the proper execution path.

To illustrate the problem with control hazards we are going to use a four stage pipeline. The stages of this pipeline are:

1. Fetch (F)
2. Get Operands (GO)
3. Execute (E)
4. Save Operands (SO)

It is not until the instruction enters the Execute state do we know if the conditional branch is or is not taken.

The code for this example will have a conditional branch at instruction 6 (I6) and if the branch is taken the execution path continues at I20. If the branch is not taken there is no control hazard. The figure below shows what happens when the branch is taken.

Cycle	F	GO	E	SO
1	I5	I4	I3	I2
2	I6	I5	I4	I3
3	I7	I6	I5	I4
4	I8	I7	I6	I5
5				I6
6	I20			
7	I21	I20		
8	I22	I22	I20	
9	I23	I22	I21	I20

Starting out in cycle 1 the pipeline is already full. In cycle 2 the conditional branch at I6 enters the pipeline. It is not until the Execute stage for I6 do we find out that instructions I7 and I8 are not part of the right execution path. In cycle 5 the CPU has to flush I7 and I8 from the pipeline. I20 does not enter the pipeline until after I6 completes the SO stage because that is when the PC is updated with the correct execution path. There are holes in the pipeline in cycle 5 through 8. As you can tell holes are when there is no instruction in the stage. Holes cause a dip in performance since no instructions get executed. In this example no instruction completes execution for three cycles (6, 7 and 8).

Another problem with this approach to pipelining and conditional branches is that the wrong instructions in the pipeline can cause other changes to the CPU that should not have been done. To prevent this, once a conditional branch is detected in the fetch the pipeline is frozen and NO new instructions are fetched.

The table below shows what happens when the pipeline is frozen and the branch is taken. I7 is still fetched but it remains in the Fetch state while I6 continues execution.

Cycle	F	GO	E	SO
1	I5	I4	I3	I2
2	I6	I5	I4	I3
3	I7	I6-Frozen	I5	I4
4	I7		I6-Frozen	I5
5	I7			I6-Unfrozen
6	I20			
7	I21	I20		
8	I22	I21	I20	
9	I23	I22	I21	I20

This still produces the same dip in performance as the previous example but there are no side effects from having the wrong instructions entering the pipeline.

If the conditional branch is NOT taken the pipeline unfreezes in the E state because the current PC contains the correct execution path with this approach there is still a performance hit but not as much as when the branch is taken.

Cycle	F	GO	E	SO
1	I5	I4	I3	I2
2	I6	I5	I4	I3
3	I7	I6-Frozen	I5	I4
4	I7		I6-Unfrozen	I5
5	I8	I7		I6
6	I9	I8	I7	
7	I10	I9	I8	I7
8	I10	I9	I9	I8

There is a term called branch penalty which refers to the extra number of clock cycles it takes to deal with conditional branches. This will vary based on the CPU. There are two branch penalty classifications one if the branch prediction is correct and the other if the branch prediction is not correct. As shown in the example above if the branch prediction is NOT correct there is a 3 cycle branch penalty. That means that for 3 cycles no instruction completes execution. For a correct prediction there is a one cycle branch penalty.

In general, the more stages in the pipeline the larger the branch penalty will be regardless if the branch prediction is correct or not.

The binary tree shown below summarizes the complexities of dealing with conditional branches.

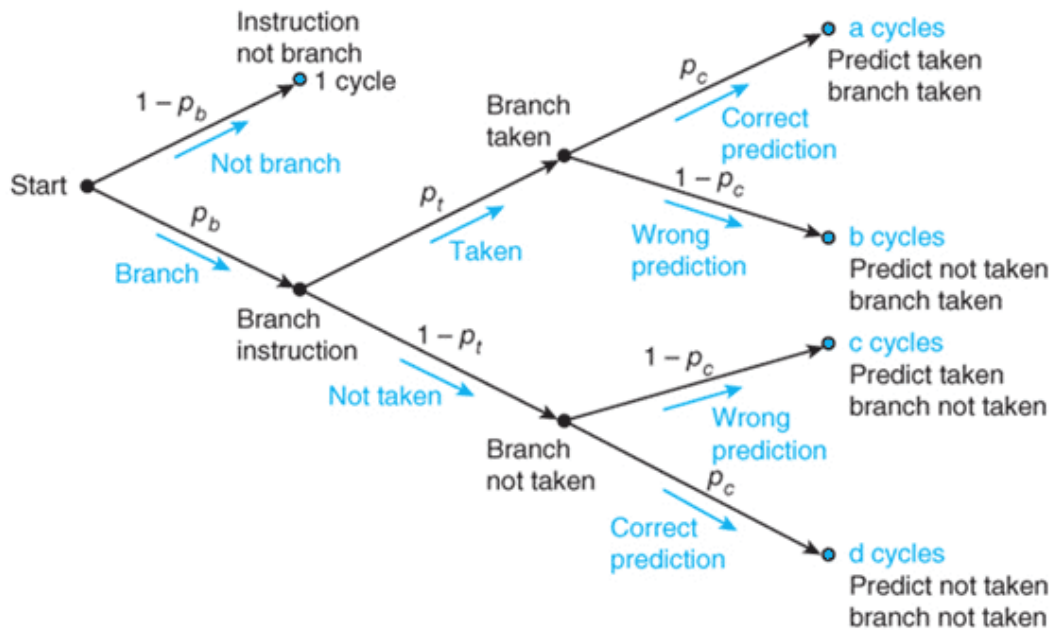
All the p 's in this figure represent a probability. The subscripts identify the different probabilities at each branch. They are:

b - probability that the current instruction is a branch

t - probability that the branch is taken

c - probability that the prediction to take or not take the branch was correct.

Given a tree like this it is possible to calculate the expected number of cycles it will take to execute one specific instruction in a given executable. The probabilities given in this tree depends on the given code and the CPU being used.



Fixing the holes

The problem with holes in the pipeline can never be prevented. There are some techniques that can be used to try and mitigate the issue. The two techniques that can be used are **delayed branching** and **branch prediction**.

Delayed Branching is really a reordering of the instructions so that an independent instruction is moved after the branch so there is not a hole in the pipeline.

For example:

How the programmer wrote it	How the compiler reordered it
$R = P + Q$ $C = B - A$ IF FRED GOTO NEXT	$R = P + Q$ IF FRED GOTO NEXT $C = B - A$

This is only an effective technique if the compiler can find an isolated (independent) instruction to place after the branch. Not just any random instruction can be placed after the branch. If an instruction cannot be found then a NOP (no operation) has to be inserted. Some references states 60% of the time an instruction can be found to fill the hole.

Branch prediction is when guesses are made if a branch will or will not be taken. There are two types of branch prediction:

1. Static
2. Dynamic

Static branch prediction either assumes that the branch is either always not taken or is taken. This turns into a basic coin flip (50-50) if the prediction is correct or wrong. Some high order languages and assemblers will allow the programmer to give a hint in their code if the branch will or will not be taken. ARM assembly does support this syntax. This happens prior to running the code and never changes during the code's runtime. That is why this is referred to as **static branch prediction**.

Dynamic branch prediction is when there is hardware used to try and guess if a branch will or will not be taken. This hardware contains memory elements that keep track of the branch, the current guess (taken or not taken) and even the next instruction of the target location if the branch is taken.

To keep track of the current guess a state machine is used. In this example two bits are needed to track of the following:

1. Strongly taken
2. Weakly taken
3. Weakly not taken
4. Strongly not taken

The diagram below shows the transitions between the states.

Based on this how well did all this extra hardware do for us?

There were 19 correct and 7 wrong guesses. This looks good. Right? $19/(19+7) = 0.7308$ or 73.1% which is much better than a coin flip (50%).

What is the true story related to execution? Let's make an assumption that for this CPU a correct guess takes two cycles or one extra cycle to execute. An incorrect guess takes four (4) cycles or a penalty of three (3) cycles.

Redoing these calculations based on the actual number of cycles vs the number of cycles if every prediction was perfect.

Actuals: $19 * 2 + 7 * 4 = 66$ cycles to execute these instructions.

Perfect prediction would be $26 * 2 = 52$ cycles so the actual prediction is: $(66-52)/52 = 0.2692$ or 27% longer than it could have been if all the guesses were correct.

That does not seem bad but we need to compare that to a coin flip time to really get a good idea

$13 * 2 + 13 * 4 = 78$ cycles to execute or $(78-52)/52 = 0.5$ or 50% longer.

This turns out to be much better than a random guess.

How do I improve my code performance?

Knowing how the compiler and CPU handle branches how should you structure your code to take advantage of this or at least mitigate issues with branching?

First thing is to avoid conditional branching as much as possible. Of course this is not practical in all cases but care should be taken to ensure any unnecessary conditional branching does not happen.

When using conditional branching make sure the most commonly occurring case takes place in the THEN part of the IF.

If the compiler or assembler supports the syntax use the HINT on the conditional branch.

Cache



Objectives

In this module students should learn:

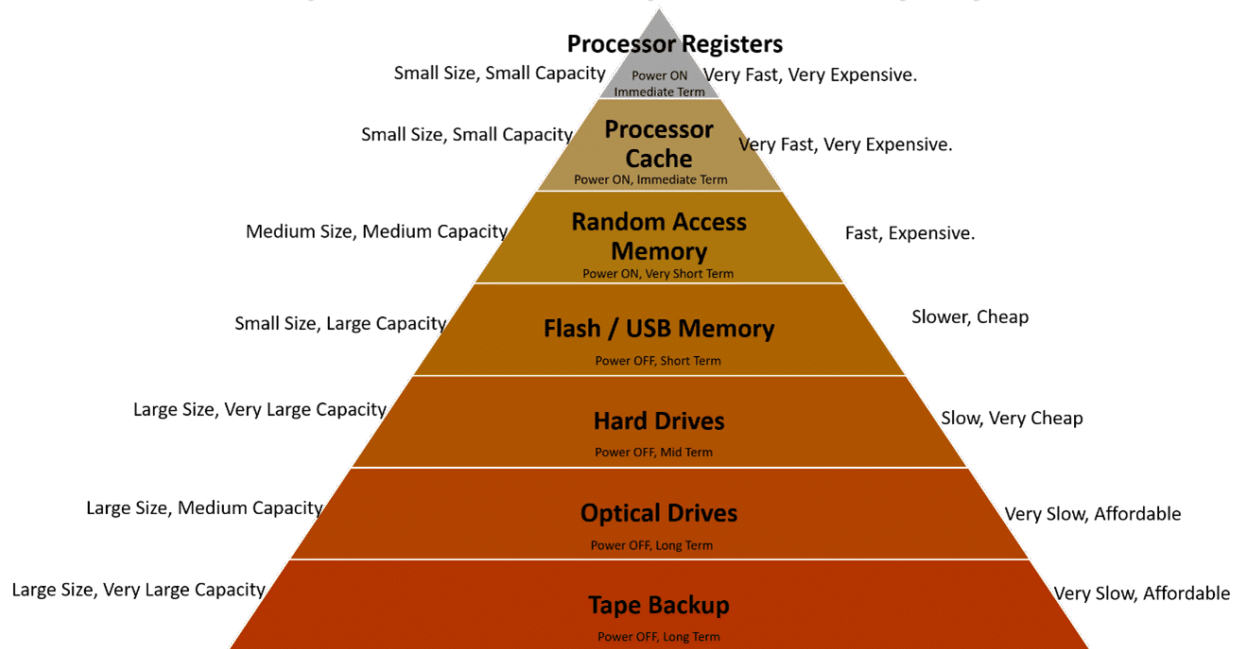
- Memory Hierarchy
- The make-up and organization of Cache memory

Memory Hierarchies and Terms

Before getting into the specifics of Cache memory an overall discussion of memory hierarchy is covered first. This is done to make sure it is clear the relation of the CPU to memory and the different parts of memory to each other.

The following figure is of a modern computer memory hierarchy. A pyramid is used to give the relation of size and speed of the memory components to each other.

Computer Memory Hierarchy Pyramid



The use of registers is the first instance you will see of measures taken to counter the fact that loading data from memory is slow. Access to data in registers, which are built into the processor, is almost instantaneous, unlike main memory, where hundreds of clock cycles can pass between requesting a data item, and it being available to the processor.

Random Access Memory (RAM) uses Dynamic RAM (DRAM) and will be described in the next Module.

Latency and **Bandwidth** are a couple of terms that are important to cache and memory access.

Latency is the delay between the processor issuing a request for a memory item, and the item actually arriving. We can distinguish between various latencies, such as the transfer from memory to cache, cache to register, or summarize them all into the latency between memory and processor. Latency is measured in (nano) seconds, or clock cycles.

Bandwidth is the rate at which data arrives at its destination, after the initial latency is overcome. Bandwidth is measured in bytes (kilobytes, megabytes, gigabytes) per second or per clock cycle. The bandwidth between two memory levels is usually the product of the cycle

speed of the channel (the bus speed) and width of the bus: the number of bits that can be sent simultaneously in every cycle of the bus clock.

Cache Introduction

As shown in the figure above the Cache is memory that is between the CPU registers and the computer's main memory (RAM). Cache has much more capacity than the CPU registers but also much less than RAM. For example in most of the ARM designs there are around 32 registers, cache is around a few Mega Bytes and main memory is in a few Giga Bytes.

Even though cache is so much smaller than main memory there is a dramatic increase in overall computer performance. One of the reasons for this is the speed of cache is faster than main memory. The other major reason is what is called the locality of reference, also known as the principle of locality, which is the tendency of a processor to access the same set of memory locations repetitively over a short period of time.

There are two basic types of locality of reference – temporal and spatial locality.

Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration. An example of this is would be the code has a for loop to do something. After several iterations the loop is completed and the code continues execution somewhere else in the code.

Spatial locality refers to the use of data elements within relatively close storage locations. Sequential locality, a special case of spatial locality, occurs when data elements are arranged and accessed linearly, such as, traversing the elements in a one-dimensional array.

Studies have shown that upwards of 98% of the memory references are found in cache.

The following came from an old (1993) white paper from Intel to explain this new technology that was part of the Pentium processor. This paper has no copyright. There are some edits from Kevin Preston for formatting, defining acronyms, etc.

Intel White Paper on Cache

The purpose of this section is twofold. The first part gives an overview of cache, while the second part explains how the Pentium Processor implements cache.

A simplified model of a cache system will be examined first. The simplified model is expanded to explain: how a cache works, what kind of cycles a cache uses, common architectures, major components, and common organization schemes.

The implementation may differ in actual designs, however the concepts remain the same. This eliminates unnecessary detail and background in hardware design or system dependency. The second part of this paper gives more detail and specifics of how the internal caches work on the Pentium Processor.

An overview of Cache —

Cache is small high speed memory usually Static RAM (SRAM) that contains the most recently accessed pieces of main memory.

Why is this high speed memory necessary or beneficial? In today's systems, the time it takes to bring an instruction (or piece of data) into the processor is very long when compared to the time to execute the instruction. For example, a typical access time for Dynamic RAM (DRAM) is 60ns. A 100 MHz processor can execute most instructions in 1 CLK or 10 ns. Therefore a bottle neck forms at the input to the processor. Cache memory helps by decreasing the time it takes to move information to and from the processor. A typical access time for SRAM is 15 ns. Therefore cache memory allows small portions of main memory to be accessed 3 to 4 times faster than DRAM (main memory).

How can such a small piece of high speed memory improve system performance? The theory that explains this performance is called "Locality of Reference." The concept is that at any given time the processor will be accessing memory in a small or localized region of memory. The cache loads this region allowing the processor to access the memory region faster. How well does this work? In a typical application, the internal 16K-byte cache of a Pentium® processor contains over 90% of the addresses requested by the processor. This means that over 90% of the memory accesses occurs out of the high speed cache. Cache performance is directly related to the applications that are run in the system. This number is based on running standard desk top applications. It is possible to have lower performance depending on how the software is designed. Performance will be discussed later in the module.

So now the question, why not replace main memory DRAM with SRAM? The main reason is cost. SRAM is several times more expensive than DRAM. Also, SRAM consumes more power and is less dense than

DRAM. Now that the reason for cache has been established, let look at a simplified model of a cache system.

Basic Model of a Cache —

Figure 2-1 shows a simplified diagram of a system with cache. In this system, every time the CPU performs a read or write, the cache may intercept the bus transaction, allowing the cache to decrease the response time of the system.

Before discussing this cache model, lets define some of the common terms used when talking about cache.

Cache Hits

When the cache contains the information requested, the transaction is said to be a cache hit.

Cache Miss

When the cache does not contain the information requested, the transaction is said to be a cache miss.

Cache Consistency

Since cache is a photo or copy of a small piece main memory, it is important that the cache always reflects what is in main memory. Some common terms used to describe the process of maintaining cache consistency are:

Snoop

When a cache is watching the address lines for transaction, this is called a snoop. This function allows the cache to see if any transactions are accessing memory it contains within itself.

Snarf

When a cache takes the information from the data lines, the cache is said to have snarfed the data. This function allows the cache to be updated and maintain consistency. Snoop and snarf are the mechanisms the cache uses to maintain consistency.

Two other terms are commonly used to describe the inconsistencies in the cache data, these terms are:

Dirty Data

When data is modified within cache but not modified in main memory, the data in the cache is called "dirty data."

Stale Data

When data is modified within main memory but not modified in cache, the data in the cache is called stale data.

Now that we have some names for cache functions let's see how caches are designed and how this effects their function.

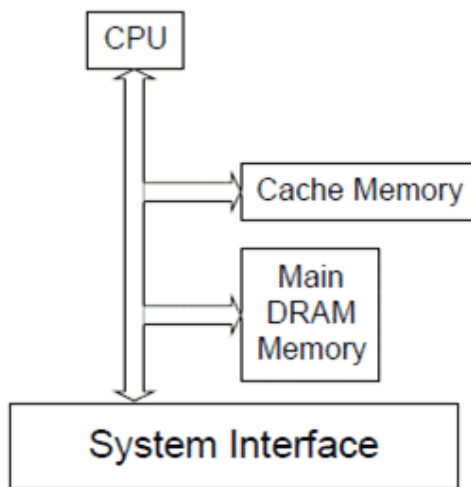


Figure 2-1 Basic Cache Model

Cache Architecture: Look Aside

Caches have two characteristics, a read architecture and a write policy. The read architecture may be either “Look Aside” or “Look Through.” The write policy may be either “Write-Back” or “Write-Through.” Both types of read architectures may have either type of write policy, depending on the design. Write policies will be described in more detail in the next section. Let's examine the read architecture now.

Look Aside

Figure 2-2 shows a simple diagram of the “look aside” cache architecture. In this diagram, main memory is located opposite the system interface. The discerning feature of this cache unit is that it sits in parallel with main memory. It is important to notice that both the main memory and the cache see a bus cycle at the same time. Hence the name “look aside.”

Look Aside Cache Example

When the processor starts a read cycle, the cache checks to see if that address is a cache hit.

HIT:

If the cache contains the memory location, then the cache will respond to the read cycle and terminate the bus cycle.

MISS:

If the cache does not contain the memory location, then main memory will respond to the processor and terminate the bus cycle. The cache will snarf the data, so next time the processor requests this data it will be a cache hit.

Look aside caches are less complex, which makes them less expensive. This architecture also provides better response to a cache miss since both the DRAM and the cache see the bus cycle at the same time. The drawback is the processor cannot access cache while another bus master is accessing main memory.

