

ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028



A T M E

College of Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(ACADEMIC YEAR 2022-23)

LESSON NOTES

SUBJECT: MICROCONTROLLER AND EMBEDDED SYSTEMS

SUB CODE: 21CS43

SEMESTER: IV

INSTITUTIONAL MISSION AND VISION

Objectives

- To provide quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To cultivate strong community relationships and involve the students and the staff in local community service.
- To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

Vision

- Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

Department of Computer Science & Engineering

Vision of the Department

- develop highly talented individuals in Computer Science and Engineering to deal with real world challenges in industry, education, research and society.

Mission of the Department

- To inculcate professional behaviour, strong ethical values, innovative research capabilities and leadership abilities in the young minds & to provide a teaching environment that emphasizes depth, originality and critical thinking.
- Motivate students to put their thoughts and ideas adoptable by industry or to pursue higher studies leading to research

Program Educational Objectives (PEO'S):

1. Empower students with a strong basis in the mathematical, scientific and engineering fundamentals to solve computational problems and to prepare them for employment, higher learning and R&D.
2. Gain technical knowledge, skills and awareness of current technologies of computer science engineering and to develop an ability to design and provide novel engineering solutions for software/hardware problems through entrepreneurial skills.
3. Exposure to emerging technologies and work in teams on interdisciplinary projects with effective communication skills and leadership qualities.
4. Ability to function ethically and responsibly in a rapidly changing environment by applying innovative ideas in the latest technology, to become effective professionals in Computer Science to bear a life-long career in related areas.

Program Specific Outcomes (PSOs)

1. Demonstrate understanding of the principles and working of the hardware and software aspects of Embedded Systems.
2. Use professional Engineering practices, strategies and tactics for the development, implementation and maintenance of software.
3. Provide effective and efficient real time solutions using acquired knowledge in various domains.

MICROCONTROLLER AND EMBEDDED SYSTEMS (Effective from the academic year 2018 -2019) SEMESTER – IV			
Course Code	21CS43	CIE Marks	50
Number of Contact Hours/Week	3:0:2:0	SEE Marks	50
Total Number of Contact Hours	40T +20 P	Exam Hours	03
CREDITS –3			
Course Learning Objectives: This course (18CS44) will enable students to:			
<ul style="list-style-type: none"> Understand the fundamentals of ARM based systems, s, including programming modules with registers and the CPSR. Use the various instructions to program the ARM controller Program various embedded components using the embedded C program Identify various components, their purpose, and their application to the embedded system's applicability. Understand the embedded system's real-time operating system and its application in IoT 			
Module 1			Contact Hours
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 Textbook 1: Chapter 3: Sections 3.1 to 3.6 (Excluding .5.2), Chapter 5 RBT: L1, L2			08
Module 2			
Introduction to the ARM Instruction Set : Introduction to the ARM Instruction Set: Data Processing Instructions , Branch Instructions, Software Interrupt Instructions, Program Status Register Instructions, Coprocessor Instructions, Loading Constants. C Compilers and Optimization :Basic C Data Types, C Looping Structures, Register Allocation, Function Calls, Pointer Aliasing, Textbook 1: Chapter 3: Sections 3.1 to 3.6 (Excluding 3.5.2), Chapter 5 RBT: L1, L2			08
Module 3			
C Compilers and Optimization : Structure Arrangement, Bit-fields, Unaligned Data and Endianness, Division, Floating Point, Inline Functions and Inline Assembly, Portability Issues. ARM programming using Assembly language: Writing Assembly code, Profiling and cycle counting, instruction scheduling, Register Allocation, Conditional Execution, Looping Constructs Textbook 1: Chapter-5,6 RBT: L1, L2			08
Module 4			
Embedded System Components: Embedded Vs General computing system, History of embedded systems, Classification of Embedded systems, Major applications areas of embedded systems, purpose of embedded systems Core of an Embedded System including all types of processor/controller, Memory, Sensors, Actuators, LED, 7 segment LED display, stepper motor, Keyboard, Push button switch, Communication Interface (onboard and external types), Embedded firmware, Other system components.			08

