

UNIT-2

ELEMENTS OF IOT

INTRODUCTION:

In the last **unit we have discuss the architecture of iot**. Today, we will study how IoT works or How the Internet of Things works. Moreover, we will the IoT components. In addition, we will see some real-time example of how IoT works.

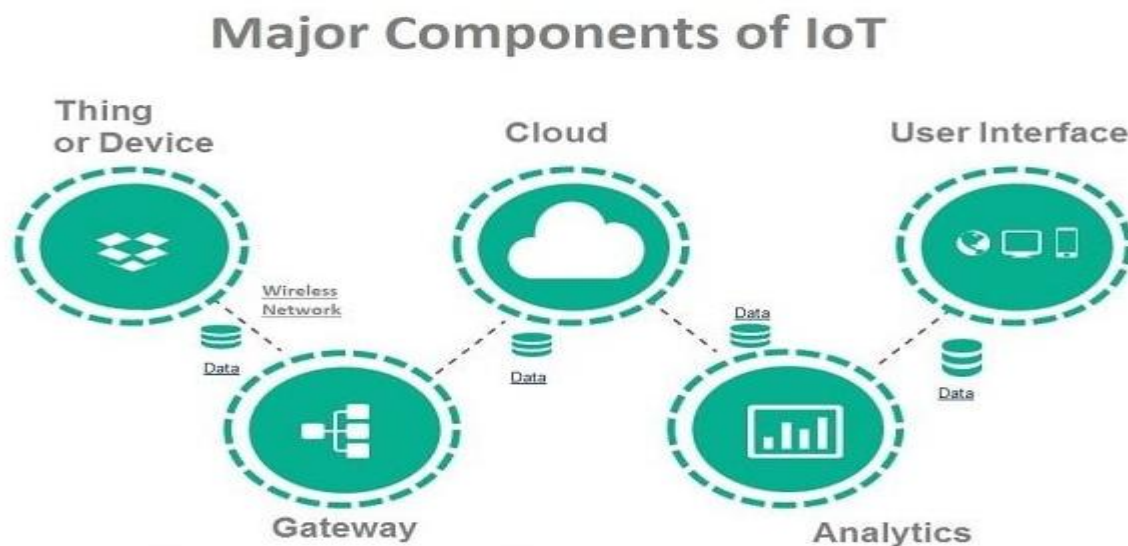
How IoT Works?

Just like Internet has changed the way we work & communicate with each other, by connecting us through the World Wide Web (internet), IoT also aims to take this connectivity to another level by connecting multiple devices at a time to the internet thereby facilitating *man to machine* and *machine to machine* interactions.

People who came up with this idea, have also realized that this IoT ecosystem is not limited to a particular field but has business applications in areas of home automation, vehicle automation, factory line automation, medical, retail, healthcare and more.

1.Components of IOT:

IoT is a transformation process of connecting our smart devices and objects to network to perform efficiently and access remotely. There are 5 fundamental components of IoT system.



i. Sensors/Devices

sensors or devices are continuously collecting very minute data from the surrounding environments & transmit the information to the next layer. All of this collected data can have various degrees of complexities ranging from a simple temperature monitoring sensor or a complex full video feed.

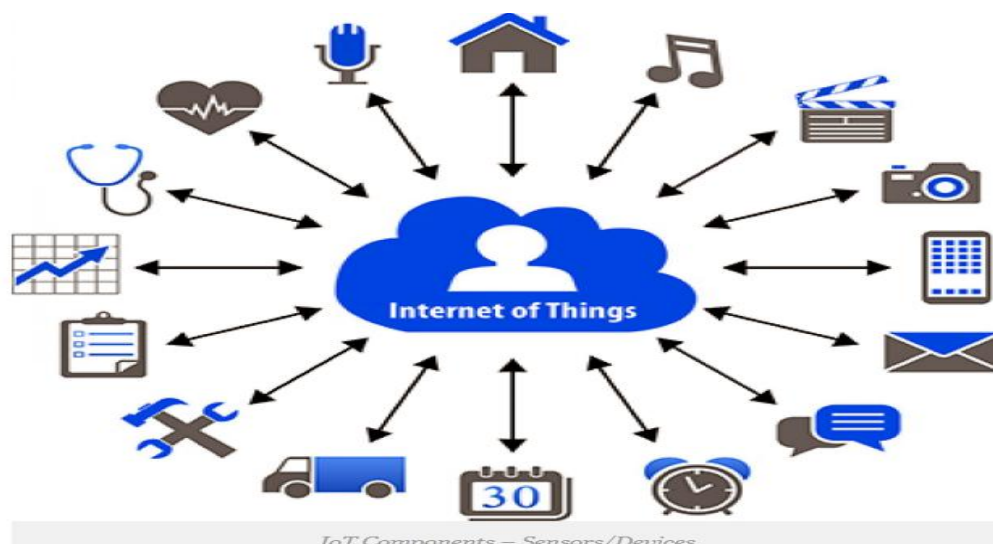
A device can have multiple sensors that can bundle together to do more than just sense things. For example, our phone is a device that has multiple sensors such as GPS, accelerometer, camera but our phone does not simply sense things.

The most rudimentary step will always remain to pick and collect data from the surrounding environment be it a standalone sensor or multiple devices.

Latest techniques in the semiconductor technology is capable of producing micro smart sensors for various applications.

Common sensors are:

- Temperature sensors and thermostats
- Pressure sensors
- Humidity / Moisture level
- Light intensity detectors
- Moisture sensors
- Proximity detection
- RFID tags



How the devices are connected? ii. Connectivity:

Next, that collected data is sent to a cloud infrastructure but it needs a medium for transport.

Most of the modern smart devices and sensors can be connected to the cloud through various mediums of communication and transports such as low power wireless networks like Wi-Fi, ZigBee, Bluetooth, Z-wave, cellular networks, LoRAWAN etc... Each of these wireless technologies has its own pros and cons in terms of power, data transfer rate and overall efficiency.



Every option we choose has some specifications and trade-offs between power consumption, range, and bandwidth. So, choosing the best connectivity option in the IOT system is important.

6LoWPAN uses reduced transmission time (typically short time pulses) and thus saves energy.

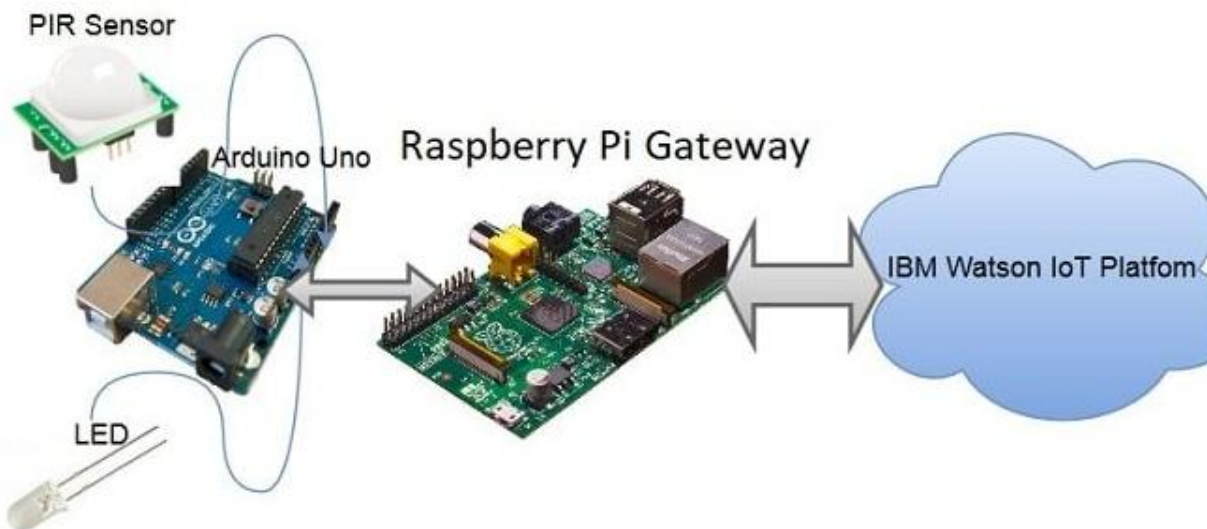


2. Gateway:

IoT Gateway manages the bidirectional data traffic between different networks and protocols. Another function of gateway is to translate different network protocols and make sure interoperability of the connected devices and sensors.

Gateways can be configured to perform pre-processing of the collected data from thousands of sensors locally before transmitting it to the next stage. In some scenarios, it would be necessary due to compatibility of TCP/IP protocol.

IoT gateway offers certain level of security for the network and transmitted data with higher order encryption techniques. It acts as a middle layer between devices and cloud to protect the system from malicious attacks and unauthorized access.



3. Cloud

Internet of things creates massive data from devices, applications and users which has to be managed in an efficient way. IoT cloud offers tools to collect, process, manage and store huge amount of data in real time. Industries and services can easily access these data remotely and make critical decisions when necessary.

Basically, IoT cloud is a sophisticated high performance network of servers optimized to perform high speed data processing of billions of devices, traffic management and deliver accurate

analytics. Distributed database management systems are one of the most important components of IoT cloud.

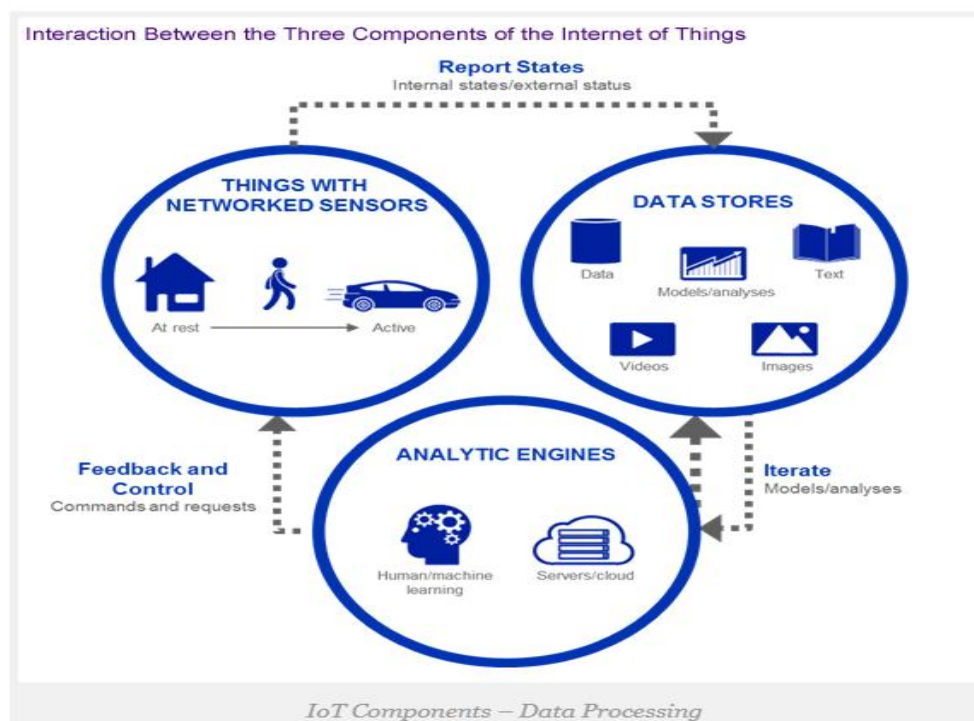
Cloud system integrates billions of devices, sensors, gateways, protocols, data storage and provides predictive analytics. Companies use these analytics data for improvement of products and services, preventive measures for certain steps and build their new business model accurately.

iii. Data Processing

Once the data is collected and it gets to the cloud, the software performs processing on the acquired data.

This can range from something very simple, such as checking that the temperature reading on devices such as AC or heaters is within an acceptable range. It can sometimes also be very complex, such as identifying objects (such as intruders in your house) using computer vision on video.

But there might be a situation when a user interaction is required, example- what if when the temperature is too high or if there *is* an intruder in your house? That's where the user comes into the picture.



4. Analytics:

Analytics is the process of converting analog data from billions of smart devices and sensors into useful insights which can be interpreted and used for detailed analysis. Smart analytics solutions are inevitable for IoT system for management and improvement of the entire system.

One of the major advantages of an efficient IoT system is real time smart analytics which helps engineers to find out irregularities in the collected data and act fast to prevent an undesired scenario. Service providers can prepare for further steps if the information is collected accurately at the right time.

Big enterprises use the massive data collected from IoT devices and utilize the insights for their future business opportunities. Careful analysis will help organizations to predict trends in the market and plan ahead for a successful implementation.

Information is very significant in any business model and predictive analysis ensures success in concerned area of business line.



5. User Interface:

User interfaces are the visible, tangible part of the IoT system which can be accessible by users. Next, the information made available to the end-user in some way. This can achieve by triggering alarms on their phones or notifying through texts or emails.

Also, a user sometimes might also have an interface through which they can actively check in on their IOT system. For example, a user has a camera installed in his house, he might want to check the video recordings and all the feeds through a web server.

However, it's not always this easy and a one-way street. Depending on the IoT application and complexity of the system, the user may also be able to perform an action that may backfire and affect the system.

For example, if a user detects some changes in the refrigerator, the user can remotely adjust the temperature via their phone.

There are also cases where some actions perform automatically. By establishing and implementing some predefined rules, the entire IOT system can adjust the settings automatically and no human has to be physically present.

Also in case if any intruders are sensed, the system can generate an alert not only to the owner of the house but to the concerned authorities.



Real Life Example Depicting Working of IoT:

Here, we will discuss some examples, which states how IoT works in real-life:

i. Say, we have an AC in our room, now the temperature sensor installed in it in the room will be integrated with a gateway. A gateway's purpose is to help connect the temperature sensor (inside the AC) to the Internet by making use of a cloud infrastructure.