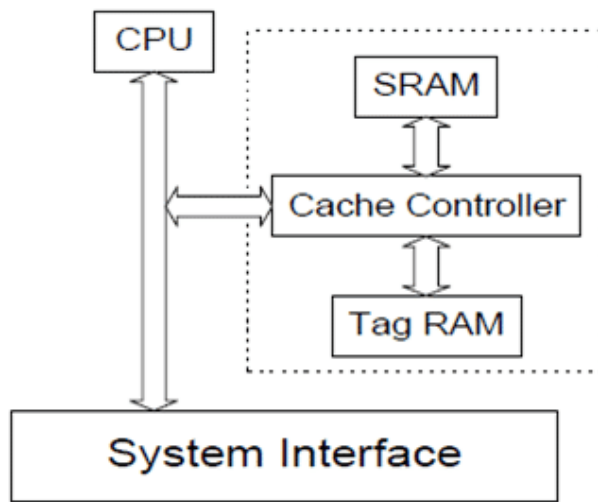


Figure 2-2 Look Aside Cache

Read Architecture: Look Through

Figure 2-3 shows a simple diagram of cache architecture. Again, main memory is located opposite the system interface. The discerning feature of this cache unit is that it sits between the processor and main memory. It is important to notice that cache sees the processors bus cycle before allowing it to pass on to the system bus.

Look Through Read Cycle Example

When the processor starts a memory access, the cache checks to see if that address is a cache hit.

HIT:

The cache responds to the processor's request without starting an access to main memory.

MISS:

The cache passes the bus cycle onto the system bus. Main memory then responds to the processors request. Cache snarfs the data so that next time the processor requests this data, it will be a cache hit.

This architecture allows the processor to run out of cache while another bus master is accessing main memory, since the processor is isolated from the rest of the system. However, this cache architecture is more complex because it must be able to control accesses to the rest of the system. The increase in complexity increases the cost. Another down side is that memory accesses on cache misses are slower because main memory is not accessed until after the cache is checked. This is not an issue if the cache has a high hit rate and there are other bus masters.

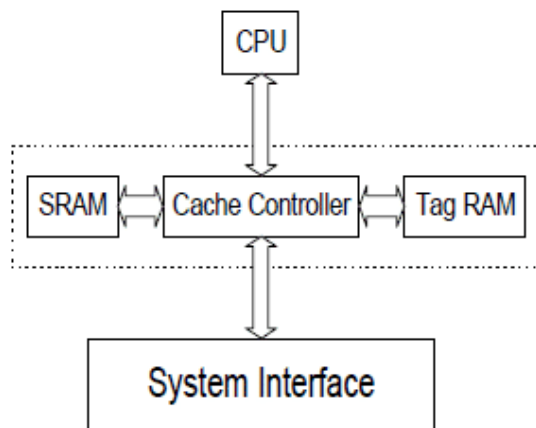


Figure 2-3 Look Through Cache

Write Policy

A write policy determines how the cache deals with a write cycle. The two common write policies are Write-Back and Write-Through. In Write-Back policy, the cache acts like a buffer. That is, when the processor starts a write cycle the cache receives the data and terminates the cycle. The cache then writes the data back to main memory when the system bus is available. This method provides the greatest performance by allowing the processor to continue its tasks while main memory is updated at a later time. However, controlling writes to main memory increase the cache's complexity and cost.

The second method is the Write-Through policy. As the name implies, the processor writes through the cache to main memory. The cache may update its contents, however the write cycle does not end until the data is stored into main memory. This method is less complex and therefore less expensive to implement. The performance with a Write-Through policy is lower since the processor must wait for main memory to accept the data.

Cache Components

The cache sub-system can be divided into three functional blocks: SRAM, Tag RAM, and the Cache Controller. In actual designs, these blocks may be implemented by multiple chips or all may be combined into a single chip.

Static Random Access Memory (SRAM) is the memory block which holds the data. The size of the SRAM determines the size of the cache.

Tag RAM (TRAM) is a small piece of SRAM that stores the addresses of the data that is stored in the SRAM.

Cache Controller - The cache controller is the brains behind the cache. Its responsibilities include: performing the snoops and snarfs, updating the SRAM and TRAM and implementing the write policy. The cache controller is also responsible for determining if memory request is cacheable and if a request is a cache hit or miss. It is not desirable to have all memory cacheable. What regions of main memory determined to be non-cacheable are dependent on the design. For example, in a PC platform the video region of main memory is not cacheable.

Cache Organization

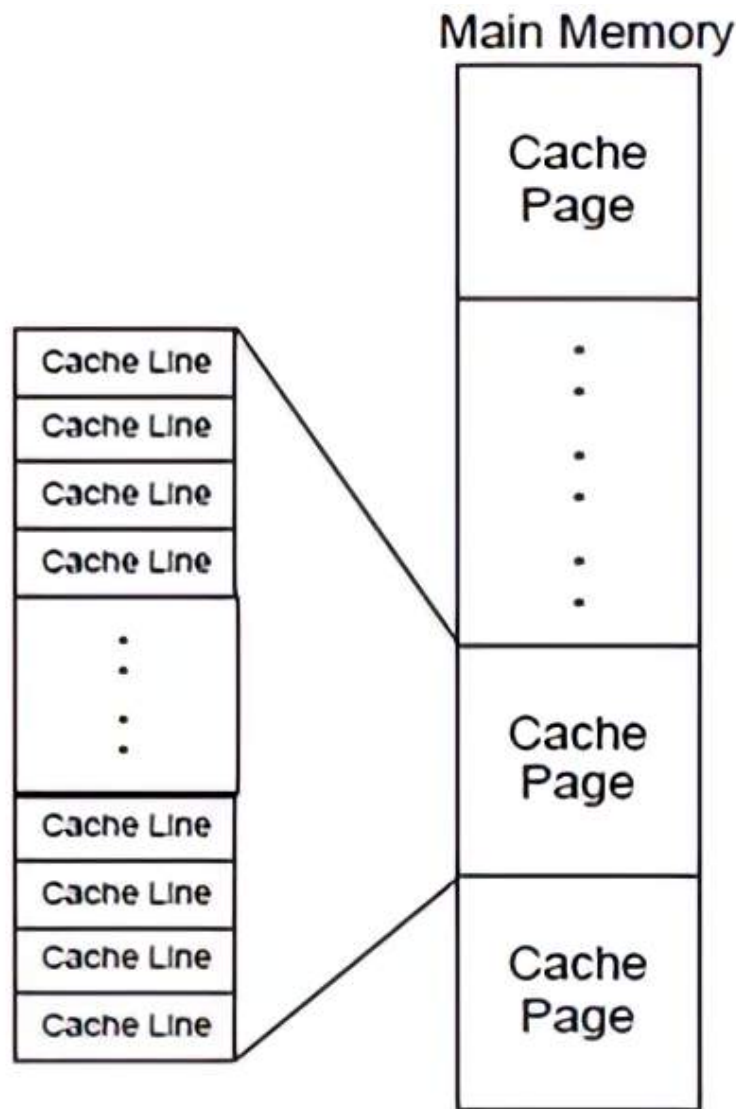


Figure 2-4 Cache Page

In order to fully understand how caches can be organized, two terms need to be defined. These terms are cache page and cache line. Let's start by defining a cache page. Main memory is divided into equal pieces called cache pages. A cache page is not associated with a memory page in page mode. The word page has several different meaning when referring to a PC architecture. The size of a page is dependent on the size of the cache and how the cache is organized. A cache page is broken into smaller pieces, each called a cache line. The size of a cache line is determined by both the processor and the cache design. Figure 2-4 shows how main memory can be broken into cache pages and how each cache page is divided into cache

lines. We will discuss cache organizations and how to determine the size of a cache page in the following sections.

Fully-Associative

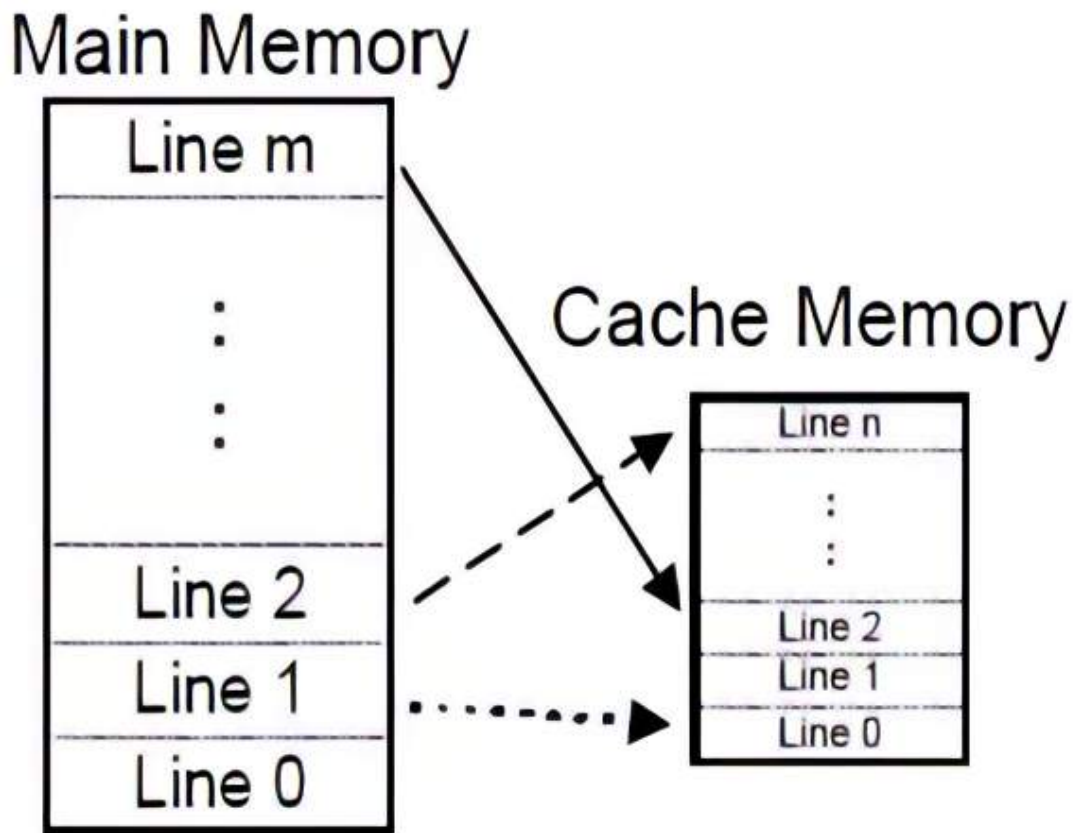


Figure 2-5 Fully-Associative Cache

The first cache organization to be discussed is **Fully-Associative cache**. Figure 2-5 shows a diagram of a Fully Associative cache. This organizational scheme allows any line in main memory to be stored at any location in the cache. Fully-Associative cache does not use cache pages, only lines. Main memory and cache memory are both divided into lines of equal size. For example Figure 2-5 shows that Line 1 of main memory is stored in Line 0 of cache. However this is not the only possibility, Line 1 could have been stored anywhere within the cache. Any cache line may store any memory line, hence the name, Fully Associative.

A **Fully Associative scheme** provides the best performance because any memory location can be stored at any cache location. The disadvantage is the complexity of implementing this scheme. The complexity comes from having to determine if the requested data is present in cache. In order to meet the timing requirements, the current address must be compared with all the addresses present in the TRAM. This requires a very large number of comparators that increase the complexity and cost of implementing large caches. Therefore, this type of cache is usually only used for small caches, typically less than 4K.

Direct Map

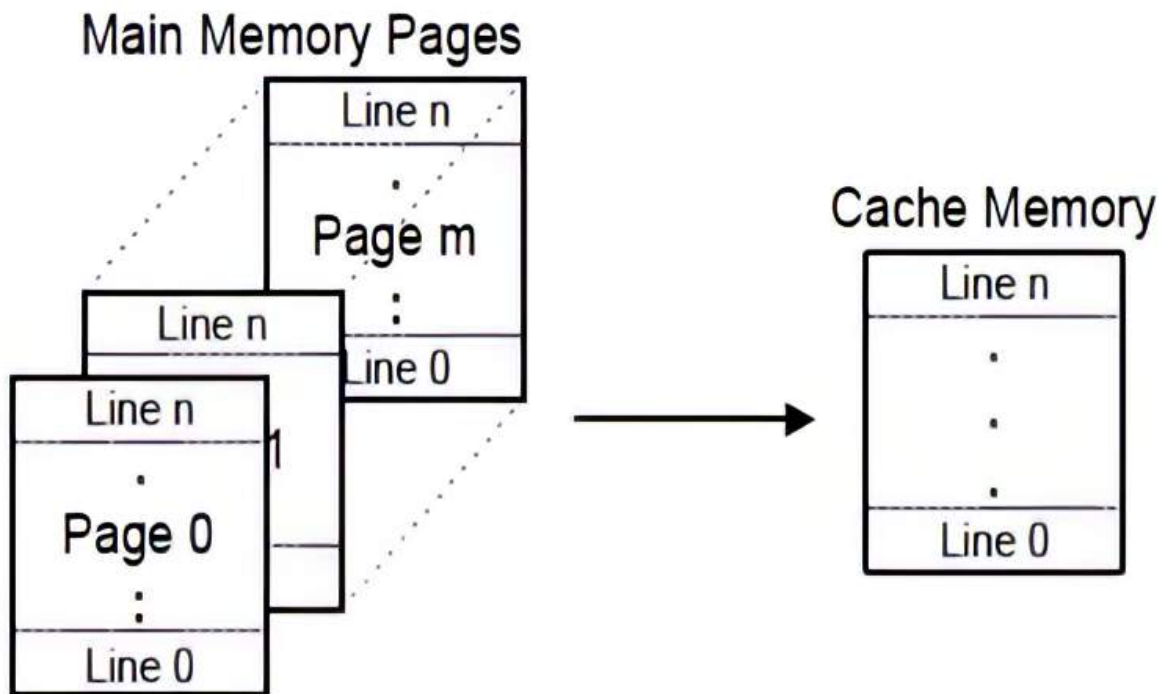


Figure 2-6 Direct Mapped

Direct Mapped cache is also referred to as 1-Way set associative cache. Figure 2-6 shows a diagram of a direct map scheme. In this scheme, main memory is divided into cache pages. The size of each page is equal to the size of the cache. Unlike the fully associative cache, the direct map cache may only store a specific line of memory within the same line of cache. For example, Line 0 of any page in memory must be stored in Line 0 of cache memory. Therefore if Line 0 of Page 0 is stored within the cache and Line 0 of page 1 is requested, then Line 0 of

Page 0 will be replaced with Line 0 of Page 1. This scheme directly maps a memory line into an equivalent cache line, hence the name Direct Mapped cache.

A Direct Mapped cache scheme is the least complex of all three caching schemes. Direct Mapped cache only requires that the current requested address be compared with only one cache address. Since this implementation is less complex, it is far less expensive than the other caching schemes. The disadvantage is that Direct Mapped cache is far less flexible making the performance much lower, especially when jumping between cache pages.

Set Associative

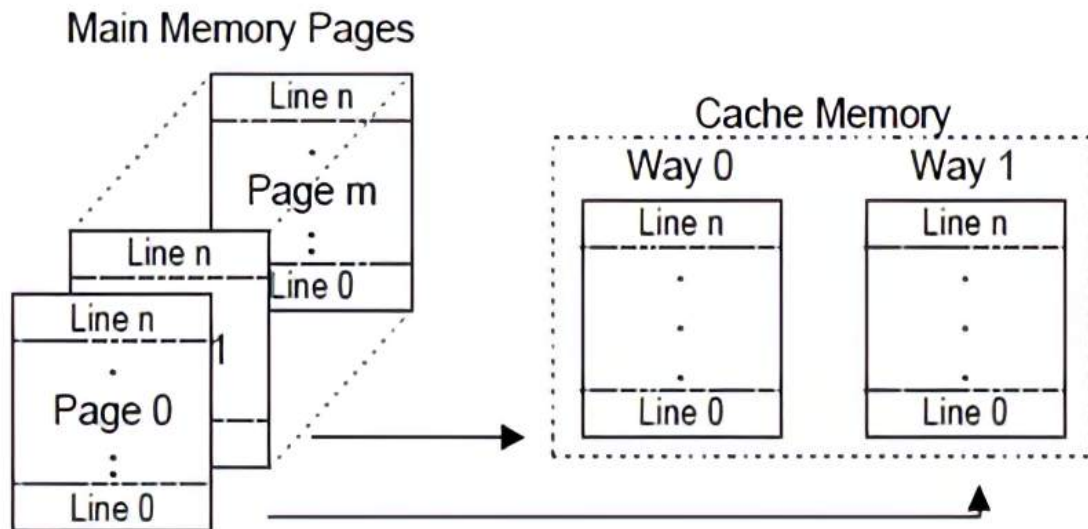


Figure 2-7 2-Way Set Associative

A **Set-Associative cache** scheme is a combination of Fully-Associative and Direct Mapped caching schemes. A set-associate scheme works by dividing the cache SRAM into equal sections (2 or 4 sections typically) called cache ways. The cache page size is equal to the size of the cache way. Each cache way is treated like a small direct mapped cache. To make the explanation clearer, let's look at a specific example. Figure 2-7 shows a diagram of a 2-Way Set Associative cache scheme. In this scheme, two lines of memory may be stored at any time. This helps to reduce the number of times the cache line data is written-over?

This scheme is less complex than a Fully-Associative cache because the number of comparators is equal to the number of cache ways. A 2-Way Set-Associate cache only requires two comparators making this scheme less expensive than a fully-associative scheme.

The Pentium(R) Processors Cache —

This section examines internal cache on the Pentium(R) processor. The purpose of this section is to describe the cache scheme that the Pentium(R) processor uses and to provide an overview of how the Pentium(R) processor maintains cache consistency within a system.

The above section broke cache into neat little categories. However, in actual implementations, cache is often a series of combinations of all the above mentioned categories. The concepts are the same, only the boundaries are different.

Pentium(R) processor cache is implemented differently than the systems shown in the previous examples. The first difference is the cache system is internal to the processor, i.e. integrated into the part. Therefore, no external hardware is needed to take advantage of this cache - helping to reduce the overall cost of the system. Another advantage is the speed of memory request responses. For example, a 100MHz Pentium(R) processor has an external bus speed of 66MHz. All external cache must operate at a maximum speed of 66mhz. However, an internal cache operates at 100MHz. Not only does the internal cache respond faster, it also has a wider data interface. An external interface is only 64-bits wide while the internal interface between the cache and processor prefetch buffer is 256-bits wide. Therefore, a huge increase in performance is possible by integrating the cache into the CPU.

A third difference is that the cache is divided into two separate pieces to improve performance - a data cache and a code cache, each at 8K.. This division allows both code and data to readily cross page boundaries without having to overwrite one another.