Text book 2: Chapter-3, Chapter-4, Chapter-7 (Sections 7.1, 7.2 only), Chapter-9 (Sections 9.1, 9.2, 9.3.1, 9.3.2 only) RBT: L1, L2	
RTOS and IDE for Embedded System Design: Operating System basics, Types of operating systems, Task, process and threads (Only POSIX Threads with an example program), Thread preemption, Multiprocessing and Multitasking, Task Communication (without any program), Task synchronization issues – Racing and Deadlock, Concept of Binary and counting semaphores (Mutex example without any program), How to choose an RTOS, Integration and testing of Embedded hardware and firmware, Embedded system Development Environment – Block diagram (excluding Keil), Disassembler/decompiler, simulator, emulator and debugging techniques, target hardware debugging, boundary scan. Text book 2: Chapter-10 (Sections 10.1, 10.2, 10.3, 10.4 , 10.7, 10.8.1.1, 10.8.1.2, 10.8.2.2, 10.10 only), Chapter 12, Chapter-13 (block diagram before 13.1, 13.3, 13.4, 13.5, 13.6 only) RBT: L1, L2	08
Course Outcomes: The student will be able to :	
<ul style="list-style-type: none"> ● Explain C-Compilers and optimization ● Describe the ARM microcontroller's architectural features and program module. ● Apply the knowledge gained from programming on ARM to different applications. ● Program the basic hardware components and their application selection method. ● Demonstrate the need for a real-time operating system for embedded system applications. 	
Textbooks:	
<ol style="list-style-type: none"> 1. Textbooks: 2. Andrew N Sloss, Dominic Symes and Chris Wright, ARM system developers guide, Elsevier, Morgan Kaufman publishers, 2008. 3. Shibu K V, “Introduction to Embedded Systems”, Tata McGraw Hill Education, Private Limited, 2nd Edition. 	
Reference Books:	
<ol style="list-style-type: none"> 1. Raghunandan..G.H, Microcontroller (ARM) and Embedded System, Cengage learning Publication, 2019 2. The Insider’s Guide to the ARM7 Based Microcontrollers, Hitex Ltd., 1st edition, 2005. 3. Steve Furber, ARM System-on-Chip Architecture, Second Edition, Pearson, 2015. 4. Raj Kamal, Embedded System, Tata McGraw-Hill Publishers, 2nd Edition, 2008. 	

MODULE-I

ARM Embedded Systems

The ARM processor core is a key component of many successful 32-bit embedded systems. ARM's designers have come a long way from the first ARM1 prototype in 1985. Over one billion ARM processors had been shipped worldwide by the end of 2001. The ARM core is not a single core, but a whole family of designs sharing similar design principles and a common instruction set.

The RISC design philosophy

The ARM core uses 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.

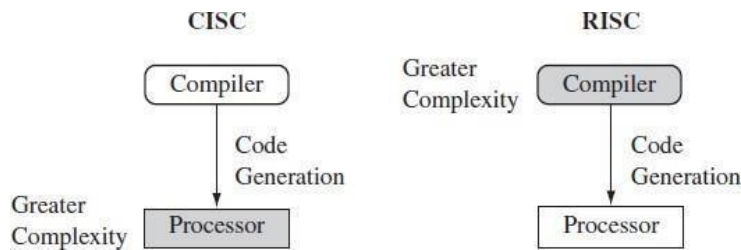
The RISC philosophy is implemented with four major design rules:

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 (for example, a divide operation) by combining several simple instructions. Each instruction is a fixed length to allow the pipeline to fetch future instructions before decoding the current instruction.
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.
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.
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.