- ii. A Cloud/server infrastructure has detailed records about each and every device connected to it such as device id, a status of the device, what time was the device last accessed, number of times the device has been accessed and much more.
- iii. A connection with the cloud then implement by making use of web services such as RESTful.
- iv. We in this system, act as end-users and through the mobile app interact with Cloud (and in turn devices installed in our homes). A request will send to the cloud infrastructure with the authentication of the device and device information. It requires authentication of the device to ensure cybersecurity.
- v. Once the cloud has authenticated the device, it sends a request to the appropriate sensor network using gateways.
- vi. After receiving the request, the temperature sensor inside the AC will read the current temperature in the room and will send the response back to the cloud.
- vii. Cloud infrastructure will then identify the particular user who has requested the data and will then push the requested data to the app. So, a user will get the current information about the temperature in the room, pick up through AC's temperature sensors directly on his screen.

IoT Hardware:

IoT Hardware includes a wide range of devices such as devices for routing, bridges, sensors etc. These IoT devices manage key tasks and functions such as system activation, security, action specifications, communication, and detection of support-specific goals and actions.

IoT Hardware components can vary from low-power boards; single-board processors like the **Arduino Uno** which are basically smaller boards that are plugged into mainboards to improve and increase its functionality by bringing out specific functions or features (such as GPS, light and heat sensors, or interactive displays).

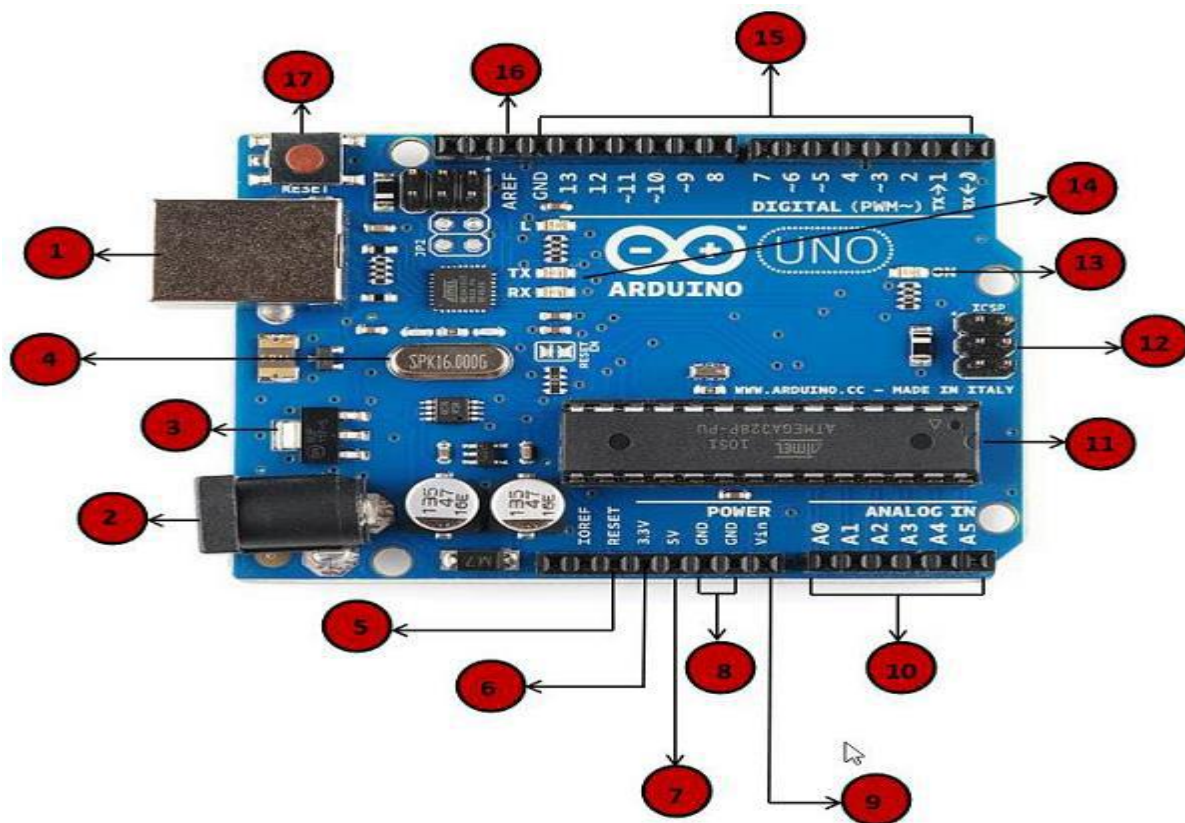
A programmer specifies a board's input and output, then creates a circuit design to illustrate the interaction of these inputs and outputs.

Arduino UNO :

Various kinds of Arduino boards are available depending on different microcontrollers used. However, all Arduino boards have one thing in common: they are programed through the Arduino IDE.







The differences are based on the number of inputs and outputs (the number of sensors, LEDs, and buttons you can use on a single board), speed, operating voltage, form factor etc. Some







boards are designed to be embedded and have no programming interface (hardware), which you would need to buy separately. Some can run directly from a 3.7V battery, others need at least 5V. **Arduino Uno** is a popular microcontroller development board based on 8-bit [ATmega328P](#) microcontroller. Along with ATmega328P MCU IC, it consists other components such as crystal oscillator, serial communication, voltage regulator, etc. to support the microcontroller. Arduino Uno has 14 digital input/output pins (out of which 6 can be used as PWM outputs), 6 analog input pins, a USB connection, A Power barrel jack, an ICSP header and a reset button.



There are three different memories available in ATmega328P. They are:

- 32 KB of Flash Memory
- 2 KB of SRAM
- 1 KB of EEPROM
- 0.5 KB of the Flash Memory is used by the bootloader code.

	<p>Power USB</p> <p>Arduino board can be powered by using the USB cable from your computer. All you need to do is connect the USB cable to the USB connection (1).</p>
	<p>Power (Barrel Jack)</p> <p>Arduino boards can be powered directly from the AC mains power supply by connecting it to the Barrel Jack (2).</p>
	<p>Voltage Regulator</p> <p>The function of the voltage regulator is to control the voltage given to the Arduino board and stabilize the DC voltages used by the processor and other elements.</p>
	<p>Crystal Oscillator</p> <p>The crystal oscillator helps Arduino in dealing with time issues. How does Arduino calculate time? The answer is, by using the crystal oscillator. The number printed on top of the Arduino crystal is 16.000H9H. It tells us that the frequency is 16,000,000 Hertz or 16 MHz.</p>
	<p>Arduino Reset</p> <p>You can reset your Arduino board, i.e., start your program from the beginning. You can reset the UNO board in two ways. First, by using the reset button (17) on the board. Second, you can connect an external reset button to the Arduino pin labelled RESET (5).</p>
	<p>Pins (3.3, 5, GND, Vin)</p> <ul style="list-style-type: none"> • 3.3V (6) – Supply 3.3 output volt • 5V (7) – Supply 5 output volt • Most of the components used with Arduino board works fine with 3.3 volt and 5 volt. • GND (8)(Ground) – There are several GND pins on the Arduino, any of which can be used to ground your circuit. • Vin (9) – This pin also can be used to power the Arduino board from an external power source, like AC mains power supply.

	<p>Analog pins</p> <p>The Arduino UNO board has six analog input pins A0 through A5. These pins can read the signal from an analog sensor like the humidity sensor or temperature sensor and convert it into a digital value that can be read by the microprocessor.</p>
	<p>Main microcontroller</p> <p>Each Arduino board has its own microcontroller (11). You can assume it as the brain of your board. The main IC (integrated circuit) on the Arduino is slightly different from board to board. The microcontrollers are usually of the ATMEL Company. You must know what IC your board has before loading up a new program from the Arduino IDE. This information is available on the top of the IC. For more details about the IC construction and functions, you can refer to the data sheet.</p>
	<p>ICSP pin</p> <p>Mostly, ICSP (12) is an AVR, a tiny programming header for the Arduino consisting of MOSI, MISO, SCK, RESET, VCC, and GND. It is often referred to as an SPI (Serial Peripheral Interface), which could be considered as an "expansion" of the output. Actually, you are slaving the output device to the master of the SPI bus.</p>
	<p>Power LED indicator</p> <p>This LED should light up when you plug your Arduino into a power source to indicate that your board is powered up correctly. If this light does not turn on, then there is something wrong with the connection.</p>
	<p>TX and RX LEDs</p> <p>On your board, you will find two labels: TX (transmit) and RX (receive). They appear in two places on the Arduino UNO board. First, at the digital pins 0 and 1, to indicate the pins responsible for serial communication. Second, the TX and RX led (13). The TX led flashes with different speed while sending the serial data. The speed of flashing depends on the baud rate used by the board. RX flashes during the receiving process.</p>
	<p>Digital I/O</p> <p>The Arduino UNO board has 14 digital I/O pins (15) (of which 6 provide PWM (Pulse Width Modulation) output. These pins can be configured to work as input digital pins to read logic values (0 or 1) or as digital output pins to drive different modules like LEDs, relays, etc. The pins labeled "~" can be used to generate PWM.</p>

16

AREF

AREF stands for Analog Reference. It is sometimes, used to set an external reference voltage (between 0 and 5 Volts) as the upper limit for the analog input pins.

Software:

Arduino IDE (Integrated Development Environment) is required to program the Arduino Uno board.

Programming Arduino

Once arduino IDE is installed on the computer, connect the board with computer using USB cable. Now open the arduino IDE and choose the correct board by selecting Tools>Boards>Arduino/Genuino Uno, and choose the correct Port by selecting Tools>Port. Arduino Uno is programmed using Arduino programming language based on Wiring. To get it started with Arduino Uno board and blink the built-in LED, load the example code by selecting Files>Examples>Basics>Blink. Once the example code (also shown below) is loaded into your IDE, click on the 'upload' button given on the top bar. Once the upload is finished, you should see the Arduino's built-in LED blinking. Below is the example code for blinking:

