Microcontrollers – BCS402

Course Learning Objectives:

This course will enable students to:

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

Course Outcomes:

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.

Syllabus

Module-1

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.

Module-2

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

Syllabus

Module-3

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

Module-4

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.

Syllabus

Module-5

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.

Text books

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

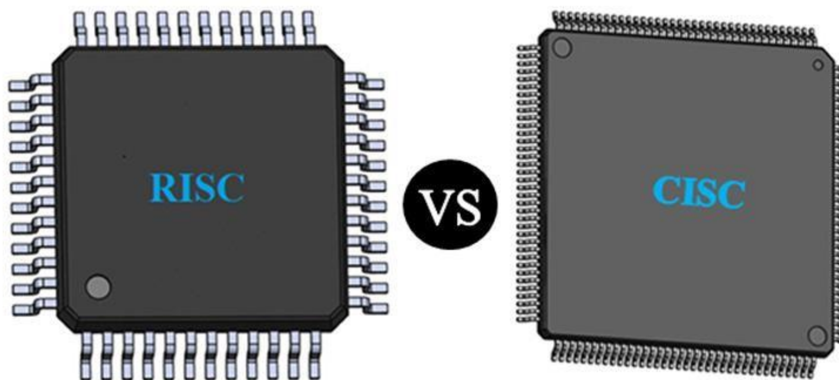
Reference Books:

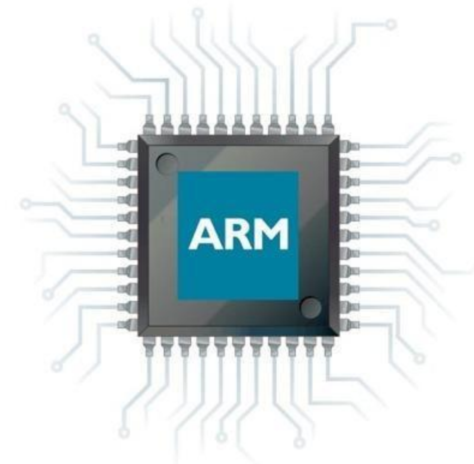
1. Raghunandan..G.H, Microcontroller (ARM) and Embedded System, Cengage learning Publication,2019
2. The Insider‟s Guide to the ARM7 Based Microcontrollers, Hitex Ltd.,1st edition, 2005.
3. Steve Furber, ARM System-on-Chip Architecture, Second Edition, Pearson, 2015.
4. Raj Kamal, Embedded System, Tata McGraw-Hill Publishers, 2nd Edition, 2008.

MODULE-I

Chapter-1: ARM Embedded Systems

- What is Micro-Processor?
- What is Micro-controller?
- What is ARM?
- What is CISC?
- What is RISC?

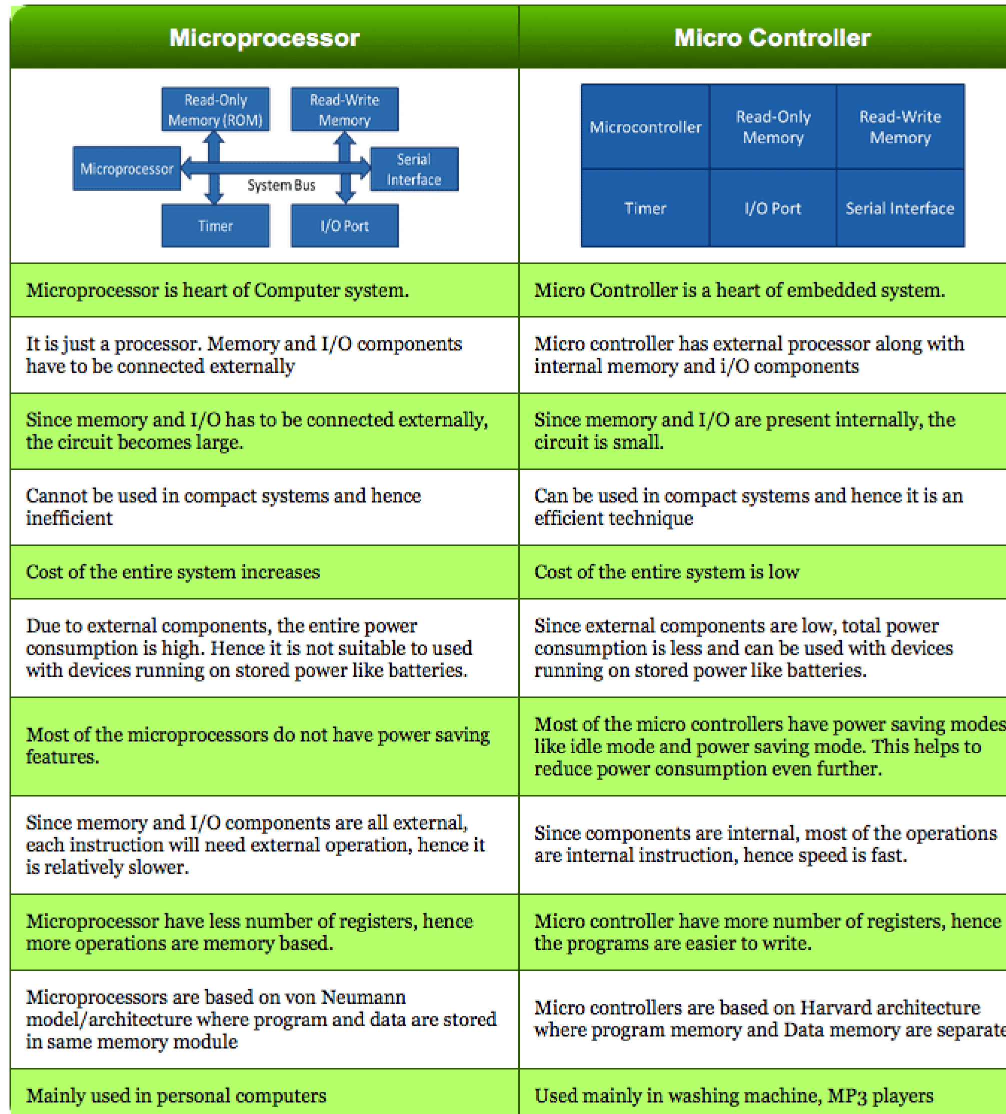




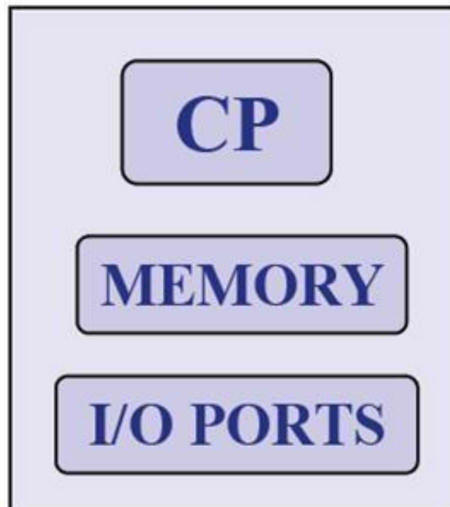
CISC vs RISC

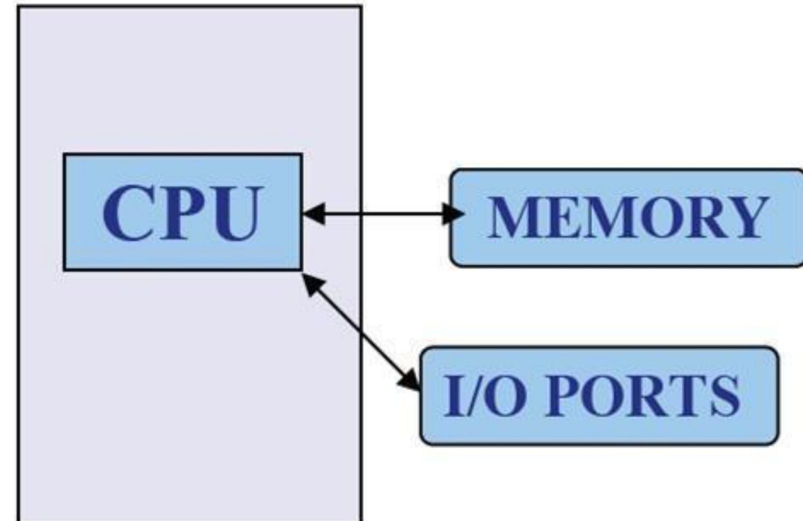
Instruction Set

Normal instructions				Compressed	
00000	SUB	10000	CMP	000	SUB
00001	AND	10001	TEST	001	AND
00010	ADD	10010	LW	010	ADD
00011	OR	10011	SW	011	CMP
00100	XOR	10100	LH	100	LW
00101	LSR	10101	SH	101	SW
00110	LSL	10110	LB	110	LDI
00111	ASR	10111	SB	111	MOV
01000	BREV	11000	LDI	Reserved for FPU	
01001	LDILO	11001			
01010	MPYUHI	Special Insn		11010	FPADD
01011	MPYSHI			11011	FPSUB
01100	MPY	11100	BREAK	11100	FPMPY
01101	MOV	11101	LOCK	11101	FPDIV
01110	DIVU	11110	SIM	11110	FPI2F
01111	DIVS	11111	NOOP	11111	FPF2I



Microprocessor	Micro Controller
Microprocessor is heart of Computer system.	Micro Controller is a heart of embedded system.
It is just a processor. Memory and I/O components have to be connected externally	Micro controller has external processor along with internal memory and i/O components
Since memory and I/O has to be connected externally, the circuit becomes large.	Since memory and I/O are present internally, the circuit is small.
Cannot be used in compact systems and hence inefficient	Can be used in compact systems and hence it is an efficient technique
Cost of the entire system increases	Cost of the entire system is low
Due to external components, the entire power consumption is high. Hence it is not suitable to used with devices running on stored power like batteries.	Since external components are low, total power consumption is less and can be used with devices running on stored power like batteries.
Most of the microprocessors do not have power saving features.	Most of the micro controllers have power saving modes like idle mode and power saving mode. This helps to reduce power consumption even further.
Since memory and I/O components are all external, each instruction will need external operation, hence it is relatively slower.	Since components are internal, most of the operations are internal instruction, hence speed is fast.
Microprocessor have less number of registers, hence more operations are memory based.	Micro controller have more number of registers, hence the programs are easier to write.
Microprocessors are based on von Neumann model/architecture where program and data are stored in same memory module	Micro controllers are based on Harvard architecture where program memory and Data memory are separate
Mainly used in personal computers	Used mainly in washing machine, MP3 players


Microcontroller
CP
MEMORY
I/O PORTS


Microprocessor
CPU
MEMORY
I/O PORTS

ARM Basics

- ARM: Advance RISC Machine
- ARM was established as a joint venture between Acorn, Apple and VLSI.
- ARM is the industry's leading provider of 16/32-bit embedded RISC microprocessor solutions.
- The company licenses its high-performance, low-cost, power designs to leading international electronics companies.
- ARM provides comprehensive support required in developing a complete system.

ARM Embedded Systems

- The ARM processor is a key component of many successful 32-bit embedded systems.
- ARM cores are widely used in mobile phones, handheld devices, and a many other everyday portable consumer devices.
- The first ARM prototype name ARM1 was designed in 1985 and continues to improve through constant technical innovation leading to ARM2, ARM3, ARM4, ARM5, ARM6, ARM7, ARM8, ARM9… ARM Cortex.
- Now, ARM sales is 7 times more than the worlds population.
- The ARM core is not a single core, but a whole family of designs sharing similar design principles and a common instruction set.
- Example, ARM7TDMI is one of the most successful ARM core widely used in majority of the devices.
- It provides up to 120 Dhrystone MIPS, high code density and low power consumption, making it ideal for mobile embedded devices.

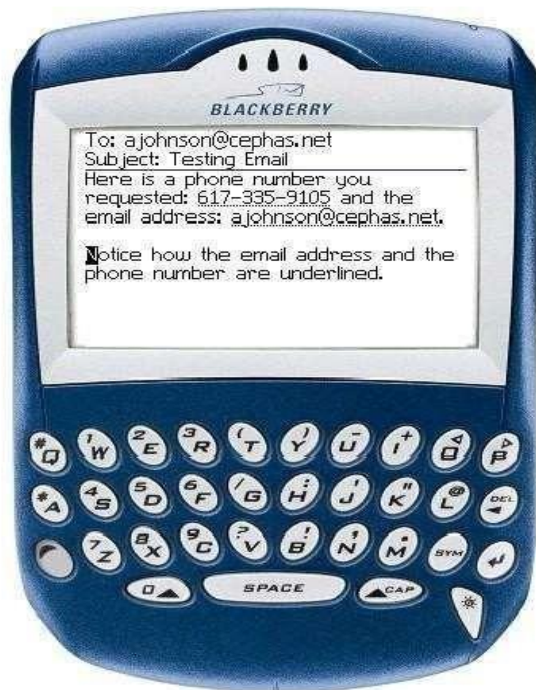
Cont.....

- ARM has adopted the RICS design philosophy.
- ARM Holdings is a technology company headquartered in Cambridge, England, UK. The company is best known for its processors, although it also designs, licenses and sells software development tools under the RealView and KEIL brands, systems and platforms, system-on-a-chip infrastructure and software.
- ARM do not make ICs !!! ARM grant license of core to different silicon vendors like ATMEL, NXP, Cirrus logic etc.. These companies make IC'S. Examples are: LPC2148 from NXP, AT91RM9200 from ATMEL.
- Mainly ARM processors are used in Handheld devices, Robotics, Automation, Consumer Electronics.
- ARM processors are available for almost every domain.