➤ Differences between CISC & RISC

CISC	RISC
<ul style="list-style-type: none">• Emphasis on hardware• Includes multi-clock complex instructions• Data transfer between memory & register incorporated in available instructions• Small code sizes, high cycles per second• Transistors used for storing complex instructions	<ul style="list-style-type: none">Emphasis on softwareSingle-clock, reduced instruction onlySeparate instructions available: "LOAD" and "STORE" are independent instructionsLow cycles per second & large code sizesSpends more transistors on registers
<ul style="list-style-type: none">• mov ax, 10 mov bx, 5 mul bx, ax	<ul style="list-style-type: none">Ex: mov ax, 0 mov bx, 10 mov cx, 5 Begin: add ax, bx

loop Begin



CISC emphasizes hardware complexity. RISC emphasizes compiler.

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.
- In addition, embedded systems are price sensitive and use slow and low-cost memory devices. The ability to use low-cost memory devices produces substantial savings.
- Another important requirement is to reduce the area of the die taken up by the embedded processor. For a single-chip solution, the smaller the area used by the embedded processor, the more available space for specialized peripherals. This in turn reduces the cost of the design and manufacturing since fewer discrete chips are required for the end product.
- 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 an issue faster, which have a direct effect on the time to market and reduces overall development costs.

The ARM core is not a pure RISC architecture because of the constraints of its primary application—the embedded system. In some sense, the strength of the ARM core is that it does not take the RISC concept too far. In today's systems the key is not raw processor speed but total effective system performance and power consumption.

Instruction Set for Embedded Systems

The ARM instruction set differs from the pure RISC definition in several ways that make the ARM instruction set suitable for embedded applications. Following 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.

- *Variable cycle execution for certain instructions*—not every ARM instruction executes in a single cycle. For example, load-store-multiple instructions vary in the number of execution cycles depending upon the number of registers being transferred. The transfer can occur on sequential memory addresses, which increases performance since sequential memory accesses are often faster than random accesses. Code density is also improved since multiple register transfers are common operations at the start and end of functions.
- *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. This expands the capability of many instructions to improve core performance and code density.

- *Thumb 16-bit instruction set*—ARM enhanced 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.
- *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.
- *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 and saturation.

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.

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

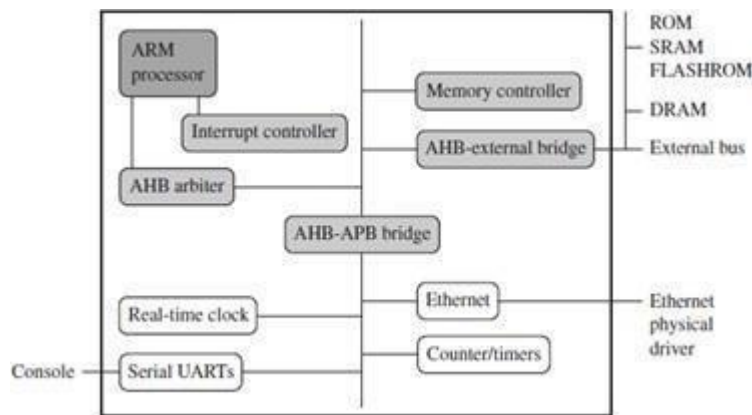


Figure shows a typical embedded device based on an ARM core. Each box represents a feature or function. The lines connecting the boxes are the buses carrying data. The device can be separated into four main hardware components:

- The *ARM processor* controls the embedded device. 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.
- *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.

1. 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 and is built into the motherboard of a PC. In contrast, 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: bus master & bus slaves.

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. ARM is primarily a design company, specifies the bus protocol.

2. 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. This plug-and-play interface for hardware developers improves availability and time to market.

AHB provides higher data throughput than ASB because it is based on a centralized multiplexed bus scheme rather than the ASB bidirectional bus design. This change allows the AHB bus to run at higher clock speeds and to be the first ARM bus to support widths of 64 and 128 bits.

ARM has introduced two variations on the AHB bus: Multi-layer AHB and AHB-Lite. 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.