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000);                     // wait for a second
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW
  delay(1000);                     // wait for a second
}
```

Raspberry Pi:

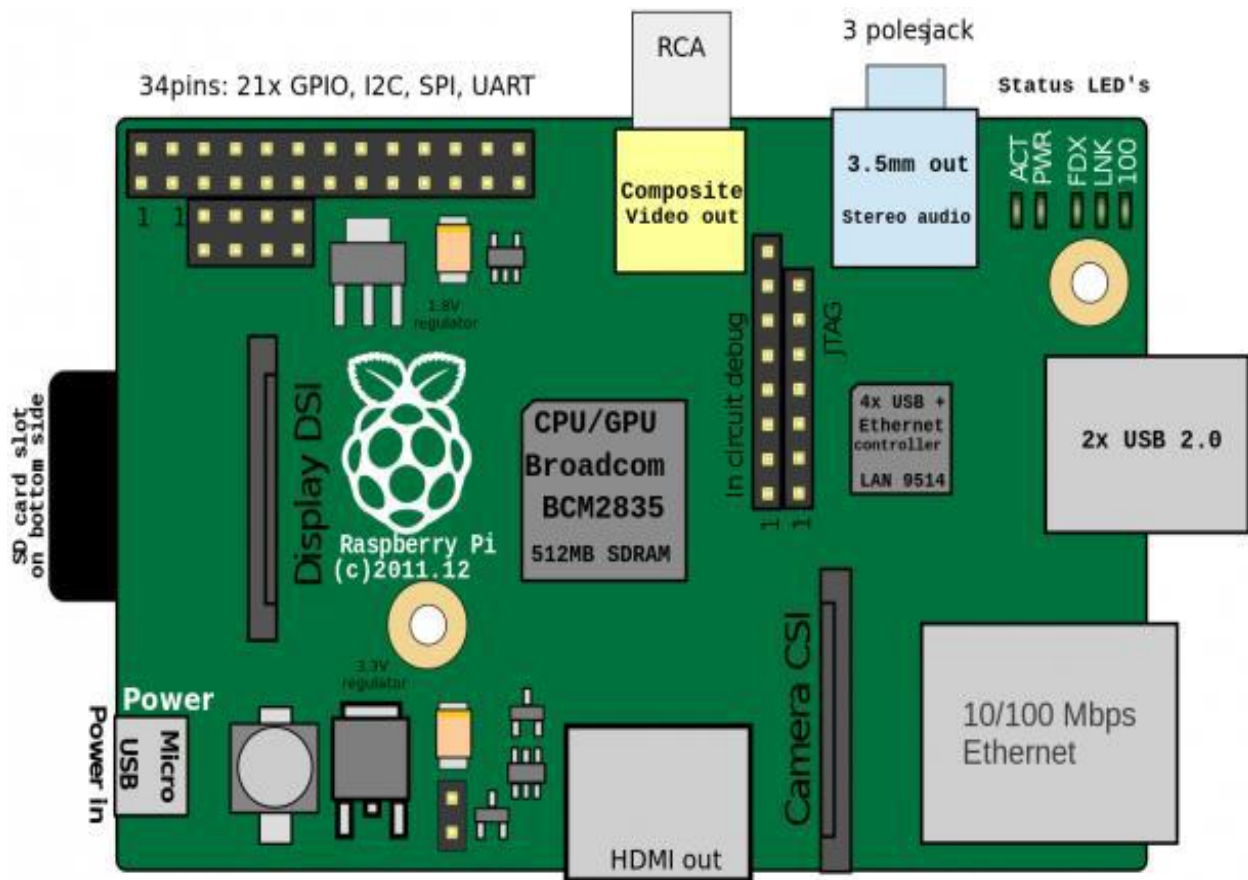
The Raspberry Pi launched in 2012 by the Raspberry Pi Foundation. it is a low-cost mini-computer with the physical size of a credit card. Raspberry Pi runs various flavors of Linux and can perform almost all tasks that a normal desktop computer can do. it also provides a set of GPIO (general purpose input/output) pins, allowing you to control electronic components (interfacing sensors and actuators) for physical computing and explore the Internet of Things (IoT). Raspberry Pi runs Linux operating system, it supports Python "out of the box".

There have been many generations of the Raspberry Pi line: from Pi 1 to 4, and even a Pi 400. There has generally been a Model A and a Model B of most generations.

- Raspberry Pi Model A – it has 256MB RAM, only one USB port and no network connection.
- Raspberry Pi Model B – it has 512MB RAM, 2 USB ports and a network connection.
- Raspberry Pi Model B+ – it has the similar specifications as the Model B, but comes with 4 USB ports, more GPIO pins, and uses less power than the Model B. This model costs \$35.
- Raspberry Pi 2 Model B – the latest version of the device, with 900 MHz quad-core ARM Cortex-A7 CPU and 1 GB of RAM.

Here is how the Raspberry Pi Model B looks like

a system on a chip (SoC) – an integrated circuit that incorporates many computer components on a single chip – the CPU, memory, and RAM. The **Raspberry Pi** model uses 700MHZ Low power **ARM1176JZ-F** processor, the powerful **GPU (Graphical Processing Unit)** capable of playing HD videos, and **512 MB** of SDRAM.



USB ports – Raspberry Pi Model B has TWO USB 2.0 ports. These USB ports can provide a current upto 100ma. standard USB 2.0 ports used to connect peripherals such as a keyboard and mouse.

Ethernet port – Raspberry Pi comes with a standard RJ45 (10/100 Mbit/s) Ethernet port used to connect an Ethernet cable or a USB wifi adapter to provide Internet connectivity.

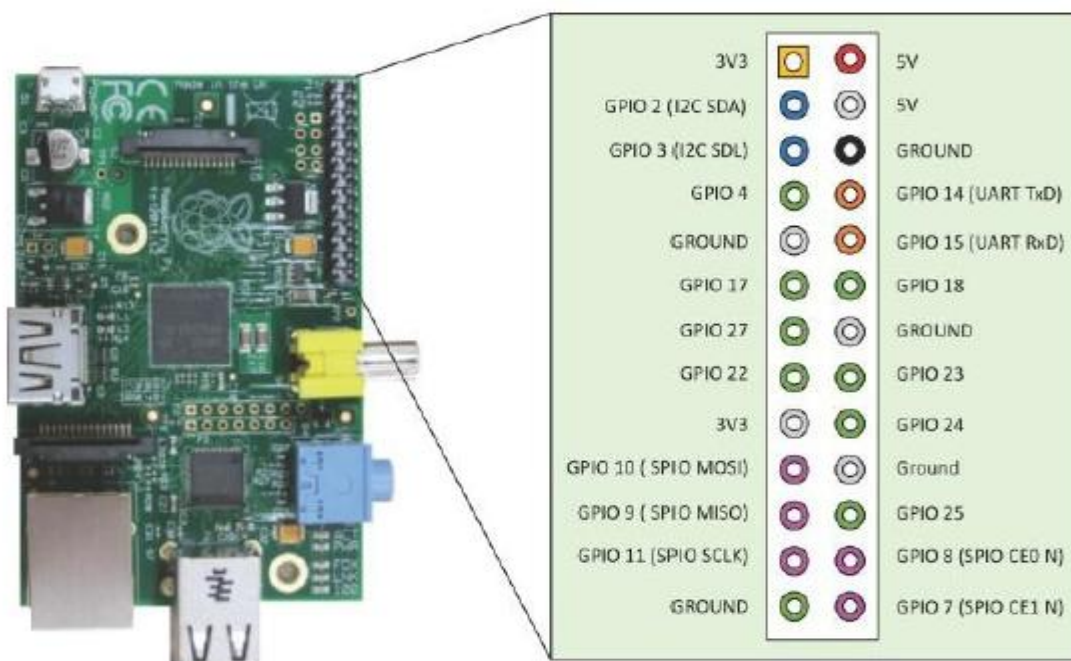
HDMI port – this port provides both video & audio output. You can connect the raspberry pi to a monitor or TV using an HDMI cable.

Composite Video and Audio Output: The composite Video output port with an RCA jack that supports both PAL & NTSC video output i.e carries video to the Video systems.the RCA jack can be used to connect old televisions that have an RAC input only.

Audio port –raspberry pi has a **3.5mm** audio output jack used to connect speakers .i.e it is used to provide audio output to old TV along with the RCA jack for video.

GPIO (General-Purpose Input/Output) pins – pins used to connect electronics devices. The Raspberry Pi Model B has **26** pins. There are 4 types of pins on raspberry pi .1. true GPIO pins, 2. I2C interface pins, 3. SPI interface pins & 4. Serial Rx & Tx pins.

Raspberry Pi GPIO



DSI (display serial interface) display connector – used to connect an LCD panel to raspberry pi.

CSI (camara serial interface) Camera connector – can be used to connect a camera module to raspberry pi to enables the capturing of photographs and videos.

Status LEDs: raspberry pi has 5 status LEDs. Table shows the list of Status LEDs & their functions.

- ACT – D5 [Green] – SD Card Access (via GPIO 16)
- PWR – D6 [Red] – 3.3 V Power is present

- FDX – D7 [Green] – Full Duplex (LAN) connected (Model B)
- LNK – D8 [Green] – Link/Activity (LAN) (Model B)
- 100 – D9 [Yellow] – 10/100 Mbit (LAN) connected (Model B)

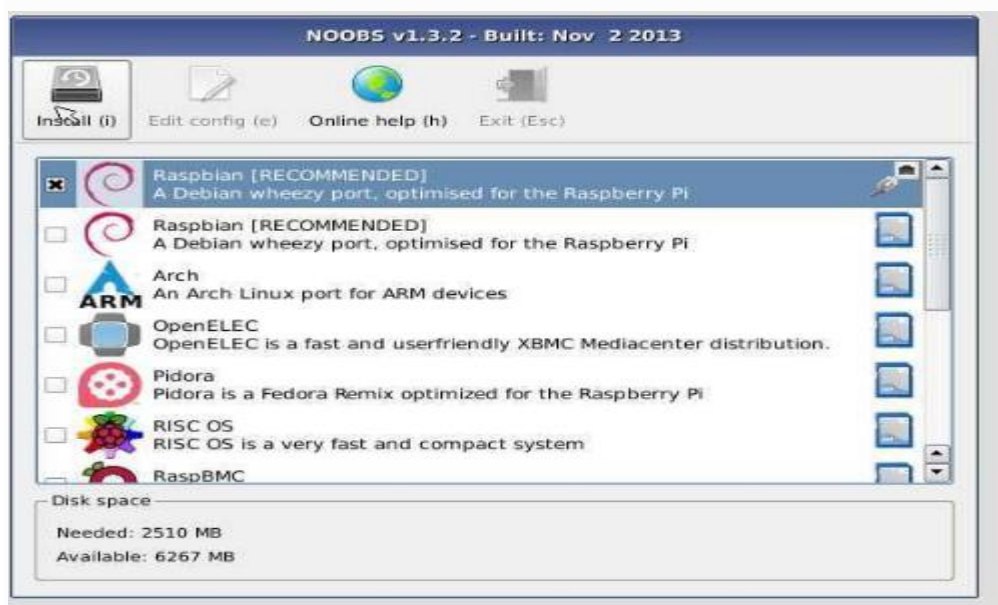
SD card slot: raspberry pi does not have a built in operating system & storage. You can plug in an SD card loaded with a linux image to the SD card slot.

Power input: raspberry pi has a micro USB connector for power input.

Linux on Raspberry pi:

Rasbian is the operating system of choice for beginners. It is based on a popular Linux distribution called **Debian** and was specially designed for the Raspberry Pi. Rasbian contains a lot of pre-installed programs that will help get you up and running and features a nice GUI. It runs **LXDE (Lightweight X11 Desktop Environment)** as the desktop environment, and **Openbox** as the window manager. It can be installed using **NOOBS**, the recommended install manager for the Raspberry Pi.

1. Download the NOOBS from [here](#).
2. Extract the zip file to a folder of your choice and transfer the files to your SD card:
3. Place the SD card in your Raspberry Pi. Connect the USB keyboard and the USB mouse. Use the HDMI port to connect your monitor.
4. Finally, power on your device by connecting it to the power outlet or the USB port on your PC. The NOOBS installer should start. Tick the checkbox next to **Rasbian** and click the **Install** button. The installation procedure should start:



5. When the install process is completed, the Raspberry Pi configuration menu (**raspi-config**) will load. To start Raspbian, select **Finish**:

Raspberry pi interfaces:

Raspberry pi has serial, SPI & I2C interfaces for data transfer.

Serial: the serial interface on raspberry pi has receive (Rx) & transmit (Tx) pins for communication with serial peripherals.

SPI (Serial Peripheral Interface) : it is a synchronous serial data protocol used for communicating with one or more peripheral devices. In SPI connection, there is one master device & one or more peripheral devices. There are 5 pins on raspberry pi for SPI interface:

- 1.MISO(Master in slave out): master line for sending data to the peripherals.
- 2.MOSI (Master out slave in): slave line for sending data to the master.
- 3.SCK(Serial clock): clock generated by master to synchronize data transmission
4. CE0 (Chip enable 0): to enable or disable devices
5. CE1 (Chip enable 1): to enable or disable devices

I2C: This pin allows you to connect hardware modules. I2C interface allows synchronous data transfer with just 2 pins- SDA (data line) & SCL (clock line).

Programming raspberry pi with python:

Raspberry pi runs linux & supports python out of the box. On raspberry pi board by using GPIO pins makes it useful device for IOT. You can interface a variety of sensors actuators with raspberry pi using the GPIO & spi, i2c & serial interfaces. input from sensors connected to raspberry pi can be processed & various actions can be taken , for instance, sending data to a server, sending an email, triggering a relay switch.

Led Blinking Program:

