IoT unit2 material

Embedded Systems (Jawaharlal Nehru Technological University, Kakinada)

UNIT II: Elements of IoT

IoT Hardware Devices

Hardware and software devices combine to form an IoT ecosystem. Hardware is a set of devices that wire together to serve some functionality. Assume to build a drone, attach sensors to this drone so that it can take photos of your agricultural crops to keep a track of their growth. Or a smart watch that keeps a track of your entire schedule, count the number of steps you take daily, measure your heart pulse. These examples will require connecting small components, tracking the battery usage. IoT devices are a combination of hardware and software. The two components integrated and perform a variety of functions.

Building blocks of IoT Hardware

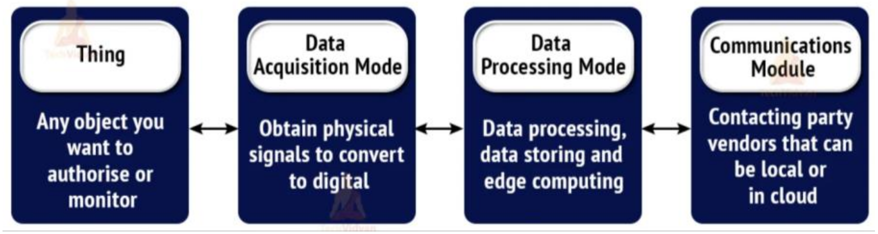

Thing
Any object you want to authorise or monitor
Data Acquisition Mode
Obtain physical signals to convert to digital
Data Processing Mode
Data processing, data storing and edge computing
Communications Module
Contacting party vendors that can be local or in cloud

Figure 1: Building blocks of IoT Hardware

1. “**Things**”: Things in IoT are any devices that are capable of connecting to the internet. They can transmit, retrieve and store data that they collect from the surrounding. They include home appliances such as geysers, microwaves, thermostats and refrigerators, etc.

2. Data Acquisition module: As the term suggests, this module is responsible for acquiring data from the physical surroundings or environment (changes in the temperature, movement, humidity and pressure) and convert the signal to digital from.

3. Data processing module: This module includes computers that process the data acquired from the previous module. They analyse the data, store data for future references and other purposes.

4. Communication module: This is the final building block and this module is responsible for communication with third party vendors. This could include device to device, device to server or device to user.

Types of IoT hardware

It is easier to develop an IoT application these days due to the ease in the availability of boards, Integrated circuits, prototype kits and platforms. These hardware components are low cost and reliable. they offer flexibility and the choice to design custom sensors with specific applications. At the same time you can also specify the networking area, data management and other functionalities as you want. There is a wide range of hardware components to choose from and based on your requirements you can pick the one that matches your proptype perfectly.

a. Microcontrollers

i. Microcontrollers are a type of SoC that provides data processing and storage units. They contain a processor for processing, ROM for storage. When you start building an IoT system, you must pick a microcontroller that fits your desired purpose. You might have to look at its datasheet to understand the properties and specifications.

ii. Microcontrollers have properties such as Datapath Bandwidth. Datapath Bandwidth specifies the number of bits in the registers. More bits, more accurate results.
iii. Microcontrollers connect to all the components of the system and thus they must have sufficient input/output pins. Microcontrollers must have performance depending on what system you are developing.
iv. Microcontrollers use a communication protocol to communicate with one another. The protocols are helpful when you are building bigger systems that require constant communication with other devices.

b. Single-Board Computer(SBC)
i. Single Board Computers (SBC) that contain all the processing and computing properties of a computer on a single board. SBCs have memory units to store code, data, input, output units and microprocessors for computing. It also includes an in-built RAM.
ii. They are a preferred choice in IoT industrial applications as they improve the functionality of a regular computer, they are easily available and reduce the cost of transportation.
iii. Based on the kind of project you are making, you choose a SBC that fits into all your needs for that specific project. SBCs are ready made and available in the market at cheap prices as compared to desktops and computers.
iv. The types of SBCs commonly available in the market are Raspberry Pis, Arduino, Beagleboard and Qualcomm DragonBoard 410c.

IoT Software
A overview of IoT Softwares are-
1. C & C++: The C programming language has its roots in embedded systems—it even got its start for programming telephone switches. It's pretty universal, it can be used almost everywhere and many programmers already know it. C++ is the object-oriented version of C, which is a language popular for both the Linux OS and **Arduino** embedded IoT software systems. These languages were basically written for the hardware systems, which makes them so easy to use.
2. Java: While C and C++ are hardware specific, the code in JAVA is more portable. It is more like a write once and read anywhere you want.
3. Python: There has been a recent surge in the number of python users and has now become one of the "go-to" languages in Web development. It is slowly spreading to the embedded control and IoT world—specially the Raspberry Pi processor. Python is an interpreted language, which is, easy to read, quick to learn and quick to write. Also, it's a powerhouse for serving data-heavy applications.
4. B#: Unlike most of the languages mentioned so far, B# was specifically designed for embedded systems having less memory size.

Arduino
i. Arduino is an open-source hardware and software company that designs and manufactures single-board microcontrollers and microcontroller kits.
ii. In Arduino, each board has clear markings on the connection pins, sockets and in-circuit connections. Thus, Arduino boards are easy to work for DIY (do-it-yourself) and simplify the prototyping of embedded platforms for IoTs.
iii. The Arduino Integrated Development Environment or Arduino Software (IDE) are open source for easy to program.

iv. Uno is most used and documented board of the whole Arduino family at present. The board's analog input pins and PWM pins can connect sensors, actuators and analog circuits. The board's digital I/O pins can connect On-Off states, set of On-Off states, digital inputs from sensors, digital outputs to actuators and other digital circuits.
v. A board with a shield inserted into it makes a wireless connection to a ZigBee, Bluetooth LE, WiFi, GSM, or RF module or a wired connection to Ethernet LAN for the Internet.
vi. Development boards for IoT devices are the Arduino Ethernet, Arduino Wi-Fi and Arduino GSM shields. Development boards for the wearable devices are Arduino Gemma, LilyPad, LilyPad Simple/SimpleSnap and LilyPad USB.
Figure 8.2 shows architecture of Arduino Fio board with Ethernet shield.

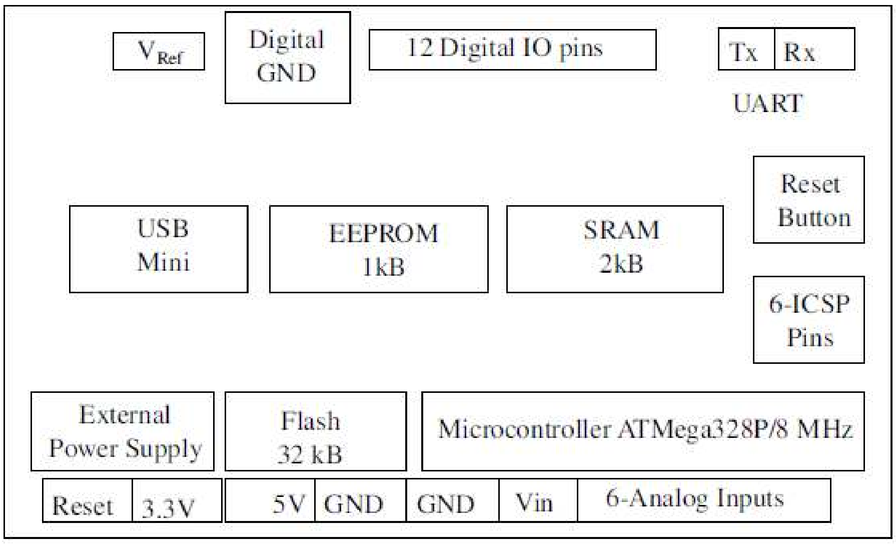


Figure 2: Architecture of Arduino Fio board ford IoT devices development

Arduino types

Arduino Uno (R3), Arduino Nano, Arduino Micro, Arduino Due, LilyPad Arduino Board, Arduino Bluetooth, Arduino Fio, Arduino Diecimila, RedBoard Arduino Board, Arduino Mega (R3) Board Arduino Leonardo Board, Arduino Robot, Arduino Esplora, Arduino Pro Mic Arduino Ethernet, Arduino Zero, Fastest Arduino Board
https://www.elprocus.com/different-types-of-arduino-boards/
https://www.elprocus.com/different-types-of-arduino-boards/

Raspberry Pi

Raspberry Pi is a low-cost mini-computer with the physical size of a credit card. Raspberry Pi runs various flavours of Linux and can perform almost all tasks that a normal desktop computer can do, In addition, Raspberry Pi also allows interfacing sensors and actuators through the general purpose I/O pins. Since Raspberry Pi runs Linux operating system, it supports Python "out of the box".
The different types of raspberry pi models are following
Raspberry Pi 1 model B
Raspberry Pi 1 model A
Raspberry Pi 1 model B+
Raspberry Pi 1model A+

Raspberry Pi Zero
Raspberry Pi 2
Raspberry Pi 3 model B
Raspberry Pi Zero W

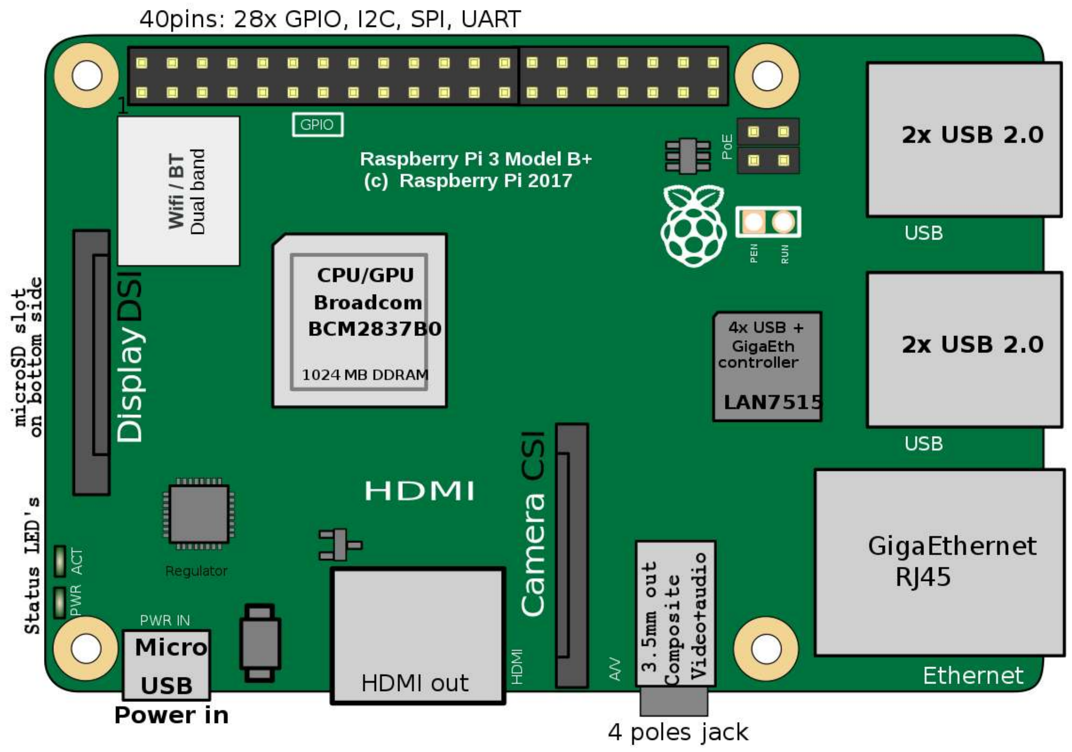


Figure 3: Raspberry Pi Model B+ layout

Types comparison https://www.efxkits.us/different-types-of-raspberry-pi-boards-its-application/
Other information https://en.wikipedia.org/wiki/Raspberry_Pi

