ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560028

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(ACADEMIC YEAR 2024-25)

LESSON NOTES

SUBJECT: MICROCONTROLLERS

SUB CODE: BCS402

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 capableat 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 realworld 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.

MICROCONTROLLERS		Semester	4
Course Code	**BCS402**	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab Slots	Total Marks	100
Credits	04	Exam Hours	3
Examination nature (SEE)	Theory		

Course Objectives:

CLO 1: Understand the fundamentals of ARM-based systems and basic architecture of CISC and RISC.
CLO 2: Familiarize with ARM programming modules along with registers, CPSR and Flags.
CLO 3: Develop ALP using various instructions to program the ARM controller.
CLO 4: Understand the Exceptions and Interrupt handling mechanism in Microcontrollers.
CLO 5: Discuss the ARM Firmware packages and Cache memory polices.

Teaching-Learning Process

These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes.

1. Lecturer method (L) needs not to be only a traditional lecture method, but alternative effective teaching methods could be adopted to attain the outcomes.
2. Use of Video/Animation to explain functioning of various concepts.
3. Encourage collaborative (Group Learning) Learning in the class.
4. Ask at least three HOT (Higher order Thinking) questions in the class, which promotes critical thinking.
5. Adopt Problem Based Learning (PBL), which fosters students' Analytical skills, develop design thinking skills such as the ability to design, evaluate, generalize, and analyze information rather than simply recall it.
6. Introduce Topics in manifold representations.
7. Show the different ways to solve the same problem with different circuits/logic and encourage the students to come up with their own creative ways to solve them.
8. Discuss how every concept can be applied to the real world - and when that's possible, it helps improve the students understanding.
9. Use any of these methods: Chalk and board, Active Learning, Case Studies.

MODULE-1 **No. of Hours: 8**

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 1 - 1.1 to 1.4, Chapter 2 - 2.1 to 2.5
RBT: L1, L2, L3

MODULE-2 **No. of Hours: 8**

Introduction to the ARM Instruction Set: Data Processing Instructions, Branch Instructions, Software Interrupt Instructions, Program Status Register Instructions, Coprocessor Instructions, Loading Constants.

Textbook 1: Chapter 3 - 3.1 to 3.6
RBT: L1, L2, L3

MODULE-3 **No. of Hours:8**

C Compilers and Optimization: Basic C Data Types, C Looping Structures, Register Allocation, Function Calls, Pointer Aliasing, Portability Issues.

Textbook 1: Chapter 5.1 to 5.7 and 5.13
RBT: L1, L2, L3

MODULE-4	No. of Hours:8
Exception and Interrupt Handling: Exception handling, ARM processor exceptions and modes, vector table, exception priorities, link register offsets, interrupts, assigning interrupts, interrupt latency, IRQ and FIQ exceptions, basic interrupt stack design and implementation. **Firmware:** Firmware and bootloader, ARM firmware suite, Red Hat redboot, Example: sandstone, sandstone directory layout, sandstone code structure. **Textbook 1: Chapter 9.1 and 9.2, Chapter 10** **RBT: L1, L2, L3**	
MODULE-5	**No. of Hours:08**
CACHES: The Memory Hierarchy and Cache Memory, Caches and Memory Management Units: CACHE Architecture: Basic Architecture of a Cache Memory, Basic Operation of a Cache Controller, The Relationship between Cache and Main Memory, Set Associativity, Write Buffers, Measuring Cache Efficiency, CACHE POLICY: Write Policy—Writeback or Writethrough, Cache Line Replacement Policies, Allocation Policy on a Cache Miss. Coprocessor 15 and caches. **Textbook 1: Chapter 12.1 to 12.4** **RBT: L1, L2, L3**	

PRACTICAL COMPONENT OF IPCC *(May cover all / major modules)*

Sl.No.	Experiments
Module – 1	
1.	Using Keil software, observe the various Registers, Dump, CPSR, with a simple Assembly Language Programs (ALP).
Module – 2	
2.	Develop and simulate ARM ALP for Data Transfer, Arithmetic and Logical operations (Demonstrate with the help of a suitable program).
3.	Develop an ALP to multiply two 16-bit binary numbers.
4.	Develop an ALP to find the sum of first 10 integer numbers.
5.	Develop an ALP to find the largest/smallest number in an array of 32 numbers.
6.	Develop an ALP to count the number of ones and zeros in two consecutive memory locations.
Module – 3	
7.	Simulate a program in C for ARM microcontroller using KEIL to sort the numbers in ascending/descending order using bubble sort.
8.	Simulate a program in C for ARM microcontroller to find factorial of a number.
9.	Simulate a program in C for ARM microcontroller to demonstrate case conversion of characters from upper to lowercase and lower to uppercase.
Module – 4 and 5	
10.	Demonstrate enabling and disabling of Interrupts in ARM.
11.	Demonstrate the handling of divide by zero, Invalid Operation and Overflow exceptions in ARM.

Course outcomes (Course Skill Set):

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

- Explain the ARM Architectural features and Instructions.
- Develop programs using ARM instruction set for an ARM Microcontroller.
- Explain C-Compiler Optimizations and portability issues in ARM Microcontroller.
- Apply the concepts of Exceptions and Interrupt handling mechanisms in developing applications.
- Demonstrate the role of Cache management and Firmware in Microcontrollers.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the

academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the theory component of the IPCC (maximum marks 50)

- IPCC means practical portion integrated with the theory of the course.
- CIE marks for the theory component are **25 marks** and that for the practical component is **25 marks**.
- 25 marks for the theory component are split into **15 marks** for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and **10 marks** for other assessment methods mentioned in 22OB4.2. The first test at the end of 40-50% coverage of the syllabus and the second test after covering 85-90% of the syllabus.
- Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for **25 marks**).
- The student has to secure 40% of 25 marks to qualify in the CIE of the theory component of IPCC.

CIE for the practical component of the IPCC

1. **15 marks** for the conduction of the experiment and preparation of laboratory record, and **10 marks** for the test to be conducted after the completion of all the laboratory sessions.
2. On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
3. The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to **15 marks**.
4. The laboratory test **(duration 02/03 hours)** after completion of all the experiments shall be conducted for 50 marks and scaled down to **10 marks.**
5. Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for **25 marks**.
6. The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

SEE for IPCC

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course **(duration 03 hours)**

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scored by the student shall be proportionally scaled down to 50 Marks.

The theory portion of the IPCC shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper may include questions from the practical component.

Suggested Learning Resources:

Text Books:

1. Andrew N Sloss, Dominic Symes and Chris Wright, ARM system developers guide, Elsevier, Morgan Kaufman publishers, 2008.

Reference Books:

1. Raghunandan.G.H, Microcontroller (ARM) and Embedded System, Cengage learning Publication, 2019.
2. Insider's Guide to the ARM7 based microcontrollers, Hitex Ltd.,1st edition, 2005

Activity Based Learning (Suggested Activities in Class)/ Practical Based Learning

Assign the group task to demonstrate the Installation and working of Keil Software.

MODULE – 1

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.

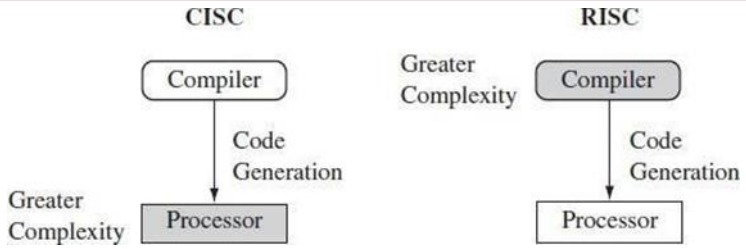
2. Pipelines—The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines. Ideally the pipeline advances by one step on each cycle for maximum throughput. Instructions can be decoded in one pipeline stage.

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
Emphasis on hardware	Emphasis on software
Includes multi-clockcomplex instructions	Single-clock, reduced instructiononly
Data transfer between memory & register incorporated in available instructions	Separate instructions available:"LOAD" and "STORE"are independent instructions
Small code sizes, highcycles per second	Low cycles per second & largecode sizes
Transistors used for storing complex instructions	Spends more transistors on registers



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.

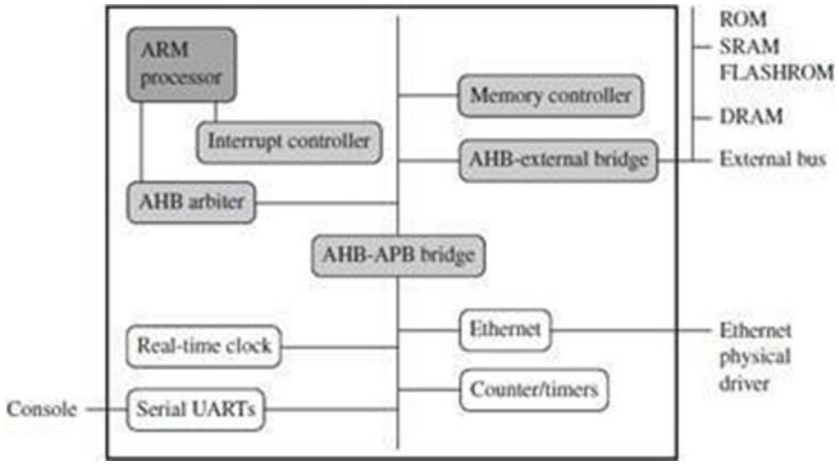


Figure shows a typical embedded device based on an ARM core. Each box represents a feature or function. The lines connecting the boxes are the buses carrying data. The device can be separated into four main hardware components:

• 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:

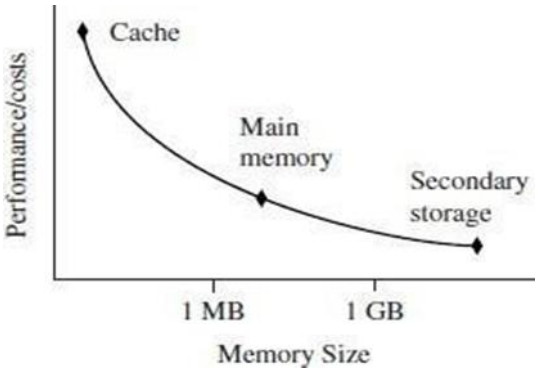
An AHB bus for the high-performance peripherals, an APB bus for the slower peripherals, and a third bus for external peripherals, proprietary to this device. This external bus requires a specialized bridge to connect with the AHB bus.

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.



Storage trade-offs.

- **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-bitARM 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

- **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 circuity 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.

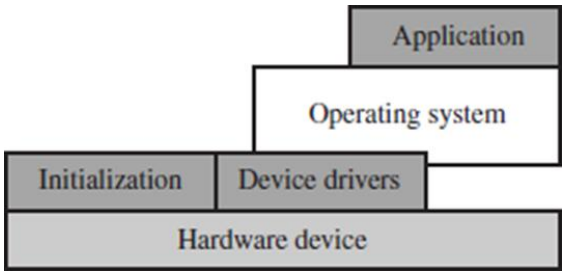
There are two types of interrupt controller available for the ARM processor: The standard interrupt controller (SIC) and the vector interrupt controller (VIC).

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.

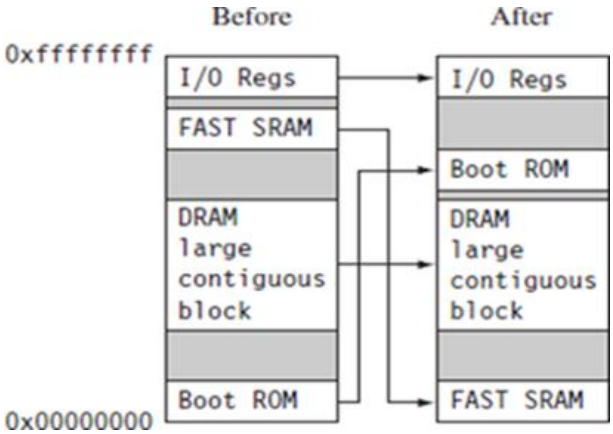
Finally, an application performs one of the tasks required for a device. There may be multiple applications running on the same device, controlled by the operating system. The software components can run from ROM or RAM. ROM code that is fixed on the device (for example, the initialization code) is called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



Memory remapping.

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 ie., 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.

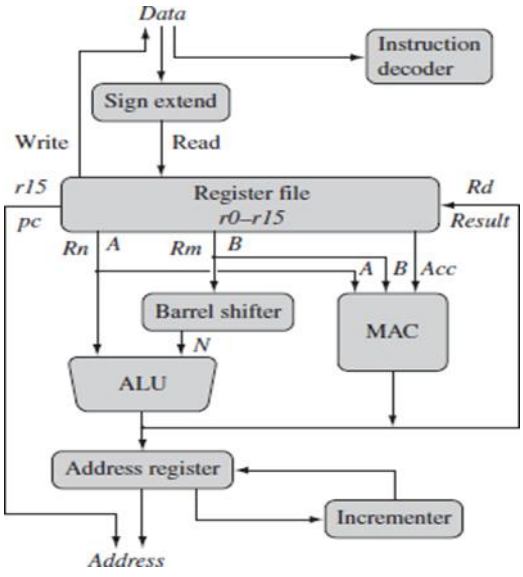
The ARM processor, like all RISC processors, uses load-store architecture. This means it has two instruction types for transferring data in and out of the processor: load instructions copies data from memory to registers in the core, and the store instructions copies data from registers to memory.

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 pre-processed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses.

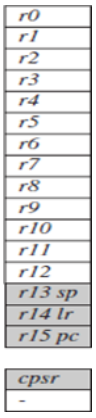
After passing through the functional units, the result in Rd is written back to the register file using the Result bus. For load and store instructions the incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location. 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:



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.

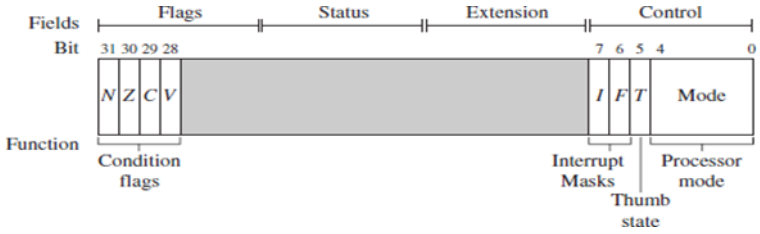
In ARM state the registers r0 to r13 are orthogonal—any instruction that you can apply to r0 you can equally well apply to any of the other registers. However, there are instructions that treat r14 and r15 in a special way.

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.

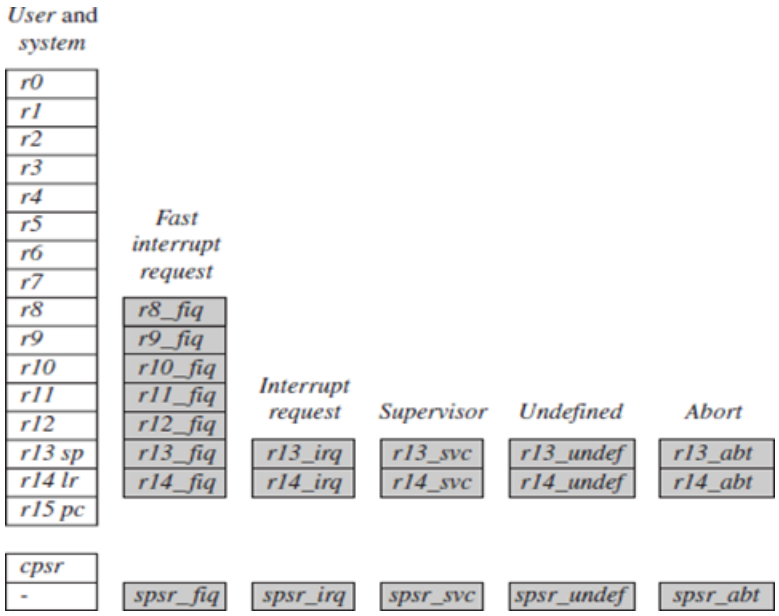
Supervisor mode is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in.