ARM Based Products

Apple iPhone ARM11

Motorola Z8 Smart phone ARM11



Blackberry ARM11

Nokia E90
Communicator ARM11

iPOD ARM7TDMI

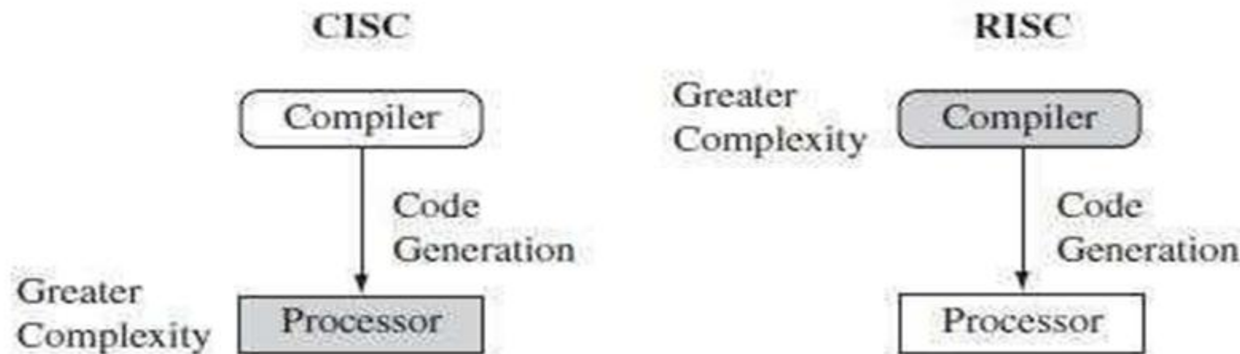
Low cost Multimedia player ARM7TDMI

Lego Mindstrome Robot ARM7

Paison Series game consoles ARM7TDMI

The RISC design philosophy

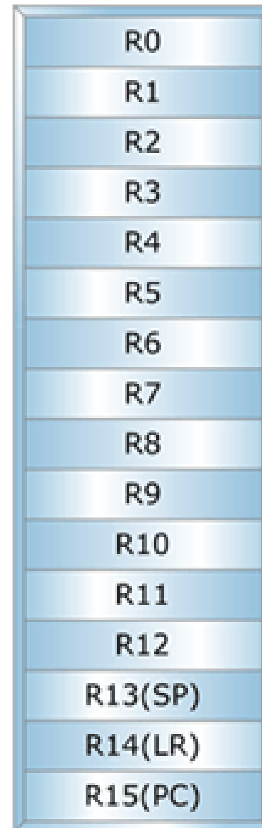
- RICS - Reduced Instruction Set Computing
- RISC is a design philosophy aims to provide few simple but powerful instructions that execute within a single clock cycle.
- The ARM processors uses a RISC architecture.
- The RISC philosophy aims to reduce the complexity of instructions performed by the hardware because it is easier to provide greater flexibility and intelligence in software rather than hardware.
- RISC design places greater burden/load on the compiler.
- Traditional complex instruction set computer (CISC) relies more on the hardware (processor) for instruction functionality, as CISC instructions are more complicated.

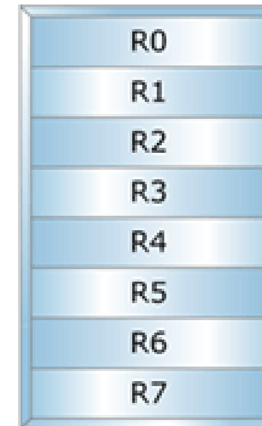


CISC vs. RISC. CISC emphasizes hardware complexity. RISC emphasizes compiler complexity.

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 processing operations.
- In contrast, CISC processors have dedicated specific purposes.





ARM

Thumb

Load-store architecture

- Processor operates only on data held in registers.

- Separate load and store instructions transfer data between the register bank and external memory.

- Memory accesses are costly, so separating memory accesses from data processing provides an advantage because you can use data items held in the register bank multiple times without needing multiple memory accesses.

- In contrast, with a CISC design the data processing operations can act on memory directly.

- These RICS design rules allow a RISC processor to be simpler, and thus the core can operate at higher clock frequencies.
- In contrast, traditional CISC processors are more complex and operate at lower clock frequencies.
- Over the course of two decades, however, the distinction between RISC and CISC has blurred as CISC processors have implemented more RISC concepts.

RISC : Reduced Instruction Set Computing

- Lesser number of fixed length instructions.
- Fewer addressing modes.
- Increased pipelining and increased execution speed.
- Orthogonal instruction set (Allows each instruction to operate on any register and use any addressing mode).
- Operations are performed on registers only. Memory operations are load and store. (Load Store Architecture).
- A large number of register are available.
- Programmer needs to write more code to execute a task since the instructions are simpler once.
- Instruction take one clock cycle. The average clock cycle per instruction (CPI) is 1.5.

The ARM Design Philosophy

RICS features accepted by ARM

- A large set of uniform register file
- A load store architecture
- Uniform and fixed length (32-bit) instruction fields
- Three address instruction formats

The ARM Design Philosophy

RICS features rejected by ARM

- Register windows – because they occupy large chip area.
- Delayed Branches
- Single cycle execution of all instructions.

Instruction Set for Embedded System

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

- **Variable cycle** execution for certain instruction
- **Barrel shifter:** Inline barrel shifter for more complex instructions.
- **Conditional execution** facility in majority of the instructions.
- **Enhanced instruction** – Enhanced DSP instructions were added to the standard ARM instruction.
- **Thumb state (16 bit- instruction set)** – improves the code density by 30% to 35% over 32-bit fixed length instructions

Embedded System Hardware

- Embedded systems are designed to 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 designed to provide higher efficiency and is designed for future extension and expansion.

An example of an ARM-based Embedded Device, a Microcontroller

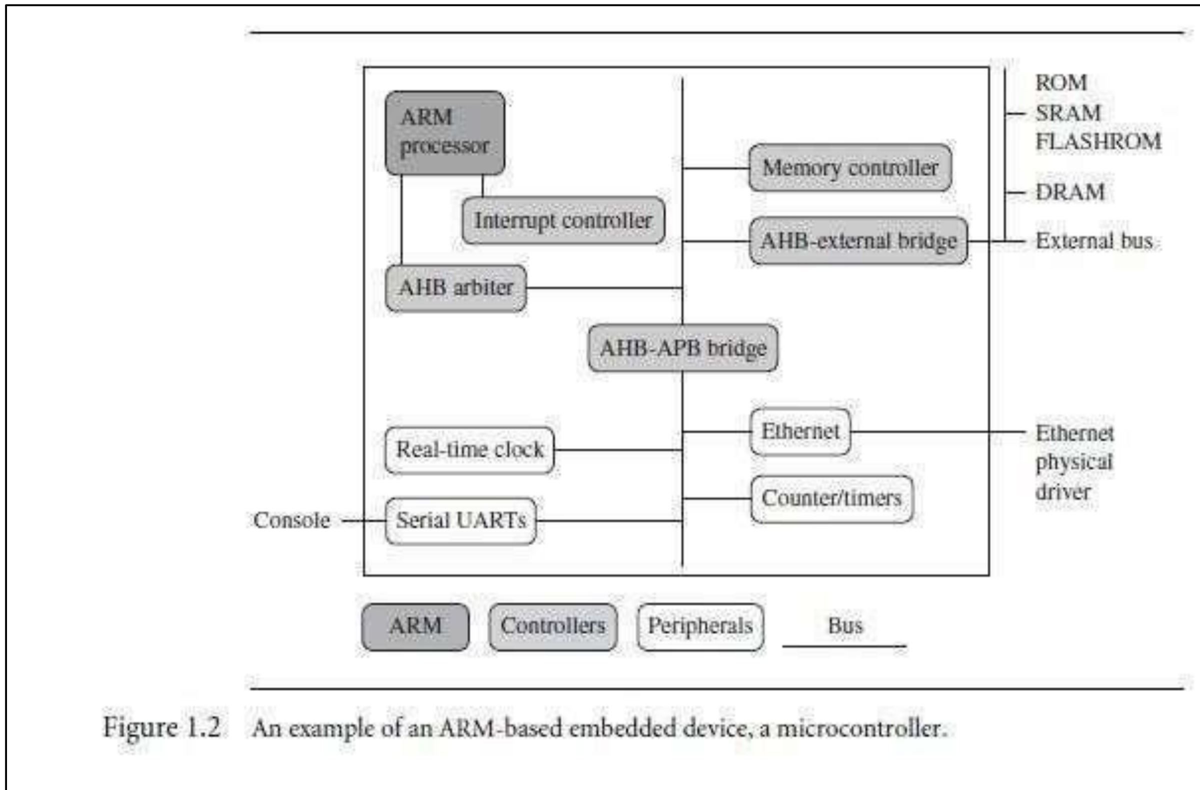


Figure 1.2 An example of an ARM-based embedded device, a microcontroller.

Boxes represent- feature or function

Lines represent - Busses connecting the devices

Divided into 4 Major components
- -ARM processor
- -Controllers
- -Peripherals
- -Bus

AHB - ARM High performance Bus
APB - ARM Peripheral Bus

ARM Processor

- **Controls** the embedded device.

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

- An ARM processor comprises

  -a core (the execution engine that processes instructions and manipulates data)

  plus

  -the **surrounding components** that interface it with a bus (include memory management and caches.)

Controllers

❖ Helps **coordinate important functional blocks** of the system.

❖ Two commonly found controllers are
-**interrupt controller** and
-**memory controllers**.

Peripherals

❖ Provides **input-output capability** external to the chip.
-includes serial I/O, Parallel I/O, Timers counters and clock circuits

❖ Responsible for the uniqueness of the embedded device.

Bus

❖ Supports communicate between different parts of the device. AHB, APB

ARM Bus Technology

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

ARM Bus Technology Cont..

❖There are **two different classes of devices** attached to the ARM on chip bus.

- ***A bus master:*** *a logical device* ***capable of initiating*** *a with another data transfer device across the same bus*. The ARM processor core is the bus master.
- **Bus slaves:** *logical devices* ***capable only of responding*** *to a transfer request from a bus master device*. All peripherals are bus slaves

❖On chip ARM bus has **two architecture levels.**

- **Physical level** *covers the* ***electrical characteristics*** *and bus width (16, 32, or 64 bits).*
- **Protocol level** *covers the* ***logical rules*** *that govern the communication between the processor and a peripheral.*

❖ARM is primarily a **design company**. It does not/rarely implements the electrical characteristics of the bus, but it specifies the bus protocol.

AMBA Bus Protocol

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

AMBA Bus Protocol Cont….

- AHB provides **higher data throughput** than ASB.
  - Because AHB is based on a **centralized multiplexed bus scheme**.
  - ASB bidirectional bus design.
- ARM has introduced **two variations on the AHB bus**:
  - **Multi-layer AHB and**
  - **AHB-Lite.**
- The original AHB, was allowing a single bus master to be active on the bus at any time, the Multi-layer AHB bus **allows multiple active bus masters.**
- AHB-Lite is a subset of the AHB bus and it is **limited to a single bus master.**
- AHB-Lite was developed for **designs that do not require the full features** of the standard AHB bus.

Memory

- An embedded system requires some form of memory to **store and execute code.**
- The memory should be **fast, large and inexpensive**.
- Unfortunately it is **impossible to meet all the 3 requirements.**
- The common solution is to have the **memory hierarchy.**

Memory Width

- The memory width is the **number of bits the memory returns** on each access which is typically 8, 16, 32, or 64 bits.

- The memory width has a direct **effect on the overall performance** of the system.

- If you have a system using **32-bit ARM instructions** and **16-bit- wide memory chips**, then the Processor will have to make **two memory fetches per instruction.**

- This **reduces the system performance**, but the **benefit is that 16-bit memory is less expensive.**

- If the core executes **16-bit Thumb instructions**, it will achieve **better performance with a 16-bit memory.**

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

Table 1.1 Fetching instructions from memory.

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

Memory Types

- There are many different types of memory
  - ROM
  - Flash ROM
  - Dynamic RAM
  - Static RAM
  - SDRAM – Synchronous Dynamic RAM

ROM- Read Only Memory

- Least flexible of all memory types because it contains an image that is **permanently stored** and **cannot be reprogrammed**.
- Many devices use a **ROM to hold boot code.**

Flash ROM

❖ Flash ROM can be **read as well as written**, but it is **slow to write** so it is not used for holding dynamic data.

❖ It is mainly used for holding the **device firmware** or **storing long-term data** that needs to be preserved after power is off.

❖ The erasing and writing of flash ROM are completely software controlled with no additional hardware circuitry required.

Dynamic Random Access Memory -DRAM

❖ The most **commonly used** RAM for devices.

❖ It has the **lowest cost per megabyte** compared with other types of RAM.

❖ DRAM is *dynamic*—it needs to have its **storage cells refreshed** and given a **new electronic charge every few milliseconds**, so you need to set up a DRAM controller before using the memory.

Static Random Access Memory -SRAM

❖ **Faster** than the more traditional DRAM

❖ 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.

Synchronous Dynamic Random Access Memory - SDRAM

❖ SDRAM is **synchronized with the processor clock speed.**

❖ SDRAM run at much **higher clock speed** than the conventional DRAMs.

Peripherals

- Embedded systems **communicate with the Outside world** via peripheral device.

- A peripheral device performs **input and output functions** for the chip by connecting to other **devices or sensors that are off- chip.**

- Peripherals range from
  - -a simple **serial communication** device to a
  - -more **complex 802.11 wireless device**.

- Two important types of controllers are
  - - **memory controllers** and
  - - **interrupt controllers.**

Memory Controller

❖ Memory controller provides the following functionalities.

-Connects different types of **memories to the processor bus**.

- On power up the memory controller **activates certain memory devices to execute initialization code -ROM**.

-It configures the **memory timings** and the **refresh rate** of the DRAM before it is accessed.

Interrupt Controllers

❖ When a peripheral or device requires processor attention, it sends an interrupt to the processor.

❖ There are two types of interrupt controller in the ARM processor:

- the **standard interrupt controller (SIC)**
- and the **vector interrupt controller (VIC).**

Standard Interrupt Controller (SIC)

- It 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 interrupt signals.
- The interrupt handler **determines which device requires servicing** by reading a **device bitmap register** in the interrupt controller.

Vectored Interrupt Controller (VIC)

- Provides **more functionality** than the SIC.
- Along with the masking functionality it also provides the functionality to **prioritize the interrupts.**

Embedded System Software

- An embedded system needs **software to drive** it.
- Figure shows **four typical software** components required to control an embedded device.

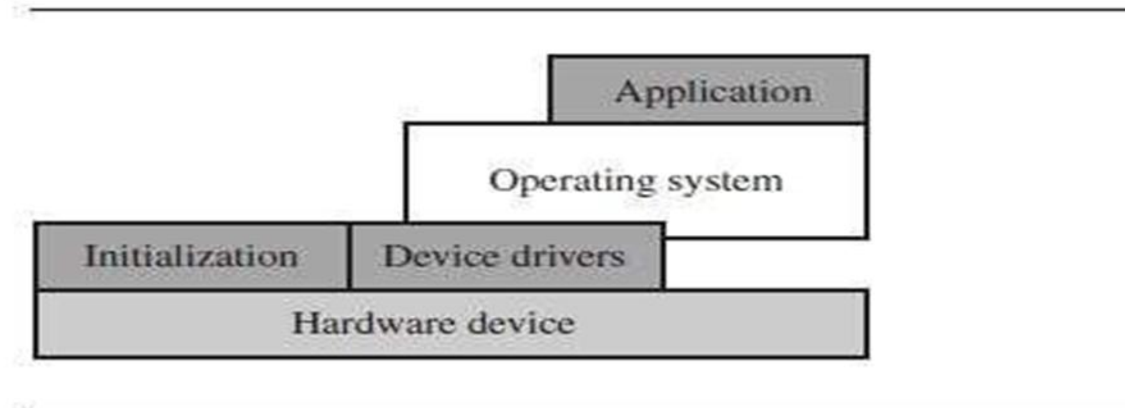


Figure 1.4 Software abstraction layers executing on hardware.