i. Raspberry Pi is based on an ARM processor. The latest version of Raspberry Pi (Model B, Revision 2) comes with 700 MHz Low Power ARM 1176JZ-F processor and 512 MB SDRAM,
ii. USB Ports : Raspberry Pi comes with two USB 2.0 ports. The USB ports on Raspberry Pi can provide a current upto 100mA. For connecting devices that draw current more than 100mA an external USB powered hub is required.
iii. Ethernet Ports : Raspberry Pi comes with a standard RJ45 Ethernet port. You can connect an Ethernet cable or a USB Wifi adapter to provide Internet connectivity.
iv. HDMI Output : The HDMI port on Raspberry Pi provides both video and audio output. You can connect the Raspberry Pi to a monitor using an EIDMI cable.
v. Composite Video Output : Raspberry Pi comes with a composite video output.
vi. Audio Output : Raspberry Pi has a 3.5mm audio output jack. The audio quality from this jack is inferior to the HDMI output.
vii. GPIO Pins : Raspberry N comes with a number of general purpose input output pins. There are four types of pins on Raspberry Pi - true GPIO pins. I2C interface pins, SPI interface pins and serial Rx and Tx pins.

a. Serial: The serial interface on Raspberry Pi has receive (Rx) and transmit (Tx) pins for communication with serial peripherals.
b. SPI: Serial Peripheral Interface (SPI) is a synchronous serial data protocol used for communicating with one or more peripheral devices. In an SPI connection, there is one master device and one or more peripheral devices. There are five pins on Raspberry Pi for SPI interface:

- MISO (Master In Slave Out) : Master line for sending data to the peripherals.
- MOSI (Master Out Slave In) : Slave line for sending data to the master.
- SCI (Serial Clock) : Clock generated by master to synchronize data transmission
- CEO (Chip Enable 0) : To enable or disable devices.
- CEO (Chip Enable 1) : To enable or disable devices,

c. I2C: The. 12C interface pins on Raspberry Pi allow you to connect hardware modules. 12C interface allows synchronous data transfer with just two pins - SDA (data line) and SCL (clock line).
viii. Display Serial Interface (1351) The DSI interface can be used to connect an LCD panel to Raspberry Pi.
ix. Camera Serial interface (CSI) : The CSI interface can be used to connect a camera module to Raspberry Pi.
x. Status LEDs : Raspberry Pi has live status LEDs, Table 7.1 lists Raspberry Pi status LEDs and their functions.
xi. SD Card Slot : Raspberry Pi does not have a built in operating system and storage. You can plug-in an SD card loaded with a Linux image to the SD card slot. Appendix-A provides instructions on setting up New Out-of-the-Box Software (NOOBS) on Raspberry Pi. You will require at least an 80H SD card for setting up NOOBS,
xii. Power Input : Raspberry Pi has a micro-USB connector for power input.

ARM Cortex-M class processor

Over the years, ARM has developed quite a number of different processor products. In the following diagram (Figure 4), the ARM processors are divided between the classic ARM processors and the newer Cortex processor product range. In addition, these processors are divided into three groups:

Application Processors – High-end processors for mobile computing, smart phone, servers, etc. These processors run at higher clock frequency (over 1GHz), and support Memory Management Unit (MMU), which is required for full feature OS such as Linux, Android, MS Windows and mobile OSs. If you are planning to develop a product that requires one of these OSs, you need to use an application processor.

Real-time Processors – These are very high-performance processors for real-time applications such as hard disk controller, automotive power train and base band control in wireless communications. Most of these processors do not have MMU, and usually have Memory Protection Unit (MPU), cache, and other memory features designed for industrial applications. They can run at a fairly high clock frequency (e.g. 200MHz to >1GHz) and have very low response latency. Although these processors cannot run full versions of Linux or Windows, there are plenty of Real Time Operating Systems (RTOS) that can be used with these processors.

Microcontroller Processors – These processors are usually designed to have a much lower silicon area, and much high-energy efficiency. Typically, they have shorter pipeline, and usually lower maximum frequency (although you can find some of these processors running at over 200MHz). At the same time, the newer Cortex-M processor family is designed to be very easy to use; therefore, they are very popular in the microcontroller and deeply embedded systems market.

Today, there are eight members in the ARM Cortex-M processor family. Different processors can have different instruction set support, system features and performance.

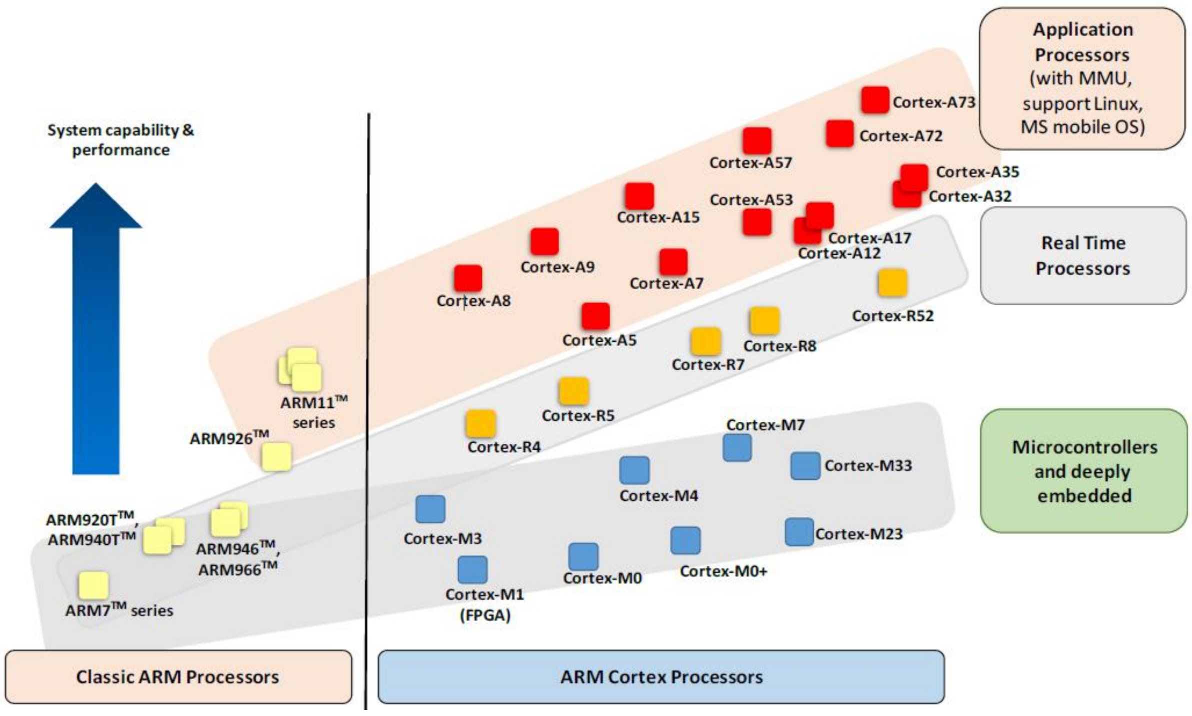


Figure 4: ARM processor family

The Cortex-M processor family

The Cortex-M processor family is more focused on the lower end of the performance scale. However, these processors are still quite powerful when compared to other typical processors used in most microcontrollers. For example, the Cortex-M4 and Cortex-M7 processors are being used in many high-performance microcontroller products, with maximum clock frequency going up to 400MHz. Of course, performance is not the only factor when selecting a processor. In many applications, low power and cost are the key selection criteria. Therefore, the Cortex-M processor family contains various products to address different needs:

Cortex-M0 A very small processor (starting from 12K gates) for low cost, ultra-low power microcontrollers and deeply embedded applications.
Cortex-M0+ The most energy-efficient processor for small embedded system. Similar size and programmer's model to the Cortex-M0 processor, but with additional features like single cycle I/O interface and vector table relocations.

Cortex-M1 A small processor design optimized for FPGA designs and provides Tightly Coupled Memory (TCM) implementation using memory blocks on the FPGAs. Same instruction set as the Cortex-M0.
Cortex-M3 A small but powerful embedded processor for low-power microcontrollers that has a rich instruction set to enable it to handle complex tasks quicker. It has a hardware divider and Multiply-Accumulate (MAC) instructions. In addition, it also has comprehensive debug and trace features to enable software developers to develop their applications quicker.
Cortex-M4 It provides all the features on the Cortex-M3, with additional instructions target at Digital Signal Processing (DSP) tasks, such as Single Instruction Multiple Data (SIMD) and faster single cycle MAC operations. In addition, it also have an optional single precision floating point unit that support IEEE 754 floating point standard.
Cortex-M7 High-performance processor for high-end microcontrollers and processing intensive applications. It has all the ISA features available in Cortex-M4, with additional support for double-precision floating point, as well as additional memory features like cache and Tightly Coupled Memory (TCM).

Arm Cortex-M0 Processor Architecture

The ARMv6-M architecture that the Cortex-M0 processor implemented covers a number of different areas. To use a Cortex-M0 device with C language, you only need to know the memory map, the peripheral programming information, the exception handling mechanism, and part of the programmer's model. Most users of the Cortex-M0 processor will work in C language; as a result, the underlying programmer's model will not be visible in the program code. However, it is still useful to know about the details, as this information is often needed during debugging and it will also help readers to understand the rest of this book.

Programmer's Model

1. Operation Modes and States

i. The Cortex-M0 processor has two operation modes (Thread mode or the Handler mode) and two states (Thumb and Deburg stated) as in the Figure 5.

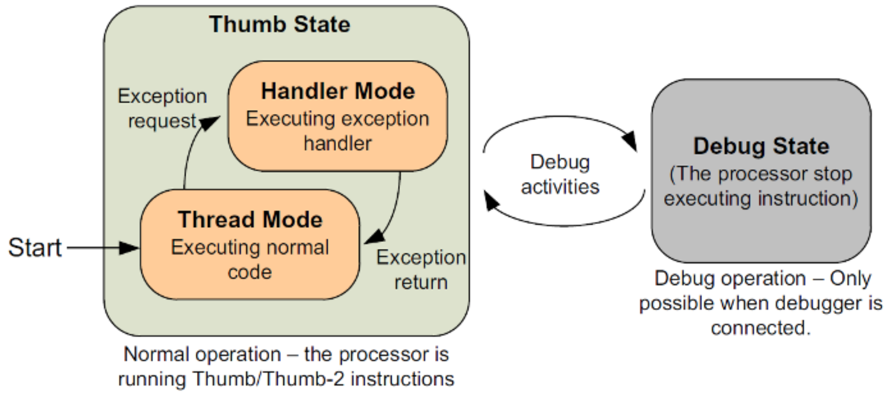

Thumb State
Exception request
Handler Mode
Executing exception handler
Start
Thread Mode
Executing normal code
Exception return
Normal operation – the processor is running Thumb/Thumb-2 instructions
Debug activities
Debug State
(The processor stop executing instruction)
Debug operation – Only possible when debugger is connected.