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.

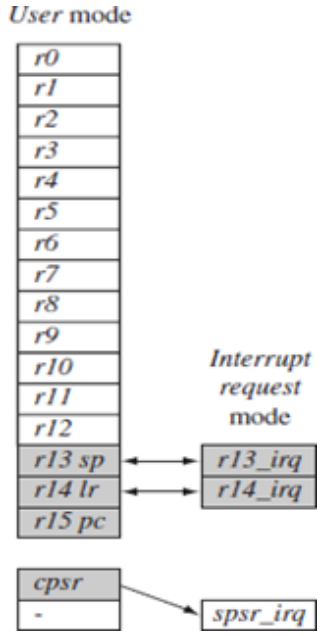
All processor modes except system mode have a set of associated banked registers that are a subset of the main 16 registers.

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]
Abort	abt	yes	10111
Fast interrupt request	fiq	yes	10001
Interrupt request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	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 (*cpsr T* = 0)	Thumb (*cpsr T* = 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 +*pc*	8 general-purpose registers +7 high registers +*pc*

Jazelle instruction set features

	Jazelle (*cpsr T* = 0, *J* = 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 above 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.

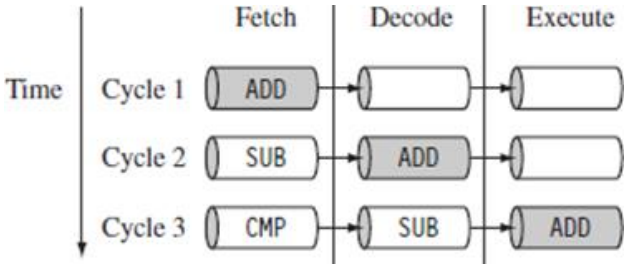
Example: The three instructions ADD,SUB & CMP are placed into the pipeline sequentially. In the first cycle the core fetches the ADD instruction from memory.

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.

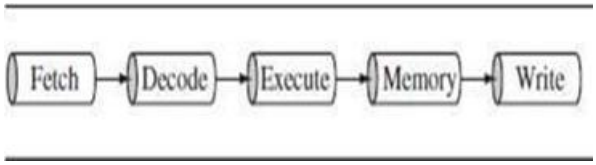
The ADD instruction is executed, the SUB instruction is decoded, and the CMP instruction is fetched.

This procedure is called filling the pipeline. The pipeline allows the core to execute an instruction every cycle.

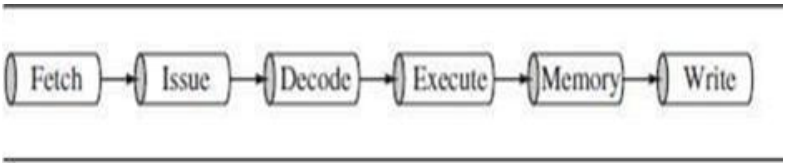
As the pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency. This in turn increases the performance. The system latency also increases because it takes more cycles to fill the pipeline before the core can execute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



Pipelined instruction sequence



ARM9 Five stage pipeline



ARM10 Six stage pipeline

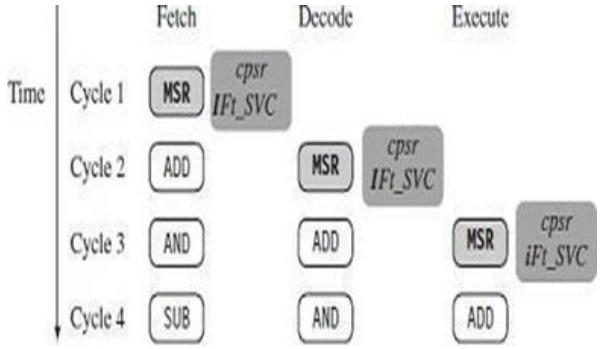
The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages, as shown in figure. The ARM9 adds a memory and write backstage, which allows the ARM9 to process on average 1.1 Dhrystone MIPS per MHz—an increase in instruction throughput by around 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**

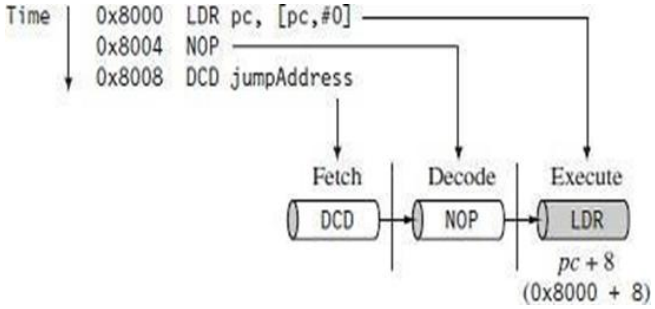
The ARM pipeline has not processed an instruction until it passes completely through the execute stage. For example, an ARM7 pipeline (with three stages) has executed an instruction only when the fourth instruction is fetched. Figure below shows an instruction sequence on an ARM7 pipeline. The MSR instruction 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.



ARM instruction Sequence

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.



Example: pc = address + 8

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

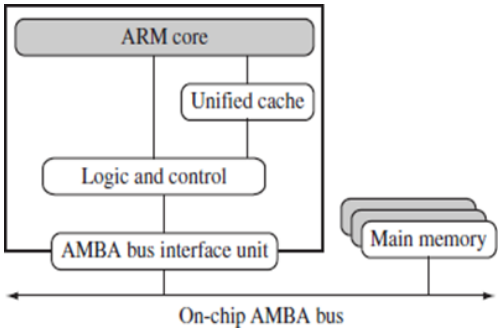
The hardware extensions are standard components placed next to the ARM core. They improve performance, manage resources, and provide extra functionality and are designed to provide flexibility in handling particular applications. Each ARM family has different extensions available.

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**

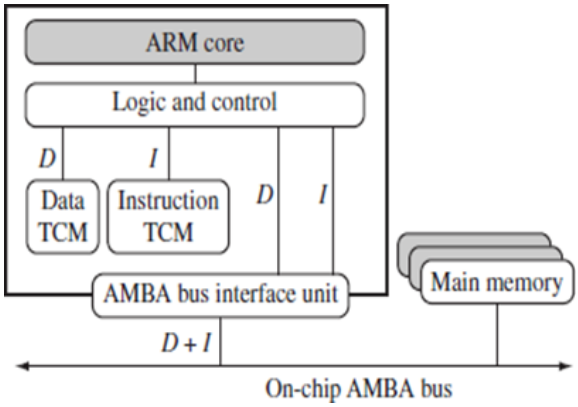
The cache is a block of fast memory placed between main memory and the core. It allows for more efficient fetches from some memory types. With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory.

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.

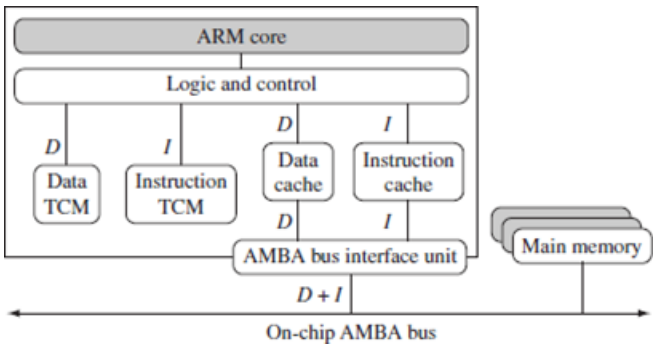
The second form, attached to the Harvard-style cores, has separate caches for data and instruction. A cache provides an overall increase in performance but at the expense of predictable execution. But for real-time systems it is paramount that code execution is deterministic— the time taken for loading 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.

MODULE 2

Introduction To the Arm Instruction Set

Data Processing Instructions

The data processing instructions manipulate data within registers. They are move instructions, arithmetic instructions, logical instructions, comparison instructions, and multiply instructions. Most data processing instructions can process one of their operands using the barrel shifter.

If you use the S suffix on a data processing instruction, then it updates the flags in the cpsr. Move and logical operations update the carry flag C, negative flag N, and zero flag Z.

The carry flag is set from the result of the barrel shift as the last bit shifted out. The N flag is set to bit 31 of the result. The Z flag is set if the result is zero.

1. Move Instructions

Move is the simplest ARM instruction. It copies N into a destination register Rd, where N is a register or immediate value. This instruction is useful for setting initial values and transferring data between registers.

Syntax: <instruction>{<cond>}{S} Rd, N

MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

MOV - Move a 32-bit value into a register Rd = N

MVN - move the NOT of the 32-bit value into a register Rd =!N

The values allowed for the second operand N for all data processing instructions. Usually it is a register Rm or a constant preceded by #.

Ex:This example shows a simple move instruction. The MOV instruction takes the contents of register r5 and copies them into register r7, in this case, taking the value 5, and overwriting the value 8 in register r7.

PRE: r5 = 5

r7 = 8

MOV r7, r5; r7 = r5

POST: r5 = 5

r7 = 5

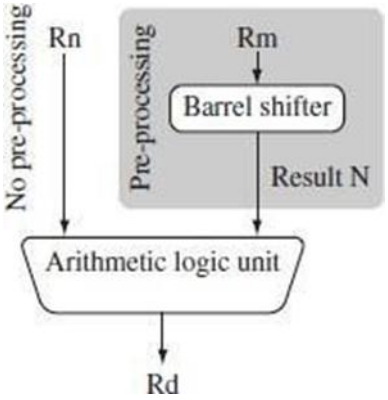
2. Barrel Shifter

MOV instruction can contain N as a simple register or immediate value or it can also be a register Rm that has been preprocessed by the barrel shifter prior to being used by a data processing instruction. Data processing instructions are processed within the arithmetic logic unit (ALU).

A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU. This shift increases the power and flexibility of many data processing operations.

There are data processing instructions that do not use the barrel shift, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions.

Pre-processing or shift occurs within the cycle time of the instruction. This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.



Barrel shifter and ALU.

Register Rn enters the ALU without any preprocessing of registers. Figure shows the data flow between the ALU and the barrel shifter.

Ex: We apply a logical shift left (LSL) to register Rm before moving it to the destination register. This is the same as applying the standard C language shift operator << to the register. The MOV instruction copies the shift operator result N into register Rd. N represents the result of the LSL operation described in Table below.

```
PRE   r5 = 5
      r7 = 8
      MOV r7, r5, LSL #2 ; r7 = r5*4 or (r5 << 2)
POST  r5 = 5
      r7 = 20
```

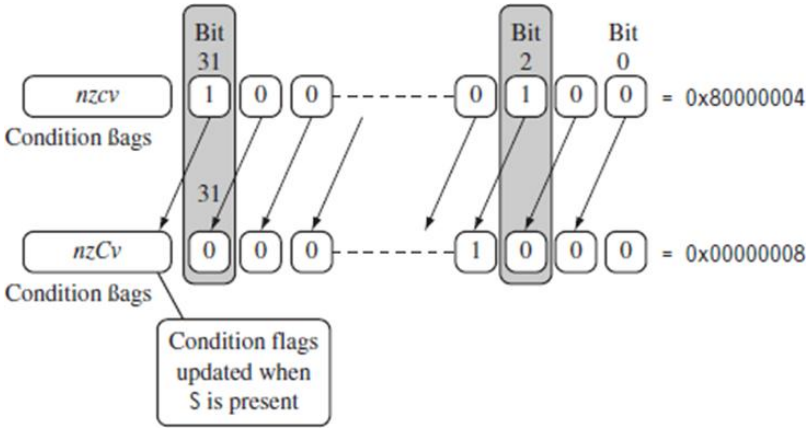
The example multiplies register r5 by four and then places the result into register r7. The five different shift operations that you can use within the barrel shifter are summarized in Table below.

Figure illustrates a logical shift left by one. For example, the contents of bit 0 are shifted to bit 1. Bit 0 is cleared. The C flag is updated with the last bit shifted out of the register. This is bit (32−y) of the original value, where y is the shift amount. When y is greater than one, then a shift by y positions is the same as a shift by one position executed y times.

Barrel shifter operations

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	*x*LSL *y*	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	*x*LSR *y*	(unsigned)$x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x*ASR *y*	(signed)$x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x*ROR *y*	((unsigned)$x \gg y$) | ($x \ll (32 - y)$)	#1–31 or *Rs*
RRX	rotate right extended	*x*RRX	(*c* flag $\ll 31$) | ((unsigned)$x \gg 1$)	none

Note: *x* represents the register being shifted and *y* represents the shift amount.



Logical shift left by one

Barrel shift operation syntax for data processing instruction

N shift operations	Syntax
Immediate	#immediate
Register	Rm
Logical shift left by immediate	Rm, LSL #shift_imm
Logical shift left by register	Rm, LSL Rs
Logical shift right by immediate	Rm, LSR #shift_imm
Logical shift right with register	Rm, LSR Rs
Arithmetic shift right by immediate	Rm, ASR #shift_imm
Arithmetic shift right by register	Rm, ASR Rs
Rotate right by immediate	Rm, ROR #shift_imm
Rotate right by register	Rm, ROR Rs
Rotate right with extend	Rm, RRX

Table lists the syntax for the different barrel shift operations available on data processing instructions. The second operand N can be an immediate constant preceded by #, a register value Rm, or the value of Rm processed by a shift.

Ex: This example of a MOVS instruction shifts register r1 left by one bit. This multiplies register r1 by a value 2. As you can see, the C flag is updated in the cpsr because the S suffix is present in the instruction mnemonic.

PRE cpsr = nzcvqiFt_USER
r0 = 0x00000000
r1 = 0x80000004
MOVS r0, r1, LSL #1

POST cpsr = nzCvqiFt_USER
r0 = 0x00000008
r1 = 0x80000004

3. Arithmetic Instructions

The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.

Syntax: <instruction>{<cond>}{S} Rd, Rn, N

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

N is the result of the shifter operation. The syntax of shifter operation is shown in the table.

Ex: This simple subtract instruction subtracts a value stored in register r2 from a value stored in register r1. The result is stored in register r0.

PRE r0 = 0x00000000
r1 = 0x00000002
r2 = 0x00000001
SUB r0, r1, r2
POST r0 = 0x00000001

Ex: This reverse subtract instruction (RSB) subtracts r1 from the constant value #0, writing the result to r0. You can use this instruction to negate numbers.

PRE r0 = 0x00000000
r1 = 0x00000077
RSB r0, r1, #0 ; Rd = 0x0 - r1
POST r0 = -r1 = 0xffffff89

Ex: The SUBS instruction is useful for decrementing loop counters. In this example we subtract the immediate value one from the value one stored in register r1. The result value zero is written to register r1. The cpsr is updated with the ZC flags being set.

