UNIT - I
INTRODUCTION TO EMBEDDED SYSTEM AND ARM PROCESSOR

1.1. Embedded System

1.1.1. Definition of Embedded System

A system is a way of working, organizing or doing one or more tasks according to a fixed plan, program or set of rules. A system is also an arrangement in which all its units assemble and work together according to the plan or program.

For example, watch is a time display system. Its parts are its hardware, needles, battery with the dial, chasis and strap. These parts organize to show the real time every second and continuosly update the time every second. Washing machine is an automatic clothes washing system.

An embedded system is a combination of computer hardware and software designed for a specific function. This system may be a specific part of an application or product or a part of a larger system. These systems are electronic systems that contain a microprocessor or microcontroller.

1.1.2. Features of Embedded system

i) Embedded systems are the modern computer devices with multifunction capabilities.

ii) An embedded system performs a specific function or a set of specific functions.

iii) Embedded systems are not always independant (standalone) devices. Embedded systems form smaller parts of a much larger device that carries out a specific task.

iv) The program instructions written for embedded systems are referred to as firmware.

v) The programs are stored in ROM or flash memory.

vi) They run with limited computer hardware resources: little memory, small or non-existent keyboard and/or screen.

vii) Embedded systems possess advanced graphical interfaces.

viii) Simple embedded system uses LEDs, buttons or LCD displays with simple menu options.

1.1.3. Characteristics of Embedded system

i) Embedded systems do a very specific task.

ii) Embedded systems have very limited resources, particularly the memory.

iii) Embedded systems are created to perform the task within a certain time frame.

iv) They have minimal or no user interface (UT).

v) Embedded systems have to work against some deadlines.

vi) Some embedded systems are designed to react to external stimuli and react accordingly.

vii) Embedded system cannot be changed or upgraded by the users.

viii) Some embedded systems have to operate in extreme environmental conditions such as very high temperatures and humidity.

ix) Embedded systems need to be highly reliable.

1.1.4. Types of Embedded systems

A) Based on the performance and functional requirements the embedded systems can be classified as follows.

a) Real time embedded systems: It provides output within a defined specific time. It can be further classified as

i) Soft real time embedded systems, and

ii) Hard real time embedded systems.

b) Standalone embedded systems: They can work themselves i.e they are self sufficient and do not depend on a host system.

c) Networked embedded systems: It depends on connected network to perform its assigned works. These systems consist of components like sensors, controllers etc., which are interconnected.

d) Mobile embedded systems: These are small sized and can be used in smaller devices. They are used in mobile phones and digital cameras.

B) Based on the performance of microcontroller the embedded systems can be classified as follows,

i) Small scale embedded systems

ii) Medium scale embedded systems

iii) Sophisticated embedded systems.

1.1.5. List of embedded system devices

a) Digital alarm clocks

b) Electronic parking meters

c) Robotic vacuum cleaners

d) Smart watches

e) Washing machines and dish washers

f) Home security systems

g) Air-conditioners

h) Electric stoves, pressure cookers, tea/coffee machines.

i) Traffic lights

j) Vending machines

k) Fire alarms and carbon monoxide detectors

l) Fax machines and scanners

m) Digital and video cameras

n) Calculators

o) Digital thermometers

p) Motion sensors

q) Handheld computers

r) Electronic toys

s) Wi-Fi routers

t) Heart rate monitors

u) Automobile systems

1.1.6. Hardvard architecture

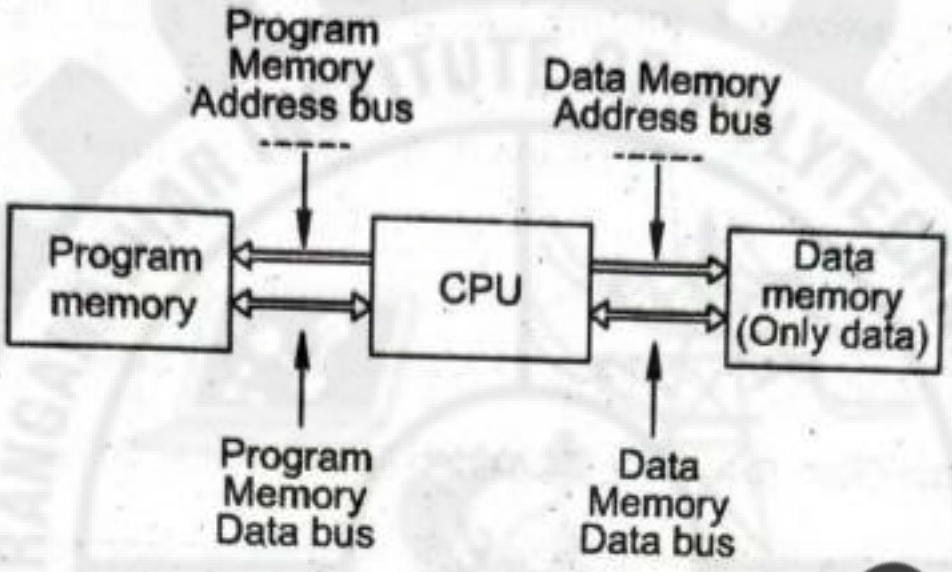


Fig.1.1 Harvard Architecture

The representation of Harvard architecture is shown in the fig 1.1. In this architecture there are two separate memory blocks. One is program memory and the other is data memory. Program memory stores only instructions and data memory stores only data. Two pairs of data buses are used between the CPU and the memory blocks. The address and data buses of program memory are used to access the program memory. The address and the data buses of data memory are used to access data memory. Certainly this architecture is much more efficient because accessing the instructions and data will be very fast.

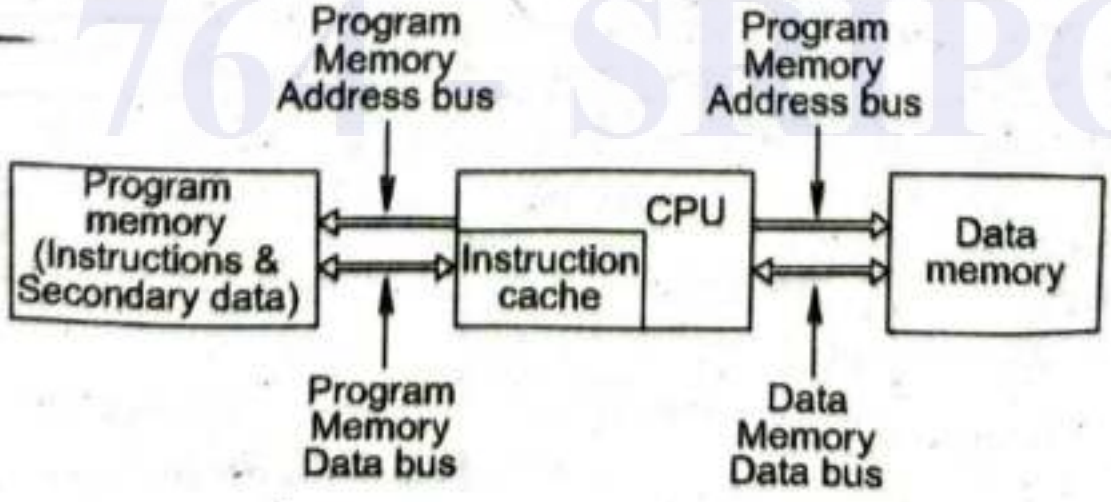


Fig.1.2 Super- Harvard Architecture

The representation of super harvard architecture (SHARC) is shown in the fig 1.2. It is a slight but significant modification of the Harvard architecture. In harvard architecture, the data memory is accessed more frequently than the program memory. Therefore in SHARC provision has been made to store some secondary data in the program memory to balance to load on both memory blocks.

1.1.7. Von-Neumann architecture

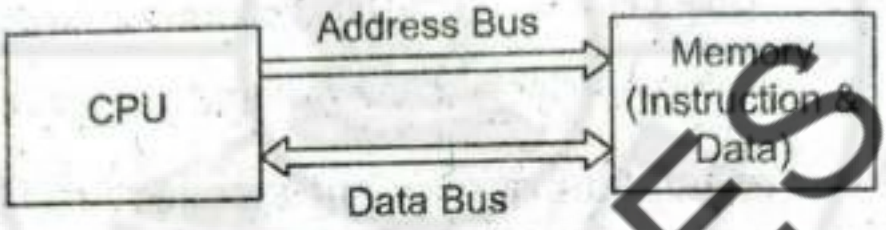


Fig.1.3 Von Neumann Architecture

The representation of Von-Neumann architecture is shown in the fig 1.3. It is the most widely used architecture. This architecture has one memory chip which stores both instructions and data. The processor interacts with the memory through address and data buses to fetch instructions and data.

1.1.8. Comparison of Von-Neumann architecture and Harvard architecture

S.no	Parameters	Von-Neumann	Harvard
i	Memory	It uses a single memory connection.	It uses separate RAM and ROM memories.
ii	Design	Simple design. It uses the same path to access instructions and data.	Complex design. It has separate path for accessing instructions and data.

iii	Hardware	It requires less hardware.	It requires more hardware.
iv	Speed	Less speed	More speed.
v	Physical space	Require less physical space	Requires more physical space.
vi	Internal memory	Internal memory is not wasted.	Internal memory is wasted.

1.1.9. RISC and CISC Processors

Processors are divided into the following categories.

i) Complex Instruction Set Computer (CISC)
ii) Reduced Instruction Set Computer (RISC)

CISC is characterized by its large instruction set. Large number of instructions are available to program the processor. Single instruction can execute several low level operations. It is also capable of executing multi-step operations or addressing mode with single instructions. So the number of instructions required to do a job is very less and hence less memory is required.

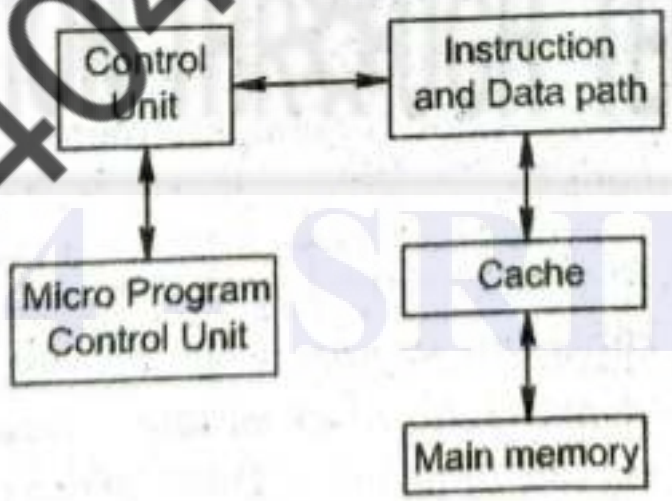


Fig.1.4 Architecture of CISC

The number of registers in CISC processors is very small. The aim of designing CISC processor is to reduce the

software complexity by increasing the complexity of the processor architecture.

Example: IntelX86 family and Motorola 68000 series processors.

The architecture of CISC is shown in the fig 1.4. It consists of Mircoprogram control unit, Control unit, Instructions and data path, Cache and main memory. The description of these units is given below.

i) **Mirco Program control unit:** The CISC uses a series of microinstructions of the microprogram stored in the control memory of the microprogram control unit and generate control signals.

ii) **Control unit:** It accesses the control signals, which are produced by the microprogram control unit.

iii) **Instructions and Data path:** It retrieves/fetches the op code and operands of the instructions from the memory.

iv) **Cache and Main memory:** Here the program instructions and operands are stored. Instructions in CISC is complex and it occupies more than a single word in memory.

RISC is characterised by its limited number of instructions. In RISC processors, the software is complex but the processor architecture is simple. However large number of registers are required in RISC processors, which are of small size and consume less power. RISC processors have pipelined instruction execution. While one instruction is being executed, second instruction is decoded and the third

instruction is fetched, leading to faster execution of the program. Embedded systems generally use RISC processors.

Examples: ARM, ATMEL, AVR, MIPS.

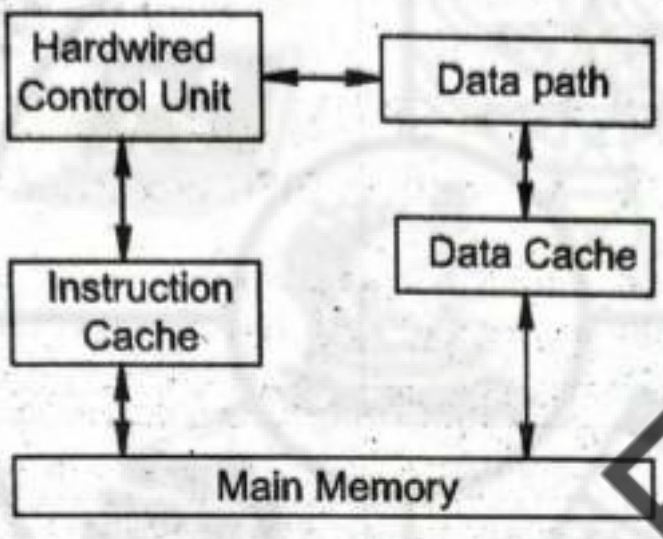


Fig.1.5 Architecture of RISC

The architecture of RISC is shown in the fig 1.5. It consists of Hardwired control unit, Data path, Instruction cache, Data cache and Main memory.

The hardwired control unit produces control signals which regulate the working of processors hardware. RISC architecture emphasizes on using the registers rather than memory.

This is because the registers are the fastest available memory source. The registers are physically small and are placed on the same chip where the ALU and control unit are placed on the processor. The RISC instructions operate on the operands present in processor's registers.

All instructions in RISC are simple and execute one instruction per cycle. So the instructions are hardwired and there is no need for control store.

1.1.10. Comparison of RISC and CISC processors.

S.no	RISC	CISC
i	It is a Reduced Instruction Set Computer.	It is a Complex Instruction Set Computer.
ii	It emphasizes on software to optimize the instruction set.	It emphasizes on hardware to optimize instruction set.
iii	It is a hardwired unit of programming in the RISC processor.	Microprogramming unit in CISC processor.
iv	It requires multiple register sets to store instructions.	It requires single register set to store instructions.
v	It has simple decoding of instruction.	It has complex decoding of instruction.
vi	Uses of pipelines are simple.	Uses of pipelines are difficult.
vii	It uses limited number of instructions.	It uses a large number of instructions.
viii	Less execution time.	More execution time.
ix	It can be used with high end applications.	It can be used with low end applications.
x	It has fixed format instruction.	It has variable format instruction.
xi	Programs need more memory space.	Programs need less memory space.

1.2. ARM Processor Architecture Fundamentals

1.2.1. ARM based Embedded system with hardware components

Embedded system can control many different devices from small sensors to the real time control systems. All these devices use a combination of software and hardware components. Selection of each component depends upon efficiency, future extension and expansion.

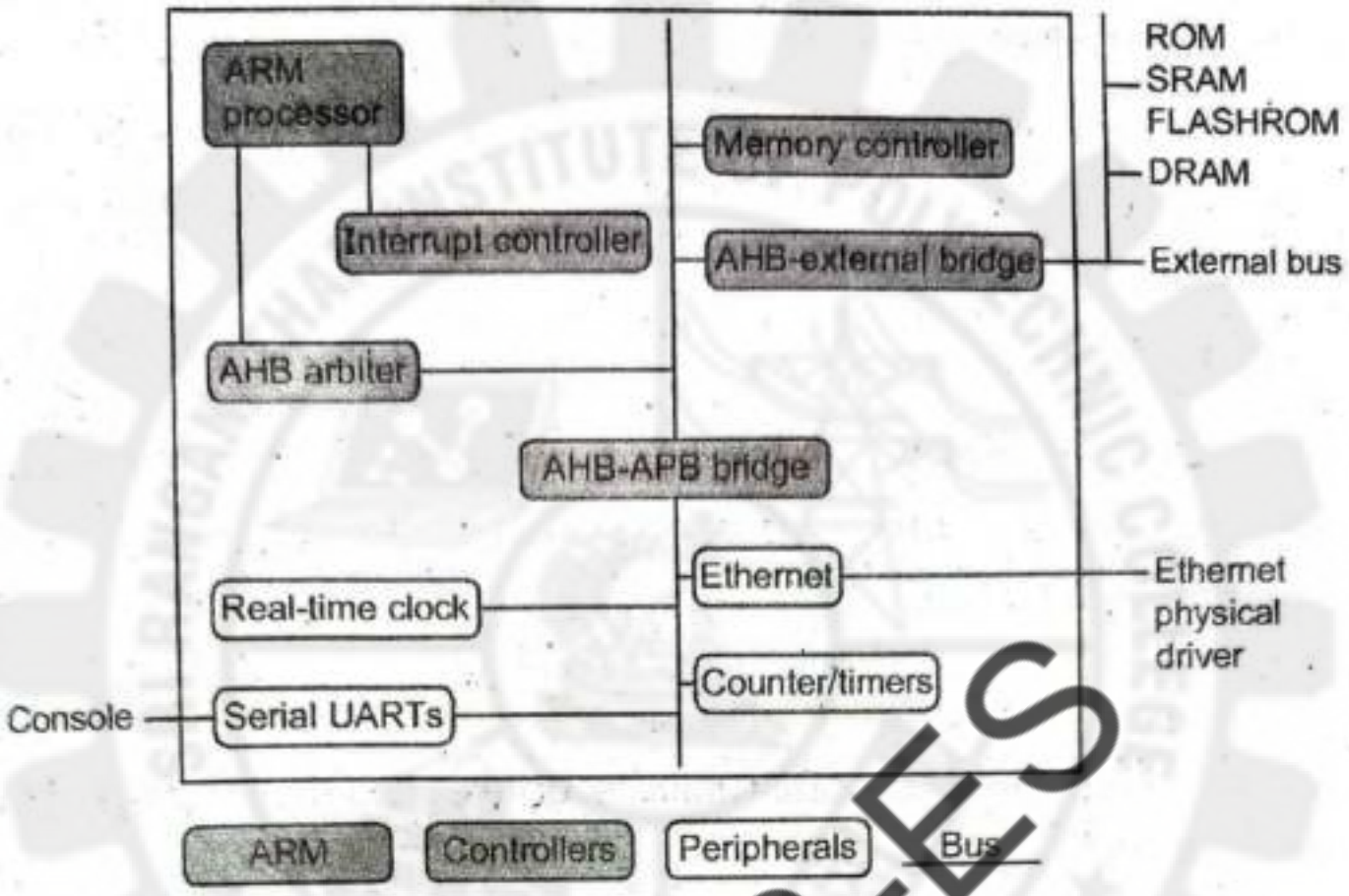


Fig.1.6 An example of an ARM-based embedded device, microcontroller

A typical embedded device based on an ARM core is shown in the fig.1.6. Each box represents a function or feature. The lines connecting the boxes are the buses carrying data.

The device is separated into four main hardware components, as described below.

i) The ARM processor

It controls the embedded device. Different versions of the ARM processors are available, with different operating characteristics. An ARM processor comprises a core plus the additional components that interface with the bus. The core is an execution engine that processes instructions and manipulates data. The components can include memory management and caches.

ii) Controllers

They coordinate important functional blocks of the system. Two commonly used controllers are interrupt and memory controllers.

iii) Peripherals

They provide all the input - output capability external to the chip and are responsible for the uniqueness of the embedded device.

iv) Bus

It is used to communicate between different parts of the device.

A) ARM bus technology

The most common bus technology is Peripheral Component Interconnect (PCI) bus and on chip bus. The PCI bus connects the devices like video cards and hard disc controllers to the X86 processor bus. This bus is designed to connect mechanically and electrically to devices external to the chip. It is built into the motherboard of the PC.

The on chip bus is internal to the chip used for interconnecting different peripheral devices with an ARM core.

B) Memory

An embedded system has some form of memory to store and execute code. The memories like cache, main and secondary storage are used with embedded system. The cache is placed between main memory and core. It is used to speed up data transfer between the processor and main memory. It increases the overall performance.

The main memory is large (256KB to 256MB) depending on application, and is generally stored in separate chips. Secondary storage is the largest and slowest form of memory. Hard disk drives and CD ROM drives are examples of secondary storage.

C) Peripherals

Embedded systems that interact with the outside world need some form of peripheral device. They perform input and output functions for the chip by connecting to other devices or sensors that are off chip.

D) Memory controllers

The memory controllers connect different types of memory to the processor bus. On power-up, a memory controller is configured in hardware to allow certain memory devices to be active. These memory devices allow the initialization code to be executed. Some memory devices must be set up by software.

E) Interrupt controller

An interrupt controller provides a programmable governing policy. It allows the software to determine which peripheral or device can interrupt the processor at any specific time by setting the appropriate bits in the interrupt controller registers.

There are two types of interrupt controller available for the ARM processor. They are the standard interrupt controller and the vector interrupt controller (VIC). The standard interrupt controller sends an interrupt signal to the processor core when an external device requests servicing.

It can be programmed to ignore or mask an individual device or set of devices. The VIC is more powerful than the standard interrupt controller because it prioritizes interrupts and simplifies the determination of which device causes the interrupt.

1.2.2. Pipeline

Pipeline is a mechanism used in RISC processors for executing instructions. Using a pipeline, execution speed will be increased by fetching the next instruction while other instructions are being decoded and executed.

A three stage pipe line has three operations of fetching, decoding and execution. The fetch loads an instruction from memory, decode identifies the instruction to be executed and execute processes the instruction and writes the result back to the register.

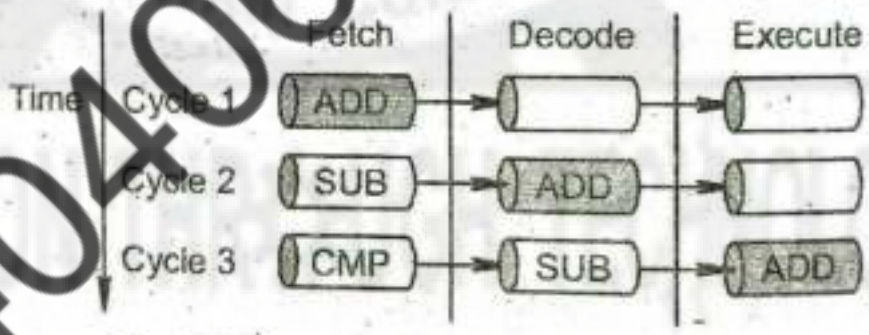


Fig.1.7 Pipelined instruction sequence

The fig.1.7. illustrates the pipeline using a simple example. It shows a sequence of three instructions being fetched, decoded and executed by the processor. Each instruction takes a single cycle to complete after the pipeline is filled. The three instructions of ADD, SUB and CMP are placed in the pipeline sequentially. In the first cycle the core fetches the ADD instruction from memory. In the second cycle the core fetches the SUB instruction and decodes the ADD

instruction. In the third cycle, both the SUB and ADD instructions are moved along the pipeline. The ADD instruction is executed, the SUB instruction is decoded and CMP instruction is fetched. The procedure is called filling the pipeline.

As the pipeline length increases, the amount of workdone at each stage is reduced. It allows the processor to attain a higher operating frequency. This in turn increases the performance.

The pipeline design for each ARM family differs. The ARM9 core increases the pipeline length to five stages, adds a memory and writeback stage. The ARM10 increases the pipeline length still further by adding a sixth stage of Issue.

Advantages

i) Amount of work done at each stage is reduced.
ii) Operating at higher frequency
iii) Performance increases.
iv) Code written for ARM 7 can be executed on an ARM 9 or ARM 10.

Disadvantages

i) Delay increases
ii) Data dependency increases.

1.2.3. Data Flow Model

The structure of data flow model is shown in the fig.1.8. The boxes represent either an operation unit or a storage area, the line represents the buses and the arrows represents the flow of data. The figure shows not only the flow of data but also the abstract components that make up an ARM core.