Figure 5: Processor modes and states in the Cortex-M0 processor

ii. 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. Both modes are almost the same. The only difference is Thread mode have a special register called CONTROL.

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

2. Registers and Special Registers

i. To perform data processing and controls, a number of registers are required inside the processor core. The data have to be loaded from the memory to a register in the register bank then processed inside the processor, and then written back to the memory if needed. This is commonly called as "load-store architecture."
ii. By having a sufficient number of registers in the register bank, this mechanism is easy to use. The register bank contains sixteen 32-bit registers. 13 are general-purpose registers, remaining have special uses as shown in the Figure 6.

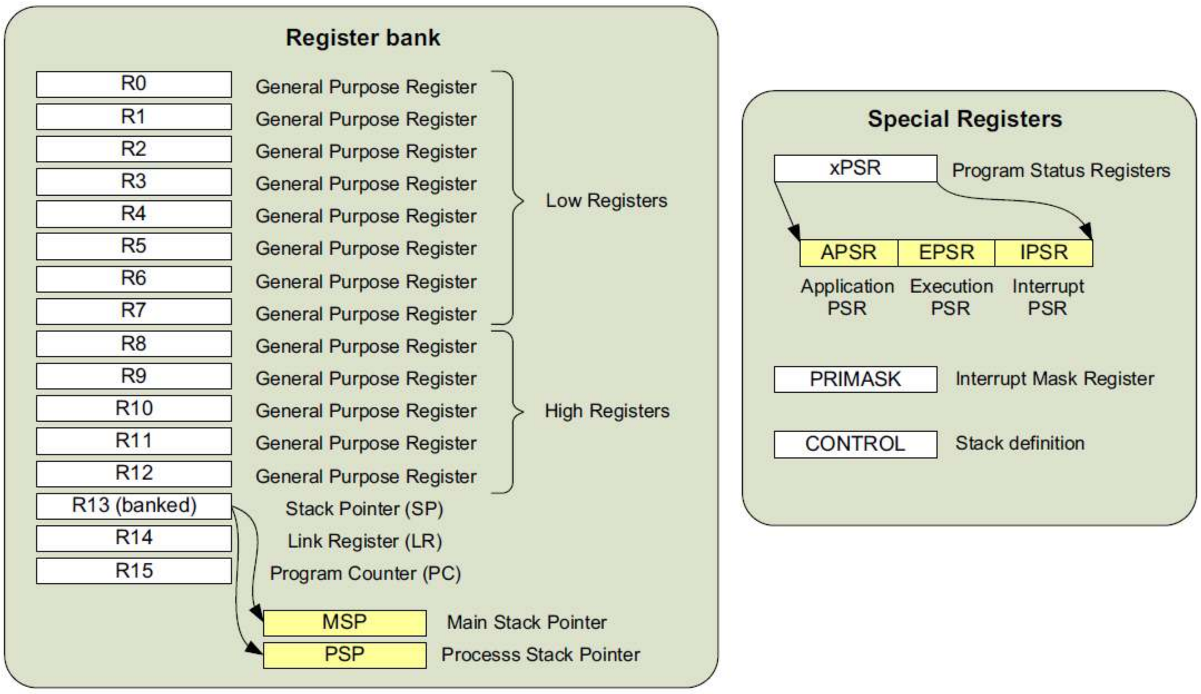


Figure 6: Registers in the Cortex-M0 processor

R0-R12: Registers R0 to R12 are for general uses. The Thumb instructions can only access low registers (R0 to R7). Some instructions like MOV (move) can use all registers. The initial values of R0 to R12 at reset are undefined.
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 used for running unusual handlers mode and process stack pointer (PSP) is used for usual Thread mode.
R14, Link Register (LR): R14 is the Link Register. The Link Register is used for storing the return address of a function call.

R15, Program Counter (PC): R15 is the Program Counter. It is readable and writeable. Call the Program Counter, using either "R15" or "PC," in either upper or lower case (e.g., "r15" or "pc").

xPSR, combined Program Status Register

The combined Program Status Register provides information about program execution and the ALU flags. It is consists of the following three Program Status Registers (PSRs) as in Figure 7:

• Application PSR (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.

• Interrupt PSR (IPSR): Contains the current executing interrupt service routine (ISR) number.

• Execution PSR (EPSR): Contains the T-bit, which indicates that the processor is in the Thumb state.

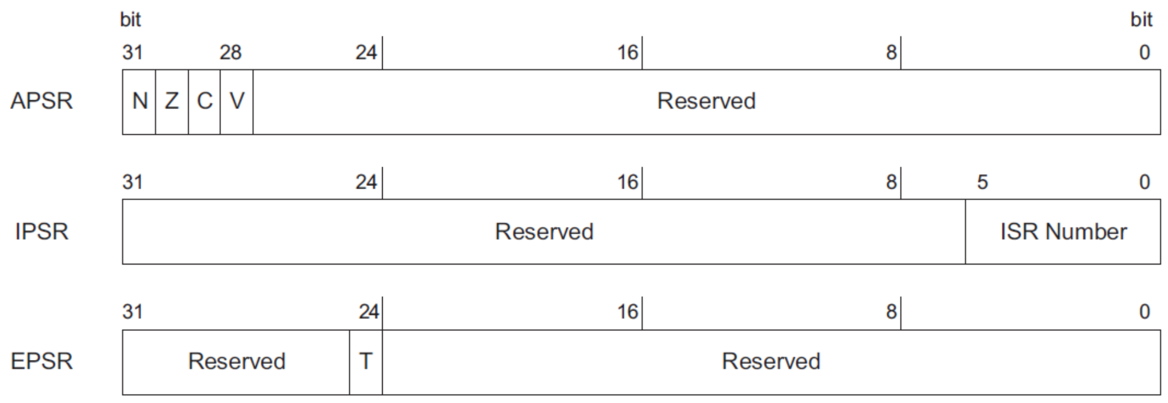


Figure 7: APSR, IPSR, and EPSR.

These three registers can be accessed as one register called xPSR as given in Figure 8.

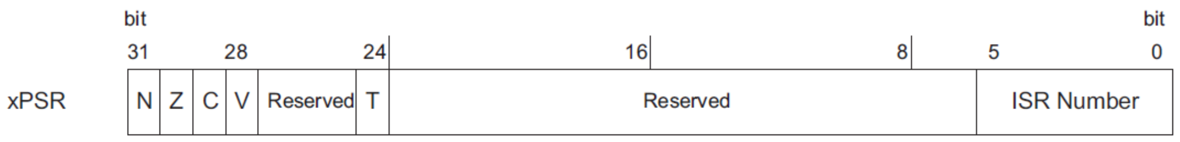


Figure 8: xPSR

PRIMASK: Interrupt Mask Special Register

The PRIMASK register is a 1-bit-wide interrupt mask register as in Figure 9. When set, it blocks all interrupts apart from the non-maskable 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. The PRIMASK register can be accessed using special register access instructions (MSR, MRS) as well as using an instruction called the Change Processor State (CPS). This is commonly used for handling time-critical routines.

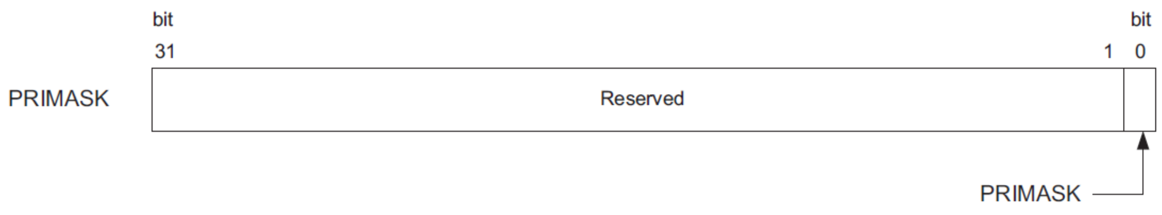


Figure 9: PRIMASK

CONTROL: Special Register

i. 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 as shown in the Figure 10.

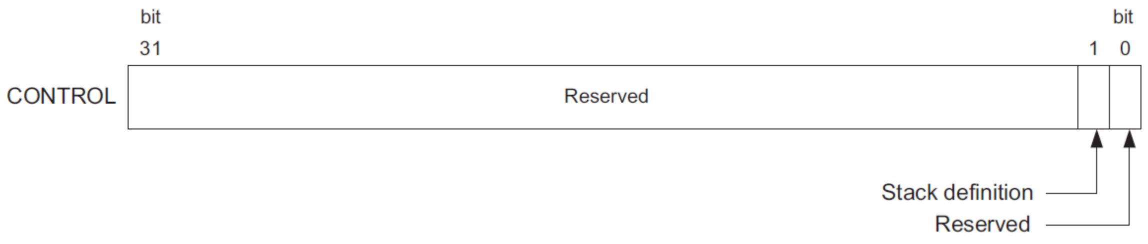


Figure 10: CONTROL

ii. After reset, the main stack pointer (MSP) is used, but can be switched to the process stack pointer (PSP) in Thread mode by setting bit [1] in the CONTROL register as shown in the Figure 11.

iii. 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.

iv. Bit 0 of the CONTROL register is reserved to maintain compatibility with the Cortex-M3 processor.

