

23CSPC208

Embedded System Design

Dr. Rejeesh Rayaroth

Asst. Professor CSE

MITE

SYLLABUS

Embedded System Design			
Semester	IV	CIE Marks	50
Course Code	23CSPC208	SEE Marks	50
Teaching Hrs/Week (L:T: P)	3:0:2	Exam Hrs	03
Total Hrs	64	Credits	04

MODULE-I

Module 1: Introduction to Microcontroller	No. of Hrs: 9+4
--	------------------------

Microprocessors versus Microcontrollers, ARM Embedded Systems: The RISC design philosophy, The ARM Design Philosophy, Embedded System Hardware, Embedded System Software, ARM Processor Fundamentals: Registers, Current Program Status Register, Pipeline, Exceptions, Interrupts, and the Vector Table, Core Extensions

Laboratory Components:

1. Using Keil software write a program to find the sum of the first 10 integer numbers
2. Using Keil software write a program to find the factorial of a number

MODULE -II

Module 2: ARM Programming Model	No. of Hrs: 8+4
--	------------------------

Introduction to the ARM Instruction Set: Data Processing Instructions, Branch Instructions, Load Store Instructions, Software Interrupt Instructions, Program Status Register Instructions, Loading Constants, Conditional Execution

Laboratory Components:

1. Using Keil software write a program to add an array of 16 bit numbers and store the 32-bit result in internal RAM
2. Using Keil software write a program to find the square of a number (1 to 10) using a look-up table

MODULE -III

Module 3: Introduction to Embedded Systems

No. of Hrs: 9+4

Embedded System Components, Embedded Vs General computing system, Classification of Embedded systems, Applications areas of embedded systems, Core of an Embedded System : processor/controller, Memory, Sensors and Actuators, LED, 7 segment LED display, stepper motor, Keyboard, Push button switch, Communication Interface (onboard and external types), Embedded firmware, Other system components

Laboratory Components:

1. Using Keil software write a program to find the largest or smallest number in an array of 32 numbers
2. Using Keil software write a program to arrange a series of 32 bit numbers in ascending/descending order

MODULE -IV

Module 4: Embedded Hardware Design and Development

No. of Hrs: 9+4

Analog & Digital Electronics Components, Electronic Design Automation tools, Embedded Firmware Design approaches: Super loop based approach & operating system based approach, Firmware development languages: Assembly language & High Level language

Laboratory Components:

1. Using Keil software write a program to count the number of ones and zeros in two consecutive memory locations.
2. Using Keil software display "Hello World" message using Internal UART.

MODULE -V

Module 5: Real Time Operating System(RTOS) Based Embedded System Design

No. of Hrs: 9+4

RTOS: Concept, task, process and threads (Only POSIX Threads with an example program), Thread preemption, Multiprocessing and Multitasking, Task Scheduling, Task Communication: Shared memory, message passing, Remote Procedure call and socket, Task synchronization issues, Task synchronization Techniques.

Laboratory Components:

1. With the help of the Embedded controller (Arduino, Raspberry Pi) control a DC motor
2. With the help of the Embedded controller (Arduino, Raspberry Pi) control a Stepper motor and rotate it in clockwise and anti-clockwise direction

Course Learning Objectives

Course Learning Objectives: This course is designed to

1. Impart the fundamentals of ARM architecture, including its features, modes of operation, instruction set, and programming model
2. Explain the basic concepts of embedded systems, including microcontrollers, sensors, actuators, and their applications in various domains
3. Provide a comprehensive knowledge of embedded system design and equips students with the skills to develop innovative and efficient solutions
4. Inculcate concepts of multitasking, task scheduling, and synchronization using an RTOS to manage real-time tasks in applications

Course Outcomes

Course Outcomes: At the end of the course, the student will be able to

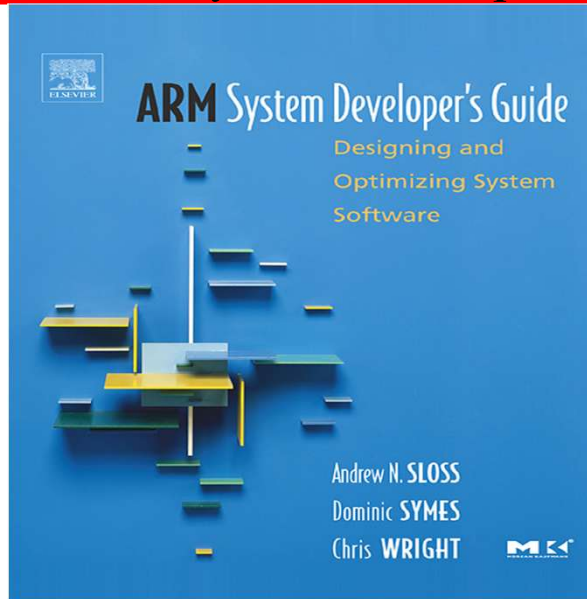
1. Explain ARM architecture, its features, modes of operation, instruction set, and memory management
2. Identify and discuss major application areas of Embedded Systems
3. Leverage the programming proficiency gained in ARM microcontrollers across various practical applications
4. Analyze the role of a real-time operating system in embedded system applications

Text Books

Textbooks:

1. Andrew N Sloss, Dominic Symes and Chris Wright, "ARM system developers guide", Morgan Kaufman publishers, 2008. (Chapter 1, Chapter 2, Chapter 3)
2. Shibu K V, "Introduction to Embedded Systems", 2nd Edition, Tata McGraw Hill Education, Private Limited, 2017 (Chapter 1, Chapter 2, Chapter 8, Chapter 9, Chapter 10)

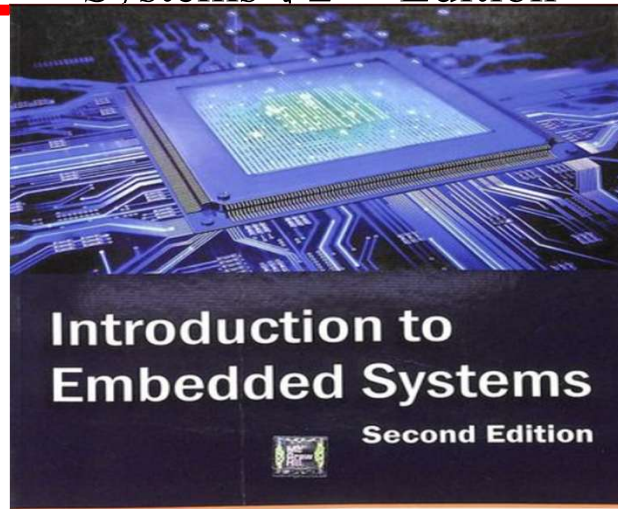
Text book 1: Andrew N Sloss, Dominic Symes and Chris Wright, "ARM system developers guide",



Text book 1: Andrew N Sloss, Dominic Symes and Chris Wright, "ARM system developers guide",

	ABOUT THE AUTHORS	ii	CHAPTER		
	PREFACE	xi			
CHAPTER					
1	ARM EMBEDDED SYSTEMS	3	3	INTRODUCTION TO THE ARM INSTRUCTION SET	47
1.1	The RISC Design Philosophy	4	3.1	Data Processing Instructions	50
1.2	The ARM Design Philosophy	5	3.2	Branch Instructions	58
1.3	Embedded System Hardware	6	3.3	Load-Store Instructions	60
1.4	Embedded System Software	12	3.4	Software Interrupt Instruction	73
1.5	Summary	15	3.5	Program Status Register Instructions	75
CHAPTER			3.6	Loading Constants	78
2	ARM PROCESSOR FUNDAMENTALS	19	3.7	ARMv5E Extensions	79
2.1	Registers	21	3.8	Conditional Execution	82
2.2	Current Program Status Register	22	3.9	Summary	84
2.3	Pipeline	29			
2.4	Exceptions, Interrupts, and the Vector Table	33			
2.5	Core Extensions	34			
2.6	Architecture Revisions	37			
2.7	ARM Processor Families	38			
2.8	Summary	43			

Text Book 2:Shibu K V, "Introduction to Embedded Systems", 2ND Edition



Shibu K V



Text Book 2:Shibu K V, "Introduction to Embedded Systems", 2ND Edition

Part-1: Embedded Systems: Understanding the Basic Concepts 1

1. Introduction to Embedded Systems

- 1.1 What is an Embedded System? 4
- 1.2 Embedded Systems vs. General Computing Systems 4
- 1.3 History of Embedded Systems 5
- 1.4 Classification of Embedded Systems 6
- 1.5 Major Application Areas of Embedded Systems 8
- 1.6 Purpose of Embedded Systems 8
- 1.7 Wearable Devices—The Innovative Bonding of Lifestyle with Embedded Technologies 11

Summary 13

Keywords 14

Objective Questions 14

Review Questions 15

3 2. The Typical Embedded System

- 2.1 Core of the Embedded System 18
- 2.2 Memory 29
- 2.3 Sensors and Actuators 35
- 2.4 Communication Interface 45
- 2.5 Embedded Firmware 59
- 2.6 Other System Components 60
- 2.7 PCB and Passive Components 64

Summary 64

Keywords 65

Objective Questions 67

Review Questions 70

Lab Assignments 72

16

Text Book 2: Shibu K V, "Introduction to Embedded Systems", 2ND Edition

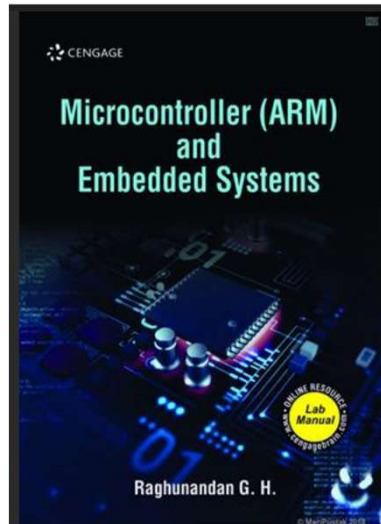
Part-2: Design and Development of Embedded Product	231	9. Embedded Firmware Design and Development	308
8. Embedded Hardware Design and Development	234	9.1 Embedded Firmware Design Approaches 309	
8.1 Analog Electronic Components 235		9.2 Embedded Firmware Development Languages 312	
8.2 Digital Electronic Components 236		9.3 Programming in Embedded C 324	
8.3 VLSI and Integrated Circuit Design 249		Summary 376	
8.4 Electronic Design Automation (EDA) Tools 254		Keywords 377	
8.5 How to use the OrCAD EDA Tool? 255		Objective Questions 378	
8.6 Schematic Design using Orcad Capture CIS 255		Review Questions 383	
8.7 The PCB Layout Design 273		Lab Assignments 386	
8.8 Printed Circuit Board (PCB) Fabrication 293		10. Real-Time Operating System (RTOS) based Embedded System Design	387
Summary 299		10.1 Operating System Basics 389	
Keywords 300		10.2 Types of Operating Systems 392	
Objective Questions 301		10.3 Tasks, Process and Threads 396	
Review Questions 304		10.4 Multiprocessing and Multitasking 408	
Lab Assignments 305		10.5 Task Scheduling 410	
		10.6 Threads, Processes and Scheduling: Putting them Altogether 428	
		10.7 Task Communication 433	
		10.8 Task Synchronisation 449	
		10.9 Device Drivers 482	
		10.10 How to Choose an RTOS 484	
		Summary 485	
		Keywords 487	
		Objective Questions 489	
		Review Questions 499	
		Lab Assignments 504	