PRE cpsr = nzcvqiFt_USER
r1 = 0x00000001
SUBS r1, r1, #1
POST cpsr = nZCvqiFt_USER
r1 = 0x00000000

4. Using the Barrel Shifter with Arithmetic Instructions

The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set. Example below illustrates the use of the

inline barrel shifter with an arithmetic instruction. The instruction multiplies the value stored in register r1 by three.
Ex: Register r1 is first shifted one location to the left to give the value of twice r1. The ADD instruction then adds the result of the barrel shift operation to register r1. The final result transferred into register r0 is equal to three times the value stored in register r1.

```
PRE   r0 = 0x00000000
      r1 = 0x00000005
      ADD r0, r1, r1, LSL #1
POST  r0 = 0x0000000f
      r1 = 0x00000005
```

5. Logical Instructions

Logical instructions perform bitwise logical operations on the two source registers.
Syntax: <instruction>{<cond>}{S} Rd, Rn, N

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

Ex: This example shows a logical OR operation between registers r1 and r2. r0 holds the result.

```
PRE   r0 = 0x00000000
      r1 = 0x02040608
      r2 = 0x10305070
      ORR r0, r1, r2
POST  r0 = 0x12345678 ■
```

Ex: This example shows a more complicated logical instruction called BIC, which carries out a logical bit clear.

```
PRE   r1 = 0b1111
      r2 = 0b0101
      BIC r0, r1, r2
POST  r0 = 0b1010
```

This is equivalent to Rd = Rn AND NOT(N)
In this example, register r2 contains a binary pattern where every binary 1 in r2 clears a corresponding bit location in register r1. This instruction is particularly useful when clearing status bits and is frequently used to change interrupt masks in the cpsr. The logical instructions update the cpsr flags only if the S suffix is present. These instructions can use barrel-shifted second operands in the same way as the arithmetic instructions.

6. Comparison Instructions

The comparison instructions are used to compare or test a register with a 32-bit value. They update the cpsr flag bits according to the result, but do not affect other registers. After the bits have been set, the information can then be used to change program flow by using conditional execution. You do not need to apply the S suffix for comparison instructions to update the flags.
Syntax: <instruction>{<cond>} Rn, N

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

N is the result of the shifter operation. The syntax of shifter operation is shown in Table 3.3.

Ex: This example shows a CMP comparison instruction. You can see that both registers, r0 and r9, are equal before executing the instruction. The value of the z flag prior to execution is 0 and is represented by a lowercase z. After execution the z flag changes to 1 or an uppercase Z. This change indicates equality.

```
PRE   cpsr = nzcvqiFt_USER
      r0 = 4
      r9 = 4
      CMP r0, r9
POST  cpsr = nZcvqiFt_USER
```

The CMP is effectively a subtract instruction with the result discarded; similarly the TST instruction is a logical AND operation, and TEQ is a logical exclusive OR operation. For each, the results are discarded but the condition bits are updated in the cpsr. It is important to understand that comparison instructions only modify the condition flags of the cpsr and do not affect the registers being compared.

7. Multiply Instructions

The multiply instructions multiply the contents of a pair of registers and, depending upon the instruction, accumulate the results in with another register. The long multiplies accumulate onto a pair of registers representing a 64-bit value. The final result is placed in a destination register or a pair of registers.

Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn MUL{<cond>}{S} Rd, Rm, Rs

```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

MLA	multiply and accumulate	$Rd = (Rm * Rs) + Rn$
MUL	multiply	$Rd = Rm * Rs$

```
Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs
```

SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm * Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm * Rs$

The number of cycles taken to execute a multiply instruction depends on the processor implementation. For some implementations the cycle timing also depends on the value in Rs.

Ex: This example shows a simple multiply instruction that multiplies registers r1 and r2 together and places the result into register r0. In this example, register r1 is equal to the value 2, and r2 is equal to 2. The result, 4, is then placed into register r0.

```
PRE -  r0 = 0x00000000
       r1 = 0x00000002
       r2 = 0x00000002
       MUL r0, r1, r2 ; r0 = r1*r2
POST- r0 = 0x00000004
       r1 = 0x00000002
       r2 = 0x00000002
```

The long multiply instructions (SMLAL, SMULL, UMLAL, and UMULL) produce a 64-bit result. The result is too large to fit a single 32-bit register so the result is placed in two registers labeled RdLo and RdHi. RdLo holds the lower 32 bits of the 64-bit result, and RdHi holds the higher 32 bits of the 64-bit result.

Ex: The instruction multiplies registers r2 and r3 and places the result into register r0 and r1. Register r0 contains the lower 32 bits, and register r1 contains the higher 32 bits of the 64-bit result.

```
PRE-   r0 = 0x00000000
       r1 = 0x00000000
       r2 = 0xf0000002
       r3 = 0x00000002
       UMULL r0, r1, r2, r3 ; [r1,r0] = r2*r3
POST- r0 = 0xe0000004 ; = RdLo
       r1 = 0x00000001 ; = RdHi
```

Branch Instructions

A branch instruction changes the flow of execution or is used to call a routine. This type of instruction allows programs to have subroutines, if-then-else structures, and loops. The change of execution flow forces the program counter pc to point to a new address.

Syntax: B{<cond>} label
BL{<cond>} label
BX{<cond>} Rm
BLX{<cond>} label | Rm

B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

The address label is stored in the instruction as a signed pc-relative offset and must be within approximately 32 MB of the branch instruction. T refers to the Thumb bit in the cpsr. When instructions set T, the ARM switches to Thumb state.
Ex: This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. The backward branch creates an infinite loop.

```
        B forward
        ADD r1, r2, #4
        ADD r0, r6, #2
        ADD r3, r7, #4
forward
        SUB    r1, r2, #4
backward
        ADD r1, r2, #4
        SUB r1, r2, #4
        ADD r4, r6, r7
        B backward
```

Branches are used to change execution flow. Most assemblers hide the details of a branch instruction encoding by using labels. In this example, forward and backward are the labels. The branch labels are placed at the beginning of the line and are used to mark an address that can be used later by the assembler to calculate the branch offset.
Ex: The branch with link, or BL, instruction is similar to the B instruction but overwrites the link register lr with a return address. It performs a subroutine call. This example shows a simple fragment of code that branch to a subroutine using the BL instruction. To return from a subroutine, you copy the link register to the pc.

```
        BL subroutine ; branch to subroutine
        CMP r1, #5     ; compare r1 with 5
        MOVEQ r1, #0          ; if (r1==5) then r1 = 0
:
subroutine
        <subroutine code>
        MOV pc, lr     ; return by moving pc = lr
```

The branch exchange (BX) and branch exchange with link (BLX) are the third type of branch instruction. The BX instruction uses an absolute address stored in register Rm. The T bit in the cpsr is updated by the least significant bit of the branch register. Similarly the BLX instruction updates the T bit of the cpsr with the least significant bit and additionally sets the link register with the return address.

Load-Store Instructions

Load-store instructions transfer data between memory and processor registers. There are three types of load-store instructions: single-register transfer, multiple-register transfer, and swap.

Single-Register Transfer: These instructions are used for moving a single data item in and out of a register. The datatypes supported are signed and unsigned words (32-bit), halfwords (16-bit), and bytes. Here are the various load-store single-register transfer instructions.

Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1

LDR{<cond>}SB|H|SH Rd, addressing2

STR{<cond>}H Rd, addressing2

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*
LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

Ex: LDR and STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored. For example, LDR can only load 32-bit words on a memory address that is a multiple of four bytes—0, 4, 8, and so on. This example shows a load from a memory address contained in register r1, followed by a store back to the same address in memory.

```
; load register r0 with the contents of the memory address pointed to by register r1.
LDR r0, [r1]    ; = LDR r0, [r1, #0]
; store the contents of register r0 to ; the memory address pointed to by register r1.
STR r0, [r1] ; = STR r0, [r1, #0]
```

The first instruction loads a word from the address stored in register r1 and places it into register r0. The second instruction goes the other way by storing the contents of register r0 to the address contained in register r1. The offset from register r1 is zero. Register r1 is called the base address register.

Single-Register Load-Store Addressing Modes

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods: preindex with writeback, preindex, and postindex.

Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

Note: ! indicates that the instruction writes the calculated address back to the base address register.

Ex: Preindex with writeback calculates an address from a base register plus address offset and then updates that address base register with the new address. In contrast, the preindex offset is the same as the preindex with writeback but does not update the address base register. Postindex only updates the address base register after the address is used. The preindex mode is useful for accessing an element in a data structure. The postindex and preindex with writeback modes are useful for traversing an array.

PRE: r0 = 0x00000000
r1 = 0x00090000
mem32[0x00009000] = 0x00000001
mem32[0x00009004] = 0x00000002
LDR r0, [r1, #4]!

Preindexing with writeback:
POST(1): r0 = 0x00000002
r1 = 0x00009004
LDR r0, [r1, #4]
Preindexing:
POST(2) : r0 = 0x00000002
r1 = 0x00009000
LDR r0, [r1], #4
Postindexing:
POST(3): r0 = 0x00000001
r1 = 0x00009004

Single-register load-store addressing, word or unsigned byte

Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	[Rn, #+/-offset_12]
Preindex with register offset	[Rn, +/-Rm]
Preindex with scaled register offset	[Rn, +/-Rm, shift #shift_imm]
Preindex writeback with immediate offset	[Rn, #+/-offset_12]!
Preindex writeback with register offset	[Rn, +/-Rm]!
Preindex writeback with scaled register offset	[Rn, +/-Rm, shift #shift_imm]!
Immediate postindexed	[Rn], #+/-offset_12
Register postindex	[Rn], +/-Rm
Scaled register postindex	[Rn], +/-Rm, shift #shift_imm

The addressing modes available with a particular load or store instruction depend on the instruction class. Table shows the addressing modes available for load and store of a 32-bit word or an unsigned byte.

A signed offset or register is denoted by "+/−", identifying that it is either a positive or negative offset from the base address register Rn. The base address register is a pointer to a byte in memory, and the offset specifies a number of bytes.

Immediate means the address is calculated using the base address register and a 12-bit offset encoded in the instruction. Register means the address is calculated using the base address register and a specific register's contents. Scaled means the address is calculated using the base address register and a barrel shift operation.

Table provides an example of the different variations of the LDR instruction. Example of LDR instructions using different addressing modes:

	Instruction	r0 =	r1 +=
Preindex with writeback	LDR r0,[r1,#0x4]!	mem32[r1+0x4]	0x4
	LDR r0,[r1,r2]!	mem32[r1+r2]	r2
	LDR r0,[r1,r2,LSR#0x4]!	mem32[r1+(r2 LSR 0x4)]	(r2 LSR 0x4)
Preindex	LDR r0,[r1,#0x4]	mem32[r1+0x4]	*not updated*
	LDR r0,[r1,r2]	mem32[r1+r2]	*not updated*
	LDR r0,[r1,-r2,LSR #0x4]	mem32[r1-(r2 LSR 0x4)]	*not updated*
Postindex	LDR r0,[r1],#0x4	mem32[r1]	0x4
	LDR r0,[r1],r2	mem32[r1]	r2
	LDR r0,[r1],r2,LSR #0x4	mem32[r1]	(r2 LSR 0x4)

Single register load store addressing, half word, signed half word, signed byte and double word

Addressing[2] mode and index method	Addressing[2] syntax
Preindex immediate offset	[Rn, #+/-offset_8]
Preindex register offset	[Rn, +/-Rm]
Preindex writeback immediate offset	[Rn, #+/-offset_8]!
Preindex writeback register offset	[Rn, +/-Rm]!
Immediate postindexed	[Rn], #+/-offset_8
Register postindexed	[Rn], +/-Rm

These operations cannot use the barrel shifter. There are no STRSB or STRSH instructions since STRH stores both a signed and unsigned halfword; similarly STRB stores signed and unsigned bytes.

Table shows the variations for STRH instructions.

	Instruction	Result	r1 +=
Preindex with writeback	STRH r0,[r1,#0x4]!	mem16[r1+0x4]=r0	0x4
	STRH r0,[r1,r2]!	mem16[r1+r2]=r0	r2
Preindex	STRH r0,[r1,#0x4]	mem16[r1+0x4]=r0	*not updated*
	STRH r0,[r1,r2]	mem16[r1+r2]=r0	*not updated*
Postindex	STRH r0,[r1],#0x4	mem16[r1]=r0	0x4
	STRH r0,[r1],r2	mem16[r1]=r0	r2

Multiple-Register Transfer

Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction. The transfer occurs from a base address register Rn pointing into memory. Multiple-register transfer instructions are more efficient from single-register transfers for moving blocks of data around memory and saving and restoring context and stacks.

Load-store multiple instructions can increase interrupt latency. ARM implementations do not usually interrupt instructions while they are executing. For example, on an ARM7 a load multiple instruction takes 2 + Nt cycles, where N is the number of registers to load and t is the number of cycles required for each sequential access to memory. If an interrupt has been raised, then it has no effect until the load-store multiple instruction is complete. Compilers, such as armcc, provide a switch to control the maximum number of registers being transferred on a load- store, which limits the maximum interrupt latency.

Syntax: <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}

LDM	load multiple registers	{Rd}^{*N} <- mem32[start address + 4*N] optional Rn updated
STM	save multiple registers	{Rd}^{*N} -> mem32[start address + 4*N] optional Rn updated

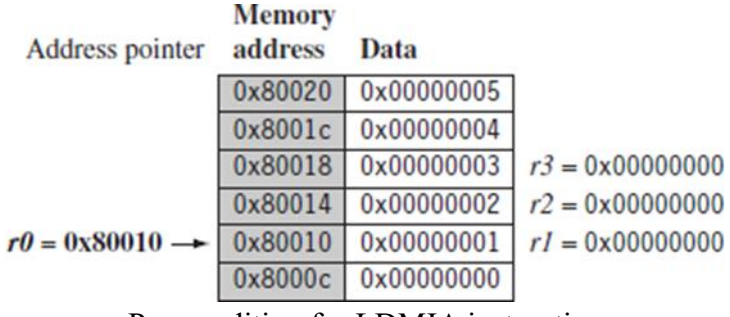
Table shows the different addressing modes for the load-store multiple instructions.

Addressing mode	Description	Start address	End address	Rn!
IA	increment after	Rn	$Rn + 4*N - 4$	$Rn + 4*N$
IB	increment before	$Rn + 4$	$Rn + 4*N$	$Rn + 4*N$
DA	decrement after	$Rn - 4*N + 4$	Rn	$Rn - 4*N$
DB	decrement before	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

Here N is the number of registers in the list of registers.
Any subset of the current bank of registers can be transferred to memory or fetched from memory. The base register Rn determines the source or destination address for a loadstore multiple instruction. This register can be optionally updated following the transfer. This occurs when register Rn is followed by the ! character, similiar to the single-register load-store using preindex with writeback.

Ex: In this example, register r0 is the base register Rn and is followed by !, indicating that the register is updated after the instruction is executed.In this case the range is from register r1 to r3 inclusive. Each register can also be listed, using a comma to separate each register within "{" and "}" brackets.