Data enters the processor core through the data bus. The data may be an instruction to execute or a data item. The given figure shows a Von Neumann implementation of the ARM – data items and instructions share the same bus. In contrast, the Harvard implementations of the ARM use two different buses.

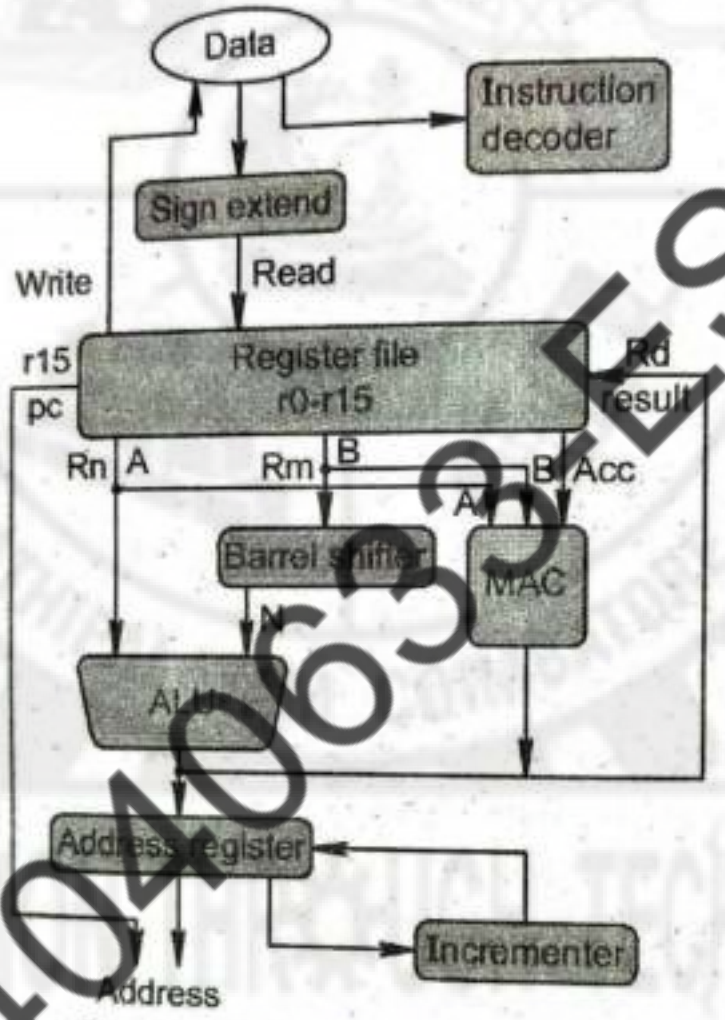


Fig.1.8 ARM core dataflow model

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

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

conversely the store instructions copy data from registers to memory. There are no data processing instructions that directly manipulate data in memory. Thus data processing is carried out in registers.

Data items are placed in register file. Register file is a storage bank made up of 32 bit registers. Since the ARM core is a 32 bit processor, most instructions treat the registers as holding signed or unsigned 32 bit values. The sign extend hardware converts signed 8 bit and 16 bit numbers to 32 bit values. Source operands are read from the register file using the internal buses A and B respectively.

The ALU (arithmetic logic unit) and MAC (multiply accumulate unit) take 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.

One important feature of the ARM is that register Rm alternatively can be preprocessed in the barrel shifter before it enters the ALU. Together the ALU and barrel shifter can calculate a wide range of expressions and addresses.

After passing through the functional units, the result in Rd is written back to the register file using the result bus. For load and store instructions the incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location. The processor continues executing instructions until an exception or interrupt changes the normal execution flow.

1.2.4. CPU Registers

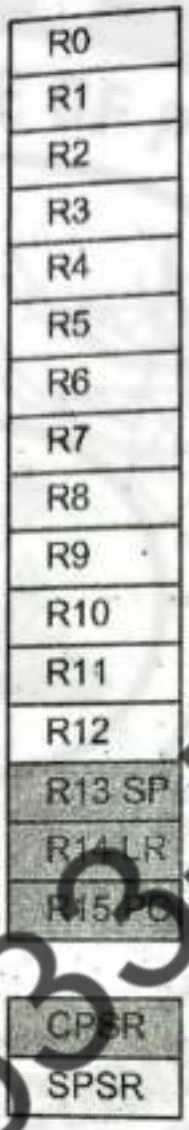


Fig.1.9 Registers available in user mode

General purpose registers hold either data or an address. They are identified with the letter "r" prefixed to the register number. For example, register 5 is given the label as *r5*. The active registers available in the user mode are shown in the fig.1.9. The user mode is a protected mode normally used when executing applications. The processor can operate in seven different modes. All the registers shown are 32 bits in size.

There are upto 18 active registers: 16 data registers and 2 processor (program) 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 functions i.e., *r13*, *r14* and *r15*. They are frequently given different labels to differentiate them from the other registers. In the given figure, the shaded registers are identified as assigned special purpose registers.

The descriptions of these registers are given below.

i) Register *r13* is called stack pointer (SP), which stores the head of the stack in the current processor mode.

ii) Register *r14* is called link register (LR), which is used for putting the return address whenever it calls a subroutine.

iii) Register *r15* is called program counter (PC), which 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. In ARM state, the registers *r0* to *r13* are orthogonal. This means any instruction we can apply to r0, we can well apply to any of the other registers.

However there are instructions that treat *r14* and *r15* in a special way. The two program status registers are called CPSR (current program status register) and SPSR (saved program status register)

1.2.5. Current Program Status Register (CPSR)

The ARM core uses the CPSR to monitor and control internal operations. The CPSR is a dedicated 32 bit register and resides in the register file. The basic layout of a generic

program status register is shown in the fig.1.10. Note that the shaded parts are reserved for future expansion.

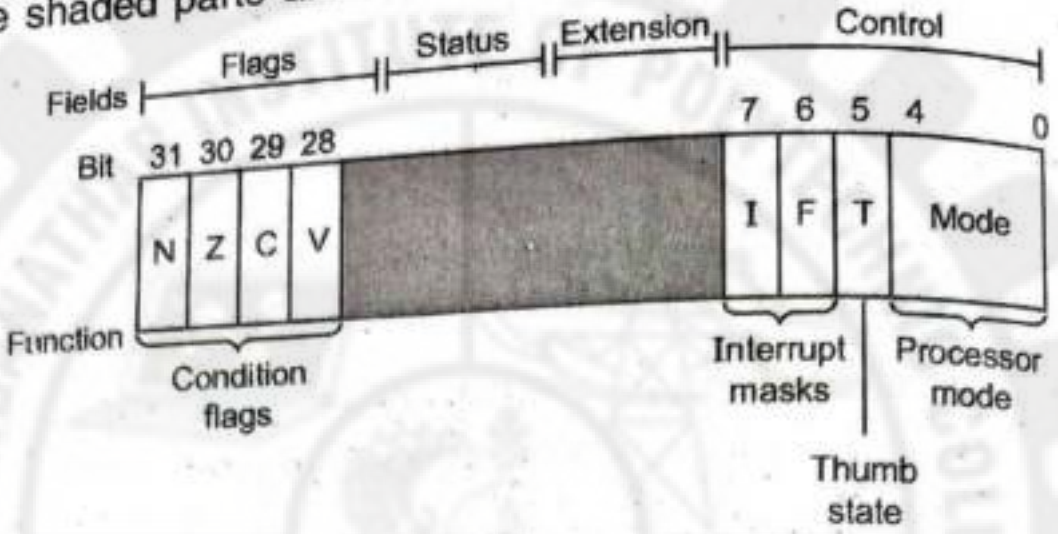


Fig.1.10 A generic program status register (PSR)

The CPSR is divided into four fields, each 8 bits wide: flags, status, extension and control. The extension and status fields are reserved for future use. The control field contains the processor mode, state and interrupt mask bits. The flag 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 executes 8 bit instructions.

A) Modes of operation

The processor mode determines which registers are active and the access rights to the CPSR register itself. Each processor mode is either privileged or nonprivileged. A privileged mode allows full read-write access to the CPSR. Conversely, a nonprivileged mode only allows read access to the control field in the CPSR but still allows read-write access to the condition flags.

Totally there are seven processor modes: Six privileged modes (abort, fast interrupt request, interrupt request, supervisor, system and undefined) and one nonprivileged mode (user).

i) Abort mode

The processor enters this mode when there is a failed attempt to access memory.

ii) Fast interrupt request and interrupt request modes

These modes correspond to the two interrupt levels available on the ARM processor.

iii) Supervisor mode

This is the mode that the processor is in after reset. It is generally the mode that an operating system kernal operates in.

iv) System mode

This mode is a special version of user mode that allows full read-write access to the CPSR.

v) Undefind mode

This mode is used when the processor encounters an instruction that is undefined or not supported by the implementation.

vi) User mode

This mode is used for programs and applications.

B) Banked register

The fig.1.11. shows all 37 registers in the register file. Of those 20 registers are hidden from a program at dif-

ferent times. These registers are called banked registers and are identified by the shading in the diagram. They are available only when the processor is in a particular mode.

For example, abort mode has banked registers *r13_abt*, *r14_abt* and *SPSR_abt*. Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic *or_mode*.

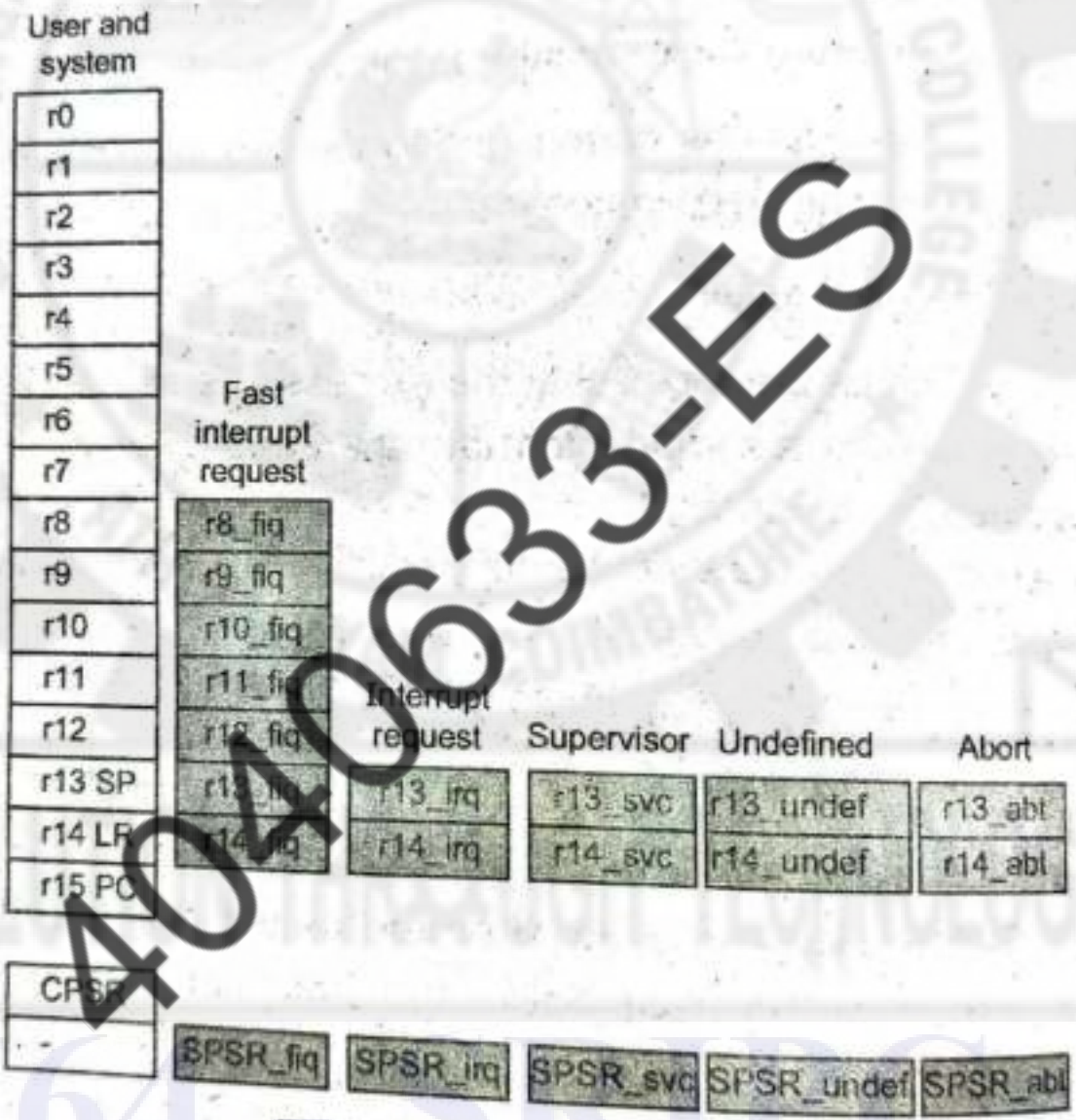


Fig.1.11 Complete ARM register set

Every processor mode except user mode can change mode by writing directly to the mode bits of CPSR. All processor modes except system mode have a set of associated banked registers that are subset of the main 16 registers. A banked register maps one to one onto a user

mode register. If we change the processor mode, a banked register from the new mode will replace an existing register.

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

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

1.2.6. Processor State and instruction sets

The state of the core determines which instruction set is being executed. There are three instruction sets: ARM, Thumb and Jazelle. The ARM instruction set is only active when the processor is in ARM state. Similarly the thumb instruction set is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16 bit instructions.

The description of state selection is shown below.

When T=0 ; The processor is in ARM state and executes ARM instructions.

When T = 1; The processor is in Thumb state and executes Thumb instructions.

When T = 0, J = 1; The processor executes Jazelle instruction.

The features of ARM and Thumb instructions are shown in the table 1.1.

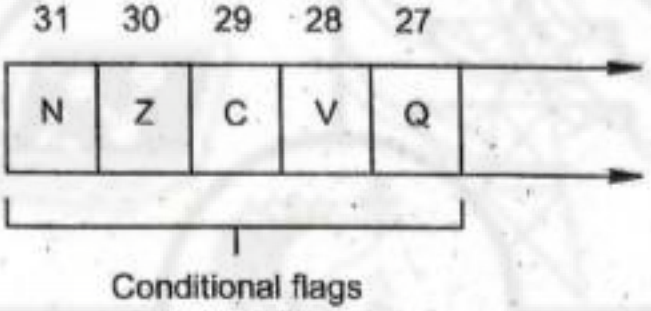
S.No.	Description	ARM	Thumb
i)	Instruction size	32 bit	16 bit
ii)	Conditional execution	Most instructions	Only branch instructions.
iii)	Data processing instructions	Access to barrel shifter and ALU	Separate barrel shifter and ALU instructions.
iv)	Program status register	Read-Write in privileged mode	No direct access
v)	Register usage	15 general purpose registers and program counter	8 general purpose registers + 7 high registers + program counter

Table 1.1

The Jazelle J and Thumb T bits in the CPSR reflect the state of the processor. When both J and T bits are '0', the processor is in ARM state and executes ARM instructions. This is the case when power is applied to the processor. When T bit is '1', then the processor is in Thumb state. To change states the core executes a specialized branch instruction.

The ARM designer introduced a third instruction set called Jazelle. Jazelle executes 8 bit instructions. It is a hybrid mix of the software and hardware designed to speed up the execution of Java bytecodes.

1.2.7. 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 zero flag in the CPSR is set. The representation of condition flags is shown in the table1.2.

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

Table 1.2

1.2.8. Exceptions, Interrupts and the vector table

1.2.8.1. Exceptions and Interrupts

Exception or interrupt is a control signal used for getting the immediate attention of processor. The description of various types of interrupts / exceptions handled by the processor is given below.

i) ***Reset vector*** is executed by the processor when power is applied. This instruction branches to the initialization code.

ii) ***Undefined instruction vector*** occurs when the processor cannot decode an instruction.

iii) ***Software interrupt vector*** occurs when executing a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.

iv) ***Prefetch abort vector*** occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.

v) ***Data abort vector*** occurs when an instruction attempts to access data memory without the correct access permissions.

vi) ***Interrupt request vector*** occurs when external hardware interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the CPSR.

vii) ***Fast interrupt request vector*** is similar to the interrupt request vector. It occurs when the hardware requires

faster response times. It can only be raised if FIQs are not masked in the CPSR.

1.2.8.2. 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

Table 1.3

When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the execution vector table shown in table1.3. Each vector table entry contains a form of branch instruction pointing to a start of a specific routine.

1.2.9. Endianness (Big endian and Little endian)

The term endianness refers to how bytes of a data word are ordered within memory. Endianness (or byte order) is also a big issue when reading data packets or compressed files.

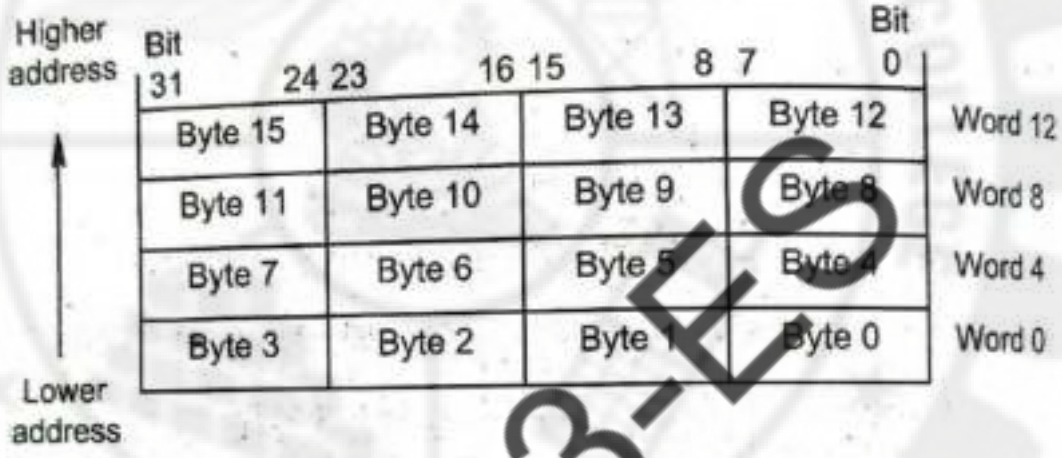
There are two methods used for storing data in ARM core. They are

i) Little endian, and

ii) Big endian

1.2.9.1. Little endian

In little endian configuration the least significant byte is placed at lower address. The memory organisation of little endian is shown below.



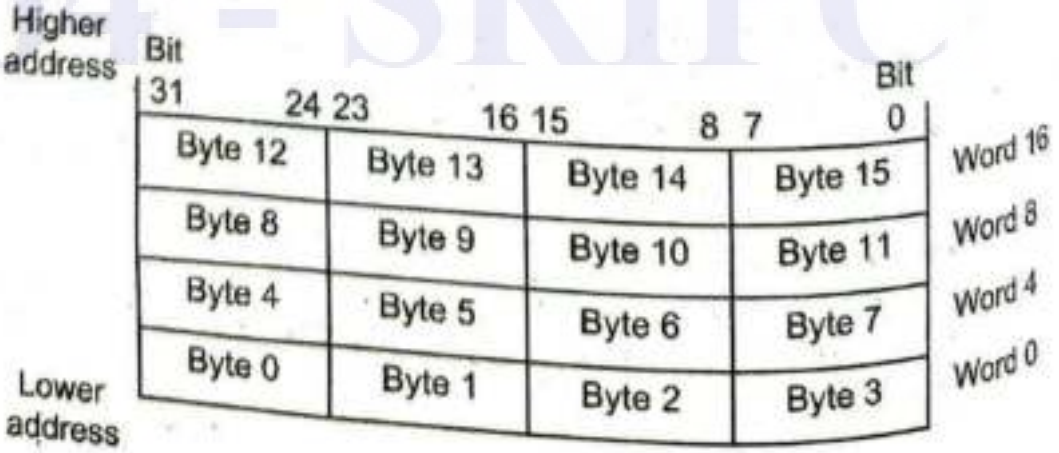
Advantages

i) Easy to place values

ii) Conversion from a 16 bit integer to a 32 bit integer address does not require any arithmetic.

1.2.9.2. Big endian

In big endian configuration the most significant byte is placed at lower address. The memory organisation of big endian is shown below.



Advantages

i) More natural.

ii) The sign of the number can be determined by looking at the byte at address offset 0.

iii) Strings and intergers are stored in the same order.

UNIT - II

ARM INSTRUCTION SET

2.1. INSTRUCTION SET

2.1.1. ARM Instruction Set - Introduction

ARM instructions process data held in registers and only access memory with load and store instructions. ARM instructions commonly contain two or three operands.

Thumb encodes a subset of the 32 bit ARM instructions into a 16 bit instruction set space. Thumb has higher performance than ARM on a processor with a 16 bit data bus, but lower performance than ARM on a processor with a 32 bit data bus. The thumb is used for memory constrained system.

Thumb has higher code density than ARM. Code density means the space taken up in memory by an executable program. Each Thumb instruction is related to a 32 bit ARM instruction.

ARM processors support six data types. They are,

i) 8 bit signed and unsigned bytes.

ii) 16 bit signed and unsigned half words.

iii) 32 bit signed and unsigned words.

All ARM instructions are 32 bit words and must be word aligned. Thumb instructions are half words and must be aligned on 2 byte boundaries. Internally all instructions

are on 32 bit operands. The shorter data types are only supported by data transfer instruction.

2.1.2. 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 we 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 with respect to bit 31 of the result. The Z flag is set if the result is zero.

2.1.2.1. Move instructions

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

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

MOV (move)	Move a 32-bit value into a register MOV Rd, N	Rd ⇐ N
MVN (move) negated	Move the NOT of the 32 bit value in to a register MVN Rd, N	Rd ⇐ ~ N

Usually N is a register Rm or a constant preceded by #.

Example: MOV r1, r2 ; (r1) ⇐ (r2)

This instruction moves the content of register *r2* into register *r1*.

Example: MVN r1, r2 ; (r1) ⇐ NOT (r2)

This instruction moves the complement of the content of register *r2* into register *r1*.

2.1.2.2. Barrel shifter

In an ordinary *MOV r1, r2* instruction, where N is a simple register *(r2)*. But N can be more than just a register or immediate value. If it can be a register Rm that has been preprocessed by a barrel shifter prior to being used by a data processing instruction.

Data processing instructions are processed within the arithmetic logic unit (ALU). A unique 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. The shift increases the power and flexibility of many data processing operations.

Some types of data processing instruction do not use the barrel shift. For example, the MUL (multiply), CLZ (count leading zeros) and QADD (signed saturated 32 bit add) instructions.

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

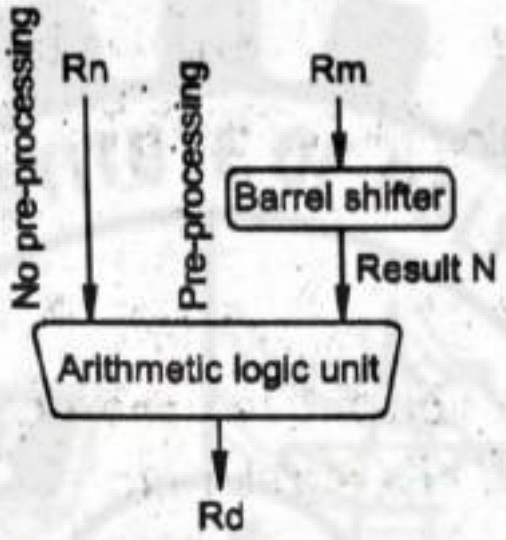


Fig.2.1 Barrel shifter and ALU

The illustration of barrel shifter and ALU is shown in the fig.2.1. Here the register Rn enters into the ALU without any preprocessing of registers, but the register Rm enters the ALU after barrel shift operations.