REFERENCE BOOK

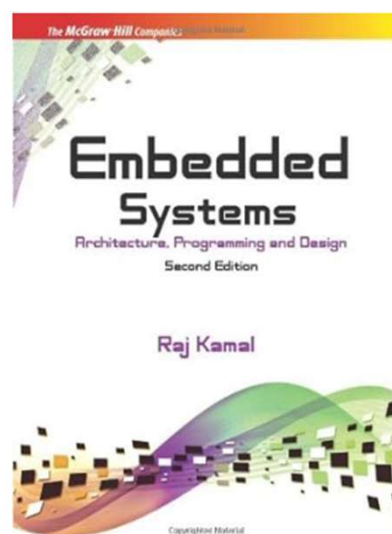
Reference Books:

1. Raghunandan. G.H, "Microcontroller (ARM) and Embedded System", Cengage learning Publication, 2019
2. Raj Kamal, "Embedded System", Tata McGraw-Hill Publishers, 2nd Edition, 2008

Reference Book1:Raghunandhan GH : “Microcontroller(ARM)and Embedded Systems”



Reference Book2: Raj Kamal “EmbeddedSystem 2ND Edition”



WEB LINKS

Web links:

1. NPTEL Course on Embedded Systems
[:https://archive.nptel.ac.in/courses/106/105/106105193/](https://archive.nptel.ac.in/courses/106/105/106105193/)
2. Youtube Videos on Embedded Systems
https://www.youtube.com/results?search_query=microcontroller+and+embedded+systems

Module I

Module 1: Introduction to Microcontroller	No. of Hrs: 9+4
<p>Microprocessors versus Microcontrollers, ARM Embedded Systems: The RISC design philosophy, The ARM Design Philosophy, Embedded System Hardware, Embedded System Software, ARM Processor Fundamentals: Registers, Current Program Status Register, Pipeline, Exceptions, Interrupts, and the Vector Table, Core Extensions</p> <p>Laboratory Components:</p> <ol style="list-style-type: none"> 1. Using Keil software write a program to find the sum of the first 10 integer numbers 2. Using Keil software write a program to find the factorial of a number 	

Microprocessors Vs MicroController

Parameter	Microprocessor	Microcontroller
Definition	Microprocessors can be understood as the heart of a computer system.	Microcontrollers can be understood as the heart of an embedded system.
What is it?	A microprocessor is a processor where the memory and I/O component are connected externally.	A microcontroller is a controlling device wherein the memory and I/O output component are present internally.
Circuit complexity	The circuit is complex due to external connection.	Microcontrollers are present on chip memory. The circuit is less complex.
Memory and I/O components	The memory and I/O components are to be connected externally.	The memory and I/O components are available.
Compact system compatibility	Microprocessors can't be used in compact system.	Microcontrollers can be used with a compact system.
Efficiency	Microprocessors are not efficient.	Microcontrollers are efficient.
Number of registers	Microprocessors have less number of registers.	Microcontrollers have more number of registers.
Applications	Microprocessors are generally used in personal computers.	Microcontrollers are generally used in washing machines, and air conditioners.

Microprocessor Vs MicroController ctd.

Microprocessors	Microcontrollers
Microprocessors generally does not have RAM, ROM and I/O pins.	Microcontroller is 'all in one' processor, with RAM, I/O ports, all on the chip.
Microprocessors usually use its pins as a bus to interface to RAM, ROM, and peripheral devices. Hence, the controlling bus is expandable at the board level.	Controlling bus is internal and not available to the board designer.
Microprocessors are generally capable of being built into bigger general purpose applications.	Microcontrollers are usually used for more dedicated applications.
Microprocessors, generally do not have power saving system.	Microcontrollers have power saving system, like idle mode or power saving; mode so overall it uses less power.
The overall cost of systems made with Microprocessors is high, because of the high number of external components required.	Microcontrollers are made by using complementary metal oxide semiconductor technology; so they are far cheaper than Microprocessors.
Processing speed of general microprocessors is above 1 GHz; so it works much faster than Microcontrollers.	Processing speed of Microcontrollers is about 8 MHz to 50 MHz.
Microprocessors are based on von-Neumann model; where, program and data are stored in same memory module.	Microcontrollers are based on Harvard architecture; where, program memory and data memory are separate.

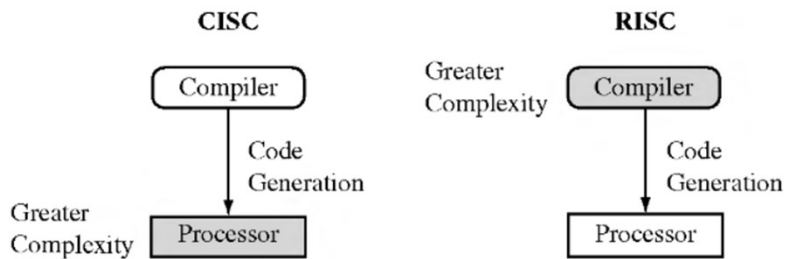
Introduction to ARM Microcontrollers

- ❑ The ARM processor core is a key component of many successful 32-bit embedded systems.
- ❑ ARM cores are widely used in mobile phones, handheld organizers, and a multitude of other everyday portable consumer devices.
- ❑ ARM's designers have come a long way from the first ARM1 prototype in 1985.
- ❑ The ARM company bases their success on a simple and powerful original design, which continues to improve today through constant technical innovation.
- ❑ In fact, the ARM core is not a single core, but a whole family of designs sharing similar design principles and a common instruction set.
- ❑ We discuss how the RISC (reduced instruction set computer) design philosophy was adapted by ARM to create a flexible embedded processor.
- ❑ We then introduce an example embedded device and discuss the typical hardware and software technologies that surround an ARM processor.

RISC DESIGN PHILOSOPHY

- ❑ The ARM core uses a RISC architecture.
- ❑ RISC is a design philosophy aimed at delivering simple but powerful instructions that execute within a single cycle at a high clock speed.
- ❑ The RISC philosophy concentrates on reducing the complexity of instructions performed by the hardware because it is easier to provide greater flexibility and intelligence in software rather than hardware.
- ❑ As a result, a RISC design places greater demands on the compiler.
- ❑ In contrast, the traditional complex instruction set computer (CISC) relies more on the hardware for instruction functionality, and consequently the CISC instructions are more complicated.

CISC Vs RISC



CISC vs. RISC. CISC emphasizes hardware complexity. RISC emphasizes compiler complexity.

CISC Vs RISC

Fig: CISC vs. RISC

CISC	RISC
1. Complex instructions, taking multiple clock	1. Simple instructions, taking single clock
2. Emphasis on hardware, complexity is in the micro-program/processor	2. Emphasis on software, complexity is in the compiler
3. Complex instructions, instructions executed by micro-program/processor	3. Reduced instructions, instructions executed by hardware
4. Variable format instructions, single register set and many instructions	4. Fixed format instructions, multiple register sets and few instructions
5. Many instructions and many addressing modes	5. Fixed instructions and few addressing modes
6. Conditional jump is usually based on status register bit	6. Conditional jump can be based on a bit anywhere in memory
7. Memory reference is embedded in many instructions	7. Memory reference is embedded in LOAD/STORE instructions

THE RISC PHILOSOPHY - Design Rules

The RISC philosophy is implemented with four major design rules:

1. Instructions

- RISC processors have a reduced number of instruction classes. These classes provide simple operations that can each execute in a single cycle.
- The compiler or programmer synthesizes complicated operations by combining several simple instructions.
- Each instruction is a fixed length to allow the pipeline to fetch future instructions before decoding the current instruction.
- In contrast, in CISC processors the instructions are often of variable size and take many cycles to execute.

THE RISC PHILOSOPHY - Design Rules ctd

2. Pipelines

- The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines.
- Ideally the pipeline advances by one step on each cycle for maximum throughput. Instructions can be decoded in one pipeline stage.
- There is no need for an instruction to be executed by a miniprogram called microcode as on CISC processors.

3. Registers

- RISC machines have a large general-purpose register set.
- Any register can contain either data or an address.
- Registers act as the fast local memory store for all data processing operations.
- In contrast, CISC processors have dedicated registers for specific purposes.

THE RISC PHILOSOPHY - Design Rules ctd.

4. Load-store architecture

- ❑ The processor operates on data held in registers.
- ❑ Separate load and store instructions transfer data between the register bank and external memory.
- ❑ Memory accesses are costly, so separating memory accesses from data processing provides an advantage because you can use data items held in the register bank multiple times without needing multiple memory accesses.
- ❑ In contrast, with a CISC design the data processing operations can act on memory directly.
- ❑ **These design rules allow a RISC processor to be simpler, and thus the core can operate at higher clock frequencies.**
- ❑ **In contrast, traditional CISC processors are more complex and operate at lower clock frequencies.**

The ARM Design Philosophy

- ❑ There are a number of physical features that have driven the ARM processor design.
- ❑ First, portable embedded systems require some form of **battery power**.
- ❑ The ARM processor has been specifically **designed to be small to reduce power consumption and extend battery operation** → essential for applications such as mobile phones and personal digital assistants (PDAs).
- ❑ **High code density** is another major requirement since embedded systems have limited memory due to cost and/or physical size restrictions.
- ❑ High code density is useful for applications that have **limited on-board memory**, such as mobile phones and mass storage devices.
- ❑ [Code density refers to **how many microprocessor instructions it takes to perform a requested action, and how much space each instruction takes up**. Generally speaking, **the less space an instruction takes and the more work per instruction that a microprocessor can do, the more dense its code is.**]

The ARM Design Philosophy ctd.

- ❑ Embedded systems are price sensitive .Hence, use slow and low-cost memory devices to get substantial savings—essential for high-volume applications like digital cameras.
- ❑ ARM has incorporated hardware debug technology within the processor so that software engineers can view what is happening while the processor is executing code. With greater visibility, software engineers can resolve issues faster.
- ❑ The ARM core is not a pure RISC architecture because of the constraints of its primary application—the embedded system.

Deviations of ARM instruction Set From Pure RISC nature

- ❑ The ARM instruction set differs from the pure RISC definition in several ways that make the ARM instruction set suitable for embedded applications:
 - 1. Variable cycle execution for certain instructions**
 - ❖ Not every ARM instruction executes in a single cycle.
 - 2. Inline barrel shifter leading to more complex instructions**
 - ❖ The inline barrel shifter is a hardware component that preprocesses one of the input registers before it is used by an instruction.