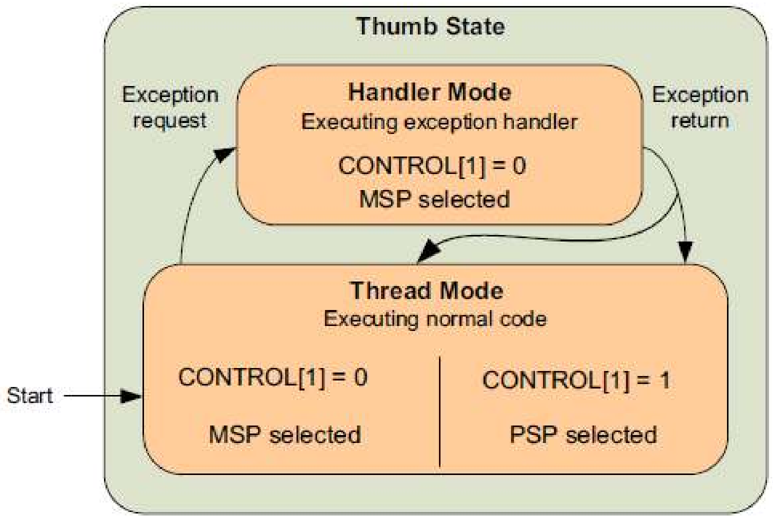
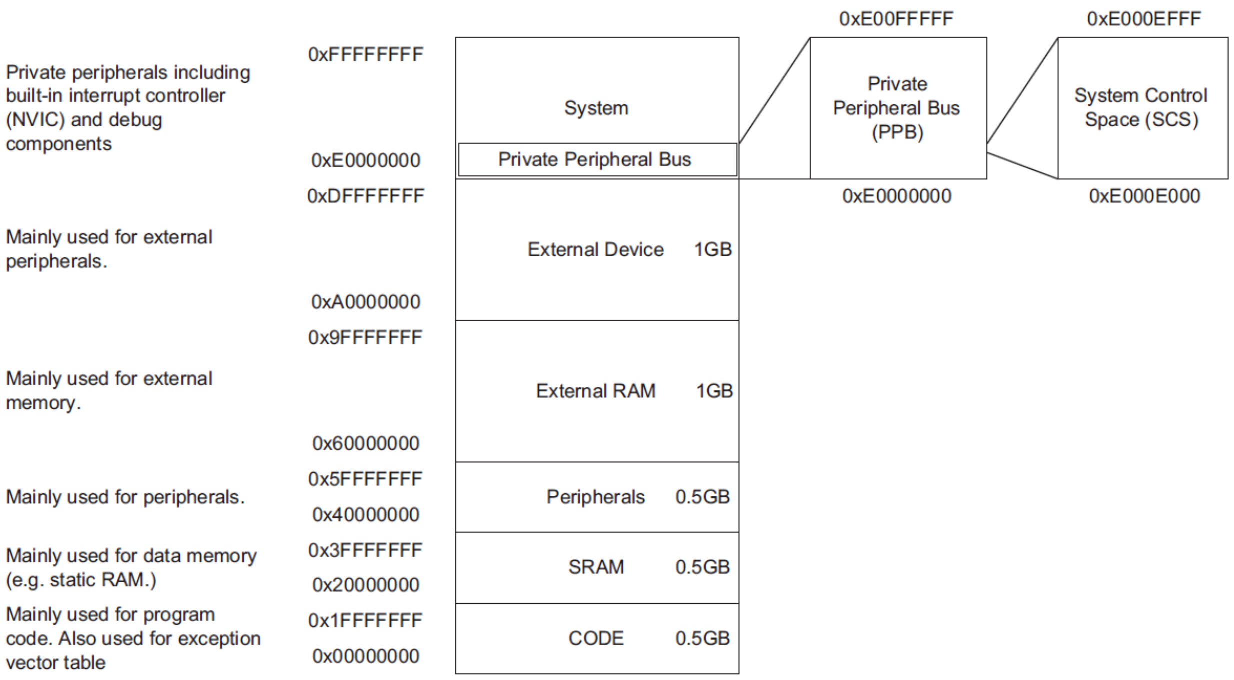


Figure 11: Stack pointer selection

3. Memory System Overview

The Cortex-M0 processor has 4 GB of memory address space. 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 as shown in the Figure 12.

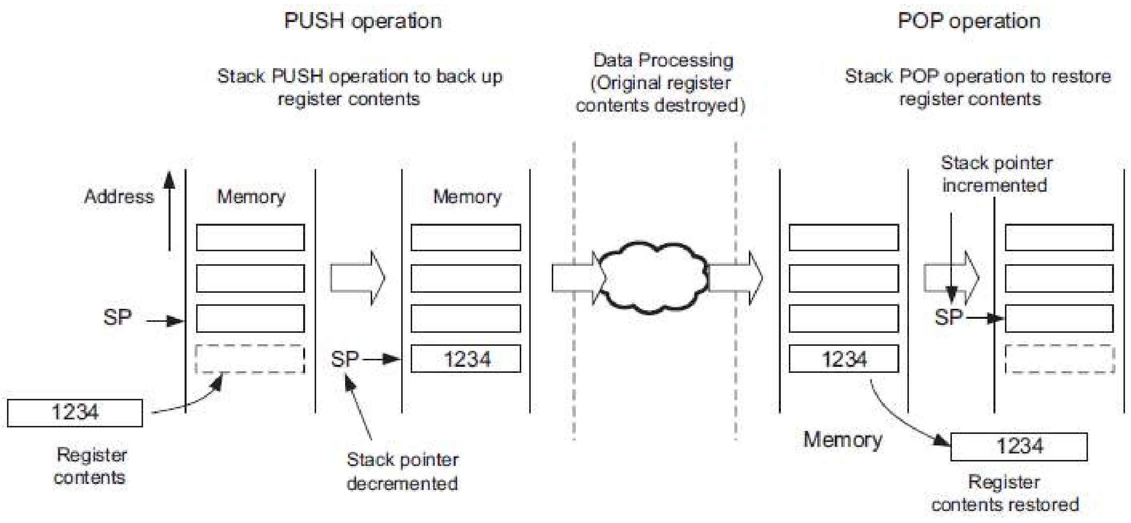


Private peripherals including built-in interrupt controller (NVIC) and debug components
Mainly used for external peripherals.
Mainly used for external memory.
Mainly used for peripherals.
Mainly used for data memory (e.g. static RAM.)
Mainly used for program code. Also used for exception vector table
0xFFFFFFFF
0xE0000000
0xDFFFFFFF
0xA0000000
0x9FFFFFFF
0x60000000
0x5FFFFFFF
0x40000000
0x3FFFFFFF
0x20000000
0x1FFFFFFF
0x00000000
System
Private Peripheral Bus
External Device 1GB
External RAM 1GB
Peripherals 0.5GB
SRAM 0.5GB
CODE 0.5GB
0xE00FFFFF
Private Peripheral Bus (PPB)
0xE0000000
0xE000EFFF
System Control Space (SCS)
0xE000E000

Figure 12: Memory map

Stack Memory Operations

Stack memory is a memory usage mechanism that allows the system memory to be used as temporary data storage. The main element 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 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) as shown in the Figure 13.



PUSH operation
Stack PUSH operation to back up register contents
Data Processing (Original register contents destroyed)
POP operation
Stack POP operation to restore register contents
Address
Memory
Memory
SP
SP
1234
1234
Register contents
Stack pointer decremented
Stack pointer incremented
SP
1234
Memory
1234
Register contents restored

Figure 13: Stack PUSH and POP in the Cortex-M0 processor.

4. Exceptions and Interrupts

i. Exceptions are events that cause change to program control: instead of continuing program execution, the processor suspends the current executing task and executes a part of the

program code called the exception handler. After the exception handler is completed, it will then resume the normal program execution.
ii. There are various types of exceptions. Interrupts are a subset of exceptions. They are 32 external interrupts (commonly referred as interrupt request, IRQs) and an additional special interrupt called the nonmaskable interrupt (NMI).
iii. The exception handlers for interrupt events are commonly known as interrupt service routines (ISRs).

Nested Vectored Interrupt Controller (NVIC)
To prioritize the interrupt requests and handle other exceptions, the Cortex-M0 processor has a built-in interrupt controller called the Nested Vectored Interrupt Controller (NVIC). The interrupt management function is controlled by a number of programmable registers in the NVIC. These registers are memory mapped, with the addresses located within the System Control Space (SCS) as illustrated in Figure 12.
The NVIC supports a number of features:
• Flexible interrupt management: each external interrupt can be enabled or disabled. It can also accept exception requests at external peripheral, at 1cycle.
• Nested interrupt support: each exception has a priority level. The priority level can be fixed or programmable.
• Vectored exception entry: When an exception occurs, the processor will need to locate the starting point of the corresponding exception handler.
• Interrupt masking

Interrupt Masking
The NVIC in the Cortex-M0 processor provides an interrupt masking feature via the PRIMASK special register. This can disable all exceptions except hard fault and NMI (non-maskable interrupt, ~~hardware failure~~). This masking is useful for operations that should not be interrupted such as time critical control tasks.

5. System Control Block (SCB)
Apart from the NVIC, the System Control Space (SCS) also contains a number of other registers for system management. This is called the System Control Block (SCB). It contains registers for sleep mode features and system exception configurations, as well as a register containing the processor identification code.

Block Diagram
i. A simplified block diagram of the Cortex-M0 is shown in Figure 14. 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.
ii. The Nested Vectored Interrupt Controller (NVIC) accepts up to 32 interrupt request signals and a non-maskable interrupt (NMI) input.
iii. It contains the functionality required for comparing priority between interrupt requests and the current priority level so that nested interrupts can be handled automatically.

iv. 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.
iv. 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.
v. 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.
vi. The serial wire protocol is a newer communication protocol that only requires two wires, but it can handle the same debug functionalities as JTAG (Joint Test Action Group). The internal bus system, the data path in the processor core, and the AHB LITE bus interface are all 32 bits wide.
vii. 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.

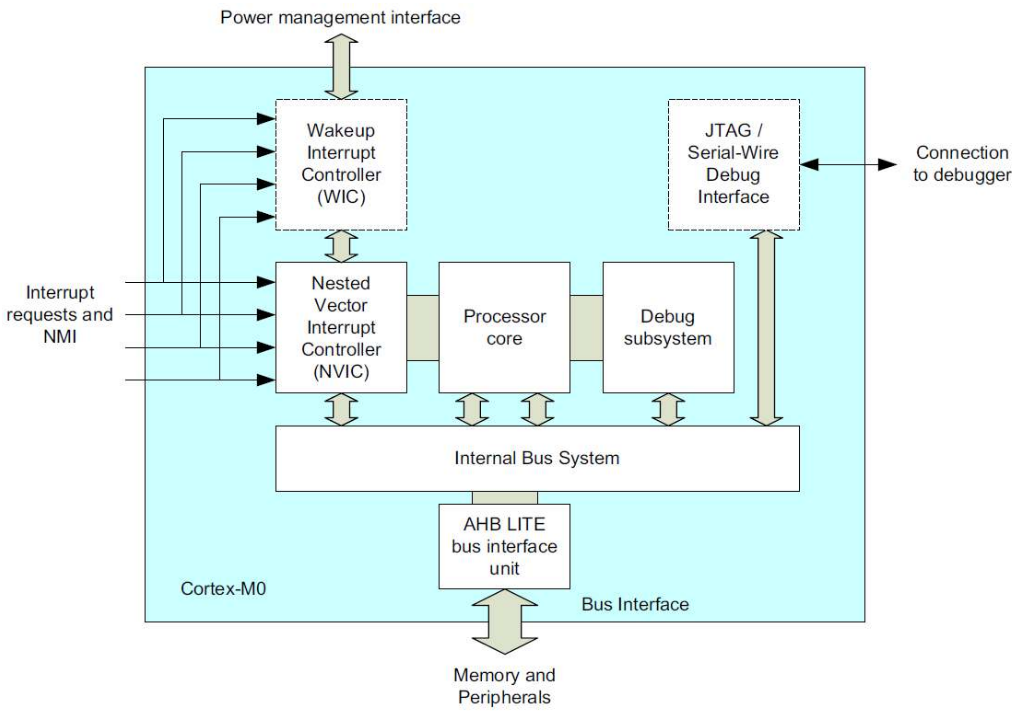

Power management interface
Wakeup Interrupt Controller (WIC)
JTAG / Serial-Wire Debug Interface
Connection to debugger
Interrupt requests and NMI
Nested Vector Interrupt Controller (NVIC)
Processor core
Debug subsystem
Internal Bus System
AHB LITE bus interface unit
Cortex-M0
Bus Interface
Memory and Peripherals

Figure 14: Simplified block diagram of the Cortex-M0 processor

CHAPTER 5

Instruction Set

Background of ARM and Thumb Instruction Set

The early ARM processors use a 32-bit instruction set called the ARM instructions. The 32-bit ARM instruction set is powerful and provides good performance, but at the same time it often requires larger program memory when compared to 8-bit and 16-bit processors. This was and still is an issue, as memory is expensive and could consume a considerable amount of power.