Example: MOV r1, r2, LSL # 2 ; (r1) $\Leftarrow 2^2$ x (r2)

This instruction logically shifts the value placed in register *r2* to left direction with 2 bit position. This instruction actually multiplies the register *r2* by 4 (= 2^2) and then places the result into register *r1*.

The different shift operations that we can use within barrel shifter are summarized below.

i) LSL:- Logical shift left by 0 to 31 bit places. Fill the vacated bits at the least significant end of the word with zeros.

ii) LSR:- Logical shift right by 0 to 32 bit places. Fill the vacated bits at the most significant end of the word with zeros.

iii) ASL:- Arithmetic shift left. This is a synonym for LSL.

iv) ASR:- Arithmetic shift right by 0 to 32 bit places. Fill the vacated bits at the most significant end of the word

with zeros if the source operand was positive or with ones if the source operand was negative.

v) ROR:- Rotate right by 0 to 32 bit places. The bits which fall off the least significant end of the word are used, in order to fill the vacated bits at the most significant end of the word.

vi) RRX:- Rotate right extended by 1 bit place. The vacated bit (bit 31) is filled with the old value of C flag and the operand is shifted one place to the right.

These shift operations are illustrated in the fig.2.2

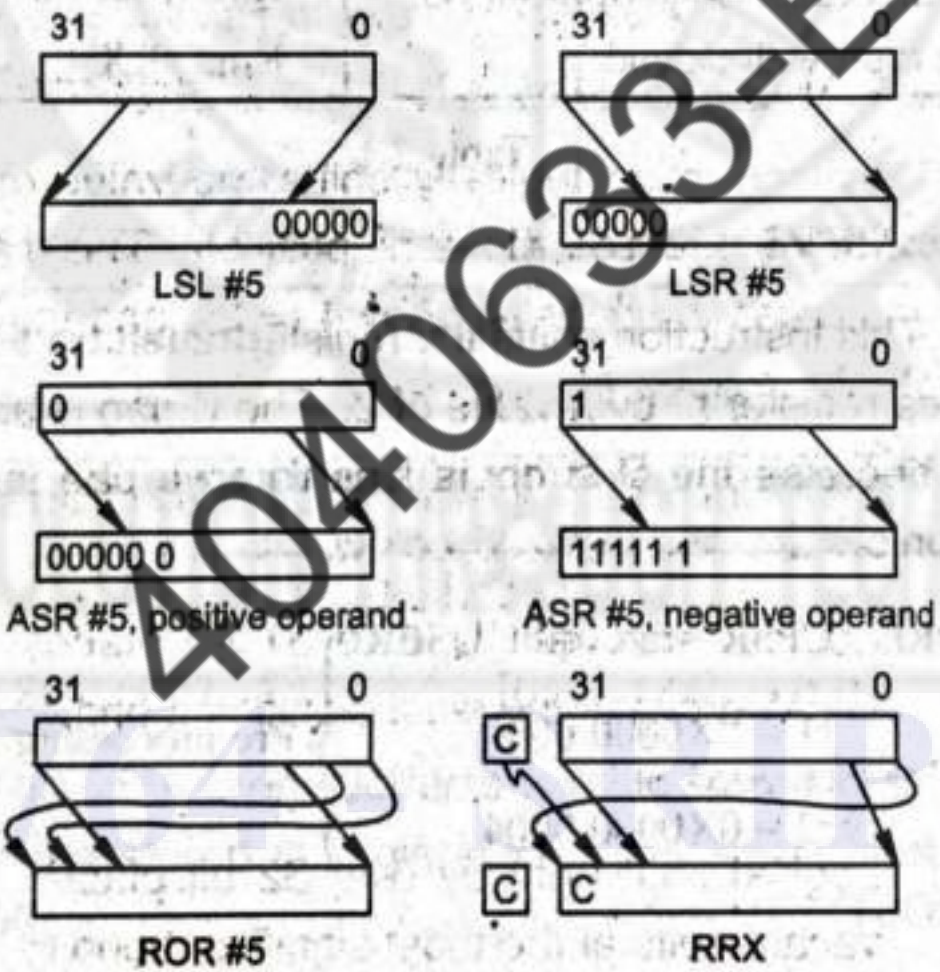


Fig.2.2 ARM shift operations

The syntax for Barrel shifter operation for data processing instructions is summarized in Table 2.1.

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

Table 2.1

Example: MOVS r1,r2, LSL #1

This instruction shifts the register r2 left by 1 bit. This multiplies register r2 by a value of 2. The C flag is updated in CPSR because the S suffix is presented in the instruction mnemonic.

PRE CPSR = nzcvqiFt_USER
r1 = 0X0000 0000
r2 = 0X0000 0004
} Pre processing

POST
CPSR = nzCvqiFt_USER
r1 = 0X0000 0008
r2 = 0X0000 0004
} Post processing

2.1.2.3. Arithmetic Instructions

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

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

N is the result of the shifter operation

The arithmetic instructions are described in the Table 2.2.

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

Table 2.2.

Example: SUB r1, r2, r3 ; [(r1) ⇐ (r2) – (r3)]

This instruction subtracts a value stored in register r3 from a value stored in register *r2*. The result is stored in register *r1*.

Example: RSB r1, r2, #2 ; [(r1) ⇐ 2 – (r2)]

This reverse subtract instruction subtracts *r2* from the constant value # 2, and stores the result in *r1*.

Example: SUBS r1, r1, #1 ;[(r1) ⇐ (r1) – 1]

This instruction is used for decrementing loop counters. This instruction subtracts 1 from the value placed in register *r1*, and also store the result in the same *r1* register.

2.1.2.4. Using the Barrel shifter with arithmetic instructions

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

Example: ADD r1, r2, r2, LSL #1. ; [(r1) ⇐ (r2) + (r2) x 2]

This instruction uses inline barrel shifter with an arithmetic instruction. This instruction actually multiplies the value stored in register *r2* by 3. The content of register *r2* is first shifted one location to the left, to give the value of twice *r2*. The ADD instruction then adds the result of the barrel shift operation to register *r2*. The final result is transferred into register *r1*, which is equal to 3 times the value stored in register *r2*.

2.1.2.5. Logical instructions

Logical instructions perform bitwise logical operation on the two source registers.

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

AND	logical AND of two 32-bit values	Rd = Rn & N
ORR	logical OR of two 32-bit values	Rd = Rn | N
EOR	logical exclusive OR of two 32-bit values	Rd = Rn ^ N
BIC	logical bit clear (AND NOT)	Rd = Rn & ~ N

Example: ORR r1, r2, r3 ; [(r1) ⇐ (r2) | (r3)]

This instruction logically OR-ed the content of registers *r2* and *r3*, and store the result in register *r1*.

Example : BIC r1, r2, r3 ; [(r1) ⇐ (r2) & (r3)]

It is a more complicated logical instruction, which carries out a logical bit clear. This instruction logically AND-ed the complement of content of register r3 with the content of r2, and the result is stored in register r1. This instruction is particularly used for clearing status bits, and is frequently used to change interrupt masks in CPSR.

2.1.2.6. Comparison instructions

The comparison instructions are used to compare or test a register with a 32 bit value. They update the CPSR flag bits according to the result, but do not affect other registers.

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 ^ N
TST	test bits of a 32-bit value	flags set as a result of Rn & N

N is the result of the shifter operation.

Example: CMP r1, r2 ; [(r1) – (r2)]

This instruction effectively subtracts the content of register r2 from the content of register r1, but the results are discarded. The condition flags are modified in accordance with the result.

2.1.2.7. Multiply Instructions

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

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

MUL {<cond>} {S} Rd, Rm, Rs

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

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

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

Example: MUL r1, r2, r3 ; [(r1) = (r2) x (r3)]

This instruction multiplies the values placed in registers *r2* and *r3*, and stores the result in *r1* register. The long multiply instructions (SMLAL, SMULL, UMLAL and UMULL) produce a 64 bit result. The result is stored in RdLo and RdHi registers. RdLo holds the lower 32 bits, and RdHi holds the higher 32 bits of the result.

Example: UMULL *r1, r2, r3, r4* ; [(r2), (r1) ⇐ (r3) x (r4)

This instruction multiplies the values placed in registers *r3* and *r4*, and the result is stored in registers *r2* (higher 32 bits) and *r1* (lower 32 bits).

2.1.3. Branch instructions

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

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

B	branch	PC= label
BL	branch with link	PC = label LR = address of the next instruction after the BL
BX	branch exchange	PC=Rm & 0xFFFFFFFE, T=Rm & 1
BLX	branch exchange with link	PC = label, T = 1 PC =Rm & 0XFFFFFFFE, T=Rm & 1 LR = address of the next instruction after the BLX

The address label is stored in the instruction as a signed PC relative offset. T refers to the Thumb bit in the CPSR. When instructions set T, the ARM switches to Thumb state.

Example: B forward

During the execution of this instruction, the control is transferred to the instruction which has a label as forward. This is an unconditional branch instruction.

Example: BL subroutine ; Branch to subroutine

This instruction is similar to the B instruction but overwrites the link register LR with a return address. It performs a subroutine call. When return from subroutine, the PC is copied from link register.

2.1.4. Load - store instructions

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

2.1.4.1. Single register transfer

These instructions are used for moving a single data in and out of a register. The data types supported are signed and unsigned words (32 bit), half words (16-bit) and bytes. The various types of load - store single register transfer instructions are described below.

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

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

STR {<cond>}H Rd, addressing

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 from 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 (mem 16[address])

Example: LDR r1, [r2] ; [(r1) ⇐ ((r2))]

This instruction loads register *r1* with the content of memory address specified by register *r2.*

Example: STR r1, [r2] ; [((r2)) ⇐ (r1)]

This instruction stores the content of register *r1* to the memory address pointed by register *r2.* The above instructions uses preindex method. The register *r2* is called base address register.

2.1.4.2. Single register load - store addressing mode

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the following indexing methods: preindex with write back, preindex and postindex, shown in the table 2.3.

Index method	Data	Base address register	Example
Preindex with writeback	mem[base+offset]	base+offset	LDR r1, [r2,#4]!
Preindex	mem[base+offset]	not updated	LDR r1,[r2,#4]
Postindex	mem[base]	base + offset	LDR r1, [r2], #4

Table 2.3 Index methods

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

(a) Pre index with writeback

Preindex with writeback calculates an address from the base register plus address offset and then updates that address base register with the new address. It is useful for transversing an array.

Example 1 : LDR r1, [r2, #4]!

This instruction uses preindexing with writeback method. In this instruction the content of register *r2* is incremented by 4. After that the content of incremented memory location specified by *r2* is moved to register *r1.*

(b) Pre index

The pre index offset is the same as the pre index with write back but does not update the address base register. It is useful for accessing an element in a data structure.

Example 2: LDR r1, [r2, #4]

This instruction uses preindexing method. In this instruction the content of memory location specified by *r2* added with 4 is moved to register *r1.* The content of *r2* is not changed.

(c) Post index

The post index only updates the address base register after the address is used. It is useful for transversing array.

Example 3: LDR r1, [r2], #4

This instruction uses postindexing method. In this instruction the content of memory location specified by register *r2* is moved to register *r1*. After that the content of register *r2* is incremented by 4.

A) Addressing modes - Load and store - 32 bit word/unsigned byte

The addressing modes available with a particular load and store instruction depend on the instruction class. The addressing modes with load and store of a 32 bit word or an unsigned byte are described in the table 2.4.

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

Table 2.4

i) A signed offset or register is denoted by "+/–", identifying that it is either a positive or negative offset from the base address register Rn.

ii) The base address register is a pointer to a byte in memory.

iii) The offset specifies a number of bytes.

iv) Immediate means the address is calculated using the base address register and a 12 bit offset encoded in the instruction.

v) Register means the address is calculated using the base address register and a specific register's contents.

vi) Scaled means the address is calculated using the base address register and a barrel shift operation.

An example of the different variations of the LDR instruction is shown in the table 2.5.

	Instruction	r0 =	r1 +=
Preindex with writeback	LDR r0, (r1, #0 x 4)! LDR r0, (r1, r2)! LDR r0, (r1, r2, LSR # 0 x 4)!	mem32(r1+0x4) mem 32 (r1 + r2) mem 32 [r1 + (r2 LSR 0 x 4)]	0 x 4 r2 (r2 LSR 0 x 4)
Preindex	LDR r0, [r1, # 0 x 4] LDR r0, (r1, r2) LDR r0, (r1, –r2, LSR # 0 x 4)	mem 32 (r1 + 0 x 4) mem 32 (r1 + r2) mem 32 [r1 – (r2 LSR 0 x 4)]	not updated not updated not updated
Postindex	LDR r0, (r1), # 0x4 LDR r0, (r1), r2 LDR r0, (r1), r2, LSR # 0 x 4	mem 32 (r1) mem 32 (r1) mem 32 (r1)	0 x 4 r2 (r2 LSR 0x4)

Table 2.5

B) Addressing mode - Load and store - 16 bit half word/signed byte

The addressing modes available on load and store instructions using 16 bit half word or signed byte data are shown in the table 2.6.

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

Table 2.6

These operations can not use the barrel shifter. There are no STRSB or STRSH instructions since STRH stores both a signed and unsigned halfword; similarly STRB stores signed and unsigned bytes. The variations of STRH instructions are shown in the table 2.7.

	Instruction	Result	r1 +=
Preindex with writeback	STRH r0, [r1, # 0 x 4]!	mem 16 [r1 + 0x4] = r0	0x4
	STRH r0,[r1, r2]!	mem 16 [r1 + r2] = r0	r2
Preindex	STRH r0, [r1, # 0 x 4]	mem 16 [r1 + 0x4] = r0	not updated
	STRH r0, [r1, r2]	mem 16 [r1 + r2] = r0	not updated

Postindex	STRH r0, [r1], #0 x 4	mem 16 [r1] = r0	0x4
	STRH r0, [r1], r2	mem 16 [r1] = r0	r2

Table 2.7 Variations of STRH instructions

(C) Load - Store multiple instructions

Load-store multiple instructions increase interrupt latency. ARM implementations do not usually interrupt instructions while they are executing.

Syntax :

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

LDM	Load multiple registers	{Rd}^N ⇐ mem 32 [start address + 4 x N] optional Rn updated
STM	Save multiple registers	{Rd}^N ⇒ mem 32 [start address + 4 x N] optional Rn updated

Addressing modes: Load-store multiple instructions

The different addressing modes for load-store multiple instructions are shown in the table 2.8.

Addressing mode	Description	Start Address	End address	Rn!
IA	increment after	Rn	Rn+4 x N–4	Rn+4 x N
IB	increment before	Rn + 4	Rn + 4 x N	Rn+4 x N
DA	decrement after	Rn – 4 x N+4	Rn	Rn–4 x N
DB	decrement before	Rn – 4 x N	Rn – 4	Rn–4 x N

Table 2.8: Addressing mode for load-store multiple instructions

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

Example 1: LDMIA r0!, {r1 – r3}

The register r0 is the base register Rn and is followed by ! indicating that the register is updated after instruction is executed. The "–" character is used to identify the range of registers. In this case the range is from registers *r1* to *r3* inclusive.

Each register can also be listed, using a comma to separate each register within "{" and "}" brackets.

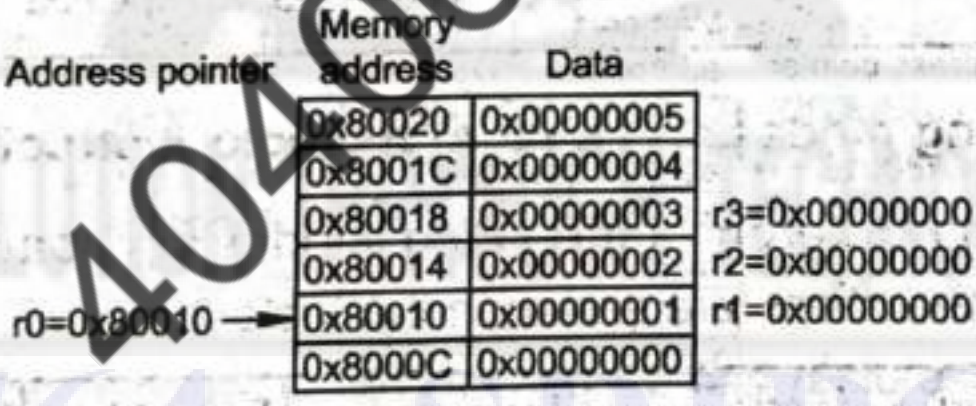


Fig.2.3 Pre-condition for LDMIA instruction

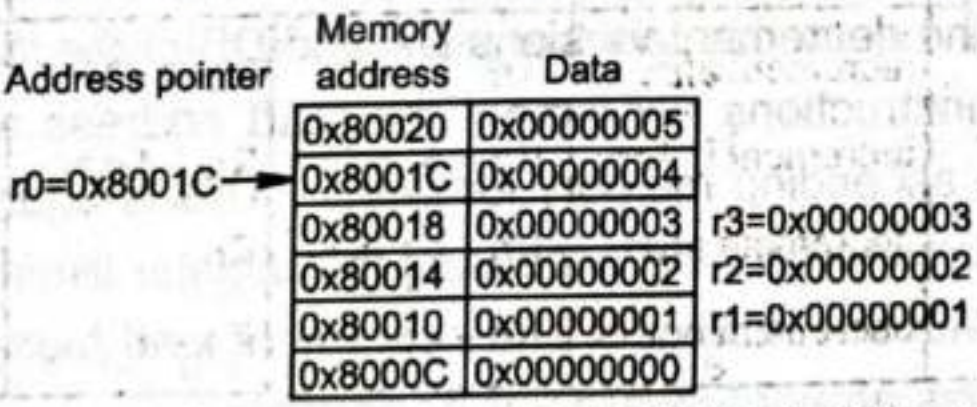


Fig.2.4 Post-condition for LDMIA instruction

The given instruction is a load multiple register increment after instruction. The graphical representation of this instruction at pre and post conditions is described in the fig 2.3 and 2.4.

The base register *r0* points to memory address 0x80010 in the pre condition. Memory address 0x80010, 0x80014 and 0x80018 contains the values of 1, 2 and 3 respectively. After the execution of load multiple instruction, the registers r1, r2 and r3 contain the values as shown in the fig.2.4.

Example 2: LDMIB r0!, {r1 – r3}

This is a load multiple register increment before instruction. By using the same pre condition, the first word pointed by register r0 is ignored and register r1 is loaded from the next memory location as shown in the fig.2.5.

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	

Fig.2.5 Post-condition for LDMIB instruction

The decrement versions DA and DB of the load-store multiple instructions decrement the start address and then store the ascending memory locations. This is equivalent to descending memory but accessing the register list in reverse order. With the increment and decrement load multiples, we can access arrays forwards or backwards.

A list of load-store multiple instruction pairs is shown in the table2.9. If we use a store with base update, then the paired load instruction of the same number of registers will reload the data and restore the base address pointer. This is used for saving a group of registers temporarily and restoring them later.

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

Table 2.9

Example: STMIB r0!, {r1 – r3}

This is an "STM increment before" instruction. This instruction stores the content of registers *r1, r2* and *r3* in memory locations. We can then corrupt registers.

Example: LDMDA r0!, {r1 – r3}

This is "LDM decrement after" instruction. This instruction reloads the original values and restores the base pointer *r0*.

The functions of above two instructions are illustrated below.

```
PRE     r0  =  0 x 00009000
        r1  =  0 x 00000009
        r2  =  0 x 00000008
        r3  =  0 x 00000007
```

```
        STMIB   r0!, {r1 – r3}
        MOV     r1, #1
        MOV     r2, #2
        MOV     r3, #3
PRE (2) r0  =   0 x 0000900C
        r1  =   0 x 00000001
        r2  =   0 x 00000002
        r3  =   0 x 00000003
        LDMDA   r0!, {r1 – r3}
POST    r0  =   0 x 00009000
        r1  =   0 x 00000009
        r2  =   0 x 00000008
        r3  =   0 x 00000007
```

Example: LDMIA r9!, {r0 – r7}

This instruction is a load-store multiple instruction. This is a simple routine that copies blocks of 32 bytes from source address location to a destination address location.

This instruction loads the data pointed by register *r9* into registers *r0* to *r7*. It also updates r9 to point to the next block of data to be copied.

Example : STMIA r10!, {r0 – r7}

This instruction copies the contents of registers *r0* to *r7* to the destination memory address pointed by register *r10*. It also updates *r10* to point to the next destination location.

```
Example :       CMP r9, r11     ; Compare (r9) and (r11)
                BNE loop        ; Branch not equal
```

CMP and BNE compare pointers *r9* and *r11* to check whether the end of the block copy has been reached. If the block copy is complete, then the routine finshes, otherwise the loop repeats with the updated values of registers *r9* and *r11*.

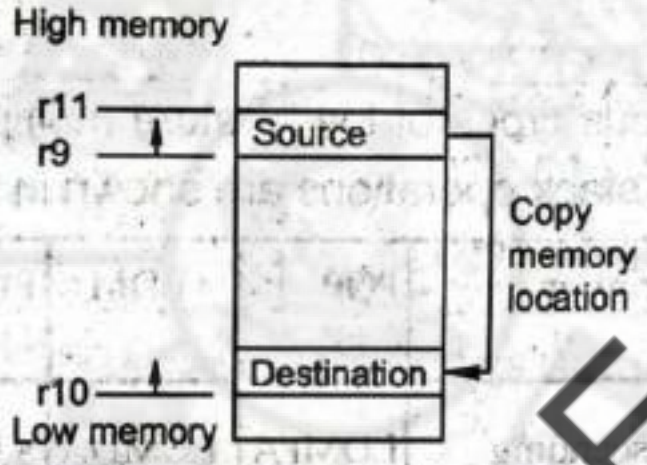


Fig.2.6 Block memory copy in the memory map

The memory map of the block memory copy and how the routine moves through memory is shown in the fig.2.6. Theoretically this loop transfers 32 bytes (8 words) in two instructions.

A) Stack operations

The ARM architecture uses the load-store multiple instructions to carry out stack operations. The POP operation (removing data from a stack) uses a load multiple instruction. Similarly the PUSH operation (placing data on to the stack) uses a store multiple instruction.

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

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

The various types of load - store multiple addressing mode to support stack operations are shown in the table 2.10.

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

Table 2.10: Addressing modes for stack operations

The LDMFD and STMFD instructions provide the POP and PUSH functions respectively.

Example: STMFD sp!, {r1, r4}

This instruction pushes registers onto the stack, updating SP. A push onto a full descending stack is described in the fig.2.7. When the stack grows, the stack pointer points to the last full entry in the stack.

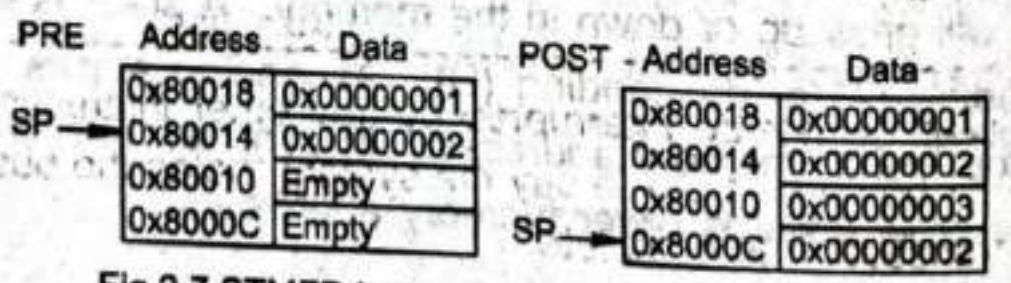


Fig.2.7 STMFD instruction-full stack push operation

Example: STMED sp!, {r1, r4}

This instruction is a push operation on any empty stack. This instruction pushes the registers onto the stack but updates register SP to point to the next empty location. The functions are described in the fig.2.8.