Initialization (Boot) Code

❖ The initialization code is the **first code executed** on the board when the device **is powered on.**

❖ It sets up the **minimum parts of the board** before handing control over to the operating system.

❖ Boot code is present **inside the ROM** and is responsible for **loading the OS to the RAM.**

❖ The initialization code handles **3 administrative tasks prior** to handing control over to an operating system image.

❖ These administrative tasks can be grouped into three phases:

- **initial hardware configuration**,
- **diagnostics**, and
- **booting**.

Initial Hardware Configuration

❖ Although the device comes up in a **standard configuration**, this initial hardware 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 below Example

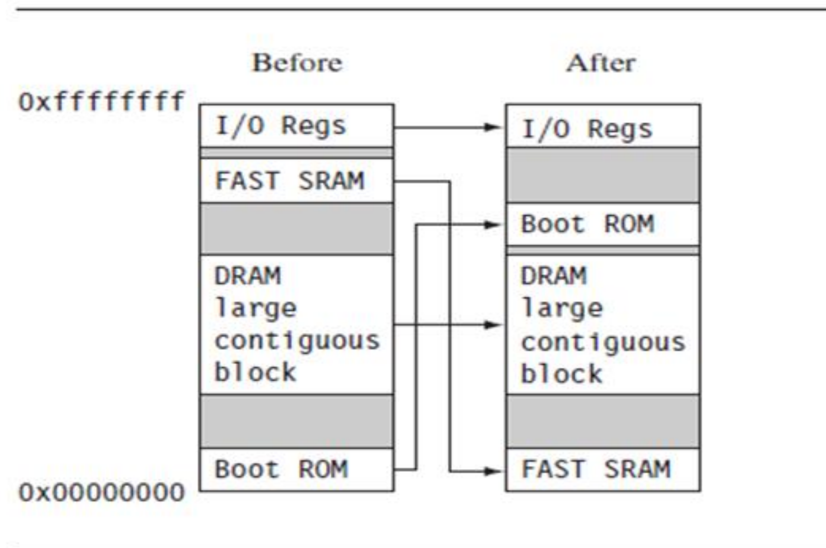


Figure 1.5 Memory remapping.

Diagnostics

- Diagnostics is a program embedded in the initialization code.

- Diagnostic code tests the system by checking if the all the hardware components is in working condition.

- It also tracks down standard system-related issues.

- The primary purpose of diagnostic code is fault identification and isolation.

Booting

❖ Booting involves loading an OS image to RAM and handing control over to that image.

❖ The boot process 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 OS image.

❖ Sometimes, to reduce the image size, an image is compressed.

❖ The image is then decompressed either when it is loaded or when control is handed over to it.

Operating System

❖The Initialization process prepares the hardware for an Operating system to take control.

❖ARM Processor supports over 50 operating systems.

❖Operating system can be divided into two main categories.

- Real Time Operating System – RTOS
- Platform Operating System –POS

❖RTOS used in embedded devices and they don't use the secondary storage and POS used in the general purpose computer systems and they use secondary storage

❖ RTOS is classified into.

-**Hard Real Time Operating System -** Used in hard real time applications which requires the guaranteed response time.

-**Soft Real Time Operating System -** Used in soft real time applications which does not require guaranteed response time and the performance gracefully reduces when the time overruns

Applications

- ARM processors are found in numerous domains, including networking, automotive, mobile and consumer devices, mass storage, and imaging devices.

- Within each domain ARM processors can be found in multiple applications.

For example, the ARM processor is found in networking applications like home gateways, modems for high-speed Internet communication, and wireless communication.

- The mobile device domain is the largest application area for ARM processors because of mobile phones.

- ARM processors are also found in mass storage devices domains such as hard drives, products such as inkjet printers

ARM7TDMI Processor

- The ARM7TDMI core is a member of the ARM family of general-purpose 32-bit microprocessors.

– T: capable of executing Thumb instruction set

–D: Featuring with IEEE Std. 1149.1 JTAG boundary-scan debugging interface.

–M: Featuring with a Multiplier-And-Accumulate (MAC) unit for DSP power applications.

– I: Featuring with the support of embedded In-Circuit Emulator.

- The ARM family offers high performance for very low consumption, and small size.

- The ARM architecture is based on Reduced Instruction Set Computer (RISC)

SUMMARY

❖ It allows **variable cycle execution** on certain instructions to save power, area, and code size.

❖ It adds a **barrel shifter** to expand the capability of certain instructions.

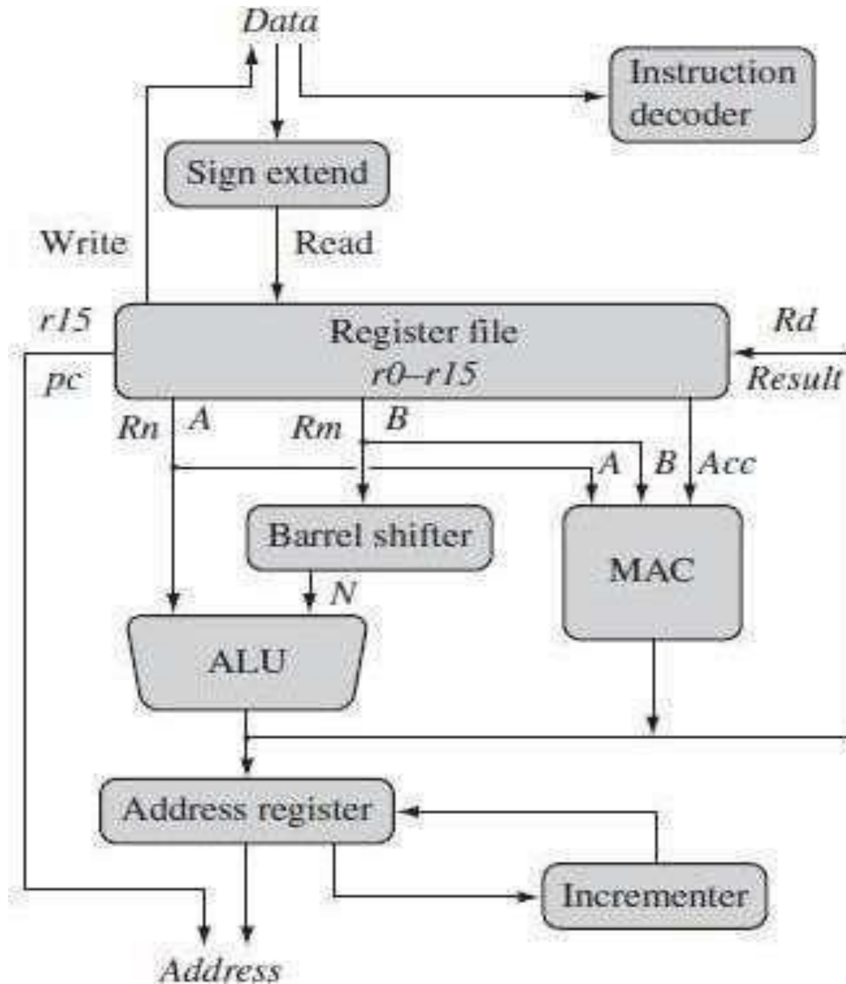
❖ It uses the **Thumb 16-bit** instruction set to improve code density.

❖ It improves **code density and performance** by conditionally executing instructions.

❖ It includes **enhanced instructions** to perform **digital signal processing** type functions.

Chapter-2: ARM Processor Fundamentals

ARM core dataflow model



- **Instruction Fetch:** Fetches the instruction form the memory
- **Instruction Decoder:** Decodes the instruction and identifies the **opcode of the instruction.**

- **Sign extend hardware:** 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.

- **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.*

- **Incrementor:** 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.

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 2.2 shows the active registers available in *user mode—a protected mode normally*

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

cpsr
-

Figure 2.2 Registers available in *user* mode.

- The processor can operate **in seven different modes**
- 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.

❖ **Register *r13*** *is traditionally used as the stack pointer (sp) and stores the head of the stack* in the current processor mode.

❖ **Register *r14*** *is called the link register (lr) and is where the core puts the return address* whenever it calls a subroutine.

❖ **Register *r15*** *is the program counter (pc) and contains the address of the next instruction* to be fetched by the processor.

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

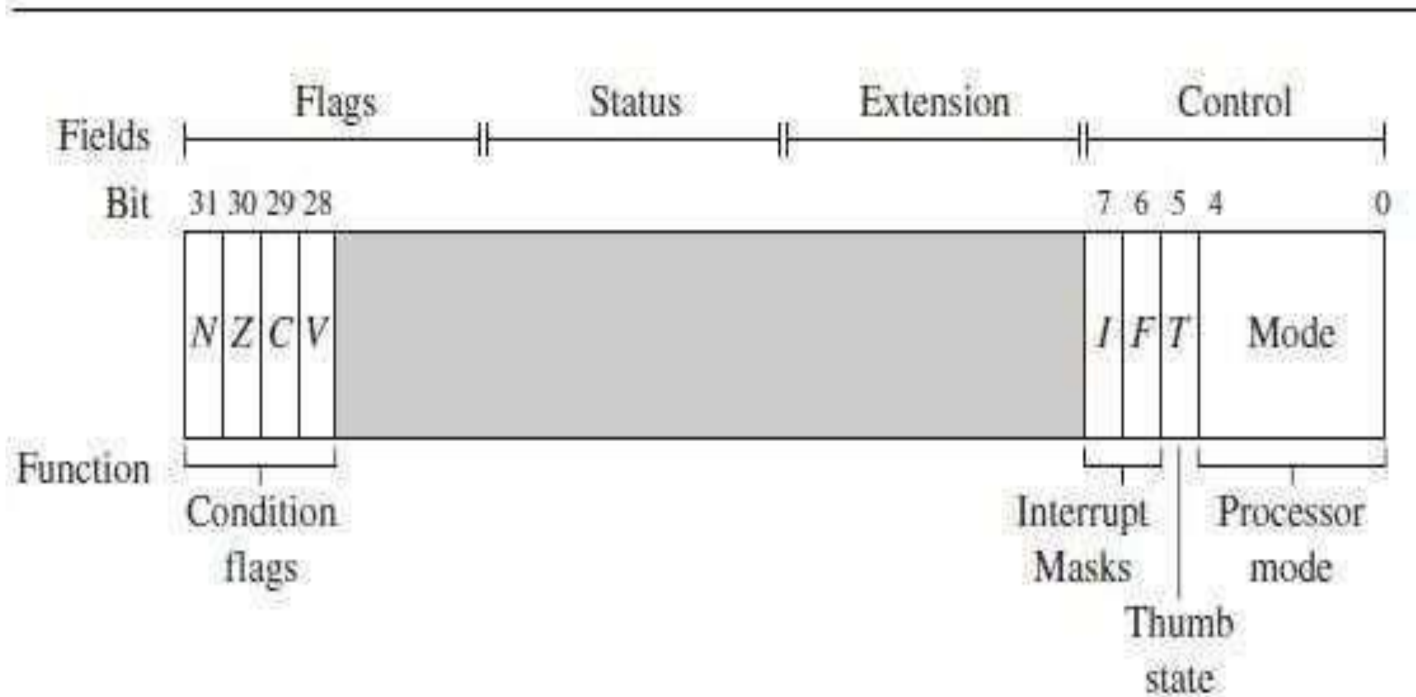
- **Dangerous to use *r13 as a general register when the processor* is running any form of operating system because operating systems often assume that *r13* always points to a valid stack frame.**

- 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).

Current Program Status Register- CSPR



- 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.
- 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**
  - **interrupt 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**

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

- There are seven processor modes in total:
  - six privileged modes (abort, fast interrupt request, interrupt request supervisor, system, and undefined)
  - one non privileged mode (User).

- A privileged mode allows **full read-write access to the *cpsr*.**
- *Conversely, 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.***

- ***Abort Mode:*** The processor enters abort mode when there is a failed attempt to **access memory.**

- ***Fast interrupt and interrupt request modes:*** Correspond to the two interrupt levels available on the ARM processor.
- ***Supervisor mode:*** is 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

User and system	Fast interrupt request	Interrupt request	Supervisor	Undefined	Abort
r0					
r1					
r2					
r3					
r4					
r5					
r6					
r7					
r8	r8_fiq				
r9	r9_fiq				
r10	r10_fiq				
r11	r11_fiq				
r12	r12_fiq				
r13 sp	r13_fiq	r13_irq	r13_svc	r13_undef	r13_abt
r14 lr	r14_fiq	r14_irq	r14_svc	r14_undef	r14_abt
r15 pc					
cpsr					
-	spsr_fiq	spsr_irq	spsr_svc	spsr_undef	spsr_abt

- There are 37 registers in the register file. Of those, 20 registers are hidden registers and are called as *banked registers i*ndicated in shade.

Table 2.1 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

State and Instruction Sets

- The state determines which **instruction set is being executed**.
- There are three instruction states:
  - **ARM state**
  - **Thumb state, and**
  - **Jazelle state.**

ARM State:

- The ARM instruction set (32 bit) is only active when the processor is in ARM state (T=0 and J=0).

Thumb State:

- Similarly the Thumb instruction set (16 bits) is only active when the processor is in Thumb state (T=1 and J=0).

- In Thumb state the processor executes only Thumb 16-bit instructions. You cannot intermingle ARM , Thumb, and Jazelle instructions

Jazelle State

- The Jazelle instruction set (8 bits) is only active when the processor is in Jazelle state (T=0 and J=1).

- *Jazelle executes* 8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java byte-codes.

Comparison of ARM and Thumb State

ARM and Thumb instruction set features.

