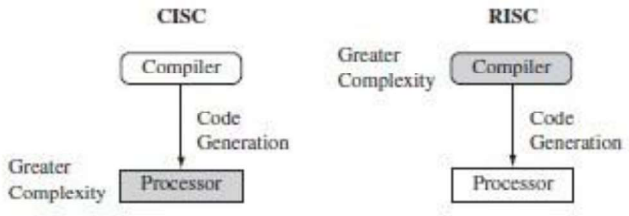
ARM Embedded Systems

1.1 The RISC design philosophy :

The ARM core uses RISC architecture. RISC is a design philosophy aimed at delivering simple but powerful instructions that execute within a single cycle at a high clock speed. The RISC philosophy concentrates on reducing the complexity of instructions performed by the hardware because it is easier to provide greater flexibility and intelligence in software rather than hardware. As a result, a RISC design places greater demands on the compiler. In contrast, the traditional complex instruction set computer (CISC) relies more on the hardware for instruction functionality, and consequently the CISC instructions are more complicated. Figure1 illustrates these major differences.



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

The RISC philosophy is implemented with four major design rules:

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

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

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

4. Load-store architecture: The processor operates on data held in registers. Separate load and store instructions transfer data between the register bank and external memory. Memory accesses are costly, so separating memory accesses from data processing provides an advantage because you can use data items

held in the register bank multiple times without needing multiple memory accesses. In contrast, with a CISC design the data processing operations can act on memory directly.

1.2 The ARM Design Philosophy

There are a number of physical features that have driven the ARM processor design.

- First, portable embedded systems require some form of battery power. The ARM processor has been designed to be small to reduce power consumption and extend battery operation—essential for applications such as mobile phones and personal digital assistants (PDAs).

- High code density is another major requirement since embedded systems have limited memory due to cost and physical size restrictions. High code density is useful for applications that have limited on-board memory, such as mobile phones and mass storage devices.

- Embedded systems are price sensitive and use slow and low-cost memory devices. For high- volume applications like digital cameras, every cent has to be accounted for in the design. The ability to use low-cost memory devices produces substantial savings.

- Another important requirement is to reduce the area of the die taken up by the embedded processor. For a single-chip solution, the smaller the area used by the embedded processor, the more available space for specialized peripherals. This in turn reduces the cost of the design and manufacturing since fewer discrete chips are required for the end product.

- ARM has incorporated hardware debug technology within the processor so that software engineers can view what is happening while the processor is executing code. With greater visibility, software engineers can resolve an issue faster, which has a direct effect on the time to market and reduces overall development costs.

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

1.3 Instruction Set for Embedded Systems

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

- Variable cycle execution for certain instructions—Not every ARM instruction executes in a single cycle. For example, load-store-multiple instructions vary in the number of execution cycles depending upon the number of registers being transferred. The transfer can occur on sequential memory addresses,

which increases performance since sequential memory accesses are often faster than random accesses. Code density is also improved since multiple register transfers are common operations at the start and end of functions.

■ Inline barrel shifter leading to more complex instructions—The inline barrel shifter is a hardware component that preprocesses one of the input registers before it is used by an instruction. This expands the capability of many instructions to improve core performance and code density.

■ Thumb 16-bit instruction set—ARM enhanced the processor core by adding a second 16-bit instruction set called Thumb that permits the ARM core to execute either 16- or 32-bit instructions. The 16-bit instructions improve code density by about 30% over 32-bit fixed-length instructions.