```
import RPi.GPIO as GPIO
import time #from time import sleep
GPIO.setmode(GPIO.BOARD) # set the numbering scheme to be the same as on the board
GPIO.setup(8, GPIO.OUT) # set the GPIO pin 8 to output mode
While true:
GPIO.output(8, true) # initiate the LED to on
Time.sleep(1)
GPIO.output(8, false)# LED OFF
Time.sleep(1)#sleep for one second
```

INTRODUCTION TO ARM PROCESSORS

The ARM microcontroller stands for Advance RISC Machine; it is one of the extensive and most licensed processor cores in the world. The first ARM processor was developed in the year 1978 by Cambridge University, and the first ARM RISC processor was produced by the Acorn Group of Computers in the year 1985. These processors are specifically used in portable devices like digital cameras, mobile phones, home networking modules and [wireless communication technologies](#) and other [embedded systems](#) due to the benefits, such as low power consumption, reasonable performance, etc.

ARM(**Advanced RISC Machines**) processors is based on the RISC architecture. it has the ability to support a wide variety of environments. Developed by British specialists, that produce different kinds of computer memory, interfaces, radios, computers, mobile devices, etc.

ARM based products are lightweight and portable. ARM products are the best solutions for embedded systems like smartphones, tablets or eBooks.

ARM Holdings offers users the following types of processors:

The cortex family has three main categories which are namely

- **Cortex-A (Application Processor cores)**
- **Cortex-R (Real Time Application cores)**
- **Cortex-M (Microcontroller Cores)**

Cortex-A

Cortex-A stands for Application processor cores which are widely used in performance-intensive applications and this will be put to best use while used with the applications related to the Android and Linux operating systems. The Cortex-A category of processors is dedicated to Linux and Android devices. Any devices – starting from smart watches and tablets and continuing with networking equipment – can be supported by Cortex-A processors.

Some technical information:

- The ARMv7-A architecture forms the basis of the A5, A7, A8, A9, A12, A15, and A17 processors;

- The set of common features for A-processors includes a media processing engine (NEON), a tool for security purposes (Trustzone), and various supported instruction sets (ARM, Thumb, DSP etc.)
- The main features of Cortex-A processors are top performance and brilliant power efficiency closely bundled to provide users with the best service possible.
- Cortex-A5 is the basic version of all the Cortex-A processors with low power consumption and desirable performance capabilities
- It is connected to large amount of memory.
- It handles large amount of applications and is capable of running complex operating system directly.
- It runs at relatively high clock frequency.

Cortex-M

The point of interest of Cortex-M processors is the MCU market. The first processor of this set was released more than 13 years ago. Since then, the popularity of M-processors has clearly risen: now, Cortex-M is known as an industry standard. The ARM processors of this type find their implementation in FPGA, integrated memories, clocks, etc.

One of the most successful processor products from ARM is the ARM7TDMI processor, which is used in many 32-bit microcontrollers around the world. Unlike traditional 32-bit processors, the ARM7TDMI supports two instruction sets, one called the ARM instruction set with 32-bit instructions and another 16-bit instruction set called Thumb. By allowing both instruction sets to be used on the processor, the code density is greatly increased, hence reducing memory footprint. At the same time, critical tasks can still execute with good speed. This enables ARM processors to be used in many portable devices that require low power and small memory. As a result, ARM processors are the first choice for mobile devices like mobile phones. It is built into microcontroller with I/O lines and designed for small factor systems that rely on heavy digital input and output.

Figure 2.10 shows a simplified block diagram of a microcontroller based on the ARM® Cortex™-M processor. It is a **Harvard architecture** because it has separate data and instruction buses. The Cortex-M instruction set combines the high performance typical of a 32-

bit processor with high code density typical of 8-bit and 16-bit microcontrollers. Instructions are fetched from flash ROM using the ICode bus. Data are exchanged with memory and I/O via the system bus interface. On the Cortex-M4 there is a second I/O bus for high-speed devices like USB. There are many sophisticated debugging features utilizing the DCode bus. The nested vectored interrupt controller (NVIC) manages **interrupts**, which are hardware-triggered software functions. Some internal peripherals, like the NVIC communicate directly with the processor via the private peripheral bus (PPB). The tight integration of the processor and interrupt controller provides fast execution of interrupt service routines (ISRs), dramatically reducing the interrupt latency.

Even though data and instructions are fetched 32-bits at a time, each 8-bit byte has a unique address. This means memory and I/O ports are byte addressable. The processor can read or write 8-bit, 16-bit, or 32-bit data. Exactly how many bits are affected depends on the instruction

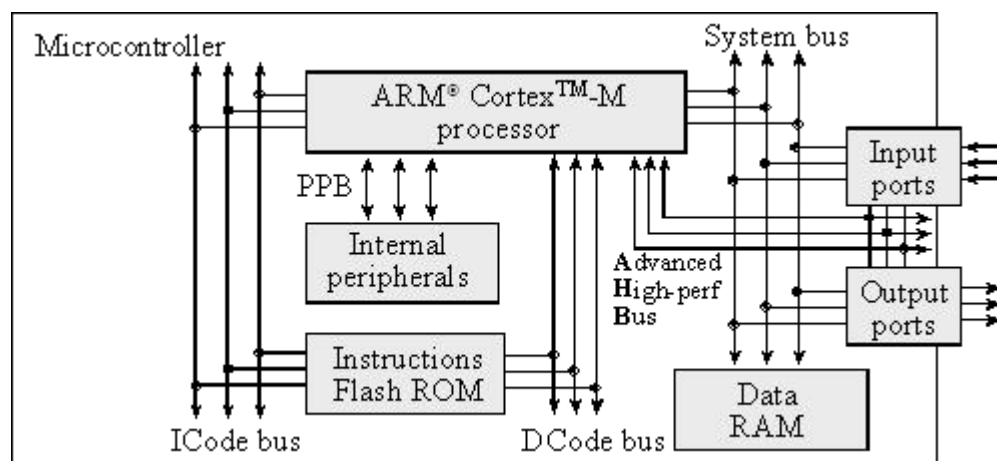


Figure 2.10. Harvard architecture of an ARM® Cortex-M-based microcontroller.

I/O Ports

The external devices attached to the microcontroller provide functionality for the system. A **pin** is one wire on the microcontroller used for input or output. There are 43 I/O pins on the TM4C123. A **port** is a collection of pins. An **input port** is hardware on the microcontroller that allows information about the external world to be entered into the computer. The microcontroller also has hardware called an **output port** to send information out to the external world

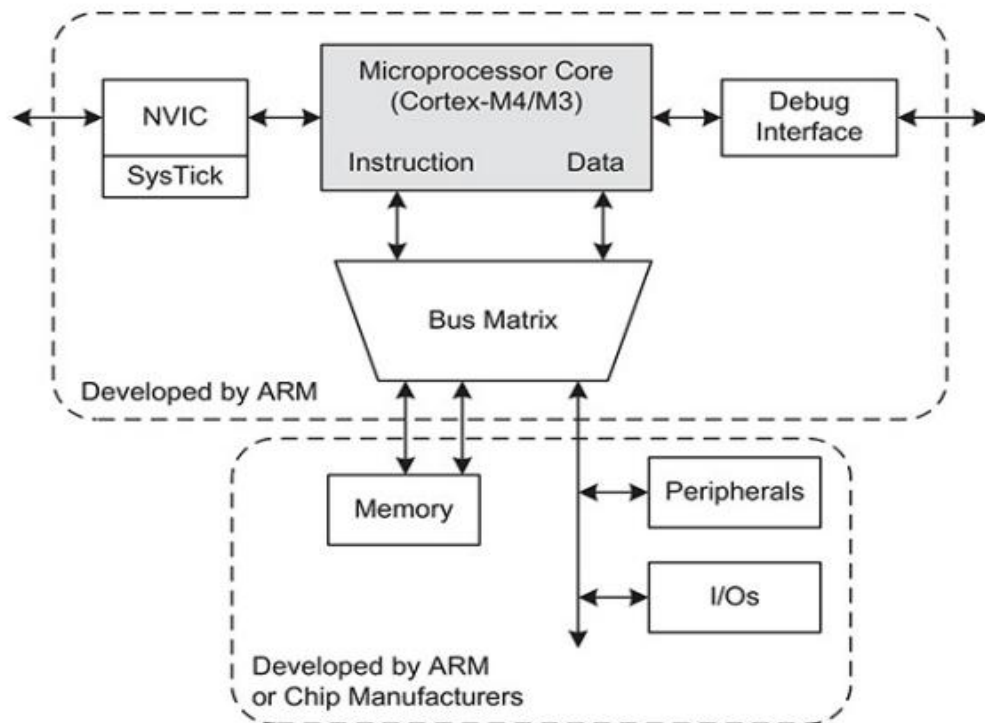


Figure 2.2: Block diagram of ARM Cortex-M based microcontroller

CPU Registers

Registers are high-speed storage inside the processor. The registers are depicted in Figure 2.12. R0 to R12 are general purpose registers and contain either data or addresses. Register R13 (also called the stack pointer, SP) points to the top element of the stack. Register R14 (also called the link register, LR) is used to store the return location for functions. The LR is also used in a special way during exceptions, such as interrupts. Register R15 (also called the program counter, PC) points to the next instruction to be fetched from memory. The processor fetches an instruction using the PC and then increments the PC.

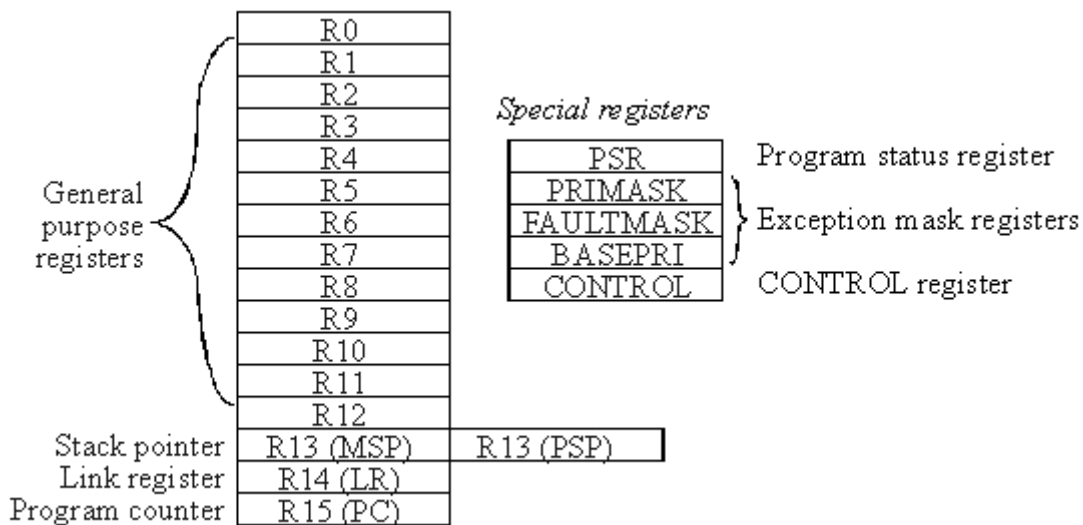


Figure 2.12. Registers on the ARM® Cortex-M processor.

There are three status registers named Application Program Status Register (APSR), the Interrupt Program Status Register (IPSR), and the Execution Program Status Register (EPSR) as shown in Figure 2.13. These registers can be accessed individually or in combination as the **Program Status Register** (PSR). The N, Z, V, C, and Q bits give information about the result of a previous ALU operation. In general, the **N bit** is set after an arithmetical or logical operation signifying whether or not the result is negative. Similarly, the **Z bit** is set if the result is zero. The **C bit** means carry and is set on an unsigned overflow, and the **V bit** signifies signed overflow. The **Q bit** indicates that “saturation” has occurred

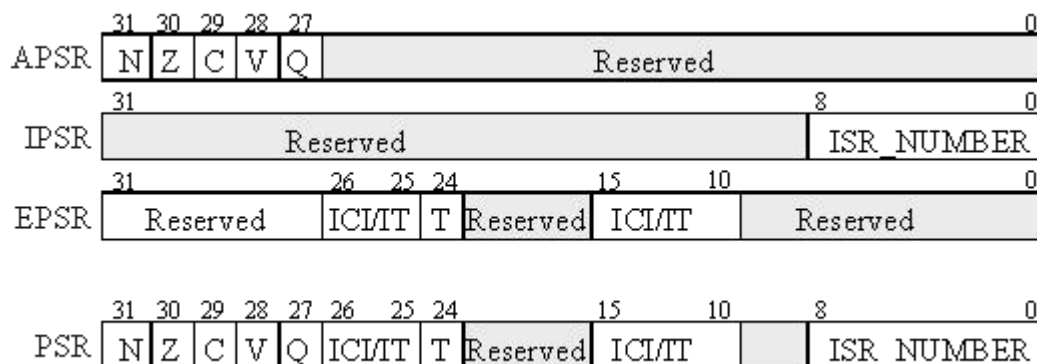


Figure 2.13. The program status register of the ARM® Cortex-M processor.

The **T bit** will always be 1, indicating the ARM® Cortex™-M processor is executing Thumb® instructions. The ISR_NUMBER indicates which interrupt if any the processor is handling. Bit 0 of the special register **PRIMASK** is the interrupt mask bit. If this bit is 1, most

interrupts and exceptions are not allowed. If the bit is 0, then interrupts are allowed. Bit 0 of the special register **FAULTMASK** is the fault mask bit. If this bit is 1, all interrupts and faults are not allowed. If the bit is 0, then interrupts and faults are allowed. The nonmaskable interrupt (NMI) is not affected by these mask bits. The **BASEPRI** register defines the priority of the executing software. It prevents interrupts with lower or equal priority but allows higher priority interrupts. For example if **BASEPRI** equals 3, then requests with level 0, 1, and 2 can interrupt, while requests at levels 3 and higher will be postponed. A lower number means a higher priority interrupt.

BLOCK DIAGRAM OF THE ARM CORTEX-M0:

The processor core contains the register banks, ALU, data path, and control logic. It is a three stage pipeline design with fetch stage, decode stage, and execution stage. The register bank has sixteen 32-bit registers. A few registers have special usages.

The Nested Vectored Interrupt Controller (NVIC) accepts up to 32 interrupt request signals and a non maskable interrupt (NMI) input. It contains the functionality required for comparing priority between interrupt requests and the current priority level so that nested interrupts can be handled automatically. If an interrupt is accepted, it communicates with the processor so that the processor can execute the correct interrupt handler.

The Wakeup Interrupt Controller (WIC) is an optional unit. In low-power applications, the microcontroller can enter standby state with most of the processor powered down. In this situation, the WIC can perform the function of interrupt masking while the NVIC and the processor core are inactive. When an interrupt request is detected, the WIC informs the power management to power up the system so that the NVIC and the processor core can then handle the rest of the interrupt processing.