In 1995, ARM introduced the ARM7TDMI processor, adding a new 16-bit instruction set called the Thumb instruction set. The ARM7TDMI supports both ARM instructions and Thumb instructions, and a 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. Nevertheless, most processing can be carried out using Thumb instructions and interrupt handlers could switch themselves to use the Thumb state, so the ARM7TDMI processor provides excellent code density when compared to other 32-bit RISC architectures.

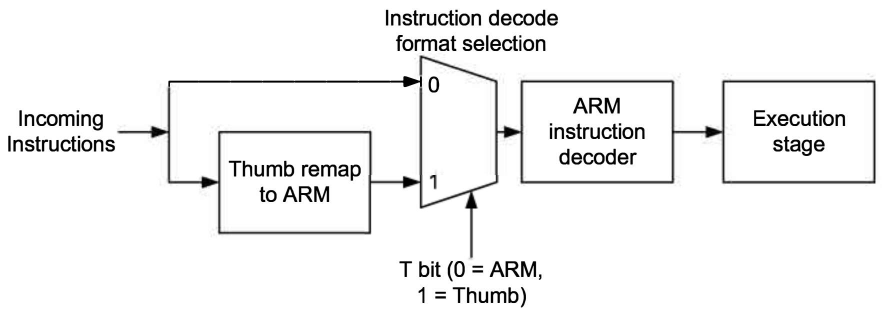


Figure 5.1:
ARM7TDMI design supports both ARM and the Thumb instruction set.

Thumb code provides a code size reduction of approximately 30% compared to the equivalent ARM code. However, it has some impact on the performance and can reduce the performance by 20%. On the other hand, in many applications, the reduction of program memory size and

The Definitive Guide to the ARM Cortex-M0. DOI: 10.1016/B978-0-12-385477-3.10005-9

the low-power nature of the ARM7TDMI processor made it extremely popular with portable electronic devices like mobile phones and microcontrollers.

In 2003, ARM introduced Thumb-2 technology. This technology provides a number of 32-bit Thumb instructions as well as the original 16-bit Thumb instructions. The new 32-bit Thumb instructions can carry out most operations that previously could only be done with the ARM instruction set. As a result, program code compiled for Thumb-2 is typically 74% of the size of the same code compiled for ARM, but it maintains similar performance.

The Cortex-M3 processor is the first ARM processor that supports only Thumb-2 instructions. It can deliver up to 1.25 DMIPS per MHz (measured with Dhrystone 2.1), and various microcontroller vendors are already shipping microcontroller products based on the Cortex-M3 processor. By implementing only one instruction set, the software development is made simpler and at the same time improves the energy efficiency because only one instruction decoder is required (Figure 5.2).

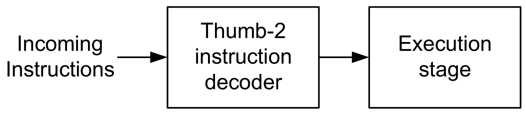


Figure 5.2:
Cortex-M processors do not have to remap instructions from Thumb to ARM.

In the ARMv6-M architecture used in the Cortex-M0 processor, in order to reduce the circuit size to a minimum, only the 16-bit Thumb instructions and a minimum subset of 32-bit Thumb instructions are supported. These 32-bit Thumb instructions are essential because the ARMv6-M architecture uses a number of features in the ARMv7-M architecture, which requires these instructions. For example, the accesses to the special registers require the MSR and MRS instructions. In addition, the Thumb-2 version of Branch and Link instruction (BL) is also included to provide a larger branch range.

Although the Cortex-M0 processor does not support many 32-bit Thumb instructions, the Thumb instruction set used in the Cortex-M0 processor is a superset of the original 16-bit Thumb instructions supported on the ARM7TDMI, which is based on ARMv4T architecture. Over the years, both ARM and Thumb instructions have gone through a number of enhancements as the architecture has evolved. For example, a number of instructions for data type conversions have been added to the Thumb instruction set for the ARMv6 and ARMv6-M architectures. These instruction set enhancements, along with various implementation optimizations, allow the Cortex-M0 processor to deliver the same level of performance as an ARM7TDMI running ARM instructions.

Table 5.1 shows the base 16-bit Thumb instructions supported in the Cortex-M0.

Table 5.1: 16-Bit Thumb Instructions Supported on the Cortex-M0 Processor

16-Bit Thumb Instructions Supported on Cortex-M0									
ADC	ADD	ADR	AND	ASR	B	BIC	BLX	BKPT	BX
CMN	CMP	CPS	EOR	LDM	LDR	LDRH	LDRSH	LDRB	LDRSB
LSL	LSR	MOV	MVN	MUL	NOP	ORR	POP	PUSH	REV
REV16	REVSH	ROR	RSB	SBC	SEV	STM	STR	STRH	STRB
SUB	SVC	SXTB	SXTH	TST	UXTB	UXTH	WFE	WFI	YIELD

The Cortex-M0 processor also supports a number of 32-bit Thumb instructions from Thumb-2 technology (Table 5.2):

- MRS and MSR special register access instructions
- ISB, DSB, and DMB memory synchronization instructions
- BL instruction (BL was supported in traditional Thumb instruction set, but the bit field definition was extended in Thumb-2)

Table 5.2: 32-Bit Thumb Instructions Supported on the Cortex-M0 Processor

32-Bit Thumb Instructions Supported on Cortex-M0					
BL	DSB	DMB	ISB	MRS	MSR

Assembly Basics

This chapter introduces the instruction set of the Cortex-M0 processor. In most situations, application code can be written entirely in C language and therefore it is not necessary to know the details of the instruction set. However, it is still useful to know what instructions are available and their usages; for example, this information might be needed during debugging.

The complete details of each instruction are documented in the ARMv6-M Architecture Reference Manual (reference 3). Here, the basic syntax and usage are introduced. First of all, to help explain the assembly instructions covered in this chapter, some of the basic information about assembly syntax is introduced here.

Quick Glance at Assembly Syntax

Most of the assembly examples in this book are written for the ARM assembler (armasm). Assembly tools from different vendors (e.g., GNU tool chain) have different assembly syntax. In most cases, the mnemonics of the assembly instructions are the same, but compile directives, definitions, labeling, and comment syntax can be different.

Most development tools still accept the pre-UAL syntax, including the ARM RealView Development Suite (RVDS) and the Keil Microcontroller Development Kit for ARM (MDK). However, the use of UAL is recommended for new projects. For assembly development with RVDS or Keil MDK, you can specify using UAL syntax with "THUMB" directives and pre-UAL syntax with "CODE16" directives. The choice of assembler syntax depends on which tool you use. Please refer to the documentation for your development suite to determine the suitable syntax.

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

In this section, the instructions are discussed in more detail. The syntax illustrated here uses symbols "Rd," "Rm," and the like. In real program code, these need to be substituted with register names R0, R1, R2, and so on.

Moving Data within the Processor

Transferring data is one of the most common tasks in a processor. In Thumb code, the instruction mnemonic for moving data is MOV. There are several types of MOV instructions, based on the operand type and opcode suffix.

Instruction	**MOV**
Function	Move register into register
Syntax (UAL)	MOV <Rd>, <Rm>
Syntax (Thumb)	MOV <Rd>, <Rm>
	CPY <Rd>, <Rm>
Note	Rm and Rn can be high or low registers
	CPY is a pre-UAL synonym for MOV (register)

If we want to copy a register value to another and update the APSR at the same time, we could use MOVS/ADDS.

Instruction	MOVS/ADDS
Function	Move register into register
Syntax (UAL)	MOVS <Rd>, <Rm>
	ADDS <Rd>, <Rm>, #0
Syntax (Thumb)	MOVS <Rd>, <Rm>
Note	Rm and Rn are both low registers
	APSR.Z, APSR.N, and APSR.C (for ADDS) update

We can also load an immediate data element into a register using the MOV instruction.

Instruction	MOV
Function	Move immediate data (sign extended) into register
Syntax (UAL)	MOVS <Rd>, #immed8
Syntax (Thumb)	MOV <Rd>, #immed8
Note	Immediate data range 0 to +255
	APSR.Z and APSR.N update

If we want to load an immediate data element into a register that is out of the 8-bit value range, we need to store the data into a program memory space and then use a memory access instruction to read the data into the register. This can be written using a pseudo instruction LDR, which the assembler converts into a real instruction. This process will be covered later in this chapter.

The MOV instructions can cause a branch to happen if the destination register is R15 (PC). However, generally the BX instruction is used for this purpose.

Another type of data transfer in the Cortex-M0 processor is Special Registers accesses. To access the Special Registers (CONTROL, PRIMASK, xPSR, etc.), the MRS and MSR instructions are needed. These two instructions cannot be generated in C language. However, they can be created using inline assembler or Embedded Assembler,[3] or another C compiler–specific feature like the named register variables feature in ARM RVDS or Keil MDK.

Instruction	MRS
Function	Move Special Register into register
Syntax	MRS <Rd>, <SpecialReg>
Note	Example:
	MRS R0, CONTROL; Read CONTROL register into R0
	MRS R9, PRIMASK; Read PRIMASK register into R9
	MRS R3, xPSR; Read xPSR register into R3

Table 5.5 shows the complete list of special register symbols that are available on the Cortex-M0 processor when MSR and MRS instructions are used.

Table 5.5: Special Register Symbols for MRS and MSR Instructions

Symbol	Register	Access Type
APSR	Application Program Status Register (PSR)	Read/Write
EPSR	Execution PSR	Read only
IPSR	Interrupt PSR	Read only
IAPSR	Composition of IPSR and APSR	Read only
EAPSR	Composition of EPSR and APSR	Read only
IEPSR	Composition of IPSR and EPSR	Read only
XPSR	Composition of APSR, EPSR, and IPSR	Read only
MSP	Main stack pointer	Read/Write
PSP	Process stack pointer	Read/Write
PRIMASK	Primary exception mask register	Read/Write
CONTROL	CONTROL register	Read/Write in Thread mode Read only in Handler mode

Instruction	MSR
Function	Move register into Special Register
Syntax	MSR <SpecialReg>, <Rd>
Note	Example: MSR CONTROL, R0; Write R0 into CONTROL register MSR PRIMASK, R9; Write R9 into PRIMASK register

Memory Accesses

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, Half Word and Byte. In addition, there are separate instructions to support signed and unsigned data. Table 5.6 summarizes the memory address instruction mnemonics.

Most of these instructions also support multiple addressing modes. When the instruction is used with different operands, the assembler will generate different instruction encoding.

Table 5.6: Memory Access Instructions for Various Transfer Sizes

Transfer Size	Unsigned Load	Signed Load	Signed/Unsigned Store
Word	LDR	LDR	STR
Half word	LDRH	LDRSH	STRH
Byte	LDRB	LDRSB	STRB

Important
It is important to make sure the memory address accessed is aligned. For example, a word size access can only be carried out on address locations when address bits[1:0] are set to zero, and a half word size access can only be carried out on address locations when an address bit[0] is set to zero. The Cortex-M0 processor does not support unaligned transfers. Any attempt at unaligned memory access results in a hard fault exception. Byte-size transfers are always aligned on the Cortex-M0 processor.