3. Thumb 16-bit instruction set

- ❖ ARM enhances the processor core by adding a second 16-bit instruction set called Thumb that permits the ARM core to execute either 16- or 32-bit instructions

4. Conditional execution

- ❖ An instruction is only executed when a specific condition has been satisfied. This feature improves performance and code density by reducing branch instructions.

5. Enhanced instructions

- ❖ The enhanced digital signal processor (DSP) instructions were added to the standard ARM instruction set to support fast 16×16 -bit multiplier operations. These instructions allow a faster-performing ARM processor

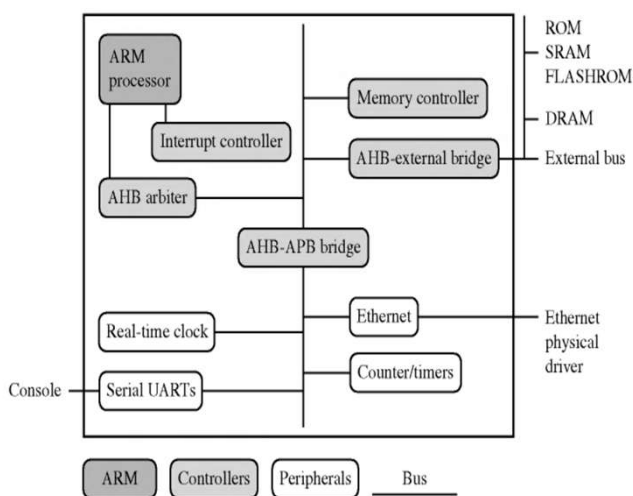
These additional features have made the ARM processor one of the most commonly used 32-bit embedded processor cores.

Many of the top semiconductor companies around the world produce products based around the ARM processor.

EMBEDDED SYSTEM HARDWARE

- ❑ Embedded systems can control many different devices, from small sensors found on a production line, to the real-time control systems used on a NASA space probe.
- ❑ All these devices use a combination of software and hardware components.
- ❑ Each component is chosen for efficiency and, if applicable, is designed for future extension and expansion.

Example of ARM based Embedded System



❑ Figure shows a typical embedded device based on an ARM core.

❑ Each box represents a feature or function.

❑ The lines connecting the boxes are the buses carrying data.

❑ We can separate the device into four main hardware components:

❑ **The ARM processor**

❑ **Controllers**

❑ **The peripherals**

❑ **Bus**

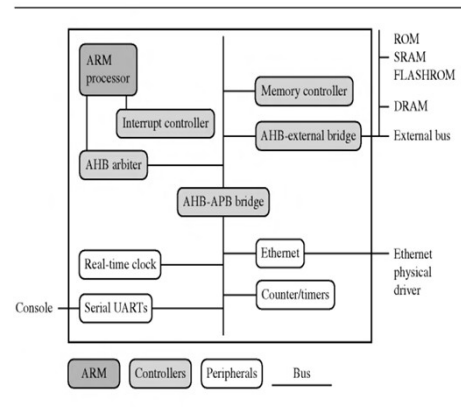
An example of an ARM-based embedded device, a microcontroller.

❑ The ARM processor controls the embedded device.

❖ Different versions of the ARM processor are available to suit the desired operating characteristics.

❑ An ARM processor comprises a core (the execution engine that processes instructions and manipulates data) plus the surrounding components that interface it with a bus.

❖ These components can include memory management and caches.

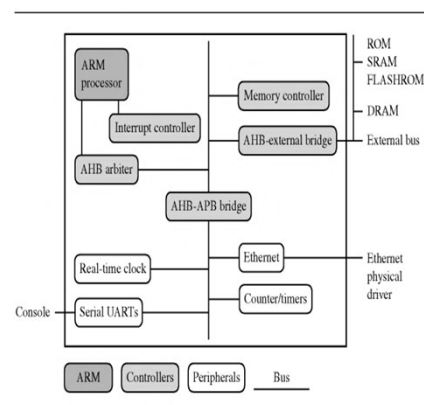


An example of an ARM-based embedded device, a microcontroller.

❑ **Controllers** coordinate important functional blocks of the system. Two commonly found controllers are interrupt and memory controllers.

❑ **The peripherals** provide all the input-output capability external to the chip and are responsible for the uniqueness of the embedded device.

❑ A **bus** is used to communicate between different parts of the device.



An example of an ARM-based embedded device, a microcontroller.

ARM Bus Technology

- ❑ Embedded systems use different bus technologies than those designed for x86 PCs.
- ❑ The most common PC bus technology, the Peripheral Component Interconnect (PCI) bus, connects such devices as video cards and hard disk controllers to the x86 processor bus.
- ❑ This type of technology is external or off-chip (i.e., the bus is designed to connect mechanically and electrically to devices external to the chip) and is built into the motherboard of a PC.
- ❑ Embedded devices use an on-chip bus that is internal to the chip and that allows different peripheral devices to be interconnected with an ARM core.

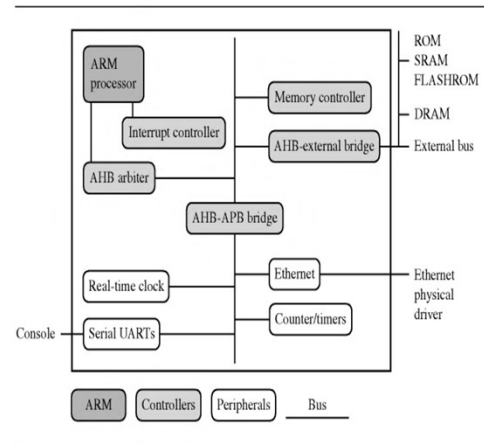
-
- ❑ There are two different classes of devices attached to the bus.
 - ❑ The **ARM processor core** is a **bus master** (a logical device capable of initiating a data transfer with another device across the same bus).
 - ❑ **Peripherals** tend to be **bus slaves** (logical devices capable only of responding to a transfer request from a bus master device.)
 - ❑ **A bus has two architecture levels.**
 - ❑ The first is a **physical level** that covers the **electrical characteristics** and bus width (16, 32, or 64 bits).
 - ❑ The second level deals with **protocol—the logical rules** that govern the communication between the processor and a peripheral

AMBA Bus Protocol

- ❑ The **Advanced Microcontroller Bus Architecture (AMBA)** was introduced in 1996 and has been widely adopted as the on-chip bus architecture used for ARM processors.
- ❑ The first AMBA buses introduced were the **ARM System Bus (ASB)** and the **ARM Peripheral Bus (APB)**.
- ❑ Later ARM introduced another bus design, called the **ARM High Performance Bus (AHB)**.
- ❑ Using AMBA, peripheral designers can reuse the same design on multiple projects.
 - ❑ A peripheral can simply be bolted onto the on-chip bus without having to redesign an interface for each different processor architecture

-
- ❑ ARM has introduced **two variations** on the AHB bus: **Multi-layer AHB** and **AHB-Lite**.
 - ❑ **Multi-layer AHB**
 - ❖ In contrast to the original AHB, which allows a single bus master to be active on the bus at any time, the Multi-layer AHB bus allows multiple active bus masters.
 - ❑ **AHB-Lite**.
 - ❖ AHB-Lite is a subset of the AHB bus and it is limited to a single bus master.
 - ❖ This bus was developed for designs that do not require the full features of the standard AHB bus.

- ❑ The example device shown in Figure has **three buses**:
- ❑ an **AHB bus** for the high- performance peripherals,
- ❑ an **APB bus** for the slower peripherals, and a third bus for external peripherals, proprietary to this device.
- ❑ This **external bus** requires a specialized bridge to connect with the AHB bus.



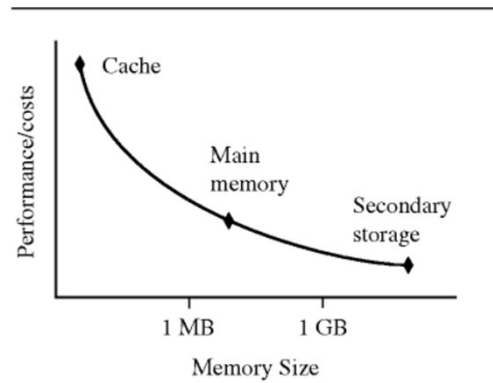
An example of an ARM-based embedded device, a microcontroller.

MEMORY

- ❑ **Memory Requirement:** An embedded system must have memory to store and execute code.
- ❑ **Comparison Factors:** When selecting memory, consider price, performance, and power consumption.
- ❑ **Memory Characteristics:**
 - ❑ Important memory characteristics include
 - ❖ **hierarchy,**
 - ❖ **width,** and
 - ❖ **type.**
- ❑ **Performance and Power:** If memory needs to run twice as fast to achieve desired bandwidth, power consumption may increase.

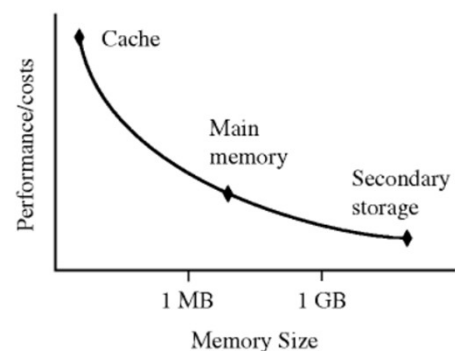
Memory Hierarchy

- ❑ All computer systems have memory arranged in some form of hierarchy.
- ❑ The Figure shows the **memory trade-offs**:
 - ❖ the fastest memory **cache** is physically **located nearer** the ARM processor core and
 - ❖ the slowest **secondary memory** is set **further away**.
- ❑ Generally the **closer** memory is to the **processor core**, the **more it costs** and the **smaller its capacity**



Storage trade-offs.

- ❑ **The cache** is placed between main memory and the core.
- ❑ It is used to *speed up data transfer between the processor and main memory*.
- ❑ **The main memory** is large—around 256 KB to 256 MB (or even greater), depending on the application—and is generally stored in separate chips.
- ❑ Load and store instructions access the main memory unless the values have been stored in the cache for fast access.
- ❑ **Secondary storage** is the largest and slowest form of memory. Hard disk drives and CD-ROM drives are examples of secondary storage.



Storage trade-offs.

Memory Width

- ❑ **Memory Width:** Number of bits memory returns per access, typically 8, 16, 32, or 64 bits.
- ❑ **Impact on Performance and Cost:** Memory width directly affects overall performance and cost ratio.
- ❑ **Uncached System:** Using 32-bit ARM instructions with 16-bit-wide memory chips requires two memory fetches per instruction (each fetch requires two 16-bit loads).
- ❑ This reduces system performance but makes 16-bit memory less expensive.
- ❑ **Thumb Instructions:** Using 16-bit Thumb instructions with 16-bit-wide memory improves performance.
- ❑ The core only needs to make a single fetch to memory to load an instruction.
- ❑ This results in better performance and reduced cost when using Thumb instructions with 16-bit-wide memory devices..