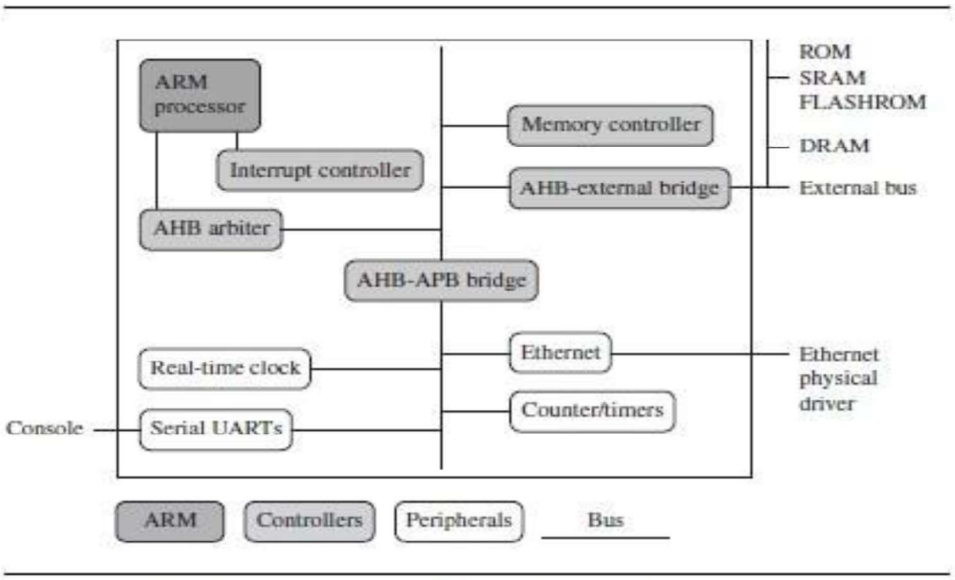
■ Conditional execution—An instruction is only executed when a specific condition has been satisfied. This feature improves performance and code density by reducing branch instructions.

■ Enhanced instructions—The enhanced digital signal processor (DSP) instructions were added to the standard ARM instruction set to support fast 16×16-bit multiplier operations and saturation. These instructions allow a faster-performing ARM processor in some cases to replace the traditional combinations of a processor plus a DSP.

1.4 Embedded System Hardware

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

■ **The *ARM processor*** controls the embedded device. Different versions of the ARM processor are available to suit the desired operating characteristics. An ARM processor comprises a core (the execution engine that processes instructions and manipulates data) plus the surrounding components that interface it with a bus. These components can include memory management and caches.
■ ***Controllers*** coordinate important functional blocks of the system. Two commonly found controllers are interrupt and memory controllers.
■ The ***peripherals*** provide all the input-output capability external to the chip and are responsible for the uniqueness of the embedded device.
■ A ***bus*** is used to communicate between different parts of the device.



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

1.2.1 ARM Bus Technology: Embedded systems use different bus technologies. The most common PC bus technology, the Peripheral Component Interconnect (PCI) bus, connects such devices as video cards and hard disk controllers to the x86 processor bus. This type of technology is external or off-chip

- In embedded devices use an on-chip bus that is internal to the chip and that allows different peripheral devices to be interconnected with an ARM core.

- There are two different classes of devices attached to the bus. The ARM processor core is a ***bus master***—a logical device capable of initiating a data transfer with another device across the same bus. Peripherals tend to be ***bus slaves***—logical devices capable only of responding to a transfer request from a bus master device.

- A bus has two architecture levels. The first is a **physical level** that covers the electrical Characteristics and bus width (16, 32, or 64 bits). The second level deals with ***protocol***—the

logical rules that govern the communication between the processor and a peripheral. ARM is primarily a design company.

1.2.2 AMBA Bus Protocol: The **Advanced Microcontroller Bus Architecture** (AMBA) been widely adopted as the on-chip bus architecture used for ARM processors. The first AMBA buses introduced were the **ARM System Bus** (ASB) and the **ARM Peripheral Bus**(APB). Later ARM introduced another bus design, called the **ARM High Performance Bus** (AHB). AHB provides higher data throughput than ASB because it is based on a centralized multiplexed bus scheme rather than the ASB bidirectional bus design.

ARM has introduced two variations on the AHB bus: **Multi-layer AHB** and **AHB-Lite**. In contrast to the original AHB, which allows a single bus master to be active on the bus at any time, the Multi-layer

AHB bus allows multiple active bus masters. AHB-Lite is a subset of the AHB bus and it is limited to a single bus master.