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

Instruction	LDR/LDRH/LDRB
Function	Read single memory data into register
Syntax	LDR <Rt>, [<Rn>, <Rm>] ; Word read
	LDRH <Rt>, [<Rn>, <Rm>] ; Half Word read
	LDRB <Rt>, [<Rn>, <Rm>] ; Byte read
Note	Rt = memory[Rn + Rm]
	Rt, Rn and Rm are low registers

The Cortex-M0 processor also supports immediate offset addressing modes:

Instruction	LDR/LDRH/LDRB
Function	Read single memory data into register
Syntax	LDR <Rt>, [<Rn>, #immed5] ; Word read
	LDRH <Rt>, [<Rn>, #immed5] ; Half Word read
	LDRB <Rt>, [<Rn>, #immed5] ; Byte read
Note	Rt = memory[Rn + ZeroExtend (#immed5 << 2)] ; Word
	Rt = memory[Rn + ZeroExtend(#immed5 << 1)] ; Half word
	Rt = memory[Rn + ZeroExtend(#immed5)] ; Byte
	Rt and Rn are low registers

The Cortex-M0 processor supports a useful PC relative load instruction for allowing efficient literal data accesses. This instruction can be generated when we use the LDR pseudo instruction for putting an immediate data value into a register. These data are stored alongside the instructions, called literal pools.

Instruction	LDR
Function	Read single memory data word into register
Syntax	LDR <Rt>, [PC, #immed8] ; Word read
Note	Rt = memory[WordAligned(PC+4) + ZeroExtend(#immed8 << 2)]
	Rt is a low register, and targeted address must be a word-aligned address, the reason for adding 4.

(*Continued*)

Instruction	LDR
Example: LDR R0,=0x12345678 ; A pseudo instruction that uses literal load ; to put an immediate data into a register LDR R0, [PC, #0x40] ; Load a data in current program address ; with offset of 0x40 into R0 LDR R0, label ; Load a data in current program ; referenced by label into R0	

There is also an SP-related load instruction, which supports a wider offset range. This instruction is useful for accessing local variables in C functions because often the local variables are stored on the stack.

Instruction	LDR
Function	Read single memory data word into register
Syntax	LDR <Rt>, [SP, #immed8] ; Word read
Note	Rt = memory[SP + ZeroExtend(#immed8 << 2)] Rt is a low register

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

Instruction	LDRSH/LDRSB
Function	Read single signed memory data into register
Syntax	LDRSH <Rt>, [<Rn>, <Rm>] ; Half word read LDRSB <Rt>, [<Rn>, <Rm>] ; Byte read
Note	Rt = SignExtend(memory[Rn + Rm]) Rt, Rn and Rm are low registers

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

Instruction	STR/STRH/STRB
Function	Write single register data into memory
Syntax	STR <Rt>, [<Rn>, <Rm>] ; Word write STRH <Rt>, [<Rn>, <Rm>] ; Half Word write STRB <Rt>, [<Rn>, <Rm>] ; Byte write
Note	memory[Rn + Rm] = Rt Rt, Rn and Rm are low registers

Like the load operation, the store operation supports an immediate offset addressing mode:

Instruction	STR/STRH/STRB
Function	Write single memory data into memory
Syntax	STR <Rt>, [<Rn>, #immed5] ; Word write
	STRH <Rt>, [<Rn>, #immed5] ; Half Word write
	STRB <Rt>, [<Rn>, #immed5] ; Byte write
Note	memory[Rn + ZeroExtend(#immed5 << 2)] = Rt ; Word
	memory[Rn + ZeroExtend(#immed5 << 1)] = Rt ; Half word
	memory[Rn + ZeroExtend(#immed5)] = Rt ; Byte
	Rt and Rn are low registers

An SP-relative store instruction, which supports a wider offset range, is also available. This instruction is useful for accessing local variables in C functions because often the local variables are stored on the stack.

Instruction	STR
Function	Write single memory data word into memory
Syntax	STR <Rt>, [SP, #immed8] ; Word write
Note	memory[SP + ZeroExtend(#immed8 << 2)] = Rt
	Rt is a low register

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.

Instruction	LDM (Load Multiple)
Function	Read multiple memory data word into registers, base address register update by memory read
Syntax	LDM <Rn>, {<Ra>, <Rb> ,....} ; Load multiple registers from memory
Note	Ra = memory[Rn],
	Rb = memory[Rn+4],
	...
	Rn, Ra, Rb .... are low registers. Rn is on the list of registers to be updated by memory read.
	For example,
	LDM R2, {R1, R2, R5 – R7} ; Read R1,R2,R5,R6 and R7 from memory.

Instruction	LDMIA (Load Multiple Increment After)/LDMFD – Base Address Register Update to Subsequence Address
Function	Read multiple memory data word into registers and update base register
Syntax	LDMIA <Rn>!, {<Ra>, <Rb> ,....} ; Load multiple registers from memory
	; and increment base register after completion
Note	Ra = memory[Rn],

(*Continued*)

Instruction	LDMIA (Load Multiple Increment After)/LDMFD — Base Address Register Update to Subsequence Address
	Rb = memory[Rn+4], … and then update Rn to last read address plus 4 Rn, Ra, Rb …. are low registers. For example, LDMIA R0!, {R1, R2, R5 — R7} ; Read multiple registers, R0 update to address after last read operation. LDMFD is another name for the same instruction, which was used for restoring data from a Full Descending stack, in traditional ARM systems that use software managed stacks.

Instruction	STMIA (Store Multiple Increment After)/STMEA
Function	Write multiple register data into memory and update base register
Syntax	STMIA <Rn>!, {<Ra>, <Rb> ,….} ; Store multiple registers to memory ; and increment base register after completion
Note	memory[Rn] = Ra, memory[Rn+4] = Rb, … and then update Rn to last store address plus 4 Rn, Ra, Rb …. are low registers. For example, STMIA R0!, {R1, R2, R5 — R7} ; Store R1, R2, R5, R6, and R7 to memory ; and update R0 to address after where R7 stored STMEA is another name for the same instruction, which was used for storing data to an Empty Ascending stack, in traditional ARM systems that use software managed stack. If <Rn> is in the register list, it must be the first register in the register list.

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. However, only low registers, LR (for PUSH operation) and PC (for POP operation), are supported.

Instruction	PUSH
Function	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 {<Ra>, <Rb>, …., LR} ; Store multiple registers and LR to ; memory and decrement SP to the lowest pushed data address

(*Continued*)

Instruction	**PUSH**
Note	memory[SP-4] = Ra, memory[SP-8] = Rb, … and then update SP to last store address. For example, PUSH {R1, R2, R5 – R7, LR} ; Store R1, R2, R5, R6, R7, and LR to stack

Instruction	**POP**
Function	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 {<Ra>, <Rb>, …., PC} ; Load multiple registers and PC from ; memory and increment SP to the last emptied stack ; address plus 4
Note	Ra = memory[SP], Rb = memory[SP+4], … and then update SP to last restored address plus 4. For example, POP {R1, R2, R5 – R7} ; Restore R1, R2, R5, R6, R7 from stack

By allowing the Link Register (LR) and Program Counter (PC) to be used with the PUSH and the POP instructions, a function call can combine the register restore and function return operations into a single instruction. For example,

```
my_function
 PUSH { R4, R5, R7, LR} ; Save R4, R5, R7 and LR (return address)
 ... ; function body
 POP  { R4, R5, R7, PC} ; Restore R4, R5, R7 and return
```

Arithmetic Operations

The Cortex-M0 processor supports a number of arithmetic operations. The most basic are add, subtract, twos 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.

Instruction	**ADD**
Function	Add two registers
Syntax (UAL)	ADDS <Rd>, <Rn>, <Rm>
Syntax (Thumb)	ADD <Rd>, <Rn>, <Rm>
Note	Rd = Rn + Rm, APSR update. Rd, Rn, Rm are low registers.

Instruction	**ADD**
Function	Add an immediate constant into a register
Syntax (UAL)	ADDS <Rd>, <Rn>, #immed3
	ADDS <Rd>, #immed8
Syntax (Thumb)	ADD <Rd>, <Rn>, #immed3
	ADD <Rd>, #immed8
Note	Rd = Rn + ZeroExtend(#immed3), APSR update, or
	Rd = Rd + ZeroExtend(#immed8), APSR update.
	Rd, Rn, Rm are low registers.

Instruction	**ADD**
Function	Add two registers without updating APSR
Syntax (UAL)	ADD <Rd>, <Rm>
Syntax (Thumb)	ADD <Rd>, <Rm>
Note	Rd = Rd + Rm.
	Rd, Rm can be high or low registers.

Instruction	**ADD**
Function	Add stack pointer to a register without updating APSR
Syntax (UAL)	ADD <Rd>, SP, <Rd>
Syntax (Thumb)	ADD <Rd>, SP
Note	Rd = Rd + SP.
	Rd can be high or low register.

Instruction	**ADD**
Function	Add stack pointer to a register without updating APSR
Syntax (UAL)	ADD SP, <Rm>
Syntax (Thumb)	ADD SP, <Rm>
Note	SP = SP + Rm.
	Rm can be high or low register.

Instruction	**ADD**
Function	Add stack pointer to a register without updating APSR
Syntax (UAL)	ADD <Rd>, SP, #immed8
Syntax (Thumb)	ADD <Rd>, SP, #immed8
Note	Rd = SP + ZeroExtend(#immed8 <<2).
	Rd is a low register.

Instruction	ADD
Function	Add an immediate constant to stack pointer
Syntax (UAL)	ADD SP, SP, #immed7
Syntax (Thumb)	ADD SP, #immed7
Note	SP = SP + ZeroExtend(#immed7 <<2).
	This instruction is useful for C functions to adjust the SP for local variables.

Instruction	ADR (ADD)
Function	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)
Syntax (Thumb)	ADR <Rd>, (normal syntax)
	ADD <Rd>, PC, #immed8 (alternate syntax)
Note	Rd = (PC[31:2]<<2) + ZeroExtend(#immed8 <<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.
	Rd is a low register.

Instruction	ADC
Function	Add with carry and update APSR
Syntax (UAL)	ADCS <Rd>, <Rm>
Syntax (Thumb)	ADC <Rd>, <Rm>
Note	Rd = Rd + Rm + Carry
	Rd and Rm are low registers.

Instruction	SUB
Function	Subtract two registers
Syntax (UAL)	SUBS <Rd>, <Rn>, <Rm>
Syntax (Thumb)	SUB <Rd>, <Rn>, <Rm>
Note	Rd = Rn − Rm, APSR update.
	Rd, Rn, Rm are low registers.

Instruction	SUB
Function	Subtract a register with an immediate constant
Syntax (UAL)	SUBS <Rd>, <Rn>, #immed3
	SUBS <Rd>, #immed8
Syntax (Thumb)	SUB <Rd>, <Rn>, #immed3
	SUB <Rd>, #immed8
Note	Rd = Rn − ZeroExtend(#immed3), APSR update, or
	Rd = Rd − ZeroExtend(#immed8), APSR update.
	Rd, Rn are low registers.