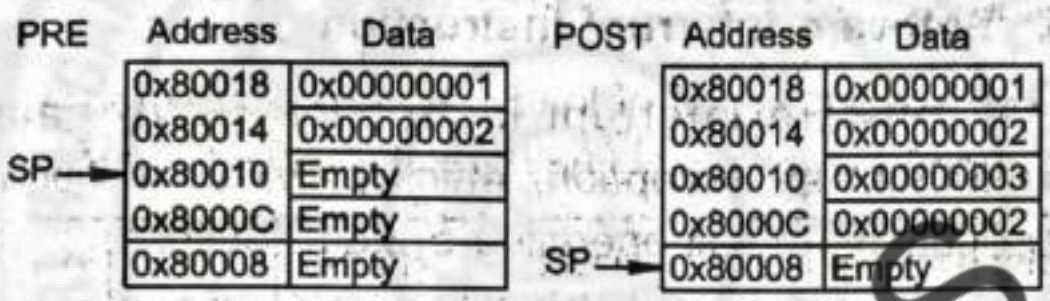


Fig.2.8 STMED instruction-empty stack push operation

B) SWAP instruction

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

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

SWP	swap a word between memory and a register	tmp = mem 32 [Rn] mem 32 [Rn] = Rm Rd = tmp
SWPB	swap a byte between memory and a register	tmp = mem 8 [Rn] mem 8 [Rn] = Rm Rd = tmp

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

Example: SWP r0, r1, [r2]

This instruction loads register r0 with the content of memory location specified by register *r2*, and also moves (stores) the content of register *r1* to the memory location specified by *r2*.

2.1.5. Software interrupt instruction

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

Syntax : SWI {<cond>} SWI_number

SWI	Software interrupt	LR_svc = address of instruction following the SWI SPSR_svc = CPSR PC = vectors + 0 x 8 CPSR mode = SVC CPSR I = 1 (mask IRQ interrupts)

When the processor executes an SWI instruction, it sets the program counter PC to the offset 0X8 in the vector table. The instruction also forces the processor mode to SVC, which allows an operating system routine to be called in a privileged mode.

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

Example: 0X0000 8000 SWI 0x123456

This is a SWI instruction with SWI number 0X123456 used by ARM toolkits as a debugging SWI. Typically the SWI instruction is executed in user mode.

The content of *CPSR*, *SPSR*, *PC*, *LR* and *r0* at preprocessing and post processing is illustrated below.

```
PRE   CPSR = nzcVqift_USER
      PC = 0x0000 8000
      LR = 0x003FFFFF;      LR = r14
      r0 = 0x12
POST  CPSR = nzcVqIft_SVC
      SPSR = nzcVqift_USER
      PC  = 0 x 0000 0008
      LR  =  0x0000 8004
      r0  = 0x12
```

Since SWI instructions are used to call operating system routines, we need some form of parameter passing. This is achieved using registers. In this example, *r0* is used to pass the parameter 0X12. The return values are also passed back via registers.

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

The SWI number is determined by

SWI_number = <SWI instruction> AND NOT (0XFF000000)

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

2.1.6. Program status register instructions

The ARM instruction set provides two instructions to directly control a program status register (PSR).

Syntax:

MRS {<cond>} Rd, <CPSR | SPSR>
MSR {<cond>} <CPSR | SPSR> <fields>, Rm
MSR {<cond>} <CPSR | SPSR>_<fields>, # immediate

The MRS instruction transfers the contents of either the CPSR or SPSR into a register; in 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.

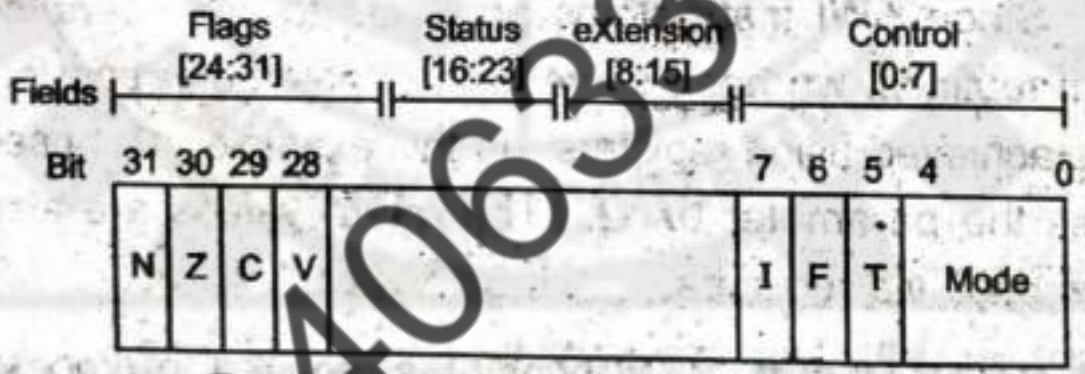


Fig.2.9 PSR byte fields

The label called "fields", can be any combination of control (c), extension (x), status (s) and flags (f). These fields related to particular byte regions in a PSR are shown in the fig.2.9.

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

The 'c' field controls the interrupt masks, Thumb state and processor mode.

Example

```
        MRS r1, CPSR
        BIC r1, r1, #0X80          ; 0B0100 0000
        MSR CPSR_c, r1
PRE     CPSR = nzcvqIFt_SVC
POST    CPSR = nzcvqiFt_SVC
```

The MRS first copies the CPSR into register *r1.* The BIC instruction clears bit 7 of *r1.* Register *r1* is then copied back into CPSR, which enables IRQ interrupts.

This example is in SVC mode. In user mode we can read all CPSR bits, but we can only update the condition flag field "F".

2.1.7. Loading constants

There is no ARM instruction to move a 32 bit constant into a register. Since ARM instructions are 32 bits in size, they obviously can not specify a general 32 bit constant.

There are two pseudo instructions to move a 32 bit value into a register.

Syntax

```
LDR Rd, = constant
ADR Rd, label.
```

LDR	load constant pseudo instruction	Rd = 32 bit constant
ADR	load address pseudo instruction	Rd = 32 bit relative address

The first pseudo instruction writes a 32 bit constant to a register. The second pseudo instruction writes a relative address into a register which will be encoded using a PC- relative expression.

2.1.8. Conditional execution

Most ARM instructions are conditionally executed. This instruction only executes if the condition code flags pass a given condition or test. By using conditional execution instructions, we can increase performance and code density.

The condition field is a two-letter mnemonic appended to the instruction mnemonic. The default mnemonic is AL or always execute.

Conditional execution reduces the number of branches, which also reduce the number of pipeline flushes and thus improves the performance of the executed code. Conditional execution depends upon two components. They are condition field and condition flags. The condition field is located in the instruction and the condition flags are located in the CPSR.

Example: ADDEQ r1, r2, r3

This is an ADD instruction with the EQ condition appended. This instruction will only be executed when the zero flag in the CPSR is set to 1.

(r1) = (r2) + (r3) if zero flag is set.

Only comparing instructions and data processing instructions with the 'S' suffix appended to the mnemonic update the condition flags in the CPSR.

2.2. THUMB INSTRUCTIONS

Thumb encodes a subset of the 32 bit ARM instructions into a 16 bit instruction set space. Each thumb instruction is related to a 32 bit ARM instruction. In thumb state, the higher registers *r8* to *r12* are only accessible with MOV, ADD or CMP instructions. CMP and all the data processing instructions that operate on low registers update the conditional flags in the CPSR.

2.2.1. Thumb instruction set

Thumb instruction is a compressed form of ARM instruction set. The T (bit 5) bit of CPSR decides whether the ARM processor will execute thumb instructions. If T is set the processor executes 16 bit thumb instructions otherwise it executes 32 bit ARM instructions. Thumb encodes a subset of 32 bit ARM instructions into a 16 bit instruction set space. Thumb has higher performance than ARM on a processor with a 16 bit data but lower performance than ARM on a 32 bit data bus. Each thumb instruction is related to a 32 bit ARM instruction.

2.2.2. ARM Thumb similarities

i) The load - store architecture with data processing, data transfer and control flow instructions.

ii) Support for 8 bit byte, 16 bit half word and 32 bit word data types.

iii) A 32 bit unsegmented memory.

2.2.3 Differences between Thumb and ARM

S.No	Thumb	ARM
i)	Most instructions are executed unconditionally	All instructions are executed conditionally
ii)	Many instructions use a 2 address format (destination register, source register)	Many instructions use a 3 address format
iii)	Less regular than ARM instructions	Regular instructions

2.2.4 Usage of thumb register

In thumb state we have to access all registers. Only the low registers r0 to r12 are fully accessible. The higher registers r8 to r12 are only accessible with MOV, ADD or CMP instructions. CMP and all the data processing instructions that operate on low registers update the conditional flags in CPSR. The summary of thumb register usage is shown in the table 2.11

Registers	Access
r0 - r7	fully accessible
r8 - r12	Only accessible by MOV, ADD and CMP
r13 SP	Limited accessible
r14 LR	Limited accessible
r15 PC	Limited accessible
CPSR	Only indirect access
SPSR	No access

Table 2.11 Summary of thumb register usage

2.2.5. Data processing instructions

The data processing instructions manipulate data within registers. They include move instructions, arithmetic instructions, shifts, logical instructions, comparison instructions and multiply instructions.

2.2.5.1. Move instructions

Move instructions are used to move a 32 bit value or logical NOT of a 32 bit value into a register.

Syntax: <instruction> Rd, N

MOV (move)	Move a 32 bit value placed in register Rm to Rd MOV Rd, Rm	Rd ⇐ Rm
MOV (move)	Move an immediate 32 bit value into register Rd MOV Rd, # immediate	Rd ⇐ immdiate
MVN (move negated)	Move the NOT of the 32 bit value placed in Rm into Rd	Rd ⇐ NOT Rm

N is a register or a constant preceded by #.

2.2.5.2. Arithmetic instructions

The arithmetic instructions implement addition and subtraction of 32 bit values. N is a register or 32 bit value.

ADC	add two 32 bit values and carry: ADC Rd, Rm	Rd ⇐ Rd + Rm + carry

ADD	add two 32 bit values
	ADD Rd, N
	ADD Rd, Rn, N
		
		Rd ⇐ Rd + N
		Rd ⇐ Rn + N

SBC	subtract with carry a 32 bit value SBC Rd, Rm	Rd ⇐ Rd – Rm – NOT (C flag)

SUB	subtract two 32 bit values
	SUB Rd, N
	SUB Rd, Rn, N
		Rd ⇐ Rd – N
		Rd ⇐ Rn – N

NEG	negate a 32 bit value NEG Rd, Rm	Rd ⇐ 0 – Rm

2.2.5.3. Logical instructions

Logical instructions perform bitwise logical operations on the two source operands.

Syntax: <instruction> Rd, Rm

AND	logical AND of two 32 bit values: AND Rd, Rn	Rd ⇐ Rd & Rn
ORR	logical OR of two 32 bit values: ORR Rd, Rn	Rd ⇐ Rd | Rn
EOR	logical Ex-OR of two 32 bit values: EOR Rd, Rn	Rd ⇐ Rd ^ Rn
BIC	logical bit clear (AND NOT) of two 32 bit values: BIC Rd, Rn	Rd ⇐ Rn & NOT Rn

2.2.5.4. Comparison instructions

The comparison instructions are used to compare or test two registers (or immediate value). They update the CPSR flag bits according to the result but do not affect registers.

Syntax:<CMP> Rn, N

<CMN | TST> Rn, Rm

(N is a register or immediate value)

CMP	Compare two 32 bit integers	Flags set as a result of Rn – N
CMN	Compare negative two 32 bit values.	Flags sets as a result of Rn + Rm
TST	Test bits of a 32 bit value.	Flags set as a result of Rn & Rm

2.2.5.5. Multiply instructions

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

MUL	Multiply two 32 bit values MUL Rd, Rm	Rd = (Rm * Rd) [31:0]

2.2.5.6. Shift instructions

Shift instructions are used to shift or rotate the 32 bit value placed in registers to either left or right direction, according to the type of instruction.

ASR	arithmetic shift right ASR Rd, Rm, # immediate	Rd ⇐ Rm >> immediate C flag ⇐ Rm [immediate – 1]
	ASR Rd, Rs	Rd ⇐ Rd >> Rs, C flag ⇐ Rd [Rs – 1]

LSL	logical shift left LSL Rd, Rm, # immediate	Rd ⇐ Rm << immediate C flag ⇐ Rm [32 - immediate]
	LSL Rd, Rs	Rd ⇐ Rd << Rs, C flag ⇐ Rd [32 - Rs]
LSR	logical shift right LSR Rd, Rm, #immediate	Rd ⇐ RM >> immediate, C flag ⇐ Rd [immediate–1]
	LSR Rd, Rs	Rd ⇐ Rd >> Rs, C fly ⇐ Rd [Rs – 1]
ROR	rotate right a 32 bit value ROR Rd, Rm, # immediate	Rd ⇐ Rd RIGHT_ROTATE Rs C flag ⇐ Rd [Rs – 1]

2.2.6. Branch instructions

There are two variations of branch instructions used in thumb. The branch instructions change the flow of execution or is used to call a routine.

B	branch B label	PC⇐ label
BL	branch with link BL label	PC ⇐ label LR ⇐ (instruction address after the BL) + 1

The first instruction is similar to the ARM version and is conditionally executed, its branch range is limited to a signed 8 bit immediate or – 256 to + 254 bytes. The second version removes the conditional part of the instruction, its range is in between – 2048 and + 2046 bytes.

2.2.7. Load - Store Instructions

Load and store instructions are used for loading data to register or storing data from register.

2.2.7.1. Single register load store instructions

These instructions use two pre indexed addressing modes. They are;

i) Offset by register: It uses a base register Rn plus the register offset Rm.

ii) Offset by immediate: It uses a base reigster Rn plus a 5 bit immediate or a value dependent on the data size.

Syntax:

<LDR | STR>{<B | H>} Rd, [Rn, #immediate]
LDR{<H | SB | SH>} Rd, [Rn,Rm]
STR{<B | H>} Rd, [Rn, Rm]
LDR Rd, [pc, #immediate]
<LDR | STR> Rd, [sp, #immediate]

LDR	load word into a register	Rd ⇐ mem32 [address]
STR	save 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])

Example

```
PRE     mem32[0x90000] = 0x00000001
        mem32[0x90004] = 0x00000002
        mem32[0x90008] = 0x00000003
        r0 = 0x00000000
        r1 = 0x00090000
        r2 = 0x00000004

        LDR r0, [r1, r4] ; register

POST    r0 = 0x00000002
        r1 = 0x00090000
        r4 = 0x00000004

        LDR r0, [r1, #0x4] ;immediate

POST    r0 = 0x00000002
```

Both instructions carry out the same operation. The first instruction uses register offset and the second one uses immediate offset.

In the first instruction, the content of array of memory locations specified by the sum of registers r1 and r4 is loaded in register r0.

In the second instruction, the content of array of memory locations specified by register r1 with the sum of immediate value is loaded in register r0.

2.2.7.2. Multiple register load - store instructions

The thumb versions of the load - store multiple instructions are reduced forms of the ARM load store multiple instructions. They only support the increment after (IA) addressing mode.

Syntax:

<LDM | STM>IA Rn!, {low Register list}

LDMIA	load multiple registers	{Rd}^{*N} ⇐ meme32 [Rn + 4 *N], Rn = Rn + 4 * N
STMIA	save multiple registers	{Rd}^{*N} ⇒ mem32 [Rn + 4 *N], Rn = Rn + 4 * N

Here N is the number of registers in the list of registers.

Example:

```
PRE   r1 = 0x00000001
      r2 = 0x00000002
      r3 = 0x00000003
      r4 = 0x9000
      STMIA r4!, {r1, r2, r3}
POST  mem32 [0x9000] = 0x00000001
      mem32 [0x9004] = 0x00000002
      mem32 [0x9008] = 0x00000003
      r4 = 0x900c
```

This example saves registers r1 to r3 to memory addresses 0X9000 to 0X900C. It also updates base register r4.

2.2.8. Stack Instructions

The Thumb stack operations are different from the equivalent ARM instructions because they use the more traditional POP and PUSH concept.

Syntax : POP {low_register_list{, pc}}

PUSH {low_register_list {, lr}}

POP	POP registers from the stack	$Rd^{*N} \Leftarrow$ mem 32 [SP + 4 x N], SP = SP + 4*N
PUSH	PUSH registers on to the stack	$Rd^{*N} \Rightarrow$ mem 32[SP + 4 x N], SP = SP – 4 x N

There is no stack pointer specified in these instructions. This is because the stack pointer is fixed at register *r13* in Thumb operations and SP is automatically updated. The list of registers is limited to the low registers *r0* to *r7*.

The PUSH register list also can include the link register LR. Similarly the POP register list can include the PC. This provides support for subroutine entry and exit.

Example

```
; Call subroutine
BL    ThumbRoutine
; continue
ThumbRoutine
PUSH  {r1, LR}          ; enter subroutine
MOV   r0, #2
POP   {r1, PC}          ; return from subroutine
```

This example uses POP and PUSH instructions. The subroutine Thumb Routine is using a branch with link (BL) instruction.

The link register LR is pushed onto the stack with register r1. Upon return, register r1 is popped off the stack as well as the return address being loaded into the PC. This returns from the subroutine.

2.2.9. Software Interrupt Instruction

Similar to the ARM equivalent, the Thumb software interrupt (SWI) instruction causes a software interrupt exception. If any interrupt or exception flag is raised in Thumb state, the processor automatically reverts back to ARM state to handle the exception.

Syntax: SWI interrupt

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) CPSR T = 0 (ARM state)

The Thumb SWI instruction has the same effect and nearly the same syntax as the ARM equivalent. It differs in that the SWI number is limited to the range 0 to 255 and it is not conditionally executed.

Example: 0X0000 8000 SWI 0X45

This example shows the execution of a Thumb SWI instruction. Note that the processor goes from Thumb state to ARM state after execution.

It's pre and post processing responses are shown below.

```
PRE   CPSR =nzcVqifT_USER
      PC = 0x00008000
      LR = 0x003FFFFF      ;LR = r14
      r0 = 0x12
POST  CPSR = nzcVqIft_SVC
      SPSR = nzcVqifT_USER
      PC = 0x00000008
      LR = 0x00008002
      r0 = 0x12
```

2.3. Simple Programs

2.3.1. To write an assembly language program for adding two values.

```
LDR r1, value 1 ; Load the first number in r1
LDR r2, value2  ; Load the second number in r2
ADD r0, r1, r2  ; Add the two numbers
LDR r4, address ; Load the address for storing the result
STR r0, [r4]    ; Store the result
END             ; End
```

2.3.2. To write an assembly language program for adding two 32 bit values placed in memory locations, and store the result in other memory locations

```
LDR r0, 0x00009000 ; Load the address of first data.
LDR r1, [r0]       ; Place the first data in r1
```

```
LDR r0, r0, #0x4    ; Adjust the pointer for second data
LDR r2, [r0]        ; Place the second data in r2
ADD r3, r1, r2      ; Add the two data
LDR r0, r0 #0x4     ; Adjust the pointer for storing data
STR r3, [r0]        ; Store the result
END
```

2.3.3. To write an assembly language program for subtracting two 32 bit values

```
LDR r1, value 1 ; Load the first data in r1
LDR r2, value 2 ; Load the second data in r2.
SUB r0, r1, r2  ; Subract the two values.
                          (r0) ⇐ (r1) – (r2)
LDR r4, address; Load the address for storing the result
STR r0, [r4]    ; Store the result
END             ; End
```

2.3.4. To write an assembly language program for multiplying two values.

```
LDR r1, value1      ; Load the first number in r1
LDR r2, value 2     ; Load the second number in r2
MUL r0, r1, r2      ; Multiply the two values
LDR r4, address     ; Load the address for storing the
                      result
STR r0, [r4]        ; Store the result.
END
```

2.3.5. To write an assembly language program for multiplying two values placed in memory locations and store the result at another memory locations

```
LDR r0, 0X00008000 ; Load the address of data in r0
LDR r1, [r0], #4   ; Place the first data in r1
LDR r2, [r0], #4   ; Place the second data in r2
UMULL r3, r4, r1, r2 ; Multiply the two values
STR r4, [r0], #4   ; Store the result (RdLo)
STR r3, [r0]       ; Store the result(RdHi)
END
```

2.3.6. To write an assembly language program for adding two values placed in memory locations and store the result at another memory locations

```
      LDR r0, 0X00009000   ; Load the address of data in r0
      LDR r1, [r0], #4     ; Place the first data in r1
      LDR r2, [r0], #4     ; Place the second data in r2
      LDR r4, 0X0000 0000  ; Initialise r4 for carry
      ADDS r5, r1, r2      ; Add the two values
      BCC LOOP             ; If no carry, jump to LOOP
      ADD r4, r4, #1       ; Set carry as 1
LOOP  STR r5, [r0], #4     ; Store the result
      STR r4, [r0]         ; Store the carry
      END
```

2.4. ARM Instructions

Mnemonics	Description
ADC	add two 32-bit values and carry
ADD	add two 32-bit values
AND	logical bitwise AND of two 32-bit values
B	branch relative +/- 32 MB
BIC	logical bit clear (AND NOT) of two 32-bit values
BKPT	breakpoint instructions
BL	relative branch with link
BLX	branch with link and exchange
BX	branch with exchange
CDP CDP2	coprocessor data processing operation
CLZ	count leading zeros
CMN	Compare negative two 32-bit values
CMP	compare two 32-bit values
EOR	logical exclusive OR of two 32-bit values
LDC LDC2	load to coprocessor single or multiple 32-bit values
LDM	load multiple 32-bit words from memory to ARM registers
LDR	load a single value from a virtual address in memory
MCR MCR2 MCRR	move to coprocessor from an ARM register or registers
MLA	multiply and accumulate 32-bit values
MOV	move a 32-bit value into a register

MRC MRC2 MRRC	move to ARM register or registers from a coprocessor
MRS	move to ARM register from a status register (CPSR or SPSR)
MSR	move to a status register (CPSR or SPSR) from an ARM register
MUL	multiply two 32-bit values
MVN	move the logical NOT of 32-bit value into a register
ORR	logical bitwise OR of two 32-bit values
PLD	preload hint instuction
QADD	signed saturated 32-bit add
QDADD	signed saturated double and 32-bit add
QDSUB	signed saturated double and 32-bit subtract
QSUB	signed saturated 32-bit subtract
RSB	reverse subtract of two 32-bit values
RSC	reverse subtract with carry of two 32-bit integers
SBC	subtract with carry of two 32-bit values
SMLAxy	signed multiply accumulate instructions (16 x 16) + 32 = 32 bit)
SMLAL	signed multiply accumulate long ((32 x 32) + 64 = 64 bit))
SMLALxy	signed multiply accumulate long ((16 x 16) + 64 = 64 bit)
SMLAWy	signed multiply accumulate instruction (((32 x 16) >> 16) + 32 = 32 bit)
SMULL	signed multiply long (32 x 32 = 64 bit)

SMULxy	signed multiply instructions (16 x 16 = 32-bit)
SMULWy	signed multiply instruction ((32 x 16) >> 16 = 32 bit.)
STC STC2	store to memory single or multiple 32-bit values from coprocessor
STM	store multiple 32-bit registers to memory
STR	store register to a virtual address in memory
SUB	subtract two 32bit values
SWI	software interrupt
SWP	swap a word/byte in memory with a register, without interruption
TEQ	test for equality of two 32-bit values
TST	test for bits in a 32-bit value
UMLAL	unsigned multiply accumulate long ((32 x 32) + 64 = 64-bit)
UMULL	unsigned multiply long (32 x 32 = 64-bit)

UNIT - I
INTRODUCTION TO EMBEDDED SYSTEM AND ARM PROCESSOR

1.1. Embedded System

1.1.1. Definition of Embedded System

A system is a way of working, organizing or doing one or more tasks according to a fixed plan, program or set of rules. A system is also an arrangement in which all its units assemble and work together according to the plan or program.