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

1.3 Embedded System Software

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

Each software component in the stack uses a higher level of abstraction to separate the code from the hardware device.

The initialization code is the first code executed on the board and is specific to a particular target or group of targets. It sets up the minimum parts of the board before handing control over to the operating system.

The operating system provides an infrastructure to control applications and manage hardware system resources. Many embedded systems do not require a full operating system

The device drivers are the third component. They provide a consistent software interface to the peripherals on the hardware device.

Finally, an application performs one of the tasks required for a device. For example,a mobile phone might have a diary application. There may be multiple applications running on the same device, controlled by the operating system.

The software components can run from ROM or RAM. ROM code that is fixed on the device (for example, the initialization code) is called *firmware*.

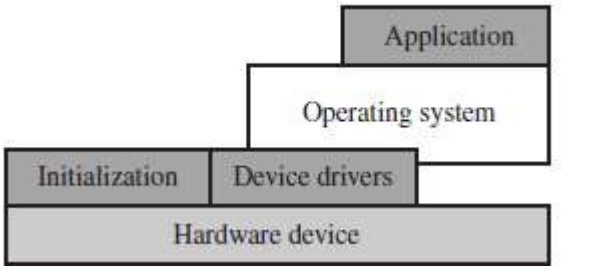


Fig:Software Abstraction layers

1.3.1 Initialization (Boot) Code

Initialization code (or boot code) takes the processor from the reset state to a state where the operating system can run. The initialization code handles a number of administrative tasks prior to handing control
over to an operating system image.This different tasks are grouped into three phases: initial hardware configuration, diagnostics, and booting

- Initial hardware configuration involves setting up the target platform so it can boot an image.
- Diagnostic code tests the system by exercising the hardware target to check if the target is in working order. It also tracks down standard system-related issues. The primary purpose of diagnostic code is fault identification and isolation.
- Booting an image is the final phase, but first load the image. Loading an image involves anything from copying an entire program including code and data into RAM, to just copying a data area containing volatile variables into RAM. Once booted, the system hands over control by modifying the program counter to point into the start of the image

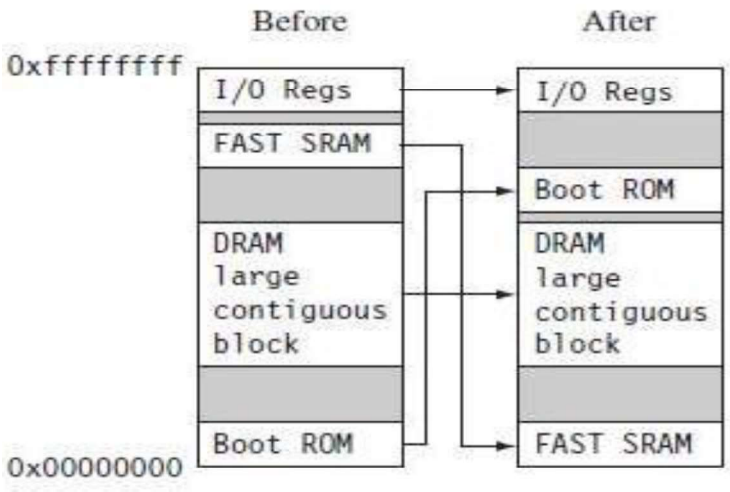


Figure shows memory before and after reorganization. It is common for ARM-based embedded systems to provide for memory remapping because it allows the system to start the initialization code from ROM at power-up.