The example device shown in figure has three buses:

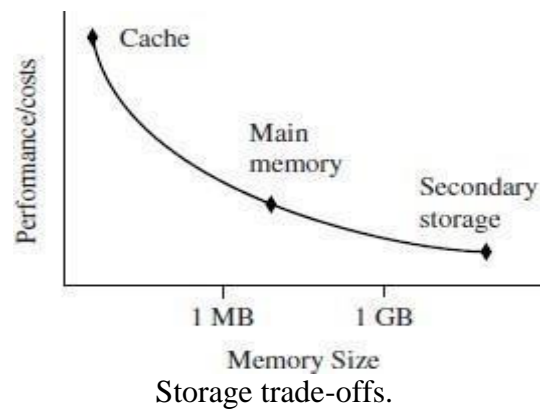
An AHB bus for the high performance peripherals, an APB bus for the slower peripherals, and a third bus for external peripherals, proprietary to this device. This external bus requires a specialized bridge to connect with the AHB bus.

3. Memory

An embedded system has to have some form of memory to store and execute code. We need to compare price, performance, and power consumption when deciding upon specific memory characteristics, such as hierarchy, width, and type.

➤ Hierarchy

All computer systems have memory arranged in some form of hierarchy. 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. The cache is placed between main memory and the core. It is used to speed up data transfer between the processor and main memory. A cache provides an overall increase in performance but with a loss of predictable execution time. 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.



➤ Width

The memory width is the number of bits the memory returns on each access—typically 8, 16, 32, or 64 bits. The memory width has a direct effect on the overall performance and cost ratio. If you have an uncached system using 32-bit ARM instructions and 16-bit-wide memory chips, then the processor will have to make two memory fetches per instruction. Each fetch requires two 16-bit loads. This obviously has the effect of reducing system performance, but the benefit is that 16-bit memory is less expensive. In contrast, if the core executes 16-bit Thumb instructions, it will achieve better performance with a 16-bit memory. The higher performance is a result of the core making only a single fetch to memory to load an instruction. Hence, using Thumb instructions with 16-bit-wide memory devices provides both improved performance and reduced cost.

Table 4.1 summarizes theoretical cycle times on an ARM processor using different memory width devices.

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

Fetching instructions from memory.

➤ Types

There are many different types of memory. In this section we describe some of the more popular memory devices found in ARM-based embedded systems.

- Read-only memory (ROM) is the least flexible of all memory types because it contains an image that is permanently set at production time and cannot be reprogrammed. ROMs are used in high-volume devices that require no updates or corrections. Many devices also use a ROM to hold boot code.
- Flash ROM can be written to as well as read, but it is slow to write so we shouldn't use it for holding dynamic data. Its main use is for holding the device firmware or storing long term data that needs to be preserved after power is off. The erasing and writing of flash ROM are completely software controlled with no additional hardware circuitry required, which reduces the manufacturing costs. Flash ROM has become the most popular of the read-only memory types and is currently being used as an alternative for mass or secondary storage.
- 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 we need to set up a DRAM controller before using the memory.
- 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 because SRAM does not require a

pause between data accesses. Because of its higher cost, it is used mostly for smaller high-speed tasks, such as fast memory and caches.

- 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. Internally the data is fetched from memory cells, pipelined, and finally brought out on the bus in a burst.

4. 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. All ARM peripherals are *memory mapped*—the programming interface is a set of memory-addressed registers. The address of these registers is an offset from a specific peripheral base address. 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. Some memory devices must be set up by software. For example, when using DRAM, the system has to set up the memory timings and refresh rate before it can be accessed.

➤ **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 (SIC) and the vector interrupt controller (VIC).

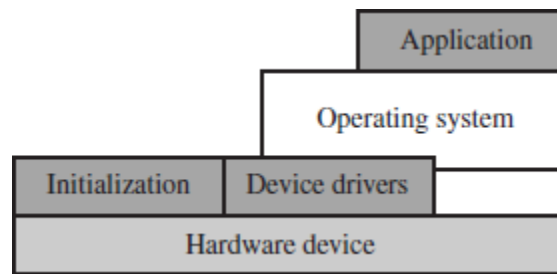
The SIC sends an interrupt signal to the processor core when an external device requests servicing. It can be programmed to ignore or mask an individual device or set of devices. The interrupt handler determines which device requires servicing by reading a device bitmap register in the interrupt controller.