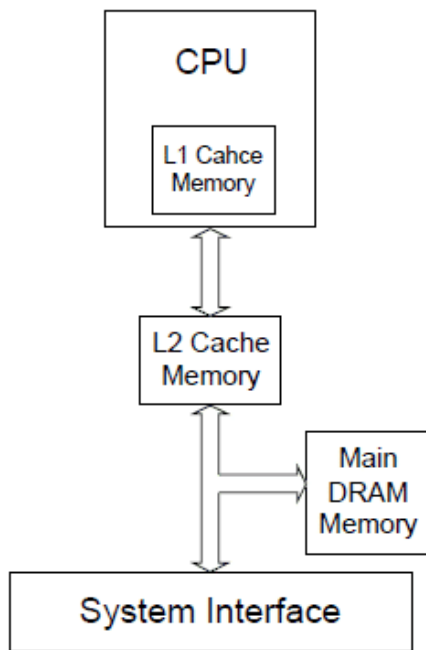


Figure 3-1 Pentium® Processor with L2 cache

Cache Organization

Both caches are 2-way set-associative in structure. The cache line size is 32 bytes, or 256 bits. A cache line is filled by a burst of four reads on the processor's 64-bit data bus. Each cache way contains 128 cache lines. The cache page size is 4K, or 128 lines. Figure 3-2 shows a diagram of the 2-way set-associate scheme with the line numbers filled in.

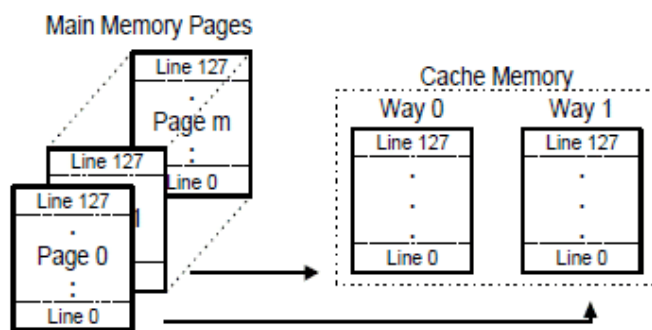


Figure 3-2 Internal Pentium® Processor Cache Scheme

Operating Modes

Unlike the cache systems discussed in section 2 (An Overview of Cache), the write policy on the Pentium(R) processor allows the software to control how the cache will function. The bits that control the cache are the CD (cache disable) and NW (not write-through) bits. As the name suggests, the CD bit allows the user to disable the Pentium(R) processors internal cache. When CD = 1, the cache is disabled, CD = 0 cache is enabled. The NW bit allows the cache to be either write-through (NW = 0) or write-back (NW = 1).

Cache Consistency —

The Pentium(R) processor maintains cache consistency with the MESI protocol. MESI stands for the four states a cache line can be: Modified (M), Exclusive (E), Shared (S), and Invalid (I). For more detailed information about MESI, please refer to the Developer's Manual.

MESI is used to allow the cache to decide if a memory entry should be updated or invalidated. With the Pentium(R) processor, two functions are performed to allow its internal cache to stay consistent, Snoop Cycles and Cache Flushing.

The Pentium(R) processor snoops during memory transactions on the system bus. That is, when another bus master performs a write, the Pentium(R) processor snoops the address. If the Pentium(R) processor contains the data, the processor will schedule a write-back.

Cache flushing is the mechanism by which the Pentium(R) processor clears its cache. A cache flush may result from actions in either hardware or software. During a cache flush, the Pentium(R) processor writes back all modified (or dirty) data. It then invalidates its cache, (i.e., makes all cache lines unavailable). After the Pentium(R) processor finishes its write-backs, it then generates a special bus cycle called the Flush Acknowledge Cycle. This signal allows lower level caches, e.g. L2 caches, to flush their contents as well.

Conclusion —

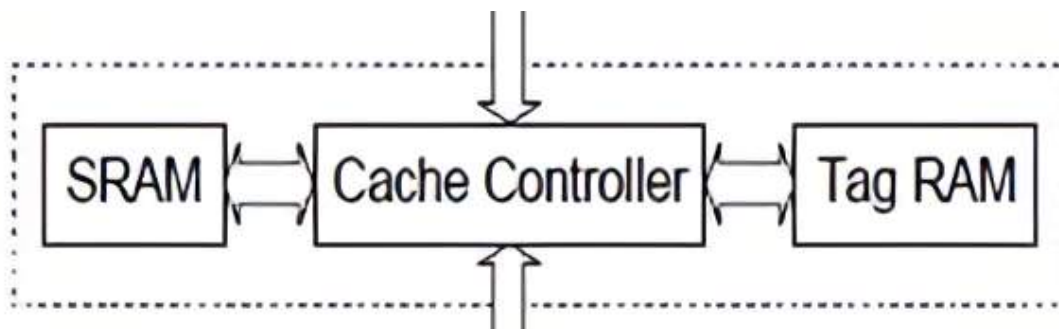
Caches are implemented in a variety of ways, though the basic concepts of cache are the same. As shown above in the overview of cache, most systems have the major pieces – SRAM, TRAM, and a controller. However, you can see the details in the Pentium(R) Processors implementation cross several of the initially defined boundaries. It is important to understand that the Pentium(R) Processor uses only one method to implement cache. There are many other valid way to do the same things, but, they all lead to the same place. Cache is simply a high speed piece of memory that stores a snapshot of main memory which enables the processor higher performance.

This is the end of the Intel Article.

Detailed Design of Cache Memory

At this point the overall terms and different organizations of cache memory have been covered. The details the different parts of cache will be covered in this section.

It should be obvious at this point that cache not only needs to contain the contents of parts of main memory (instructions or data) but also the address of main memory that is being stored in the cache. In Figures 2-2 and 2-3 above a block diagram was given that provide some more detail on what cache consisted of.

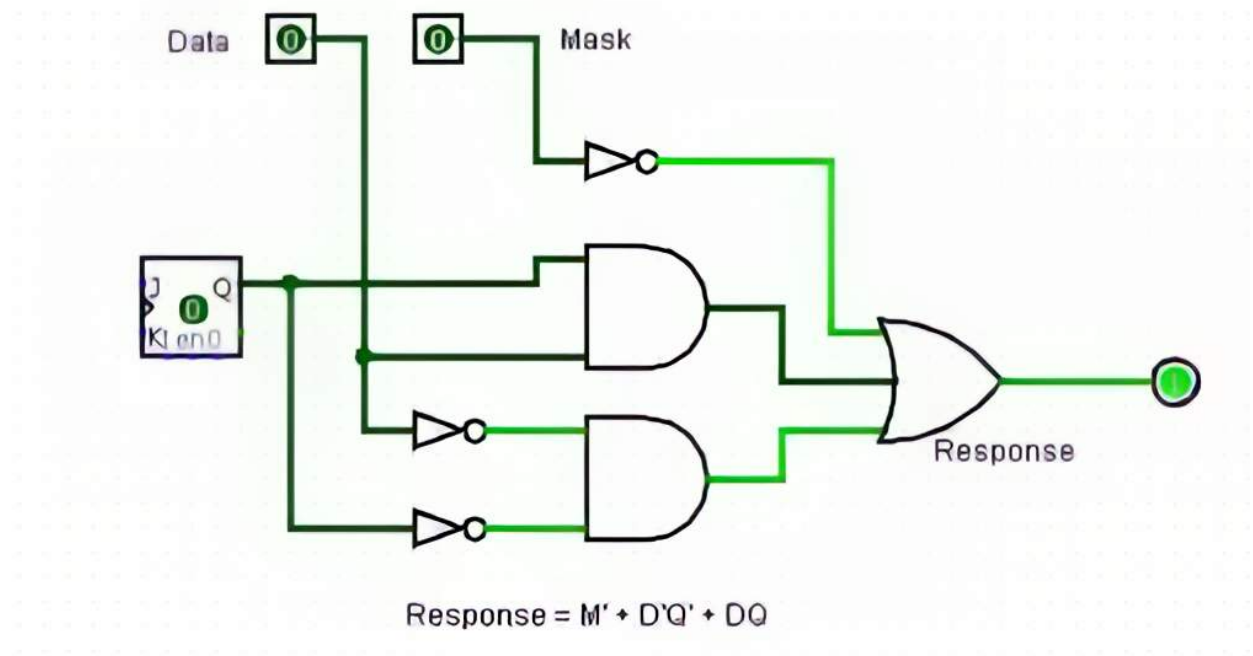


The details of these three components are:

1. **SRAM** - Static Random Access Memory is made of memory components that were described in Module 03. This contains the data or instructions that is being accessed by the CPU. The difference between SRAM and DRAM will be in the next Module.
2. **Cache Controller** - Well described in the article above.
3. **Tag RAM** - is made of what is called Associative Memory. This contains the address of what is contained in the SRAM and determines if the referenced address is or is not in cache.

What you may be thinking is: "How can cache be so fast if we have to search through all of the Tag RAM to see if there is a hit or miss?" Short answer is it doesn't search through this memory the way you think it does. A unique piece of hardware called Associative Memory is

used to store data and quickly determine if there is a cache hit or cache miss. The figure below shows one cell of associative memory. This hardware will compare what is in the JK-Flip Flop to what is on the Data line but only if the bit is not Mask off.



Associative Memory can quickly determine if it contains a match with what is on the data line. The time it takes to determine the Response is just a few gate delays which is measured in a few nanoseconds. Because of all this additional hardware Associative Memory is very fast to determine if there is a cache hit or miss.

Now that we have covered the details on the hardware of Cache how is it put together to in a real system? The best way to show this is with an example.

Cache Design Example

The computer system in this example has the following:

24-bit address line

Word size of 8-bits or one Byte

This translates into main memory being 16MB in total size. The calculation is $2^{24} = 16,777,216$ or 16M and each memory location accesses 8-bits or one Byte.

The cache is 64KB in size. In this example the cache is direct map for fully associative cache memory. Its organization is:

Each cache line consists of 16-Bytes or 16 main memory locations. Organizing cache this way reduces the number of address bit that need to be in Associative Memory thus reducing its cost. Instead of needed to track the whole 24-bit address on the most significant 20-bits of the address are needed to be stored.

64 KB of cache. Each cache line is four 32-bit (4 bytes) words or 16-Bytes. Because of this we do not need to store the last 4 LSBs of the address in the cache assuming we keep up with the byte addressable memory.

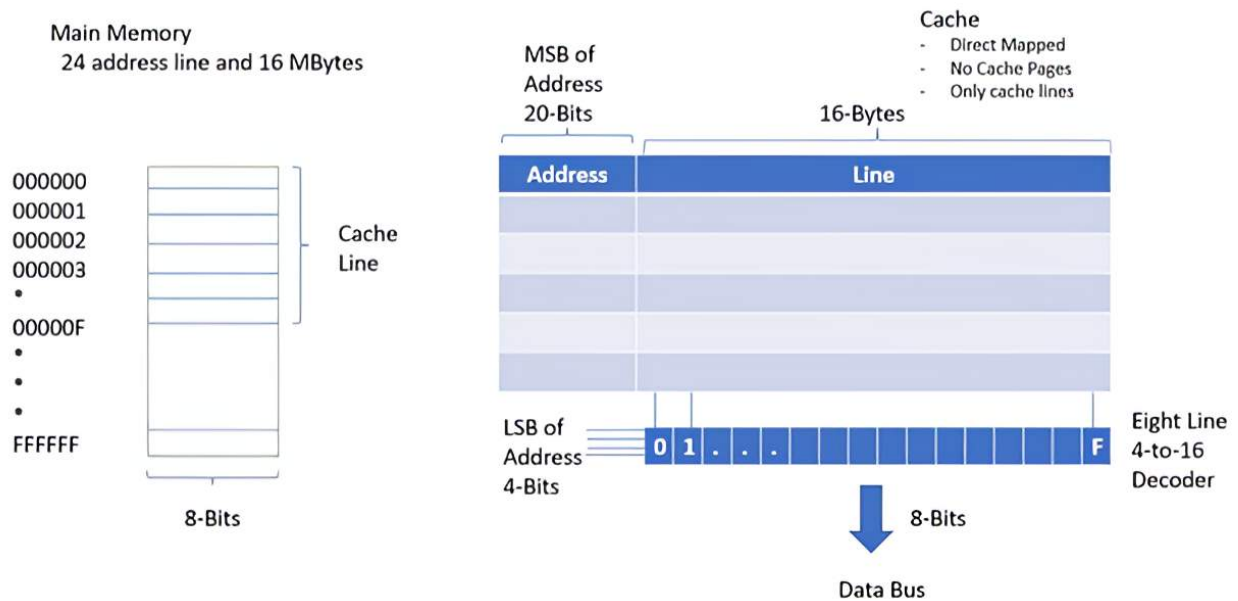
Main memory has 24 address bits.

Main Memory size: $2^{24} = 16,777,216$ 16M Words

Each cache line has 16-bytes which takes 4 bits for each line to select the byte we want.

Only need $24 - 4 = 20$ bits to specify the cache line address. This will be the 20 most significant bits of the main memory address.

If there is a cache hit then the entire cache line is returned. Associative memory is used for these 20 address bits then a 8 line 16-to-1 mux is used to select the reference Byte OR 4-to-1 mux to select the word.



Fully Associative cache is the most flexible of the other cache designs. It is also the most expensive to build. The n-Way set associative cache's design is not as flexible but is less complex in its design thus it is cheaper than fully associative cache.

Cache Misses

A cache miss is when a CPU memory reference is made and that reference is not in the cache.

The three classifications of cache misses for set-associative cache:

1. Compulsory - The miss cannot be avoided. First reference. Can try to make this better by using pre-fetching by guessing what the next reference will be.
2. Capacity - Ran out of room in the cache. The cache gets full and then we get a reference to memory which is not in the cache
3. Conflict - The cache is not yet full but the line needs to be replaced. There is free space in the cache but the line we need already has a different line stored there.

Cache Replacement Policy

If there is a cache miss that is related to a capacity or conflict miss the next question is which cache line gets replaced?

With the 1-Way set there is no decision to be made. The line that is needed replaces the current line in the cache.

Fully associative and n-Way the problem is a little more complex. The more complex the replacement policy the more hardware is needed to implement which makes it more expensive and slower. The two most common replacement policies are:

1. Random - Just pick one. It is quick and requires little hardware to implement.
2. Least recently used (LRU) - Whichever is the oldest line that has not had a recent referenced.

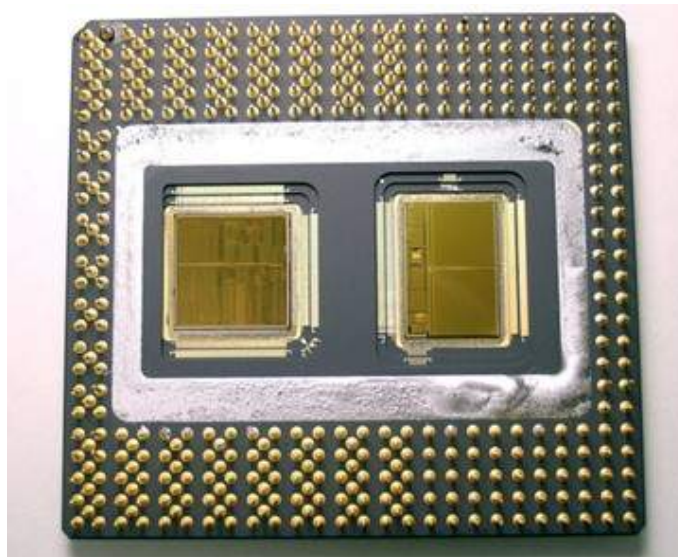
Of course it would be best to pick one that does not require a write-back to main memory.

Levels of Cache and Organization

There are several different levels of Cache and there seems to be more all the time.

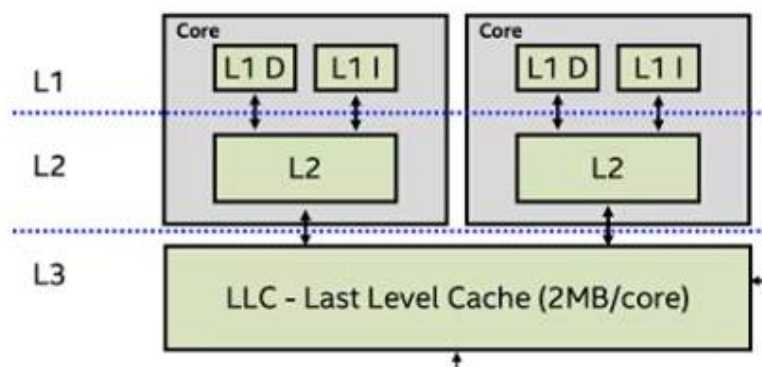
Cache is identified as Level 1 (L1), Level 2 (L2) and Level 3 (L3):

- L1 is located on the same piece of the silicon as the CPU. L1 is the smallest cache and the fastest. L1 size is often measured in the Kbyte range.
- L2 is larger than L1 cache and is located on the same chip as the CPU. L2 is larger than L1 and is often in the hundreds of Kbyte.
- L3 is larger than L2 cache as is located off chip. L3 is larger than L2 and is often in the Mbyte range.



The rectangle on the left is a CPU and has a L1 cache
 The rectangle on the right is the L2 cache

Some CPUs may have separate caches for data and instructions as shown in the figure below.



Conclusion

There are still a lot more details related to cache design that have not been covered in this module. Also not covered are all the studies related to cache design trade-offs and implementation details that effect cache, CPU and computer performance. Think of this as a disclaimer. There is a lot of detailed information that have been covered about Cache but there is a real possibility that you will run across cache design and terminology that has not been covered in this module.

Main and Virtual Memory



Objectives

From this module students should learn:

- Memory Hierarchy
- Random Access Memory technologies
- Read Only Memory devices
- Hardware support for Virtual Memory

Students should:

- Read the following material
- Watch the associated video lecture
- Take the Canvas quiz on this topic

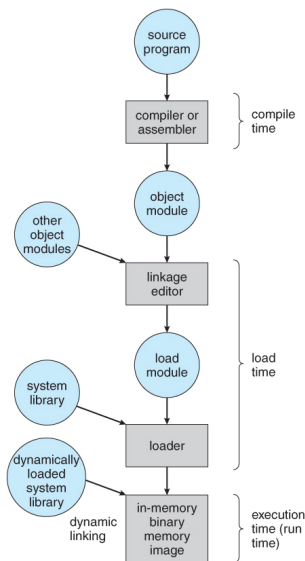


How Computer Memory Works

Types of Computer Memory - Types of computer memory include two caches, system RAM, virtual memory and a hard drive. Learn about the types of computer memory and what they do.

READ MORE HOWSTUFFWORKS >

UIC



Main Memory

The CPU can only access its registers and main memory. It cannot, for example, make direct access to the hard drive, so any data stored there must first be transferred into the main memory chips before the CPU can work with it.

Main Memory

Main memory stores data and allows reading of data. That is about it.

Memory's function is not complicated, just how it is organized in order to:

1. Make it faster
2. Make it cheaper
3. Make it take less power
4. Make it take up less space.

From a software standpoint main memory looks like a big array. The array subscript can be thought of as the memory address and what is stored in that array element is what is stored in that memory location.

Address	Contents
0000	DE00
0001	0F78
0002	4230
FFFF	983C

This section will cover the various type of computer memory and how it has evolved to what is used in most computers and portable devices today. The implementation details of memory

design are not covered in this course.

RANDOM ACCESS MEMORY (RAM)	STATIC RAM (SRAM)	DYNAMIC RAM (DRAM)	STATIC DYNAMIC RAM (SDRAM)
-------------------------------	-------------------	--------------------	-------------------------------

RAM (Random Access Memory) is the internal memory of the CPU for storing data, program, and program result. It is a read/write memory which stores data while the machine is working. As soon as the machine is switched off, data is erased.