1.3.2 Operating System

The initialization process prepares the hardware for an operating system to take control. An operating system organizes the system resources: the peripherals, memory, and processing time

ARM processors support over 50 operating systems. Operating systems can be divided into two main categories: **real-time operating systems (RTOSs**) and **platform operating systems**

- RTOSs provide guaranteed response times to events. A hard real-time application requires a guaranteed response to work at all. In contrast, a soft real-time application requires a good response time, but the performance degrades more gracefully if the response time overruns.
- Platform operating systems require a memory management unit to manage large, nonreal- time applications and tend to have secondary storage. The Linux operating system is a typical example of a platform operating system.

1.3.2Applications

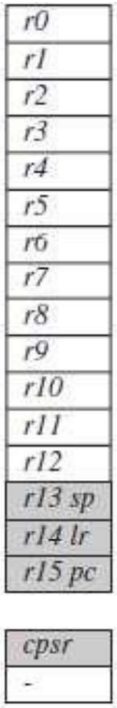
The operating system schedules applications—code dedicated to handling a particular task.An application implements a processing task; the operating system controls the environment. ARM processors can be found in multiple applications.

For example, the ARM processor is found in networking applications like home gateways, DSL modems for high-speed Internet communication, and 802.11 wireless communication.

The mobile device segment is the largest application area for ARM processors because of mobile phones. ARM processors are also found in mass storage devices such as hard drives and imaging products such as inkjet printers—applications that are cost sensitive and high volume.

1.4 Registers

General-purpose registers hold either data or an address. They are identified with the letter *r* prefixed to the register number. For example, register 4 is given the label *r4*. Figure shows the active registers available in *user* mode



Registers available in *user* mode.

There are up to 18 active registers: 16 data registers and 2 processor status registers. The data registers are visible to the programmer as *r0* to *r15*.

The ARM processor has three registers assigned to a particular task or special function:*r13*, *r14*, and *r15*. They are frequently given different labels to differentiate them from the other registers.

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

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

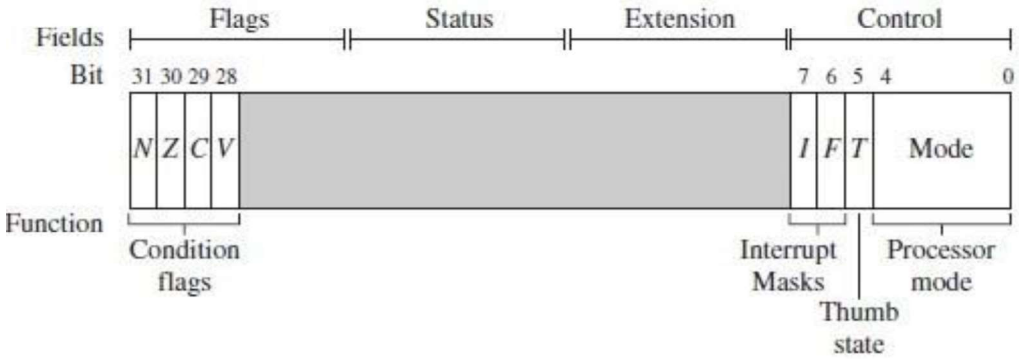
■ **Register *r14*** is called the link register (*lr*) and is where the core puts the return address whenever it calls a subroutine.

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

There are two program status registers: *cpsr* and *spsr* (the current and saved program status registers, respectively).

1.5 ***Current Program Status Register***

The ARM core uses the *cpsr* to monitor and control internal operations. The *cpsr* is a dedicated 32-bit register and resides in the register file.



A generic program status register (*psr*).

Above Figure shows the basic layout of a generic program status register. The *cpsr* is divided into four fields, each 8 bits wide: flags, status, extension, and control. In current designs the extension and status fields are reserved for future use. The control field contains the processor mode, state, and interrupts mask bits. The flags field contains the condition flags.