```
PRE     mem32[0x80018] = 0x03
        mem32[0x80014] = 0x02
        mem32[0x80010] = 0x01
        r0 = 0x00080010
        r1 = 0x00000000
        r2 = 0x00000000
        r3 = 0x00000000
        LDMIA r0!, {r1-r3}
POST r0 = 0x0008001c
        r1 = 0x00000001
        r2 = 0x00000002
        r3 = 0x00000003
```



Pre-condition for LDMIA instruction

Figure above shows a graphical representation. The base register r0 points to memory address 0x80010 in the PRE condition. Memory addresses 0x80010, 0x80014, and 0x80018 contain the values 1, 2, and 3 respectively. After the load multiple instruction executes registers r1, r2,

and r3 contain these values as shown in Figure below. The base register r0 now points to memory address 0x8001c after the last loaded word.

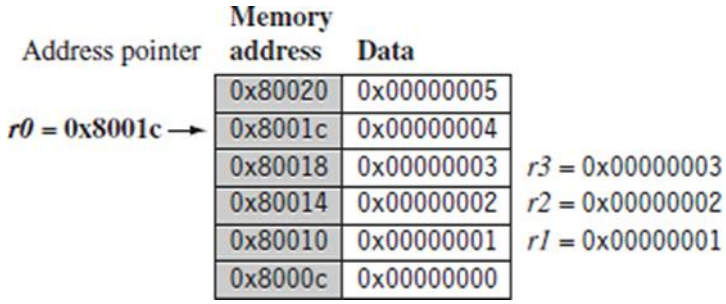
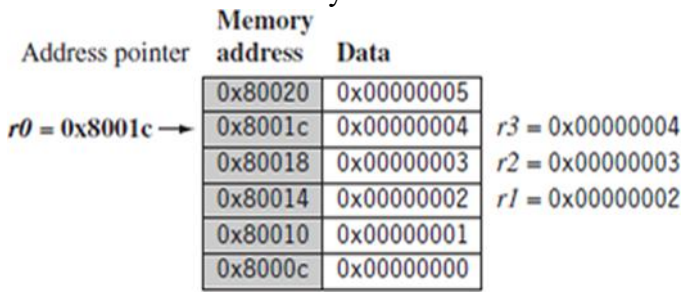


Fig. Post-condition for LDMIA instruction

Now replace the LDMIA instruction with a load multiple and increment before LDMIB instruction and use the same PRE conditions. The first word pointed to by register r0 is ignored and register r1 is loaded from the next memory location as shown below.



Post-condition for LDMIB instruction

After execution, register r0 now points to the last loaded memory location. This is in contrast with the LDMIA example, which pointed to the next memory location.

The decrement versions DA and DB of the load-store multiple instructions decrement the start address and then store to ascending memory locations. This is equivalent to descending memory but accessing the register list in reverse order. With the increment and decrement load multiples, you can access arrays forwards or backwards. They also allow for stack push and pull operations, illustrated later in this section.

Load Store Multiple pairs when base update used

Store multiple	Load multiple
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

Table shows a list of load-store multiple instruction pairs. If you use a store with base update, then the paired load instruction of the same number of registers will reload the data and restore the base address pointer. This is useful when you need to temporarily save a group of registers and restore them later.

This example shows an STM increment before instruction followed by an LDM decrement after instruction.

PRE r0 = 0x00009000
r1 = 0x00000009
r2 = 0x00000008
r3 = 0x00000007

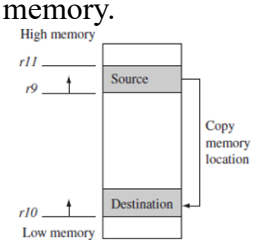
```
        STMIB r0!, {r1-r3} MOV r1, #1
        MOV r2, #2
        MOV r3, #3

PRE     r0 = 0x0000900c
        r1 = 0x00000001
        r2 = 0x00000002
        r3 = 0x00000003
        LDMDA r0!, {r1-r3}
POST r0 = 0x00009000
        r1 = 0x00000009
        r2 = 0x00000008
        r3 = 0x00000007
```

The STMIB instruction stores the values 7, 8, 9 to memory.
We then corrupt register r1 to r3.
The LDMDA reloads the original values and restores the base pointer r0.

Ex: This example is a simple routine that copies blocks of 32 bytes from a source address location to a destination address location. The example has two load-store multiple instructions, which use the same increment after addressing mode.

```
; r9 points to start of source data
; r10 points to start of destination data
; r11 points to end of the source loop
; load 32 bytes from source and update r9 pointer
        LDMIA r9!, {r0-r7}
; store 32 bytes to destination and update r10 pointer
        STMIA r10!, {r0-r7} ; and store them
; have we reached the end
        CMP r9, r11
        BNE loop
```

This routine relies on registers r9, r10, and r11 being set up before the code is executed. Registers r9 and r11 determine the data to be copied and register r10 points to the destination in memory for the data. LDMIA loads the data pointed to by register r9 into registers r0 to r7. It also updates r9 to point to the next block of data to be copied. STMIA copies the contents of registers r0 to r7 to the destination memory address pointed to by register r10. It also updates r10 to point to the next destination location. CMP and BNE compare pointers r9 and r11 to check whether the end of the block copy has been reached. If the block copy is complete, then the routine finishes; otherwise, the loop repeats with the updated values of register r9 and r10. The BNE is the branch instruction B with a condition mnemonic NE (not equal). If the previous compare instruction sets the condition flags to not equal, the branch instruction is executed. Figure shows the memory map of the block memory copy and how the routine moves through memory. Block memory copy in the memory.



Stack Operations

The ARM architecture uses the load-store multiple instructions to carry out stack operations. The pop operation (removing data from a stack) uses a load multiple instruction; similarly, the push operation (placing data onto the stack) uses a store multiple instruction.

When using a stack you have to decide whether the stack will grow up or down in memory. A stack is either ascending (A) or descending (D). Ascending stacks grow towards higher memory addresses; in contrast, descending stacks grow towards lower memory addresses.

When you use a full stack (F), the stack pointer sp points to an address that is the last used or full location (i.e., sp points to the last item on the stack). In contrast, if you use an empty stack (E) the sp points to an address that is the first unused or empty location (i.e., it points after the last item on the stack).

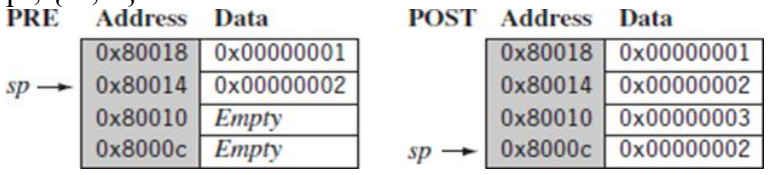
There is a number of load-store multiple addressing mode aliases available to support stack .Next to the pop column is the actual load multiple instruction equivalents. For example, a full ascending stack would have the notation FA appended to the load multiple instruction—LDMFA. This would be translated into an LDMDA instruction.

Addressing modes for stack operation

Addressing mode	Description	Pop	= LDM	Push	= STM
FA	full ascending	LDMFA	LDMDA	STMFA	STMIB
FD	full descending	LDMFD	LDMIA	STMFD	STMDB
EA	empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	empty descending	LDMED	LDMIB	STMED	STMDA

Ex: The STMFD instruction pushes registers onto the stack, updating the sp. Figure shows a push onto a full descending stack. You can see that when the stack grows the stack pointer points to the last full entry in the stack.

PRE r1 = 0x00000002
r4 = 0x00000003
sp = 0x00080014
STMFD sp!, {r1,r4}



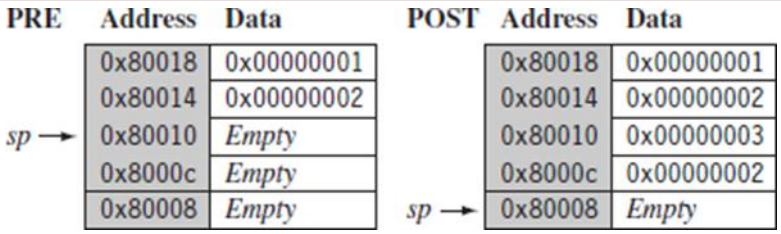
STMFD instruction—full stack push operation.

POST r1 = 0x00000002
r4 = 0x00000003
sp = 0x0008000c

Ex: In contrast, Figure shows a push operation on an empty stack using the STMED instruction. The STMED instruction pushes the registers onto the stack but updates register sp to point to the next empty location.

PRE r1 = 0x00000002
r4 = 0x00000003
sp = 0x00080010
STMED sp!, {r1,r4}

POST r1 = 0x00000002
r4 = 0x00000003
sp = 0x00080008



STMED instruction—empty stack push operation.

When handling a checked stack there are three attributes that need to be preserved: the stack base, the stack pointer, and the stack limit. The stack base is the starting address of the stack in memory. The stack pointer initially points to the stack base; as data is pushed onto the stack, the stack pointer descends memory and continuously points to the top of stack. If the stack pointer passes the stack limit, then a stack overflow error has occurred. Here is a small piece of code that checks for stack overflow errors for a descending stack:

```
; check for stack overflow
SUB sp, sp, #size
CMP sp, r10
BLLO _stack_overflow ; condition
```

The BLLO instruction is a branch with link instruction plus the condition mnemonic LO. If sp is less than register r10 after the new items are pushed onto the stack, then stack overflow error has occurred. If the stack pointer goes back past the stack base, then a stack underflow error has occurred.

Swap Instruction

The swap instruction is a special case of a load-store instruction. It swaps the contents of memory with the contents of a register. This instruction is an atomic operation—it reads and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.

```
Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]
```

SWP	swap a word between memory and a register	*tmp = mem32[Rn]* *mem32[Rn] = Rm* *Rd = tmp*
SWPB	swap a byte between memory and a register	*tmp = mem8[Rn]* *mem8[Rn] = Rm* *Rd = tmp*

Swap cannot be interrupted by any other instruction or any other bus access. We say the system "holds the bus" until the transaction is complete.

Ex: The swap instruction loads a word from memory into register r0 and overwrites the memory with register r1.

```
PRE    mem32[0x9000] = 0x12345678
       r0 = 0x00000000
       r1 = 0x11112222
       r2 = 0x00009000
       SWP r0, r1, [r2]
```

POST mem32[0x9000] = 0x11112222
r0 = 0x12345678
r1 = 0x11112222
r2 = 0x00009000

This instruction is particularly useful when implementing semaphores and mutual exclusion in an operating system. You can see from the syntax that this instruction can also have a byte size qualifier B, so this instruction allows for both a word and a byte swap.

Ex:This example shows a simple data guard that can be used to protect data from being written by another task. The SWP instruction "holds the bus" until the transaction is complete.

```
spin    MOV r1, =semaphore
        MOV r2, #1
        SWP r3, r2, [r1] ; hold the bus until complete
        CMP r3, #1
        BEQ spin
```

The address pointed to by the semaphore either contains the value 0 or 1. When the semaphore equals 1, then the service in question is being used by another process. The routine will continue to loop around until the service is released by the other process—in other words, when the semaphore address location contains the value 0.

Software Interrupt Instruction

A software interrupt instruction (SWI) causes a software interrupt exception, which provides a mechanism for applications to call operating system routines.

Syntax: SWI{<cond>} SWI_number

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts)

When the processor executes an SWI instruction, it sets the program counter pc to the offset 0x8 in the vector table. The instruction also forces the processor mode to SVC, which allows an operating system routine to be called in a privileged mode.

Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.

Ex: Here we have a simple example of an SWI call with SWI number 0x123456, used by ARM toolkits as a debugging SWI. Typically the SWI instruction is executed in user mode.

PRE cpsr = nzcVqift_USER
pc = 0x00008000
lr = 0x003fffff; lr = r14
r0 = 0x12
0x00008000 SWI 0x123456
POST cpsr = nzcVqIft_SVC
spsr = nzcVqift_USER
pc = 0x00000008
lr = 0x00008004

r0 = 0x12

Since SWI instructions are used to call operating system routines, you need some form of parameter passing. This is achieved using registers. In this example, register r0 is used to pass the parameter 0x12. The return values are also passed back via registers.

Code called the SWI handler is required to process the SWI call. The handler obtains the SWI number using the address of the executed instruction, which is calculated from the link register lr.

The SWI number is determined by

SWI_Number = <SWI instruction> AND NOT(0xff000000)

Here the SWI instruction is the actual 32-bit SWI instruction executed by the processor.

Ex: This example shows the start of an SWI handler implementation. The code fragment determines what SWI number is being called and places that number into register r10. You can see from this example that the load instruction first copies the complete SWI instruction into register r10. The BIC instruction masks off the top bits of the instruction, leaving the SWI number. We assume the SWI has been called from ARM state.

```
SWI_handler
        ;
        ;Store registers r0-r12 and the link register
        ;STMFD sp!, {r0-r12, lr}
        ; Read the SWI instruction
        LDR r10, [lr, #-4]
        ; Mask off top 8 bits
        BIC r10, r10, #0xff000000
        ; r10 - contains the SWI number
        BL service_routine
        ; return from SWI handler
        LDMFD sp!, {r0-r12, pc}^
```

The number in register r10 is then used by the SWI handler to call the appropriate SWI service routine.

Program Status Register Instructions

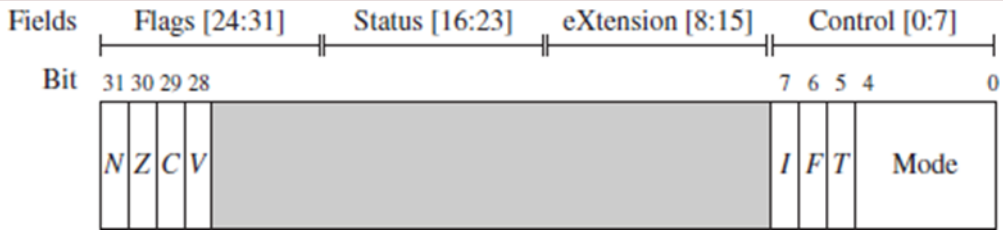
The ARM instruction set provides two instructions to directly control a program status register (psr). The MRS instruction transfers the contents of either the cpsr or spsr into a register; in the reverse direction, the MSR instruction transfers the contents of a register into the cpsr or spsr. Together these instructions are used to read and write the cpsr and spsr.

In the syntax you can see a label called fields. This can be any combination of control(c), extension (x), status (s), and flags (f). These fields relate to particular byte regions in a psr, as shown in Figure 3.9.

Syntax: MRS{<cond>} Rd,<cpsr|spsr>

MSR{<cond>} <cpsr|spsr>_<fields>,Rm

MSR{<cond>} <cpsr|spsr>_<fields>,#immediate



psr byte fields.

MRS	copy program status register to a general-purpose register	*Rd = psr*
MSR	move a general-purpose register to a program status register	*psr[field] = Rm*
MSR	move an immediate value to a program status register	*psr[field] = immediate*

The c field controls the interrupt masks, Thumb state, and processor mode.Example below shows how to enable IRQ interrupts by clearing the I mask. This operation involves using both the MRS and MSR instructions to read from and then write to the cpsr.

Ex: The MSR first copies the cpsr into register r1. The BIC instruction clears bit 7 of r1. Register r1 is then copied back into the cpsr, which enables IRQ interrupts. You can see from this example that this code preserves all the other settings in the cpsr and only modifies the I bit in the control field.