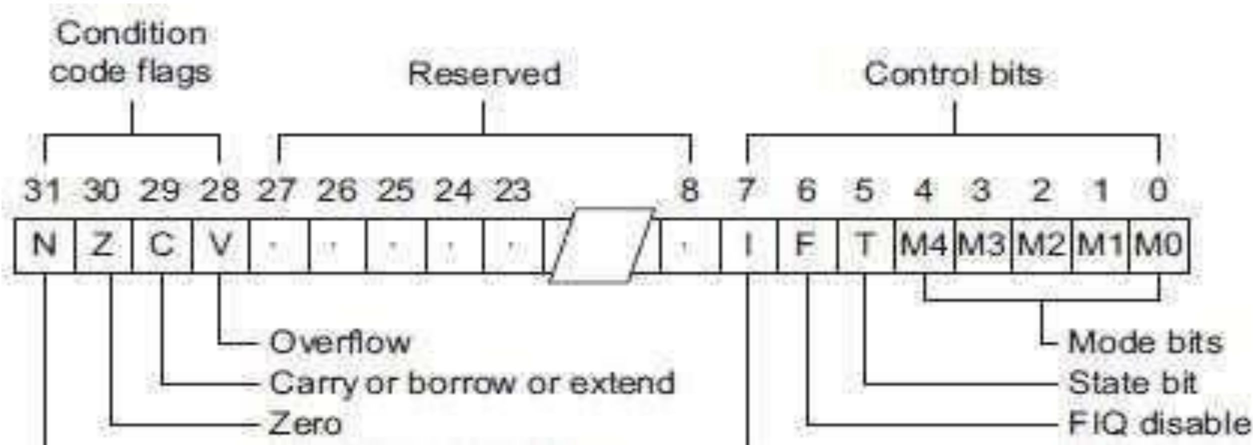
	ARM ($cpsr$ $T = 0$)	Thumb ($cpsr$ $T = 1$)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution[a]	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged 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 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.*

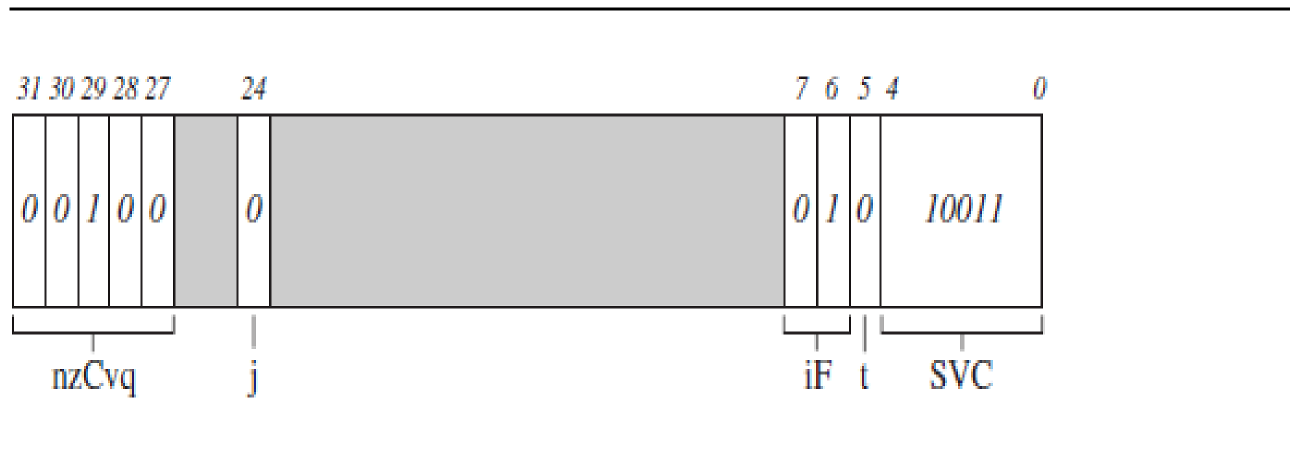


Condition Flags

- 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.**

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

- **Notation** that presents the *cpsr data in a more human readable form.*
- When a bit is a **binary 1** we use a **capital letter**;
- when a bit is a **binary 0**, we use **a lowercase letter**.
- For the **condition flags a capital letter** shows that the **flag has been set.**
- **For interrupts a capital letter** shows that an **interrupt is disabled.**



Example: *cpsr = nzCvqjiFt_SVC.*

Conditional Execution

- Conditional execution **controls the execution of an instruction**.
- Most instructions **have a condition attribute that determines** If the Microcontroller core will execute it or not 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.

Condition mnemonics

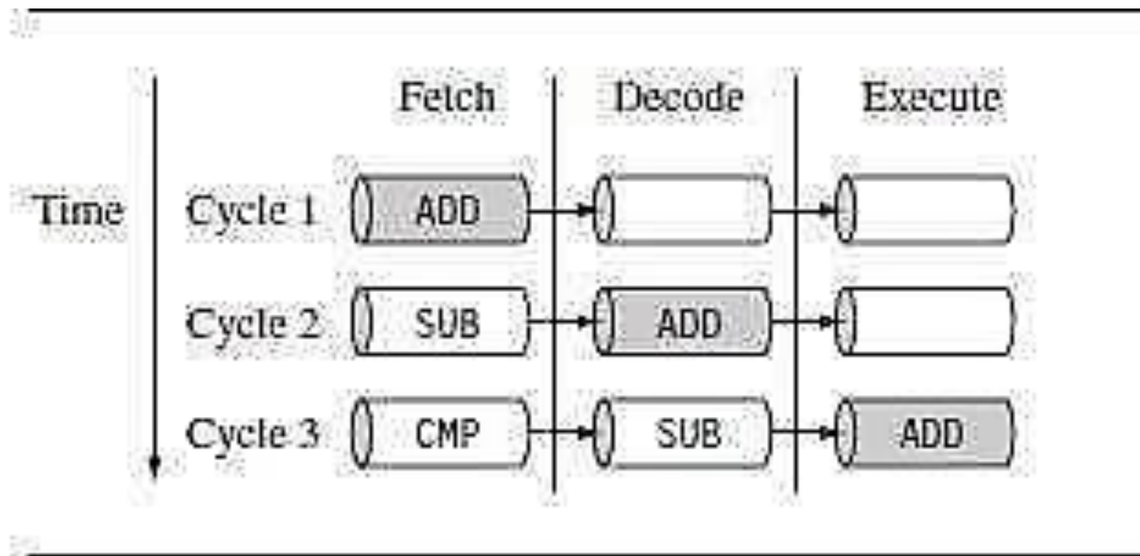
Mnemonic	Name	Condition flags
EQ	equal	*Z*
NE	not equal	*z*
CS HS	carry set/unsigned higher or same	*C*
CC LO	carry clear/unsigned lower	*c*
MI	minus/negative	*N*
PL	plus/positive or zero	*n*
VS	overflow	*V*
VC	no overflow	*v*
HI	unsigned higher	*zC*
LS	unsigned lower or same	*Z* or *c*
GE	signed greater 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

ARM7 Three-stage pipeline.

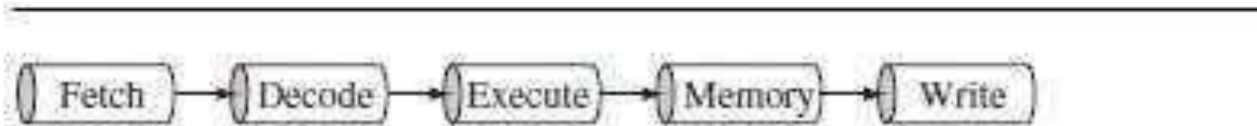
- ***Fetch loads an instruction from memory.***
- ***Decode identifies the instruction to be executed.***
- ***Execute processes the instruction and writes the result back to a register***

Example

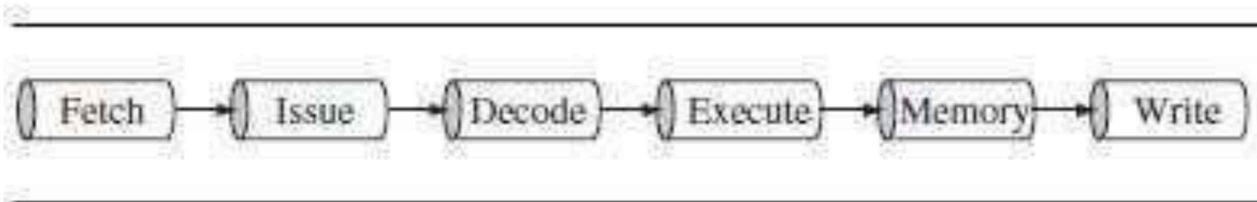


Pipelined instruction sequence.

ARM9 and ARM10 Pipeline



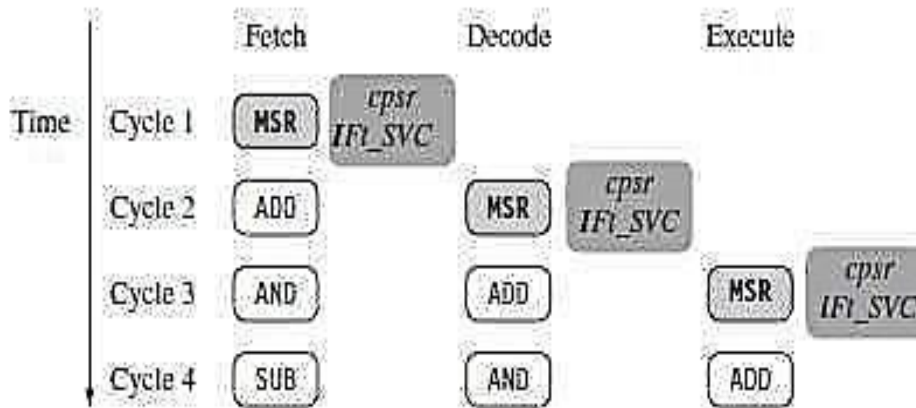
ARM9 five-stage pipeline.



ARM10 six-stage pipeline.

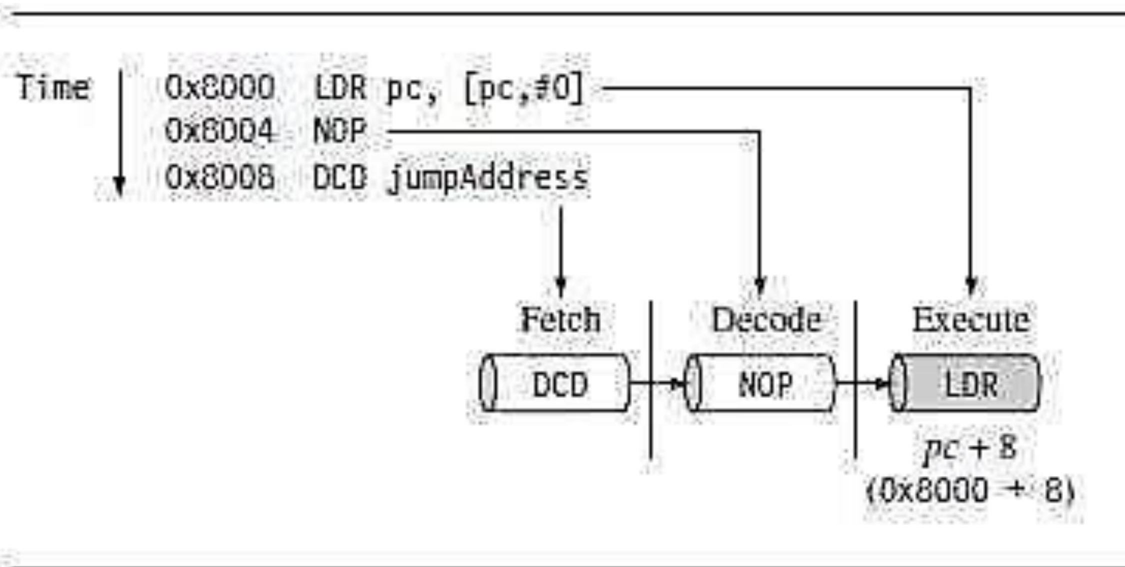
- ARM9 has instruction throughput of around 13% more compared with an ARM7.
- The ARM10 has instruction throughput of around 34% more than

Pipeline Executing Characteristics



ARM instruction sequence.

- 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



- In the execute stage, the *pc always points to the* **address of the instruction plus 8 bytes.**
- *In other words,* the *pc alwayspoints to the* ***address of the instruction being executed plus two instructions*** **ahead.**

Exceptions, Interrupts, and the Vector Table

- When an exception or interrupt occurs, the **processor sets the *pc* *to a specific memory* address.**
- This 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.
- On some processors the vector table is located at a **higher address in memory** (**starting at the offset 0xffff0000**).

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

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

- **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 called 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.**

- ***Interrupt request vector:***

  is used by external hardware ***to interrupt the normal execution*** **flow of the processor**. It can only be raised **if IRQs are not masked in the *cpsr*.**

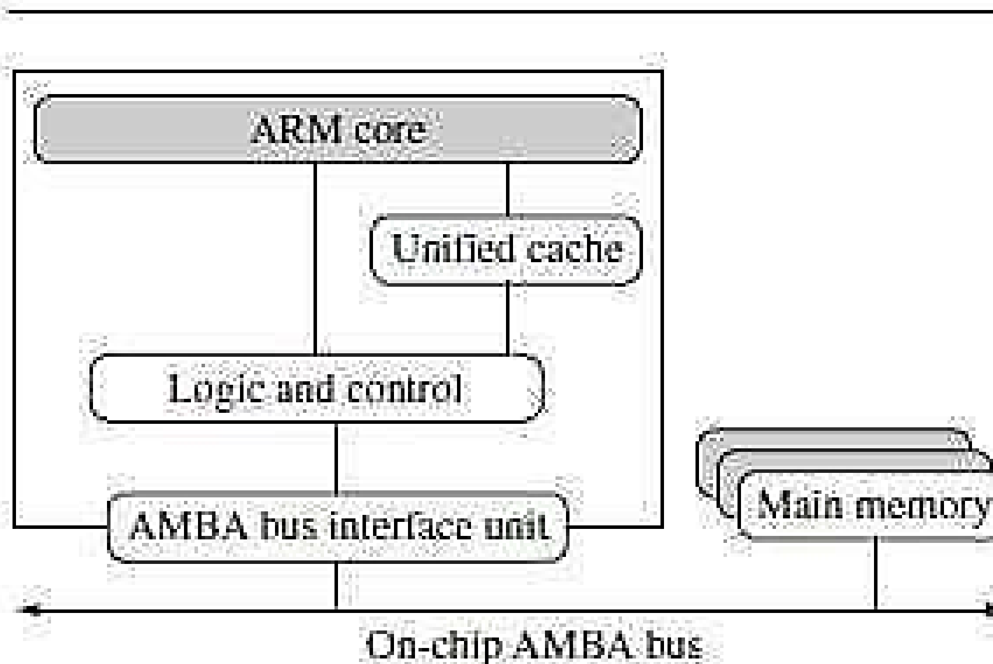
- ***Fast interrupt request vector:***

  is similar to the interrupt request but is ***reserved for hardware*** **requiring faster response times**. It can only be raised if **FIQs are not masked in the *cpsr*.**