For example, watch is a time display system. Its parts are its hardware, needles, battery with the dial, chasis and strap. These parts organize to show the real time every second and continuosly update the time every second. Washing machine is an automatic clothes washing system.

An embedded system is a combination of computer hardware and software designed for a specific function. This system may be a specific part of an application or product or a part of a larger system. These systems are electronic systems that contain a microprocessor or microcontroller.

1.1.2. Features of Embedded system

i) Embedded systems are the modern computer devices with multifunction capabilities.

ii) An embedded system performs a specific function or a set of specific functions.

iii) Embedded systems are not always independant (standalone) devices. Embedded systems form smaller parts of a much larger device that carries out a specific task.

iv) The program instructions written for embedded systems are referred to as firmware.

v) The programs are stored in ROM or flash memory.

vi) They run with limited computer hardware resources: little memory, small or non-existent keyboard and/or screen.

vii) Embedded systems possess advanced graphical interfaces.

viii) Simple embedded system uses LEDs, buttons or LCD displays with simple menu options.

1.1.3. Characteristics of Embedded system

i) Embedded systems do a very specific task.

ii) Embedded systems have very limited resources, particularly the memory.

iii) Embedded systems are created to perform the task within a certain time frame.

iv) They have minimal or no user interface (UT).

v) Embedded systems have to work against some deadlines.

vi) Some embedded systems are designed to react to external stimuli and react accordingly.

vii) Embedded system cannot be changed or upgraded by the users.

viii) Some embedded systems have to operate in extreme environmental conditions such as very high temperatures and humidity.

ix) Embedded systems need to be highly reliable.

1.1.4. Types of Embedded systems

A) Based on the performance and functional requirements the embedded systems can be classified as follows.

a) Real time embedded systems: It provides output within a defined specific time. It can be further classified as

i) Soft real time embedded systems, and

ii) Hard real time embedded systems.

b) Standalone embedded systems: They can work themselves i.e they are self sufficient and do not depend on a host system.

c) Networked embedded systems: It depends on connected network to perform its assigned works. These systems consist of components like sensors, controllers etc., which are interconnected.

d) Mobile embedded systems: These are small sized and can be used in smaller devices. They are used in mobile phones and digital cameras.

B) Based on the performance of microcontroller the embedded systems can be classified as follows,

i) Small scale embedded systems

ii) Medium scale embedded systems

iii) Sophisticated embedded systems.

1.1.5. List of embedded system devices

a) Digital alarm clocks

b) Electronic parking meters

c) Robotic vacuum cleaners

d) Smart watches

e) Washing machines and dish washers

f) Home security systems

g) Air-conditioners

h) Electric stoves, pressure cookers, tea/coffee machines.

i) Traffic lights

j) Vending machines

k) Fire alarms and carbon monoxide detectors

l) Fax machines and scanners

m) Digital and video cameras

n) Calculators

o) Digital thermometers

p) Motion sensors

q) Handheld computers

r) Electronic toys

s) Wi-Fi routers

t) Heart rate monitors

u) Automobile systems

1.1.6. Hardvard architecture

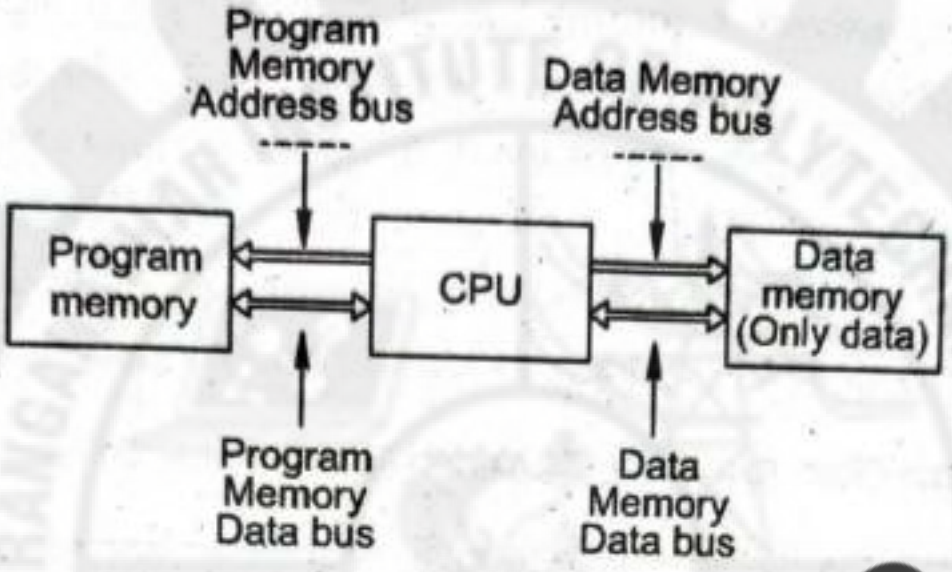


Fig.1.1 Harvard Architecture

The representation of Harvard architecture is shown in the fig 1.1. In this architecture there are two separate memory blocks. One is program memory and the other is data memory. Program memory stores only instructions and data memory stores only data. Two pairs of data buses are used between the CPU and the memory blocks. The address and data buses of program memory are used to access the program memory. The address and the data buses of data memory are used to access data memory. Certainly this architecture is much more efficient because accessing the instructions and data will be very fast.

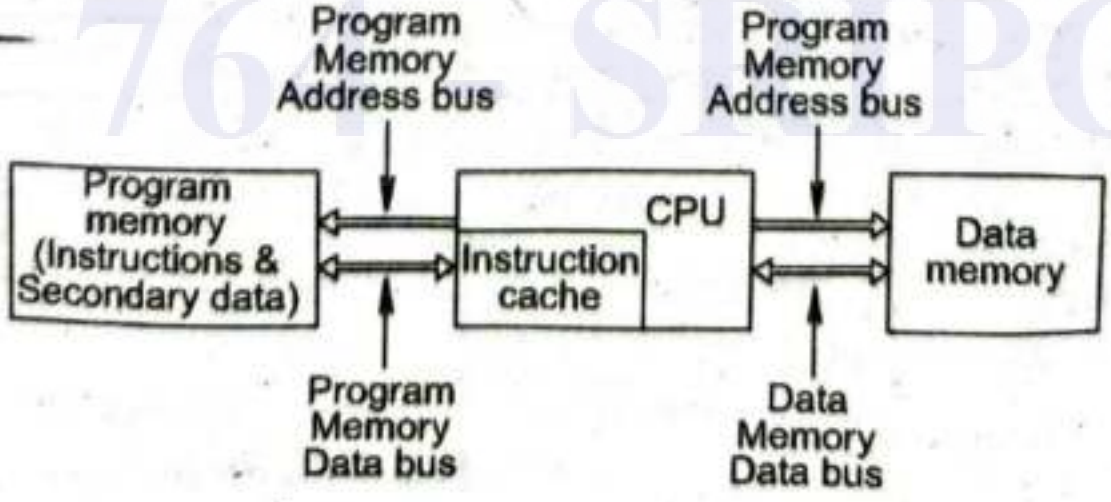


Fig.1.2 Super- Harvard Architecture

The representation of super harvard architecture (SHARC) is shown in the fig 1.2. It is a slight but significant modification of the Harvard architecture. In harvard architecture, the data memory is accessed more frequently than the program memory. Therefore in SHARC provision has been made to store some secondary data in the program memory to balance to load on both memory blocks.

1.1.7. Von-Neumann architecture

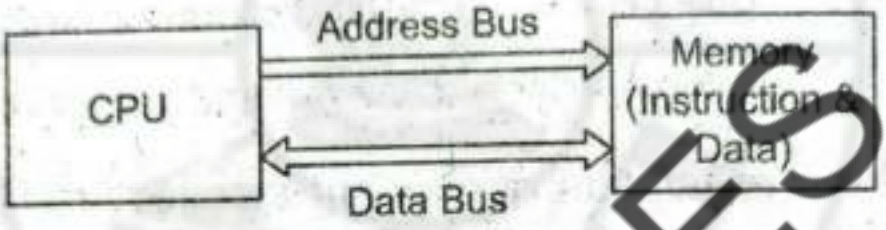


Fig.1.3 Von Neumann Architecture

The representation of Von-Neumann architecture is shown in the fig 1.3. It is the most widely used architecture. This architecture has one memory chip which stores both instructions and data. The processor interacts with the memory through address and data buses to fetch instructions and data.

1.1.8. Comparison of Von-Neumann architecture and Harvard architecture

S.no	Parameters	Von-Neumann	Harvard
i	Memory	It uses a single memory connection.	It uses separate RAM and ROM memories.
ii	Design	Simple design. It uses the same path to access instructions and data.	Complex design. It has separate path for accessing instructions and data.

iii	Hardware	It requires less hardware.	It requires more hardware.
iv	Speed	Less speed	More speed.
v	Physical space	Require less physical space	Requires more physical space.
vi	Internal memory	Internal memory is not wasted.	Internal memory is wasted.

1.1.9. RISC and CISC Processors

Processors are divided into the following categories.

i) Complex Instruction Set Computer (CISC)
ii) Reduced Instruction Set Computer (RISC)

CISC is characterized by its large instruction set. Large number of instructions are available to program the processor. Single instruction can execute several low level operations. It is also capable of executing multi-step operations or addressing mode with single instructions. So the number of instructions required to do a job is very less and hence less memory is required.

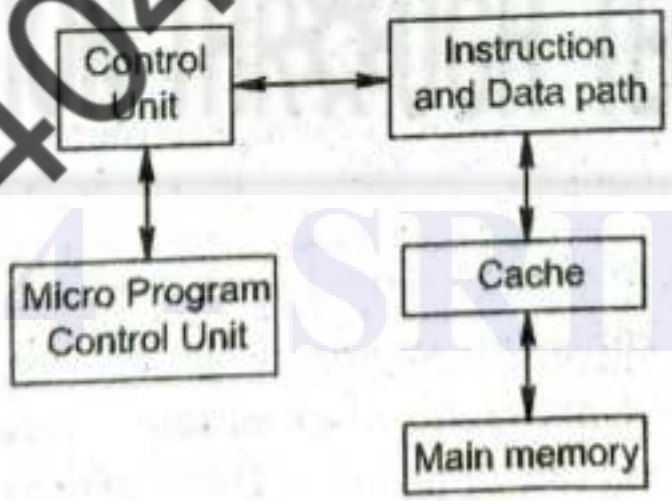


Fig.1.4 Architecture of CISC

The number of registers in CISC processors is very small. The aim of designing CISC processor is to reduce the

software complexity by increasing the complexity of the processor architecture.

Example: IntelX86 family and Motorola 68000 series processors.

The architecture of CISC is shown in the fig 1.4. It consists of Mircoprogram control unit, Control unit, Instructions and data path, Cache and main memory. The description of these units is given below.

i) **Mirco Program control unit:** The CISC uses a series of microinstructions of the microprogram stored in the control memory of the microprogram control unit and generate control signals.

ii) **Control unit:** It accesses the control signals, which are produced by the microprogram control unit.

iii) **Instructions and Data path:** It retrieves/fetches the op code and operands of the instructions from the memory.

iv) **Cache and Main memory:** Here the program instructions and operands are stored. Instructions in CISC is complex and it occupies more than a single word in memory.

RISC is characterised by its limited number of instructions. In RISC processors, the software is complex but the processor architecture is simple. However large number of registers are required in RISC processors, which are of small size and consume less power. RISC processors have pipelined instruction execution. While one instruction is being executed, second instruction is decoded and the third

instruction is fetched, leading to faster execution of the program. Embedded systems generally use RISC processors.

Examples: ARM, ATMEL, AVR, MIPS.

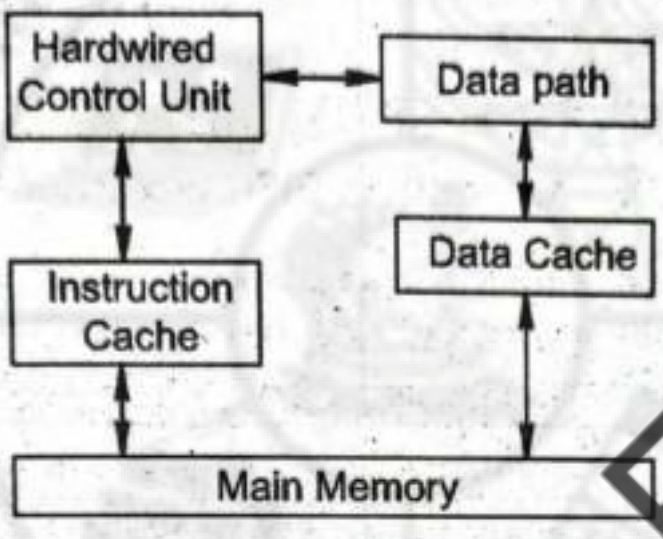


Fig.1.5 Architecture of RISC

The architecture of RISC is shown in the fig 1.5. It consists of Hardwired control unit, Data path, Instruction cache, Data cache and Main memory.

The hardwired control unit produces control signals which regulate the working of processors hardware. RISC architecture emphasizes on using the registers rather than memory.

This is because the registers are the fastest available memory source. The registers are physically small and are placed on the same chip where the ALU and control unit are placed on the processor. The RISC instructions operate on the operands present in processor's registers.

All instructions in RISC are simple and execute one instruction per cycle. So the instructions are hardwired and there is no need for control store.

1.1.10. Comparison of RISC and CISC processors.

S.no	RISC	CISC
i	It is a Reduced Instruction Set Computer.	It is a Complex Instruction Set Computer.
ii	It emphasizes on software to optimize the instruction set.	It emphasizes on hardware to optimize instruction set.
iii	It is a hardwired unit of programming in the RISC processor.	Microprogramming unit in CISC processor.
iv	It requires multiple register sets to store instructions.	It requires single register set to store instructions.
v	It has simple decoding of instruction.	It has complex decoding of instruction.
vi	Uses of pipelines are simple.	Uses of pipelines are difficult.
vii	It uses limited number of instructions.	It uses a large number of instructions.
viii	Less execution time.	More execution time.
ix	It can be used with high end applications.	It can be used with low end applications.
x	It has fixed format instruction.	It has variable format instruction.
xi	Programs need more memory space.	Programs need less memory space.

1.2. ARM Processor Architecture Fundamentals

1.2.1. ARM based Embedded system with hardware components

Embedded system can control many different devices from small sensors to the real time control systems. All these devices use a combination of software and hardware components. Selection of each component depends upon efficiency, future extension and expansion.

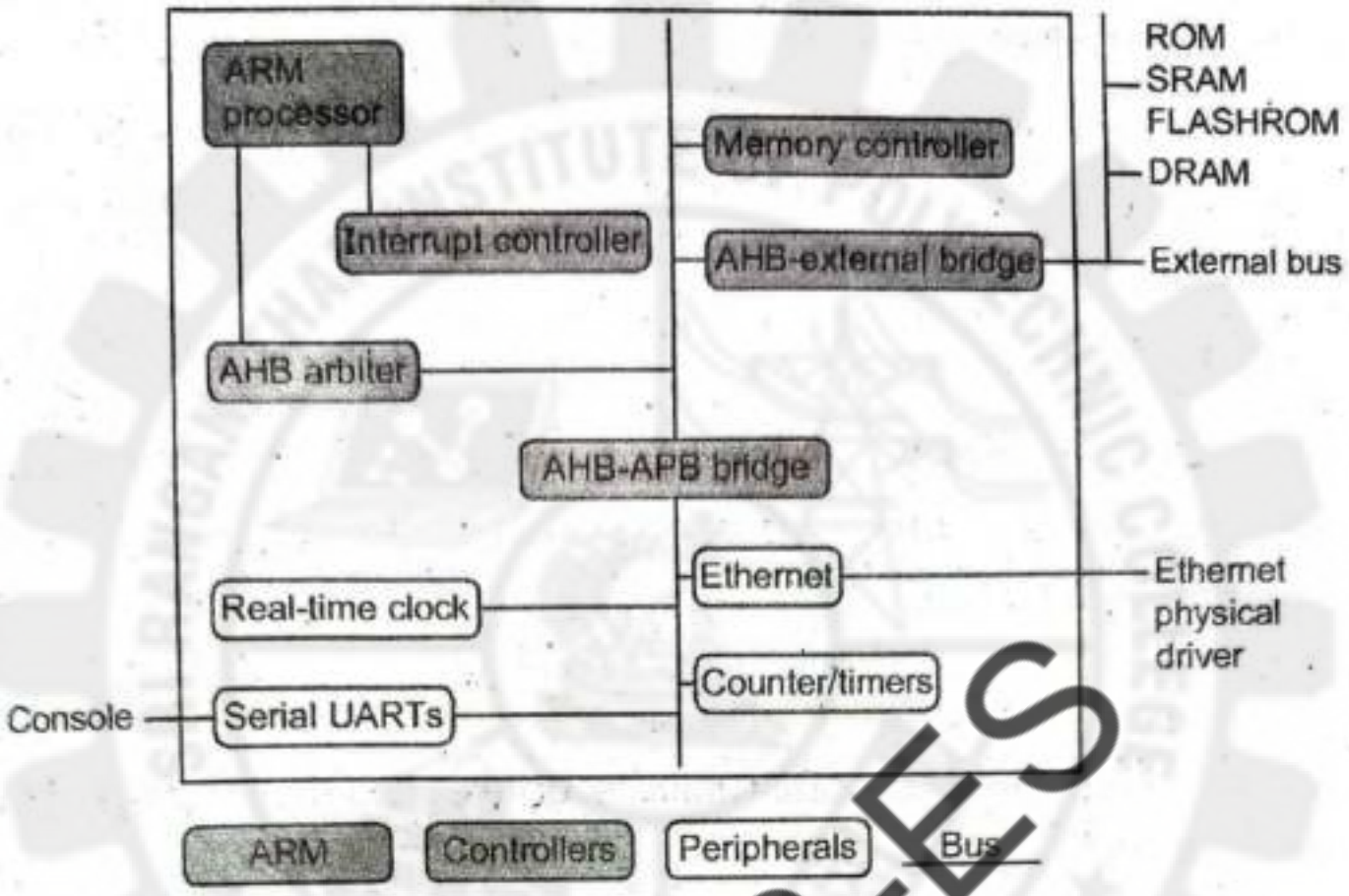


Fig.1.6 An example of an ARM-based embedded device, microcontroller

A typical embedded device based on an ARM core is shown in the fig.1.6. Each box represents a function or feature. The lines connecting the boxes are the buses carrying data.

The device is separated into four main hardware components, as described below.

i) The ARM processor

It controls the embedded device. Different versions of the ARM processors are available, with different operating characteristics. An ARM processor comprises a core plus the additional components that interface with the bus. The core is an execution engine that processes instructions and manipulates data. The components can include memory management and caches.

ii) Controllers

They coordinate important functional blocks of the system. Two commonly used controllers are interrupt and memory controllers.

iii) Peripherals

They provide all the input - output capability external to the chip and are responsible for the uniqueness of the embedded device.

iv) Bus

It is used to communicate between different parts of the device.

A) ARM bus technology

The most common bus technology is Peripheral Component Interconnect (PCI) bus and on chip bus. The PCI bus connects the devices like video cards and hard disc controllers to the X86 processor bus. This bus is designed to connect mechanically and electrically to devices external to the chip. It is built into the motherboard of the PC.

The on chip bus is internal to the chip used for interconnecting different peripheral devices with an ARM core.

B) Memory

An embedded system has some form of memory to store and execute code. The memories like cache, main and secondary storage are used with embedded system. The cache is placed between main memory and core. It is used to speed up data transfer between the processor and main memory. It increases the overall performance.

The main memory is large (256KB to 256MB) depending on application, and is generally stored in separate chips. Secondary storage is the largest and slowest form of memory. Hard disk drives and CD ROM drives are examples of secondary storage.

C) Peripherals

Embedded systems that interact with the outside world need some form of peripheral device. They perform input and output functions for the chip by connecting to other devices or sensors that are off chip.

D) Memory controllers

The memory controllers connect different types of memory to the processor bus. On power-up, a memory controller is configured in hardware to allow certain memory devices to be active. These memory devices allow the initialization code to be executed. Some memory devices must be set up by software.

E) Interrupt controller

An interrupt controller provides a programmable governing policy. It allows the software to determine which peripheral or device can interrupt the processor at any specific time by setting the appropriate bits in the interrupt controller registers.

There are two types of interrupt controller available for the ARM processor. They are the standard interrupt controller and the vector interrupt controller (VIC). The standard interrupt controller sends an interrupt signal to the processor core when an external device requests servicing.

It can be programmed to ignore or mask an individual device or set of devices. The VIC is more powerful than the standard interrupt controller because it prioritizes interrupts and simplifies the determination of which device causes the interrupt.

1.2.2. Pipeline

Pipeline is a mechanism used in RISC processors for executing instructions. Using a pipeline, execution speed will be increased by fetching the next instruction while other instructions are being decoded and executed.

A three stage pipe line has three operations of fetching, decoding and execution. The fetch loads an instruction from memory, decode identifies the instruction to be executed and execute processes the instruction and writes the result back to the register.

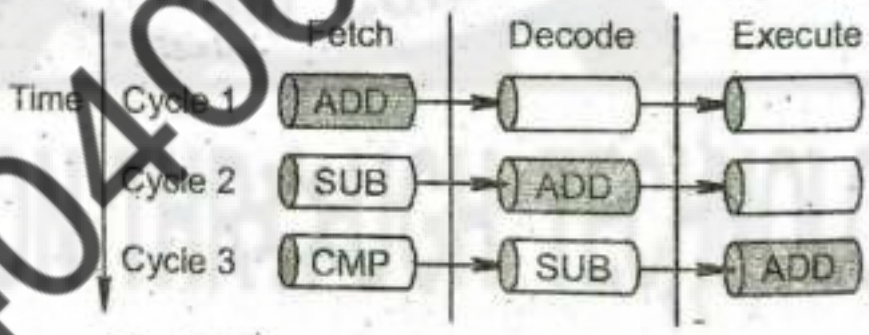


Fig.1.7 Pipelined instruction sequence

The fig.1.7. illustrates the pipeline using a simple example. It shows a sequence of three instructions being fetched, decoded and executed by the processor. Each instruction takes a single cycle to complete after the pipeline is filled. The three instructions of ADD, SUB and CMP are placed in the pipeline sequentially. In the first cycle the core fetches the ADD instruction from memory. In the second cycle the core fetches the SUB instruction and decodes the ADD

instruction. In the third cycle, both the SUB and ADD instructions are moved along the pipeline. The ADD instruction is executed, the SUB instruction is decoded and CMP instruction is fetched. The procedure is called filling the pipeline.

As the pipeline length increases, the amount of workdone at each stage is reduced. It allows the processor to attain a higher operating frequency. This in turn increases the performance.

The pipeline design for each ARM family differs. The ARM9 core increases the pipeline length to five stages, adds a memory and writeback stage. The ARM10 increases the pipeline length still further by adding a sixth stage of Issue.

Advantages

i) Amount of work done at each stage is reduced.
ii) Operating at higher frequency
iii) Performance increases.
iv) Code written for ARM 7 can be executed on an ARM 9 or ARM 10.

Disadvantages

i) Delay increases
ii) Data dependency increases.

1.2.3. Data Flow Model

The structure of data flow model is shown in the fig.1.8. The boxes represent either an operation unit or a storage area, the line represents the buses and the arrows represents the flow of data. The figure shows not only the flow of data but also the abstract components that make up an ARM core.