```
PRE   cpsr = nzcvqIFt_SVC
      MRS r1, cpsr
      BIC r1, r1, #0x80 ; 0b01000000
      MSR cpsr_c, r1
POST cpsr = nzcvqiFt_SVC
```

This example is in SVC mode. In user mode you can read all cpsr bits, but you can only update the condition flag field f.

Coprocessor Instructions

Coprocessor instructions are used to extend the instruction set. A coprocessor can either provide additional computation capability or be used to control the memory subsystem including caches and memory management. The coprocessor instructions include data processing, register transfer, and memory transfer instructions. We will provide only a short overview since these instructions are coprocessor specific. Note that these instructions are only used by cores with a coprocessor.

Syntax: CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2}
<MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}
<LDC|STC>{<cond>} cp, Cd, addressing

CDP	coprocessor data processing—perform an operation in a coprocessor
MRC MCR	coprocessor register transfer—move data to/from coprocessor registers
LDC STC	coprocessor memory transfer—load and store blocks of memory to/from a coprocessor

In the syntax of the coprocessor instructions, the cp field represents the coprocessor number between p0 and p15. The opcode fields describe the operation to take place on the coprocessor. The Cn, Cm, and Cd fields describe registers within the coprocessor. The coprocessor operations and registers depend on the specific coprocessor you are using. Coprocessor 15 (CP15) is reserved for system control purposes, such as memory management, write buffer control, cache control, and identification registers.

Ex: This example shows a CP15 register being copied into a general-purpose register.

; transferring the contents of CP15 register c0 to register r10 MRC p15, 0, r10, c0, c0, 0

Here CP15 register-0 contains the processor identification number. This register is copied into the general-purpose register r10.

Loading Constants

You might have noticed that there is no ARM instruction to move a 32-bit constant into a register. Since ARM instructions are 32 bits in size, they obviously cannot specify a general 32- bit constant.

To aid programming there are two pseudo instructions to move a 32-bit value into a register.
Syntax: LDR Rd, =constant
ADR Rd, label

LDR	load constant pseudoinstruction	*Rd* = 32-bit constant
ADR	load address pseudoinstruction	*Rd* = 32-bit relative address

LDR load constant pseudo instruction Rd=32-bit constant

ADR load address pseudo instruction Rd=32-bit relative address

The first pseudo instruction writes a 32-bit constant to a register using whatever instructions are available. It defaults to a memory read if the constant cannot be encoded using other instructions. The second pseudo instruction writes a relative address into a register, which will be encoded using a pc-relative expression.

Pseudoinstruction	Actual instruction
`LDR r0, =0xff`	`MOV r0, #0xff`
`LDR r0, =0x55555555`	`LDR r0, [pc, #offset_12]`

Ex: Loading the constant 0xff00ffff using an MVN.
PRE none...
MVN r0, #0x00ff00ffff
POST r0 = 0xff00ffff

Another useful pseudo instruction is the ADR instruction or address relative. This instruction places the address of the given label into register Rd, using a pc-relative add or subtract.

Module III

C Compilers and optimization

Overview of C Compilers and optimization

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e., CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand a smaller number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Optimizing code takes time and reduces source code readability. Usually, it's only worth optimizing functions that are frequently executed and important for performance. It is recommended to use a performance profiling tool, found in most ARM simulators, to find these frequently executed functions.

C compilers must translate your C function literally into assembler so that it works for all possible inputs. In practice, many of the input combinations are not possible or won't occur.

Let's start by looking at an example of the problems the compiler faces.

Example: The memclr function clears N bytes of memory at address data.

```
void memclr(char *data, int N)
{ for (; N>0; N-
-)
{
*data=0; data++;
}
}
```

- No matter how advanced the compiler, it does not know whether N can be 0 on input or not. Therefore, the compiler needs to test for this case explicitly before the first iteration of the loop.
- The compiler doesn't know whether the data array pointer is four-byte aligned or not. If it is four-byte aligned, then the compiler can clear four bytes at a time using an int store rather than a char store.
- Nor does it know whether N is a multiple of four or not. If N is a multiple of four, then the compiler can repeat the loop body four times or store four bytes at a time using an int store.

The compiler must be conservative and assume all possible values for N and all possible alignments for data. To write efficient C code, you must be aware of areas where the C compiler must be conservative, the limits of the processor architecture the C compiler is mapping to, and the limits of a specific C compiler.

Basic C Data Types

ARM processors have 32-bit registers and 32-bit data processing operations. The ARM architecture is a RISC load/store architecture. In other words, you must load values from memory into registers before acting on them. There are no arithmetic or logical instructions that manipulate values in memory directly.

- Early versions of the ARM architecture (ARMv1 to ARMv3) provided hardware support for loading and storing unsigned 8-bit and unsigned or signed 32-bit values.

Table 1 shows the load/store instruction classes available by ARM architecture.

- In Table 1 loads that act on 8- or 16-bit values extend the value to 32 bits before writing to an ARM register. Unsigned values are zero-extended, and signed values sign extended. This means that the cast of a loaded value to an int type does not cost extra instructions. Similarly, a store of an 8- or 16-bit value selects the lowest 8 or 16 bits of the register. The cast of an int to smaller type does not cost extra instructions on a store.
- The ARMv4 architecture and above support signed 8-bit and 16-bit loads and stores directly, through new instructions.
- Finally, ARMv5 adds instruction support for 64-bit load and stores. This is available in ARM9E and later cores.
- Prior to ARMv4, ARM processors were not good at handling signed 8-bit or any 16-bit values. Therefore, ARM C compilers define char to be an unsigned 8-bit value, rather than a signed 8-bit value as is typical in many other compilers.

Table 1: Load and store instructions by ARM architecture.

Architecture	Instruction	Action
Pre-ARMv4	LDRB	load an unsigned 8-bit value
	STRB	store a signed or unsigned 8-bit value
	LDR	load a signed or unsigned 32-bit value
	STR	store a signed or unsigned 32-bit value
ARMv4	LDRSB	load a signed 8-bit value
	LDRH	load an unsigned 16-bit value
	LDRSH	load a signed 16-bit value
	STRH	store a signed or unsigned 16-bit value
ARMv5	LDRD	load a signed or unsigned 64-bit value
	STRD	store a signed or unsigned 64-bit value

Table 2: C compiler datatype mappings.

C Data Type	Implementation
char	unsigned 8-bit byte
short	signed 16-bit halfword
int	signed 32-bit word
long	signed 32-bit word
long long	signed 64-bit double word

Local variable types

- ARMv4-based processors can efficiently load and store 8-, 16-, and 32-bit data. However, most ARM data processing operations are 32-bit only. For this reason, you should use a 32-bit datatype, int or long, for local variables wherever possible.
- Avoid using char and short as local variable types, even if you are manipulating an 8- or 16-bit value. The one exception is when you want wrap-around to occur (If you require modulo arithmetic of the form 255+1=0, then use the char type.)

To see the effect of local variable types, let's consider a simple example.

Examples: We'll look in detail at a checksum function that sums the values in a data packet. Most communication protocols (such as TCP/IP) have a checksum or cyclic redundancy check (CRC) routine to check for errors in a data packet.

1. The following code checksums a data packet containing 64 words. It shows why you should avoid using char for local variables.

```
int checksum_v1(int *data)
{ char i; int sum = 0;
for (i = 0; i < 64;
i++)
{
sum += data[i];
}
return sum; }
```

At first sight it looks as though declaring i as a char is efficient. You may be thinking that a char uses less register space or less space on the ARM stack than an int. On the ARM, both these assumptions are wrong.

All ARM registers are 32-bit and all stack entries are at least 32-bit. Furthermore, to implement the i++ exactly, the compiler must account for the case when i = 255. Any attempt to increment 255 should produce the answer 0.

Consider the compiler output for this function. We've added labels and comments to make the assembly clear.

```
checksum_v1
        BCC
        MOV
        MOV r2,r0 ; r2 = data
        MOV r0,#0 ; sum = 0
        MOV r1,#0 ;i=0
checksum_v1_loop
        LDR r3,[r2,r1,LSL #2] ; r3 = data[i]
        ADD r1,r1,#1 ; r1 = i+1
        AND r1,r1,#0xff ; i = (char)r1
        CMP r1,#0x40 ; compare i, 64
        ADD r0,r3,r0 ; sum += r3
```

```
checksum_v1_loop ; if (i<64) loop
        pc,r14 ; return sum
```

2. Now compare this to the compiler output where instead we declare i as an unsigned int.

```
checksum_v2
MOV r2,r0 ; r2 = data
MOV r0,#0 ; sum = 0
MOV r1,#0;i=0
checksum_v2_loop
LDR r3,[r2,r1,LSL #2] ; r3 = data[i]
ADD r1,r1,#1 ; r1++
CMP r1,#0x40 ; compare i, 64
ADD r0,r3,r0 ; sum += r3
BCC checksum_v2_loop ; if (i<64) goto loop
MOV pc,r14 ; return sum
```

In the first case, the compiler inserts an extra AND instruction to reduce i to the range 0 to 255 before the comparison with 64. This instruction disappears in the second case.

3. Next, suppose the data packet contains 16-bit values and we need a 16-bit checksum.

```
short checksum_v3(short *data)
{ unsigned int i;
short sum = 0; for (i
= 0; i < 64; i++)
{
sum = (short)(sum + data[i]);
}
return sum; }
```

You may wonder why the for loop body doesn't contain the code: sum += data[i];

The expression sum + data[i] is an integer and so can only be assigned to a short using an (implicit or explicit) narrowing cast. As you can see in the following assembly output, the compiler must insert extra instructions to implement the narrowing cast:

```
checksum_v3
MOV r2,r0 ; r2 = data
MOV r0,#0 ; sum = 0
MOV r1,#0;i=0
checksum_v3_loop
ADD r3,r2,r1,LSL #1 ; r3 = &data[i]
LDRH r3,[r3,#0] ; r3 = data[i]
ADD r1,r1,#1 ; i++
CMP r1,#0x40 ; compare i, 64
ADD r0,r3,r0; r0 = sum + r3
MOV r0,r0,LSL #16
MOV r0,r0,ASR #16 ; sum = (short)r0
checksum_v3_loop ; if (i<64) goto loop
```

pc,r14 ; return sum

The loop is now three instructions longer than the loop for example checksum_v2 earlier. There are two reasons for the extra instructions:

- The LDRH instruction does not allow for a shifted address offset as the LDR instruction did in checksum_v2. Therefore, the first ADD in the loop calculates the address of item i in the array. The LDRH loads from an address with no offset.
- The cast reducing total + array[i] to a short requires two MOV instructions. The compiler shifts left by 16 and then right by 16 to implement a 16-bit sign extend. The shift right is a sign-extending shift so it replicates the sign bit to fill the upper 16 bits.

We can avoid the second problem by using an int type variable to hold the partial sum. We only reduce the sum to a short type at the function exit.

However, the first problem is a new issue. We can solve it by accessing the array by incrementing the pointer data rather than using an index as in data[i]. This is efficient regardless of array type size or element size. All ARM load and store instructions have a post increment addressing mode.

4. The checksum_v4 code fixes all the problems we have discussed in this section. It uses int type local variables to avoid unnecessary casts. It increments the pointer data instead of using an index offset data[i].

```
short checksum_v4(short *data)
{ unsigned int i; int
sum=0; for (i=0;
i<64; i++)
{
sum += *(data++);
} return
(short)sum; }
```

The *(data++) operation translates to a single ARM instruction that loads the data and increments the data pointer. Of course, you could write sum += *data; data++; or even *data++ instead if you prefer.

The compiler produces the following output. Three instructions have been removed from the inside loop, saving three cycles per loop compared to checksum_v3.

```
checksum_v4
MOV r2,#0 ; sum = 0
MOV r1,#0;i=0
checksum_v4_loop
LDRSH r3,[r0],#2 ; r3 = *(data++)
ADD r1,r1,#1 ; i++
CMP r1,#0x40 ; compare i, 64
ADD r2,r3,r2 ; sum += r3
checksum_v4_loop; if (sum<64) goto loop
r0,r2,LSL #16
```

```
MOV r0,r0,ASR #16 ; r0 = (short)sum
MOV pc,r14 ; return r0
```

The compiler is still performing one cast to a 16-bit range, on the function return. You could remove this also by returning an int result.

Function Argument types

We know that converting local variables from types char or short to type int increases performance and reduces code size. The same holds for function arguments.

Consider the following simple function, which adds two 16-bit values, halving the second, and returns a 16-bit sum:

```
short add_v1(short a, short b)
{ return a+ (b >>
1); }
```

This function is useful test case to illustrate the problems faced by the compiler. The input values a, b, and the return value will be passed in 32-bit ARM registers.

Should the compiler assume that these 32-bit values are in the range of a short type, that is, 32,768 to 32,767? Or should the compiler force values to be in this range by sign-extending the lowest 16 bits to fill the 32-bit register? The compiler must make compatible decisions for the function caller and callee. Either the caller or callee must perform the cast to a short type.

We say that function arguments are passed wide if they are not reduced to the range of the type and narrow if they are.

If the compiler

- passes arguments wide, then the callee must reduce function arguments to the correct range.
- passes arguments narrow, then the caller must reduce the range.
- returns values wide, then the caller must reduce the return value to the correct range.
- returns values narrow, then the callee must reduce the range before returning the value.

The armcc output for add_v1 shows that the compiler casts the return value to a short type but does not cast the input values. It assumes that the caller has already ensured that the 32-bit values r0 and r1 are in the range of the short type. This shows narrow passing of arguments and return value.

```
add_v1
ADD r0,r0,r1,ASR #1 ; r0 = (int)a + ((int)b >> 1)
MOV r0,r0,LSL #16
MOV r0,r0,ASR #16 ; r0 = (short)r0 MOV
pc,r14 ; return r0
```

The gcc compiler we used is more cautious and makes no assumptions about the range of argument value. This version of the compiler reduces the input arguments to the range of a short in both the caller and the callee. It also casts the return value to a short type. Here is the compiled code for add_v1:

```
add_v1_gcc
MOV r0, r0, LSL #16
MOV r1, r1, LSL #16
MOV r1, r1, ASR #17 ; r1 = (int)b >> 1
ADD r1, r1, r0, ASR #16 ; r1 += (int)a
MOV r1, r1, LSL #16
MOV r0, r1, ASR #16 ; r0 = (short)r1
MOV pc, lr ; return r0
```

Whatever the merits of different narrow and wide calling protocols, you can see that char or short type function arguments and return values introduce extra casts. These increase code size and decrease performance. It is more efficient to use the int type for function arguments and return values, even if you are only passing an 8-bit value.

Signed versus Unsigned Types

The previous sections demonstrate the advantages of using int rather than a char or short type for local variables and function arguments. This section compares the efficiencies of signed int and unsigned int.

- If your code uses addition, subtraction, and multiplication, then there is no performance difference between signed and unsigned operations.
- However, there is a difference when it comes to division. Consider the following short example that averages two integers:

```
int average_v1(int a, int b)
{
return (a+b)/2;
} This
compiles to
average_v1
ADD r0,r0,r1 ; r0=a+b
ADD r0,r0,r0,LSR #31 ; if (r0<0) r0++
MOV r0,r0,ASR #1 ; r0 = r0 >> 1
MOV pc,r14 ; return r0
```

Notice that the compiler adds one to the sum before shifting by right if the sum is negative. In other words it replaces x/2 by the statement: (x<0) ? ((x+1) >> 1): (x >> 1)

It must do this because x is signed. In C on an ARM target, a divide by two is not a right shift if x is negative. For example, -3 >> 1= -2 but -3/2= -1. Division rounds towards zero, but arithmetic right shift rounds towards -infinity.

It is more efficient to use unsigned types for divisions. The compiler converts unsigned power of two divisions directly to right shifts. For general divisions, the divide routine in the C library is faster for unsigned types.

SUMMARY: The Efficient Use of C Types

- For local variables held in registers, don't use a char or short type unless 8-bit or 16-bit modular arithmetic is necessary. Use the signed or unsigned int types instead. Unsigned types are faster when you use divisions.
- For array entries and global variables held in main memory, use the type with the smallest size possible to hold the required data. This saves memory footprint. The ARMv4 architecture is efficient at loading and storing all data widths provided you traverse arrays by incrementing the array pointer. Avoid using offsets from the base of the array with short type arrays, as LDRH does not support this.
- Use explicit casts when reading array entries or global variables into local variables or writing local variables out to array entries.

The casts make it clear that for fast operation you are taking a narrow width type stored in memory and expanding it to a wider type in the registers. Switch on implicit narrowing cast warnings in the compiler to detect implicit casts.

- Avoid implicit or explicit narrowing casts in expressions because they usually cost extra cycles. Casts on loads or stores are usually free because the load or store instruction performs the cast for you.
- Avoid char and short types for function arguments or return values. Instead use the int type even if the range of the parameter is smaller. This prevents the compiler performing unnecessary casts.

C Looping Structures

Loops with a fixed number of iterations

What is the most efficient way to write a for loop on the ARM? Let's return to our checksum example and look at the looping structure. Here is the last version of the 64-word packet checksum routine. This shows how the compiler treats a loop with incrementing count i++.