Access time in RAM is independent of the address, that is, each storage location inside the memory is as easy to reach as other locations and takes the same amount of time. Data in the RAM can be accessed randomly but it is very expensive.

RAM is volatile, i.e. data stored in it is lost when we switch off the computer or if there is a power failure. Hence, a backup Uninterruptible Power System (UPS) is often used with computers. RAM is small, both in terms of its physical size and in the amount of data it can hold.

RAM is of two types:

- Static RAM (SRAM)
- Dynamic RAM (DRAM)



RANDOM ACCESS
MEMORY (RAM)

STATIC RAM (SRAM)

DYNAMIC RAM (DRAM)

STATIC DYNAMIC RAM
(SDRAM)

The word static indicates that the memory retains its contents as long as power is being supplied. However, data is lost when the power gets down due to volatile nature. SRAM chips use a matrix of 6-transistors and no capacitors. Transistors do not require power to prevent leakage, so SRAM need not be refreshed on a regular basis. This is the same design that was covered in Module 03.

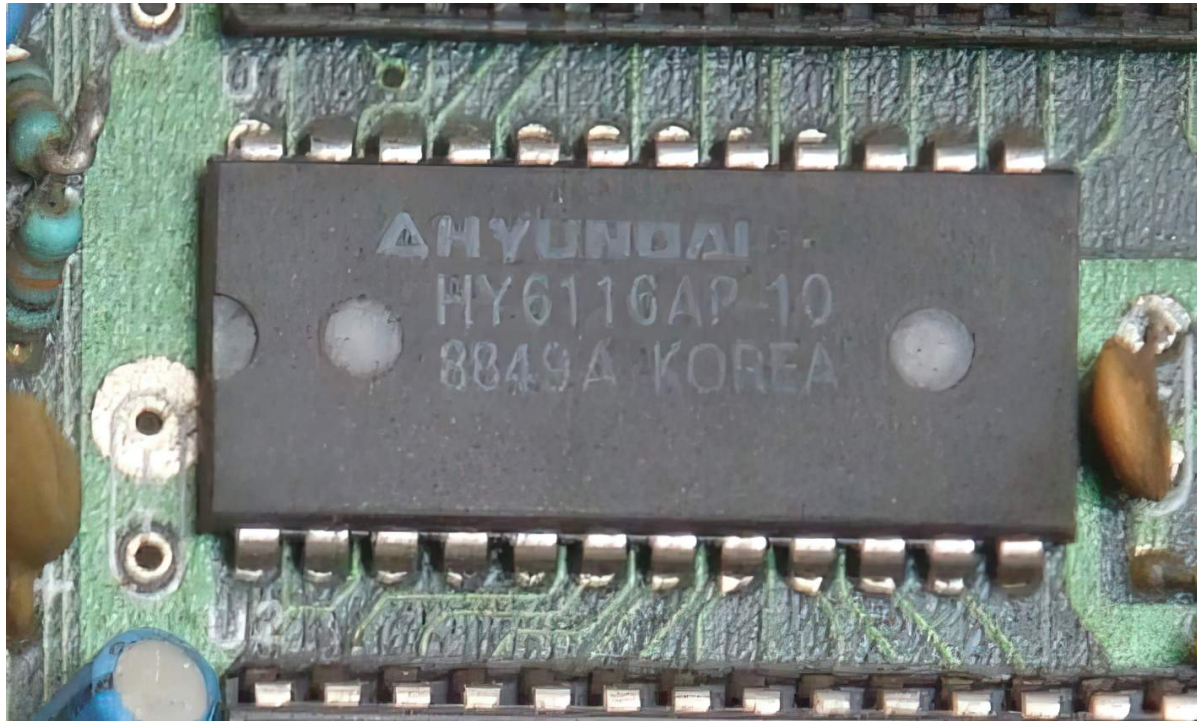
SRAM takes up more room on the silicon than DRAM for the same amount of storage space, making the manufacturing costs higher. As mentioned in Module 11 SRAM is used as cache memory and has very fast access.

Characteristic of Static RAM

- Long life
- No need to refresh
- Faster
- Used as cache memory
- Large size
- Expensive

- High power consumption

Image: By Satyamfifa – Own work, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=10606128>



RANDOM ACCESS
MEMORY (RAM)

STATIC RAM (SRAM)

DYNAMIC RAM (DRAM)

STATIC DYNAMIC RAM
(SDRAM)

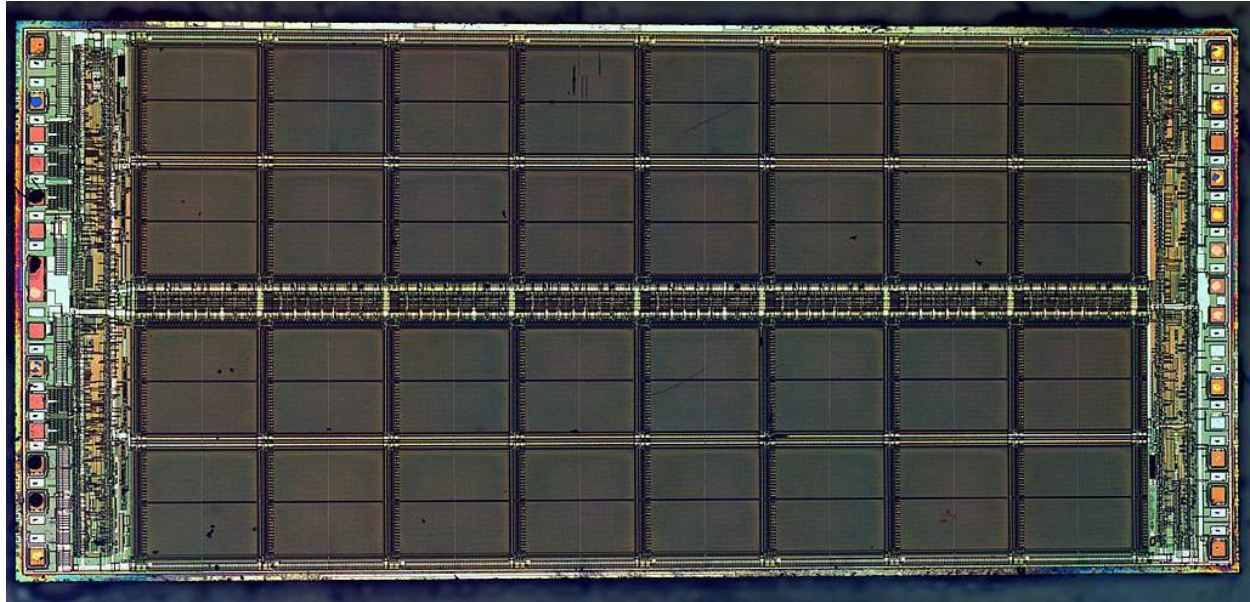
DRAM, unlike SRAM, must be refreshed on a periodic basis in order to maintain the data. This is done by placing the memory on a refresh circuit that rewrites the data several hundred times per second. DRAM is used for most system memory as it is cheap and small. All DRAMs are made up of memory cells, which are composed of one capacitor and one transistor.

Characteristics of Dynamic RAM

- Short data lifetime
- Needs to be refreshed continuously
- Slower as compared to SRAM
- Used as RAM

- Smaller in size
- Less expensive
- Less power consumption

Image: ZeptoBars, CC BY 3.0 <<https://creativecommons.org/licenses/by/3.0/>>, via Wikimedia Commons



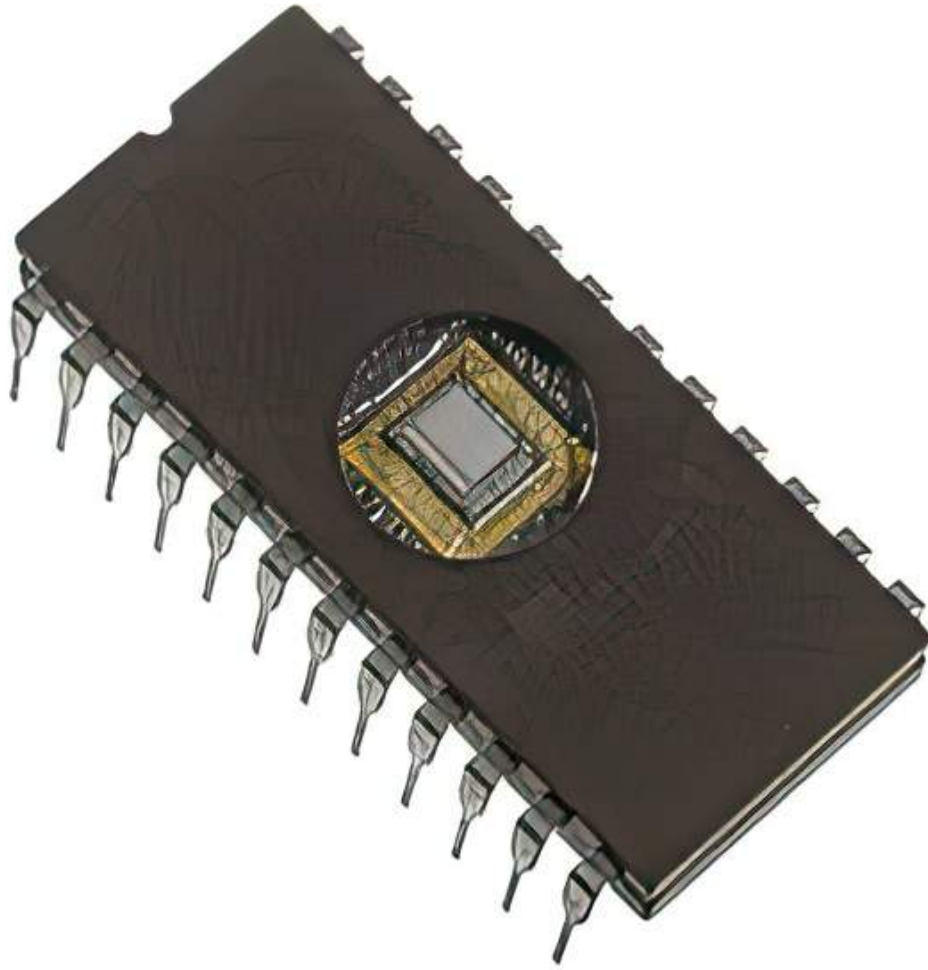
RANDOM ACCESS MEMORY (RAM)	STATIC RAM (SRAM)	DYNAMIC RAM (DRAM)	STATIC DYNAMIC RAM (SDRAM)
To increase the speed of main memory and take advantage of the density of DRAM, Static RAM technology was added to DRAM technology. There is not time in this course to cover the details of the design. The way the design works is that the data contained in the DRAM is transferred to the SRAM components in a "parallel" fashion. Once loaded the SRAM can deliver the data at a much higher rate than the DRAM alone can provide. This increase in speed after the initial loading of the SRAM is referred to as "burst mode".			

DDR SDRAM

Double Data Rate (DDR) is based on the SDRAM technology. You can think of DDR as the second generation of the SDRAM. Internally the DDR SDRAM is not that different from SDRAM. The big difference is with the interface. The SDRAM part of the name was dropped so it was marketed as DDR memory. As new versions of DDR came out the manufacturers used a number to identify the next generation. That is why you see your computer's main memory as DDR2, DDR3, DDR4, DDR5, etc.

Read Only Memory (ROM)

ROM stands for Read Only Memory. The memory from which we can only read but cannot write on it. Actually, these devices are written to at least once. The write process for ROM is referred to as "burn-in". This type of memory is non-volatile. The information is stored permanently in such memories during manufacture. A ROM stores such instructions that are required to start a computer. This operation is referred to as bootstrap. ROM chips are not only used in the computer but also in other electronic items like washing machine and microwave oven.



Let us now discuss the various types of ROMs and their characteristics.

MROM (Masked ROM)

The very first ROMs were hard-wired devices that contained a pre-programmed set of data or instructions. These kind of ROMs are known as masked ROMs, which are inexpensive. These devices can only be programmed one time.

PROM (Programmable Read Only Memory)

PROM is read-only memory that can be modified only once by a user. The user buys a blank PROM and enters the desired contents using a PROM program. Inside the PROM chip, there are small fuses which are burnt open during programming. It can be programmed only once and is not erasable.

EPROM (Erasable Programmable Read Only Memory)

EPROM can be erased by exposing it to ultra-violet light. Usually, an EPROM eraser achieves this function. During programming, an electrical charge is trapped in an insulated gate region. The charge is retained for more than 10 years because the charge has no leakage path. For erasing this charge, ultra-violet light is passed through a quartz crystal window (lid). This exposure to ultra-violet light dissipates the charge. During normal use, the quartz lid is sealed with a sticker.

EEPROM (Electrically Erasable Programmable Read Only Memory)

EEPROM is programmed and erased electrically. It can be erased and reprogrammed several thousand times. Both erasing and programming take about 4 to 10 ms (millisecond). In EEPROM, any location can be selectively erased and programmed. EEPROMs can be erased one byte at a time, rather than erasing the entire chip. Hence, the process of reprogramming is flexible but slow. EEPROMs technology is what is used in USB drives and Solid State Drives (SSDs).

Virtual Memory

Virtual Memory is a storage allocation scheme in which secondary storage (hard drives and solid state drives) can be addressed as though it were part of the main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses.

The size of virtual storage is limited by the addressing scheme of the computer system and the amount of secondary storage that is available not by the actual number of the main storage locations.

It is a technique that is implemented using both hardware and software. This section of the course will address the hardware that is needed to implement virtual memory. The hardware maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

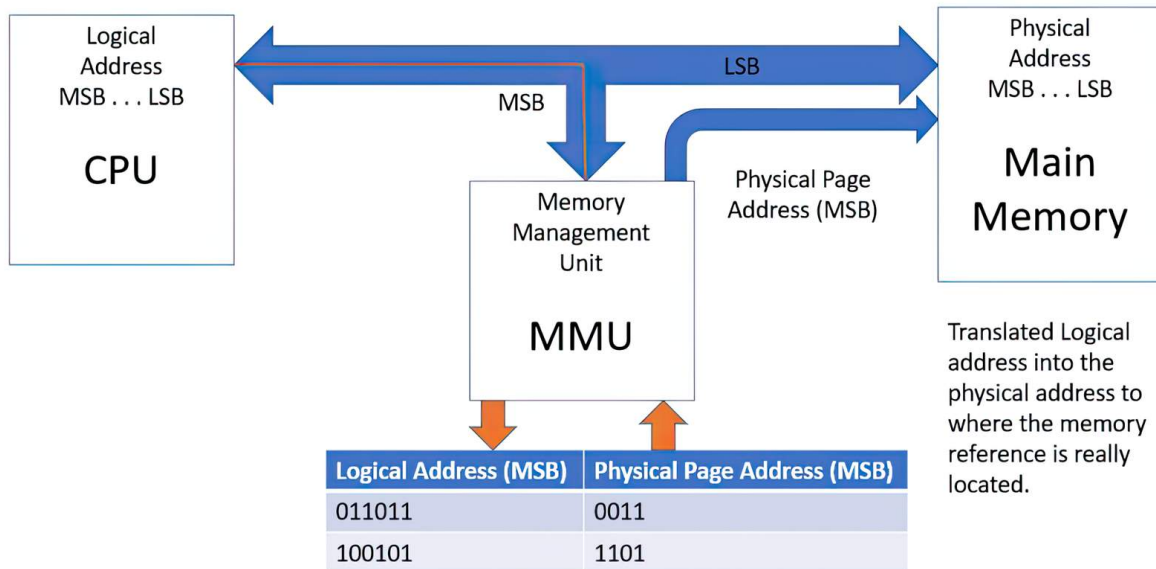
All memory references within a process are **logical addresses** that are dynamically translated into **physical addresses** at run time. This means that a process can be swapped in and out of the main memory such that it occupies different places in the main memory at different times during the course of execution.

A process may be broken into a number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and use of page or segment table permits this.

If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution.

Memory Management Unit (MMU)

The major piece of hardware that is needed to support virtual memory is the Memory Management Unit (MMU).



This is the process as show in the figure above:

1. The CPU has a memory reference. This is referred to as the **logical address**.
2. The most significant bits (MSBs) are provided to the Memory Management Unit (MMU).
3. The MMU consists of associative memory and it searches for the logical address and if it is present will return the MSBs of the physical page address.
4. The physical page address MSBs are appended to the LSBs of the logical address to create the physical address that is in main memory.

There are some additional functions and checks that have to take place in order to make this hardware to work correctly. If the logical address is not in the MMU an interrupt is generated to inform the CPU that the memory reference is currently not in main memory. When this happens the Operating System need to kick-in to load the data into main memory and provide the mapping information to the CPU so it can update the MMU.

More about virtual memory management under the Operating System part of this course.

Resources:

<https://www.geeksforgeeks.org/virtual-memory-in-operating-system/>

https://www.tutorialspoint.com/computer_fundamentals/computer_ram.htm

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/8_MainMemory.html

<https://computer.howstuffworks.com/computer-memory2.htm>

Input/Output



Objectives

From this module students should learn:

- Memory Mapped
- Programmed I/O.
- Interrupt I/O
- Direct Memory Access

Students should:

- Read the following material
- Watch the associated video lecture
- Take the Canvas quiz on this topic

<https://www.geeksforgeeks.org/io-interface-interrupt-dma-mode/>

In order to properly perform input/output hardware connections need to be made between the CPU and the interfacing I/O devices. Each I/O method/type requires its own unique hardware set-up. Of course, with anything to do with modern computers the I/O set-up and protocols can become very complicated. Computer systems typically support multiple I/O schemes at the same time. We will cover each of the major I/O methods that are in common use.

Data transfer to and from the peripherals may be done in any of these four possible ways:

- Memory Mapped
- Programmed I/O
- Interrupt- initiated I/O
- Direct memory access (DMA)

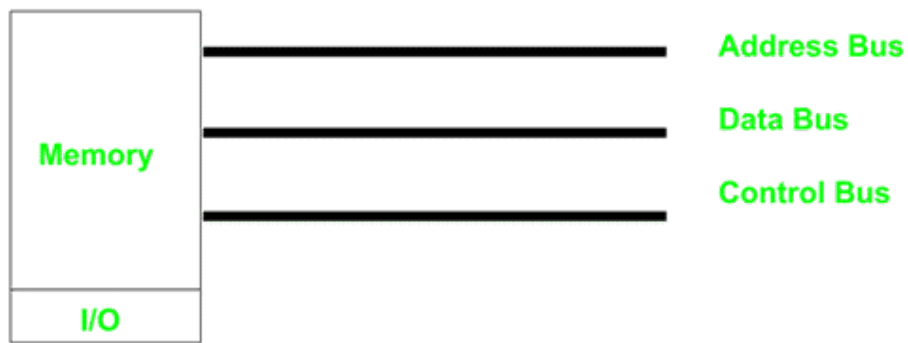
Each of these I/O methods are covered in detailed in the following sections.

Regardless of the I/O modes they all have some things in common. They need to:

- Connect to the I/O device- Select the device.
- Initiate the I/O - Make sure the device is ready for I/O and there are no error conditions.
- Transfer the data
- Conclude the I/O - Check for any error codes and close the I/O device so that it can be ready for the next I/O.