The VIC is more powerful than the SIC because it prioritizes interrupts and simplifies the determination of which device caused the interrupt. After associating a priority and a handler address with each interrupt, the VIC only asserts an interrupt signal to the core if the priority of a new interrupt is higher than the currently executing interrupt handler. Depending on its type, the VIC will either call the standard interrupt exception handler, which can load the address of the handler for the device from the VIC, or cause the core to jump to the handler for the device directly.

Embedded System Software

An embedded system needs software to drive it. Figure below shows four typical software components required to control an embedded device. Each software component in the stack uses a higher level of abstraction to separate the code from the hardware device. 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.

The operating system provides an infrastructure to control applications and manage hardware system resources. Many embedded systems do not require a full operating system but merely a simple task scheduler that is either event or poll driven. The device drivers are the third component shown in Figure. They provide a consistent software interface to the peripherals on the hardware device.

Finally, an application performs one of the tasks required for a device. There may be multiple applications running on the same device, controlled by the operating system. The software components can run from ROM or RAM. ROM code that is fixed on the device (for example, the initialization code) is called *firmware*.

1. Initialization (Boot) Code

Initialization code (or 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. The initialization code handles a number of administrative tasks prior to handing control over to an operating system image. We can group these different tasks into three phases: initial hardware configuration, diagnostics, and booting.

- 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 in example below.
- 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 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. Once booted, the system hands over control by modifying the program counter to point into the start of the image.

Example: 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.

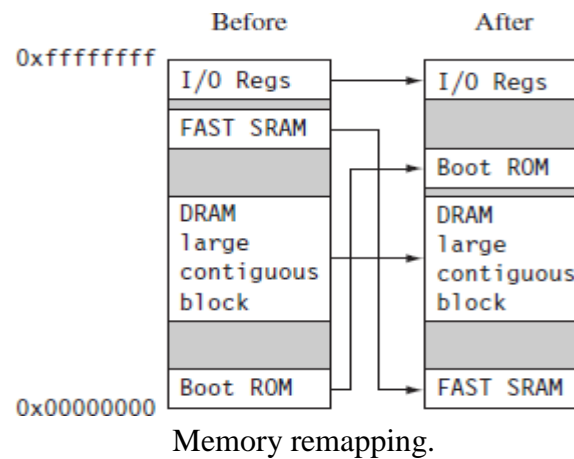


Figure above shows memory before and after reorganization. It is common for ARM-based embedded systems to provide for memory remapping because it allows the system to start the initialization code from ROM at power-up. The initialization code then redefines or remaps the memory map to place RAM at address 0x00000000—an important step because then the exception vector table can be in RAM and thus can be reprogrammed.

2. 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: real-time operating systems (RTOSs) and platform operating systems.

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. Systems running an RTOS generally do not have secondary storage.

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

These two categories of operating system are not mutually exclusive: there are operating systems that use an ARM core with a memory management unit and have real-time characteristics. ARM has developed a set of processor cores that specifically target each category.

3 Applications

The operating system schedules application i.e., the 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. ARM processors are found in numerous market segments, including networking, automotive, mobile and consumer devices, mass storage, and imaging. Within each segment ARM processors can be found in multiple applications.

Example: The ARM processor is found in networking applications like home gateways, DSL modems for high-speed 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—applications that are cost sensitive and high volume.

In contrast, ARM processors are not found in applications that require leading-edge high performance. Because these applications tend to be low volume and high cost, ARM has decided not to focus designs on these types of applications.

ARM Processor Fundamentals

A programmer can think of an ARM core as functional units connected by data buses, as shown in figure below, where, 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. Data enters the processor core through the *Data* bus. The data may be an instruction to execute or a data item.