```
int checksum_v5(int *data)
{ unsigned int i; int
sum=0; for (i=0; i<64;
i++)
{
sum += *(data++);
}
return sum;
}
```

This compiles to

```
checksum_v5
MOV r2,r0 ; r2 = data
MOV r0,#0 ; sum = 0
MOV r1,#0;i=0
checksum_v5_loop
LDR r3,[r2],#4 ; r3 = *(data++) ADD
r1,r1,#1 ; i++
CMP r1,#0x40 ; compare i, 64
```

```
ADD r0,r3,r0; sum += r3
BCC checksum_v5_loop ; if (i<64) goto loop
MOV pc,r14 ; return sum
```

It takes three instructions to implement the for loop structure:

- An ADD to increment i
- A compare to check if i is less than 64
- A conditional branch to continue the loop if i < 64

This is not efficient. On the ARM, a loop should only use two instructions:

- A subtract to decrement the loop counter, which also sets the condition code flags on the result.
- A conditional branch instruction

The key point is that the loop counter should count down to zero rather than counting up to some arbitrary limit. Then the comparison with zero is free since the result is stored in the condition flags.

Example:

1. shows the improvement if we switch to a decrementing loop rather than an incrementing loop.

```
int checksum_v6(int *data)
{ unsigned int i; int
sum=0; for (i=64; i!=0; i-
-)
{
sum += *(data++);
}
return sum;
}
```

This compiles to

```
checksum_v6
MOV r2,r0 ; r2 = data
MOV r0,#0 ; sum = 0
MOV r1,#0x40 ;i= 64
checksum_v6_loop
LDR r3,[r2],#4 ; r3 = *(data++)
SUBS r1,r1,#1 ; i-- and set flags
ADD r0,r3,r0 ; sum += r3
BNE checksum_v6_loop ; if (i!=0) goto loop
MOV pc,r14 ; return sum
```

The SUBS and BNE instructions implement the loop. Our checksum example now has the minimum number of four instructions per loop. This is much better than six for checksum_v1 and eight for checksum_v3.

For an unsigned loop counter i we can use either of the loop continuation conditions i!=0 or i>0. As i can't be negative, they are the same condition.

For a signed loop counter, it is tempting to use the condition i>0 to continue the loop. You might expect the compiler to generate the following two instructions to implement the loop:

SUBS r1,r1,#1 ; compare i with 1, i=i-1
BGT loop ; if (i+1>1) goto loop

In fact, the compiler will generate,

SUB r1,r1,#1 ; i--
CMP r1,#0 ; compare i with 0
BGT loop ; if (i>0) goto loop

For the first piece of code the SUBS instruction compares i with 1 and then decrements i. Since -0x80000000 < 1, the loop terminates.

For the second piece of code, we decrement i and then compare with 0. Modulo arithmetic means that i now has the value+0x7fffffff, which is greater than zero. Thus, the loop continues for many iterations.

Therefore you should use the termination condition i!=0 for signed or unsigned loop counters. It saves one instruction over the condition i>0 for signed i.

Loops using a variable number of iterations.

Now suppose we want our checksum routine to handle packets of arbitrary size. We pass in a variable N giving the number of words in the data packet. Using the lessons from the last section we count down until N=0 and don't require an extra loop counter i.

Example: 1. The checksum_v7 shows how the compiler handles a for loop with a variable number of iterations N.

```
int checksum_v7(int *data, unsigned int N)
{ int sum=0; for
(; N!=0; N--)
{
sum += *(data++);
}
return sum;
}
This compiles to
checksum_v7
MOV r2,#0 ; sum = 0
CMP r1,#0 ; compare N, 0
BEQ checksum_v7_end; if (N==0) goto end
checksum_v7_loop

LDR r3,[r0],#4 ; r3 = *(data++)
SUBS r1,r1,#1 ; N-- and set flags
ADD r2,r3,r2 ; sum += r3
```

```
BNE checksum_v7_loop; if (N!=0) goto loop
checksum_v7_end
MOV r0,r2 ; r0 = sum
MOV pc,r14 ; return r0
```

Notice that the compiler checks that N is nonzero on entry to the function. Often this check is unnecessary since you know that the array won't be empty. In this case a do-while loop gives better performance and code density than a for loop.

2. This example shows how to use a do-while loop to remove the test for N being zero that occurs in a for loop.

```
int checksum_v8(int *data, unsigned int N)
{ int
sum=0; do
{
sum += *(data++);
} while (--N!=0); return sum;
}
```

The compiler output is now

```
checksum_v8

MOV r2,#0; sum = 0
checksum_v8_loop
LDR r3,[r0],#4 ; r3 = *(data++)
SUBS r1,r1,#1 ; N-- and set flags
ADD r2,r3,r2 ; sum += r3
BNE checksum_v8_loop ; if (N!=0) goto loop
MOV r0,r2 ; r0 = sum
MOV pc,r14 ; return r0
```

Compare this with the output for checksum_v7 to see the two-cycle saving.

Loop Unrolling

We saw in previous Section that each loop iteration costs two instructions in addition to the body of the loop: a subtract to decrement the loop count and a conditional branch. We call these instructions the loop overhead.

On ARM7 or ARM9 processors the subtract takes one cycle and the branch three cycles, giving an overhead of four cycles per loop.

You can save some of these cycles by unrolling a loop - repeating the loop body several times, and reducing the number of loop iterations by the same proportion.

Example,

1. let's unroll our packet checksum example four times.

The following code unrolls our packet checksum loop by four times. We assume that the number of words in the packet N is a multiple of four.

```
int checksum_v9(int *data, unsigned int N)
{
int sum=0;
do {
sum += *(data++);
sum += *(data++);
sum += *(data++);
sum += *(data++);
N-= 4;
} while ( N!=0);
Return sum;
}
```

This compiles to

```
checksum_v9
MOV r2,#0 ; sum = 0
checksum_v9_loop
LDR r3,[r0],#4 ; r3 = *(data++)
SUBS r1,r1,#4 ; N -= 4 & set flags
ADD r2,r3,r2 ;sum += r3
LDR r3,[r0],#4 ;r3 = *(data++)
ADD r2,r3,r2 ;sum += r3
LDR r3,[r0],#4 ;r3 = *(data++)
ADD r2,r3,r2 ;sum += r3
LDR r3,[r0],#4 ;r3 = *(data++)
ADD r2,r3,r2 ;sum += r3
BNE checksum_v9_loop ;if (N!=0) goto loop
MOV r0,r2 ;r0 = sum
MOV pc,r14 ;return r0
```

loop overhead has been reduced the from 4N cycles to (4N)/4=N cycles.

On the ARM7TDMI, this accelerates the loop from 8 cycles per accumulate to 20/4=5 cycles per accumulate, nearly doubling the speed!

There are two questions you need to ask when unrolling a loop:

• How many times should I unroll the loop?

• What if the number of loop iterations is not a multiple of the unroll amount? For example, what if N is not a multiple of four in checksum_v9?

To start with the first question, only unroll loops that are important for the overall performance of the application. Otherwise unrolling will increase the code size with little performance benefit. Unrolling may even reduce performance by evicting more important code from the cache.

For the second question, try to arrange it so that array sizes are multiples of unroll amount. If this isn't possible, then you must add extra code to take care of the leftover cases. This increases the code size a little but keeps the performance high.

2. This example handles the checksum of any size of data packet using a loop that has been unrolled four times.

```
int checksum_v10(int *data, unsigned int N)
{ unsigned int i; int
sum=0; for (i=N/4; i!=0; i--)
{
sum += *(data++);
sum+= *(data++);
sum +=*(data++);
sum +=*(data++);
} for (i=N&3; i!=0; i--)
{
sum += *(data++);
}
return sum; }
```

The second for loop handles the remaining cases when N is not a multiple of four. Note that both N/4 and N&3 can be zero, so we can't use do-while loops.

SUMMARY: Writing Loops Efficiently

- Use loops that count down to zero. Then the compiler does not need to allocate a register to hold the termination value, and the comparison with zero is free.
- Use unsigned loop counters by default and the continuation condition i!=0 rather than i>0. This will ensure that the loop overhead is only two instructions.
- Use do-while loops rather than for loops when you know the loop will iterate at least once. This saves the compiler checking to see if the loop count is zero.
- Unroll important loops to reduce the loop overhead. Do not over unroll. If the loop overhead is small as a proportion of the total, then unrolling will increase code size and hurt the performance of the cache.
- Try to arrange that the number of elements in arrays are multiples of four or eight. You can then unroll loops easily by two, four, or eight times without worrying about the leftover array elements.

Register Allocation

The compiler attempts to allocate a processor register to each local variable you use in a C function. It will try to use the same register for different local variables if the use of the variables does not overlap.

When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called spilled or swapped out variables since they are written out to memory. Spilled variables are slow to access compared to variables allocated to registers.

To implement a function efficiently, need to

- minimize the number of spilled variables
- ensure that the most important and frequently accessed variables are stored in registers

First let's look at the number of processor registers the ARM C compilers have available for allocating variables. Table 3 shows the standard register names and usage when following the ARM-Thumb procedure call standard (ATPCS), which is used in code generated by C compilers.

Table 3: C compiler register usage.

Register number	Alternate register names	ATPCS register usage
r0	*a1*	Argument registers. These hold the first four function arguments on a function call and the return value on a function return. A function may corrupt these registers and use them as general scratch registers within the function.
r1	*a2*	
r2	*a3*	
r3	*a4*	
r4	*v1*	General variable registers. The function must preserve the callee values of these registers.
r5	*v2*	
r6	*v3*	
r7	*v4*	
r8	*v5*	
r9	*v6 sb*	General variable register. The function must preserve the callee value of this register except when compiling for *read-write position independence* (RWPI). Then *r9* holds the *static base* address. This is the address of the read-write data.
r10	*v7 sl*	General variable register. The function must preserve the callee value of this register except when compiling with stack limit checking. Then *r10* holds the stack limit address.
r11	*v8 fp*	General variable register. The function must preserve the callee value of this register except when compiling using a frame pointer. Only old versions of *armcc* use a frame pointer.
r12	*ip*	A general scratch register that the function can corrupt. It is useful as a scratch register for function veneers or other intraprocedure call requirements.
r13	*sp*	The stack pointer, pointing to the full descending stack.
r14	*lr*	The link register. On a function call this holds the return address.
r15	*pc*	The program counter.

Provided the compiler is not using software stack checking, then the C compiler can use registers r0 to r12 and r14 to hold variables. It must save the callee values of r4 to r11 and r14 on the stack if using these registers.

In theory, the C compiler can assign 14 variables to registers without spillage. In practice, some compilers use a fixed register such as r12 for intermediate scratch working and do not assign variables to this register. Also, complex expressions require intermediate working registers to

evaluate. Therefore, to ensure good assignment to registers, you should try to limit the internal loop of functions to using at most 12 local variables.

If the compiler does need to swap out variables, then it chooses which variables to swap out based on frequency of use. A variable used inside a loop counts multiple times. You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.

The register keyword in C hints that a compiler should allocate the given variable to a register. However, different compilers treat this keyword in different ways, and different architectures have a different number of available registers (for example, Thumb and ARM). Therefore, we recommend that you avoid using register and rely on the compiler's normal register allocation routine.

SUMMARY: Efficient Register Allocation

- Try to limit the number of local variables in the internal loop of functions to 12. The compiler should be able to allocate these to ARM registers.
- You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.

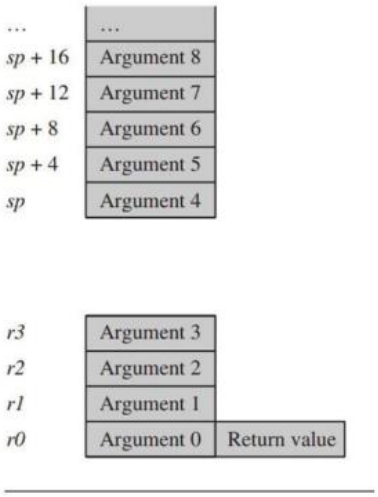
Function calls

The ARM Procedure Call Standard (APCS) defines how to pass function arguments and return values in ARM registers.

The more recent ARM-Thumb Procedure Call Standard (ATPCS) covers ARM and Thumb interworking as well.

The first four integer arguments are passed in the first four ARM registers: r0, r1, r2, and r3. Subsequent integer arguments are placed on the full descending stack, ascending in memory as in Figure 1. Function return integer values are passed in r0.

Two-word arguments such as long long or double are passed in a pair of consecutive argument registers and returned in r0, r1. The compiler may pass structures in registers or by reference according to command line compiler options.



ATPCS argument passing.

The first point to note about the procedure call standard is the four-register rule. Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments.

- For functions with four or fewer arguments, the compiler can pass all the arguments in registers.
- For functions with more arguments, both the caller and callee must access the stack for some arguments.

If your C function needs more than four arguments, or your C++ method more than three explicit arguments, then it is almost always more efficient to use structures. Group related arguments into structures, and pass a structure pointer rather than multiple arguments.

Example: 1. illustrates the benefits of using a structure pointer. First, we show a typical routine to insert N bytes from array data into a queue. We implement the queue using a cyclic buffer with start address Q_start (inclusive) and end address Q_end (exclusive).

```
char *queue_bytes_v1( char *Q_start, /* Queue
buffer start address */ char *Q_end, /* Queue
buffer end address */ char *Q_ptr, /* Current
queue pointer position */ char *data, /* Data to
insert into the queue */ unsigned int N) /* Number of bytes
to insert */
{ do
{
*(Q_ptr++) = *(data++); if
(Q_ptr == Q_end)
{
Q_ptr = Q_start;
}
} while (--N); return
Q_ptr;
}
```

This compiles to

```
queue_bytes_v1
STR r14,[r13,#-4]! ; save lr on the stack
LDR r12,[r13,#4] ; r12 = N
queue_v1_loop
LDRB r14,[r3],#1 ; r14 = *(data++)
STRB r14,[r2],#1 ; *(Q_ptr++) = r14
CMP r2,r1 ; if (Q_ptr == Q_end)
MOVEQ r2,r0 ; {Q_ptr = Q_start;}
SUBS r12,r12,#1 ; --N and set flags
BNE queue_v1_loop ; if (N!=0) goto loop
MOV r0,r2 ; r0 = Q_ptr
LDR pc,[r13],#4 ; return r0
```

Compare this with a more structured approach using three function arguments.

2. The following code creates a Queue structure and passes this to the function to reduce the number of function arguments.