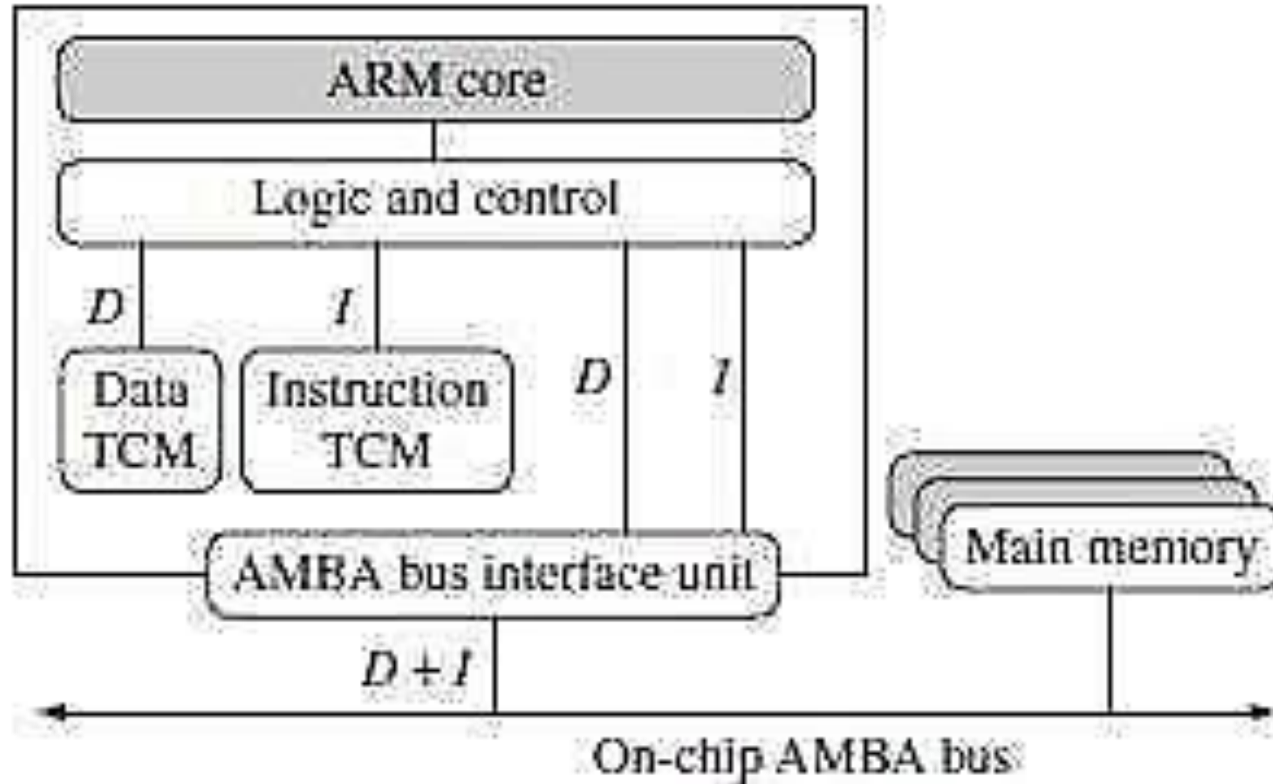
Core Extensions

- The hardware extensions (Assistants) are **standard components placed next to the ARM core**.
- They improve **performance, manage resources, and provide extra functionality** and are designed to provide flexibility in handling particular applications.
- There are **three hardware extensions** ARM wraps around the core:
  - cache and Tightly Coupled Memory(TCM),
  - memory management, and
  - the coprocessor interface.

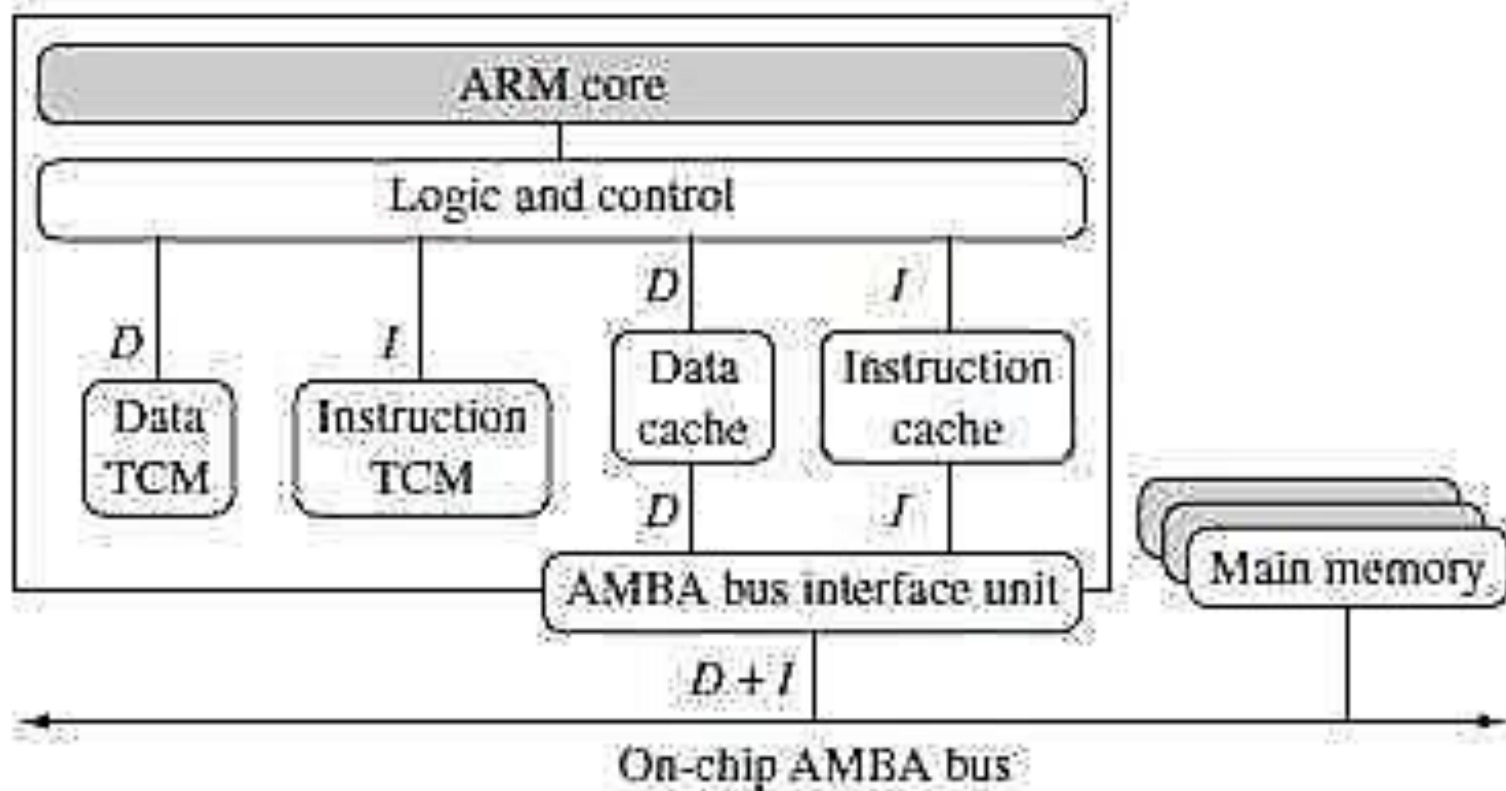
Cache and Tightly Coupled Memory



A simplified Von Neumann architecture with cache.



A simplified Harvard architecture with 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**.
  - ***Non-protected memory – no protection***
  - ***MPUs – limited protection***
  - ***MMUs – full protection***

Coprocessors

- Coprocessors can be **attached to the ARM processor**.
- A coprocessor extends the processing features of a core and increases the throughput
- **More than one coprocessor** can be added to the ARM core via the coprocessor interface.

Module-2

Introduction to ARM Instruction Sets

ARM Instruction Set

Mnemonics	ARM ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
B	v1	branch relative +/− 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation
CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register
MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values

SBC	v1	subtract with carry of two 32-bit values
SMLA*xy*	v5E	signed multiply accumulate instructions $((16 \times 16) + 32 = 32\text{-bit})$
SMLAL	v3M	signed multiply accumulate long $((32 \times 32) + 64 = 64\text{-bit})$
SMLAL*xy*	v5E	signed multiply accumulate long $((16 \times 16) + 64 = 64\text{-bit})$
SMLAW*y*	v5E	signed multiply accumulate instruction $(((32 \times 16) \gg 16) + 32 = 32\text{-bit})$
SMULL	v3M	signed multiply long $(32 \times 32 = 64\text{-bit})$

SMUL*xy*	v5E	signed multiply instructions $(16 \times 16 = 32\text{-bit})$
SMULW*y*	v5E	signed multiply instruction $((32 \times 16) \gg 16 = 32\text{-bit})$
STC STC2	v2 v5	store to memory single or multiple 32-bit values from coprocessor
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interruption
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long $((32 \times 32) + 64 = 64\text{-bit})$
UMULL	v3M	unsigned multiply long $(32 \times 32 = 64\text{-bit})$

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.*

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	$Rd = \sim N$

Examples

```
PRE     r5 = 5
        r7 = 8
        MOV     r7, r5      ; let r7 = r5
POST    r5 = 5
        r7 = 5
```

Condition Embedded in ADD Instruction

{cond} Rd, N

```
MOVEQ   r0,   r1      ; If zero flag set then…
                      ; … r0 = r1
```

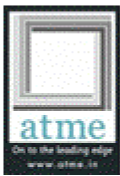
Example

```
    AREA MOV1, CODE, READONLY

START
    MOV R0,#5
    MOV R1,#6
    SUBS R0,R0,#4       ; FIRST SUBTRACT 5 AND NEXT 4
    MOVEQ R2,R1
    MOV R3,#10
BACK B
    BACK
    END
```

- For example to move the value the zero to the destination register and set condition flags.

MOVS Rd, N

```
MOVS r0,r1      ; r0 = r1
                ; ... and set flags
```

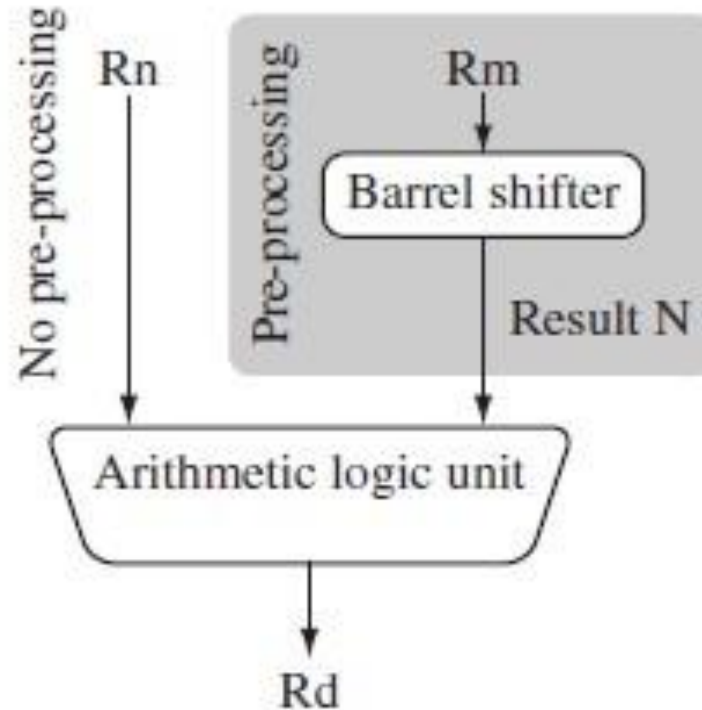


A T M E College of Engineering

Example

```
    AREA MOV2, CODE, READONLY

START
    MOV R0,#0 MOV
    R1,#6 MOVS
    R2,R0 ; Z MOVS
    R3,R1 ; z
BACK B BACK
    END
```

Barrel Shifter



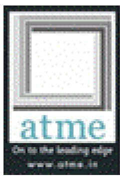
Barrel shifter and ALU.

- In MOV instruction *N is a simple register/*immediate value.
- It can also be a register *Rm that has been* preprocessed by the barrel shifter prior to being used by a data processing instruction.
- Majority of the 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)



Example

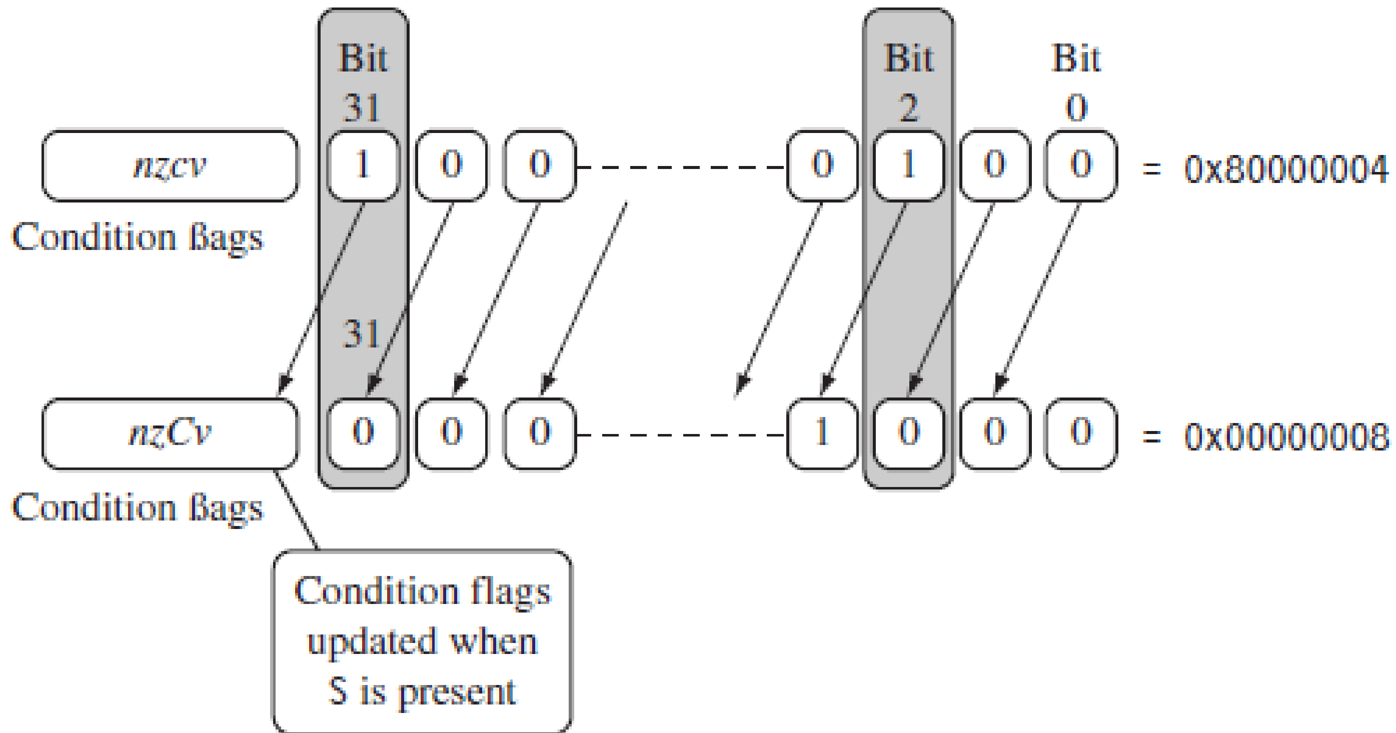
```
PRE     r5 = 5
        r7 = 8

        MOV     r7, r5, LSL #2   ; let r7 = r5*4 = (r5<<2)

POST  r5 = 5
      r7 = 20
```

Barrel shifter operations

Table 3.2 Barrel shifter operations.