Instruction	SUB
Function	Subtract SP by an immediate constant
Syntax (UAL)	SUB SP, SP, #immed7
Syntax (Thumb)	SUB SP, #immed7
Note	SP = SP - ZeroExtend(#immed7 <<2). This instruction is useful for C functions to adjust the SP for local variables.

Instruction	SBC
Function	Subtract with carry (borrow)
Syntax (UAL)	SBCS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	SBC <Rd>, <Rm>
Note	Rd = Rd − Rm − Borrow, APSR update. Rd and Rm are low registers.

Instruction	RSB
Function	Reverse Subtract (negative)
Syntax (UAL)	RSBS <Rd>, <Rn>, #0
Syntax (Thumb)	NEG <Rd>, <Rn>
Note	Rd = 0 − Rm, APSR update. Rd and Rm are low registers.

Instruction	MUL
Function	Multiply
Syntax (UAL)	MULS <Rd>, <Rm>, <Rd>
Syntax (Thumb)	MUL <Rd>, <Rm>
Note	Rd = Rd * Rm, APSR.N, and APSR.Z update. Rd and Rm are low registers.

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.

Instruction	CMP
Function	Compare
Syntax (UAL)	CMP <Rn>, <Rm>
Syntax (Thumb)	CMP <Rn>, <Rm>
Note	Calculate Rn − Rm, APSR update but subtract result is not stored.

Instruction	CMP
Function	Compare
Syntax (UAL)	CMP <Rn>, #immed8
Syntax (Thumb)	CMP <Rn>, #immed8
Note	Calculate Rd – ZeroExtended(#immed8), APSR update but subtract result is not stored. Rn is a low register.

Instruction	CMN
Function	Compare negative
Syntax (UAL)	CMN <Rn>, <Rm>
Syntax (Thumb)	CMN <Rn>, <Rm>
Note	Calculate Rn – 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.

Instruction	AND
Function	Logical AND
Syntax (UAL)	ANDS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	AND <Rd>, <Rm>
Note	Rd = AND(Rd, Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.

Instruction	ORR
Function	Logical OR
Syntax (UAL)	ORRS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	ORR <Rd>, <Rm>
Note	Rd = OR(Rd, Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.

Instruction	EOR
Function	Logical Exclusive OR
Syntax (UAL)	EORS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	EOR <Rd>, <Rm>
Note	Rd = XOR(Rd, Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.

Instruction	BIC
Function	Logical Bitwise Clear
Syntax (UAL)	BICS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	BIC <Rd>, <Rm>
Note	Rd = AND(Rd, NOT(Rm)), APSR.N, and APSR.Z update. Rd and Rm are low registers.

Instruction	MVN
Function	Logical Bitwise NOT
Syntax (UAL)	MVNS <Rd>, <Rm>
Syntax (Thumb)	MVN <Rd>, <Rm>
Note	Rd = NOT(Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.

Instruction	TST
Function	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. Rd and Rm are low registers.

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.

Instruction	ASR
Function	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. Rd and Rm are low registers.

Instruction	ASR
Function	Arithmetic Shift Right
Syntax (UAL)	ASRS <Rd>, <Rm>, #immed5

(*Continued*)

Instruction	ASR
Syntax (Thumb)	ASR <Rd>, <Rm>, #immed5
Note	Rd = Rm >> immed5, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.

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).

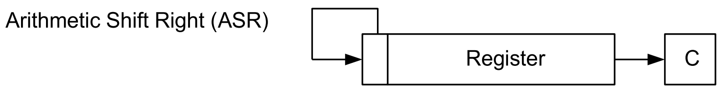


Figure 5.3:
Arithmetic Shift Right.

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

Instruction	LSL
Function	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. Rd and Rm are low registers.

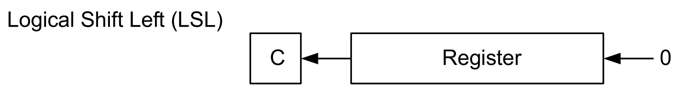


Figure 5.4:
Logical Shift Left.

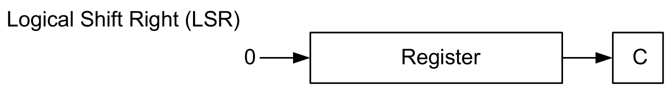


Figure 5.5:
Logical Shift Right.

Instruction	LSL
Function	Logical Shift Left
Syntax (UAL)	LSLS <Rd>, <Rm>, #immed5
Syntax (Thumb)	LSL <Rd>, <Rm>, #immed5
Note	Rd = Rm << #immed5, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.

Instruction	LSR
Function	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. Rd and Rm are low registers.

Instruction	LSR
Function	Logical Shift Right
Syntax (UAL)	LSRS <Rd>, <Rm>, #immed5
Syntax (Thumb)	LSR <Rd>, <Rm>, #immed5
Note	Rd = Rm >> #immed5, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.

There is only one rotate instruction, ROR (Figure 5.6).

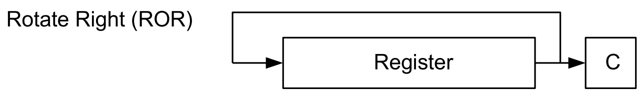


Figure 5.6:
Rotate Right.

Instruction	ROR
Function	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. Rd and Rm are low registers.

If a rotate left operation is needed, this can be done using a ROR with a different offset:

$$\text{Rotate\_Left}(\text{Data}, \text{offset}) == \text{Rotate\_Right}(\text{Data}, (32 - \text{offset}))$$

Extend and Reverse Ordering Operations

The Cortex-M0 processor supports a number of instructions that can perform data reordering or extraction (Figures 5.7, 5.8, and 5.9).

Instruction	REV (Byte-Reverse Word)
Function	Byte Order Reverse
Syntax	REV <Rd>, <Rm>
Note	Rd = {Rm[7:0] , Rm[15:8], Rm[23:16], Rm[31:24]} Rd and Rm are low registers.

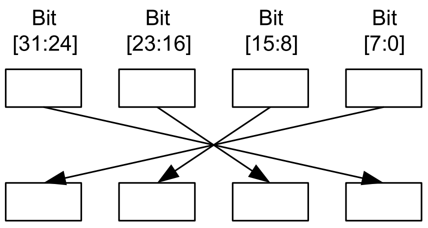


Figure 5.7:
REV operation.

Instruction	REV16 (Byte-Reverse Packed Half Word)
Function	Byte Order Reverse within half word
Syntax	REV16 <Rd>, <Rm>
Note	Rd = {Rm[23:16], Rm[31:24], Rm[7:0] , Rm[15:8]} Rd and Rm are low registers.

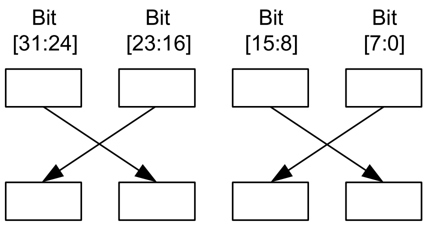


Figure 5.8:
REV16 operation.

Instruction	REVSH (Byte-Reverse Signed Half Word)
Function	Byte order reverse within lower half word, then sign extend result
Syntax	REVSH <Rd>, <Rm>
Note	Rd = SignExtend({Rm[7:0] , Rm[15:8]})
	Rd and Rm are low registers.

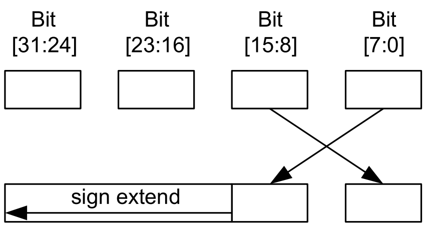


Figure 5.9:
REVSH operation.

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.

Instruction	SXTB (Signed Extended Byte)
Function	SignExtend lowest byte in a word of data
Syntax	SXTB <Rd>, <Rm>
Note	Rd = SignExtend(Rm[7:0])
	Rd and Rm are low registers.

Instruction	SXTH (Signed Extended Half Word)
Function	SignExtend lower half word in a word of data
Syntax	SXTH <Rd>, <Rm>
Note	Rd = SignExtend(Rm[15:0])
	Rd and Rm are low registers.

Instruction	UXTB (Unsigned Extended Byte)
Function	Extend lowest byte in a word of data
Syntax	UXTB <Rd>, <Rm>
Note	Rd = ZeroExtend(Rm[7:0])
	Rd and Rm are low registers.

Instruction	UXTH (Unsign Extended Half Word)
Function	Extend lower half word in a word of data
Syntax	UXTH <Rd>, <Rm>
Note	Rd = ZeroExtend(Rm[15:0]) Rd and Rm are low registers.

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.

Instruction	B (Branch)
Function	Branch to an address (unconditional)
Syntax	B <label>
Note	Branch range is +/− 2046 bytes of current program counter

Instruction	B<cond> (Conditional Branch)
Function	Depending of APSR, branch to an address
Syntax	B<cond> <label>
Note	Branch range is +/− 254 bytes of current program counter. For example, CMP R0, #0x1 ; Compare R0 with 0x1 BEQ process1 ; Branch to process1 if R0 equal 1

The <cond> is one of the 14 possible condition suffixes (Table 5.7).

For example, a simple loop that runs three times could be

```
   MOVS      R0, #3     ; Loop counter starting value is 3
loop                    ; "loop" is an address label
   SUBS      R0, #1     ; Decrement by 1 and update flag
   BGT       loop       ; branch to loop if R0 is Greater Than (GT) 1
```

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)

The loop will execute three times. The third time, R0 is 1 before the SUBS instruction. After the SUBS instruction, the zero flag is set, so the condition for the branch failed and the program continues execution after the BGT instruction.

Instruction	BL (Branch and Link)
Function	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 +/− 16MB of current program counter. For example, BL functionA ; call a function called functionA

Instruction	BX (Branch and Exchange)
Function	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).

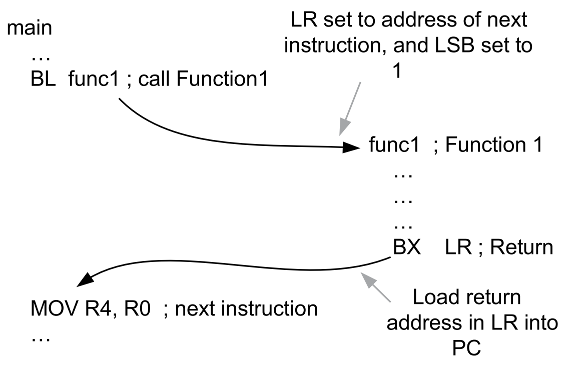

main
…
BL func1 ; call Function1
LR set to address of next instruction, and LSB set to 1
func1 ; Function 1
…
…
…
BX LR ; Return
Load return address in LR into PC
MOV R4, R0 ; next instruction
…

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.

Instruction	BLX (Branch and Link with Exchange)
Function	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 support 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. However, memory barriers may be necessary on other ARM processors that have more complex memory systems. If the software needs to be portable to other ARM processors, then the uses of memory barrier instructions could be essential. Therefore, the memory barrier instructions are supported on the Cortex-M0 to provide better compatibility between the Cortex-M0 processor and other ARM processors.