Data enters the processor core through the data bus. The data may be an instruction to execute or a data item. The given figure shows a Von Neumann implementation of the ARM – data items and instructions share the same bus. In contrast, the Harvard implementations of the ARM use two different buses.

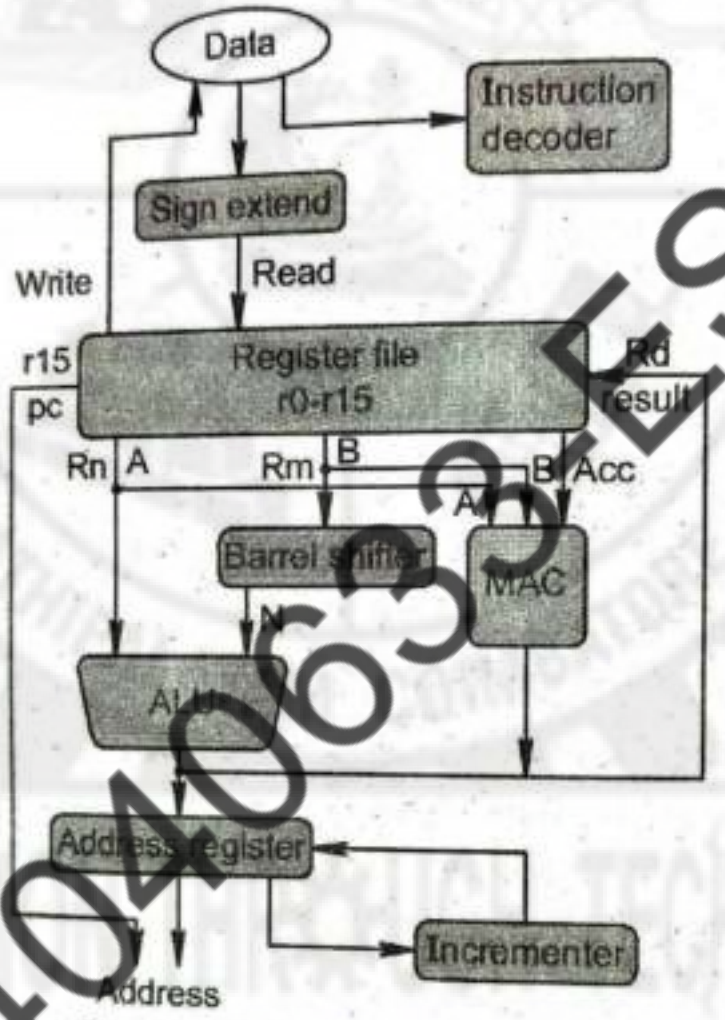


Fig.1.8 ARM core dataflow model

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

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

conversely the store instructions copy data from registers to memory. There are no data processing instructions that directly manipulate data in memory. Thus data processing is carried out in registers.

Data items are placed in register file. Register file is a storage bank made up of 32 bit registers. Since the ARM core is a 32 bit processor, most instructions treat the registers as holding signed or unsigned 32 bit values. The sign extend hardware converts signed 8 bit and 16 bit numbers to 32 bit values. Source operands are read from the register file using the internal buses A and B respectively.

The ALU (arithmetic logic unit) and MAC (multiply accumulate unit) take 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.

One important feature of the ARM is that register Rm alternatively can be preprocessed in the barrel shifter before it enters the ALU. Together the ALU and barrel shifter can calculate a wide range of expressions and addresses.

After passing through the functional units, the result in Rd is written back to the register file using the result bus. For load and store instructions the incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location. The processor continues executing instructions until an exception or interrupt changes the normal execution flow.

1.2.4. CPU Registers

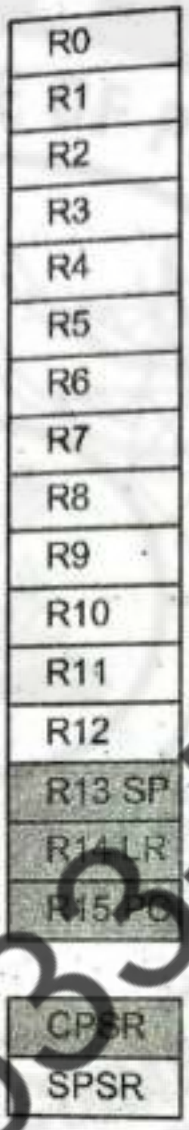


Fig.1.9 Registers available in user mode

General purpose registers hold either data or an address. They are identified with the letter "r" prefixed to the register number. For example, register 5 is given the label as *r5*. The active registers available in the user mode are shown in the fig.1.9. The user mode is a protected mode normally used when executing applications. The processor can operate in seven different modes. All the registers shown are 32 bits in size.

There are upto 18 active registers: 16 data registers and 2 processor (program) 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 functions i.e., *r13*, *r14* and *r15*. They are frequently given different labels to differentiate them from the other registers. In the given figure, the shaded registers are identified as assigned special purpose registers.

The descriptions of these registers are given below.

i) Register *r13* is called stack pointer (SP), which stores the head of the stack in the current processor mode.

ii) Register *r14* is called link register (LR), which is used for putting the return address whenever it calls a subroutine.

iii) Register *r15* is called program counter (PC), which 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. In ARM state, the registers *r0* to *r13* are orthogonal. This means any instruction we can apply to r0, we can well apply to any of the other registers.

However there are instructions that treat *r14* and *r15* in a special way. The two program status registers are called CPSR (current program status register) and SPSR (saved program status register)

1.2.5. Current Program Status Register (CPSR)

The ARM core uses the CPSR to monitor and control internal operations. The CPSR is a dedicated 32 bit register and resides in the register file. The basic layout of a generic

program status register is shown in the fig.1.10. Note that the shaded parts are reserved for future expansion.

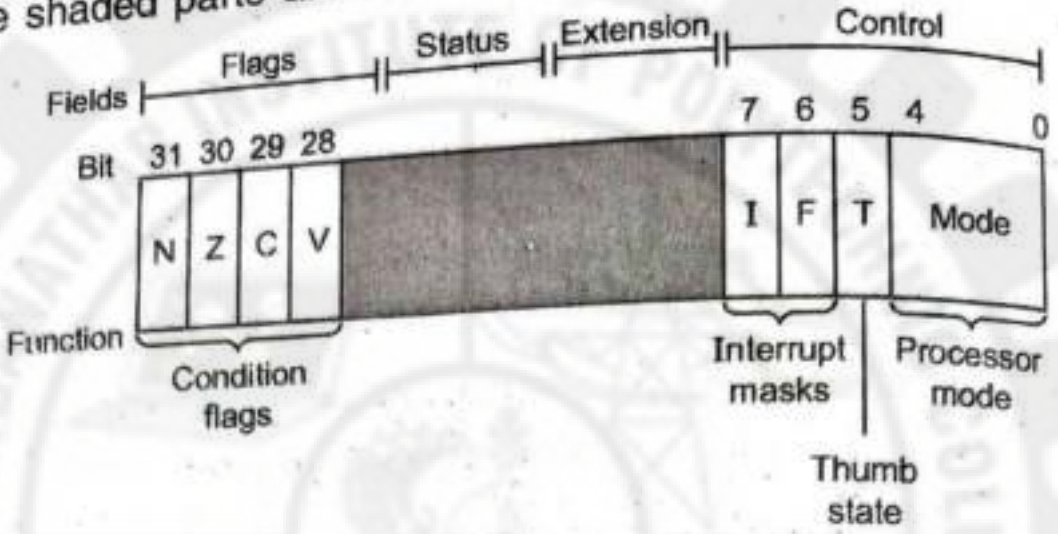


Fig.1.10 A generic program status register (PSR)

The CPSR is divided into four fields, each 8 bits wide: flags, status, extension and control. The extension and status fields are reserved for future use. The control field contains the processor mode, state and interrupt mask bits. The flag 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 executes 8 bit instructions.

A) Modes of operation

The processor mode determines which registers are active and the access rights to the CPSR register itself. Each processor mode is either privileged or nonprivileged. A privileged mode allows full read-write access to the CPSR. Conversely, a nonprivileged mode only allows read access to the control field in the CPSR but still allows read-write access to the condition flags.

Totally there are seven processor modes: Six privileged modes (abort, fast interrupt request, interrupt request, supervisor, system and undefined) and one nonprivileged mode (user).

i) Abort mode

The processor enters this mode when there is a failed attempt to access memory.

ii) Fast interrupt request and interrupt request modes

These modes correspond to the two interrupt levels available on the ARM processor.

iii) Supervisor mode

This is the mode that the processor is in after reset. It is generally the mode that an operating system kernal operates in.

iv) System mode

This mode is a special version of user mode that allows full read-write access to the CPSR.

v) Undefind mode

This mode is used when the processor encounters an instruction that is undefined or not supported by the implementation.

vi) User mode

This mode is used for programs and applications.

B) Banked register

The fig.1.11. shows all 37 registers in the register file. Of those 20 registers are hidden from a program at dif-

ferent times. These registers are called banked registers and are identified by the shading in the diagram. They are available only when the processor is in a particular mode.

For example, abort mode has banked registers *r13_abt*, *r14_abt* and *SPSR_abt*. Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic *or_mode*.

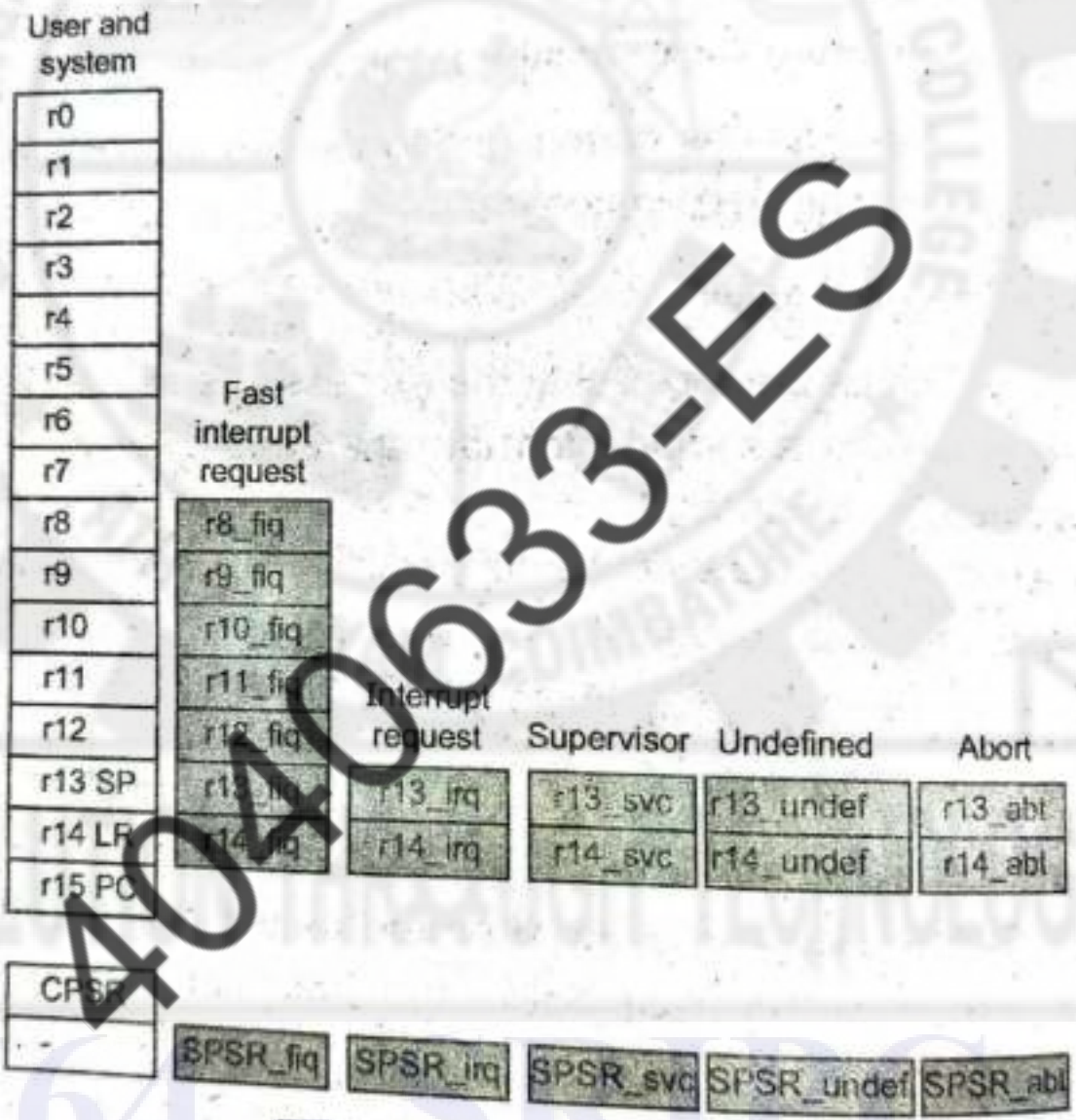


Fig.1.11 Complete ARM register set

Every processor mode except user mode can change mode by writing directly to the mode bits of CPSR. All processor modes except system mode have a set of associated banked registers that are subset of the main 16 registers. A banked register maps one to one onto a user

mode register. If we change the processor mode, a banked register from the new mode will replace an existing register.

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

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

1.2.6. Processor State and instruction sets

The state of the core determines which instruction set is being executed. There are three instruction sets: ARM, Thumb and Jazelle. The ARM instruction set is only active when the processor is in ARM state. Similarly the thumb instruction set is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16 bit instructions.

The description of state selection is shown below.

When T=0 ; The processor is in ARM state and executes ARM instructions.

When T = 1; The processor is in Thumb state and executes Thumb instructions.

When T = 0, J = 1; The processor executes Jazelle instruction.

The features of ARM and Thumb instructions are shown in the table 1.1.

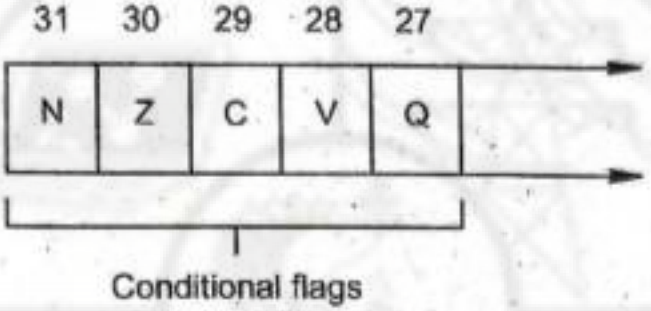
S.No.	Description	ARM	Thumb
i)	Instruction size	32 bit	16 bit
ii)	Conditional execution	Most instructions	Only branch instructions.
iii)	Data processing instructions	Access to barrel shifter and ALU	Separate barrel shifter and ALU instructions.
iv)	Program status register	Read-Write in privileged mode	No direct access
v)	Register usage	15 general purpose registers and program counter	8 general purpose registers + 7 high registers + program counter

Table 1.1

The Jazelle J and Thumb T bits in the CPSR reflect the state of the processor. When both J and T bits are '0', the processor is in ARM state and executes ARM instructions. This is the case when power is applied to the processor. When T bit is '1', then the processor is in Thumb state. To change states the core executes a specialized branch instruction.

The ARM designer introduced a third instruction set called Jazelle. Jazelle executes 8 bit instructions. It is a hybrid mix of the software and hardware designed to speed up the execution of Java bytecodes.

1.2.7. 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 zero flag in the CPSR is set. The representation of condition flags is shown in the table1.2.

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

Table 1.2

1.2.8. Exceptions, Interrupts and the vector table

1.2.8.1. Exceptions and Interrupts

Exception or interrupt is a control signal used for getting the immediate attention of processor. The description of various types of interrupts / exceptions handled by the processor is given below.

i) ***Reset vector*** is executed by the processor when power is applied. This instruction branches to the initialization code.

ii) ***Undefined instruction vector*** occurs when the processor cannot decode an instruction.

iii) ***Software interrupt vector*** occurs when executing a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.

iv) ***Prefetch abort vector*** occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.

v) ***Data abort vector*** occurs when an instruction attempts to access data memory without the correct access permissions.

vi) ***Interrupt request vector*** occurs when external hardware interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the CPSR.

vii) ***Fast interrupt request vector*** is similar to the interrupt request vector. It occurs when the hardware requires

faster response times. It can only be raised if FIQs are not masked in the CPSR.

1.2.8.2. 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

Table 1.3

When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the execution vector table shown in table1.3. Each vector table entry contains a form of branch instruction pointing to a start of a specific routine.

1.2.9. Endianness (Big endian and Little endian)

The term endianness refers to how bytes of a data word are ordered within memory. Endianness (or byte order) is also a big issue when reading data packets or compressed files.

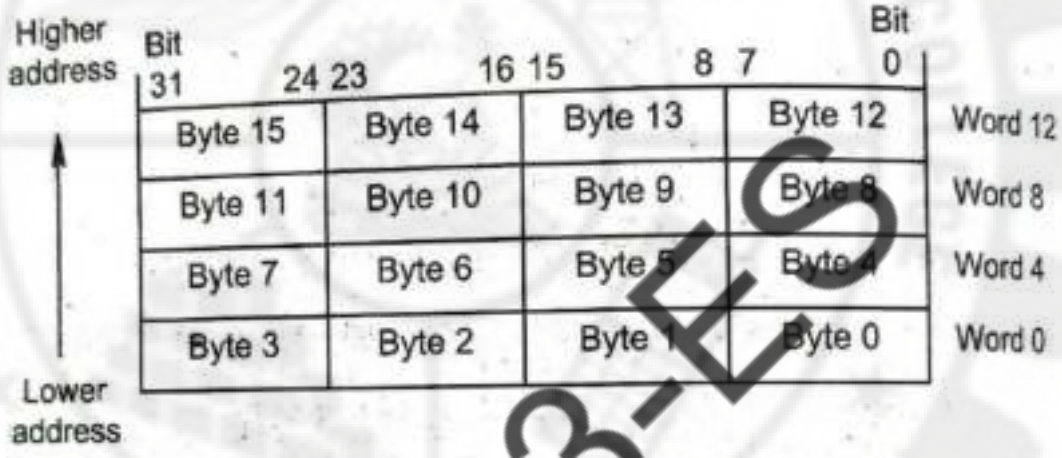
There are two methods used for storing data in ARM core. They are

i) Little endian, and

ii) Big endian

1.2.9.1. Little endian

In little endian configuration the least significant byte is placed at lower address. The memory organisation of little endian is shown below.



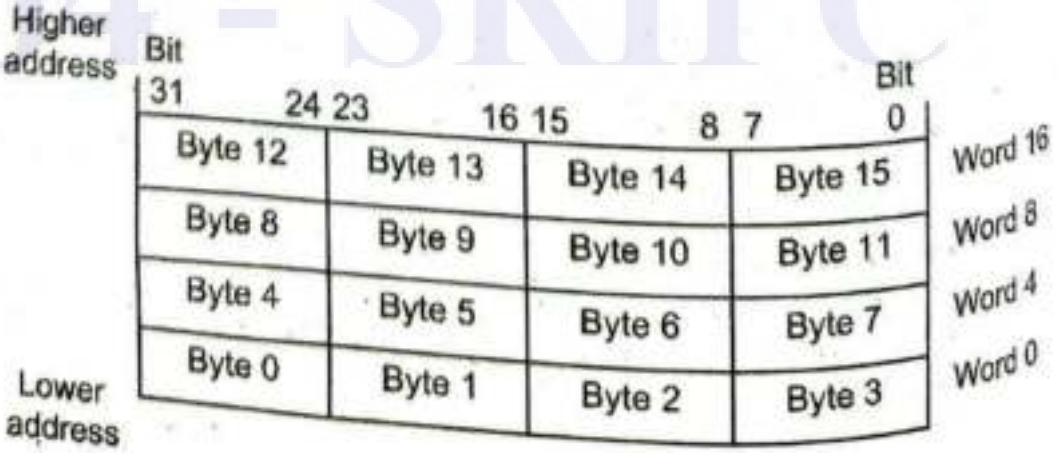
Advantages

i) Easy to place values

ii) Conversion from a 16 bit integer to a 32 bit integer address does not require any arithmetic.

1.2.9.2. Big endian

In big endian configuration the most significant byte is placed at lower address. The memory organisation of big endian is shown below.



Advantages

i) More natural.

ii) The sign of the number can be determined by looking at the byte at address offset 0.

iii) Strings and intergers are stored in the same order.

UNIT - II

ARM INSTRUCTION SET

2.1. INSTRUCTION SET

2.1.1. ARM Instruction Set - Introduction

ARM instructions process data held in registers and only access memory with load and store instructions. ARM instructions commonly contain two or three operands.

Thumb encodes a subset of the 32 bit ARM instructions into a 16 bit instruction set space. Thumb has higher performance than ARM on a processor with a 16 bit data bus, but lower performance than ARM on a processor with a 32 bit data bus. The thumb is used for memory constrained system.

Thumb has higher code density than ARM. Code density means the space taken up in memory by an executable program. Each Thumb instruction is related to a 32 bit ARM instruction.

ARM processors support six data types. They are,

i) 8 bit signed and unsigned bytes.

ii) 16 bit signed and unsigned half words.

iii) 32 bit signed and unsigned words.

All ARM instructions are 32 bit words and must be word aligned. Thumb instructions are half words and must be aligned on 2 byte boundaries. Internally all instructions

are on 32 bit operands. The shorter data types are only supported by data transfer instruction.

2.1.2. 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 we 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 with respect to bit 31 of the result. The Z flag is set if the result is zero.

2.1.2.1. Move instructions

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

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

MOV (move)	Move a 32-bit value into a register MOV Rd, N	Rd ⇐ N
MVN (move) negated	Move the NOT of the 32 bit value in to a register MVN Rd, N	Rd ⇐ ~ N

Usually N is a register Rm or a constant preceded by #.

Example: MOV r1, r2 ; (r1) ⇐ (r2)

This instruction moves the content of register *r2* into register *r1*.

Example: MVN r1, r2 ; (r1) ⇐ NOT (r2)

This instruction moves the complement of the content of register *r2* into register *r1*.

2.1.2.2. Barrel shifter

In an ordinary *MOV r1, r2* instruction, where N is a simple register *(r2)*. But N can be more than just a register or immediate value. If it can be a register Rm that has been preprocessed by a barrel shifter prior to being used by a data processing instruction.

Data processing instructions are processed within the arithmetic logic unit (ALU). A unique 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. The shift increases the power and flexibility of many data processing operations.

Some types of data processing instruction do not use the barrel shift. For example, the MUL (multiply), CLZ (count leading zeros) and QADD (signed saturated 32 bit add) instructions.

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

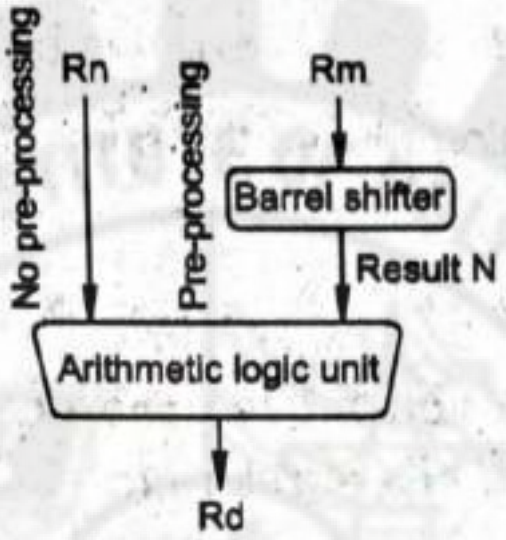


Fig.2.1 Barrel shifter and ALU

The illustration of barrel shifter and ALU is shown in the fig.2.1. Here the register Rn enters into the ALU without any preprocessing of registers, but the register Rm enters the ALU after barrel shift operations.