Mnemonic	Description	Shift
LSL	logical shift left	*x*LSL *y*
LSR	logical shift right	*x*LSR *y*
ASR	arithmetic right shift	*x*ASR *y*
ROR	rotate right	*x*ROR *y*
RRX	rotate right extended	*x*RRX

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



Logical shift left by one.

Barrel Shifter - Left Shift

- Shifts left by the specified amount
- Multiplies by powers of two
- e.g.

    LSL #5 = multiply by 32

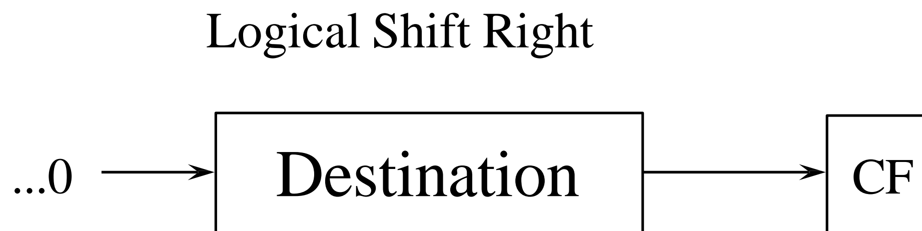
Logical Shift Left (LSL)

Barrel Shifter - Right Shifts

Logical Shift Right

- Shifts right by the specified amount (divides by powers of two)
- e.g.

LSR #5 = divide by 32



Arithmetic Shift Right

<u>Arithmetic Shift Right</u>

- Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations.
- e.g. ASR #5 = divide by 32

Arithmetic Shift Right

Sign bit shifted in

Barrel Shifter - Rotations

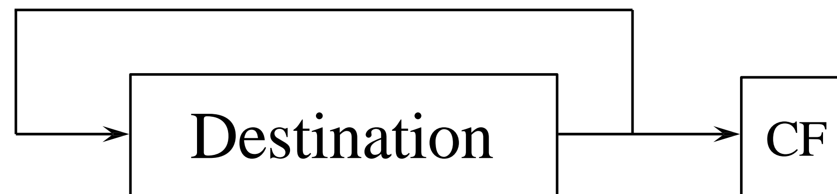
Rotate Right (ROR)

- Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB.

  e.g. ROR #5

- Note the last bit rotated is also used as the Carry Out.

Rotate Right



Barrel Shifter - Rotations

Rotate Right Extended (RRX)

Rotate Right Extended (RRX)

- This operation uses the CPSR C flag as a 33rd bit.
- Rotates right by 1 bit. Encoded as ROR #0.

Rotate Right through Carry

MOVS - example

```
PRE     cpsr = nzcvqiFt_USER
        r0 = 0x00000000
        r1 = 0x80000004

        MOVS    r0, r1, LSL #1

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

Barrel shift operation syntax for data processing instructions.

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

Arithmetic Instructions

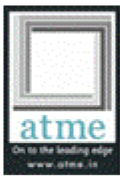
Syntax: <instruction>{<cond>}{S} Rd, Rn, N

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	$Rd = Rn - N$

Example-1

- 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
```



Example-2

- 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
```

Example-3

- 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
```

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.

```
PRE     r0 = 0x00000000
        r1 = 0x00000005

        ADD     r0, r1, r1, LSL #1

POST    r0 = 0x0000000f
        r1 = 0x00000005
```

Example-1

```
    AREA MOV2, CODE, READONLY

START
    MOV R0,#5 MOV
    R1,#6
    add R2,R0,R1, LSL #0x2
BACK B BACK
    END
```

Example-2

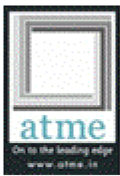
```
AREA ADD1, CODE, READONLY

START
    MOV R0,#5
    MOV R1,#6
    SUBS R0,R0,#4          ; FIRST SUBTRACT 5 AND NEXT 4
    ADDEQ R2,R1,R0
    MOV R3,#10
BACK B
    BACK
    END
```

Logical Instructions

- Logical instructions perform bitwise logical operations on the two source registers.

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

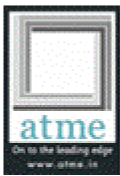
AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$



Example-1

- 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
```



Example-2

- A more complicated logical instruction called BIC, which carries out a logical bit clear.

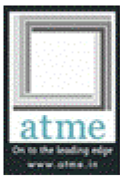
```
PRE     r1 = 0b1111
        r2 = 0b0101

        BIC   r0, r1, r2

POST    r0 = 0b1010
```

This is equivalent to

```
        Rd = Rn AND NOT(N)
```

Example-3

```
    AREA ADD1, CODE, READONLY

START
    MOV R0,#5 MOV
    R1,#6 AND
    R2,R1,R0
BACK B BACK
    END
```



Example-4

```
    AREA ADD1, CODE, READONLY

START
    MOV R0,#5 MOV
    R1,#6 AND
    R2,R1,R0
BACK B BACK
    END
```

Example-5

```
    AREA ADD1, CODE, READONLY

START
    mov R0,#15 mov
    R1,#5
    bic R2,R0,R1 ; r2=r0 & (~R1)
BACK B BACK
    END
```

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.

```
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 $Rn \& N$

CMP Example

```
PRE     cpsr = nzcvqiFt_USER
        r0 = 4
        r9 = 4

        CMP   r0, r9

POST    cpsr = nZcvqiFt_USER
```

CMN Instruction

- The CMN instruction adds the value of *N* to the value in *Rn*.
- This is the same as an ADDS instruction, except that the result is discarded.
- The conditional flags are updated accordingly.

RO=50

R1=10

CMN   RO, R1

TEQ

- Test Equivalence.
- The TEQ instruction performs a bitwise Exclusive OR operation on the value in *Rn* and the value of *N*. the result is discarded.
- Use the TEQ instruction to test if two values are equal, without affecting the V or C flags (as CMP does). It affects the N and Z flags.

TEQ- Example

```
    AREA TEST, CODE, READONLY ENTRY

START
    MOV R0, #6
    MOV R1, #5
    TEQ R0,R1
BACK B BACK
END
```

TST - Instruction

- This instruction tests the value in a register against N. It updates the condition flags on the result, but does not place the result in any register.
- The TST instruction performs a bitwise AND operation on the value in Rn and the value of N.
- This is the same as an ANDS instruction, except that the result is discarded.

TST – EXAMPLE-1

```
    AREA TEST, CODE, READONLY
ENTRY

   START
      MOV    R0,#6
      MOV    R1,#9
      TST R0,R1              ; 6 & 9 = 0 Z
   BACK B BACK
      END
```

TST – EXAMPLE-2

```
    AREA TEST, CODE, READONLY
ENTRY
    START
        MOV     R0,#6
        MOV     R1,#5
        TST R0,R1               ; 6 & 5 = 4 z
    BACK B BACK
        END
```

Multiply Instructions

- The multiply instructions multiply the contents of a pair of registers.

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

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

MLA- EXAMPLE-1

```
    AREA TEST, CODE, READONLY
ENTRY
      START
          MOV    R0,#6
          MOV    R1,#5
          MOV R2,#1                ;R3=(R0*R1) + R2
          MLA R3,R0,R1,R2
      BACK B BACK
          END
```

MLA- EXAMPLE-2

```
    AREA TEST, CODE, READONLY
ENTRY
START
    MOV R0,#6
    MOV R1,#5
    MOV R2,#1
    MLA R0,R0,R1,R2    ;R0=(R0*R1) + R2 NOT ALLOWED
BACK B BACK
    END
```

MLA- EXAMPLE-3

```
    AREA TEST, CODE, READONLY
ENTRY
START
    MOV     R0,#6
    MOV     R1,#5
    MOV R2,#1
    MLA R1,R0,R1,R2        ;R1=(R0*R1) + R2 ALLOWED
BACK B BACK
    END
```

MLA- EXAMPLE-4

```
    AREA TEST, CODE, READONLY
  ENTRY
    START
      MOV  R0,#6
      MOV  R1,#5
      MOV R2,#1
      MLA R2,R0,R1,R2          ;R2=(R0*R1) + R2 ALLOWED
BACK B BACK
   END
```

MUL – EXAMPLE-1

```
    AREA TEST, CODE, READONLY
  ENTRY
 START
      MOV R0,#6
      MOV R1,#5
      MUL R3,R0,R1            ;R3=R0*R1
BACK B BACK
   END
```

MUL- EXAMPLE-2

```
      AREA TEST, CODE, READONLY
  ENTRY

  START
     MOV R0,#6
     MOV R1,#5
     MUL R0,R0,R1   ; THIS REGISTER SEQUENCE NOT ALLOWED
BACK B BACK
   END
```

MUL- EXAMPLE-2

```
    AREA TEST, CODE, READONLY
ENTRY

START
    MOV R0,#6
    MOV R1,#5
    MUL R1,R0,R1      ; THIS REGISTER SEQUENCE IS
                      ;ALLOWED BECAUSE N IS COPIED TO
                      TEMP REG
BACK B BACK
      END
```

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	$[RdHi, RdLo] = Rm * Rs$

SMLAL - EXAMPLE

```
    AREA TEST, CODE, READONLY
ENTRY
START
    ldr R0,=0x5
    ldr R1,=0x5
    LDR R2,=0X2
    LDR R3,=0X4
    SMLAL R3,R2,R0,R1        ;[R3Lo,R2Hi]=[R3,R2]+(R0*R1)
BACK B BACK
    END
```

SMULL – EXAMPLE

```
    AREA TEST, CODE, READONLY
ENTRY
 START
     ldr R0,=0x5
     ldr R1,=0x5
     LDR R2,=0X2
     LDR R3,=0X4
     SMULL R3,R2,R0,R1          ;[R3Lo,R2Hi]=(R0*R1)
 BACK B BACK
     END
```

UMULL- Example

```
AREA UMULL1, CODE, READONLY
START
    ldr R0,=0xf0000002 ldr
    R1,=0x00000002
    umull r3,R2,R1,R0
    MOV R3,#10
BACK B BACK
    END
```

UMLAL- Example

```
    AREA TEST, CODE, READONLY
ENTRY
START
    ldr R0,=0x5
    ldr R1,=0x5
    LDR R2,=0X2
    LDR R3,=0X4
    UMLAL R3,R2,R0,R1   ;[RdHi, RdLo] = [RdHi, RdLo] +(Rm *Rs)
BACK B BACK
    END
```

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** ***label is the address 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.***

Forward and Backward branch

- These loops are address specific, we do not include the pre- and post-conditions.
- Following example 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
```

BL (Branch with Link instruction)

- BL, instruction is similar to the B instruction but overwrites the link register *lr with a return address.*
- *It performs a subroutine call.*

  pc = *label*

  lr = address of the next instruction after the BL

- *Example*

:

```
        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
```

BX (Branch Exchange), BLX (Branch Exchange with Link)

- 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*.
- *It* is used to branch to and from Thumb code.
- The *T bit* in the *cpsr is updated by the least significant bit of the branch register. pc = Rm & 0xfffffffe, T = Rm & 1*
- *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.
  - *pc = label, T = 1*
  - *pc = Rm & 0xfffffffe, T = Rm & 1*
  - *lr = address of the next instruction after the BLX*

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.

```
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])*

- 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]
```

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 methods.

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.

Preindexing with writeback: (Base Register Updated)

```
PRE      r0 = 0x00000000
         r1 = 0x00090000
         mem32[0x00009000] = 0x01010101
         mem32[0x00009004] = 0x02020202

         LDR     r0, [r1, #4]!

POST(1)  r0 = 0x02020202
         r1 = 0x00009004
```

Preindexing: (Base Register not updated)

```
PRE     r0 = 0x00000000
        r1 = 0x00090000
        mem32[0x00009000] = 0x01010101
        mem32[0x00009004] = 0x02020202

        LDR    r0, [r1, #4]
```

Preindexing:

```
POST(2)  r0 = 0x02020202
         r1 = 0x00009000
```

Postindexing: (Base Register Updated)

```
PRE       r0 = 0x00000000
          r1 = 0x00090000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202
          LDR     r0, [r1], #4
```

Postindexing:

```
POST(3)   r0 = 0x01010101
          r1 = 0x00009004
```

Table 3.6 Examples 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)

Table 3.8 Variations of 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 Load-Store Instructions

- 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.
- Load-store multiple instructions increases interrupt latency.

- For example, on an ARM7 a load multiple instruction takes
  - $2 + Nt$ *cycles,*
  - *where N is the number of registers to load*
  - -- *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.

```
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

Addressing mode for load-store multiple instructions.

Addressing mode for 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	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

Example

```
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
```

Graphical Representation (IA)

Address pointer	Memory address	Data	
	0x80020	0x00000005	
r0 = 0x8001c →	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000003
	0x80014	0x00000002	*r2* = 0x00000002
	0x80010	0x00000001	*r1* = 0x00000001
	0x8000c	0x00000000	

Post-condition for LDMIA instruction.

Graphical Representation (IB)

Address pointer	Memory address	Data	
	0x80020	0x00000005	
r0 = 0x8001c →	0x8001c	0x00000004	*r3* = 0x00000004
	0x80018	0x00000003	*r2* = 0x00000003
	0x80014	0x00000002	*r1* = 0x00000002
	0x80010	0x00000001	
	0x8000c	0x00000000	

Post-condition for `LDMIB` instruction.