Memory Mapped

As implied in the name the I/O devices are given addresses in main memory in order to perform I/O. To perform an output operation the only thing that is needed by the CPU is to do a memory store or write operation. A read operation just uses a memory read. The figure below illustrates the relations of main memory and the I/O devices.



Differences between memory mapped I/O and isolated I/O –

Isolated I/O	Memory Mapped I/O
Memory and I/O have separate address space	Both have same address space
All address can be used by the memory	Due to addition of I/O addressable memory become less for memory
Separate instruction control read and write operation in I/O and Memory	Same instructions can control both I/O and Memory
In this I/O address are called ports.	Normal memory address are for both
More efficient due to separate buses	Lesser efficient
Larger in size due to more buses	Smaller in size
It is complex due to separate logic	Simpler logic is used as I/O is also

Isolated I/O	Memory Mapped I/O
is used to control both.	treated as memory only.

Advantages of Memory-Mapped I/O:

Faster I/O Operations: Memory-mapped I/O allows the CPU to access I/O devices at the same speed as it accesses memory. This means that I/O operations can be performed much faster compared to isolated I/O.

Simplified Programming: Memory-mapped I/O simplifies programming as the same instructions can be used to access memory and I/O devices. This means that software developers do not have to use specialized I/O instructions, which can reduce programming complexity.

Efficient Use of Memory Space: Memory-mapped I/O is more memory-efficient as I/O devices share the same address space as the memory. This means that the same memory address space can be used to access both memory and I/O devices.

Disadvantages of Memory-Mapped I/O:

Limited I/O Address Space: Memory-mapped I/O limits the I/O address space as I/O devices share the same address space as the memory. This means that there may not be enough address space available to address all I/O devices.

Slower Response Time: If an I/O device is slow to respond, it can delay the CPU's access to memory. This can lead to slower overall system performance

Memory-mapped I/O applications:

Graphics processing: Memory-mapped I/O is often used in graphics cards to provide fast access to frame buffers and control registers. The graphics data is mapped directly to

memory, allowing the CPU to read from and write to the graphics card as if it were accessing regular memory.

Network communication: Network interface cards (NICs) often utilize memory-mapped I/O to transfer data between the network and the system memory. The NIC registers are mapped to specific memory addresses, enabling efficient data transfer and control over network operations.

Direct memory access (DMA): DMA controllers employ memory-mapped I/O to facilitate high-speed data transfers between devices and system memory without CPU intervention. By mapping the DMA controller registers to memory, data can be transferred directly between devices and memory, reducing CPU overhead.

Programmed I/O

It is due to the result of the I/O instructions that are written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually the transfer is from a CPU register and memory. In this case it requires constant monitoring by the CPU of the peripheral devices.

Example of Programmed I/O: In this case, the I/O device does not have direct access to the memory unit. A transfer from I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from device to the CPU and store instruction to transfer the data from CPU to memory. In programmed I/O, the CPU stays in the program loop until the I/O unit indicates that it is ready for data transfer. This is a time consuming process since it needlessly keeps the CPU busy. This situation can be avoided by using an interrupt facility. This is discussed below.

Interrupt- initiated I/O

Since in the above case we saw the CPU is kept busy unnecessarily. This situation can very well be avoided by using an interrupt driven method for data transfer. By using interrupt facility and special commands to inform the interface to issue an interrupt request signal whenever data is available from any device. In the meantime the CPU can proceed for any other program execution. The interface meanwhile keeps monitoring the device. Whenever it is determined that the device is ready for data transfer it initiates an interrupt request signal to the computer.

Upon detection of an external interrupt signal the CPU stops momentarily the task that it was already performing, branches to the service program to process the I/O transfer, and then return to the task it was originally performing.

1

The I/O transfer rate is limited by the speed with which the processor can test and service a device.

2

The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

Terms

1

Hardware Interrupts: Interrupts present in the hardware pins.

2

Software Interrupts: These are the instructions used in the program whenever the required functionality is needed.

3

Vectored interrupts: These interrupts are associated with the static vector address.

4

Non-vectored interrupts: These interrupts are associated with the dynamic vector address.

5

Maskable Interrupts: These interrupts can be enabled or disabled explicitly.

6

Non-maskable interrupts: These are always in the enabled state. we cannot disable them.

7

External interrupts: Generated by external devices such as I/O.

8

Internal interrupts: These devices are generated by the internal components of the processor such as power failure, error instruction, temperature sensor, etc.

9

Synchronous interrupts: These interrupts are controlled by the fixed time interval. All the interval interrupts are called as synchronous interrupts.

Asynchronous interrupts: These are initiated based on the feedback of previous instructions. All the external interrupts are called as asynchronous interrupts.

Direct Memory Access

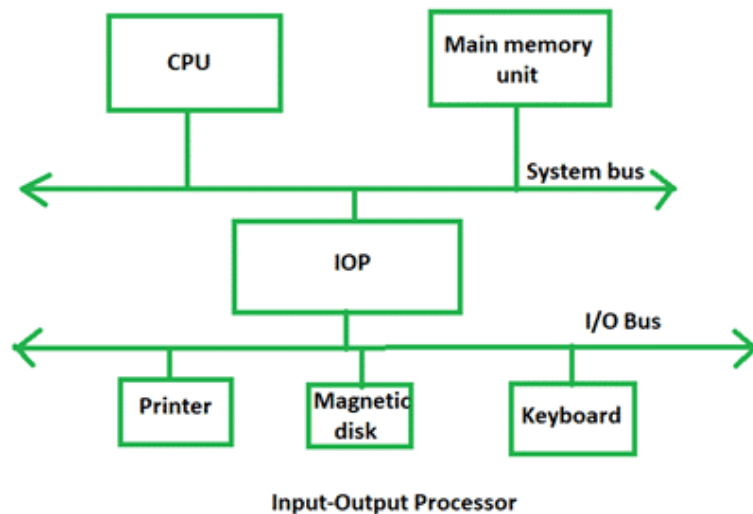
The binary information that is received from an external device is usually stored in the memory unit. The information that is transferred from the CPU to the external device is originated from the memory unit. CPU merely processes the information but the source and target is always the memory unit.

The data transfer between a fast storage media such as magnetic disk and memory unit is limited by the speed of the CPU. Thus we can allow the peripherals directly communicate with each other using the memory buses, removing the intervention of the CPU. This type of data transfer technique is known as DMA or direct memory access. During DMA the CPU is idle and it has no control over the memory buses. The DMA controller takes over the buses to manage the transfer directly between the I/O devices and the memory unit.

- Bus grant request time.
- Transfer the entire block of data at transfer rate of device because the device is usually slow than the speed at which the data can be transferred to CPU.
- Release the control of the bus back to CPU So, total time taken to transfer the N bytes = Bus grant request time + $(N) * (\text{memory transfer rate})$ + Bus release control time.
- Buffer the byte into the buffer
- Inform the CPU that the device has 1 byte to transfer (i.e. bus grant request)
- Transfer the byte (at system bus speed)
- Release the control of the bus back to CPU.

The **DMA mode** of data transfer reduces CPU's overhead in handling I/O operations. It also allows parallelism in CPU and I/O operations. Such parallelism is necessary to avoid wastage of valuable CPU time while handling I/O devices whose speeds are much slower as compared to CPU. The concept of DMA operation can be extended to relieve the CPU further from getting involved with the execution of I/O operations. This gives rise to the development of special purpose processor called **Input-Output Processor (IOP)** or IO channel. The Input Output

Processor (IOP) is just like a CPU that handles the details of I/O operations. It is more equipped with facilities than those are available in typical DMA controller. The IOP can fetch and execute its own instructions that are specifically designed to characterize I/O transfers. In addition to the I/O – related tasks, it can perform other processing tasks like arithmetic, logic, branching and code translation. The main memory unit takes the pivotal role. It communicates with processor by the means of DMA. **The block diagram –**



The Input Output Processor is a specialized processor which loads and stores data into memory along with the execution of I/O instructions. It acts as an interface between system and devices. It involves a sequence of events to executing I/O operations and then store the results into the memory.

Features of an IOP

Specialized Hardware

Specialized Hardware: An IOP is equipped with specialized hardware that is optimized for handling input/output operations. This hardware includes input/output ports, DMA controllers, and interrupt controllers.

DMA Capability —

DMA Capability: An IOP has the capability to perform Direct Memory Access (DMA) operations. DMA allows data to be transferred directly between peripheral devices and memory without going through the CPU, thereby freeing up the CPU for other tasks.

Interrupt Handling —

Interrupt Handling: An IOP can handle interrupts from peripheral devices and manage them independently of the CPU. This allows the CPU to focus on executing application programs while the IOP handles interrupts from peripheral devices.

Protocol Handling —

Protocol Handling: An IOP can handle communication protocols for different types of devices such as Ethernet, USB, and SCSI. This allows the IOP to interface with a wide range of devices without requiring additional software support from the CPU.

Buffering —

Buffering: An IOP can buffer data between the CPU and peripheral devices. This allows the IOP to handle large amounts of data without overloading the CPU or the peripheral devices.

Command Processing —

Command Processing: An IOP can process commands from peripheral devices independently of the CPU. This allows the CPU to focus on executing application programs while the IOP handles peripheral device commands.

Parallel Processing —

Parallel Processing: An IOP can perform input/output operations in parallel with the CPU. This allows the system to handle multiple tasks simultaneously and improve overall system performance.

Advantages

- The I/O devices can directly access the main memory without the intervention by the processor in I/O processor based systems.
- It is used to address the problems that arise in Direct memory access method.
- Reduced processor workload: With an I/O processor, the main processor doesn't have to deal with I/O operations, allowing it to focus on other tasks. This results in more efficient use of the processor's resources and can lead to faster overall system performance.
- Improved data transfer rates: Since the I/O processor can access memory directly, data transfers between I/O devices and memory can be faster and more efficient than with other methods.
- Increased system reliability: By offloading I/O tasks to a dedicated processor, the system can be made more fault-tolerant. For example, if an I/O operation fails, it won't affect other system processes.
- Scalability: I/O processor based systems can be designed to scale easily, allowing for additional I/O processors to be added as needed. This can be particularly useful in large-scale data centers or other environments where the number of I/O devices is constantly changing.
- Flexibility: I/O processor based systems can be designed to handle a wide range of I/O devices and interfaces, providing more flexibility in system design and allowing for better customization to meet specific requirements.

Disadvantages

- **Cost:** I/O processors can add significant cost to a system due to the additional hardware and complexity required. This can be a barrier to adoption, especially for smaller systems.
- **Increased complexity:** The addition of an I/O processor can increase the overall complexity of a system, making it more difficult to design, build, and maintain. This can also make it harder to diagnose and troubleshoot issues.
- **Limited performance gains:** While I/O processors can improve system performance by offloading I/O tasks from the main processor, the gains may not be significant in all cases. In some cases, the additional overhead of the I/O processor may actually slow down the system.
- **Synchronization issues:** With multiple processors accessing the same memory, synchronization issues can arise, leading to potential data corruption or other errors.
- **Lack of standardization:** There are many different I/O processor architectures and interfaces available, which can make it difficult to develop standardized software and hardware solutions. This can limit interoperability and make it harder for vendors to develop compatible products.

Operating System: Introduction



Introduction

One of the major responsibilities of the OS is the management of resources. Since you have been a computer user before you entered pre-school some of the resources associated with the computer are very obvious and others not as much.

My short list:

1. CPU and/or Cores
2. Main memory
3. Hard disk space
4. Files
5. Other attached hardware (tape units, network devices, etc.)

You may be asking yourself why you should be concerned about these things since you are the only user of your smart phone, tablet, laptop, desktop or gaming system and there is no one else to compete with?

You may be the only user but there are multiple programs or executables that are running concurrently that you may not even be aware of. There is also an expectation that you paid for the device you need to use the device to its upmost efficiency.

If you paid for a CPU that has eight cores and you only ever use two of those cores because of an issue with the OS THEN you would be more than disappointed!

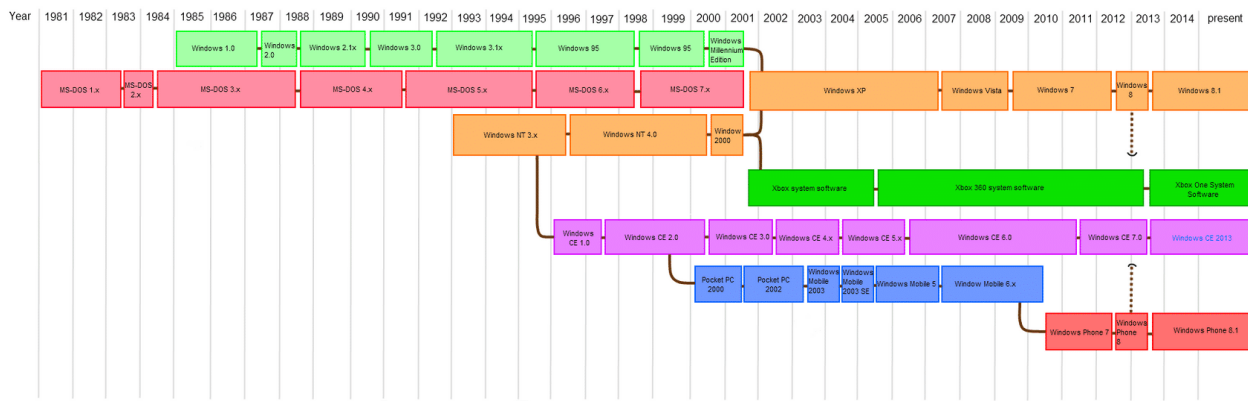


Image by Paulire, This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 License.

Historical development

The modern OS just did not happen overnight. Much like the CPUs themselves the OS has evolved over time. The goal for the OS has always remained the same:

Get maximum use of the computer that we paid a lot of money for.

You can see the change from people following a process, to a single computer operator, to several different programs used to control the user programs, to batch systems, to time-shared to the more modern distributed computing and virtual machines.

Each iteration was to fix issues with previous version or take advantage of new hardware or techniques. Some of these OS technologies are still in our modern OSs while other quickly died away only to be mentioned in some textbook when it covers the history of the OS.

There are several terms in this section that are important. My list may not be all the terms you need to know. The definition provided here is not necessary a complete description.

Term	Definition
Resident monitor or just "monitor"	A program that loads the next program in order, give control to the program and get control back when the program finishes. The monitor would then load the next program.
Control Languages or Job Control Languages	Computer Language whose purpose was to control aspects of the computer system. For example - loading a running a program from the disk.
Interrupt	A hardware signal that causes the CPU to save the current processor status and give control to the interrupt routine and when it is completed return control back to the program that was interrupted.
Scheduler	Program that determines which program to run next.
Spooling system	Next generation of the OS from the monitor. Can pick jobs based on priority and trade out jobs if they are waiting on I/O.
Multiprogramming or Multitasking	OS have several programs that can run at any time and if a program stalls (or has to wait for I/O or something else) another program can start running until that program completes or stalls then the OS can run a different program or swap back to the first program if the wait is over. The OS gives the appearance that several programs are running concurrently but they are only running one at a time but switches them out so quickly it only seems they run in parallel.
Batch System	Simple OS that loads and runs the next program in the queue. The executable code, data, other resources and

Term	Definition
	metadata that are need for the program to run. Much more details on this in a later chapter
CPU-bound	The program that is running spends most of its runtime using the CPU with little or no input/output.
I/O - bound	The program that is running spends most of its runtime doing input/output with little CPU utilization.
Process	The executable code, data, other resources and metadata that are need for the program to run. Much more details on this in a later chapter
Kernel or nucleus	Most frequently accessed parts of the OS.
Concurrency	Programs or parts of programs that are run at the same time on the same Computer
Interactive multiprogramming or time-sharing system	OS that handles multiple interactive jobs between several users. Multiple users have terminals to the CPU. The OS interleaves execution of each user program in short bursts. If there are n users/programs the time slice x is divided between them so they each get x/n time.
Distributed computing	Collection of computers that are made to look like a single computer, main memory, secondary storage.
Multiprocessors	One computer system that consists of several CPUs and/or cores. Programs can actually run in parallel.

The Beautification Principle

This principle is OK but a little mushy, abstract and vague. Security and abstraction are important functions of the OS. More details on these topic will be covered in other modules.

The Kernel and Processes

More details about the kernel will come in later modules. What is part of the kernel varies from OS to OS (Windows, UNIX, Mac OS, etc.). The kernel has the highest priority when running and has access to all parts of the computer system.

Context Block or Process Control Block (PCB) contains the metadata associated with the processes being controlled by the OS.

This data is typically:

- state and scheduling statistics
- use of main and virtual memory
- other resources held
- Input/Output devices required and current status
- Files required and current status
- accounting statistics
- privileges

The PCB can be thought of as a data structure that contains all the information the OS needs to know in order to control and run a program. What all this contains and how it is used is a major part of this part of the course.

A Process switch is when the OS changes the current running process to a different process.

A Context switch is when a running process has its privilege changed in order to access resources that are only accessible by the kernel.

Process lists or Process State - The text only lists three different states a process can be in but there are actually more states that will be covered later. The text is correct that processes will be in one of the following three states:

1. Running

2. Ready

3. Waiting

In order for the OS to better manage the processes additional states are used and will be discussed in a later module.

Service calls are how processes get access to privileged parts of the system. The requests are made to the kernel. These service calls will typically cause the process to leave the running state and go to a waiting state. Mostly because these requests are for I/O which takes a long time in computer terms to respond to and provide access.

Virtual machines is a topic that we do not have time for in this course. These virtual machines are becoming more common place and are not that difficult to understand.

OS Time Management



I am definitely going to take a course on time management...
just as soon as I can work it into my schedule.

Louis E. Boone

Scheduling

Scheduling is broken down into how often each of the queue are checked (managed).