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

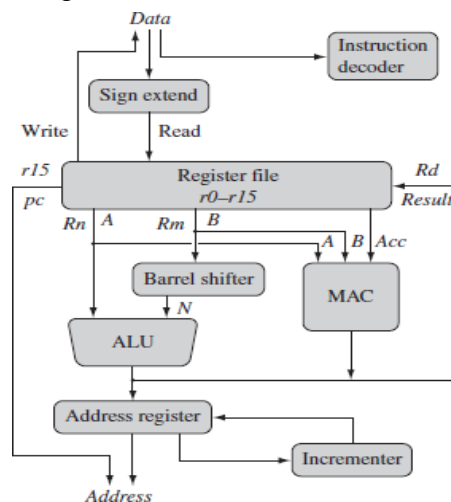
The ARM processor, like all RISC processors, uses *load-store architecture*. This means it has two instruction types for transferring data in and out of the processor: load instructions copies data from memory to registers in the core, and the store instructions copies data from registers to memory.

There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out solely in registers.

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, *Rn* and *Rm*, and a single result or destination register, *Rd*. Source operands are read from the register file using the internal buses *A* and *B*, respectively.

The ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) takes the register values *Rn* and *Rm* from the *A* and *B* buses and computes a result. Data processing instructions write the result in *Rd* 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.



ARM Core Dataflow Model

One important feature of the ARM is that register *Rm* 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.

After passing through the functional units, the result in *Rd* is written back to the register file using the *Result* bus. 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.

1. Registers

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*. Figure below 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 introduce shortly. All the registers shown are 32 bits in size.

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.

In Figure below, the shaded registers identify the assigned special-purpose registers:

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

Registers available in *user* mode.

- 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*) which gets loaded with 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.

Depending upon the context, registers *r13* and *r14* can also be used as general-purpose registers. However, 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.

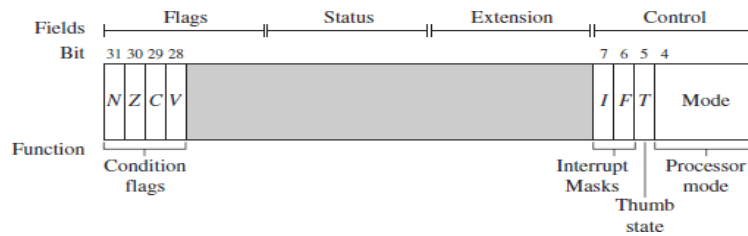
In ARM state the registers *r0* to *r13* are *orthogonal*—any instruction that you can apply to *r0* you can equally well apply to any of the other registers. However, there are instructions that treat *r14* and *r15* in a special way.

In addition to the 16 data registers, there are two program status registers: *cpsr* and *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.

2. Current Program Status Register

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. Figure below shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future expansion.

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 interrupts mask bits. The flags field contains the condition flags. Some ARM processor cores have extra bits allocated. For example, 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*).

➤ 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 non privileged: A privileged mode allows full read-write access to the *cpsr*.

A non privileged 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*) and one non privileged 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.

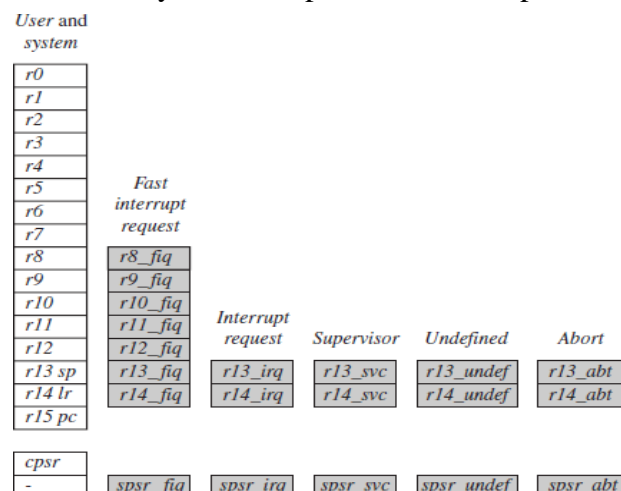
System mode is a special version of *user* mode that allows full read-write access to the *cpsr*.