- ❑ Table 1.1 summarizes theoretical cycle times on an ARM processor using different memory width devices.

Table 1.1 Fetching instructions from memory.

Instruction size	8-bit memory	16-bit memory	32-bit memory
ARM 32-bit	4 cycles	2 cycles	1 cycle
Thumb 16-bit	2 cycles	1 cycle	1 cycle

Memory Types

□Types:

There are many different types of memory:

- 1. Read-only memory (ROM)** is a **memory** device or storage medium that stores information permanently.
 - ❖ The memory from which we can only read but cannot write on it.
 - ❖ ROMs are used in high-volume devices that require no updates or corrections.
- 2. Flash ROM** It can keep stored data and information even when the power is off.
 - ❖ It can be electrically erased and reprogrammed.
 - ❖ The erasing and writing of flash ROM are completely software controlled with no additional hardware required, which reduces the manufacturing costs.

Memory Types ctd.

- 3. Dynamic random access memory (DRAM)** is the most commonly used RAM for devices.
 - ❖ It has the lowest cost per megabyte compared with other types of RAM.
 - ❖ DRAM is dynamic—it needs to have its storage cells refreshed and given a new electronic charge every few milliseconds, so you need to set up a DRAM controller before using the memory.
- 4. Static random access memory (SRAM)** is faster than the more traditional DRAM, but requires more silicon area.
 - ❖ SRAM is static—the RAM does not require refreshing.
 - ❖ The access time for SRAM is considerably shorter than the equivalent DRAM.
 - ❖ But cost of SRAM is high.

Memory Types ctd.

- 5. Synchronous dynamic random access memory (SDRAM)** is one of many subcategories of DRAM.
- ❖ It can run at much higher clock speeds than conventional memory.
 - ❖ SDRAM synchronizes itself with the processor bus, because it is clocked.

Peripherals:

- Embedded systems that interact with the outside world need some form of peripheral device.
- A peripheral device performs input and output functions for the chip by connecting to other devices or sensors that are off-chip.
- Each peripheral device usually performs a single function and may reside on-chip.
- Peripherals range from a simple serial communication device to a more complex 802.11 wireless device.

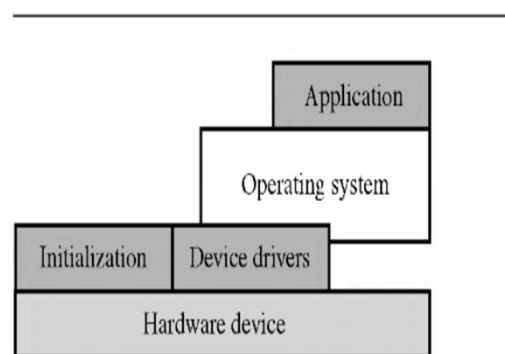
-
- ❑ **Controllers** are specialized peripherals that implement higher levels of functionality within an embedded system.
 - ❑ Two important types of controllers are
 - ❖ **Memory controllers** and
 - ❖ **Interrupt controllers.**

-
- ❖ **Memory Controllers:** Memory controllers connect different types of memory to the processor bus.
 - On power-up a memory controller is configured in hardware to allow certain memory devices to be active. These memory devices allow the initialization code to be executed.
 - ❖ **Interrupt Controllers:** When a peripheral or device requires attention, it raises an interrupt to the processor.
 - An interrupt controller provides a programmable governing policy that allows software to determine which peripheral or device can interrupt the processor at any specific time by setting the appropriate bits in the interrupt controller registers.

-
- ❑ There are **two types of interrupt controller** available for the ARM processor:
 - ❖ The standard interrupt controller and
 - ❖ The vector interrupt controller
1. **The standard interrupt controller** sends an interrupt signal to the processor core when an external device requests servicing.
 - ❖ The **interrupt handler** determines which device requires servicing by reading a **device bitmap register** in the interrupt controller.
 2. **The vector interrupt controller (VIC)** is more powerful than the standard interrupt controller, because it prioritizes interrupts and simplifies the determination of which device caused the interrupt.

Embedded System Software

- ❑ An embedded system needs software to drive it.
 - ❑ Fig. shows four typical software components required to control an embedded device.
1. **The initialization code**
 - ❖ is the first code executed on the board and is specific to a particular target or group of targets.
 - ❖ It sets up the minimum parts of the board before handing control over to the operating system.



Software abstraction layers executing on hardware.

2. The operating system

❖ provides an infrastructure to control applications and manage hardware system resources.

3. The device drivers

❖ provide a software interface to the peripheral devices

4. An application

❖ performs one of the tasks required for a device.

Initialization (Boot) Code

❑ **Initialization (Boot) Code** takes the processor from the reset state to a state where the operating system can run.

❖ It usually *configures the memory controller and processor caches and initializes some devices.*

❖ In a simple system the *operating system* might be replaced by a *simple scheduler or debug monitor.*

❑ The initialization code handles a number of *administrative tasks* prior to handing control over to an operating system image.

❖ These administrative tasks handled by initialization code can be grouped into three phases

1. **initial hardware configuration,**
2. **diagnostics,** and
3. **booting.**

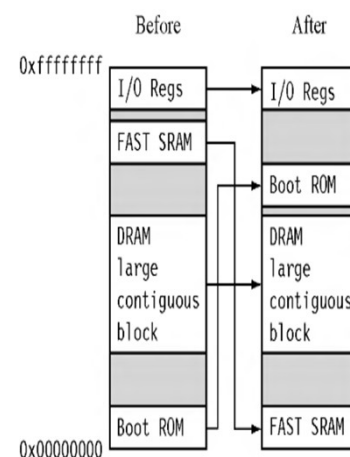
Initial hardware configuration

1. Initial hardware configuration

- ❖ involves *setting up the target platform* so it can boot an image.
- ❖ Although the target platform itself comes up in a **standard configuration**, this configuration normally **requires modification** to satisfy the requirements of the booted image.

❑ For example, the memory system normally requires reorganization of the memory map, as shown

- ❖ **Initializing** or **organizing memory** is an important part of the **initialization code** because many operating systems expect a known memory layout before they can start.
- ❖ It is common for ARM-based embedded systems to provide for **memory remapping** because it allows the system to start the initialization code from ROM at power-up.
- ❖ The initialization code then redefines or *remaps the memory map* to place RAM at address 0x00000000



2. Diagnostics

2. **Diagnostics** are often embedded in the initialization code.
 - ❖ Diagnostic code tests the system by exercising the hardware target to check if the target is in working order.
 - ❖ It also *tracks down standard system-related issues*.
 - ❖ The *primary purpose* of diagnostic code is *fault identification* and *isolation*.

Booting

3. **Booting** involves loading an image and handing control over to that image.
 - ❖ The boot process itself can be complicated if the system must boot different operating systems or different versions of the same operating system.
 - ❖ Booting an image is the final phase, but first you must *load* the image.
- ❑ **Loading an image** involves anything from copying an entire program including code and data into RAM, to just copying a data area containing volatile variables into RAM.
 - ❖ Once booted, the system hands over control by modifying the program counter to point into the start of the image.
 - ❖ Sometimes, to reduce the image size, an image is compressed. The image is then decompressed either when it is loaded or when control is handed over to it.

Operating System

- ❑ The initialization process prepares the hardware for an operating system to take control.
- ❑ *An operating system organizes the system resources: the peripherals, memory, and processing time.*
- ❑ With an operating system controlling these resources, they can be efficiently used by different applications running within the operating system environment.
- ❑ ARM processors support over 50 operating systems. We can divide operating systems into two main categories:
 1. **real-time operating systems (RTOSs)** and
 2. **platform operating systems.**

RTOS

- ❑ RTOSs provide *guaranteed response times* to events.
- ❑ Different operating systems have different amounts of control over the system response time.
- ❑ **A hard real-time** application requires a guaranteed response to work at all.
- ❑ In contrast, a **soft real-time** application requires a good response time, but the performance degrades more gracefully if the response time overruns.
 - ❖ **Hard Real-Time operating system:** These operating systems guarantee that critical tasks be completed within a range of time.
 - ❖ **Soft real-time operating system:** This operating system provides some relaxation in the time limit.

Platform operating systems

- ❑ **Platform operating systems** tend to have secondary storage. require a memory management unit to manage large, non-real-time applications
- ❑ The Linux operating system is a typical example of a platform operating system

Application:

- ❑ The operating system schedules **applications**:
- ❑ **Application** is a code dedicated to handling a particular task.
- ❑ An application implements a processing task; the operating system controls the environment.
- ❑ An embedded system can have one active application or several applications running simultaneously

Application area of ARM based Embedded System

- ❑ ARM processors are found in numerous market segments, including *networking, auto-motive, mobile and consumer devices, mass storage, and imaging*.
- ❑ ARM processor is found in networking applications like **home gateways, DSL modems** for highspeed Internet communication, and *802.11 wireless communications*.
- ❑ The *mobile device segment* is the largest application area for ARM processors, because of *mobile phones*.
- ❑ ARM processors are also found in *mass storage devices* such as *hard drives* and *imaging products* such as *inkjet printers*.

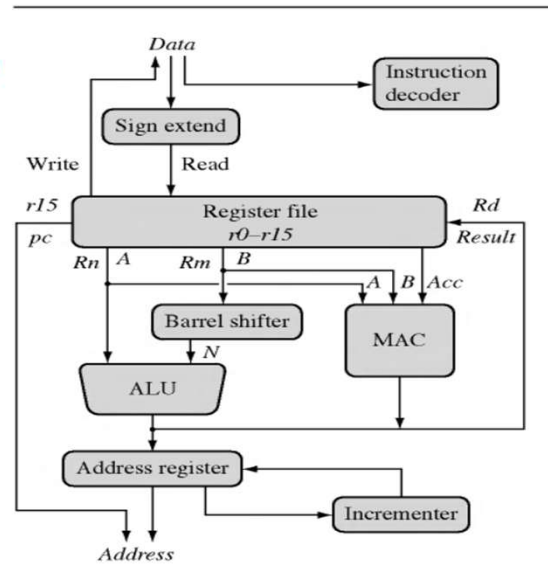
ARM Processor Fundamentals

- ❑ Previous section covered embedded systems with an ARM processor.
- ❑ Next we will deal with actual processor itself.
- ❑ First, we will provide an overview of the processor core and describe how data moves between its different parts.
- ❑ We will describe the programmer's model from a software developer's view of the ARM processor, which will show you the functions of the processor core and how different parts interact.

❑ A programmer can think of an ARM core as **functional units** connected by **data buses**, as shown in Figure

❑ The **arrows** represent the **flow of data**, the **lines** represent the **buses**, and the **boxes** represent either an **operation unit** or a **storage area**.

❑ The figure shows *not only the flow of data but also the abstract components that make up an ARM core*.