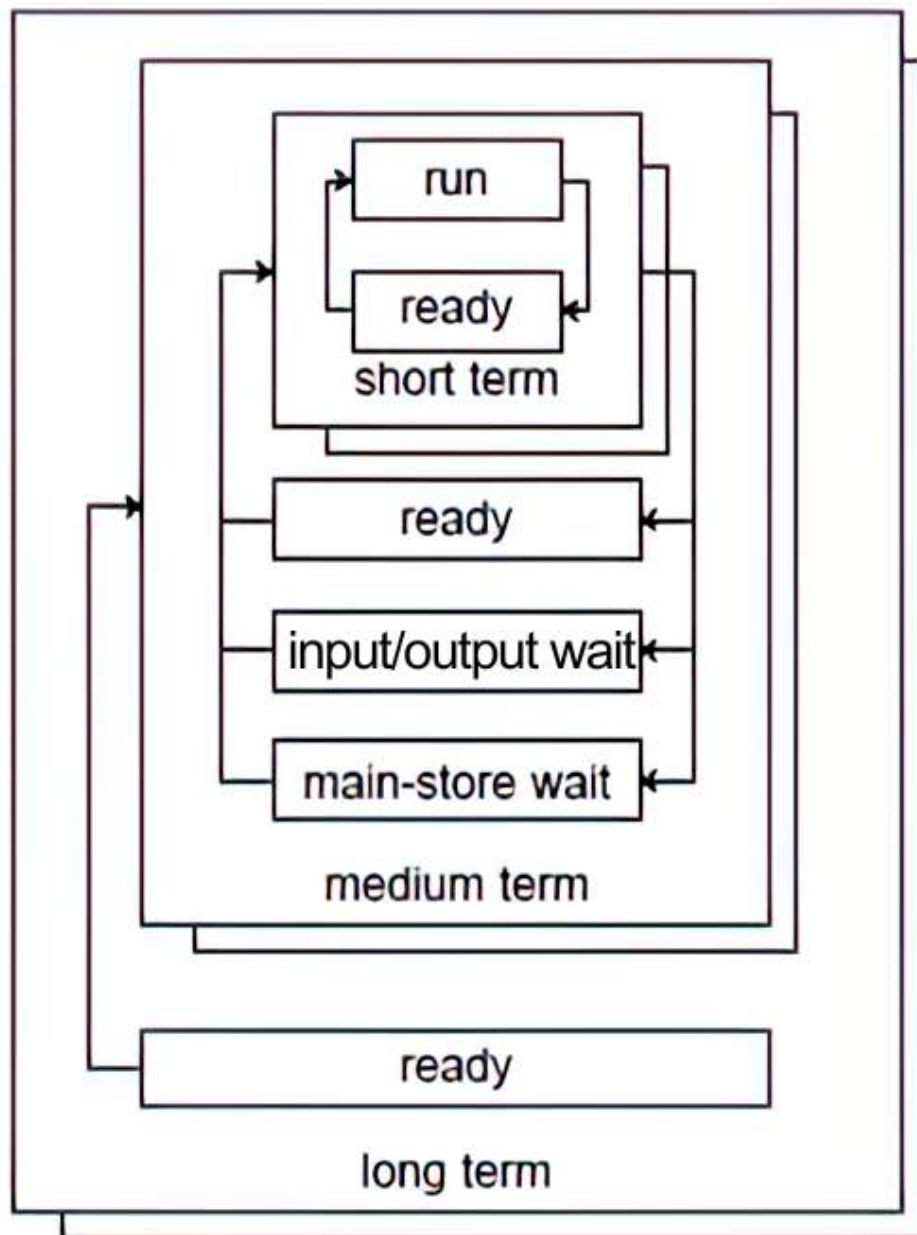


Figure 2.1 Three levels of scheduling

Long Term Scheduling – Deals mostly with the overall system loading of new jobs. Runs the least frequent of all the schedulers.

Medium Term Scheduling – Determines which processes are moved from the waiting states into a running state.

Short Term Scheduling – Determines which process are next to run. Executes the most frequent of the schedulers.

We will stick with I/O Bound and CPU Bound for processes not the terms introduced in this chapter (long process, short processes). I think the terms used in the text will cause confusion later on as we discuss jobs that take a long amount of CPU time to run vs a little amount of CPU time to run.

Goals, measures, and assumptions

What are the goals we are trying to achieve when it comes with processing?

Optimize system objectives:

1. Response time – When a request is made how long until the system responds.
2. Throughput – How much work is completed in a given amount of time.
3. Processor efficiency – How much of the time is the CPU doing real work.

Pick one of the above. At best you may be able to optimize two. With all three there is no way. Then when the mix of jobs is changed then all the optimization that was done can be thrown away.

We are going to stick with the simplest case with a uniprocessor.

Symbol	Description
T	Response Time- Run time t plus missed or wait time
t	How much time process needs to execute
p	Process

Symbol	Description
M	Missed or wait time - $(T - t)$
R	Response Ratio - Percent of the time spend running. If R is 1 then the process starts and runs to completion with no waiting. If R is 0.5 then the process takes twice as long as needed to complete OR half of its time is spend waiting.
P	Penalty ratio - Percent of the time spent waiting. If P is 1 there is no waiting. If P is 2 then the process takes twice as long as needed to complete.

Kernel Time - This is the time the OS spends running and not running user processes. This can be considered overhead.

Idle time - The are times when there are no jobs being run.

If you have had a probability and statics or operation research or industrial engineering class you might have run across some of this information before.

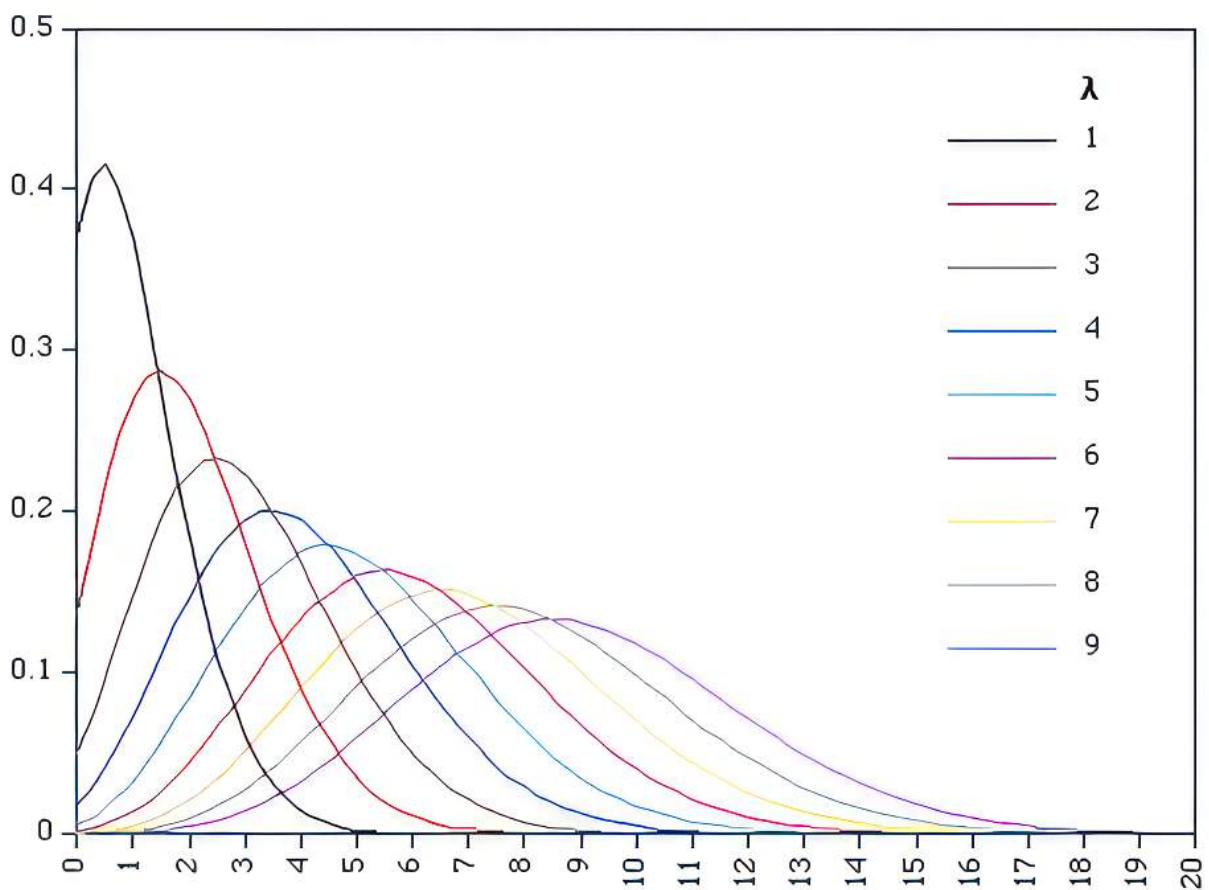
There are two things that need to be known when dealing with any type of queuing and servicing system.

1. When do new jobs show up? Customers showing up at an service area.
2. How much time will the job need to run to completion? Once the customer is at the counter how long does it take for the service are to complete the request.

Arrival and service times are exponential distribution with random rates which are clustered near zero then we have a Poisson distribution. Think of it as customers arriving at a fast food restaurant and how long it takes to wait on a customer.

$$f(x) = \frac{\lambda^x}{x!} e^{-\lambda}$$

Lambda is the mean of the distribution and the textbook uses the Alpha and Beta symbols.



The larger lambda becomes the more this distribution looks like the standard normal (bell curve).

Do not worry about the math. The purpose of this section is get the specifics of the policies not the details on the math. But you should feel good that there is a lot of solid mathematics behind everything in this chapter.

Policies or Scheduling Algorithms

I am not going into details on all of these policies (aka scheduling algorithms). The text does a good job explaining the details on each of these. Each policy goes through a simple example of five processes then a simulation with a large number of processes. The scheduling algorithms discussed here are not all of the algorithms that OS use.

There is new term introduced in this chapter that will become more important in another module.

Starvation - This is when a process does not get its required resources (CPU, Input/Output, Memory, etc.) and is not able to run.

When each of these policies are evaluated in detail there are some general items that concern us:

1. Ease of implementation - If it takes more time to run the policy than it saves in time it may not be useful.
2. Fairness - How does the policy treat different types of jobs (I/O Bound, CPU bound, short, long processes, newly arriving processes vs processes that have been running for a while already). Are there built in biases and it is on purpose.
3. **Starvation** - Is the policy so unfair that it causes processes to starve.

The policies in the text are not all of the policies that are out there. Each one of them can also have variations:

- With or without **priorities**

- With or without **preemption**

As a software engineer it is important to understand the OS you are using and your application to take advantage (or not penalize yourself) of the policies employed by the OS.

The parameters used to compare each of these different policies are:

1. Selection Function - How the next process is selected to run.
2. Decision Mode - Preemptive/non-preemptive
3. Throughput
4. Response time
5. Overhead
6. Effect on processes
7. Starvation - Possible or not possible

But these are not all of the parameters used to compare these policies or scheduling algorithms. Depending on your applications and OS some of these parameters are more important than others. Of course these are not the only policies or scheduling algorithms that are in use.

Non-Preemptive - Once a process start running it will continue to run to completion (or it needs I/O maybe).

Preemptive - A process can be moved from the running state even if it is not blocked by I/O or has not completed.

First come, first served (FCFS) —

It does not come any simpler than this. Also called First-in First-out (FIFO).

Round robin (RR) —

Uses a quantum of time to allocate to each running process. Once there is an I/O request or the time is up the OS gets the next process and starts it running.

Shortest process next (SPN) —

The problem with this one is someone has to make an estimate on how long the process will take to run. The short jobs in the ready queue are run to completion or I/O request blocks them. It would make sense to always guess low BUT the OS may make provisions to penalize those processes if they exceed the guess.

Preemptive shortest process next (PSPN) —

Preempts the current process when another process arrives with a total service time requirement less than the remaining service time required by the current process. The arriving processes are either new ones showing up for ones that are no longer blocked.

Highest penalty ratio next (HPRN) —

Introduces a metric P. The longer the process stays in the system the larger this number gets and the OS picks the process with the highest number. Helps longer jobs to keep them from starving.

Multiple-level feedback (FB) —

Has several different running queues. Always runs the jobs in queue 0 (highest priority) and once it runs its total allotted time it goes to the next lowest queue. Lower priority queues get more time so longer

processes will not starve.

Many different variations of this and could even be "tuned" on the fly.

Selfish round robin (SRR)

Older processors get better service than newer ones. Newer processes priority increase by a different rate than older processes. Can be turned on the fly.

Hybrid methods

Pick any of the previous algorithms and combine them together. Because of this there is no way to compare Hybrid against the other algorithms.

State-dependent priority methods

There are different algorithms for each different state that the process can be in.

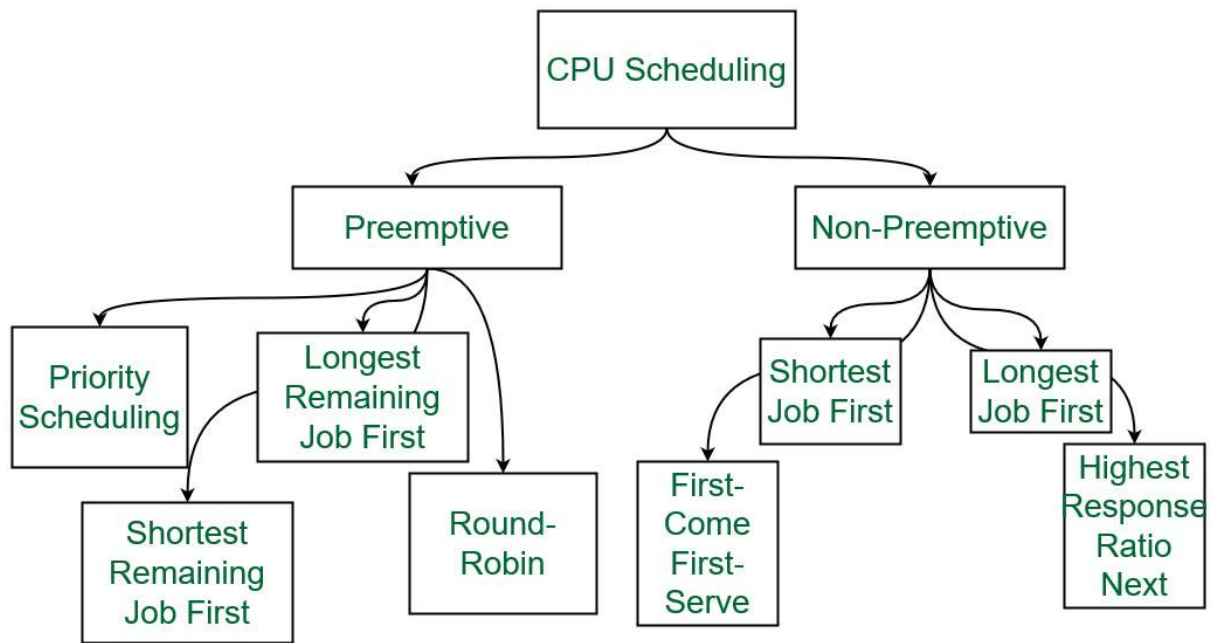
External priority methods

- External priority set by the user
- Greatest suffering limit
- User buys a response ration guarantee

Perspective

No modern OS uses just one of these scheduling policy. OS typically will significantly modify from what is described in the text. Just the movement you think you understand what your OS is doing it can change. Does not even need a "patch" can and will do it on the fly.

CLASSIFICATION OF SCHEDULING POLICIES	EVALUATING POLICIES	SCHEDULING LEVELS	COUNTERMEASURES	GUIDE
How can all these scheduling policies be categorized to make is easier to understand and compare? Try this as an starting point: See Image Intrinsic properties - These are the characteristics of the running processes. CPU time, I/O requests, HRRN, etc. Extrinsic properties - These have to do with the user requests. Urgency, \$\$, etc. Dynamic properties - Not just this one process but take in account all the currently running processes and the status of the computer (CPU utilization, Main Memory, I/O, etc.)				



CLASSIFICATION OF SCHEDULING POLICIES

EVALUATING POLICIES

SCHEDULING LEVELS

COUNTERMEASURES

GUIDE

Always a good idea to use math to compare scheduling policies. The good news is there is a lot of existing analysis on lots of the "base" and some of the "advanced" scheduling policies.

The most popular (and best) method to determine performance is to do model (simulation) of the scheduling policies with the representation of the actual processes that are expected to run. With any simulation (model) the actuals of the real system need to be fed back into the model to make it more accurate. Goal is to get the simulation output to match the real data.

As a last resort the OS can be instrumented to provide some of the "important" data. Remember gathering this data adds to the OS overhead and slows down the system.

CLASSIFICATION OF SCHEDULING POLICIES	EVALUATING POLICIES	SCHEDULING LEVELS	COUNTERMEASURES	GUIDE
There are three levels or frequencies of scheduling: 1. Short 2. Medium 3. Long Each process state queue has its own frequency of scheduling. The closer the process is to the CPU the more frequent the scheduling. There is blurring of lines with each of these scheduling levels.				

CLASSIFICATION OF SCHEDULING POLICIES	EVALUATING POLICIES	SCHEDULING LEVELS	COUNTERMEASURES	GUIDE
This is not a section on OS security. The OS is making an attempt to "keep things fair" and not allow a user to take advantage of what they know about the OS.				

CLASSIFICATION OF SCHEDULING POLICIES	EVALUATING POLICIES	SCHEDULING LEVELS	COUNTERMEASURES	GUIDE
It depends on your system and its workload. The more complex the OS the more overhead it takes to run. With the modern multicore CPUs one or more cores can be given to the OS to keep it running all the time. Not as worried about the overhead of the OS in those cases.				

OS Space Management



A better name for this module would be ***main memory management***. This makes a clear distinction between **cache memory** and **secondary storage**.

Preliminaries

The previous module dealt with CPU utilization this module deals with main memory utilization. Processes cannot run unless at least part of their code and required data are loaded into main memory.

With multiprogramming there needs to be a balance of having enough memory allocated to a process so it can run efficient and enough different processes so the OS can moved quickly from a blocked process to another process that is ready to run.

Processes that the OS has determined cannot or should not run need to be swapped out of main memory to allow processes in main memory that may be able to run.

The rational for swapping processes out of main memory to secondary storage can include:

1. The process is waiting for Input/Output operation or other request and cannot continue running.
2. Higher priority processes are ready to run and need the memory
3. The time allocated for the process has expired

This is not intended to be an all-inclusive list but should give you a good idea on why swapping out processes is necessary.

Thrashing can be problem. **Thrashing** is when the CPU spends more time swapping processes in and out of main memory and as such no useful work gets accomplished.

Storage hierarchies —

Computer storage hierarchy was covered in the hardware section of this course. Not going over that again.

This text is really dated on the sizes and speeds of the level of memory given in Figure 3.1. These exact number are not that important. What is important the relative speed (access time). The farther from the CPU the slower things go.

Physical and virtual store —

Physical memory is the DDR4, DDR5, etc. that you paid a lot of money for on your desktop or laptop computer. This was covered in the hardware section of this course.

Virtual Memory - When the hard drive (secondary storage) is looked to as main memory to allow for larger programs to run in a smaller main memory than was is actually available.

Keep in mind that a process can be thought as sperate executables and its associated data. In a virtual memory system all the address references (program and data) have to be translated to a physical address. Since this is really important and needs to be done quickly there is hardware in place to do the translation. The details on this hardware was covered in Module 12 Main and Virtual Memory.

This text is old so the numbers (sizes) are lower than what is present in modern computer systems. With the ARM CPUs they mostly have 32 bits for address references. This gets 2^{32} or 4 Giga-words of address

range. The Intel Core i7 has a 48-bit virtual address space or a range of 2^{48} or over 280 Tera-words of address range.

Address translation —

The virtual address reference has to be translated to a physical address. Depending on the hardware design this may consist of several tables that track the virtual address references the physical address.

To make the process of tracking, looking-up and translating all the memory references cache memory technology is used. This hardware is given a special name - Translation Look-aside Buffer.

Depending on the OS the kernel is placed in specific physical memory locations and does not need to use the translation tables required for virtual memory. This makes the kernel run faster which is one of the overall goals of OS design. Reducing the overhead of the OS improves the overall performance of the computer system.

Sharing —

OS allows for different processes to run the same code or access the same data. This saves on space and allows often used/accessed code and data to remain in memory and thus save on time with swapping.

An example, having one copy of printf in main memory and every running process accesses the same code.

There are lots of potential issues with sharing programs and data between several running process that will be covered in detail in later modules.

Initial loading and execution —

This section covers the creation, make-up and terminating a process. This Module is mostly about Virtual Memory. It is important to understand the details on the life-cycle of the process because it has an impact on the memory management.