Load-store multiple pairs when base update used

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

EXAMPLE-1 LDMIA

```
 AREA  ASCENDING , CODE, READONLY

ENTRY                                  ;Mark first instruction to execute
START
        LDR R2,=CVALUE  ; ADDRESS OF CODE REGION
        LDMIA R2!,{R3-R6}
BACK B BACK
CVALUE     DCD    0X44444444 ;
           DCD   0X11111111  ;
           DCD   0X33333333  ;
           DCD   0X22222222  ;
END
```

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	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

EXAMPLE-2 LDMIB

```
    AREA  ASCENDING , CODE, READONLY

ENTRY                                   ;Mark first instruction to execute
START
     LDR R2,=CVALUE  ; ADDRESS OF CODE REGION
     LDMIB R2!,{R3-R6}
BACK B BACK
CVALUE    DCD    0X44444444 ;
          DCD   0X11111111  ;
          DCD   0X33333333  ;
          DCD   0X22222222  ;
    END
```

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	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

EXAMPLE (STMIA)

```
    AREA  ASCENDING , CODE, READONLY
ENTRY                                   ;Mark first instruction to execute
START
        LDR R2,=CVALUE                  ; ADDRESS OF CODE REGION
        LDMIA R2!,{R3-R6}
        LDR R7,=DVALUE
        STMIA R7!,{R3-R6}

BACK B BACK
CVALUE
        DCD     0X44444444 ;
        DCD     0X11111111 ;
        DCD     0X33333333 ;
        DCD     0X22222222 ;
    AREA DATA1, DATA, READWRITE
DVALUE DCD 0X00
    END
```

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	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

EXAMPLE (STMIB and LDMDA)

```
    AREA  ASCENDING , CODE, READONLY
ENTRY      ;Mark first instruction to execute
START
LDR R0,=0X40000000
LDR R1,=0x00000009
LDR R2,=0x00000008
LDR R3,=0x00000007
STMIB r0!, {r1-r3}
MOV r1, #1
MOV r2, #2
MOV r3, #3
LDMDA r0!, {r1-r3}
BACK B BACK
```

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	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

EXAMPLE (STMIB and LDMDB)

```
    AREA  ASCENDING , CODE, READONLY
ENTRY                                        ;Mark first instruction to execute
START
LDR R0,=0X40000000
LDR R1,=0x00000009
LDR R2,=0x00000008
LDR R3,=0x00000007
STMIB r0!, {r1-r3}
MOV r1, #1
MOV r2, #2
MOV r3, #3
LDMDB r0!, {r1-r3}
BACK B BACK
end
```

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	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

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;*

  -- The *push operation (placing data onto the stack) uses a store multiple instruction*
- We 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;

Descending stacks grow towards lower memory addresses.

- In *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 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).

Addressing methods for stack operations.

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

Example-STMFD

- The STMFD instruction pushes registers onto the stack, updating the *sp. Below example p*ush onto a full descending stack.
- When the stack grows the stack pointer (sp) points to the **last full entry in the stack**. Hence it is called as full and descending because address decreases.

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080014

        STMFD   sp!, {r1,r4}

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

PRE

	Address	Data
	0x80018	0x00000001
sp →	0x80014	0x00000002
	0x80010	*Empty*
	0x8000c	*Empty*

POST

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
sp →	0x8000c	0x00000002

Example-STMED

- The STMED instruction pushes the registers onto the stack but updates register *sp to* point to the next empty location downward.

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080010

        STMED   sp!, {r1,r4}

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

PRE

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
sp →	0x80010	*Empty*
	0x8000c	*Empty*
	0x80008	*Empty*

POST

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
	0x8000c	0x00000002
sp →	0x80008	*Empty*

Example-1

POPING ONE AT A TIME

```
    AREA  ASCENDING , CODE,
    READONLY
ENTRY                                   ;Mark first instruction to execute
START
            LDR     SP,=0X40000000
            LDR     R1,=0x00000009
            LDR     R2,=0x00000008
            LDR     R3,=0x00000007
            STMFA SP!, {r1-r3}
            LDMFA SP!,{R4}
            LDMFA SP!,{R5}
            LDMFA SP!,{R6}
BACK    B    BACK
    END
```

Note: STMFA == STMIB, LDMFA == LDMDA

Example-2

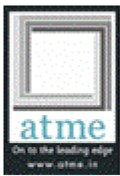
POPING ALL 3 ELEMENTS

```
    AREA  ASCENDING , CODE, READONLY

ENTRY                                   ;Mark first instruction to execute START
            LDR SP,=0X40000000
            LDR R1,=0x00000009
            LDR R2,=0x00000008
            LDR R3,=0x00000007
            STMFA SP!, {r1-r3}
            LDMFA SP!,{R4-R6}

        BACK B BACK
          END
```

Example-3

POPING FROM EMPTY STACK

```
        AREA  ASCENDING , CODE, READONLY

ENTRY                           ;Mark first instruction to execute
START

        LDR  SP,=0X40000000
        LDR   R1,=0x00000009
        LDR   R2,=0x00000008
        LDR   R3,=0x00000007
        STMFA SP!, {r1-r3}
        LDMFA SP!,{R4-R6}
        LDMFA SP!,{R7}      ; UNDER FLOW

   BACK B BACK
      END
```



A T M E College of Engineering

Example-4

Full Descending Stack

```
            AREA ASCENDING , CODE, READONLY
ENTRY                           ;Mark first instruction to execute
START
            LDR   SP,=0X4000001F
            LDR    R1,=0x00000009
            LDR    R2,=0x00000008
            LDR    R3,=0x00000007
            STMFD SP!, {r1-r3}
            LDMFD SP!,{R4-R6}
            LDMFD SP!,{R7}
      BACK B BACK
            END
```

Checking Stack Under Flow

- In handling the 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 occurs*

- 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
```

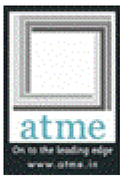
Checking Stack Under Flow

; check for stack overflow Mov

r10,rn

CMP sp, r10

BLT _stack_underflow ; condition



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.

Swap Syntax

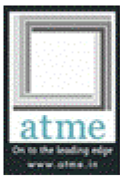
```
Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]
```

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	$tmp = mem8[Rn]$ $mem8[Rn] = Rm$ $Rd = tmp$

Note: Swap cannot be interrupted by any other instruction or any other bus access. The system “holds the bus” until the transaction is complete

Example-1

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

         SWP    r0, r1, [r2]

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



Example-2

- 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.
- If current value of the semaphore is 1, the resource is currently used by other process.
- If current value of the semaphore is 0, the resource is currently available/ not used by any of the process.
- The process that intends to use the resource (register/ memory) will make the change the semaphore from 0 to 1
- The process that intends to release the resource (register/memory) will change the semaphore from 1 to 0

EXAMPLE-3

```
        AREA  ASCENDING , CODE, READONLY
ENTRY                                   ;Mark first instruction to execute
START
                LDR R0,=0X11111111 LDR
                R1,=CVALUE
                LDR R2,[R1]
                LDR R1,=DVALUE
                STR R2,[R1] SWP
                R3,R0,[R1]
BACK B BACK
CVALUE DCD 0X22222222

    AREA DATA1, DATA, READWRITE
DVALUE DCD 0X22222222

    END
```

Software Interrupt Instruction

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

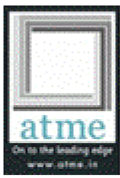
```
Syntax: SWI{<cond>} SWI_number
```

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 processor mode changes to *SVC, This* 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.

Example

```
    AREA HelloW,CODE,READONLY; declare area
SWIWrite EQU 0x00                   ; Angel SWI number
ENTRY                               ; code entry point
START ADR r1, TEXT-1                ; r1 -> “Hello World” -1
LOOP MOV r0, #0x1                   ; Angel write char in [r1]
    LDRB r2, [r1,#1]!               ; get the next byte
    CMP r2, #0                      ; check for text end
    SWINE SWIWrite                  ; if not end print ..
    BNE LOOP                        ; .. and loop back
    MOV r0, #0x18                   ; Angel exception call
    LDR r1, =0x20026                ; Exit reason
    SWI &11                         ; end of execution
TEXT = "Hello World",0xA,0xD,0
    END                             ; end of program source
```



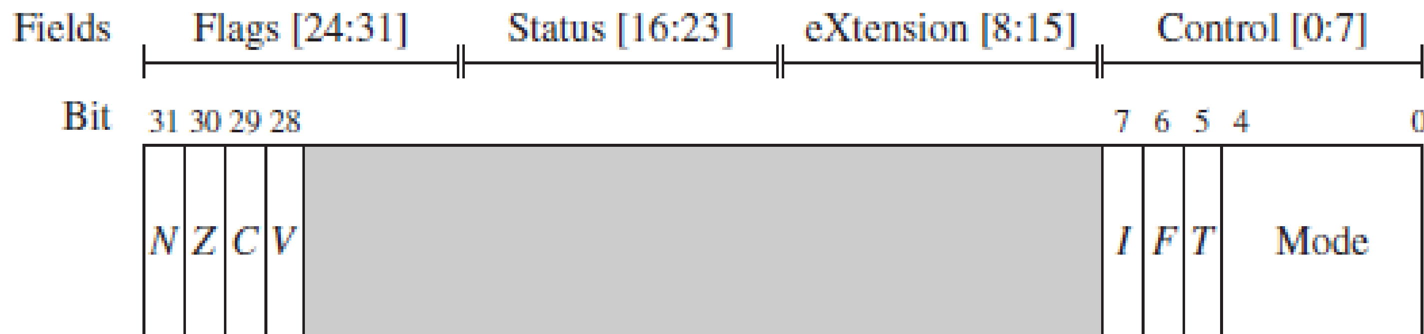
Program Status Register Instructions

- 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.*

MRS, MSR Syntax

- In syntax the label called fields *can be any combination of control (c), extension (x), status (s), and flags (f).*

```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```



Example

```
PRE     cpsr = nzcvqIFt_SVC

        MRS    r1, cpsr
        BIC    r1, r1, #0x80 ; 0b01000000
        MSR    cpsr_c, r1

POST    cpsr = nzcvqiFt_SVC
```

- 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.*
- This code preserves all the other settings in the *cpsr and only modifies the I bit* in the control field.
- 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 provides **additional computation capability**.
- The coprocessor instructions include

  - data processing,

  - register transfer, and

  - memory transfer instructions.

- Note that these instructions are only used by **cores with a coprocessor**.

Co-processor Instruction Syntax

```
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

- 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.*
- Coprocessor 15 (CP15) is reserved for system control purposes, such as memory management, write buffer control, cache control, and identification registers.

Example

- Transferring the contents of CP15 coprocessor register c0 to register r10.

```
; 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.*

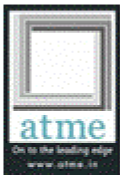
Loading Constants

- In ARM there is no **ARM instruction to move a 32-bit constant into a register**.
- 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

Module-3:C Compilers and Optimization



A T M E
College of Engineering

Basic C Data types

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

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

LOOPING CONSTRUCTS

Decremented counted LOOPS

For a decrementing loop of N iterations, the loop counter i counts down from N to 1 inclusive. The loop terminates with $i = 0$. An efficient implementation is

```
    MOV i, N

Loop
    ; loop body goes here and i=N,N-1,...,1
    SUBS i, i, #1
    BGT loop
```

Looping Constructs

Decremented counted LOOPS

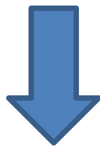
For a decrementing loop of *N* iterations, the loop counter *i* counts down from *N* to 1 inclusive. The loop terminates with *i* = 0. An efficient implementation is

```
MOV i, N

Loop
    ; loop body goes here and i=N,N-1,...,1
    SUBS i, i, #1
    BGE loop
```

Unrolled counted LOOPS

- This brings us to the subject of loop unrolling. Loop unrolling reduces the loop overhead by executing the loop body multiple times.

```
for(i=1000; i!=0; i--)
    a[i]=a[i]+s;
```



```
Loop:  LW    R2,0(R1)
       ADD   R2,R2,R3
       SW    R2,0(R1)
       ADDI  R1,R1,-4
       BNE   R1,R5,Loop
```

```
for(i=1000; i!=0; i=i-2)
    a[i]=a[i]+s;
    a[i-1]=a[i-1]+s;
```

```
Loop:  LW    R2,0(R1)
       ADD   R2,R2,R3
       SW    R2,0(R1)
       LW    R2,-4(R1)
       ADD   R2,R2,R3
       SW    R2,-4(R1)
       ADDI  R1,R1,-8
       BNE   R1,R5,Loop
```

Register Allocation

- You can use 14 of the 16 visible ARM registers to hold general-purpose data. The other two registers are the stack pointer *r13* and the program counter *r15*. For a function to be ATPCS compliant it must preserve the callee values of registers *r4* to *r11*. ATPCS also specifies that the stack should be eight-byte aligned;

```
routine_name
STMFD sp!,     {r4-r12, lr}     ; stack saved registers
    ; body of routine
    ; the fourteen registers r0-r12 and lr are available
LDMFD sp!,     {r4-r12, pc}     ; restore registers and return
```

Allocating variables To register numbers

- When you write an assembly routine, it is best to start by using names for the variables, rather than explicit register numbers. This allows you to change the allocation of variables to register numbers easily. You can even use different register names for the same physical register number when their use doesn't overlap. Register names increase the clarity and readability of optimized code.
  - *Argument registers.* The ATPCS convention defines that the first four arguments to a function are placed in registers *r0* to *r3*.
  - *Registers used in a load or store multiple.* Load and store multiple instructions LDM and STM operate on a list of registers in order of ascending register number
  - *Load and store double word.* The LDRD and STRD instructions introduced in ARMv5E operate on a pair of registers with sequential register numbers, *Rd* and *Rd* + 1.

Function calls

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. Function return integer values are passed in r0.

...	...	
sp + 16	Argument 8	
sp + 12	Argument 7	
sp + 8	Argument 6	
sp + 4	Argument 5	
sp	Argument 4	
r3	Argument 3	
r2	Argument 2	
r1	Argument 1	
r0	Argument 0	Return value

ATPCS argument passing.

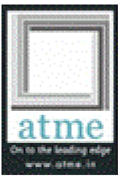
Pointer Aliasing

- 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 assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency.

Example

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

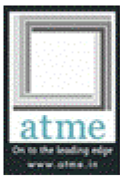
```
timers_v1
        LDR     r3,[r0,#0]          ; r3 = *timer1
        LDR     r12,[r2,#0]         ; r12 = *step
        ADD     r3,r3,r12           ; r3 += r12
        STR     r3,[r0,#0]          ; *timer1 = r3
        LDR     r0,[r1,#0]          ; r0 = *timer2
        LDR     r2,[r2,#0]         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



A T M E
College of Engineering

Solution: create new local variable

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

     timers->timer1 += step;
     timers->timer2 += step;
}
```



A T M E
College of Engineering

Portability Issues

- The char type.
- The int type.
- Unaligned data pointers.
- Endian assumptions.
- Function prototyping.
- Use of bit-fields.
- Use of enumerations.
- The volatile keyword.

Module-4

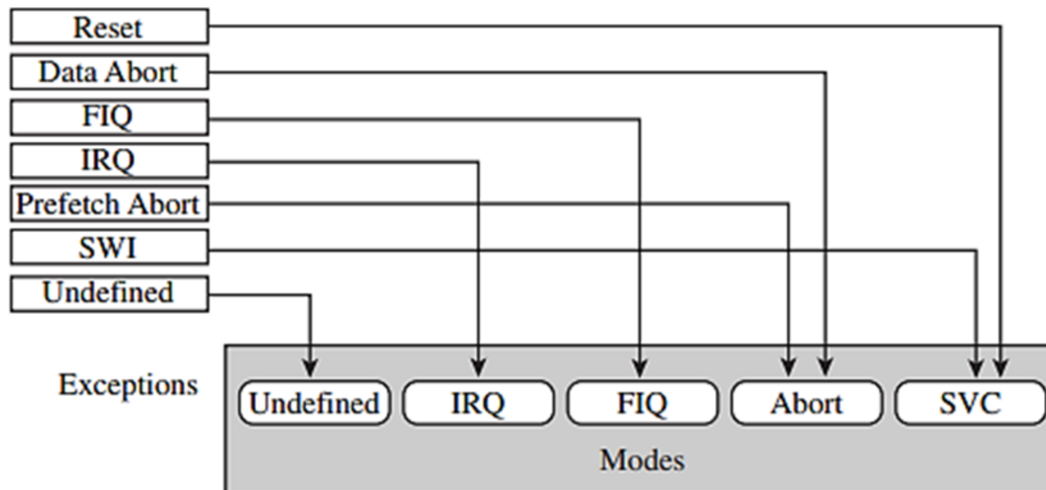
Chapter-1: Exception and Interrupt Handling

Exception handling

- An exception is any condition that needs to halt the normal sequential execution of instructions.
- Examples are when the ARM core is reset, when an instruction fetch, or memory access fails, when an undefined instruction is encountered, when a software interrupt instruction is executed, or when an external interrupt has been raised.
- Exception handling is the method of processing these exceptions.