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

Figure 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.

Example: *abort* mode has banked registers *r13_abt*, *r14_abt* and *spsr_abt*.

Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or *_mode*.

Every processor mode except *user* mode can change mode by writing directly to the mode bits of the *cpsr*.

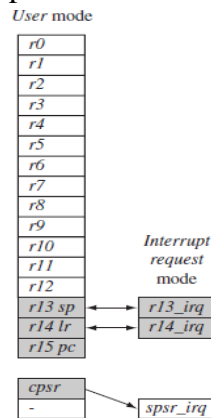
All processor modes except *system* mode have a set of associated banked registers that are a subset of the main 16 registers.

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.

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*. 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.



Changing mode on an exception.

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.

Figure below 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.

Figure 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.

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

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.

The table 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. Table 4.2 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.

Similarly the 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. We cannot intermingle sequential ARM, Thumb, and Jazelle instructions. 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. When the *T* bit is 1, then the processor is in Thumb state. To change states the core executes a specialized branch instruction.

Table below compares the ARM and Thumb instruction set features.

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.

ARM and Thumb instruction set features.

	ARM (<i>cpsr T</i> = 0)	Thumb (<i>cpsr T</i> = 1)
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 instruction set features		
Jazelle (<i>cpsr T</i> = 0, <i>J</i> = 1)		
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*.

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

Condition flags

With processor cores that include the DSP extensions, the *Q* bit 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. Most ARM instructions can be executed conditionally on the value of the condition flags. Table 4.5 lists the condition flags and a short description on what causes them to be set. These flags are located in the most significant bits in the *cpsr*. These bits are used for conditional execution.

Figure below shows a typical value for the *cpsr* with both DSP extensions and Jazelle. In the *cpsr* example shown in figure, 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. Finally, you 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.

Conditional Mnemonics

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

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. One way to view the pipeline is to think of it as an automobile assembly line, with each stage carrying out a particular task to manufacture the vehicle.



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.

Figure 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.

Example: The three instructions ADD, SUB & CMP are placed into the pipeline sequentially.

In the first cycle the core fetches the ADD instruction from memory.

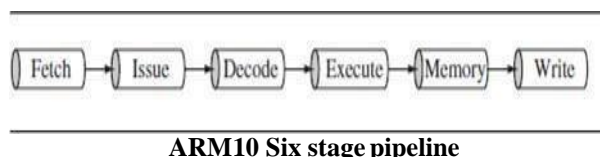
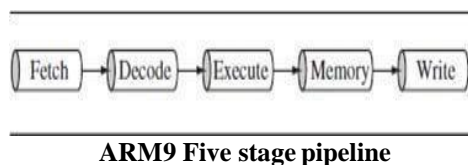
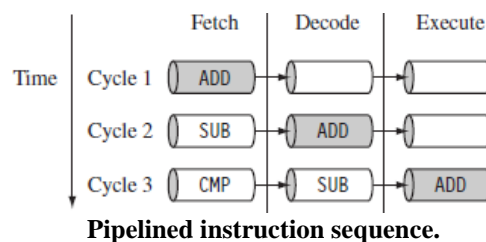
In the second cycle the core fetches the SUB instruction and decodes the ADD instruction.

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.

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.



The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages, as shown in figure. The ARM9 adds a memory and write back 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.

The maximum core frequency attainable using an ARM9 is also higher. The ARM10 increases the pipeline length still further by adding a sixth stage, as shown in figure . Even though the

ARM9 and ARM10 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.

➤ 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. Figure below 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.