Example: MOV r1, r2, LSL # 2 ; (r1) $\Leftarrow 2^2$ x (r2)

This instruction logically shifts the value placed in register *r2* to left direction with 2 bit position. This instruction actually multiplies the register *r2* by 4 (= 2^2) and then places the result into register *r1*.

The different shift operations that we can use within barrel shifter are summarized below.

i) LSL:- Logical shift left by 0 to 31 bit places. Fill the vacated bits at the least significant end of the word with zeros.

ii) LSR:- Logical shift right by 0 to 32 bit places. Fill the vacated bits at the most significant end of the word with zeros.

iii) ASL:- Arithmetic shift left. This is a synonym for LSL.

iv) ASR:- Arithmetic shift right by 0 to 32 bit places. Fill the vacated bits at the most significant end of the word

with zeros if the source operand was positive or with ones if the source operand was negative.

v) ROR:- Rotate right by 0 to 32 bit places. The bits which fall off the least significant end of the word are used, in order to fill the vacated bits at the most significant end of the word.

vi) RRX:- Rotate right extended by 1 bit place. The vacated bit (bit 31) is filled with the old value of C flag and the operand is shifted one place to the right.

These shift operations are illustrated in the fig.2.2

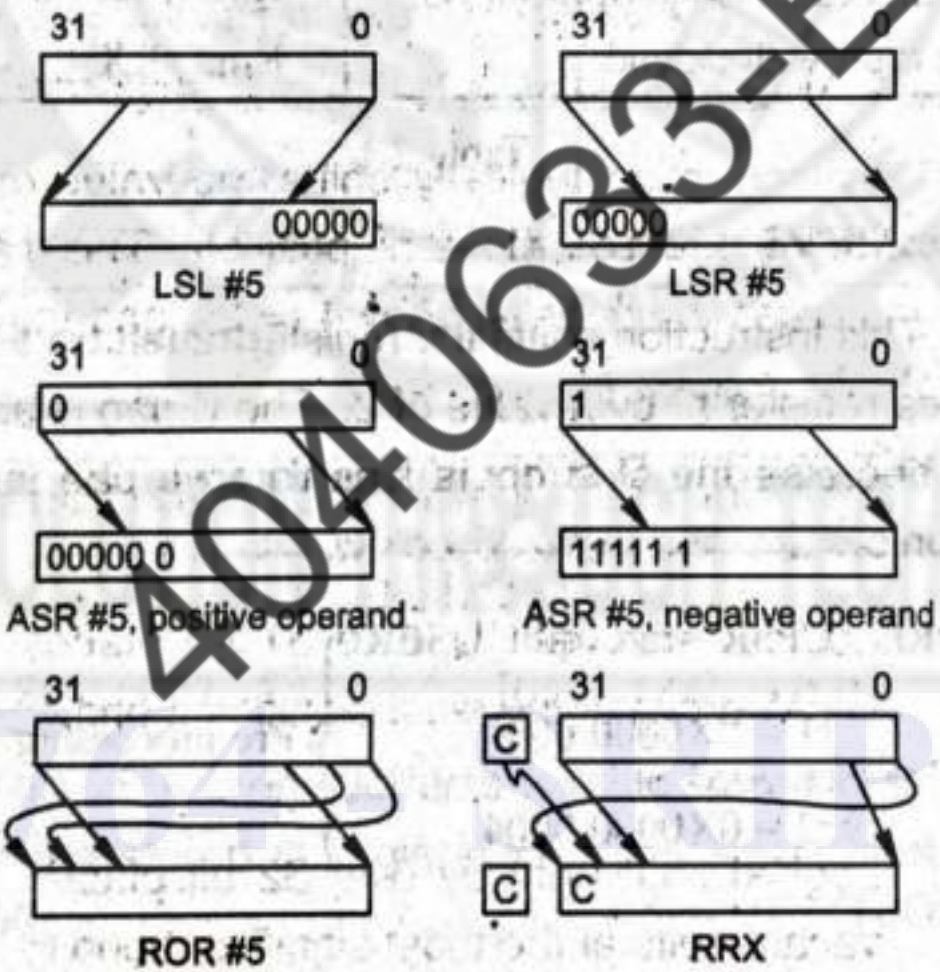


Fig.2.2 ARM shift operations

The syntax for Barrel shifter operation for data processing instructions is summarized in Table 2.1.

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

Table 2.1

Example: MOVS r1,r2, LSL #1

This instruction shifts the register r2 left by 1 bit. This multiplies register r2 by a value of 2. The C flag is updated in CPSR because the S suffix is presented in the instruction mnemonic.

PRE CPSR = nzcvqiFt_USER
r1 = 0X0000 0000
r2 = 0X0000 0004
} Pre processing

POST
CPSR = nzCvqiFt_USER
r1 = 0X0000 0008
r2 = 0X0000 0004
} Post processing

2.1.2.3. Arithmetic Instructions

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

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

N is the result of the shifter operation

The arithmetic instructions are described in the Table 2.2.

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

Table 2.2.

Example: SUB r1, r2, r3 ; [(r1) ⇐ (r2) – (r3)]

This instruction subtracts a value stored in register r3 from a value stored in register *r2*. The result is stored in register *r1*.

Example: RSB r1, r2, #2 ; [(r1) ⇐ 2 – (r2)]

This reverse subtract instruction subtracts *r2* from the constant value # 2, and stores the result in *r1*.

Example: SUBS r1, r1, #1 ;[(r1) ⇐ (r1) – 1]

This instruction is used for decrementing loop counters. This instruction subtracts 1 from the value placed in register *r1*, and also store the result in the same *r1* register.

2.1.2.4. Using the Barrel shifter with arithmetic instructions

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

Example: ADD r1, r2, r2, LSL #1. ; [(r1) ⇐ (r2) + (r2) x 2]

This instruction uses inline barrel shifter with an arithmetic instruction. This instruction actually multiplies the value stored in register *r2* by 3. The content of register *r2* is first shifted one location to the left, to give the value of twice *r2*. The ADD instruction then adds the result of the barrel shift operation to register *r2*. The final result is transferred into register *r1*, which is equal to 3 times the value stored in register *r2*.

2.1.2.5. Logical instructions

Logical instructions perform bitwise logical operation on the two source registers.

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

AND	logical AND of two 32-bit values	Rd = Rn & N
ORR	logical OR of two 32-bit values	Rd = Rn | N
EOR	logical exclusive OR of two 32-bit values	Rd = Rn ^ N
BIC	logical bit clear (AND NOT)	Rd = Rn & ~ N

Example: ORR r1, r2, r3 ; [(r1) ⇐ (r2) | (r3)]

This instruction logically OR-ed the content of registers *r2* and *r3*, and store the result in register *r1*.

Example : BIC r1, r2, r3 ; [(r1) ⇐ (r2) & (r3)]

It is a more complicated logical instruction, which carries out a logical bit clear. This instruction logically AND-ed the complement of content of register r3 with the content of r2, and the result is stored in register r1. This instruction is particularly used for clearing status bits, and is frequently used to change interrupt masks in CPSR.

2.1.2.6. Comparison instructions

The comparison instructions are used to compare or test a register with a 32 bit value. They update the CPSR flag bits according to the result, but do not affect other registers.

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 ^ N
TST	test bits of a 32-bit value	flags set as a result of Rn & N

N is the result of the shifter operation.

Example: CMP r1, r2 ; [(r1) – (r2)]

This instruction effectively subtracts the content of register r2 from the content of register r1, but the results are discarded. The condition flags are modified in accordance with the result.

2.1.2.7. Multiply Instructions

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

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

MUL {<cond>} {S} Rd, Rm, Rs

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

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

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

Example: MUL r1, r2, r3 ; [(r1) = (r2) x (r3)]

This instruction multiplies the values placed in registers *r2* and *r3*, and stores the result in *r1* register. The long multiply instructions (SMLAL, SMULL, UMLAL and UMULL) produce a 64 bit result. The result is stored in RdLo and RdHi registers. RdLo holds the lower 32 bits, and RdHi holds the higher 32 bits of the result.

Example: UMULL *r1, r2, r3, r4* ; [(r2), (r1) ⇐ (r3) x (r4)

This instruction multiplies the values placed in registers *r3* and *r4*, and the result is stored in registers *r2* (higher 32 bits) and *r1* (lower 32 bits).

2.1.3. Branch instructions

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

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

B	branch	PC= label
BL	branch with link	PC = label LR = address of the next instruction after the BL
BX	branch exchange	PC=Rm & 0xFFFFFFFE, T=Rm & 1
BLX	branch exchange with link	PC = label, T = 1 PC =Rm & 0XFFFFFFFE, T=Rm & 1 LR = address of the next instruction after the BLX

The address label is stored in the instruction as a signed PC relative offset. T refers to the Thumb bit in the CPSR. When instructions set T, the ARM switches to Thumb state.

Example: B forward

During the execution of this instruction, the control is transferred to the instruction which has a label as forward. This is an unconditional branch instruction.

Example: BL subroutine ; Branch to subroutine

This instruction is similar to the B instruction but overwrites the link register LR with a return address. It performs a subroutine call. When return from subroutine, the PC is copied from link register.

2.1.4. Load - store instructions

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

2.1.4.1. Single register transfer

These instructions are used for moving a single data in and out of a register. The data types supported are signed and unsigned words (32 bit), half words (16-bit) and bytes. The various types of load - store single register transfer instructions are described below.

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

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

STR {<cond>}H Rd, addressing

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 from 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 (mem 16[address])

Example: LDR r1, [r2] ; [(r1) ⇐ ((r2))]

This instruction loads register *r1* with the content of memory address specified by register *r2.*

Example: STR r1, [r2] ; [((r2)) ⇐ (r1)]

This instruction stores the content of register *r1* to the memory address pointed by register *r2.* The above instructions uses preindex method. The register *r2* is called base address register.

2.1.4.2. Single register load - store addressing mode

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the following indexing methods: preindex with write back, preindex and postindex, shown in the table 2.3.

Index method	Data	Base address register	Example
Preindex with writeback	mem[base+offset]	base+offset	LDR r1, [r2,#4]!
Preindex	mem[base+offset]	not updated	LDR r1,[r2,#4]
Postindex	mem[base]	base + offset	LDR r1, [r2], #4

Table 2.3 Index methods

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

(a) Pre index with writeback

Preindex with writeback calculates an address from the base register plus address offset and then updates that address base register with the new address. It is useful for transversing an array.

Example 1 : LDR r1, [r2, #4]!

This instruction uses preindexing with writeback method. In this instruction the content of register *r2* is incremented by 4. After that the content of incremented memory location specified by *r2* is moved to register *r1.*

(b) Pre index

The pre index offset is the same as the pre index with write back but does not update the address base register. It is useful for accessing an element in a data structure.

Example 2: LDR r1, [r2, #4]

This instruction uses preindexing method. In this instruction the content of memory location specified by *r2* added with 4 is moved to register *r1.* The content of *r2* is not changed.

(c) Post index

The post index only updates the address base register after the address is used. It is useful for transversing array.

Example 3: LDR r1, [r2], #4

This instruction uses postindexing method. In this instruction the content of memory location specified by register *r2* is moved to register *r1*. After that the content of register *r2* is incremented by 4.

A) Addressing modes - Load and store - 32 bit word/unsigned byte

The addressing modes available with a particular load and store instruction depend on the instruction class. The addressing modes with load and store of a 32 bit word or an unsigned byte are described in the table 2.4.

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

Table 2.4

i) A signed offset or register is denoted by "+/–", identifying that it is either a positive or negative offset from the base address register Rn.

ii) The base address register is a pointer to a byte in memory.

iii) The offset specifies a number of bytes.

iv) Immediate means the address is calculated using the base address register and a 12 bit offset encoded in the instruction.

v) Register means the address is calculated using the base address register and a specific register's contents.

vi) Scaled means the address is calculated using the base address register and a barrel shift operation.

An example of the different variations of the LDR instruction is shown in the table 2.5.

	Instruction	r0 =	r1 +=
Preindex with writeback	LDR r0, (r1, #0 x 4)! LDR r0, (r1, r2)! LDR r0, (r1, r2, LSR # 0 x 4)!	mem32(r1+0x4) mem 32 (r1 + r2) mem 32 [r1 + (r2 LSR 0 x 4)]	0 x 4 r2 (r2 LSR 0 x 4)
Preindex	LDR r0, [r1, # 0 x 4] LDR r0, (r1, r2) LDR r0, (r1, –r2, LSR # 0 x 4)	mem 32 (r1 + 0 x 4) mem 32 (r1 + r2) mem 32 [r1 – (r2 LSR 0 x 4)]	not updated not updated not updated
Postindex	LDR r0, (r1), # 0x4 LDR r0, (r1), r2 LDR r0, (r1), r2, LSR # 0 x 4	mem 32 (r1) mem 32 (r1) mem 32 (r1)	0 x 4 r2 (r2 LSR 0x4)

Table 2.5

B) Addressing mode - Load and store - 16 bit half word/signed byte

The addressing modes available on load and store instructions using 16 bit half word or signed byte data are shown in the table 2.6.

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

Table 2.6

These operations can not use the barrel shifter. There are no STRSB or STRSH instructions since STRH stores both a signed and unsigned halfword; similarly STRB stores signed and unsigned bytes. The variations of STRH instructions are shown in the table 2.7.

	Instruction	Result	r1 +=
Preindex with writeback	STRH r0, [r1, # 0 x 4]!	mem 16 [r1 + 0x4] = r0	0x4
	STRH r0,[r1, r2]!	mem 16 [r1 + r2] = r0	r2
Preindex	STRH r0, [r1, # 0 x 4]	mem 16 [r1 + 0x4] = r0	not updated
	STRH r0, [r1, r2]	mem 16 [r1 + r2] = r0	not updated

Postindex	STRH r0, [r1], #0 x 4	mem 16 [r1] = r0	0x4
	STRH r0, [r1], r2	mem 16 [r1] = r0	r2

Table 2.7 Variations of STRH instructions

(C) Load - Store multiple instructions

Load-store multiple instructions increase interrupt latency. ARM implementations do not usually interrupt instructions while they are executing.

Syntax :

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

LDM	Load multiple registers	{Rd}^N ⇐ mem 32 [start address + 4 x N] optional Rn updated
STM	Save multiple registers	{Rd}^N ⇒ mem 32 [start address + 4 x N] optional Rn updated

Addressing modes: Load-store multiple instructions

The different addressing modes for load-store multiple instructions are shown in the table 2.8.

Addressing mode	Description	Start Address	End address	Rn!
IA	increment after	Rn	Rn+4 x N–4	Rn+4 x N
IB	increment before	Rn + 4	Rn + 4 x N	Rn+4 x N
DA	decrement after	Rn – 4 x N+4	Rn	Rn–4 x N
DB	decrement before	Rn – 4 x N	Rn – 4	Rn–4 x N

Table 2.8: Addressing mode for load-store multiple instructions

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

Example 1: LDMIA r0!, {r1 – r3}

The register r0 is the base register Rn and is followed by ! indicating that the register is updated after instruction is executed. The "–" character is used to identify the range of registers. In this case the range is from registers *r1* to *r3* inclusive.

Each register can also be listed, using a comma to separate each register within "{" and "}" brackets.

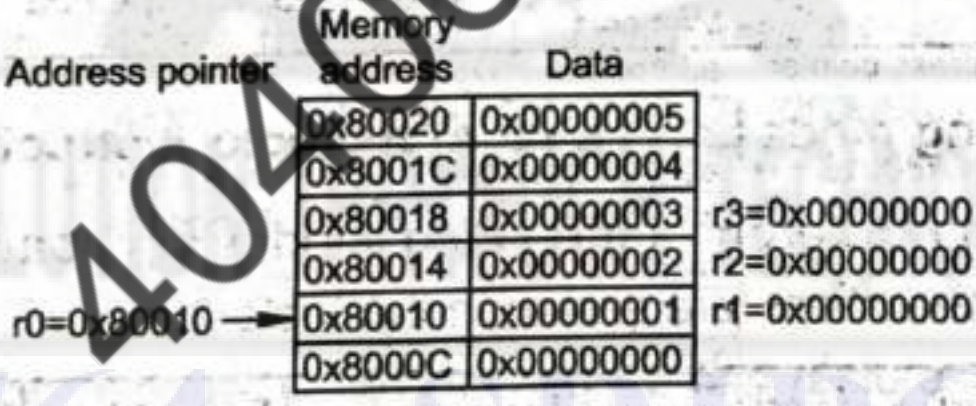


Fig.2.3 Pre-condition for LDMIA instruction

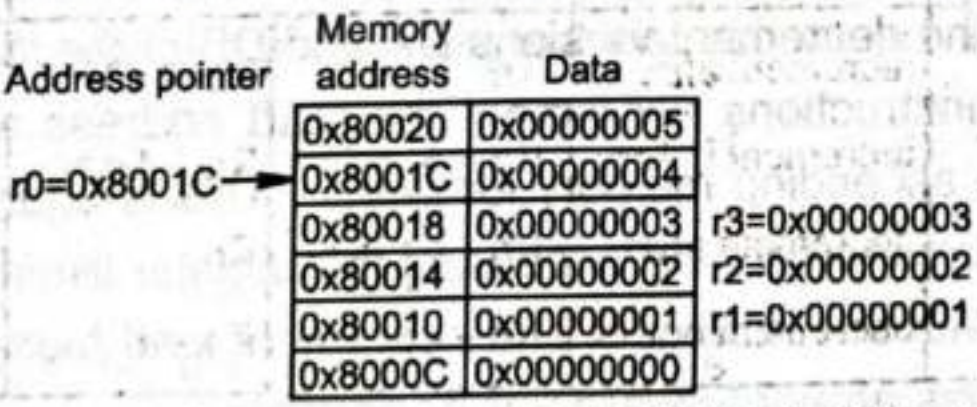


Fig.2.4 Post-condition for LDMIA instruction

The given instruction is a load multiple register increment after instruction. The graphical representation of this instruction at pre and post conditions is described in the fig 2.3 and 2.4.

The base register *r0* points to memory address 0x80010 in the pre condition. Memory address 0x80010, 0x80014 and 0x80018 contains the values of 1, 2 and 3 respectively. After the execution of load multiple instruction, the registers r1, r2 and r3 contain the values as shown in the fig.2.4.

Example 2: LDMIB r0!, {r1 – r3}

This is a load multiple register increment before instruction. By using the same pre condition, the first word pointed by register r0 is ignored and register r1 is loaded from the next memory location as shown in the fig.2.5.

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	

Fig.2.5 Post-condition for LDMIB instruction

The decrement versions DA and DB of the load-store multiple instructions decrement the start address and then store the ascending memory locations. This is equivalent to descending memory but accessing the register list in reverse order. With the increment and decrement load multiples, we can access arrays forwards or backwards.

A list of load-store multiple instruction pairs is shown in the table2.9. If we use a store with base update, then the paired load instruction of the same number of registers will reload the data and restore the base address pointer. This is used for saving a group of registers temporarily and restoring them later.

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

Table 2.9

Example: STMIB r0!, {r1 – r3}

This is an "STM increment before" instruction. This instruction stores the content of registers *r1, r2* and *r3* in memory locations. We can then corrupt registers.

Example: LDMDA r0!, {r1 – r3}

This is "LDM decrement after" instruction. This instruction reloads the original values and restores the base pointer *r0*.

The functions of above two instructions are illustrated below.

```
PRE     r0  =  0 x 00009000
        r1  =  0 x 00000009
        r2  =  0 x 00000008
        r3  =  0 x 00000007
```

```
        STMIB   r0!, {r1 – r3}
        MOV     r1, #1
        MOV     r2, #2
        MOV     r3, #3
PRE (2) r0  =   0 x 0000900C
        r1  =   0 x 00000001
        r2  =   0 x 00000002
        r3  =   0 x 00000003
        LDMDA   r0!, {r1 – r3}
POST    r0  =   0 x 00009000
        r1  =   0 x 00000009
        r2  =   0 x 00000008
        r3  =   0 x 00000007
```

Example: LDMIA r9!, {r0 – r7}

This instruction is a load-store multiple instruction. This is a simple routine that copies blocks of 32 bytes from source address location to a destination address location.

This instruction loads the data pointed by register *r9* into registers *r0* to *r7*. It also updates r9 to point to the next block of data to be copied.

Example : STMIA r10!, {r0 – r7}

This instruction copies the contents of registers *r0* to *r7* to the destination memory address pointed by register *r10*. It also updates *r10* to point to the next destination location.

```
Example :       CMP r9, r11     ; Compare (r9) and (r11)
                BNE loop        ; Branch not equal
```

CMP and BNE compare pointers *r9* and *r11* to check whether the end of the block copy has been reached. If the block copy is complete, then the routine finshes, otherwise the loop repeats with the updated values of registers *r9* and *r11*.

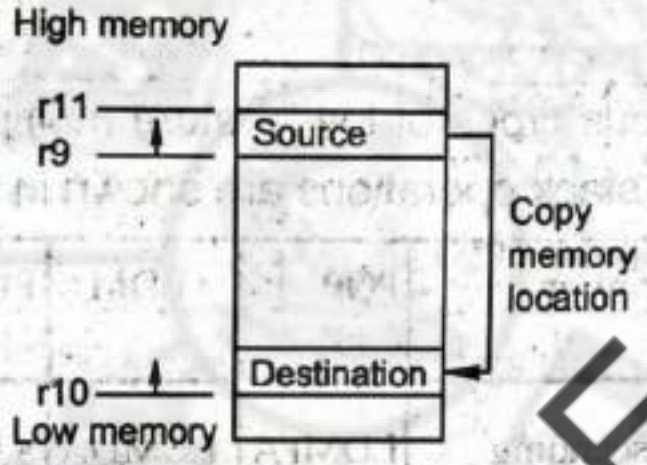


Fig.2.6 Block memory copy in the memory map

The memory map of the block memory copy and how the routine moves through memory is shown in the fig.2.6. Theoretically this loop transfers 32 bytes (8 words) in two instructions.

A) Stack operations

The ARM architecture uses the load-store multiple instructions to carry out stack operations. The POP operation (removing data from a stack) uses a load multiple instruction. Similarly the PUSH operation (placing data on to the stack) uses a store multiple instruction.

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

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

The various types of load - store multiple addressing mode to support stack operations are shown in the table 2.10.

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

Table 2.10: Addressing modes for stack operations

The LDMFD and STMFD instructions provide the POP and PUSH functions respectively.

Example: STMFD sp!, {r1, r4}

This instruction pushes registers onto the stack, updating SP. A push onto a full descending stack is described in the fig.2.7. When the stack grows, the stack pointer points to the last full entry in the stack.

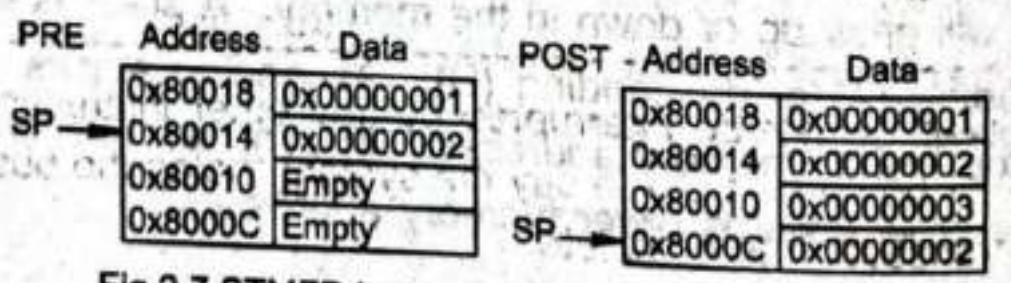


Fig.2.7 STMFD instruction-full stack push operation

Example: STMED sp!, {r1, r4}

This instruction is a push operation on any empty stack. This instruction pushes the registers onto the stack but updates register SP to point to the next empty location. The functions are described in the fig.2.8.