A process can be created from scratch or by splitting or copying parts of an existing process. This load image contains everything needed to execute and manage the process from the OS viewpoint. The parts of the load image include:

1. Header information about the image
2. Program instructions
3. Program data and initialization values
4. Debugging information if applicable
5. Needed space for instructions and data
6. Input/Output and file access requirements

When the demand is made to run a new process the OS has to evaluate the current status of the system to determine if there are sufficient resources for the process to run. The process could be put on hold or out-and-out terminated depending on the current status of the system. Even when the process is allowed to run it can be paused or swapped out depending on the overall demands of the system.

When the process is terminated the OS needs to free up any of the resources allocated to the process so that other process are allowed to use them. This include real and virtual memory and any other allocated items like files and input/output devices.

Fixed partitions

This section starts the different ways virtual memory can be organized in computer systems. Each of these have their own advantages and disadvantages. There is not a perfect memory management system and students need to understand the drawbacks of each method and determine how it can impact the performance of the software they are developing or running.

In Module 8 Program Counter relative addressing was discussed. This addressing mode will allow programs to be anywhere in memory and can run properly because all the address references are made to the current program counter. So if a running process is swapped out of main memory it can be swapped back into any free partition and continue without any issues.

For fixed partitions the OS is allocated a part of memory and the rest is divided into partitions.

There can be equal size partitions. Take how many partitions you want and divide it into how much memory you have left. Or how big do you want the partitions and divide into the amount of memory you have to determine how many partitions you have. This technique is covered in the text.

Problems with equal size partitions:

- Program too big for one partition. Programmer has to manage this to allow for overlays so only a part of the program is in main memory.
- Programs are given the whole partition. So with many small programs the method is inefficient. Internal fragmentation.

What is not covered in the text is unequal sized partitions which may mitigate some of these problems. The Figure below shows how this can be accomplished.

As the text illustrates in Figure 3.3 when processes A and B are placed in a partition there is wasted memory in that partition that is not used by the process and cannot be used by any other processes either. This unused memory is referred to as internal waste or internal fragmentation. You may be thinking what is the big deal if there is some unused memory at the end of these partitions? I have 4GB of main memory what is it to me if a few MB go unused? Well, you paid for all 4GB of main memory you should expect to use all 4GB of memory. Depending on the number and sizes of the partitions and your size of the processes being run a significant amount of main memory may always be unused. Again, I paid for it and I expect to be able to use it.

This memory organization requires either all or none of the process to be in main memory. There is no in-between with this partition method.

Process must fit into a single partition. So, if your program and data are too big to fit in the largest partition it is up to you as the programmer to split your program into several smaller partitions so it can run.

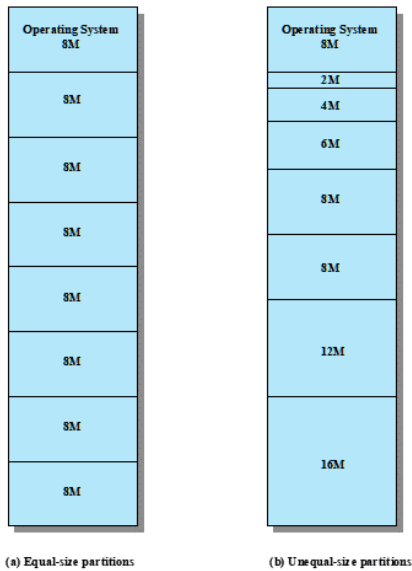


Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

PARTITION SIZES

PLACEMENT ALGORITHM

How do you figure out what goes where?

With equal size partitions this is trivial. Put the process in the first free space. If the loaded processes are not ready to run then pick one to trade out with a new process.

With variable size or unequal sized partitions you can:

1. Assign each process to the smallest partition it will fit in. Need a scheduling queue for each sized partition. This minimizes Internal fragmentation. Leads to a different inefficiency if processes do not fit in a partition size and the partition goes unused.
2. Single queue for all processes – Put in smallest available memory. If none open then use some swapping algorithm to place the process.

Drawbacks for the Unequal-size partitions

1. The number of processes in memory is always fixed.
2. Small jobs will never utilize the memory space efficiently.

Fixed Partitions are almost never used with modern systems.

Single segment or Dynamic Partitioning

Processes are allocated exactly as much memory as it requires.

The text goes through an example on how this works with Process A and B in Figure 3.6. Below is another example and you can quickly see the problem with that approach. It all starts out good but when processes either complete or are swapped out this shortly leads to lots of small holes in memory called external fragmentation.

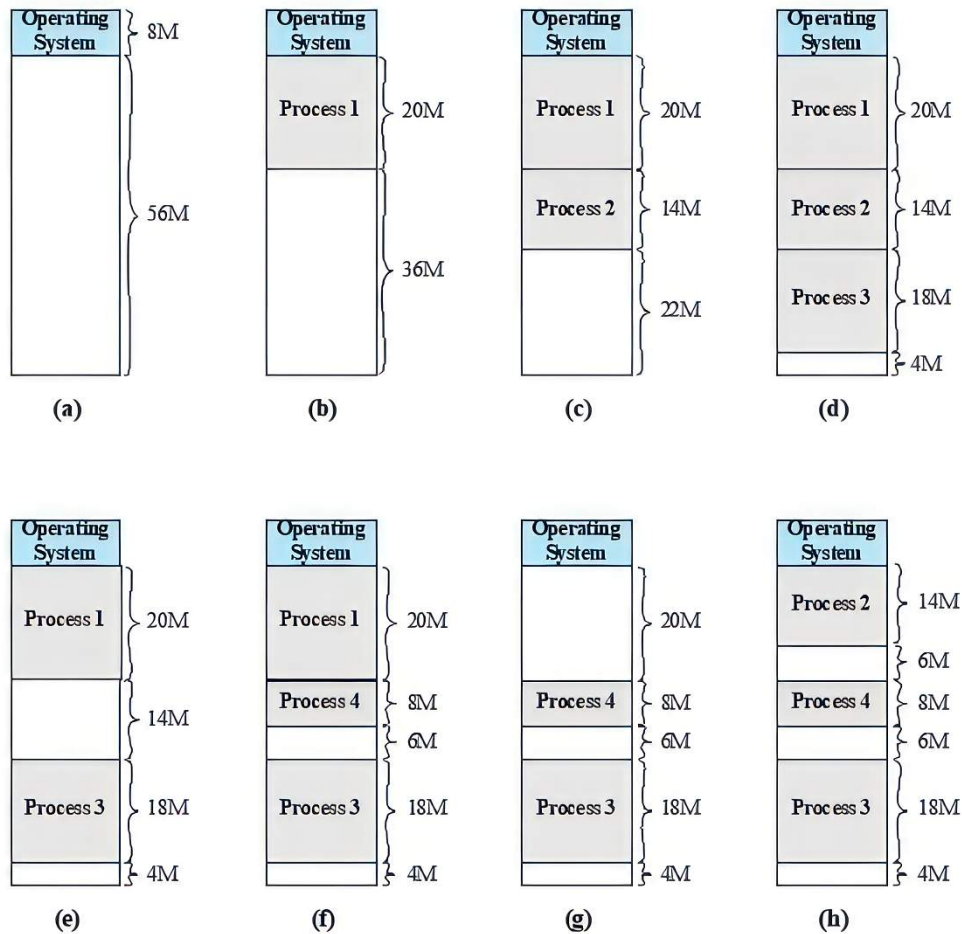


Figure 7.4 The Effect of Dynamic Partitioning

Can this be fixed? Yes via a technique called compaction or shuffling. The OS shifts all the processes to make a large contiguous free memory.

Drawback - takes time to relocate the processes and all the processes have to be able to be relocated.

Placement Algorithm

To help avoid external fragmentation the placement of processes is important.

1. Best-fit - Pick the block of memory that is closest in size to the process request. Takes CPU time to search all the open memory locations. Makes even smaller fragments. Worst of the three.
2. First-fit - Pick the first block of memory that is large enough to meet the request. Simple and fast and best in most applications. Leads to small fragments at the front of memory.
3. Circular first-fit or Next-fit - Starting at the current location pick the next block of memory that is large enough to meet the request. Each memory request wastes the smallest fragment. Simple and fast but leads to small fragments at the end of memory. Need to compact more often.
4. Worst-fit - The largest free block of memory is used for bringing in the process. The logic behind this is to leave larger fragments which may be big enough for other processes. The disadvantage of this approach is that the largest blocks are allocated first; therefore a request for a large area is more likely to fail.

Replacement Policies —

When it comes to swapping out a segment there are some things that need to be taken into consideration.

Pick processes that are currently waiting for some request like input/output. They are not going to run anytime soon so no big deal if they are taken out of memory. It is also best to select a segment that has not been modified (i.e., no memory writes). If there have not been any modifications there is no reason to save the current contents back to secondary storage.

The ones above are the easy selections. After that things can get really complicated. Some options are:

1. Pick the largest segment - This frees up the most memory to allow as more processes to run.
2. Pick the segment that has been allocated the longest - It already has enough time running give some other process a chance.
3. Pick the segment that has been in there the shortest amount of time - It has not been running long so no one will really notice it was swapped out.
4. Pick the lowest priority process assuming the OS has gone to all the trouble to set-up a priority system.

Segmentation

A program can be subdivided into segments

- May vary in length
- There is a maximum length

Addressing consists of two parts:

- Segment number
- An offset from the start of the segment

Segmentation is similar to **dynamic partitioning** because of the ability of having unequal sized segments.

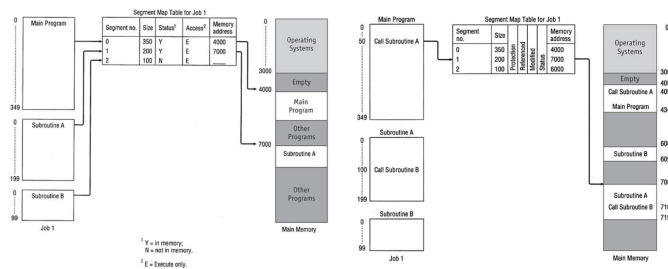
Eliminates **internal fragmentation** but still causes **external fragmentation** in main memory. At some point the OS needs to kick-in and perform **compaction** or **shuffling**.

Segmentation is visible to the programmer and can be controlled by the programmer. There can be a different segment for the program vs the data. Program and/or data can be broken into multiple segments.

Programmer must know the maximum segment size to ensure they do not declare things that turn out to be larger than the largest segment.

The text has an example of how Segmentation works. What is provided here is a slightly different example. To provide another view point to this process.

In this example Job 1 includes a main program and two subroutines. It is a single job that is structurally divided into three segments of different sizes.



The Segment Map Table tracks each segment for this job. Notice that Subroutine B has not yet been loaded into memory.

During execution, the main program calls Subroutine A, which triggers the Segment Map Table (SMT) to look up its location in memory.

There is no simple relationship between the logical address and the physical address like there is with Paging because of the high likelihood of the unequal size segments.

Each segment table entry needs the starting address and the length to ensure references stay within the segment boundary. The following steps are needed for address translation:

1. Extract the segment number as the leftmost n-bits of the logical address.
2. Use the segment number as an index into the process segment table to find the starting physical address of the segment.
3. Compare the offset, expressed in the rightmost m-bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid.
4. The desired physical address is the sum of the starting physical address of the segment plus the offset.

This memory management process allows processes to run and the entire process does not have to be in memory to execute. With the use of the Segment Map table and virtual addressing the process does not have to be in contiguous memory to run. The parts of the process can be located anywhere in main memory and can be moved to different locations without effecting the code. Only the Segment Map Table needs to be updated.

When a memory reference is made the physical address is looked up in the Segment Map Table. If it is not present in main memory a segment fault (which is also referred to as a miss) is triggered causing the OS to halt the current process and start the processes of swapping the referenced location back into main memory. If the address is in main memory this is referred to as a hit.

There are the same types of problems with Segmentation when it comes to selecting the segment to swap out and placement of the new segment.

Paging

Think of it like a picture frame vs the picture or window frame vs the window pane (the glass). Pages can be moved around to different frames. Frames are fixed locations in main memory which is typically done as part of the OS design.

Main memory is divided into "small" equal size "chunks" called frames or page frames. Process are also divided into the same size chunks and these are called pages.

Shuffling or Compaction is not necessary. There may be some internal frame fragments but because the frames are relatively small as compared to the previous methods it is not enough for the OS to worry about.

As in the previous sections of the text the sizes for virtual address bits is small compared to systems of today. The math related to the number of pages, size of pages (offset) is still the same but with larger numbers.

There is no external fragments because all of main memory is allocated to a frame.

The first big question is how large do you want to make your frame? These are the things that have to be considered because it will affect the design and performance of the OS.

1. Memory overhead to track the pages. More pages you have the more memory is required to track the pages.
2. Internal waste - Small page sizes will have less internal waste but problems with #1 above. Large pages helps #1 but more internal fragmentation or waste.

3. Cold start faults - aka mandatory page faults because the newly started process has no pages loaded into main memory.
4. Page fault rate - The more memory is given to the running process the lower the page fault rate aka miss. This could be due to allocation of more smaller pages or larger pages.
5. Transfer efficiency - When things have to be swapped how long does that take? There is a fixed overhead when doing I/O then a "price by bit". Larger pages reduces the overall rate per bit.

In general, large pages are better overall but they get too big then only a few processes can be running at the same time. This in turn make for inefficiencies that we have already touched upon.

Implementation

The page table which does the address translation contains the hardware which translates the virtual address to the page number and the offset for that specific address reference. To speed this process up a piece of hardware called translation look-aside buffer is used to look up these addresses to see if there is a page hit or page miss.

Page-replacement policies

The efficiency of these memory management systems are directly affected by the page replacement policy chosen by the OS. Some of the issues with replacement for non-paging systems (partitions, segments) are the same as with paging systems but paging has some unique issues.

1

Random - This is not a serious algorithm but can be used to check the algorithm against no real policy. If your algorithms performs worse than random then you should give up on the algorithm.

2

Optimal text refers to this as Min - replace the page that will have the longest time to its next reference. Needs a crystal ball in order to implement. Cannot do this in the real world but it is what is

used to set the standard and measure other replacement algorithms.

3

Least recently used - (LRU) the page that has not been referenced in the longest time. Need a method to time them which is more hardware or overhead to track this. Hard to implement but works well.

4

Clock text refers to this as Not used recently (NUR) - Use bit set to 1 when the page is loaded and anytime it is referenced. When does the bit get set to 0? The OS scans the buffer periodically and sets all the 1's it encounters with a 0. When it comes time to replace a page it finds the first 0 frame and replaces it. Like FIFO but with a second chance for items recently referenced.

Global policies

The following sections describe some policies the OS can apply to all running processes to improve the overall performance of the virtual memory system. Collectively these are referred to as Global Policies.

Working Set - This is the set of pages that the program needs to run efficiently. This is different from the Resident set (current pages loaded into memory) in that the working set may be smaller or larger than the resident set. Like the resident set the working set can change over time and the program and data references change over the execution of the program.

If the number of pages allocated to a process is smaller than the working set then a lot of page faults will happen along with the associated swapping of pages. If the number of pages is larger than the working set then this does not allow for other process to run. This not only effect the process but the overall system performance as well.

One method to help determine the Working Set is to check the page-fault frequency of the process. If the page faults are too high then the OS can allocate more pages to the process to reduce the frequency of the page-faults. There is also a negative of this approach in that if the

page-faults of a process is too low then the OS will take away pages from the process to raise the page-faults to an acceptable level.

Classification of replacement methods

When is a page swapped in?

One obvious answer is when it is called for and only then. (Demand paging).

Another option is prefetching. With this method an educated guess is made or the programmer provides a hint to the OS on which page needs to be brought in (advised paging).

Clustering is when page n is brought in that pages $n+1$, $n+2$, etc. are also loaded.

When should a modified page be written back out to secondary storage? Dirty is when main memory does not match what is on secondary storage. When both match it is clean.

Which page should be swapped out?

Selecting the page to swap out can be applied locally to the pages allocated to the process or is can be applied globally by selecting from all the current pages which one to swap out.

One thing not covered in the text is when should pages be written back to secondary storage. If the page has been written to it has to be saved prior to it being swapped out. Here are some options:

Demand Cleaning - a page is written out to secondary storage only when it has been selected for replacement. This means there is a write out followed by a read in to replace the page.

Precleaning - Allows the writing of page in batches and writes pages that may not be swapped out anytime soon. Does work that does not need to happen right now.

Use page buffering - Clean only pages that are replaceable but decouple the cleaning and replacement operations.

Startup-phase conflicts can be a big problem if lots of processes are started at the same time.

OS Resource Deadlock



Introduction

The text lays out how deadlock and starvation are serious issues with computers. My lecture will cover the most important concepts from the text.

An important concept to remember from the previous section is that the state of a running process can be saved. This is part of the swapping out process. Once saved the process can be brought back into main memory and resume running like nothing ever happened to it. This technique can be used by the OS if there is an error condition to restore the process back to a known good state and allow it to try to execute again. When the saved process is restored this is called a rollback. Many rollback points can be saved at various execution points. If the OS suspects a process may either have an execution error or be part of a deadlock condition the last known good state can be restored and the process is allowed to try again.

It does not matter if a resource is a reusable resource or consumable resource deadlock and starvation can still be a problem. A reusable resource is one that can be safely used by only one process at a time and is not depleted by that use. Processes obtain resource units that they later release for reuse by other processes.

Examples of reusable resources include processors; I/O channels; main and secondary memory; devices; and data structures such as files, databases, and semaphores. A consumable resource is one that can be created (produced) and destroyed (consumed). Examples of consumable resources are interrupts, signals, messages, and information in I/O buffers.

Deadlock is a result of our own creation. Because we think it is a good idea for the following first three conditions to exist in a computer system. An actual deadlock happens when the

forth thing in this list happens.

1. Mutual exclusion or non-shareable . Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.
2. Hold and wait. A process may hold allocated resources while awaiting assignment of other resources.
3. No preemption. No resource can be forcibly removed from a process holding it.
4. Circular wait. A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Nuts and bolts

Processes and use of resources go through a straight forward process that is controlled by the OS.

1. Ask for a resource
2. Wait for the resource
3. Get the resource
4. Use the resource
5. Return the resource

There are a couple of things that can go wrong in these steps that the OS may have to deal with.

1. What happens if the process is given the resource, abnormally terminates and does not release the resource? The OS is going to have to be able to detect this and put the resource back in a state that can be used by other processes.

2. The process releases the resource, terminates but leave the resource in a state that makes it unusable by other processes? The OS has to detect this and again put the resource back in a known good condition.

Deadlock and Starvation

The text shows one of the simplest examples of deadlock with two processes and two resources. Because of Hold and Wait, Mutual exclusion and no preemption a circular wait has happened and there is a deadlock condition.

Deadlock - *permanent* blocking of a set of processes that either compete for system resources or communicate with each other. Each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set.

Deadlock is permanent because none of the events is ever triggered. Unlike other problems in concurrent process management, there is no efficient solution in the general case to prevent deadlock.