The debug subsystem contains various functional blocks to handle debug control, program breakpoints, and data watch points. When a debug event occurs, it can put the processor core in a halted state so that embedded developers can examine the status of the processor at that point. The JTAG or serial wire interface units provide access to the bus system and debugging functionalities. The JTAG protocol is a popular five-pin communication protocol commonly used for testing. The serial wire protocol is a newer communication protocol that only requires two wires, but it can handle the same debug functionalities as JTAG.

The internal bus system, the data path in the processor core, and the AHB LITE bus interface are all 32 bits wide. AHB-Lite is an on-chip bus protocol used in many ARM processors. This bus protocol is part of the Advanced Microcontroller Bus Architecture (AMBA) specification, a bus architecture developed by ARM that is widely used in the IC design industry.

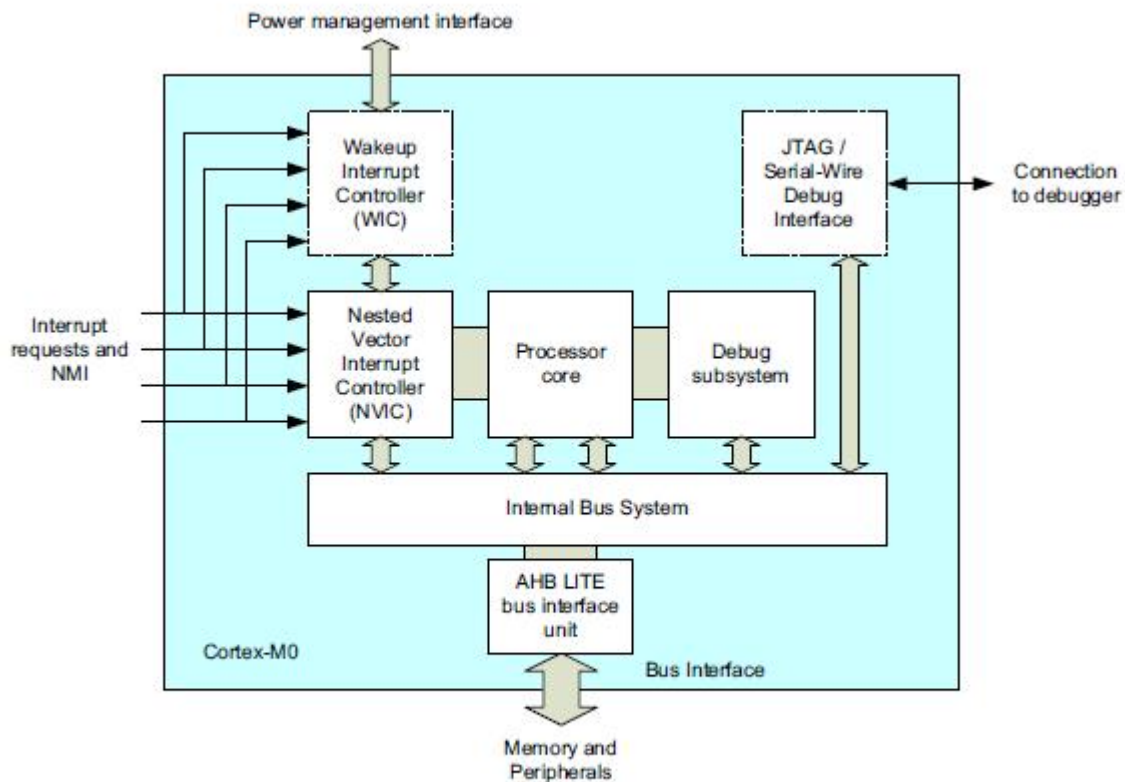


Figure 2.1:
Simplified block diagram of the Cortex-M0 processor.

ARCHITECTURE OF ARM CORTEX M0 PROCESSOR:

The Cortex-M0 processor has two operation modes and two states

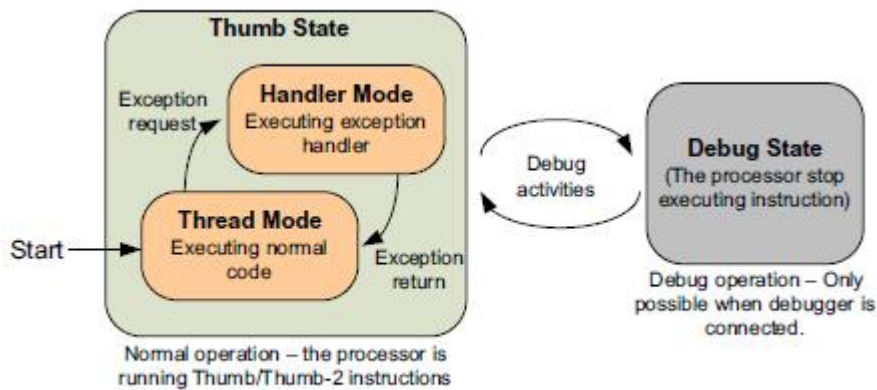


Figure 3.1:

Processor modes and states in the Cortex-M0 processor.

When the processor is running a program, it is in the Thumb state. In this state, it can be either in the Thread mode or the Handler mode. In the ARMv6-M architecture, the programmer's model of Thread mode and Handler mode are almost completely the same. The only difference is that Thread mode can use a shadowed stack pointer by configuring a special register called CONTROL. if control=1 ARM is in thread mode . in this mode it execute normal program. If control=0 ARM is in handler mode . in this mode it execute interrupt program

The Debug state is used for debugging operation only. Halting the processor stops the instruction execution and enters debug state. This state allows the debugger to access or change the processor register values. The debugger can access system memory locations in either the Thumb state or the Debug state. When the processor is powered up, it will be running in the Thumb state and Thread mode by default.

Registers and Special Registers

To perform data processing and controls, a number of registers are required inside the processor core. If data from memory are to be processed, they have to be loaded from the memory to a register in the register bank, processed inside the processor, and then written back to the memory if needed. This is commonly called a "load-store architecture." By having a sufficient number of registers in the register bank, this mechanism is easy to use and is C friendly.

The Cortex-M0 processor provides a register bank of 13 general-purpose 32-bit registers and a number of special registers. The register bank contains sixteen 32-bit registers. Most of them are general-purpose registers, but some have special uses.

R0-R12

Registers R0 to R12 are for general uses. Because of the limited space in the 16-bit Thumb instructions, many of the Thumb instructions can only access R0 to R7, which are also called the low registers, whereas some instructions, like MOV (move), can be used on all registers.

R13, Stack Pointer (SP)

R13 is the stack pointer. It is used for accessing the stack memory via PUSH and POP operations. There are physically two different stack pointers in Cortex-M0. The main stack pointer (MSP) is the default stack pointer after reset, and it is used when running exception handlers. The process stack pointer (PSP) can only be used in Thread mode (when not handling exceptions). The stack pointer selection is determined by the CONTROL register. Only one of the stack pointers is visible at a given time. In ARM processors, PUSH and POP are always 32-bit accesses because the registers are 32-bit, and the transfers in stack operations must be aligned to a 32-bit word boundary

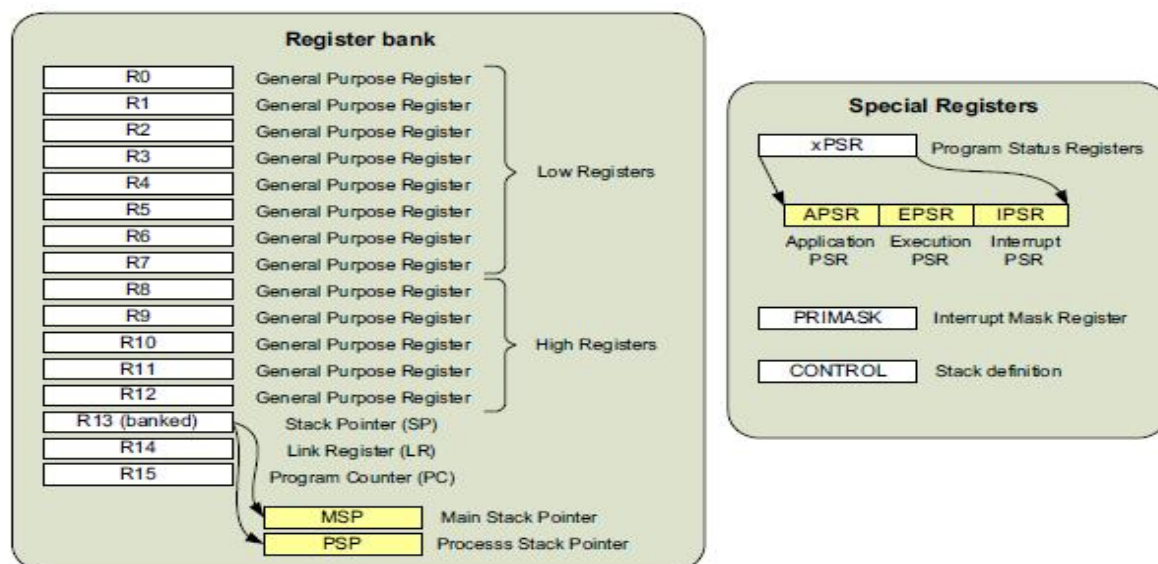


Figure 3.2:
Registers in the Cortex-M0 processor.

R14, Link Register (LR)

R14 is the Link Register. The Link Register is used for storing the return address of a subroutine or function call. At the end of the subroutine or function, the return address stored in LR is loaded into the program counter so that the execution of the calling program can be resumed. In the case where an exception occurs, the LR also provides a special code value, which is used by the exception return mechanism.

R15, Program Counter (PC)

Register R15 (also called the program counter, PC) points to the next instruction to be fetched from memory. The processor fetches an instruction using the PC and then increments the PC

xPSR, combined Program Status Register

The combined Program Status Register provides information about program execution and the ALU flags. It consists of the following three Program Status Registers (PSRs) (Figure 3.3):

- Application PSR (APSR)
- Interrupt PSR (IPSR)
- Execution PSR (EPSR)

The APSR contains the ALU flags: N (negative flag), Z (zero flag), C (carry or borrow flag), and V (overflow flag). These bits are at the top 4 bits of the APSR. The common use of these flags is to control conditional branches. The N, Z, V, C, and Q bits give information about the result of a previous ALU operation. In general, the **N bit** is set after an arithmetical or logical operation signifying whether or not the result is negative. Similarly, the **Z bit** is set if the result is zero. The **C bit** means carry and is set on an unsigned overflow, and the **V bit** signifies signed overflow. The **Q bit** indicates that “saturation” has occurred

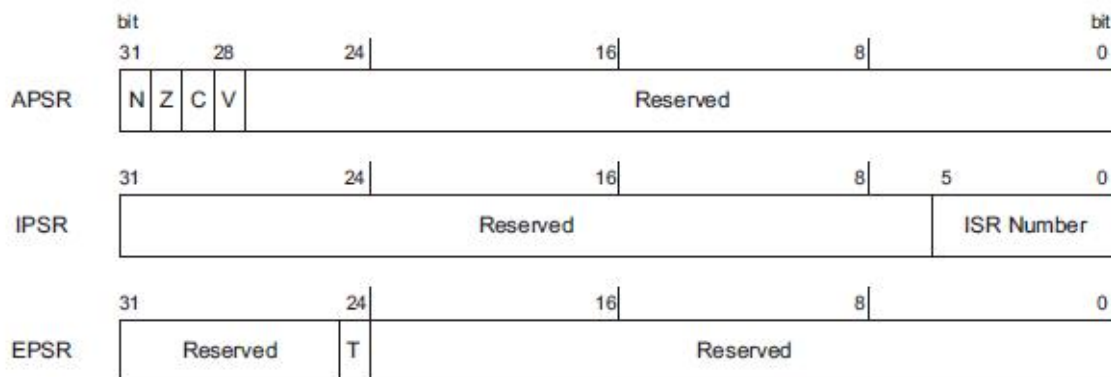


Figure 3.3:
APSR, IPSR, and EPSR.

The IPSR contains the current executing interrupt service routine (ISR) number. Each exception on the Cortex-M0 processor has a unique associated ISR number (exception type). This is useful for identifying the current interrupt type during debugging and allows an exception handler that is shared by several exceptions to know what exception it is serving.

The EPSR on the Cortex-M0 processor contains the T-bit, which indicates that the processor is in the Thumb state. On the Cortex-M0 processor, this bit is normally set to 1 because the Cortex-

M0 only supports the Thumb state. If this bit is cleared, a hard fault exception will be generated in the next instruction execution.

These three registers can be accessed as one register called xPSR (Figure 3.4). For example, when an interrupt takes place, the xPSR is one of the registers that is stored onto the stack memory automatically and is restored automatically after returning from an exception. During the stack store and restore, the xPSR is treated as one register.



Figure 3.4:
xPSR.

Direct access to the Program Status Registers is only possible through special register access instructions. However, the value of the APSR can affect conditional branches and the carry flag in the APSR can also be used in some data processing instructions.

PRIMASK: Interrupt Mask Special Register

The PRIMASK register is a 1-bit-wide interrupt mask register (Figure 3.5). When set, it blocks all interrupts apart from the nonmaskable interrupt (NMI) and the hard fault exception. Effectively it raises the current interrupt priority level to 0, which is the highest value for a programmable exception.

CONTROL: Special Register

There are two stack pointers in the Cortex-M0 processor. The stack pointer selection is determined by the processor mode as well as the configuration of the CONTROL register.

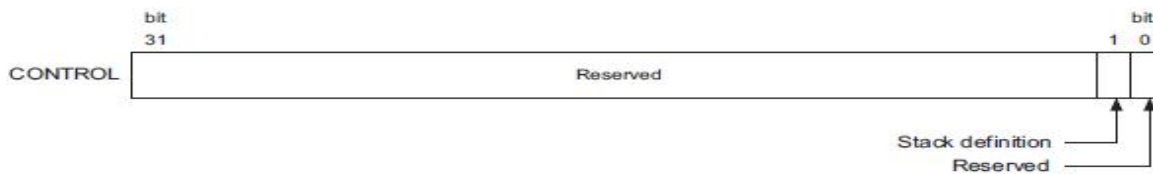


Figure 3.6:
CONTROL.

After reset, the main stack pointer (MSP) is used, but can be switched to the process stack pointer (PSP) in Thread mode (when not running an exception handler) by setting bit [1] in the CONTROL register (Figure 3.7). During running of an exception handler (when the processor is in Handler mode), only the MSP is used, and the CONTROL register reads as zero. The

CONTROL register can only be changed in Thread mode or via the exception entrance and return mechanism.

Bit 0 of the CONTROL register is reserved to maintain compatibility with the Cortex-M3 processor. In the Cortex-M3 processor, bit 0 can be used to switch the processor to User mode (non-privileged mode). This feature is not available in the Cortex-M0 processor.

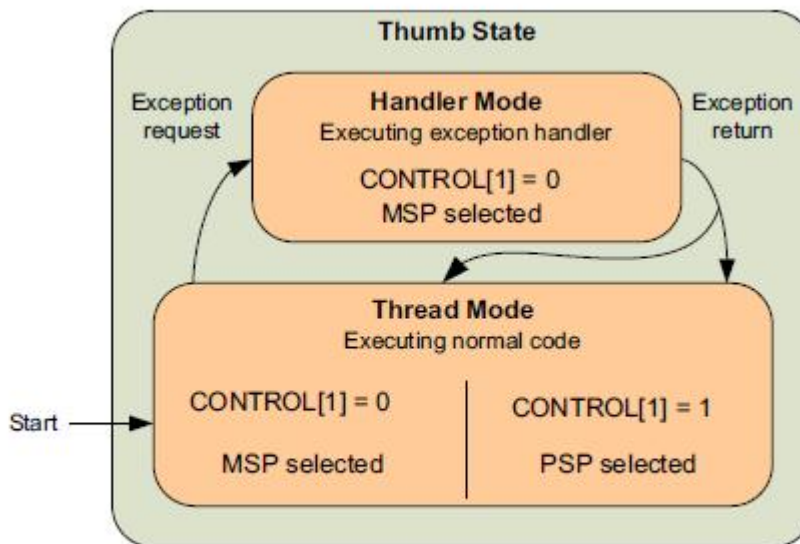


Figure 3.7:
Stack pointer selection.

Memory System Overview

The Cortex-M0 processor has 4 GB of memory address space (Figure 3.8). The memory space is architecturally defined as a number of regions, with each region having a recommended usage to help software porting between different devices.

The Cortex-M0 processor contains a number of built-in components like the NVIC and a number of debug components. These are in fixed memory locations within the system region of the memory map. As a result, all the devices based on the Cortex-M0 have the same programming model for interrupt control and debug. This makes it convenient for software porting and helps debug tool vendors to develop debug solutions for the Cortex-M0 based microcontroller or system-on-chip (SoC) products.

In most cases, the memories connected to the Cortex-M0 are 32-bits, but it is also possible to connect memory of different data widths to the Cortex-M0 processor with suitable memory

interface hardware. The Cortex-M0 memory system supports memory transfers of different sizes such as byte (8-bit), half word (16-bit), and word (32-bit). The Cortex-M0 design can be configured to support either little endian or big endian memory systems, but it cannot switch from one to another in an implemented design.

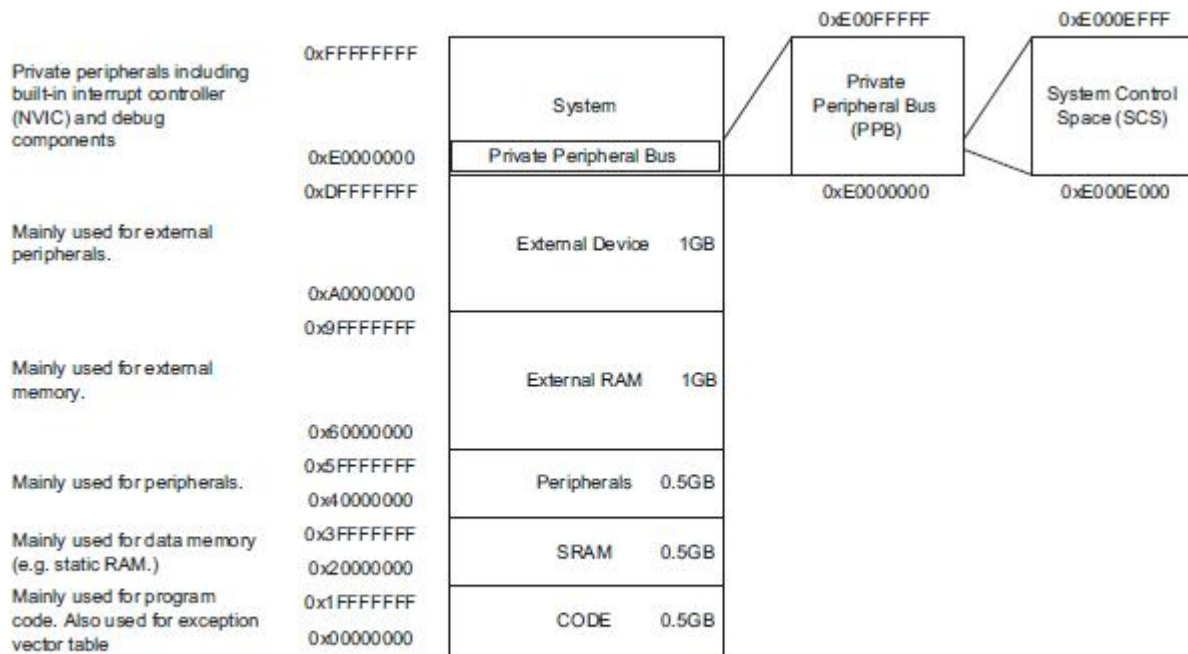


Figure 3.8:
Memory map.

Stack Memory Operations