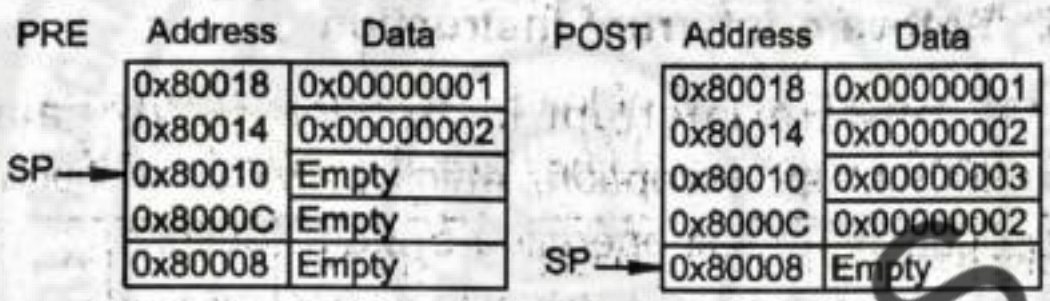


Fig.2.8 STMED instruction-empty stack push operation

B) SWAP instruction

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

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

SWP	swap a word between memory and a register	tmp = mem 32 [Rn] mem 32 [Rn] = Rm Rd = tmp
SWPB	swap a byte between memory and a register	tmp = mem 8 [Rn] mem 8 [Rn] = Rm Rd = tmp

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

Example: SWP r0, r1, [r2]

This instruction loads register r0 with the content of memory location specified by register *r2*, and also moves (stores) the content of register *r1* to the memory location specified by *r2*.

2.1.5. Software interrupt instruction

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

Syntax : SWI {<cond>} SWI_number

SWI	Software interrupt	LR_svc = address of instruction following the SWI SPSR_svc = CPSR PC = vectors + 0 x 8 CPSR mode = SVC CPSR I = 1 (mask IRQ interrupts)

When the processor executes an SWI instruction, it sets the program counter PC to the offset 0X8 in the vector table. The instruction also forces the processor mode to SVC, which allows an operating system routine to be called in a privileged mode.

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

Example: 0X0000 8000 SWI 0x123456

This is a SWI instruction with SWI number 0X123456 used by ARM toolkits as a debugging SWI. Typically the SWI instruction is executed in user mode.

The content of *CPSR*, *SPSR*, *PC*, *LR* and *r0* at preprocessing and post processing is illustrated below.

```
PRE   CPSR = nzcVqift_USER
      PC = 0x0000 8000
      LR = 0x003FFFFF;      LR = r14
      r0 = 0x12
POST  CPSR = nzcVqIft_SVC
      SPSR = nzcVqift_USER
      PC  = 0 x 0000 0008
      LR  =  0x0000 8004
      r0  = 0x12
```

Since SWI instructions are used to call operating system routines, we need some form of parameter passing. This is achieved using registers. In this example, *r0* is used to pass the parameter 0X12. The return values are also passed back via registers.

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

The SWI number is determined by

SWI_number = <SWI instruction> AND NOT (0XFF000000)

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

2.1.6. Program status register instructions

The ARM instruction set provides two instructions to directly control a program status register (PSR).

Syntax:

MRS {<cond>} Rd, <CPSR | SPSR>
MSR {<cond>} <CPSR | SPSR> <fields>, Rm
MSR {<cond>} <CPSR | SPSR>_<fields>, # immediate

The MRS instruction transfers the contents of either the CPSR or SPSR into a register; in 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.

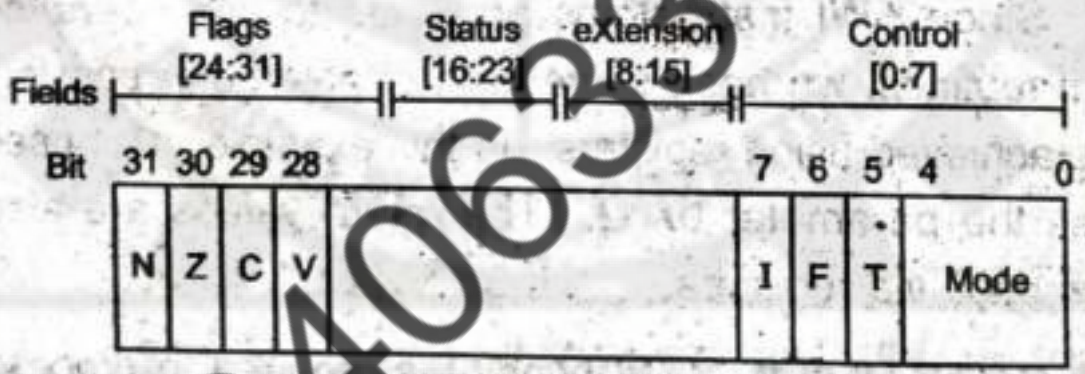


Fig.2.9 PSR byte fields

The label called "fields", can be any combination of control (c), extension (x), status (s) and flags (f). These fields related to particular byte regions in a PSR are shown in the fig.2.9.

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

The 'c' field controls the interrupt masks, Thumb state and processor mode.

Example

```
        MRS r1, CPSR
        BIC r1, r1, #0X80          ; 0B0100 0000
        MSR CPSR_c, r1
PRE     CPSR = nzcvqIFt_SVC
POST    CPSR = nzcvqiFt_SVC
```

The MRS first copies the CPSR into register *r1.* The BIC instruction clears bit 7 of *r1.* Register *r1* is then copied back into CPSR, which enables IRQ interrupts.

This example is in SVC mode. In user mode we can read all CPSR bits, but we can only update the condition flag field "F".

2.1.7. Loading constants

There is no ARM instruction to move a 32 bit constant into a register. Since ARM instructions are 32 bits in size, they obviously can not specify a general 32 bit constant.

There are two pseudo instructions to move a 32 bit value into a register.

Syntax

```
LDR Rd, = constant
ADR Rd, label.
```

LDR	load constant pseudo instruction	Rd = 32 bit constant
ADR	load address pseudo instruction	Rd = 32 bit relative address

The first pseudo instruction writes a 32 bit constant to a register. The second pseudo instruction writes a relative address into a register which will be encoded using a PC- relative expression.

2.1.8. Conditional execution

Most ARM instructions are conditionally executed. This instruction only executes if the condition code flags pass a given condition or test. By using conditional execution instructions, we can increase performance and code density.

The condition field is a two-letter mnemonic appended to the instruction mnemonic. The default mnemonic is AL or always execute.

Conditional execution reduces the number of branches, which also reduce the number of pipeline flushes and thus improves the performance of the executed code. Conditional execution depends upon two components. They are condition field and condition flags. The condition field is located in the instruction and the condition flags are located in the CPSR.

Example: ADDEQ r1, r2, r3

This is an ADD instruction with the EQ condition appended. This instruction will only be executed when the zero flag in the CPSR is set to 1.

(r1) = (r2) + (r3) if zero flag is set.

Only comparing instructions and data processing instructions with the 'S' suffix appended to the mnemonic update the condition flags in the CPSR.

2.2. THUMB INSTRUCTIONS

Thumb encodes a subset of the 32 bit ARM instructions into a 16 bit instruction set space. Each thumb instruction is related to a 32 bit ARM instruction. In thumb state, the higher registers *r8* to *r12* are only accessible with MOV, ADD or CMP instructions. CMP and all the data processing instructions that operate on low registers update the conditional flags in the CPSR.

2.2.1. Thumb instruction set

Thumb instruction is a compressed form of ARM instruction set. The T (bit 5) bit of CPSR decides whether the ARM processor will execute thumb instructions. If T is set the processor executes 16 bit thumb instructions otherwise it executes 32 bit ARM instructions. Thumb encodes a subset of 32 bit ARM instructions into a 16 bit instruction set space. Thumb has higher performance than ARM on a processor with a 16 bit data but lower performance than ARM on a 32 bit data bus. Each thumb instruction is related to a 32 bit ARM instruction.

2.2.2. ARM Thumb similarities

i) The load - store architecture with data processing, data transfer and control flow instructions.

ii) Support for 8 bit byte, 16 bit half word and 32 bit word data types.

iii) A 32 bit unsegmented memory.

2.2.3 Differences between Thumb and ARM

S.No	Thumb	ARM
i)	Most instructions are executed unconditionally	All instructions are executed conditionally
ii)	Many instructions use a 2 address format (destination register, source register)	Many instructions use a 3 address format
iii)	Less regular than ARM instructions	Regular instructions

2.2.4 Usage of thumb register

In thumb state we have to access all registers. Only the low registers r0 to r12 are fully accessible. The higher registers r8 to r12 are only accessible with MOV, ADD or CMP instructions. CMP and all the data processing instructions that operate on low registers update the conditional flags in CPSR. The summary of thumb register usage is shown in the table 2.11

Registers	Access
r0 - r7	fully accessible
r8 - r12	Only accessible by MOV, ADD and CMP
r13 SP	Limited accessible
r14 LR	Limited accessible
r15 PC	Limited accessible
CPSR	Only indirect access
SPSR	No access

Table 2.11 Summary of thumb register usage

2.2.5. Data processing instructions

The data processing instructions manipulate data within registers. They include move instructions, arithmetic instructions, shifts, logical instructions, comparison instructions and multiply instructions.

2.2.5.1. Move instructions

Move instructions are used to move a 32 bit value or logical NOT of a 32 bit value into a register.

Syntax: <instruction> Rd, N

MOV (move)	Move a 32 bit value placed in register Rm to Rd MOV Rd, Rm	Rd ⇐ Rm
MOV (move)	Move an immediate 32 bit value into register Rd MOV Rd, # immediate	Rd ⇐ immdiate
MVN (move negated)	Move the NOT of the 32 bit value placed in Rm into Rd	Rd ⇐ NOT Rm

N is a register or a constant preceded by #.

2.2.5.2. Arithmetic instructions

The arithmetic instructions implement addition and subtraction of 32 bit values. N is a register or 32 bit value.

ADC	add two 32 bit values and carry: ADC Rd, Rm	Rd ⇐ Rd + Rm + carry

ADD	add two 32 bit values
	ADD Rd, N
	ADD Rd, Rn, N
		
		Rd ⇐ Rd + N
		Rd ⇐ Rn + N

SBC	subtract with carry a 32 bit value SBC Rd, Rm	Rd ⇐ Rd – Rm – NOT (C flag)

SUB	subtract two 32 bit values
	SUB Rd, N
	SUB Rd, Rn, N
		Rd ⇐ Rd – N
		Rd ⇐ Rn – N

NEG	negate a 32 bit value NEG Rd, Rm	Rd ⇐ 0 – Rm

2.2.5.3. Logical instructions

Logical instructions perform bitwise logical operations on the two source operands.

Syntax: <instruction> Rd, Rm

AND	logical AND of two 32 bit values: AND Rd, Rn	Rd ⇐ Rd & Rn
ORR	logical OR of two 32 bit values: ORR Rd, Rn	Rd ⇐ Rd | Rn
EOR	logical Ex-OR of two 32 bit values: EOR Rd, Rn	Rd ⇐ Rd ^ Rn
BIC	logical bit clear (AND NOT) of two 32 bit values: BIC Rd, Rn	Rd ⇐ Rn & NOT Rn

2.2.5.4. Comparison instructions

The comparison instructions are used to compare or test two registers (or immediate value). They update the CPSR flag bits according to the result but do not affect registers.

Syntax:<CMP> Rn, N

<CMN | TST> Rn, Rm

(N is a register or immediate value)

CMP	Compare two 32 bit integers	Flags set as a result of Rn – N
CMN	Compare negative two 32 bit values.	Flags sets as a result of Rn + Rm
TST	Test bits of a 32 bit value.	Flags set as a result of Rn & Rm

2.2.5.5. Multiply instructions

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

MUL	Multiply two 32 bit values MUL Rd, Rm	Rd = (Rm * Rd) [31:0]

2.2.5.6. Shift instructions

Shift instructions are used to shift or rotate the 32 bit value placed in registers to either left or right direction, according to the type of instruction.

ASR	arithmetic shift right ASR Rd, Rm, # immediate	Rd ⇐ Rm >> immediate C flag ⇐ Rm [immediate – 1]
	ASR Rd, Rs	Rd ⇐ Rd >> Rs, C flag ⇐ Rd [Rs – 1]

LSL	logical shift left LSL Rd, Rm, # immediate	Rd ⇐ Rm << immediate C flag ⇐ Rm [32 - immediate]
	LSL Rd, Rs	Rd ⇐ Rd << Rs, C flag ⇐ Rd [32 - Rs]
LSR	logical shift right LSR Rd, Rm, #immediate	Rd ⇐ RM >> immediate, C flag ⇐ Rd [immediate–1]
	LSR Rd, Rs	Rd ⇐ Rd >> Rs, C fly ⇐ Rd [Rs – 1]
ROR	rotate right a 32 bit value ROR Rd, Rm, # immediate	Rd ⇐ Rd RIGHT_ROTATE Rs C flag ⇐ Rd [Rs – 1]

2.2.6. Branch instructions

There are two variations of branch instructions used in thumb. The branch instructions change the flow of execution or is used to call a routine.

B	branch B label	PC⇐ label
BL	branch with link BL label	PC ⇐ label LR ⇐ (instruction address after the BL) + 1

The first instruction is similar to the ARM version and is conditionally executed, its branch range is limited to a signed 8 bit immediate or – 256 to + 254 bytes. The second version removes the conditional part of the instruction, its range is in between – 2048 and + 2046 bytes.

2.2.7. Load - Store Instructions

Load and store instructions are used for loading data to register or storing data from register.

2.2.7.1. Single register load store instructions

These instructions use two pre indexed addressing modes. They are;

i) Offset by register: It uses a base register Rn plus the register offset Rm.

ii) Offset by immediate: It uses a base reigster Rn plus a 5 bit immediate or a value dependent on the data size.

Syntax:

<LDR | STR>{<B | H>} Rd, [Rn, #immediate]
LDR{<H | SB | SH>} Rd, [Rn,Rm]
STR{<B | H>} Rd, [Rn, Rm]
LDR Rd, [pc, #immediate]
<LDR | STR> Rd, [sp, #immediate]

LDR	load word into a register	Rd ⇐ mem32 [address]
STR	save 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])

Example

```
PRE     mem32[0x90000] = 0x00000001
        mem32[0x90004] = 0x00000002
        mem32[0x90008] = 0x00000003
        r0 = 0x00000000
        r1 = 0x00090000
        r2 = 0x00000004

        LDR r0, [r1, r4] ; register

POST    r0 = 0x00000002
        r1 = 0x00090000
        r4 = 0x00000004

        LDR r0, [r1, #0x4] ;immediate

POST    r0 = 0x00000002
```

Both instructions carry out the same operation. The first instruction uses register offset and the second one uses immediate offset.

In the first instruction, the content of array of memory locations specified by the sum of registers r1 and r4 is loaded in register r0.

In the second instruction, the content of array of memory locations specified by register r1 with the sum of immediate value is loaded in register r0.

2.2.7.2. Multiple register load - store instructions

The thumb versions of the load - store multiple instructions are reduced forms of the ARM load store multiple instructions. They only support the increment after (IA) addressing mode.

Syntax:

<LDM | STM>IA Rn!, {low Register list}

LDMIA	load multiple registers	{Rd}^{*N} ⇐ meme32 [Rn + 4 *N], Rn = Rn + 4 * N
STMIA	save multiple registers	{Rd}^{*N} ⇒ mem32 [Rn + 4 *N], Rn = Rn + 4 * N

Here N is the number of registers in the list of registers.

Example:

```
PRE   r1 = 0x00000001
      r2 = 0x00000002
      r3 = 0x00000003
      r4 = 0x9000
      STMIA r4!, {r1, r2, r3}
POST  mem32 [0x9000] = 0x00000001
      mem32 [0x9004] = 0x00000002
      mem32 [0x9008] = 0x00000003
      r4 = 0x900c
```

This example saves registers r1 to r3 to memory addresses 0X9000 to 0X900C. It also updates base register r4.

2.2.8. Stack Instructions

The Thumb stack operations are different from the equivalent ARM instructions because they use the more traditional POP and PUSH concept.

Syntax : POP {low_register_list{, pc}}

PUSH {low_register_list {, lr}}

POP	POP registers from the stack	$Rd^{*N} \Leftarrow$ mem 32 [SP + 4 x N], SP = SP + 4*N
PUSH	PUSH registers on to the stack	$Rd^{*N} \Rightarrow$ mem 32[SP + 4 x N], SP = SP – 4 x N

There is no stack pointer specified in these instructions. This is because the stack pointer is fixed at register *r13* in Thumb operations and SP is automatically updated. The list of registers is limited to the low registers *r0* to *r7*.

The PUSH register list also can include the link register LR. Similarly the POP register list can include the PC. This provides support for subroutine entry and exit.

Example

```
; Call subroutine
BL    ThumbRoutine
; continue
ThumbRoutine
PUSH  {r1, LR}          ; enter subroutine
MOV   r0, #2
POP   {r1, PC}          ; return from subroutine
```

This example uses POP and PUSH instructions. The subroutine Thumb Routine is using a branch with link (BL) instruction.

The link register LR is pushed onto the stack with register r1. Upon return, register r1 is popped off the stack as well as the return address being loaded into the PC. This returns from the subroutine.

2.2.9. Software Interrupt Instruction

Similar to the ARM equivalent, the Thumb software interrupt (SWI) instruction causes a software interrupt exception. If any interrupt or exception flag is raised in Thumb state, the processor automatically reverts back to ARM state to handle the exception.

Syntax: SWI interrupt

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) CPSR T = 0 (ARM state)

The Thumb SWI instruction has the same effect and nearly the same syntax as the ARM equivalent. It differs in that the SWI number is limited to the range 0 to 255 and it is not conditionally executed.

Example: 0X0000 8000 SWI 0X45

This example shows the execution of a Thumb SWI instruction. Note that the processor goes from Thumb state to ARM state after execution.

It's pre and post processing responses are shown below.

```
PRE   CPSR =nzcVqifT_USER
      PC = 0x00008000
      LR = 0x003FFFFF      ;LR = r14
      r0 = 0x12
POST  CPSR = nzcVqIft_SVC
      SPSR = nzcVqifT_USER
      PC = 0x00000008
      LR = 0x00008002
      r0 = 0x12
```

2.3. Simple Programs

2.3.1. To write an assembly language program for adding two values.

```
LDR r1, value 1 ; Load the first number in r1
LDR r2, value2  ; Load the second number in r2
ADD r0, r1, r2  ; Add the two numbers
LDR r4, address ; Load the address for storing the result
STR r0, [r4]    ; Store the result
END             ; End
```

2.3.2. To write an assembly language program for adding two 32 bit values placed in memory locations, and store the result in other memory locations

```
LDR r0, 0x00009000 ; Load the address of first data.
LDR r1, [r0]       ; Place the first data in r1
```

```
LDR r0, r0, #0x4    ; Adjust the pointer for second data
LDR r2, [r0]        ; Place the second data in r2
ADD r3, r1, r2      ; Add the two data
LDR r0, r0 #0x4     ; Adjust the pointer for storing data
STR r3, [r0]        ; Store the result
END
```

2.3.3. To write an assembly language program for subtracting two 32 bit values

```
LDR r1, value 1 ; Load the first data in r1
LDR r2, value 2 ; Load the second data in r2.
SUB r0, r1, r2  ; Subract the two values.
                          (r0) ⇐ (r1) – (r2)
LDR r4, address; Load the address for storing the result
STR r0, [r4]    ; Store the result
END             ; End
```

2.3.4. To write an assembly language program for multiplying two values.

```
LDR r1, value1      ; Load the first number in r1
LDR r2, value 2     ; Load the second number in r2
MUL r0, r1, r2      ; Multiply the two values
LDR r4, address     ; Load the address for storing the
                      result
STR r0, [r4]        ; Store the result.
END
```

2.3.5. To write an assembly language program for multiplying two values placed in memory locations and store the result at another memory locations

```
LDR r0, 0X00008000 ; Load the address of data in r0
LDR r1, [r0], #4   ; Place the first data in r1
LDR r2, [r0], #4   ; Place the second data in r2
UMULL r3, r4, r1, r2 ; Multiply the two values
STR r4, [r0], #4   ; Store the result (RdLo)
STR r3, [r0]       ; Store the result(RdHi)
END
```

2.3.6. To write an assembly language program for adding two values placed in memory locations and store the result at another memory locations

```
      LDR r0, 0X00009000   ; Load the address of data in r0
      LDR r1, [r0], #4     ; Place the first data in r1
      LDR r2, [r0], #4     ; Place the second data in r2
      LDR r4, 0X0000 0000  ; Initialise r4 for carry
      ADDS r5, r1, r2      ; Add the two values
      BCC LOOP             ; If no carry, jump to LOOP
      ADD r4, r4, #1       ; Set carry as 1
LOOP  STR r5, [r0], #4     ; Store the result
      STR r4, [r0]         ; Store the carry
      END
```

2.4. ARM Instructions

Mnemonics	Description
ADC	add two 32-bit values and carry
ADD	add two 32-bit values
AND	logical bitwise AND of two 32-bit values
B	branch relative +/- 32 MB
BIC	logical bit clear (AND NOT) of two 32-bit values
BKPT	breakpoint instructions
BL	relative branch with link
BLX	branch with link and exchange
BX	branch with exchange
CDP CDP2	coprocessor data processing operation
CLZ	count leading zeros
CMN	Compare negative two 32-bit values
CMP	compare two 32-bit values
EOR	logical exclusive OR of two 32-bit values
LDC LDC2	load to coprocessor single or multiple 32-bit values
LDM	load multiple 32-bit words from memory to ARM registers
LDR	load a single value from a virtual address in memory
MCR MCR2 MCRR	move to coprocessor from an ARM register or registers
MLA	multiply and accumulate 32-bit values
MOV	move a 32-bit value into a register

MRC MRC2 MRRC	move to ARM register or registers from a coprocessor
MRS	move to ARM register from a status register (CPSR or SPSR)
MSR	move to a status register (CPSR or SPSR) from an ARM register
MUL	multiply two 32-bit values
MVN	move the logical NOT of 32-bit value into a register
ORR	logical bitwise OR of two 32-bit values
PLD	preload hint instuction
QADD	signed saturated 32-bit add
QDADD	signed saturated double and 32-bit add
QDSUB	signed saturated double and 32-bit subtract
QSUB	signed saturated 32-bit subtract
RSB	reverse subtract of two 32-bit values
RSC	reverse subtract with carry of two 32-bit integers
SBC	subtract with carry of two 32-bit values
SMLAxy	signed multiply accumulate instructions (16 x 16) + 32 = 32 bit)
SMLAL	signed multiply accumulate long ((32 x 32) + 64 = 64 bit))
SMLALxy	signed multiply accumulate long ((16 x 16) + 64 = 64 bit)
SMLAWy	signed multiply accumulate instruction (((32 x 16) >> 16) + 32 = 32 bit)
SMULL	signed multiply long (32 x 32 = 64 bit)

SMULxy	signed multiply instructions (16 x 16 = 32-bit)
SMULWy	signed multiply instruction ((32 x 16) >> 16 = 32 bit.)
STC STC2	store to memory single or multiple 32-bit values from coprocessor
STM	store multiple 32-bit registers to memory
STR	store register to a virtual address in memory
SUB	subtract two 32bit values
SWI	software interrupt
SWP	swap a word/byte in memory with a register, without interruption
TEQ	test for equality of two 32-bit values
TST	test for bits in a 32-bit value
UMLAL	unsigned multiply accumulate long ((32 x 32) + 64 = 64-bit)
UMULL	unsigned multiply long (32 x 32 = 64-bit)