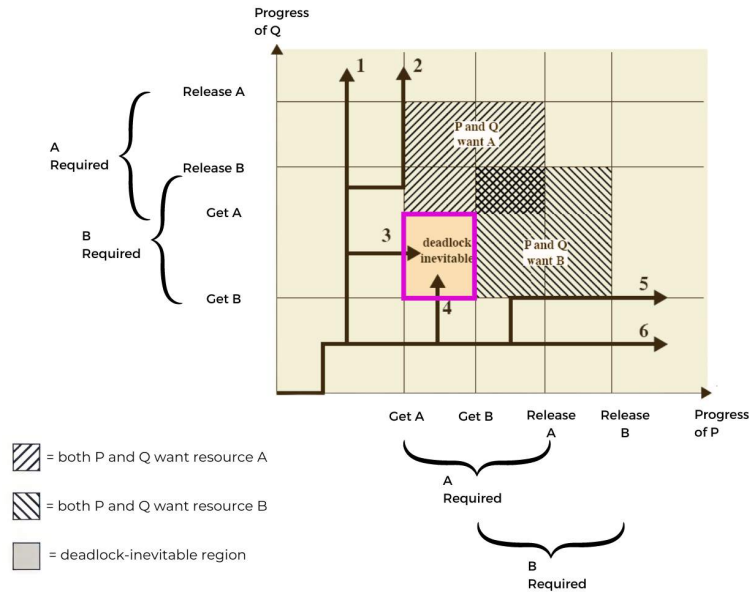
A **progress diagram** or **Joint progress diagram** (See below) shows the progress of processes competing for resources. The problem with these diagrams is they get really complicated if there are more than two processes after the same resources. The diagram would require n-dimensions for n-processes and are not practical to use by the OS.

The serialization approach described in the text basically comes down to once a process is granted a resource no other processes are given resources until the first process releases all the resources. Even if resources are available to grant to the other processes none are given until all resources are released.

I will give the serialization approach its due - The circular wait condition will NEVER happen so deadlock will never happen. There is another name for this approach - Stupid. This is extremely inefficient and is not used in most modern Operating Systems. I would say it is never used but there are some real-time Operating Systems where the serialization approach is absolutely necessary.

Another issue to take into account is starvation – A runnable process is overlooked indefinitely by the OS although it can run it is never chosen to run.

Joint Progress Diagram



Dining philosophers

The Dining Philosophers problem is just one of dozens and dozens of easy to state problems that have possible deadlock and/or starvation issues if not handled properly. Each of these types of problems has direct applicability to modern Operating Systems when it comes to resource allocation.

There are several forms of the Dining Philosophers problem that vary slightly in the set-up and rules of the house, dining room, dining table and/or eating instruments. They still all can result in Deadlock and/or starvation if no additional rules are applied to either the house, dining room, seating or obtaining the two required eating instruments (chopsticks, forks, spoons, etc.).

Just like in the text a simple, elegant and wrong solution is provided to "fix" the problem for all the variations of this problem. Then a correct solution is provided that ensures there is no deadlock or

starvation can take place.

One-shot allocation

One solution to the Dining Philosophers problems is when the philosopher sits down at the table they must request both the left and right eating instruments at the same time. If they cannot get both of them they do not get either of them. Deadlock does not happen but starvation can still occur.

This is referred to as a one-shot allocation.

If this is applied to processes then a process is not allowed any resources until all the resources they need to run are currently free. There are several issues with this approach.

1. The process is forced to ask for a resource it may never need.
2. Processes that only need a single resource will be able to run
3. Processes that require several resources to run are subject to starvation.

Hierarchical allocation

The process works by providing a linear ordering of all the resources by assigning them increasing numbers. If a process requests an additional resource it is only granted if the new request is a higher (or lower) number than the resources the process currently holds.

One of the advantages of this process is that it is quick to check but it still slows down the system by denying resource allocations that would not lead to deadlock.

Advance-claim algorithm

Do not grant an incremental resource request to a process if this allocation might lead to a deadlock.

This is called the banker's algorithm and is clearly described in the text.

For the banker's algorithm to work the state of the system current allocation of resources and processes must be known and updated.

Two vectors (one dimensional arrays) are needed:

Resource - List of all the resources. $R = (r_1, r_2, r_3, \dots R_m)$

Available - The amount of each resources (currently not in use by any process)

$V = (v_1, v_2, v_3, \dots V_m)$

Two two-dimensional arrays are needed:

Claim - Process i has a claim for resource j .

$C = c_{11}, c_{12} \dots c_{1m}$

.

;

$c_{n1} c_{n2} \dots c_{nm}$

Allocation - process i has been allocated resource j .

$A = a_{11} a_{12} \dots a_{1m}$

;

;

$a_{n1} a_{n2} \dots a_{nm}$

In order for this to work all the resources that a process will need has to be known in advance.

With all this known a new process will only be started if its resource requirements will not lead to a possible deadlock. Do not even start a process if its demands might lead to a deadlock.

In English the formula means the process is only started if the maximum claim of all current processes plus those of the new process can be met.

A safe state is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock (i.e., all of the processes can be run to completion).

An unsafe state is a state that is not safe.

Drawback - assumes the worst case all the time. Pushes the problem back upstream with allowing or not allowing processes to start execution.

The exact details of the Banker's algorithm is not that important for this course.

Deadlock detection

How does the OS detect a deadlock condition? It uses the banker's algorithm to find the circular wait condition.

The exact details on this algorithm is not that important for this course.

Deadlock recovery

We have detected a deadlock condition. What can be done to fix it?

1. Kill them all and let God sort them out. Abort all the deadlock processes. Most common. Quick and easy.
2. Rollback the deadlock processes and try again. No guarantee it will not happen again. Also assumes there is a rollback status saved by the OS. Rollback is when the Process Control Block was saved as a "Known good working condition".
3. Kill one deadlock process at a time until the deadlock no longer exists.

Preempt resources (take them away from a process and give it to another process) until the deadlock is resolved. Rollback the process that the resource was taken away from.

Starvation

The same type of algorithm used for deadlock detection can be used to detect processes that are suffering from starvation. How to fix the problem is to provide the resources to the starved process. This can be accomplished by not allowing any new processes into the system until the starved process completes or at least gets some runtime.

Perspective

The Three/Four necessary Conditions for Deadlock —

Deadlock is a result of our own creation. Because we think it is a good idea for the following first three conditions to exist in a computer system. An actual deadlock happens when the forth thing in this list

happens.

1. Mutual exclusion . Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.
2. Hold and wait. A process may hold allocated resources while awaiting assignment of other resources.
3. No preemption. No resource can be forcibly removed from a process holding it.
4. Circular wait. A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Deadlock prevention are the methods used to preclude deadlocks from occurring.

- The indirect methods is to prevent the necessary condition for deadlocks like mutual exclusion, Hold and wait and no preemption.
- The direct method is to prevent circular wait conditions.

Deadlock Avoidance Advantages —

- No need to stop a running process and roll it back.
- Less restrictive than deadlock prevention. Because with deadlock prevention the worst case has to be presumed.

Deadlock Avoidance Disadvantages —

- The maximum resource requirement for each process must be stated in advance.
- The processes under consideration must be independent; that is, the order in which they execute must be unconstrained by any synchronization requirements.
- There must be a fixed number of resources to allocate.
- No process may exit while holding resources.

Deadlock Detection —

Three approaches to prevent deadlock and will be covered later in this course.

1. Deadlock prevention: Disallow one of the three necessary conditions for deadlock occurrence (cover these in a minute), or prevent circular wait condition from happening.
2. Deadlock avoidance: Do not grant a resource request if this allocation might lead to deadlock.

Deadlock detection: Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover.

OS Concurrency



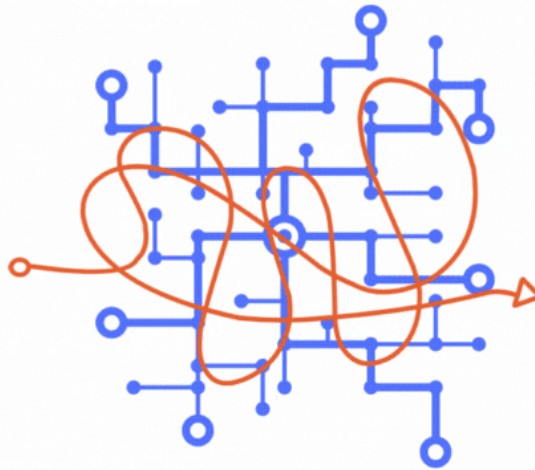
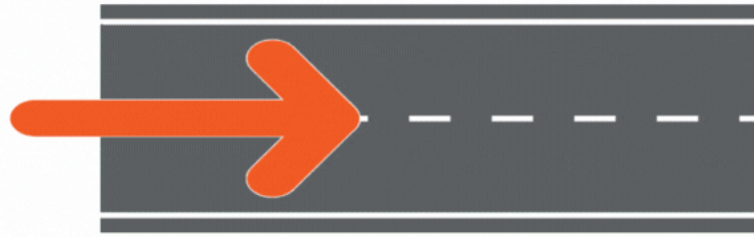
Introduction

The previous module covered some very serious issues with OS software. Deadlock, starvation, spin waiting (trashing) and livelock continue to be issues that computer science and software engineers to this day have to deal with. With the modern multiple cores, parallel processing and the like they will continue to be issues.

One of the key principles mentioned in the previous module was mutual exclusion. Making sure that only one process had exclusive access to resources. This module will go deeper into this topic to explain the scope of this issue and look at potential solutions.

The problems

One of the problems with analyzing these issues is we humans think linear. Instruction 1 executes then instruction 2, 3 etc. What can really happen is show in the text with Process A and Process B. An unexpected out-of-order execution will produce unexpected or wrong results.



MUTUAL EXCLUSION

SYNCHRONIZATION

Ensuring **mutual exclusion** helps prevent these issues but this is easier said than done. The text has the `BeginRegion` and `EngRegion` syntax that identifies the section of code that should not be interrupted. This is sometimes called the critical section of code.

MUTUAL EXCLUSION

SYNCHRONIZATION

Synchronization is a generalization of mutual exclusion. Since synchronization is a generalization of mutual exclusion, we can expect that the techniques developed for solving the mutual-exclusion problem are also useful for achieving synchronization

Mechanisms

There are over a dozen synchronization mechanisms that the text discusses. Keep in mind that for this course it is not necessary to memorize the details of each of these methods. Students are expected to identify these mechanisms and explain how they work to ensure synchronization. Several of these depend upon the computer hardware for interrupts and atomic operations.

Disable interrupts —

Disabling interrupts work very well on a single CPU with only one core. The currently executing process cannot be stopped as long as the interrupts are disabled.

This process falls apart in a multiprocessor system where disabling interrupt only works on a single CPU or core.

Busy wait on Mutex —

- Programming flag used to grab and release an object.
- Lock or True - data that cannot be shared or processes that cannot be performed simultaneously elsewhere.

- Unlock or False- data is no longer needed or routine is finished.

The text shows in the sample code how difficult it can be to ensure mutual exclusion using a software solution.

Switch variable —

Using a shared variable or a single memory location

Issues-

1. Process switch back and forth. A then B then A again etc. A then A again it not allowed. If A wants to execute once an hour and B want to execute 1000 times an hour then B has to adopts the rate of A.
2. If one process hangs up or dies the other process is permanently blocked.

An important thing to remember about Mutual Exclusion- Getting control or access is usually not a problem. The biggest problem is giving control back when the process is done with it or cannot use it anymore.

Non-alternating switch —

View the following video on Dekker's Algorithm:

[Race Conditions and How to Prevent Them - A Look at Dekker's Algorithm](#)

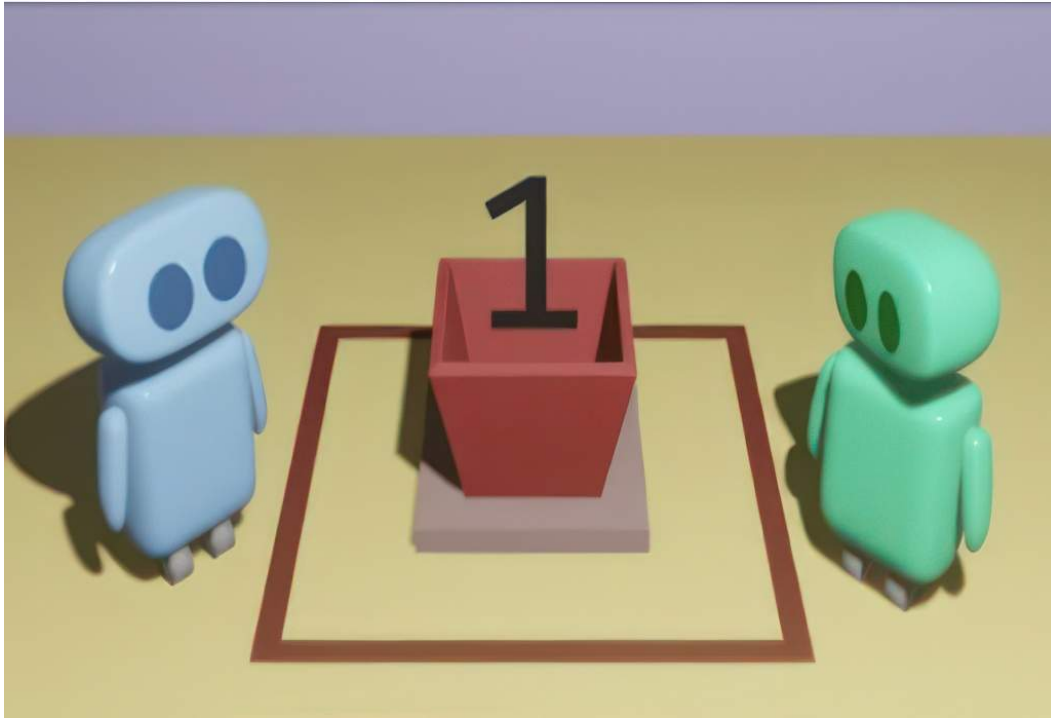
Dekker's algorithm fixes the issues with the previous attempts at mutual exclusion:

1. Forcing competing processes to run at the same intervals
2. Deadlock
3. Starvation
4. Not ensuring mutual exclusion in the first place.

Stepping through the code you can see how Dekker's works to address all these issues but it is complex and "proving" it actually works is very difficult.

Peterson's Algorithm on the other hand provides a simpler approach which is easier to prove.

The text covers a lot of complicated code. It is not necessary for this course for students to understand all the details for each of the algorithms discussed so far. The important things to know is mutual exclusion is difficult to get right.



Locks —

With the Test and Set approach the atomic principle of code execution comes into play. The reading of the Lock and the changing of the Lock value cannot be interrupted in the middle of reading or writing Lock.

This method does present some problems with busy-waiting and starvation.

Semaphores —

Semaphores is from the Greek it means Sign-bearer. Prior to computer use the Navy used flags to signal others.

Semaphore is a special name for variables to control concurrency.

1. A semaphore may be initialized to a nonnegative integer value. This integer provides the number of those types of resources are available.
2. The semWait operation decrements the semaphore value – I want something. If it is available give me the resource. If it is not available put me on the waiting list for the resource.
3. The semSignal operation increments the semaphore value – I'm done with the resource and if anyone is waiting for this resource give it to them.

Semaphore or counting semaphore or general semaphore.

If the semaphore value becomes negative or zero then the process making the call is blocked.

If the semaphore value is positive then the process can continue.

Semaphore = 0 ; the next process to issue a wait is blocked and Semaphore = -1

Semaphore = 8 ;The number of processes that can issue a wait and immediately continue to execute.

Semaphore = -3; number of processes waiting to be unblocked.

Example on how a counting semaphore works:

5 cars and 80 people waiting to get in a car. Each person spends some random time in the car then gets out. Because there are 5 cars the semaphore value is set to 5.

- Assume all 80 people ask to get in the car at the same time. The first five semWait(Cars) will decrement Cars and will allocate a Car to a person because there are still Cars that have not been allocated. So after 5 cars are allocated the variable Cars = 0.
- The 6th semWait(Cars) will decrement Cars (which is now = -1). Because the semaphore is negative this person cannot get into a car so they are along with the other 74 people are blocked.
- Person is done and issues a semSignal(Cars). Cars is incremented to -74 and a person is selected to unblock and allow on the just freed up car.

Things we have to deal with related to **Semaphores**:

1. No way to know if a process will be blocked or not until the semWait is called and it goes negative.
2. After a process increments a semaphore and another process gets woken up, both processes continue running concurrently. There is no way to know which process, if either, will continue immediately on a **uniprocessor** system.
3. When you signal a **semaphore**, you don't necessarily know whether another process is waiting, so the number of unblocked processes may be zero or one.

Binary Semaphore – only takes on the values of 0 or 1.

1. A binary semaphore may be initialized to 0 or 1.
2. The semWaitB operation checks the semaphore value. If the value is zero, then the process executing the semWaitB is blocked. If the value is one, then the value is changed to zero and the process continues execution.
3. The semSignalB operation checks to see if any processes are blocked on this semaphore (semaphore value equals 0). If so, then a process blocked by a semWaitB operation is unblocked. If no processes are blocked, then the value of the semaphore is set to one.

Critical regions —

As with any current processes and shared resources the following three problems need to be addressed:

- (1) Failure to protect shared variables
- (2) Failure to invoke semaphore operations in the right order and at the right time
- (3) Deadlock and starvation

The critical region or critical section is that part of code that requires access to a shared resource that must never be accessed while another process is running that same section of code. If critical regions of code do get interrupted it is highly likely that incorrect results could result.

Monitors —

Monitors are a programming language construct vs a semaphore that the programmer has to implement within their code.

The chief characteristics of a **monitor** are the following:

1. The local data variables (**condition variables**) are accessible only by the monitor's procedures and not by any external procedure.
2. A process enters the monitor by invoking one of its procedures.
3. Only one process may be executing in the monitor at a time; any other processes that have invoked the monitor are blocked, waiting for the monitor to become available. (**Mutual exclusion**)

Synchronization (In Java) via the **condition variables**. Only operated on by two functions:

- cwait(c) – suspend execution

- `csignal(c)` - resume execution

If a process in a monitor signals and no task is waiting on the condition variable, the signal is lost.

Producer Consumer Problem

The problem statement:

- One or more producers are generating data and placing these a buffer.
- Single consumer is taking items out of the buffer one at a time
- Only one producer or consumer may access the buffer any one time.

What is the problem?

- Make sure the producer does not write to a full buffer AND the consumer does not try to read from an empty buffer.

The text provide a solution of the Producer Consumer problem using the Monitor construct.

Readers-Writers Problem

There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers). The conditions that must be satisfied are as follows:

1. **Any number of readers may simultaneously read the file.**
2. **Only one writer at a time may write to the file.**
3. **If a writer is writing to the file, no reader may read it.**

Thus, readers are processes that are not required to exclude one another and writers are processes that are required to exclude all other processes, readers and writers alike.

How is this different from mutual exclusion and producers/consumers problem?

- Readers do not write data
- Writers do not read data
- Producers has to read to find where in the queue to write
- Consumer has to write to adjust the queue pointers

Perspective

To summarize this chapter:

1. Concurrency is highly desired in computer operations. To increase overall processing speeds having multiple CPUs or cores working together is essential.
2. Sharing the same resources with all these CPUs or cores presents some difficult issues that are not easy to overcome. These include:
 - Mutual exclusion
 - Spin waiting
 - Starvation
 - Deadlock
3. The algorithms and methods used to try and obtain sharing without these issues are complex and difficult to prove correct. The problems presented in this text written over 40 years ago still plague us today. Even with the more modern programming languages like Java that support concurrency it is still highly dependent on the programmer to code it right.