```
typedef struct {
char *Q_start; /* Queue buffer start address */
char *Q_end; /* Queue buffer end address */
char *Q_ptr;
} Queue;
/* Current queue pointer position */
void queue_bytes_v2(Queue *queue, char *data, unsigned int N)
{
char *Q_ptr = queue->Q_ptr; char *Q_end = queue->Q_end; do
{
*(Q_ptr++) = *(data++); if
(Q_ptr == Q_end)
{
Q_ptr = queue->Q_start;
}
} while (--N); queue->Q_ptr
= Q_ptr;
}
```

This compiles to

```
queue_bytes_v2
STR r14,[r13,#-4]! ; save lr on the stack
LDR r3,[r0,#8] ; r3 = queue->Q_ptr
LDR r14,[r0,#4] ; r14 = queue->Q_end queue_v2_loop
LDRB r12,[r1],#1 ; r12 = *(data++) STRB r12,[r3],#1 ;
*(Q_ptr++) = r12
CMP r3,r14 ; if (Q_ptr == Q_end) LDREQ
r3,[r0,#0] ; Q_ptr = queue->Q_start SUBS
r2,r2,#1 ; --N and set flags
BNE queue_v2_loop ; if (N!=0) goto loop
STR r3,[r0,#8] ; queue->Q_ptr = r3
LDR pc,[r13],#4 ; return
```

The queue_bytes_v2 is one instruction longer than queue_bytes_v1, but it is in fact more efficient overall.

The second version has only three function arguments rather than five. Each call to the function requires only three register setups. This compares with four register setups, a stack push, and a stack pull for the first version. There is a net saving of two instructions in function call overhead.

There are other ways of reducing function call overhead if your function is very small and corrupts few registers (uses few local variables). Put the C function in the same C file as the functions that will call it. The C compiler then knows the code generated for the callee function and can make optimizations in the caller function:

- The caller function need not preserve registers that it can see the callee doesn't corrupt. Therefore, the caller function need not save all the ATPCS corruptible registers.
- If the callee function is very small, then the compiler can inline the code in the caller function. This removes the function call overhead completely.

3. The function uint_to_hex converts a 32-bit unsigned integer into an array of eight hexa-decimal digits. It uses a helper function nybble_to_hex, which converts a digit d in the range 0 to 15 to a hexadecimal digit.

```
unsigned int nybble_to_hex(unsigned int d)
{ if (d<10) {
return d + '0';
} returnd- 10+
'A';
}
void uint_to_hex(char *out, unsigned int in)
{ unsigned int i;
for (i=8; i!=0; i--)
{
in = (in << 4) | (in >> 28); /* rotate in left by 4 bits */
*(out++) = (char)nybble_to_hex(in & 15);
}
}
```

When we compile this, we see that uint_to_hex doesn't call nybble_to_hex at all! In the following compiled code, the compiler has inlined the uint_to_hex code. This is more efficient than generating a function call.

```
uint_to_hex
MOV r3,#8 ;i=8
uint_to_hex_loop
MOV r1,r1,ROR #28 ; in = (in << 4)|(in >> 28)
AND r2,r1,#0xf ; r2 = in & 15
CMP r2,#0xa ; if (r2>=10)
ADDCS r2,r2,#0x37 ; r2 +='A'-10
ADDCC r2,r2,#0x30 ; else r2 +='0'
STRB r2,[r0],#1 ; *(out++) = r2
SUBS r3,r3,#1 ; i-- and set flags
BNE uint_to_hex_loop ; if (i!=0) goto loop
MOV pc,r14 ; return
```

The compiler will only inline small functions. You can ask the compiler to inline a function using the __inline keyword. Inlining large functions can lead to big increases in code size without much performance improvement.

SUMMARY: Calling Functions Efficiently

- Try to restrict functions to four arguments. This will make them more efficient to call. Use structures to group related arguments and pass structure pointers instead of multiple arguments.

- Define small functions in the same source file and before the functions that call them. The compiler can then optimize the function call or inline the small function.
- Critical functions can be inlined using the __inline keyword.

Pointer Aliasing

Two pointers are said to alias when they point to the same address. If you write to one pointer, it will affect the value you read from the other pointer.

In a function, the compiler often doesn't know which pointers can alias and which pointers can't. The compiler must be very pessimistic and assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency.

Example: The following function increments two timer values by a step amount:

```
void timers_v1(int *timer1, int *timer2, int *step)
{
*timer1 += *step;
*timer2 += *step;
} This
```

compiles to

```
timers_v1
LDR r3,[r0,#0] ; r3 = *timer1
LDR r12,[r2,#0] ; r12 = *step
ADD r3,r3,r12 ; r3 += r12
STR r3,[r0,#0] ; *timer1 = r3
LDR r0,[r1,#0] ; r0 = *timer2
LDR r2,[r2,#0] ; r2 = *step
ADD r0,r0,r2; r0 += r2
STR r0,[r1,#0] ; *timer2 = t0
MOV pc,r14 ; return
```

Note that the compiler loads from step twice. Usually, a compiler optimization called common subexpression elimination would kick in so that *step was only evaluated once, and the value reused for the second occurrence. However, the compiler can't use this optimization here. The pointers timer1 and step might alias one another. In other words, the compiler cannot be sure that the write to timer1 doesn't affect the read from step.

In this case the second value of *step is different from the first and has the value *timer1. This forces the compiler to insert an extra load instruction.

The same problem occurs if you use structure accesses rather than direct pointer access. The following code also compiles inefficiently:

```
typedef struct {int step;} State; typedef
struct {int timer1, timer2;} Timers; void
timers_v2(State *state, Timers *timers)
{
timers->timer1 += state->step; timers->timer2
+= state->step;
```

}

The compiler evaluates state->step twice in case state->step and timers->timer1 are at the same memory address. The fix is easy: Create a new local variable to hold the value of state->step so the compiler only performs a single load.

In the code for timers_v3 we use a local variable step to hold the value of state->step. Now the compiler does not need to worry that state may alias with timers.

```
void timers_v3(State *state, Timers *timers)
{
int step = state->step;
timers->timer1 += step;
timers->timer2 += step; }
```

You must also be careful of other, less obvious situations where aliasing may occur. When you call another function, this function may alter the state of memory and so change the values of any expressions involving memory reads. The compiler will evaluate the expressions again.

Another pitfall is to take the address of a local variable. Once you do this, the variable is referenced by a pointer and so aliasing can occur with other pointers. The compiler is likely to keep reading the variable from the stack in case aliasing occurs. Consider the following example, which reads and then checksums a data packet:

```
int checksum_next_packet(void)
{ int *data; int N, sum=0;
data = get_next_packet(&N);
do {
sum += *(data++);
} while (--N);
return sum; }
```

Here get_next_packet is a function returning the address and size of the next data packet. The previous code compiles to checksum_next_packet

```
STMFD r13!,{r4,r14} ; save r4, lr on the stack
SUB r13,r13,#8 ; create two stacked variables
ADD r0,r13,#4 ; r0 = &N, N stacked
MOV r4,#0 ; sum = 0
BL get_next_packet ; r0 = data checksum_loop
LDR r1,[r0],#4 ; r1 = *(data++)
ADD r4,r1,r4; sum += r1
LDR r1,[r13,#4] ; r1 = N (read from stack)
SUBS r1,r1,#1 ; r1-- & set flags
STR r1,[r13,#4] ;N= r1 (write to stack)
BNE checksum_loop ; if (N!=0) goto loop
MOV r0,r4 ; r0 = sum
ADD r13,r13,#8 ; delete stacked variables
LDMFD r13!,{r4,pc} ; return r0
```

Note how the compiler reads and writes N from the stack for every N--. Once you take the address of N and pass it to get_next_packet, the compiler needs to worry about aliasing because the pointers data and &N may alias. To avoid this, don't take the address of local variables. If you must do this, then copy the value into another local variable before use.

SUMMARY: Avoiding Pointer Aliasing

- Do not rely on the compiler to eliminate common subexpressions involving memory accesses. Instead create new local variables to hold the expression. This ensures the expression is evaluated only once.
- Avoid taking the address of local variables. The variable may be inefficient to access from then on.

MODULE-4

Exception and Interrupt Handling

Exceptions and interrupts are unexpected events which will disrupt the normal flow of execution of instruction. An exception is an unexpected event from within the processor. Interrupt is an unexpected event from outside the process. Whenever an exception or interrupt occurs, the hardware starts executing the code that performs an action in response to the exception.

The following types of action can cause an exception:

- Reset is called by the processor when power is applied. This instruction branches to the initialization code.
- Undefined instruction is used when the processor cannot decode an instruction.
- Software interrupt is called when we execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.
- Prefetch abort 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 is similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions.
- Interrupt request 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.

Exception handling

An exception is any condition that needs to halt the normal sequential execution of instructions. Example for exceptions are: ARM core reset, instruction fetch or memory access failure, an undefined instruction fetch, execution of software interrupt instruction, when an external interrupt has been raised.

Exception handling is the method of processing these exceptions. Most exceptions have an associated software exception handler. Software exception handler is software routine that executes when an exception occurs. The handler first determines the cause of the exception and then services the exception. Servicing takes place either within the handler or by branching to a specific service routine.

The Reset exception is a special case of exception, and it is used to initialize an embedded system.

ARM Processor Exceptions and Modes

Whenever an exception occurs, the core enters a specific mode. The ARM processor modes can be entered manually by changing the cpsr.

When an exception occurs the ARM processor always switches to ARM state. Figure 4.1 shows an exceptions and associated modes.

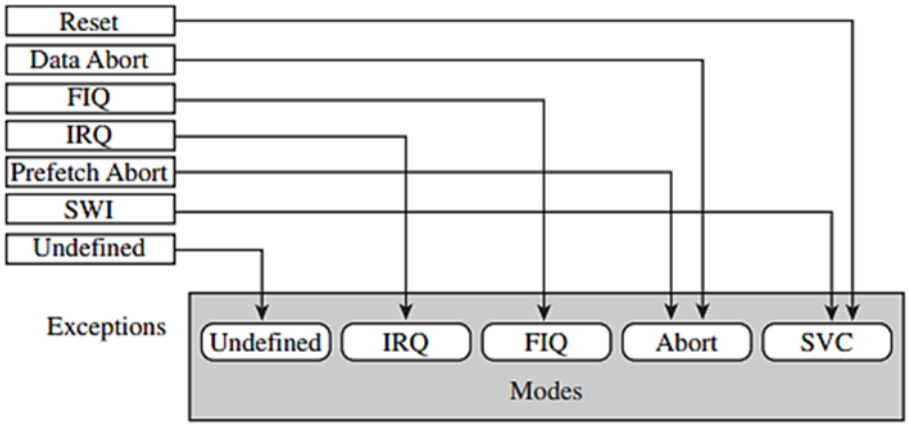


Figure 4.1 Exceptions and associated modes.

The user and system mode are the only two modes that are not entered by an exception. When an exception causes a mode change, the core automatically

- saves the cpsr to the spsr of the exception mode
- saves the pc to the lr of the exception mode
- sets the cpsr to the exception mode
- sets pc to the address of the exception handler

Vector Table

The vector table is a table of addresses that the ARM core branches to when an exception is raised. These addresses contain branch instructions. 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).

Exception	Mode	Vector table offset
Reset	SVC	+0x00
Undefined Instruction	UND	+0x04
Software Interrupt (SWI)	SVC	+0x08
Prefetch Abort	ABT	+0x0c
Data Abort	ABT	+0x10
Not assigned	—	+0x14
IRQ	IRQ	+0x18
FIQ	FIQ	+0x1c

Table 4.2 Vector table and processor modes

The branch instruction can be any of the following forms:

B <address>—This branch instruction provides a branch relative from the pc.

LDR pc, [pc, #offset]—This load register instruction loads the handler address from memory to the pc. This form gives slight delay in branching as it need the extra memory access. But using this form we can branch to any address in memory.

LDR pc, [pc, #-0xff0]—This load register instruction loads a specific interrupt service routine address from address 0xfffff030 to the pc. This specific instruction is used when a vector interrupt controller is present (VIC PL190).

MOV pc, #immediate—This move instruction copies an immediate value into the pc. The address must be an 8-bit immediate rotated right by an even number of bits.

```
0x00000000: 0xe59ffa38  RESET: > ldr  pc, [pc, #reset]
0x00000004: 0xea000502  UNDEF:   b    undInstr
0x00000008: 0xe59ffa38  SWI  :   ldr  pc, [pc, #swi]
0x0000000c: 0xe59ffa38  PABT :   ldr  pc, [pc, #prefetch]
0x00000010: 0xe59ffa38  DABT :   ldr  pc, [pc, #data]
0x00000014: 0xe59ffa38   -   :   ldr  pc, [pc, #notassigned]
0x00000018: 0xe59ffa38  IRQ  :   ldr  pc, [pc, #irq]
0x0000001c: 0xe59ffa38  FIQ  :   ldr  pc, [pc, #fiq]
```

Figure 4.2 Example vector table.

Exception Priorities

Exceptions can occur simultaneously, so the processor has to adopt a priority mechanism. Each exception is dealt with according to the priority level set out in Table 4.3.

Table 4.3 shows the various exceptions that occur on the ARM processor and their associated priority level.

Exceptions	Priority	*I* bit	*F* bit
Reset	1	1	1
Data Abort	2	1	—
Fast Interrupt Request	3	1	1
Interrupt Request	4	1	—
Prefetch Abort	5	1	—
Software Interrupt	6	1	—
Undefined Instruction	6	1	—

Table 4.3 Exception priority levels.

- The Reset exception is the highest priority, and it occurs when power is applied to the processor. The reset handler initializes the system and setting up memory and caches. The reset handler must also set up the stack pointers for all processor modes. When a reset occurs, it takes precedence over all other exceptions.
- The lowest priority level is shared by two exceptions, the Software Interrupt and Undefined Instruction exceptions.
- As shown in the table Certain exceptions also disable interrupts by setting the I or F bits in the cpsr
- Code should be designed such that there are no exceptions or interrupts will occur during the first few instructions of the handler.

- Data Abort exceptions occur when the memory controller or MMU indicates that an invalid memory address has been accessed or when the current code attempts to read or write to memory without the correct access permissions. When Data Abort occurs, it takes precedence over all other exceptions except Reset exception.
- A Fast Interrupt Request (FIQ) exception occurs when an external peripheral sets the FIQ pin to nFIQ. An FIQ exception is the highest priority interrupt. The core disables both IRQ and FIQ exceptions on entry into the FIQ handler. Thus, no external source can interrupt the processor unless the IRQ and/or FIQ exceptions are reenabled by software.
- An Interrupt Request (IRQ) exception occurs when an external peripheral sets the IRQ pin to nIRQ. An IRQ exception is the second highest priority interrupt. The IRQ handler will be entered if neither an FIQ exception nor Data Abort exception occurs. On entry to the IRQ handler, the IRQ exceptions are disabled and should remain disabled until the current interrupt source has been cleared.
- A Prefetch Abort exception occurs when an attempt to fetch an instruction results in a memory fault. This exception is raised when the instruction is in the execute stage of the pipeline and if none of the higher exceptions have been raised. After entering to the handler, IRQ exceptions will be disabled, but the FIQ exceptions will remain unchanged. If FIQ is enabled and an FIQ exception occurs, it can be taken while servicing the Prefetch Abort.
- A Software Interrupt (SWI) exception occurs when the SWI instruction is executed and none of the other higher-priority exceptions have been flagged. On entry to the handler, the cpsr will be set to supervisor mode. If the system uses nested SWI calls, the link register r14 and spsr must be stored away before branching to the nested SWI to avoid possible corruption of the link register and the spsr.
- An Undefined Instruction exception occurs when an instruction not in the ARM or Thumb instruction set reaches the execute stage of the pipeline and none of the other exceptions have been flagged. The ARM processor “asks” the coprocessors if they can handle this as a coprocessor instruction. Since coprocessors follow the pipeline, instruction identification can take place in the execute stage of the core. If none of the coprocessors claims the instruction, an Undefined Instruction exception is raised. Both the SWI instruction and Undefined Instruction have the same level of priority, since they cannot occur at the same time.

Link Register Offsets

When an exception occurs, the link register is set to a specific address based on the current pc. For example, when an IRQ exception is raised, the link register lr points to the last executed instruction plus 8 because of three stage pipeline. Care has to be taken to make sure the exception handler does not corrupt lr because lr is used to return from an exception handler. The IRQ exception is taken only after the current instruction is executed, so the return address has to point to the next instruction, or lr − 4. Table 4.4 provides a list of useful addresses for the different exceptions.

Table 4.4 Useful link-register-based addresses.

Exception	Address	Use
Reset	—	*lr* is not defined on a Reset
Data Abort	*lr* − 8	points to the instruction that caused the Data Abort exception
FIQ	*lr* − 4	return address from the FIQ handler
IRQ	*lr* − 4	return address from the IRQ handler
Prefetch Abort	*lr* − 4	points to the instruction that caused the Prefetch Abort exception
SWI	*lr*	points to the next instruction after the SWI instruction
Undefined Instruction	*lr*	points to the next instruction after the undefined instruction

Example-1: This example shows that a typical method of returning from an IRQ and FIQ handler is to use a SUBS instruction:

handler