ARM core dataflow model.

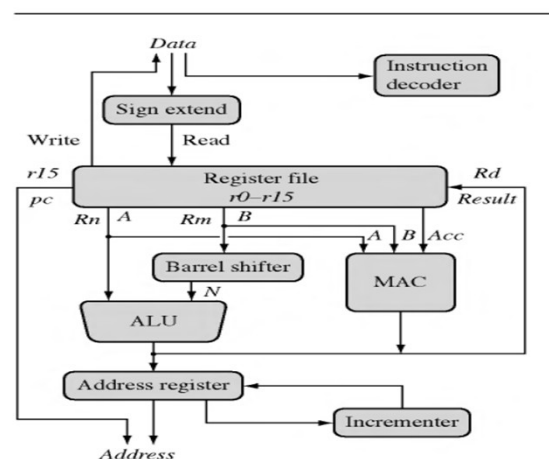
❑ Data enters the processor core through the Data bus. The data maybe an *instruction to execute* or a *data item*.

❑ Fig. shows a Von Neumann implementation of the ARM—

❖ data items and instructions share the same bus.

❖ In contrast, Harvard implementations of the ARM use two different buses.

❑ The **instruction decoder** translates instructions before they are executed. Each instruction executed belongs to a particular instruction set.

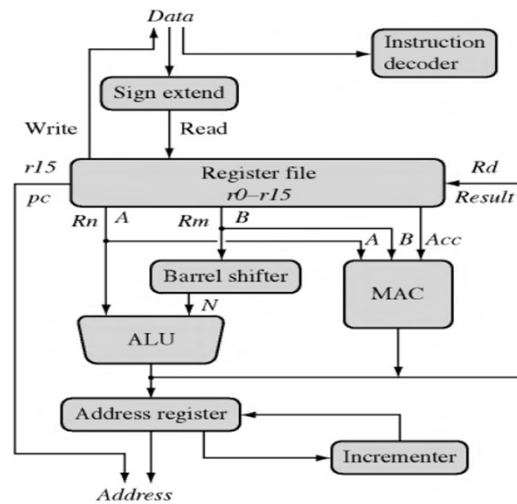


ARM core dataflow model.

❑ The ARM processor, like all RISC processors, uses a **load-store architecture**.

❑ This means it has two instruction types for transferring data in and out of the processor:

- ❖ **Load instructions** copy data from memory to registers in the core.
- ❖ **The Store instructions** copy data from registers to memory.
- ❖ There are no **data processing instructions** that *directly manipulate data in memory*.
- ❖ Thus, data processing is carried out solely in registers.



ARM core dataflow model.

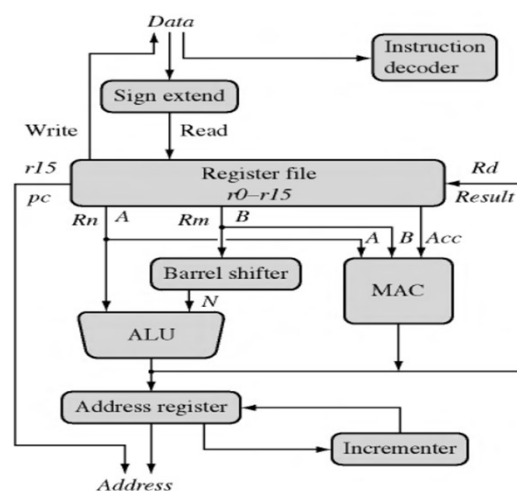
❑ Data items are placed in the **register file**—a storage bank made up of 32-bit registers.

❑ Since the ARM core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values.

❑ The **sign extend hardware** converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.

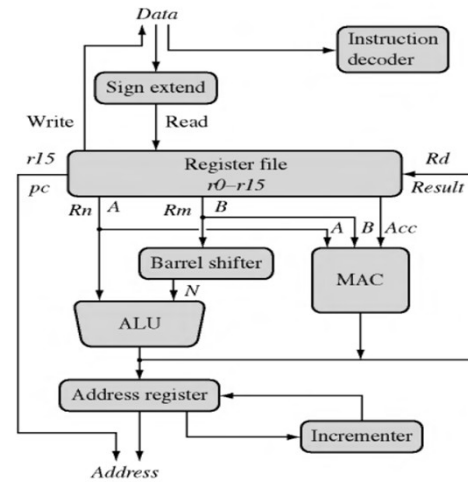
❑ ARM instructions typically have **two source registers, R_n and R_m** , and **a single result or destination register, R_d** .

❑ **Source operands** are read from the register file using the **internal buses A and B**, respectively.



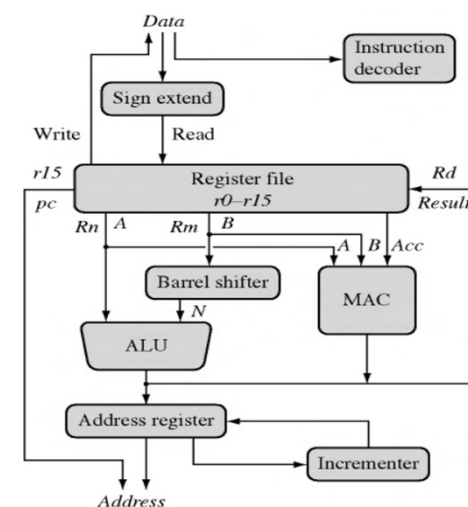
ARM core dataflow model.

- ❑ The **ALU (arithmetic logic unit)** or **MAC (multiply-accumulate unit)** takes the register values R_n and R_m from the A and B buses and computes a result.
- ❑ **Data processing instructions** write the result in R_d directly to the register file.
- ❑ **Load and store instructions** use the ALU to generate an address to be held in the address register and broadcast on the Address bus.
- ❑ One important feature of the ARM is that register R_m alternatively can be preprocessed in the **barrel shifter** before it enters the **ALU**.
- ❑ Together the *barrel shifter* and **ALU** can calculate a wide range of expressions and addresses.



ARM core dataflow model.

- ❑ For load and store instructions the incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location.
- ❑ The processor continues executing instructions until an exception or interrupt changes the normal execution flow.



ARM core dataflow model.

-
- ❑ Now we have got an overview of the processor core
 - ❑ Next we'll take a more detailed look at some of the key components of the processor:
 - ❖ **The Registers,**
 - ❖ **The Current Program Status Register (CPSR), and**
 - ❖ **The pipeline.**

-
- ❑ General-purpose registers hold either *data or an address*.
 - ❑ They are identified with the letter **r** prefixed to the register number. For example, **register 4** is given the label **r4**.
 - ❑ Fig. shows the active registers available in **user mode**—a protected mode normally used when *executing applications*.
 - ❑ The processor can operate in **seven** different modes, which we will be discussed later.
 - ❑ All the registers shown are **32 bits** in size.

r0	
r1	
r2	
r3	
r4	
r5	
r6	
r7	
r8	
r9	
r10	
r11	
r12	
r13 sp	
r14 lr	
r15 pc	
cpsr	
-	

Registers available in *user mode*.

- ❑ There are up to **18 active registers**:
 - ❖ **16 data registers** and **2 processor status registers**.
- ❑ **The data registers** are visible to the programmer as **r0 to r15**.
- ❑ The **ARM processor** has **three registers** assigned to a **particular task** or **special function**:
 - ❑ **r13, r14, and r15**.
 - ❑ They are frequently given **different labels** to differentiate them from the other registers

r0	
r1	
r2	
r3	
r4	
r5	
r6	
r7	
r8	
r9	
r10	
r11	
r12	
r13 sp	
r14 lr	
r15 pc	
cpsr	
-	

Registers available in *user mode*.

- ❑ In Fig., the shaded registers identify the **assigned special-purpose registers**:
- ❑ **Register r13** is traditionally used as the **stack pointer (sp)** and stores the head of the stack in the current processor mode.
- ❑ **Register r14** is called the **link register (lr)** and is where the core puts the **return address** whenever it calls a subroutine.
- ❑ **Register r15** is the **program counter (pc)** and contains the **address of the next instruction** to be fetched by the processor.

r0	
r1	
r2	
r3	
r4	
r5	
r6	
r7	
r8	
r9	
r10	
r11	
r12	
r13 sp	
r14 lr	
r15 pc	
cpsr	
-	

Registers available in *user mode*.

- ❑ Depending upon the context, *registers r13 and r14* can also be used as *general-purpose registers*, which can be particularly useful *since these registers are banked during a processor mode change*.
- ❑ It is dangerous to use *r13 as a general register* when the processor is running any form of *operating system* because *operating systems* often assume that *r13 always points to a valid stack frame*.
- ❑ In **ARM state** the registers **r0 to r13** are *orthogonal*—any instruction that you can apply to r0 you can equally well apply to any of the other registers.
- ❑ There are instructions that treat **r14** and **r15** in a *special way*.

r0	
r1	
r2	
r3	
r4	
r5	
r6	
r7	
r8	
r9	
r10	
r11	
r12	
r13 sp	
r14 lr	
r15 pc	
cpsr	
-	

Registers available in *user mode*.

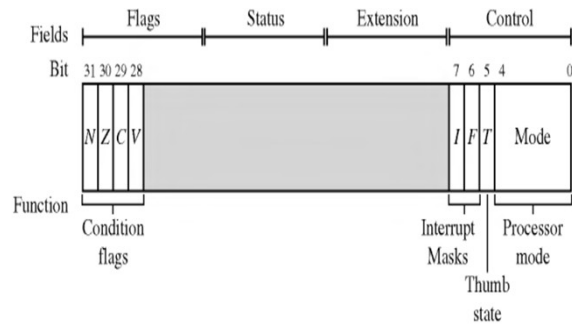
- ❑ In addition to the *16 data registers*, there are *two program status registers*:
- ❑ **CPSR & SPSR** (the current and saved program status registers, respectively).
- ❑ *The register file* contains all the registers available to a programmer.
- ❑ Which registers are visible to the programmer depend upon the current mode of the processor.

r0	
r1	
r2	
r3	
r4	
r5	
r6	
r7	
r8	
r9	
r10	
r11	
r12	
r13 sp	
r14 lr	
r15 pc	
cpsr	
-	

Registers available in *user mode*.

Current Program Status Register (CPSR)

- ❑ The ARM core uses the CPSR to monitor and control internal operations.
- ❑ The cpsr is a dedicated 32-bit register and resides in the register file.
- ❑ Fig shows the basic layout of a generic program status register.
- ❑ shaded parts are reserved for future expansion.

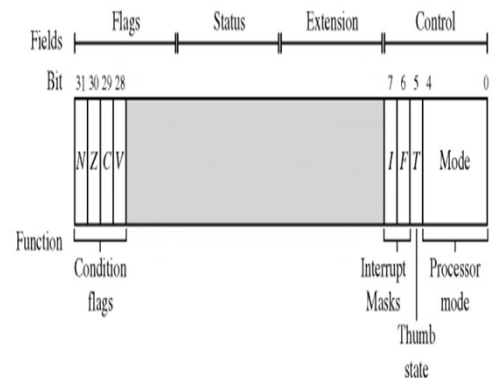


A generic program status register (*psr*).