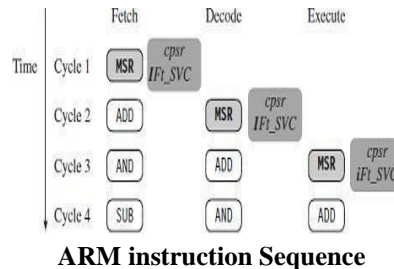
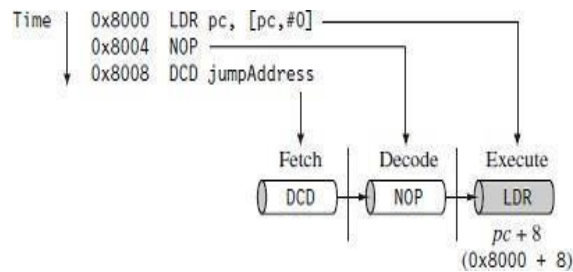


Figure below illustrates the use of the pipeline and the program counter *pc*. In the execute stage, the *pc* always points to the address of the instruction plus 8 bytes. In other words, the *pc* always points to the address of the instruction being executed plus two instructions ahead. This is important when the *pc* is used for calculating a relative offset and is an architectural characteristic across all the pipelines.



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 entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt. 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).

- When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table. Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:
- *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*.

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

Core Extensions

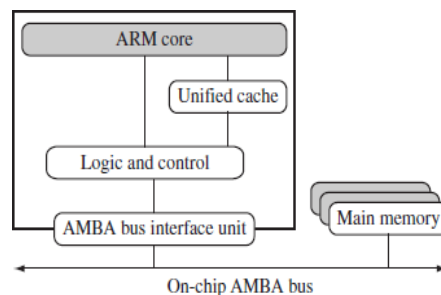
The hardware extensions are standard components placed next to the ARM core. They improve performance, manage resources, and provide extra functionality and are designed to provide flexibility in handling particular applications. Each ARM family has different extensions available.

There are three hardware extensions ARM wraps around the core: cache and tightly coupled memory, memory management, and the 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.

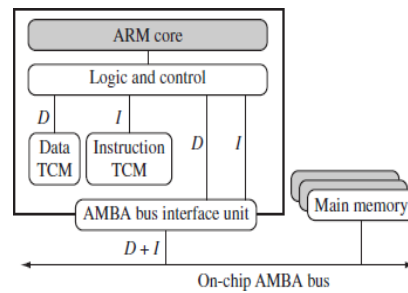
Most ARM-based embedded systems use a single-level cache internal to the processor. ARM has two forms of cache. The first is found attached to the Von Neumann-style cores. It combines both data and instruction into a single unified cache, as shown in Figure.



A simplified Von Neumann architecture with cache.

The second form, 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. But for real-time systems it is paramount that code execution is

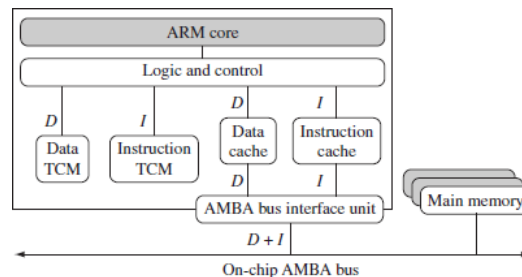
deterministic—the time taken for loading and storing instructions or data must be predictable. This is achieved using a form of memory called *tightly coupled memory* (TCM).



A simplified Harvard architecture with TCMs.

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 Figure below.

By combining both technologies, ARM processors can have both improved performance and predictable real-time response. The figure shows an example core with a combination of caches and TCMs.



A simplified Harvard architecture with Caches and TCMs

➤ 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—no extensions providing no protection, a memory protection unit (MPU) providing limited protection, and a memory management unit (MMU) providing full protection:

- *Non protected 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.

➤ 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. More than one coprocessor can be added to the ARM core via the coprocessor interface.

The coprocessor can be accessed through a group of dedicated ARM instructions that provide a load-store type interface. Consider, 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.