ARM Processor Exceptions and Modes

Exception	Mode	Main purpose
Fast Interrupt Request	*FIQ*	fast interrupt request handling
Interrupt Request	*IRQ*	interrupt request handling
SWI and Reset	*SVC*	protected mode for operating systems
Prefetch Abort and Data Abort	*abort*	virtual memory and/or memory protection handling
Undefined Instruction	*undefined*	software emulation of hardware coprocessors



Exceptions and associated modes.

Vector table

- The vector table : a table of addresses that the ARM core branches to when an exception is raised.
- These addresses commonly contain branch instructions of one of the following forms:

B <address>

LDR pc, [pc, #offset]

LDR pc, [pc, #-0xff0]

Vector table and processor modes.

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

Exception Priorities

- Exceptions can occur simultaneously, so the processor has to adopt a priority mechanism.

Exception priority levels.

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	—

Link register offsets

- When an exception occurs, the link register is set to a specific address based on the current pc.

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

Interrupts

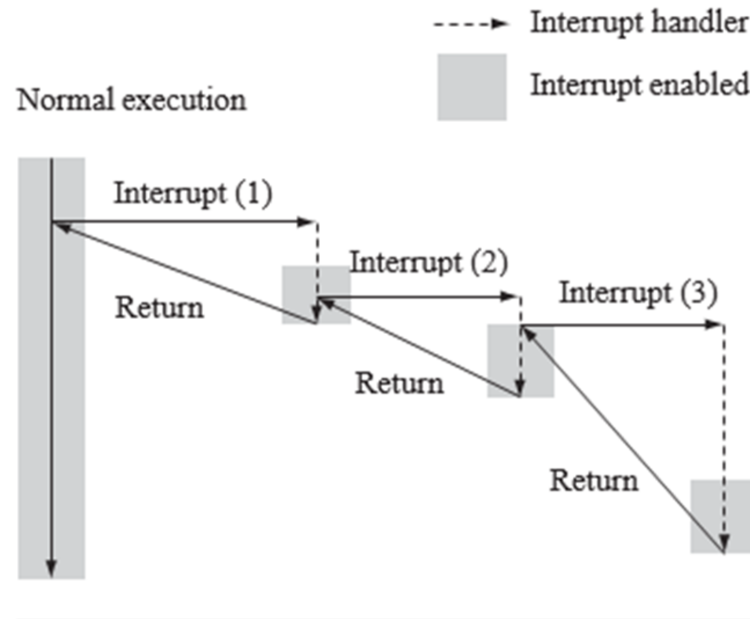
- There are two types of interrupts available on the ARM processor. Both types suspend the normal flow of a program.
- The first type of interrupt causes an exception raised by an external peripheral - namely, IRQ and FIQ.
- The second type is a specific instruction that causes an exception - the SWI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

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.

Interrupt Latency

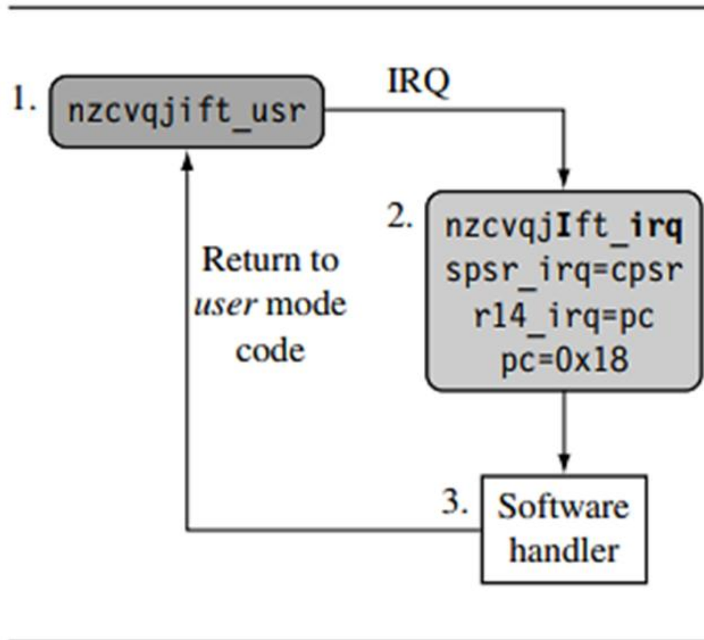
- The interval of time from an external interrupt request signal being raised to the first fetch of an instruction of a specific interrupt service routine (ISR).
- Software handlers have two main methods to minimize interrupt latency.
- The first method is to use a nested interrupt handler
- The second method involves prioritization.



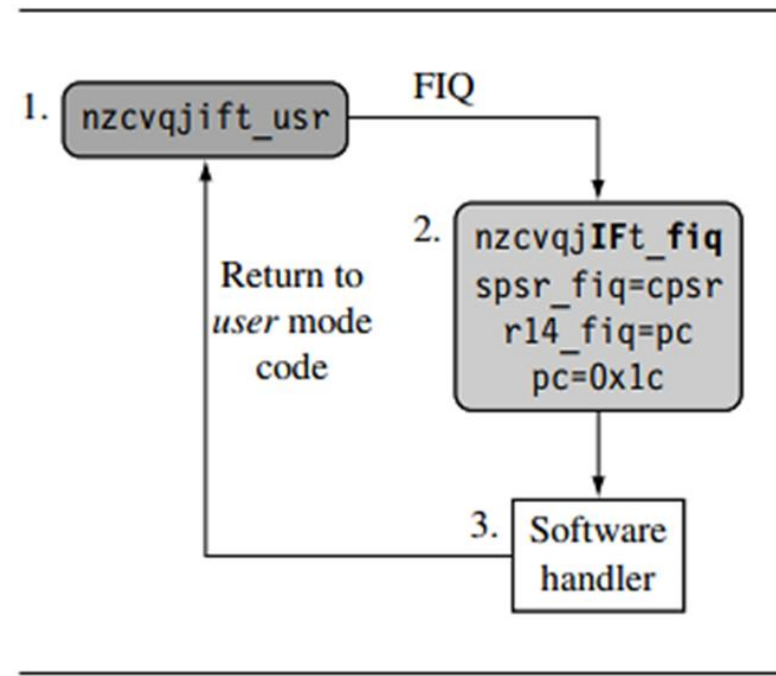
A three-level nested interrupt.

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.



Interrupt Request (IRQ).

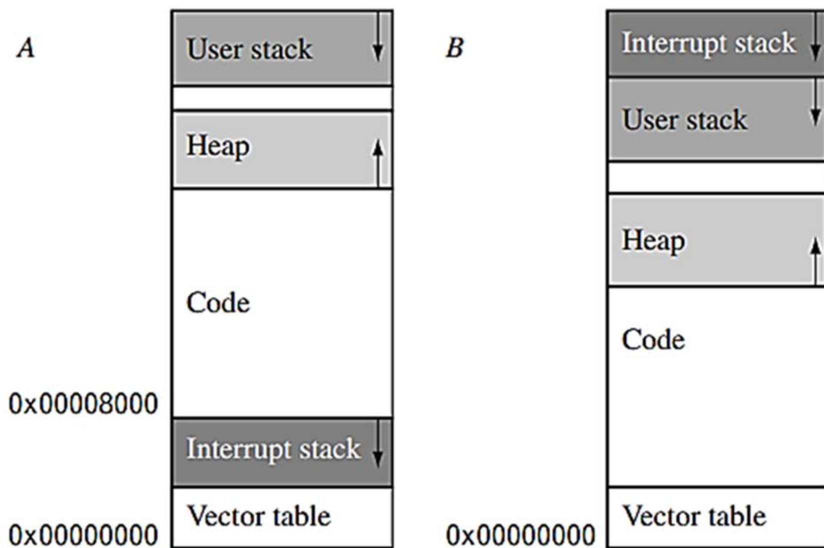


Fast Interrupt Request (FIQ).

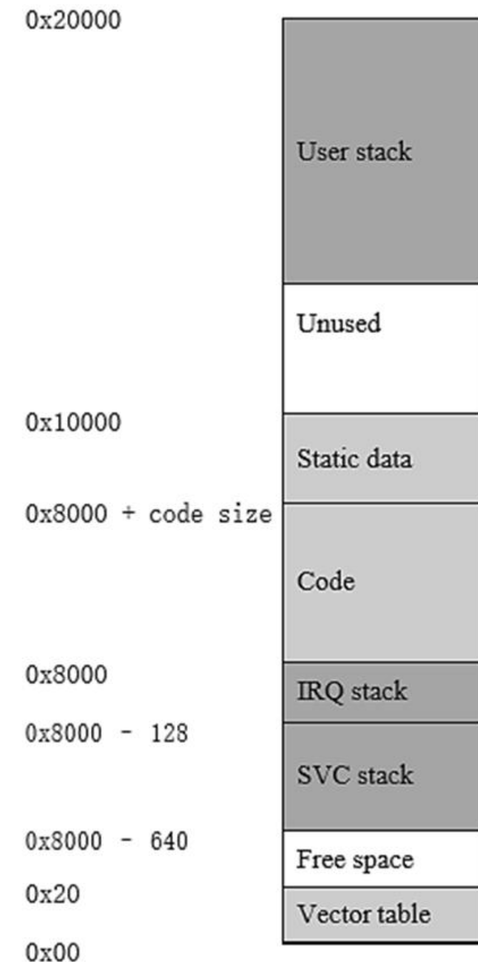
ATME College of Engineering

Basic interrupt stack design & implementation

- Stacks are needed extensively for context switching between different modes when interrupts are raised.
- The design of the exception stack depends on two factors: OS Requirements & Target hardware.



Typical memory layouts.



Example implementation using layout A.

Chapter-2: Firmware and Bootloader

Firmware

- The firmware is the deeply embedded, low-level software that provides an interface between the hardware and the application/operating system level software.
- It resides in the ROM
- Executes when power is applied to the embedded hardware system. Firmware can remain active after system initialization and supports basic system operations.

Bootloader

- The bootloader is a small application that installs the operating system or application onto a hardware target.
- The bootloader only exists up to the point that the operating system or application is executing

Firmware Execution Flow

Firmware execution flow.

Stage	Features
Set up target platform	Program the hardware system registers Platform identification Diagnostics Debug interface Command line interpreter
Abstract the hardware	Hardware Abstraction Layer Device driver
Load a bootable image	Basic filing system
Relinquish control	Alter the *pc* to point into the new image

ARM Firmware Suite

- ARM has developed a firmware package called the ARM Firmware Suite (AFS).
- AFS is designed purely for ARM-based embedded systems.
- The package includes two major pieces of technology, a Hardware Abstraction Layer called µHAL and a debug monitor called Angel.
- µHAL supports these main features: System initialization, Polled serial driver, LED support, Timer support, Interrupt controllers
- The second technology, Angel, allows communication between a host debugger and a target platform and allows to inspect and modify memory, download and execute images, set breakpoints, and display processor register contents. All this control is through the host debugger.

Red Hat RedBoot

- RedBoot is a firmware tool developed by Red Hat.
- RedBoot is designed to execute on different CPUs
- The RedBoot software core is based on a HAL.

RedBoot supports these main features:

- Communication
- Flash ROM memory management
- Full operating system support

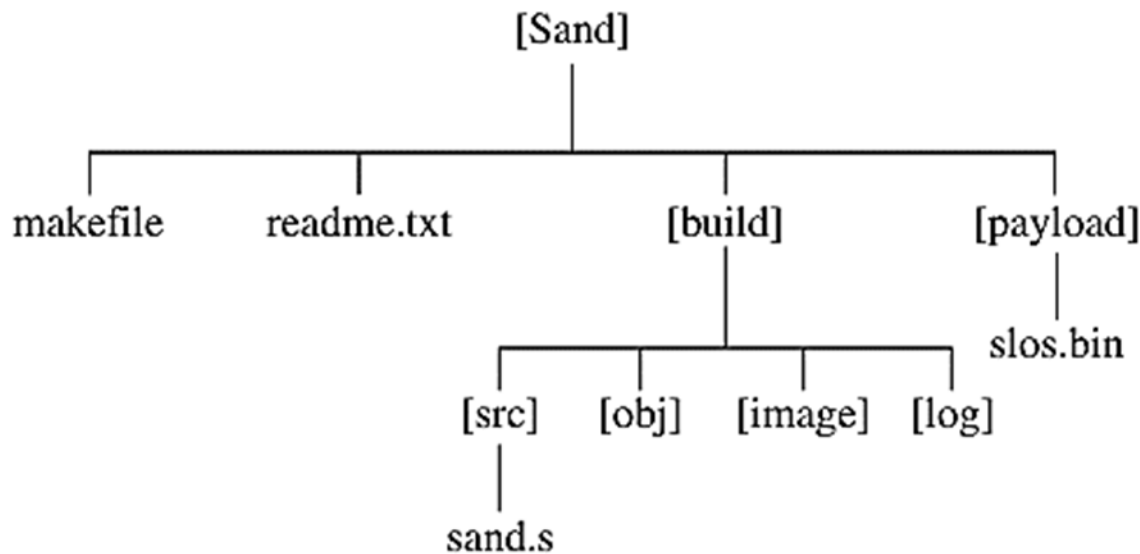
Example: Sandstone

- It is a minimal system that carries out only the following tasks: set up target platform environment, load a bootable image into memory, and relinquish control to an operating system.

Summary of Sandstone.

Feature	Configuration
Code	ARM instructions only
Tool chain	ARM Developer Suite 1.2
Image size	700 bytes
Source	17 KB
Memory	remapped

Sandstone Directory Layout



Standstone directory layout.

Sandstone code structure

- The code structure focuses more on the actual initialization and boot process.

Sandstone execution flow.

Step	Description
1	Take the Reset exception
2	Start initializing the hardware
3	Remap memory
4	Initialize communication hardware
5	Bootloader—copy payload and relinquish control