- ❑ The cpsr is divided into **four fields**, each **8 bits wide**:

❖ **flags, status, extension, and control.**

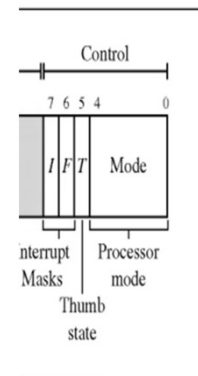
- ❑ In current designs **the extension and status fields are reserved for future use.**
- ❑ The **control field** contains **the processor mode, state, and interrupt mask bits.**
- ❑ **The flags field** contains the **condition flags.**
- ❑ Some ARM processor cores have extra bits allocated. For ex:, the **J bit**, which can be found in the flags field, is only available on **Jazelle-enabled processors**, which execute 8-bit instructions.



A generic program status register (*psr*).

Control field of cpsr

- ❑ We will start with control field of our cpsr
- ❑ it consist of
 - ❖ Processor Mode
 - ❖ State
 - ❖ Interrupt masks



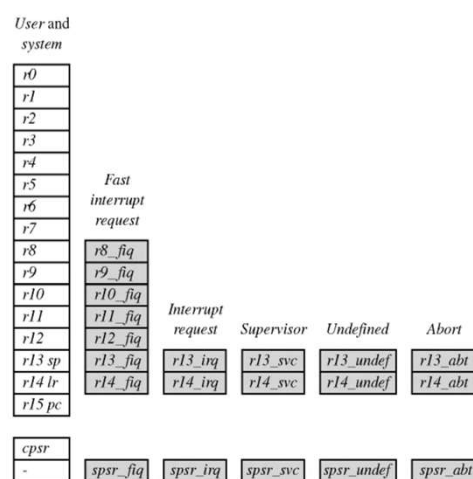
Processor Modes

- ❑ The **processor mode** determines *which registers are active* and *the access rights to the cpsr register itself*.
- ❑ Each processor mode is either *privileged or nonprivileged*:
- ❑ A **privileged** mode allows *full read-write access to the cpsr*.
- ❑ *a nonprivileged mode* only allows *read access to the control field in the cpsr* but still allows *read-write access to the condition flags*.
- ❑ *There are seven processor modes* in total:
 - ❑ **six privileged modes**
 - ❑ (abort, fast interrupt request, interrupt request, supervisor, system, and undefined)
 - ❑ *one nonprivileged mode (user)*.

- ❑ The processor enters **abort mode** when there is a failed attempt to access memory.
- ❑ **Fast interrupt request and interrupt request modes** correspond to the two interrupt levels available on the ARM processor.
- ❑ **Supervisor mode** is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in.
- ❑ **System mode** is a special version of user mode that allows full read-write access to the cpsr.
- ❑ **Undefined mode** is used when the processor encounters an instruction that is undefined or not supported by the implementation.
- ❑ **User mode** is used for programs and applications

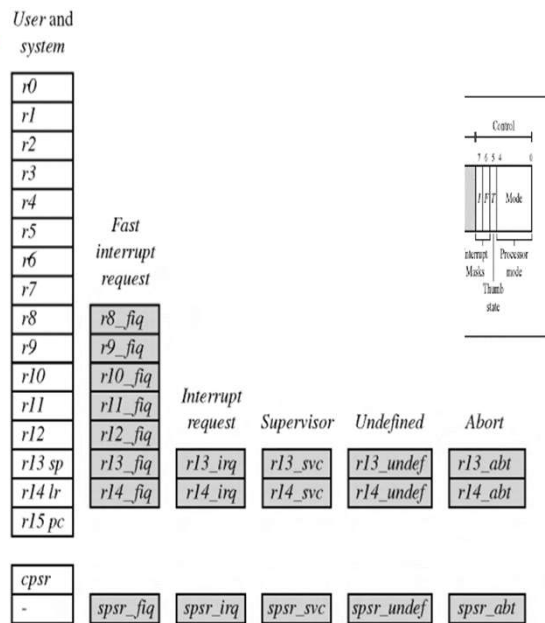
Banked Registers

- ❑ Fig. shows all 37 registers in the register file. Of those, 20 registers are hidden from a program at different times.
- ❑ These registers are called banked registers and are identified by the shading in the diagram.
- ❑ They are available only when the processor is in a particular mode

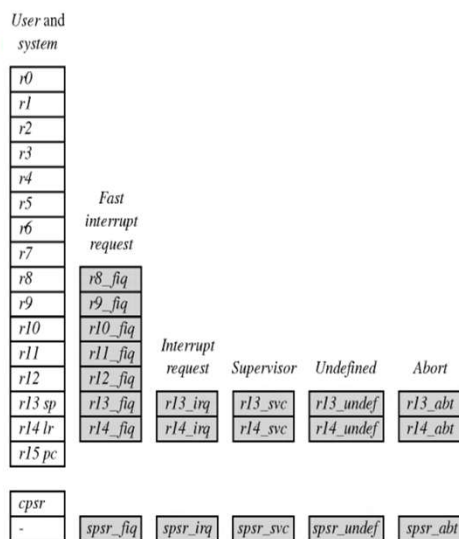


Complete ARM register set.

- ❑ Every processor mode except user mode can change mode by writing directly to the mode bits of the cpsr.
- ❑ All processor modes except system mode have a set of associated banked registers that are a subset of the main 16 registers.
- ❑ A banked register maps one-to-one onto a user mode register. If you change processor mode, a banked register from the new mode will replace an existing register.



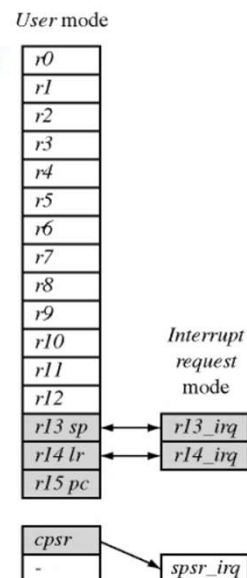
- ❑ For example, when the processor is in the interrupt request mode, the instructions you execute still access registers named r13 and r14.
- ❑ However, these registers are the banked registers r13_irq and r14_irq. The user mode registers r13_usr and r14_usr are not affected by the instruction referencing these registers.
- ❑ A program still has normal access to the other registers r0 to r12.



Complete ARM register set.

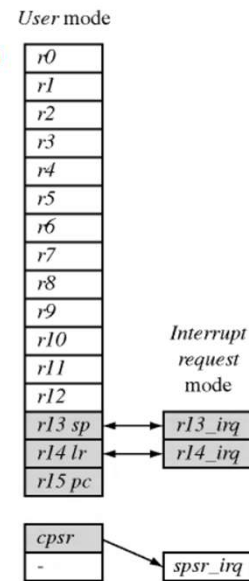
- ❑ The processor mode can be changed by a program that writes directly to the cpsr (the processor core has to be in privileged mode) or by hardware when the core responds to an exception or interrupt.
- ❑ The following exceptions and interrupts cause a mode change:
- ❑ **reset, interrupt request, fast interrupt request, software interrupt, data abort, prefetch abort, and undefined instruction.**
- ❑ Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location.

- ❑ Fig. illustrates what happens when an interrupt forces a mode change.
- ❑ The figure shows the core changing from user mode to interrupt request mode, which happens when an interrupt request occurs due to an external device raising an interrupt to the processor core.
- ❑ This change causes user registers r13 and r14 to be banked.
- ❑ The user registers are replaced with registers r13_irq and r14_irq, respectively.
- ❑ Note r14_irq contains the return address and r13_irq contains the stack pointer for interrupt request mode.



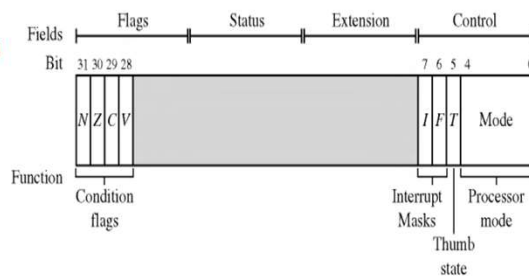
Changing mode on an exception.

- ❑ Fig. also shows a new register appearing in interrupt request mode: the saved program status register (spsr), which stores the previous mode's cpsr.
- ❑ You can see in the diagram the cpsr being copied into spsr_irq.
- ❑ To return back to user mode, a special return instruction is used that instructs the core to restore the original cpsr from the spsr_irq and bank in the user registers r13 and r14.
- ❑ Note that the spsr can only be modified and read in a privileged mode. There is no spsr available in user mode.
- ❑ Another important feature to note is that the cpsr is not copied into the spsr when a mode change is forced due to a program writing directly to the cpsr. The saving of the cpsr only occurs when an exception or interrupt is raised

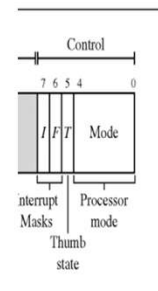


Changing mode on an exception.

- ❑ Fig. shows that the current active processor mode occupies the five least significant bits of the cpsr.
- ❑ When power is applied to the core, it starts in supervisor mode, which is privileged.
- ❑ Starting in a privileged mode is useful since initialization code can use full access to the cpsr to set up the stacks for each of the other modes.



A generic program status register (psr).



Processor mode.

Mode	Abbreviation	Privileged	Mode[4:0]
<i>Abort</i>	abt	yes	10111
<i>Fast interrupt request</i>	fiq	yes	10001
<i>Interrupt request</i>	irq	yes	10010
<i>Supervisor</i>	svc	yes	10011
<i>System</i>	sys	yes	11111
<i>Undefined</i>	und	yes	11011
<i>User</i>	usr	no	10000

- Table lists the various modes and the associated binary patterns.
- The last column of the table gives the bit patterns that represent each of the processor modes in the cpsr

STATE AND INSTRUCTION SETS

- The state of the core determines which instruction set is being executed. There are three instruction sets:
 - ❖ **ARM,**
 - ❖ **Thumb, and**
 - ❖ **Jazelle.**
- The *ARM instruction set* is only active when the processor is in **ARM state**.
- Thumb instruction set* is only active when the processor is in **Thumb state**.
- Once in Thumb state the processor is executing purely Thumb 16-bit instructions.
- You cannot *intermingle sequential ARM, Thumb, and Jazelle* instructions.

ARM STATE

- The Jazelle J and Thumb T bits in the cpsr reflect the state of the processor.
- When both J and T bits are 0, the processor is in ARM state and executes ARM instructions.
- This is the case when power is applied to the processor

THUMB STATE

- When the T bit is 1, then the processor is in Thumb state.
- To change states the core executes a specialized branch instruction.