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

- DMB
- DSB
- ISB

Instruction	**DMB**
Function	Data Memory Barrier
Syntax	DMB
Note	Ensures that all memory accesses are completed before new memory access is committed

Instruction	**DSB**
Function	Data Synchronization Barrier
Syntax	DSB
Note	Ensures that all memory accesses are completed before the next instruction is executed

Instruction	**ISB**
Function	Instruction Synchronization Barrier
Syntax	ISB
Note	Flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

Architecturally, there are various cases where these instructions are needed. Although in practice omitting the memory barrier instruction might not cause any issue on the Cortex-M0, it could be an issue when the same software is used on another ARM processor. For example, after changing the CONTROL register with MSR instruction, architecturally an ISB should be used after writing to the CONTROL register to ensure subsequent instructions use the updated settings. Although the Cortex-M0 omits the ISB instruction in this case, the omission does not cause an issue.

Another example is memory remap control. In some microcontrollers, a hardware register can change the memory map. After writing to the memory map switching register, you need to use the DSB instruction to ensure the write has been completed and memory configuration has been updated before carrying out the next step. Otherwise, if the memory switching is delayed, possibly because of a write buffer in the system bus interface (e.g., the Cortex-M3 has a write buffer in the system bus interface to allow higher performance), and the processor starts to access the switched memory region immediately, the access could be using the old memory mapping, or the transfer could become corrupted by the memory map switching.

Memory barrier instruction is also needed when the program contains self-modifying code. For example, if an application changes its own program code, the instruction execution that follows should use the updated program code. However, if the processor is pipelined or has a fetch buffer, the processor may have already fetched an old copy of the modified instruction. In this case, the program should use a DSB operation to ensure the write to the memory is completed; then it should use an ISB instruction to ensure the instruction fetch buffer is updated with the new instructions.

More details about memory barriers can be found in the ARMv6-M Architecture Reference manual (reference 3).

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.

Instruction	**SVC**
Function	Supervisor call
Syntax	SVC #<immed8> SVC <immed8>
Note	Trigger the SVC exception. For example, SVC #3 ; SVC instruction, with parameter, equals 3. Alternative syntax without the "#" is also allowed. For example, SVC 3 ; this is the same as SVC #3.

An 8-bit immediate data element is used with SVC instruction. This parameter does not affect the SVC exception directly, but it can be extracted by the SVC handler and be used as an input to the SVC function. Typically the SVC can be used to provide access to system service or the application programming interface (API), and this parameter can be used to indicate which system service is required.

If the SVC instruction is used in an exception handler that has the same or a higher priority than the SVC, this will cause a fault exception. As a result, the SVC cannot be used in the hard fault handler, the NMI handler, or the SVC handler itself.

Besides using MSR instruction, the PRIMASK special register can also be changed using an instruction called CPS:

Instruction	CPS
Function	Change processor state: enable or disable interrupt
Syntax	CPSIE I ; Enable Interrupt (Clearing PRIMASK) CPSID I ; Disable Interrupt (Setting PRIMASK)
Note	PRIMASK only block external interrupts, SVC, PendSV, SysTick. But it does not block NMI and the hard fault handler.

The switching of PRIMASK to disable and enable the interrupt is commonly used for timing critical code.

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. Note that for the Cortex-M1 processor, as the design is implemented in a FPGA design, which does not have sleep mode, these two instructions execute as NOP and will not cause the processor to stop.

Instruction	WFI
Function	Wait for Interrupt
Syntax	WFI
Note	Stops program execution until an interrupt arrives or until the processor enters a debug state.

WFE is just like WFI, except that it can also be awakened by events. An event can be an interrupt, the execution of an SEV instruction (see next page), or the entering of a debug state. A previous event also affects a WFE instruction: Inside the Cortex-M0 processor, there is an event register that records whether an event has occurred (exceptions, external events, or the execution of an SEV instruction). If the event register is not set when the WFE is executed, the WFE instruction execution will cause the processor to enter sleep mode. If the event register is set when WFE is executed, it will cause the event register to be cleared and the processor proceeds to the next instruction.

Instruction	WFE
Function	Wait for Event
Syntax	WFE
Note	If the internal event register is set, it clears the internal event register and continues execution. Otherwise, stop program execution until an event (e.g., an interrupt) arrives or until the processor enters a debug state.

WFE can also be awakened by an external event input signal, which is normally used in a multiprocessing environment.

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. For single-processor systems, where the processor does not have a multiprocessor communication interface or the multiprocessor communication interface is not used, the SEV can only affect the local event register inside the processor itself.

Instruction	SEV
Function	Send event to all processors in multiprocessing environment (including itself)
Syntax	SEV
Note	Set local event register and send out an event pulse to other microprocessor in a multiple processor system

Other Instructions

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

Instruction	NOP
Function	No operation
Syntax	NOP
Note	The NOP instruction takes one cycle minimum on Cortex-M0. In general, delay timing produced by NOP instruction is not guaranteed and can vary among different systems (e.g., memory wait states, processor type). If the timing delay needs to be accurate, a hardware timer should be used.

The breakpoint instruction 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. This is limited to four breakpoints. Because many microcontrollers use flash memory, which can be reprogrammed a number of times, using software breakpoint instruction allows more breakpoints to be set at no extra cost. The breakpoint instruction has an 8-bit immediate data field. This immediate value does not affect the breakpoint operation directly, but the debugger can extract this value and use it for debug operation.

Instruction	**BKPT**
Function	Breakpoint
Syntax	BKPT #<immed8> BKPT <immed8>
Note	BKPT instruction can have an 8-bit immediate data field. The debugger can use this as an identifier for the BKPT. For example, BKPT #0 ; breakpoint, with immediate field equal zero Alternative syntax without the "#" is also allowed. For example, BKPT 0 ; This is the same as BKPT #0.

The YIELD instruction is a hint instruction targeted for embedded operating systems. This is not implemented in the current releases of the Cortex-M0 processor and executes as NOP.

When used in multithread systems, YIELD can indicate that the current thread is delayed (e.g., waiting for hardware) and can be swapped out. In this case, the processor does not have to spend too much time on an idle task and can switch to other tasks earlier to get better system throughput. On the Cortex-M0 processor, this instruction is executed as an NOP (no operation) because it does not have special support for multithreading. This instruction is included for better software compatibility with other ARM processors.

Instruction	**YIELD**
Function	Indicate task is stalled
Syntax	YIELD
Note	Execute as NOP on the Cortex-M0 processor

Pseudo Instructions

Apart from the instructions listed in the previous section, a few pseudo instructions are also available. The pseudo instructions are provided by the assembler tools, which convert them into one or more real instructions.

The most commonly used pseudo instruction is the LDR. This allows a 32-bit immediate data item to be loaded into a register.

Pseudo Instruction	**LDR**
Function	Load a 32-bit immediate data into register Rd
Syntax	LDR <Rd>, =immed32
Note	This is translated to a PC-related load from a literal pool. For example, LDR R0, =0x12345678 ; Set R0 to hexadecimal value 0x12345678 LDR R1, =10 ; Set R1 to decimal value 10 LDR R2, ='A' ; Set R2 to character 'A'

Pseudo Instruction	**LDR**
Function	Load a data in specified address (label) into register
Syntax	LDR <Rd>, label

Note	The address of label must be word aligned and should be closed to the current program counter. For example, you can put a data item in program ROM using DCD and then access this data item using LDR.
	`   LDR R0, CONST_NUM       ; Load CONST_NUM (0x17) in R0`
	`   ...`
	`   ALIGN 4                 ; make sure next data are word aligned`
	`CONST_NUM DCD 0x17         ; Put a data item in program code`

Other pseudo instructions depend on the tool chain being used. For more information, please refer to the tools documentation for details.

Additional data

Embedded Systems
An Embedded System is a computer system that has computer hardware and software embedded to perform specific tasks. In contrast to general purpose computers or personal computers (PCs)' wbich can perform various types of tasks, embedded systems are designed to perform a specific set of tasks. Key components of an embedded system include. microprocessor or microcontroller, memory (RAM, ROM, cache), networking units (Etllemet, WiFi adapters), input/output units (display, keyboard, etc.) and storage (such as Hash memory). Some embedded systems have specialized processors such as digital signal processors (DSPs), graphics processors and application specific processors. Embedded systems run embedded operating systems such as real-time operating systems (RTOS). Embedded systems range from low-cost miniaturized devices such as digital watches to devices such as digital cameras, point of vending machines, appliances (such as washing machines), etc.

IoT Hardware Devices
1. Sensors: A sensor is an IoT device that senses physical changes in the environment and sends the data for manipulation via a network. Clouds store the data for future references. Sensors monitor data and collect information constantly.
2. Microcontrollers: A microcontroller is a small computer that is capable of performing operations. It sits on a semiconductor integrated circuit chip. Microcontollers usually operate on a single function and hence differ from regular computers. They perform a variety of tasks in a relatively simpler manner.
3. Wearable devices: Wearable devices are a benchmark revolution of the IoT industry. These are Iot devices that humans can wear on their bodies to regulate and perform a variety of tasks. These wearables are capable of tracking glucose levels, monitor heart attack risks, coagulation and asthma monitoring, daily step and calorie consumption tracking.
4. Basic devices: Traditional computers such as desktops, tablets and cellphones are still an integral part of any IoT ecosystem. Desktops offer users with simple access to a lot of information and cell phones allow remote access to Iot devices using APIs.
5. Datasheets: Datasheets give the details about the functionality of any hardware components. It is important to study the datasheet of any hardware before making a purchase to make sure you are buying the right product.
Datasheets offer you detailed information on the parameters of the hardware, its physical size, different voltage and electrical parameters, maximum current usage and the number of input/output pins. Datasheets are highly useful as they give you all the information you need before buying complicated hardware components.
6. Integrated circuits: Integrated circuits are chips. They are microcontrollers. You can buy empty chips in the market and download any kind of design into the chip. They are made using Silicon and it is packaged into shapes of rectangles. These chips contain complicated logic circuits, gates, registers, switches, I/O terminals and flip flops.

Integrated circuits do a variety of functions, they can perform arithmetic and logical calculations. They act as processors too. They contain binary coded information which is programmed to perform a set of tasks.
Standard chips are available in the market that perform a fixed set of operations. You can also construct chips to perform your desired set of functions and these are known as custom made chips.

IoT Hardware Providers

Various companies have come up with their own personalized IoT hardware and software and many emerging companies are adapting to these policies. However, the most common Iot hardware providers are listed below:
a. Adafruit is best if you want to get hands-on experience with IoT. The company sells IoT DIY kits with an online guide to help you through the initial setting up. You can interact, manipulate and store your data.
b. Raspberry Pi is best at student level to get hands-on experience with IoT. You can interact, manipulate and store your data.
b. Arduino, the company brands microcontrollers, IoT kits and software tools.
c. Lantronix provides solutions for the IoT such as smart hardware, networking, engineering and artificial intelligence.
d. Espressif can interconnect with the system to provide wifi and bluetooth. It has high level integration. It uses low power and has a robust design.