Stack memory is a memory usage mechanism that allows the system memory to be used as temporary data storage that behaves as a first-in, last-out buffer. One of the essential elements of stack memory operation is a register called the stack pointer. The stack pointer is adjusted automatically each time a stack operation is carried out. In the Cortex-M0 processor, the stack pointer is register R13 in the register bank. Physically there are two stack pointers in the Cortex-M0 processor, but only one of them is used at one time, depending on the current value of the CONTROL register and the state of the processor (see [Figure 3.7](#)).

In common terms, storing data to the stack is called pushing (using the PUSH instruction) and restoring data from the stack is called popping (using the POP instruction). Depending on processor architecture, some processors perform storing of new data to stack memory using incremental address indexing and some use decrement address indexing. In the Cortex-M0

processor, the stack operation is based on a “full-descending” stack model. This means the stack pointer always points to the last filled data in the stack memory, and the stack pointer predecrements for each new data store (PUSH) (Figure 3.9).

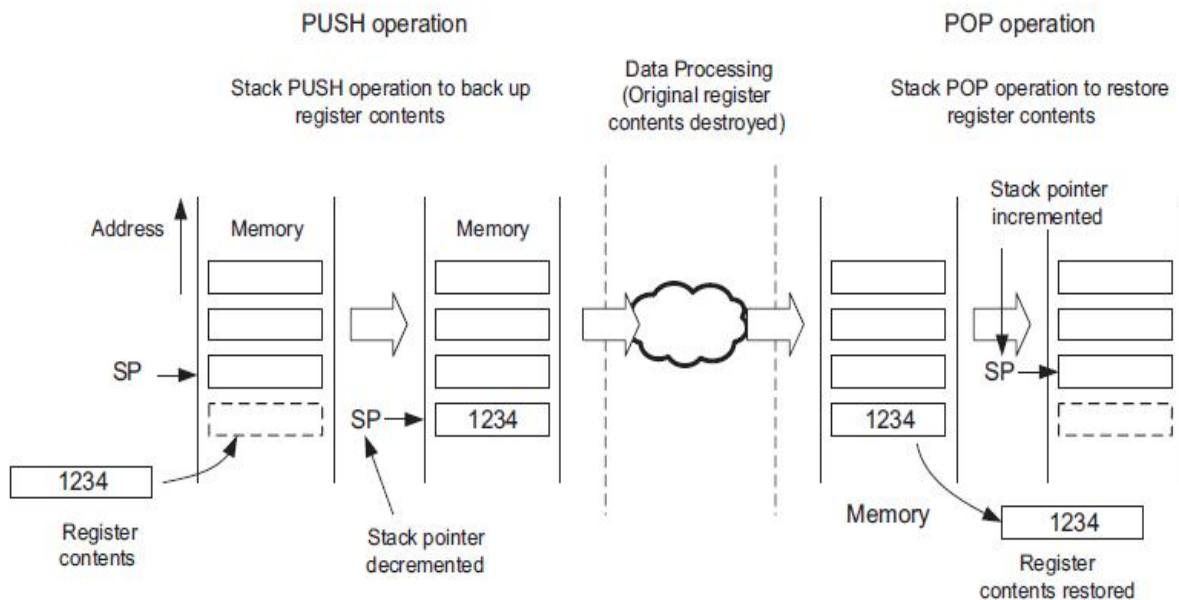


Figure 3.9:
Stack PUSH and POP in the Cortex-M0 processor.

PUSH and POP are commonly used at the beginning and end of a function or subroutine. At the beginning of a function, the current contents of the registers used by the calling program are stored onto the stack memory using a PUSH operation, and at the end of the function, the data on the stack memory is restored to the registers using a POP operation. Typically, each register PUSH operation should have a corresponding register POP operation, otherwise the stack pointer will not be able to restore registers to their original values. This can result in unpredictable behavior, for example, stack overflow.

The minimum data size to be transferred for each push and pop operations is one word (32-bit), and multiple registers can be pushed or popped in one instruction. The stack pointer can be accessed as either R13 or SP. Depending on the processor state and the CONTROL register value, the stack pointer accessed can either be the main stack pointer (MSP) or the process stack pointer (PSP). In many simple applications, only one stack pointer is needed and by default the main stack pointer (MSP) is used. The process stack pointer (PSP) is usually only required when an operating system (OS) is used in the embedded application (Table 3.3).

Table 3.3: Stack Pointer Usage Definition

Processor State	CONTROL[1] = 0 (Default Setting)	CONTROL[1] = 1 (OS Has Started)
Thread mode	Use MSP (R13 is MSP)	Use PSP (R13 is PSP)
Handler mode	Use MSP (R13 is MSP)	Use MSP (R13 is MSP)

In a typical embedded application with an OS, the OS kernel uses the MSP and the application processes use the PSP. This allows the stack for the kernel to be separate from stack memory for the application processes.

The initial value of MSP is stored at the beginning of the program memory. The initial value of PSP is undefined, and therefore the PSP must be initialized by software before using it.

Instruction set of arm cortex m0 processor:

One of the most successful processor products from ARM is the ARM7TDMI processor, which is used in many 32-bit microcontrollers around the world. Unlike traditional 32-bit processors, the ARM7TDMI supports two instruction sets, one called the ARM instruction set with 32-bit instructions and another 16-bit instruction set called Thumb. By allowing both instruction sets to be used on the processor, the code density is greatly increased, hence reducing memory footprint. At the same time, critical tasks can still execute with good speed. This enables ARM processors to be used in many portable devices that require low power and small memory. As a result, ARM processors are the first choice for mobile devices like mobile phones.

The state-switching mechanism is used to allow the processor to decide which instruction decode scheme should be used ([Figure 5.1](#)). The Thumb instruction set provides a subset of the ARM instructions. By itself it can perform most of the normal functions, but interrupt entry sequence and boot code must still be in ARM state.

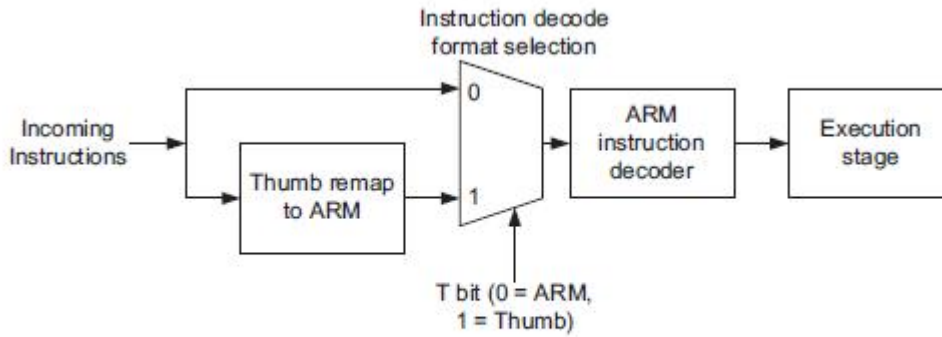


Figure 5.1:
ARM7TDMI design supports both ARM and the Thumb instruction set.

Instruction List

The instructions in the Cortex-M0 processor can be divided into various groups based on functionality:

- Moving data within the processor
- Memory accesses
- Stack memory accesses
- Arithmetic operations
- Logic operations
- Shift and rotate operations
- Extend and reverse ordering operations
- Program flow control (branch, conditional branch, and function calls)
- Memory barrier instructions
- Exception-related instructions
- Other functions

Memory Accesses:

Used to move signed & unsigned word, halfword & byte to & from registers.

Load--- memory to register

Store---- register to memory

Does not support memory to memory data processing operations.

The Cortex-M0 processor supports a number of memory access instructions, which support various data transfer sizes and addressing modes. The supported data transfer sizes are Word,

HalfWord and Byte. In addition, there are separate instructions to support signed and unsigned data.

For memory read operations, the instruction to carry out single accesses is LDR (load):

LDR/LDRH/LDRB: Read single memory data into register

LDR <Rt>, [<Rn>, <Rm>] ; Word read i.e load word data into register

LDRH <Rt>, [<Rn>, <Rm>] ; Half Word read

LDRB <Rt>, [<Rn>, <Rm>] ; Byte read

Rt = memory[Rn + Rm]

The Cortex-M0 processor can also sign extend the read data automatically using the LDRSB and LDRSH instructions. This is useful when a signed 8-bit/16-bit data type is used,

LDRSH/LDRSB: Read single signed memory data into register

Syntax- LDRSH <Rt>, [<Rn>, <Rm>] ; Half word read i.e load signed half word

LDRSB <Rt>, [<Rn>, <Rm>] ; Byte read

For single data memory writes, the instruction is STR (store):

STR/STRH/STRB: Write single register data into memory

Syntax STR <Rt>, [<Rn>, <Rm>] ; Word write i.e store word

STRH <Rt>, [<Rn>, <Rm>] ; Half Word write

STRB <Rt>, [<Rn>, <Rm>] ; Byte write

Note : memory[Rn + Rm] = Rt:

One of the important features in ARM processors is the ability to load or store multiple registers with one instruction. There is also an option to update the base address register to the next location. For load/store multiple instructions, the transfer size is always in word size.

LDM (Load Multiple): Read multiple memory data word into registers, base address register update by memory read

Syntax : LDM <Rn>, {<Ra>, <Rb> ,...} ; Load multiple registers from memory

Ex: LDM R2, {R1, R2, R5 e R7} ; Read R1,R2,R5,R6 and R7 from memory.

LDMIA (Load Multiple Increment After): Read multiple memory data word into registers and update base register

LDMIA <Rn>!, {<Ra>, <Rb> ,...} ; Load multiple registers from memory ; and increment base register after completion

LDMIA R0!, {R1, R2, R5 e R7} ; Read multiple registers, R0 update to address after last read operation.

STMIA (Store Multiple Increment After: Write multiple register data into memory and update base register

STMIA <Rn>!, {<Ra>, <Rb> ,...} ; Store multiple registers to memory ; and increment base register after completion

STMIA R0!, {R1, R2, R5 e R7} ; Store R1, R2, R5, R6, and R7 to memory ; and update R0 to address after where R7 stored

Stack Memory Accesses

Two memory access instructions are dedicated to stack memory accesses. The PUSH instruction is used to decrement the current stack pointer and store data to the stack. The POP instruction is used to read the data from the stack and increment the current stack pointer. Both PUSH and POP instructions allow multiple registers to be stored or restored.

PUSH: Write single or multiple registers (low register and LR) into memory and update base register (stack pointer)

Syntax: PUSH {<Ra>, <Rb> ,...} ; Store multiple registers to memory and ; decrement SP to the lowest pushed data address

PUSH {R1, R2, R5 e R7, LR} ; Store R1, R2, R5, R6, R7, and LR to stack

POP: Read single or multiple registers (low register and PC) from memory and update base register (stack pointer)

Syntax: POP {<Ra>, <Rb> ,...} ; Load multiple registers from memory ; and increment SP to the last emptied stack address plus 4

POP {R1, R2, R5 e R7} ; Restore R1, R2, R5, R6, R7 from stack

Arithmetic Operations

The Cortex-M0 processor supports a number of arithmetic operations. The most basic are add, subtract, two's complement, and multiply. For most of these instructions, the operation can be carried out between two registers, or between one register and an immediate constant.

ADD: Add two registers

Syntax (UAL): ADDS <Rd>, <Rn>, <Rm>

Syntax (Thumb): ADD <Rd>, <Rn>, <Rm>

Note: Rd= Rn + Rm, APSR update.

ADD: Add an immediate constant into a register

Syntax (UAL) ADDS <Rd>, <Rn>, #immed3

ADDS <Rd>, #immed8

Note: $Rd = Rn + \text{ZeroExtend}(\#immed3)$, APSR update

ADR (ADD): Add an immediate constant with PC to a register without updating APSR

Syntax (UAL): ADR <Rd>, <label> (normal syntax)

ADD <Rd>, PC, #immed8 (alternate syntax)

Note: $Rd = (PC[31:2] \ll 2) \vee \text{ZeroExtend}(\#immed8 \ll 2)$.

This instruction is useful for locating a data address within the program memory near to the current instruction. The result address must be word aligned.

ADC: Add with carry and update APSR

Syntax (UAL): ADCS <Rd>, <Rm>

Note: $Rd = Rd + Rm + \text{Carry}$

SUB: Subtract two registers

Syntax (UAL): SUBS <Rd>, <Rn>, <Rm>

Note: $Rd = Rn - Rm$, APSR update.

SBC: Subtract with carry (borrow)

Syntax (UAL) :SBCS <Rd>, <Rd>, <Rm>

Syntax (Thumb) : SBC <Rd>, <Rm>

Note: $Rd = Rd - Rm - \text{Borrow}$, APSR update.

RSB: Reverse Subtract (negative)

Syntax (UAL): RSBS <Rd>, <Rn>, #0

Syntax (Thumb) : NEG <Rd>, <Rn>

Note: $Rd = 0 - Rm$, APSR update.

MUL: Multiply

Syntax: (UAL):MULS <Rd>, <Rm>, <Rd>

Syntax (Thumb); MUL <Rd>, <Rm>

Note: $Rd = Rd * Rm$, APSR.N, and APSR.Z update.