Comparison of ARM & THUMB Instruction

ARM and Thumb instruction set features.

	ARM (<i>cpsr T = 0</i>)	Thumb (<i>cpsr T = 1</i>)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution ^a	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers + <i>pc</i>	8 general-purpose registers +7 high registers + <i>pc</i>

JAZELLE INSTRUCTIONS

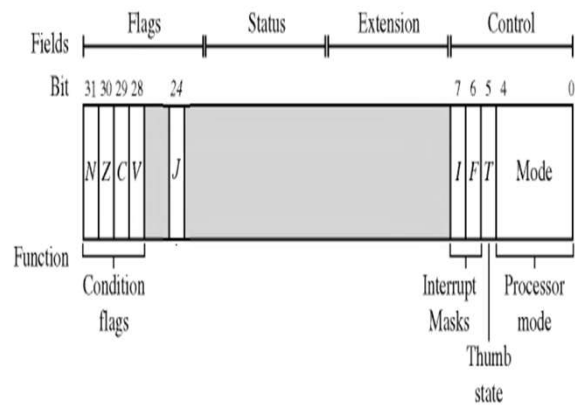
- ❑ The ARM designers introduced a third instruction set called Jazelle.
- ❑ Jazelle executes 8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java bytecodes.
- ❑ To execute Java bytecodes, you require the Jazelle technology plus a specially modified version of the Java virtual machine.
- ❑ It is important to note that the hardware portion of Jazelle only supports a subset of the Java bytecodes; the rest are emulated in software.

Jazelle instruction set features.

	Jazelle (<i>cpsr T = 0, J = 1</i>)
Instruction size	8-bit
Core instructions	Over 60% of the Java bytecodes are implemented in hardware; the rest of the codes are implemented in software.

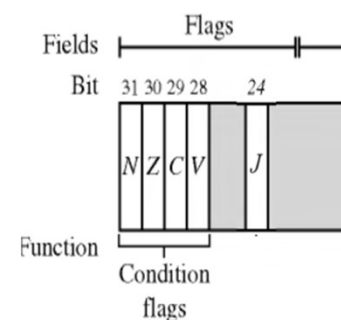
Interrupt Masks

- ❑ Interrupt masks are used to stop specific interrupt requests from interrupting the processor.
- ❑ There are two interrupt request levels available on the ARM processor core—
- ❑ Interrupt request (IRQ) and fast interrupt request (FIQ).
- ❑ The cpsr has two interrupt mask bits, 7 and 6 (or I and F), which control the masking of IRQ and FIQ, respectively.
- ❑ The I bit masks IRQ when set to binary 1, and similarly the F bit masks FIQ when set to binary 1.



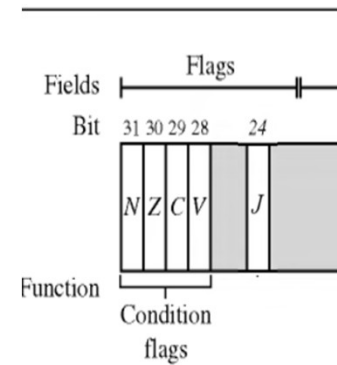
Condition Flags

- ❑ Condition flags are updated by *comparisons* and the *result of ALU operations* that specify the *S instruction suffix*.
- ❑ For example, if a **SUBS** subtract instruction results in a register value of **zero**, then the **Z flag in the cpsr is set**.
- ❑ This particular subtract instruction specifically updates the cpsr.

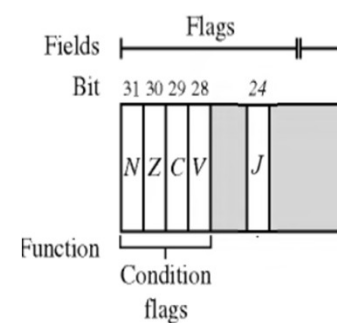


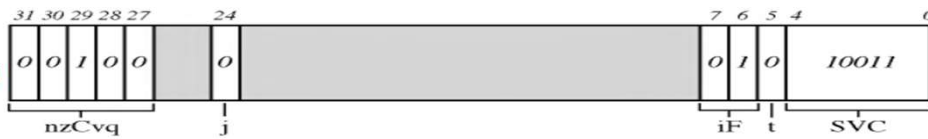
Condition flags.

Flag	Flag name	Set when
Q	Saturation	the result causes an overflow and/or saturation
V	oVerflow	the result causes a signed overflow
C	Carry	the result causes an unsigned carry
Z	Zero	the result is zero, frequently used to indicate equality
N	Negative	bit 31 of the result is a binary 1



- ❑ With processor cores that include the DSP extensions, the Qbit indicates if an overflow or saturation has occurred in an enhanced DSP instruction.
- ❑ The flag is "sticky" in the sense that the hardware only sets this flag. To clear the flag you need to write to the cpsr directly.
- ❑ In Jazelle-enabled processors, the J bit reflects the state of the core; if it is set, the core is in Jazelle state.
- ❑ The J bit is not generally usable and is only available on some processor cores.
- ❑ To take advantage of Jazelle, extra software has to be licensed from both ARM Limited and Sun Microsystems.





Example: $cpsr = nzCvqjiFt_SVC$.

- Fig shows a typical value for the cpsr with both DSP extensions and Jazelle.
- When a bit is a binary 1 we use a capital letter; when a bit is a binary 0, we use a lowercase letter.
- For the condition flags a capital letter shows that the flag has been set.
- For interrupts a capital letter shows that an interrupt is disabled.
- In the cpsr example shown in Fig, the C flag is the only condition flag set. The rest nzvq flags are all clear.
- The processor is in ARM state because neither the Jazelle j or Thumb t bits are set.
- The IRQ interrupts are enabled, and FIQ interrupts are disabled

- In the cpsr example shown in Fig. the C flag is the only condition flag set.

- The rest nzvq flags are all clear.

- The processor is in ARM state because neither the Jazelle j or Thumb t bits are set.

- The IRQ interrupts are enabled, and FIQ interrupts are disabled.

- We can see from the figure the processor is in supervisor (SVC) mode since the mode[4:0] is equal to binary 10011.



Example: $cpsr = nzCvqjiFt_SVC$.

Conditional execution

- ❑ Conditional execution controls whether or not the core will execute an instruction.
- ❑ Most instructions have a **condition attribute** that determines if the core will execute it based on the setting of the condition flags.
- ❑ Prior to execution, the processor compares the **condition attribute** with the **condition flags** in the cpsr. If they match, then the instruction is executed; otherwise the instruction is ignored.

- ❑ The condition attribute is postfixed to the **instruction mnemonic**, which is encoded into the instruction.

- ❑ Table lists the conditional execution code mnemonics.

- ❑ When a condition mnemonic is not present, the default behavior is to set it to always (AL) execute.

Mnemonic	Name	Condition flags
EQ	equal	Z
NE	not equal	z
CS HS	carry set/unsigned higher or same	C
CC LO	carry clear/unsigned lower	c
MI	minus/negative	N
PL	plus/positive or zero	n
VS	overflow	V
VC	no overflow	v
HI	unsigned higher	zC
LS	unsigned lower or same	Z or c
GE	signed greater than or equal	NV or nV
LT	signed less than	Nv or nV
GT	signed greater than	NzV or nzv
LE	signed less than or equal	Z or Nv or nV
AL	always (unconditional)	ignored

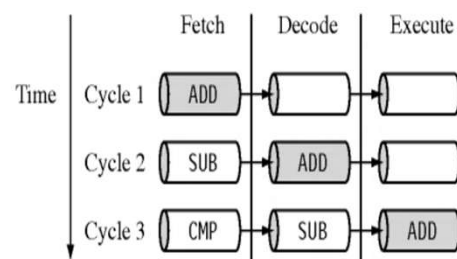
PIPELINE

- ❑ A pipeline is the mechanism a RISC processor uses to execute instructions.
- ❑ Using a pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed.
- ❑ Fig. shows a three-stage pipeline:
- ❑ **Fetch** loads an instruction from memory.
- ❑ **Decode** identifies the instruction to be executed.
- ❑ **Execute** processes the instruction and writes the result back to a register.



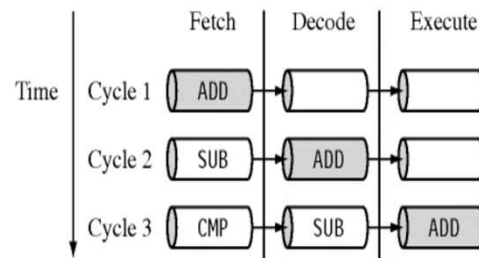
ARM7 Three-stage pipeline.

- ❑ Fig. illustrates the pipeline using a simple example.
- ❑ It shows a sequence of three instructions being fetched, decoded, and executed by the processor.
- ❑ Each instruction takes a single cycle to complete after the pipeline is filled.
- ❑ The three instructions are placed into the pipeline sequentially.



Pipelined instruction sequence.

- ❑ In the first cycle the core fetches the ADD instruction from memory.
- ❑ In the second cycle the core fetches the SUB instruction and decodes the ADD instruction.
- ❑ In the third cycle, both the SUB and ADD instructions are moved along the pipeline. The ADD instruction is executed, the SUB instruction is decoded, and the CMP instruction is fetched.
- ❑ This procedure is called filling the pipeline. The pipeline allows the core to execute an instruction every cycle



Pipelined instruction sequence.

- ❑ As the pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain **a higher operating frequency**. This in turn **increases the performance**.
- ❑ The **system latency also increases** because **it takes more cycles to fill the pipeline** before the core can execute an instruction.
- ❑ The increased pipeline length also means there can be data dependency between certain stages.
- ❑ You can write code to reduce this dependency by using **instruction scheduling**

ARM 9 PIPELINE

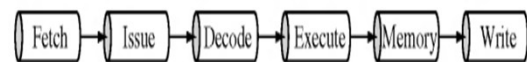
- ❑ The pipeline design for each ARM family differs.
- ❑ For example, The ARM9 core increases the pipeline length to five stages, as shown in Fig.
- ❑ The ARM9 adds a memory and writeback stage, which allows the ARM9 to process on average *1.1 Dhrystone MIPS per MHz*—an increase in instruction throughput by around *13% compared* with an ARM7.
- ❑ Dhrystone MIPS (Million Instructions Per Second) is a benchmark used to measure the performance of computer processors
- ❑ The Dhrystone benchmark measures the number of integer operations a processor can perform in one second,
- ❑ The maximum *core frequency* attainable using an ARM9 is also higher.



ARM9 five-stage pipeline.