```
<handler code>
...
SUBS  pc, r14, #4       ; pc=r14-4
```

Interrupts

There are two types of interrupts available on the ARM processor. The first type of interrupt causes an exception raised by an external peripheral—namely, IRQ and FIQ.
The second type is a specific instruction that causes an exception—the SWI instruction.
Both types suspend the normal flow of a program.

Assigning Interrupts

A system designer can decide which hardware peripheral can produce which interrupt request. This decision can be implemented in hardware or software (or both) and depends upon the embedded system being used.
An interrupt controller unit is used to connects multiple external interrupts to one of the two ARM interrupt requests either IRQ or FIQ. The system designers will use a standard design practice to assigning interrupts.

- Software Interrupts are normally reserved to call privileged operating system routines. For example, an SWI instruction can be used to change a program running in user mode to a privileged mode.
- IRQ Requests are normally assigned for general-purpose interrupts. The IRQ exception has a lower priority and higher interrupt latency than the FIQ exception.
- Fast Interrupt Requests are normally reserved for a single interrupt source that requires a fast response time.
- In an embedded operating system design, the FIQ exception is used for a specific application and the IRQ exception are used for more general operating system activities.

Interrupt Latency

It is the time interval, from an external interrupt request signal being raised to the first fetch of an instruction of a specific interrupt service routine (ISR).

Interrupt latency depends on a combination of hardware and software. System designer must balance the system design to handle multiple simultaneous interrupt sources and minimize interrupt latency.
If the interrupts are not handled in a timely manner, then the system will exhibit slow response times. Software handlers have two main methods to minimize interrupt latency.
1) Nested interrupt handler,
2) Prioritization.

Nested interrupt handler

Nested interrupt handler allows other interrupts to occur even when it is currently servicing an existing interrupt. This is achieved by reenabling the interrupts as soon as the interrupt source has been serviced but before the interrupt handling is complete. Once a nested interrupt has been serviced, then control is relinquished to the original interrupt service routine.
Fig 4.3 shows the three-level nested interrupt,

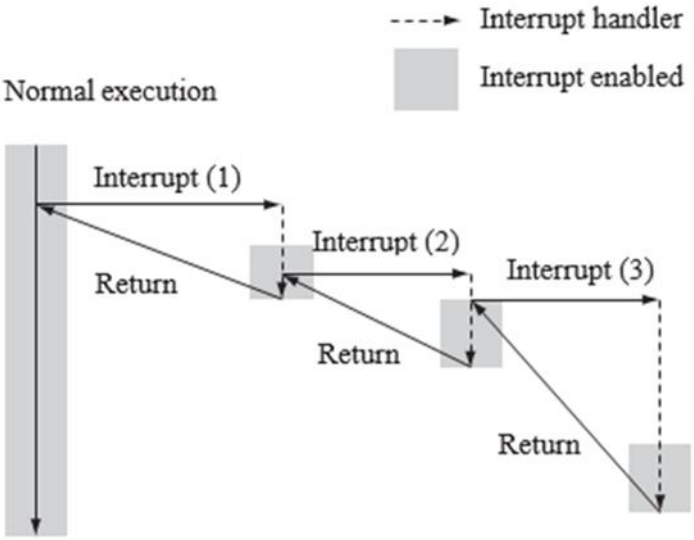


Figure 4.3 A three-level nested interrupt.

Prioritization

We can program the interrupt controller to ignore interrupts of the same or lower priority than the interrupt we are handling presently, so only a higher-priority task can interrupt our handler. We then re-enable the interrupts. The processor spends time in the lower-priority interrupts until a higher-priority interrupt occurs. Therefore higher-priority interrupts have a lower average interrupt latency than the lower-priority interrupts. It reduces latency by speeding up the completion time on the critical time-sensitive interrupts.

IRQ and FIQ Exceptions

IRQ and FIQ exceptions only occur when a specific interrupt mask is cleared in the cpsr.
The ARM processor will continue executing the current instruction in the execution stage of the pipeline before handling the interrupt. An IRQ or FIQ exception causes the processor hardware to go through a standard procedure listed below,
1) The processor changes to a specific interrupt request mode, which being raised.
2) The previous mode's cpsr is saved into the spsr of the new interrupt request mode.

3) The pc is saved in the lr of the new interrupt request mode.
4) Interrupt/s are disabled—either the IRQ or both IRQ and FIQ exceptions are disabled in the cpsr. This immediately stops another interrupt request of the same type being raised.
5) The processor branches to a specific entry in the vector table.

Example 4.5: what happens when an IRQ exception is raised when the processor is in user mode?

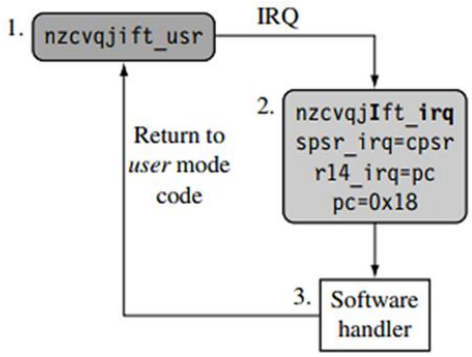


Figure 4.4 Interrupt Request (IRQ).

i. The processor starts in state 1. In this mode both the IRQ and FIQ exception bits in the cpsr are enabled.
ii. When an IRQ occurs the processor moves into state 2.->
a) This transition automatically sets the IRQ bit to one, disabling any further IRQ exceptions,
b) The FIQ exception, remains enabled because FIQ has a higher priority and does not get disabled when a low-priority IRQ exception is raised,
c) The cpsr processor mode changes to IRQ mode,
d) The user mode cpsr is automatically copied into spsr_irq,
e) Register r14_irq is assigned the value of the pc when the interrupt was raised,
f) The pc is then set to the IRQ entry +0x18 in the vector table.
iii. In state 3 the software handler takes over and calls the appropriate interrupt service routine to service the source of the interrupt. After completion, the processor mode reverts back to the original user mode code in state 1.

Example 4.6 what happens when an FIQ exception is raised when the processor is in user mode?
Figure 4.5 shows an example of an FIQ exception.

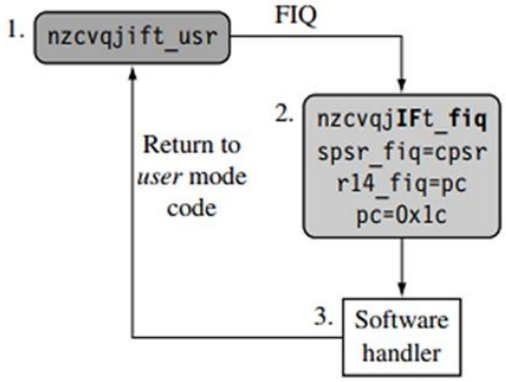


Figure 4.5 Fast Interrupt Request (FIQ).

i) The processor starts in state 1. In this mode both the IRQ and FIQ exception bits in the cpsr are enabled.
ii) When an FIQ occurs the processor moves into state 2.->
a) This transition automatically sets the IRQ bit and FIQ to one, disabling both IRQ and FIQ exceptions,
b) The cpsr processor mode changes to FIQ mode,
c) The user mode cpsr is automatically copied into spsr_fiq,
d) Register r14_fiq is assigned the value of the pc when the interrupt was raised,
e) The pc is then set to the FIQ entry +0x1c in the vector table.
iii) In state 3 the software handler takes over and calls the appropriate interrupt service routine to service the source of the interrupt. After completion, the processor mode reverts back to the original user mode code in state 1.
iv) When processor changes from user mode to FIQ mode, there is no requirement to save registers r8 to r12 since these registers are banked in FIQ mode. These registers can be used to hold temporary data, such as buffer pointers or counters. This makes FIQ ideal for servicing a single-source, high priority, low-latency interrupt.

Enabling and Disabling FIQ and IRQ Exceptions

The ARM processor core has a simple procedure to manually enable and disable interrupts by modifying the cpsr when the processor is in a privileged mode.
The procedure uses three ARM instructions.
1)The instruction MRS copies the contents of the cpsr into register r1.
2)The instruction BIC clears the IRQ or FIQ mask bit.
3) The instruction MSR then copies the updated contents in register r1 back into the cpsr, to enable the interrupt request.
Table 4.5 shows how IRQ and FIQ interrupts are enabled.
The postfix _c identifies that the bit field being updated is the control field bit [7:0] of the cpsr.

cpsr value	IRQ	FIQ
Pre	*nzcvqj**I**Ft_SVC*	*nzcvqj**I**Ft_SVC*
Code	`enable_irq`	`enable_fiq`
	`MRS r1, cpsr`	`MRS r1, cpsr`
	`BIC r1, r1, #0x80`	`BIC r1, r1, #0x40`
	`MSR cpsr_c, r1`	`MSR cpsr_c, r1`
Post	*nzcvqj**i**Ft_SVC*	*nzcvqj**I**ft_SVC*

Table 4.5 Enabling an interrupt.

Table 4.6 shows procedure to disable or mask an interrupt request.

cpsr	IRQ	**FIQ**
Pre	*nzcvqj**i**ft_SVC*	*nzcvqj**i**ft_SVC*
Code	`disable_irq`	`disable_fiq`
	`MRS r1, cpsr`	`MRS r1, cpsr`
	`ORR r1, r1, #0x80`	`ORR r1, r1, #0x40`
	`MSR cpsr_c, r1`	`MSR cpsr_c, r1`
Post	*nzcvqj**I**ft_SVC*	*nzcvqj**i**Ft_SVC*

Table 4.6 Disabling an interrupt.

To enable and disable both the IRQ and FIQ exceptions, the immediate value on the data processing BIC or ORR instruction has to be changed to 0xc0.
The interrupt request is either enabled or disabled only once the MSR instruction has completed the execution stage of the pipeline. Interrupts can still be raised or masked prior to the MSR completing this stage.

Basic Interrupt Stack Design and Implementation

Exceptions handlers use the stacks to save the register contents. Each mode has dedicated register containing the stack pointer. The design of the exception stacks depends upon these factors:

- Operating system requirements—Each operating system has its own requirements for stack design.
- Target hardware—The target hardware provides a physical limit to the size and positioning of the stack in memory

Two design decisions need to be made for the stacks:

- The location: which determines where in the memory map the stack begins. Most ARM-based systems are designed with a stack that descends downwards, with the top of the stack at a high memory address.
- Stack size: depends upon the type of handler, nested or non-nested. A nested interrupt handler requires more memory space since the stack will grow with the number of nested interrupts.

Stack Overflow

When the stack extends beyond the allocated memory. It causes instability in embedded systems. There are software techniques that identify overflow and that allow corrective measures to take place to repair the stack before irreparable memory corruption occurs.
The two main methods are
(1) use memory protection
(2) call a stack check function at the start of each routine.
The IRQ mode stack has to be set up during the initialization code for the system. The stack size is reserved in the initial stages of boot-up. Figure 4.6 shows two memory layouts in a linear address space.

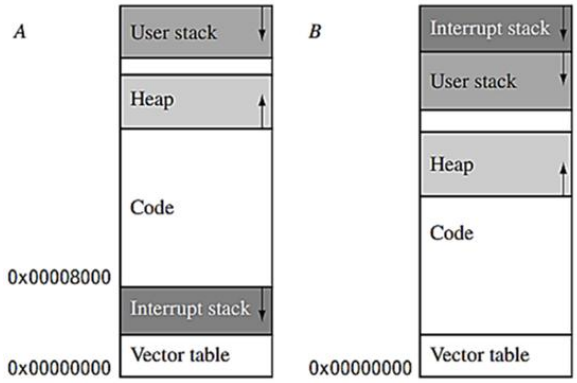


Figure 4.6 Memory layouts.

The first layout, A, shows a traditional stack layout with the interrupt stack stored underneath the code segment.
The second layout, B, shows the interrupt stack at the top of the memory above the user stack. The main advantage of layout B over A is that Layout B does not corrupt the vector table when a stack overflow occurs, and so the system has a chance to correct itself when an overflow has been identified.
For each processor mode a stack has to be set up. This is carried out every
time the processor is reset. Figure 4.7 shows an implementation of stack using layout A.

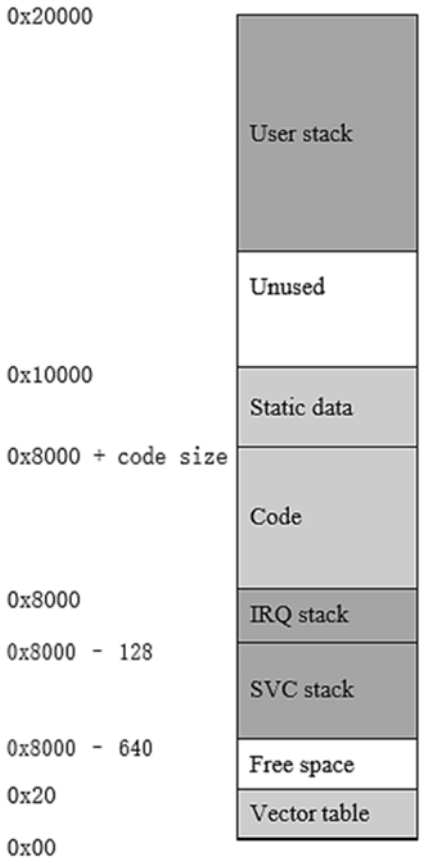


Fig.4.7 Example implementation using layout A

There is an advantage using separate stacks for each mode rather than using a single stack. Errant tasks can be debugged and isolated from the rest of the system.
Each mode stack must be set up. Here is an example to set up three different stacks when the processor core comes out of reset.
Initialization code starts by setting up the stack registers for each processor mode. The stack register r13 is one of the registers that is always banked when a mode change occurs.
A set of defines are declared that map the memory region names with an absolute address.

Example, the User stack is given the label USR_Stack and is set to address 0x20000. The Supervisor stack is set to an address that is 128 bytes below the IRQ stack.

```
USR_Stack   EQU     0x20000
IRQ_Stack   EQU     0x8000
SVC_Stack   EQU     IRQ_Stack-128
```

A set of defines that map each processor mode with a particular mode bit pattern. These labels can then be used to set the cpsr to a new mode.

```
Usr32md         EQU 0x10            ; User mode
FIQ32md         EQU 0x11            ; FIQ mode
IRQ32md         EQU 0x12            ; IRQ mode
SVC32md         EQU 0x13            ; Supervisor mode
Abt32md         EQU 0x17            ; Abort mode
Und32md         EQU 0x1b            ; Undefined instruction mode
Sys32md         EQU 0x1f            ; System mode
```