There are also a few compare instructions that compare (using subtract) values and update flags (APSR), but the result of the comparison is not stored.

CMP: Compare

Syntax (UAL): CMP <Rn>, <Rm>

Syntax (Thumb): CMP <Rn>, <Rm>

Note : Calculate $Rn - Rm$, APSR update but subtract result is not stored.

CMN: Compare negative

Syntax (UAL): CMN <Rn>, <Rm>

Syntax (Thumb): CMN <Rn>, <Rm>

Note : Calculate $Rn - \text{NEG}(Rm)$, APSR update but subtract result is not stored. Effectively the operation is an ADD.

Logic Operations

Another set of essential operations in most processors is made up of logic operations. For logical operations, the Cortex-M0 processor has a number of instructions available, including basic features like AND, OR, and the like. In addition, it has a number of instructions for compare and testing.

AND: Logical AND

Syntax (UAL) :ANDS <Rd>, <Rd>, <Rm>

Syntax (Thumb): AND <Rd>, <Rm>

Note: $Rd = \text{AND}(Rd, Rm)$, APSR.N, and APSR.Z update.

ORR: Logical OR

Syntax (UAL): ORRS <Rd>, <Rd>, <Rm>

Syntax (Thumb): ORR <Rd>, <Rm>

Note: $Rd = \text{OR}(Rd, Rm)$, APSR.N, and APSR.Z update.

EOR: Logical Exclusive OR

Syntax (UAL) :EORS <Rd>, <Rd>, <Rm>

Syntax (Thumb); EOR <Rd>, <Rm>

Note : $Rd = \text{XOR}(Rd, Rm)$, APSR.N, and APSR.Z update.

BIC: Logical Bitwise Clear

Syntax (UAL) :BICS <Rd>, <Rd>, <Rm>

Syntax (Thumb) :BIC <Rd>, <Rm>

Note: $Rd = \text{AND}(Rd, \text{NOT}(Rm))$, APSR.N, and APSR.Z update

MVN: Logical Bitwise NOT

Syntax (UAL): MVNS <Rd>, <Rm>

Syntax (Thumb): MVN <Rd>, <Rm>

Note :Rd = NOT(Rm), APSR.N, and APSR.Z update.

TST: Test (bitwise AND)

Syntax (UAL) :TST <Rn>, <Rm>

Syntax (Thumb): TST <Rn>, <Rm>

Note :Calculate AND(Rn, Rm), APSR.N, and APSR.Z update, but the AND result is not stored.

Shift and Rotate Operations

The Cortex-M0 also supports shift and rotate instructions. It supports both arithmetic shift operations (the datum is a signed integer value where MSB needs to be reserved) as well as logical shift.

ASR: Arithmetic Shift Right

Syntax (UAL): ASRS <Rd>, <Rd>, <Rm>

Syntax (Thumb) :ASR <Rd>, <Rm>

Note :Rd = Rd >> Rm, last bit shift out is copy to APSR.C,

APSR.N and APSR.Z are also updated.

When ASR is used, the MSB of the result is unchanged, and the Carry flag is updated using the last bit shifted out (Figure 5.3).

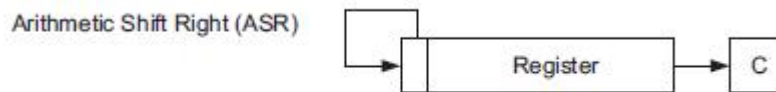


Figure 5.3:
Arithmetic Shift Right.

For logical shift operations, the instructions are LSL (Figure 5.4) and LSR (Figure 5.5).

LSL: Logical Shift Left

Syntax (UAL): LSLS <Rd>, <Rd>, <Rm>

Syntax (Thumb): LSL <Rd>, <Rm>

Note: Rd = Rd << Rm, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated.

Logical Shift Left (LSL)



Figure 5.4:
Logical Shift Left.

LSR: Logical Shift Right

Syntax (UAL): LSRS <Rd>, <Rd>, <Rm>

Syntax (Thumb): LSR <Rd>, <Rm>

Note :Rd = Rd >> Rm, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated.

Logical Shift Right (LSR)



Figure 5.5:
Logical Shift Right.

ROR: Rotate Right

Syntax (UAL): RORS <Rd>, <Rd>, <Rm>

Syntax (Thumb): ROR <Rd>, <Rm>

Note :Rd = Rd rotate right by Rm bits, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated.

Rotate Right (ROR)

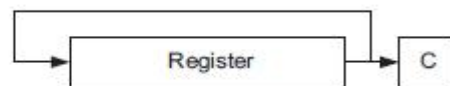


Figure 5.6:
Rotate Right.

Extend and Reverse Ordering Operations:

The Cortex-M0 processor supports a number of instructions that can perform data reordering or Extraction.

REV (Byte-Reverse Word): Byte Order Reverse

Syntax: REV <Rd>, <Rm>

Note :Rd = {Rm[7:0], Rm[15:8], Rm[23:16], Rm[31:24]}

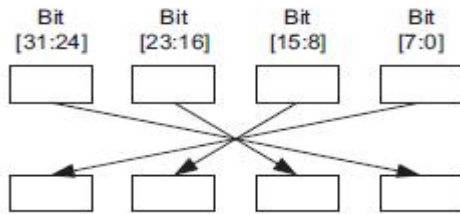


Figure 5.7:
REV operation.

REV16 (Byte-Reverse Packed Half Word): Byte Order Reverse within half word

Syntax : REV16 <Rd>, <Rm>

Note : Rd = {Rm[23:16], Rm[31:24], Rm[7:0] , Rm[15:8]}

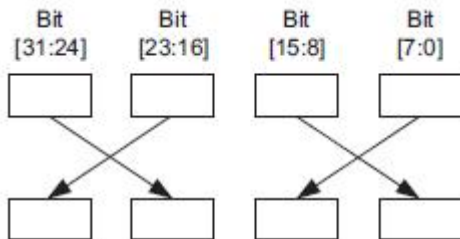


Figure 5.8:
REV16 operation.

REVSH (Byte-Reverse Signed Half Word): Byte order reverse within lower half word, then sign extend result

Syntax: REVSH <Rd>, <Rm>

Note: Rd = SignExtend({Rm[7:0] , Rm[15:8]})

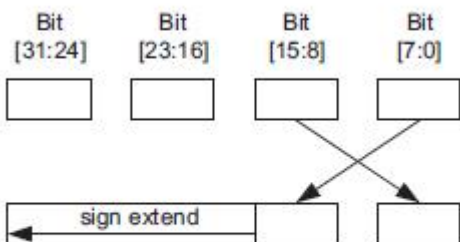


Figure 5.9:
REVSH operation.

These reverse instructions are usually used for converting data between little endian and big endian systems.

The SXTB, SXTH, UXT, and UXTH instructions are used for extending a byte or half word data into a word. They are usually used for data type conversions.

SXTB (Signed Extended Byte): SignExtend lowest byte in a word of data

Syntax : SXTB <Rd>, <Rm>

Note : Rd = SignExtend(Rm[7:0])

SXTH (Signed Extended Half Word): SignExtend lower half word in a word of data

Syntax: SXTH <Rd>, <Rm>

Note: Rd = SignExtend(Rm[15:0])

UXTB (Unsigned Extended Byte): Extend lowest byte in a word of data

Syntax : UXTB <Rd>, <Rm>

Note: Rd = ZeroExtend(Rm[7:0])

UXTH (Unsign Extended Half Word): Extend lower half word in a word of data

Syntax: UXTH <Rd>, <Rm>

Note: Rd = ZeroExtend(Rm[15:0])

With SXTB or SXTH, the data are extended using bit[7] or bit[15] of the input data, whereas for UXTB and UXTH, the data are extended using zeros. For example, if R0 is 0x55AA8765, the result of these extended instructions is

SXTB R1, R0 ; R1 = 0x00000065

SXTH R1, R0 ; R1 = 0xFFFF8765

UXTB R1, R0 ; R1 = 0x00000065

UXTH R1, R0 ; R1 = 0x00008765

Program Flow Control

There are five branch instructions in the Cortex-M0 processor. They are essential for program flow control like looping and conditional execution, and they allow program code to be partitioned into functions and subroutines.

B (Branch): Branch to an address (unconditional)

Syntax: B <label>

B<cond> (Conditional Branch): Depending of APSR, branch to an address

Syntax: B<cond> <label>

The <cond> is one of the 14 possible condition suffixes

Table 5.7: Condition Suffixes for Conditional Branch

Suffix	Branch Condition	Flags (APSR)
EQ	Equal	Z flag is set
NE	Not equal	Z flag is cleared
CS/HS	Carry set / unsigned higher or same	C flag is set
CC/LO	Carry clear / unsigned lower	C flag is cleared
MI	Minus / negative	N flag is set (minus)
PL	Plus / positive or zero	N flag is cleared
VS	Overflow	V flag is set
VC	No overflow	V flag is cleared
HI	Unsigned higher	C flag is set and Z is cleared
LS	Unsigned lower or same	C flag is cleared or Z is set
GE	Signed greater than or equal	N flag is set and V flag is set, or N flag is cleared and V flag is cleared (N == V)
LT	Signed less than	N flag is set and V flag is cleared, or N flag is cleared and V flag is set (N != V)
GT	Signed greater then	Z flag is cleared, and either both N flag and V flag are set, or both N flag and V flag are cleared (Z == 0 and N == V)
LE	Signed less than or equal	Z flag is set, or either N flag set with V flag cleared, or N flag cleared and V flag set (Z == 1 or N != V)

BL (Branch and Link): Branch to an address and store return address to LR. Usually use for function calls, and can be used for long-range branch that is beyond the branch range of branch instruction (B <label>).

Syntax : BL <label>

Note: Branch range is \pm 16MB of current program counter.

For example,

BL function A ; call a function called function A

BX (Branch and Exchange): Branch to an address specified by a register, and change processor state depending on bit[0] of the register.

Syntax: BX <Rm>

Note: Because the Cortex-M0 processor only supports Thumb code, bit[0] of the register content (Rm) must be set to 1, otherwise it means it is trying to switch to the ARM state and this will generate a fault exception.

BL is commonly used for calling a subroutine or function. When it is executed, the address of the next instruction will be stored to the Link Register (LR), with the LSB set to 1. When the subroutine or function completes the required task, it can then return to the calling program by executing a “BX LR” instruction (Figure 5.10).

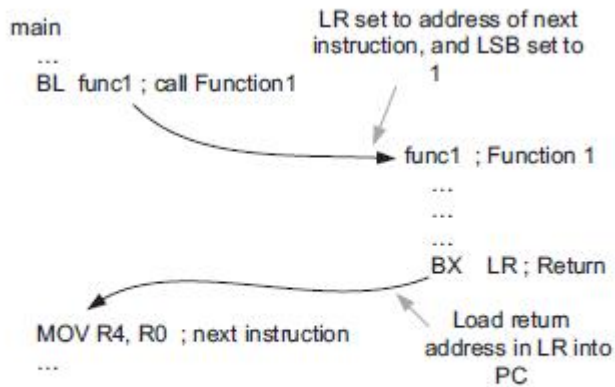


Figure 5.10:
Function call and return using BL and BX instructions.

BX can also be used to branch to an address that has an offset that is more than the normal branch instruction. Because the target is specified by a 32-bit register, it can branch to any address in the memory map.

BLX (Branch and Link with Exchange):

Branch to an address specified by a register, save return address to LR, and change processor state depending on bit[0] of the register.

Syntax: BLX <Rm>

Note: Because the Cortex-M0 processor only supports Thumb code, the bit [0] of the register content (Rm) must be set to 1, otherwise it means it is trying to switch to the ARM state and this will create a fault exception.

BLX is used when a function call is required but the address of the function is held inside a register (e.g., when working with function pointers).

Memory Barrier Instructions

Memory barrier instructions are often needed when the memory system is complex. In some cases, if the memory barrier instruction is not used, race conditions could occur and cause system failures. For example, in some ARM processors that supports simultaneous bus transfers (as a processor can have multiple memory interfaces), the transfer sequence of these transfers might overlap. If the software code relies on strict ordering of memory access sequences, it could result in software errors in corner cases. The memory barrier instructions allow the processor to stop executing the next instruction, or stop starting a new transfer, until the current memory access has completed.

Because the Cortex-M0 processor only has a single memory interface to the memory system and does not have a write buffer in the system bus interface, the memory barrier instruction is rarely needed.

There are three memory barrier instructions that support on the Cortex-M0 processor:

- DMB
- DSB
- ISB

DMB: Data Memory Barrier

Syntax: DMB

Note: Ensures that all memory accesses are completed before new memory access is committed

DSB: Data Synchronization Barrier

Syntax: DSB

Note: Ensures that all memory accesses are completed before the next instruction is executed

ISB: Instruction Synchronization Barrier

Syntax: ISB

Note: Flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