ARM 10 PIPELINE

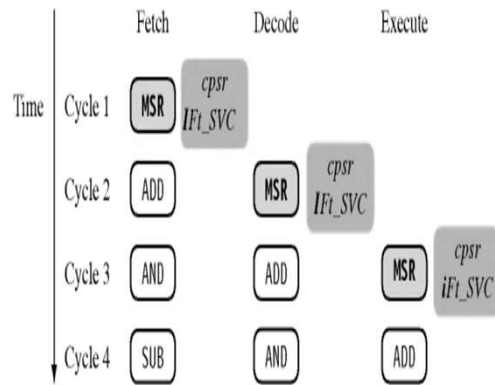
- ❑ The ARM 10 increases the pipeline length still further by adding a sixth stage, as shown in Fig.
- ❑ The ARM10 can process on average 1.3 Dhrystone MIPS per MHz, about 34% more throughput than an ARM7 processor core, but again at a higher latency cost.
- ❑ Even though the ARM9 and ARM 10 pipelines are different, they still use the same pipeline executing characteristics as an ARM7.
- ❑ Code written for the ARM7 will execute on an ARM9 or ARM10.




ARM10 six-stage pipeline.

Pipeline Executing Characteristics

- ❑ The ARM pipeline has not processed an instruction until it passes completely through the execute stage.
- ❑ For example, an ARM7 pipeline (with three stages) has executed an instruction only when the fourth instruction is fetched.
- ❑ Fig. shows an instruction sequence on an ARM7 pipeline.
- ❑ The MSR instruction is used to enable IRQ interrupts, which only occurs once the MSR instruction completes the execute stage of the pipeline.
- ❑ It clears the I bit in the cpsr to enable the IRQ interrupts.
- ❑ Once the ADD instruction enters the execute stage of the pipeline, IRQ interrupts are enabled.

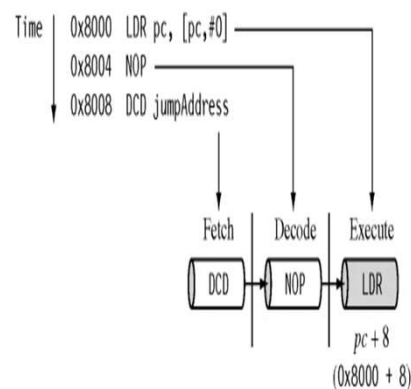


ARM instruction sequence.

Note: 

- In ARM architecture, MSR stands for "Move to Status Register" instruction. It is used to move data into the Current Program Status Register (CPSR) or the Saved Program Status Register (SPSR).
- The MSR instruction allows you to set specific fields in the CPSR or SPSR, such as condition code flags, mode bits, or interrupt disable bits. This instruction is commonly used for tasks such as changing the processor mode or modifying the condition flags for conditional branching.

- ❑ Fig. illustrates the use of the pipeline and the program counter pc. In the execute stage, the pc always points to the address of the instruction plus 8 bytes.
- ❑ In other words, the pc always points to the address of the instruction being executed plus two instructions ahead.
- ❑ This is important when the pc is used for calculating a relative offset and is an architectural characteristic across all the pipelines.
- ❑ Note when the processor is in Thumb state the pc is the instruction address plus 4.



Exceptions, Interrupts, and the Vector Table

- ❑ When an exception or interrupt occurs, the processor sets the pc to a specific memory address.
- ❑ The address is within a special address range called *the vector table*.
- ❑ The memory map address **0x00000000** is reserved for the **vector table**, a set of 32-bit words.
- ❑ On some processors the vector table can be optionally located at a higher address in memory (starting at the offset **0xffff0000**). Operating systems such as Linux and Microsoft's embedded products can take advantage of this feature.
- ❑ When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table :

The vector table.

Exception/interrupt	Shorthand	Address	High address
Reset	RESET	0x00000000	0xffff0000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	DABT	0x00000010	0xffff0010
Reserved	—	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

DIFFERENT Exception/ Interrupt Vectors

❑ Reset vector

- ❖ is the location of the first instruction executed by the processor *when power is applied*.
- ❖ This instruction branches to the initialization code.

❑ Undefined instruction vector

- ❖ is used when the processor cannot decode an instruction.

❑ Software interrupt vector

- ❖ is called when you execute a SWI instruction.
- ❖ The SWI instruction is frequently used as the mechanism to invoke an operating system routine.

❑ Prefetch abort vector occurs when the processor attempts to fetch an instruction from an address without the correct access permissions.

- ❖ The actual abort occurs in the decode stage.

❑ Data abort vector

❖ is similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions.

❑ Interrupt request vector

❖ is used by external hardware to interrupt the normal execution flow of the processor.

❖ It can only be raised if IRQs are not masked in the cpsr.

❑ Fast interrupt request vector

❖ is similar to the interrupt request but is reserved for *hardware requiring faster response times*.

❖ It can only be raised if FIQs are not masked in the cpsr.

Core Extensions

❑ The hardware extensions covered in this section are *standard components placed next to the ARM core*.

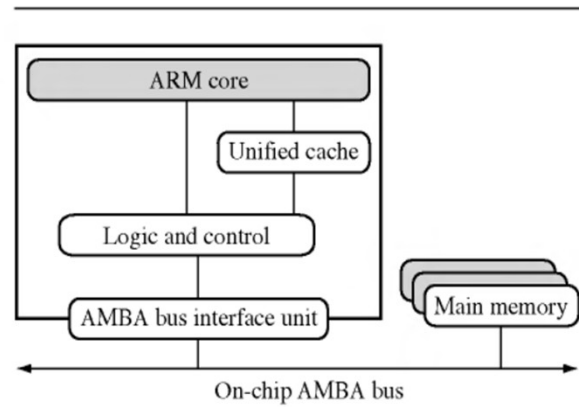
❑ They improve *performance, manage resources, and provide extra functionality* and are designed to provide flexibility in handling particular applications.

❑ There are *three hardware extensions* ARM wraps around the core:

1. *Cache And Tightly Coupled Memory,*
2. *Memory Management, and the*
3. *Coprocessor Interface.*

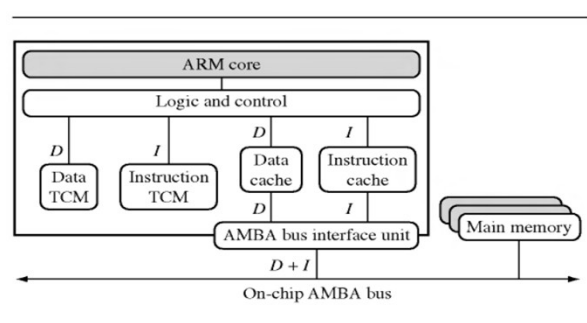
Cache and Tightly Coupled Memory

- ❑ The *cache* is a *block of fast memory placed between main memory and the core*.
- ❑ It allows for more efficient fetches from some memory types.
- ❑ With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory.
- ❑ ARM has *two forms of cache*.
- ❑ The first is found attached to the *Von Neumann-style cores*. It combines both data and instruction into a single unified cache, as shown in Figure



A simplified Von Neumann architecture with cache.

- ❑ The second form of Cache is attached to the Harvard-style cores, has separate caches for data and instruction.
- ❑ A cache provides an overall increase in performance but at the expense of predictable execution



A simplified Harvard architecture with caches and TCMs.

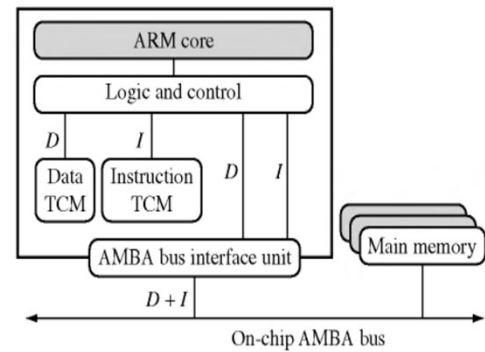
❑ But for real-time systems it is important that code execution is *deterministic*—the time taken for loading and storing instructions or data must be predictable.

❑ This is achieved using a form of memory called tightly coupled memory (TCM).

❑ TCM is fast SRAM located close to the core and guarantees the clock cycles required to fetch instructions or data—critical for real-time algorithms requiring deterministic behavior.

❑ TCMs appear as memory in the address map and can be accessed as fast memory.

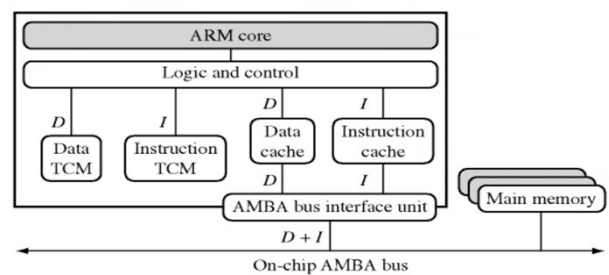
❑ An example of a processor with TCMs is shown in Fig



A simplified Harvard architecture with TCMs.

❑ By combining both technologies, ARM processors can have both improved performance and predictable real-time response.

❑ Figure 2 shows an example core with a combination of caches and TCMs.



A simplified Harvard architecture with caches and TCMs.

2.Memory Management

- ❑ Embedded systems often use multiple memory devices.
- ❑ It is usually necessary to have a method to help organize these devices and protect the system from applications trying to make inappropriate accesses to hardware.
- ❑ This is achieved with the assistance of memory management hardware.
- ❑ ARM cores have *three different types of memory management hardware*—
 1. **No Extensions** providing no protection,
 2. a **Memory Protection Unit (MPU)** providing limited protection, and
 3. a **Memory Management Unit (MMU)** providing full protection

❑ *Nonprotected memory*

- ❖ is fixed and provides very little flexibility.
- ❖ It is normally used for small, simple embedded systems that require no protection from rogue applications

❑ *MPUs*

- ❖ employ a simple system that uses a *limited number of memory regions*.
- ❖ These regions are controlled with a set of *special coprocessor registers*, and each region is defined with *specific access permissions*.
- ❖ This type of memory management *is used for systems that require memory protection but don't have a complex memory map*.

❑ *MMUs*

- ❖ are the most comprehensive memory management hardware available on the ARM.
- ❖ The MMU uses a set of *translation tables* to provide fine-grained control over memory.
- ❖ These tables are stored in *main memory* and provide a *virtual-to-physical address map as well as access permissions*.
- ❖ MMUs are designed for more sophisticated platform operating systems that support multitasking

3. Coprocessors

- ❑ *Coprocessors* can be attached to the ARM processor. A coprocessor extends the processing features of a core by extending the instruction set or by providing configuration registers.
- ❑ for example, *coprocessor 15*: The ARM processor uses coprocessor 15 registers to control the cache, TCMs, and memory management.
- ❑ The coprocessor can also extend the instruction set by providing a specialized group of new instructions.
- ❑ For example, there are a set of specialized instructions that can be added to the standard ARM instruction set to process vector floating-point (VFP) operations.

-
- ❑ These new instructions are processed in the decode stage of the ARM pipeline.
 - ❑ If the decode stage sees a coprocessor instruction, then it offers it to the relevant coprocessor.
 - ❑ But if the coprocessor is not present or doesn't recognize the instruction, then the ARM takes an undefined instruction exception, which allows you to emulate the behavior of the coprocessor in software.