Exception-Related Instructions

The Cortex-M0 processor provides an instruction called supervisor call (SVC). This instruction causes the SVC exception to take place immediately if the exception priority level of SVC is higher than current level.

SVC: Supervisor call

Syntax: SVC #<immed8>

SVC <immed8>

Note: Trigger the SVC exception. For example, SVC #3 ; SVC instruction, with parameter, equals 3.

CPS: Change processor state: enable or disable interrupt

Syntax: CPSIE I ; Enable Interrupt (Clearing PRIMASK)

CPSID I ; Disable Interrupt (Setting PRIMASK)

Sleep Mode Feature Related Instructions

The Cortex-M0 processor can enter sleep mode by executing the Wait-for-Interrupt (WFI) and Wait-for-Event (WFE) instructions. The Send Event (SEV) instruction is normally used in multiprocessor systems to wake up other processors that are in sleep mode by means of the WFE instruction.

Other Instructions

The Cortex-M0 processor supports an NOP instruction. This instruction can be used to produce instruction alignment or to introduce delay.

The breakpoint instruction (BKPT) is used to provide a breakpoint function during debug. Usually a debugger, replacing the original instruction, inserts this instruction. When the breakpoint is hit, the processor would be halted, and the user can then carry out the debug tasks through the debugger. The Cortex-M0 processor also has a hardware breakpoint unit.

UNIT:3

12. Transmission Control Protocol:

The transmission Control Protocol (TCP) is one of the most important protocols of Internet Protocols suite. It is most widely used protocol for data transmission in communication network such as internet.

For example, When a user requests a web page on the internet, somewhere in the world, the server processes that request and sends back an HTML Page to that user. The server makes use of a protocol called the HTTP Protocol. The HTTP then requests the TCP layer to set the required connection and send the HTML file.

Now, the TCP breaks the data into small packets and forwards it towards the Internet Protocol (IP) layer. The packets are then sent to the destination through different routes.

The TCP layer in the user's system waits for the transmission to get finished and acknowledges once all packets have been received.

Features

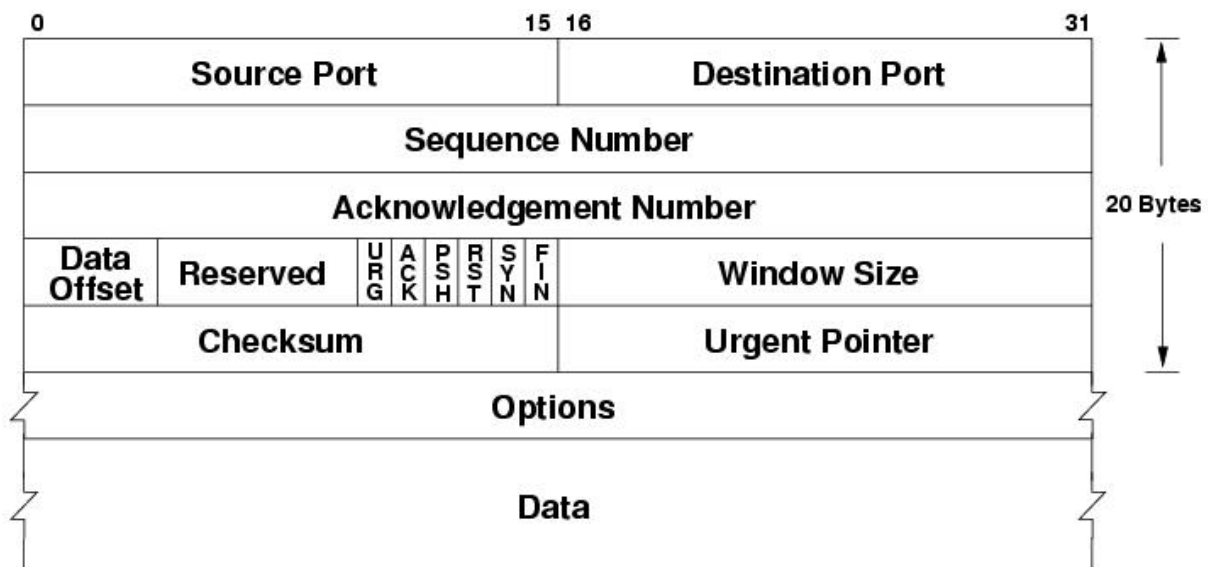
- TCP is reliable protocol. That is, the receiver always sends either positive or negative acknowledgement about the data packet to the sender, so that the sender always has

bright clue about whether the data packet is reached the destination or it needs to resend it.

- TCP ensures that the data reaches intended destination in the same order it was sent.
- TCP is connection oriented. TCP requires that connection between two remote points be established before sending actual data.
- TCP provides error-checking and recovery mechanism.
- TCP provides end-to-end communication.
- TCP provides flow control and quality of service.
- TCP operates in Client/Server point-to-point mode.
- TCP provides full duplex server, i.e. it can perform roles of both receiver and sender.

Header

- The length of TCP header is minimum 20 bytes long and maximum 60 bytes.



Source port

It is a 16-bit field that indicates the port number of the source sending the data.

Destination port

It is a 16-bit field. The destination port is the numerical value indicating the destination port..

Sequence number

It is a 32-bit field that is used to put the data back in the correct order and also used to re-transmit missing or damaged data segments.

Acknowledgment number

It is a 32-bit field that is used by the receiving host to acknowledge the successful delivery of segments based on which the next stream of data segments is sent by the source. When the ACK bit is set, this field contains the next sequence number that the sender of the segment is expecting to receive. This value is always sent.

Header Length(HLEN)

It is a 4-bit field . This field indicates the length of the TCP header so that we know where the actual data begins.

Reserved

It is a 3 bits field with value always set to 0.

Window

It is a 16-bit field that is used to negotiate the window size b/w sending and receiving hosts. This field specifies the number of bytes the receiver is willing to receive. It is used so the receiver can tell the sender that it would like to receive more data than what it is currently receiving. Window size is negotiated based on sender and receiver buffers and is negotiated to the lowest value.

Checksum

It is a 16 bits field that is used for integrity checks TCP segment. It is like CRC because TCP doesn't trust the lower layers and checks everything. The Cyclic Redundancy Check (CRC) checks the header and data fields.

Urgent Pointer

It is a 16 bits field that indicates the end of urgent data. This field is used when the URG bit is set in Code bits(flags).

Options

Sets the maximum TCP segment size to either 0 or 32 bits, if any.

Code Bits(Flags)

There are 6 bits in this field with each have a specific purpose. These Bits are used to establish a connection between source and destination hosts before sending the data. Some of the fields are used while sending data and while the connection is terminated with the destination host. Each bit of the code bits is 1 bit long. Below are the details:

URG Bit

URG bit indicates that the Urgent pointer field is significant. When this bit is set, the data should be treated as a priority over other data.

ACK Bit

ACK bit used for the acknowledgment of successful delivery of the previous segment.

PSH Bit

PSH Bit is used for Push function. Updates the receiving host to push the buffered data to the receiving application.

RST Bit

RST bit is used to reset the connection, when the TCP host receives the segment with RST bit set the connection is reset immediately. This bit is used when there are unrecoverable errors and it's not a normal way to finish the TCP connection.

SYN Bit

SYN bit is used during TCP 3 Way handshake process

FIN Bit

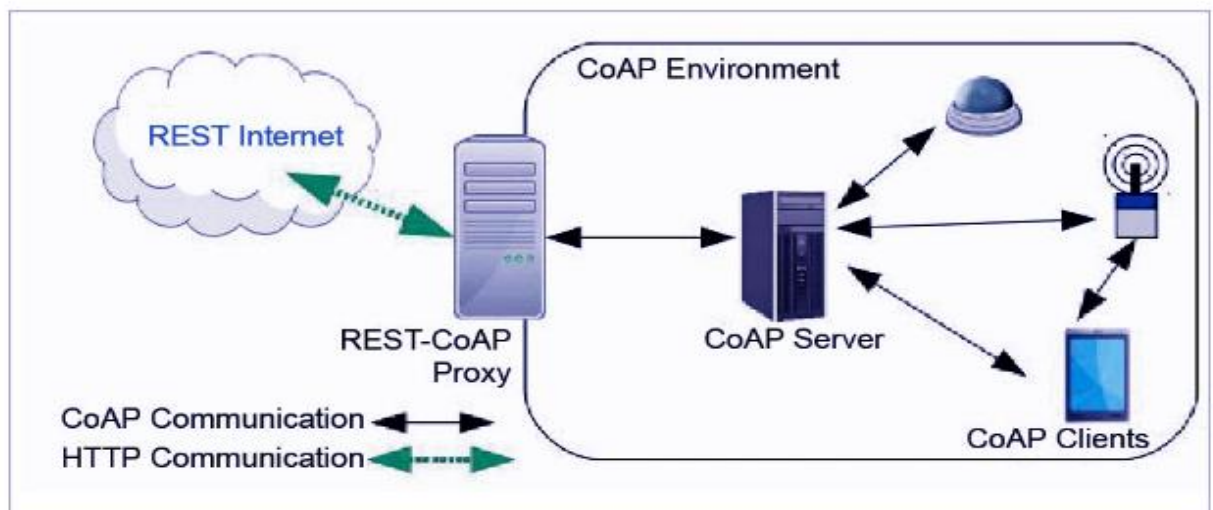
FIN bit is used to finish end the TCP connection in a normal way by both sending and receiving hosts. This bit also specifies end of data.

Disadvantages

- TCP is made for Wide Area Networks, thus its size can become an issue for small networks with low resources
- TCP runs several layers so it can slow down the speed of the network
- It is not generic in nature. Meaning, it cannot represent any protocol stack other than the TCP/IP suite. E.g., it cannot work with a Bluetooth connection.
- No modifications since their development around 30 years ago.

6. Constrained Application Protocol (CoAP):

- CoAP is an internet utility protocol for constrained gadgets. It is designed to enable simple, constrained devices to join IoT through constrained networks having low bandwidth availability.
- This protocol is primarily used for machine-to-machine (M2M) communication and is particularly designed for IoT systems that are based on HTTP protocols. COAP protocol was developed by IETF (Internet Engineering Task Force).

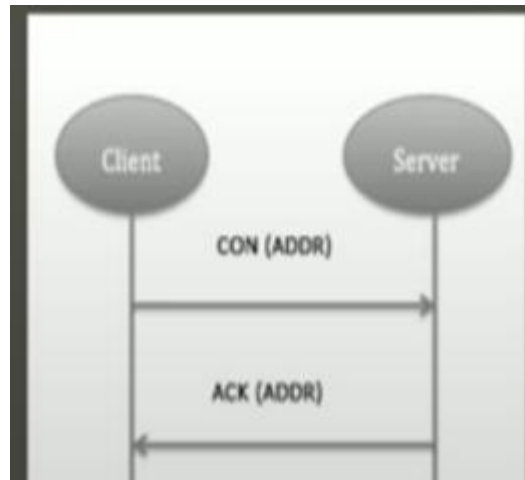


- CoAP is also a RESTful protocol, which reduces the complexity of the implementation and the communication overhead as compared to HTTP.

- Constrained Application Protocol is specifically designed for constrained (limited) Hardware. The hardware that doesn't support HTTP or TCP/IP can use CoAP Protocol. the CoAP protocol using UDP and IP protocol. It is a lightweight protocol that needs low power IOT application like for communication between battery powered IOT devices. Like HTTP, it also follows client-server model. The clients can GET, PUT, DELETE or POST informational resources over the network.
- CoAP makes use of two message types – requests and responses. The messages contain a base header followed by message body whose length is specified by the datagram. The messages or data packets are small in size, so that they can be communicated among constraint devices without data losses.



- Message Model :The CoAP messaging model consists of four types of messages : Confirmable (CON), NonConfirmable (NON), Acknowledgement (ACK) and Reset (RST).
- **The Confirmable message** provides reliability; the message will keep being retransmitted until the sender receive a corresponding Acknowledgement message. If the message does not need reliability, the Non-Confirmable message can be used. Both CON and NON message provide duplication detection by adding a message ID in the header; when the recipient is unable to process the message, it can reply with a Reset message. The request with four basic methods (GET, PUT, POST, DELETE) can be carried by a CON or a NON message.



- CoAP provides three different types of responses. If the request is in a CON message and the resource is immediately available, then a Piggybacked response will be used, which means the response is carried by the ACK message. But when the server needs a longer time to obtain the representation of the resource, and in order to avoid the situation that the client keeps sending the same request, the server may use the Separate response. The server can reply with the ACK as an empty message to inform the client that the request is received.
- When the response is available, the response will be sent back to the client in a CON message. Furthermore, if the request is in NON, the response will be a Non-Confirmable response, a NON message will carry the response back to the client. Caching for requests and responses may be enabled by the CoAP endpoints. In some cases, the request/response pair can be reused by using the prior request/response payload (supposing the client is checking the state of the same resource and its state has not changed); this feature increases performance by reducing response time and reducing network bandwidth consumption.

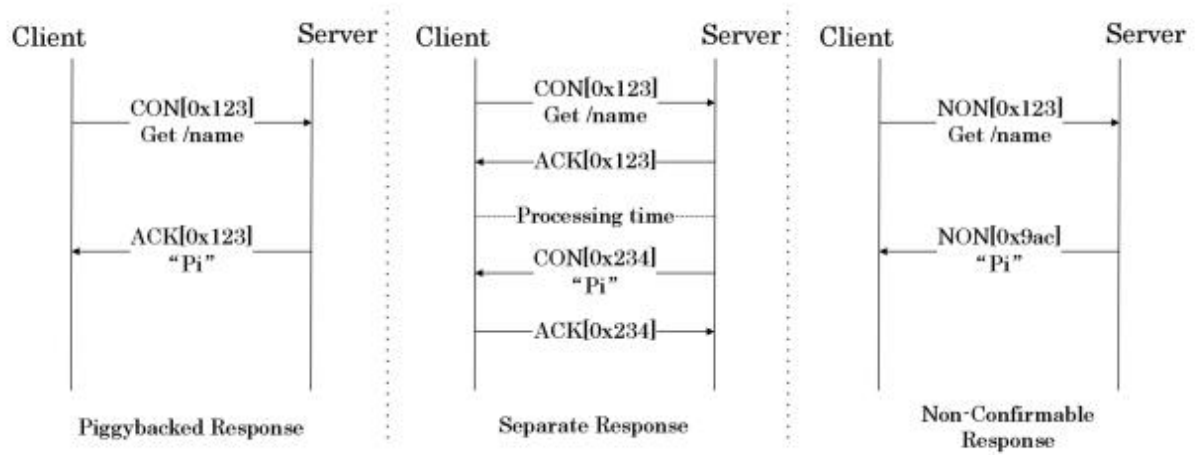


FIGURE 2.10: CoAP request and response (adapted from [3])