Some ARM processor cores have extra bits allocated. For example, the *J* bit, which can be found in the flags field, is only available on Jazelle-enabled processors

1.5.1 Processor Modes

The processor mode determines which registers are active and the access rights to the *cpsr* register itself. Each processor mode is either privileged or nonprivileged:

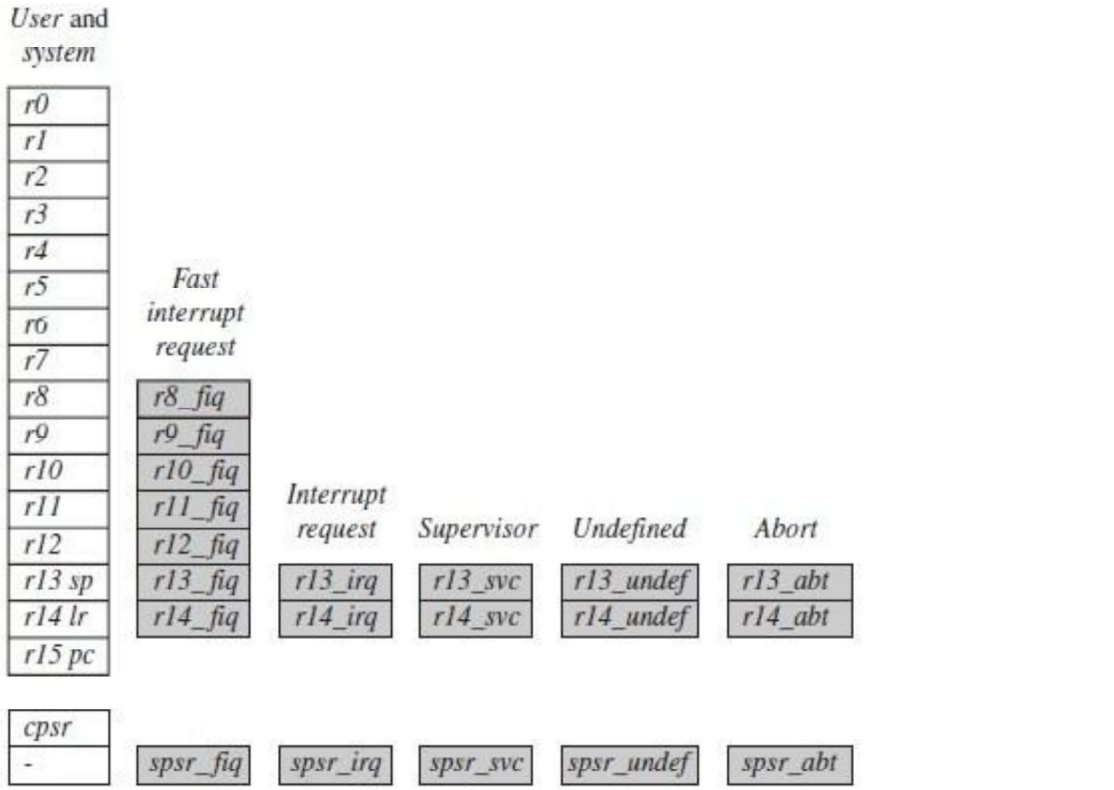
A privileged mode allows full read-write access to the *cpsr*. Conversely, a nonprivileged mode only allows read access to the control field in the *cpsr* but still allows read-write access to the condition flags. There are seven processor modes in total: six privileged modes and one nonprivileged mode(*user*).

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

1.5.2 Banked Registers

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

If we change processor mode, a banked register from the new mode will replace an existing register. For example, when the processor is in the *interrupt request* mode, the instructions can still access registers named *r13* and *r14*. 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



Complete ARM register set.

The processor mode can be changed by a program that writes directly to the *cpsr* or by hardware when the core responds to an exception or interrupt. The following exceptions and interrupts cause a mode change: *reset*, *interrupt request*, *fast interrupt request*, *software interrupt*, *data abort*, *prefetch abort*, and *undefined instruction*.

Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location.

. The figure shows the core changing from *user* mode to *interrupt request* mode..This change causes *user* registers *r13* and *r14* to be banked. The *user* registers are replaced with registers *r13_irq* and *r14_irq*, respectively. *r14_irq* contains the return address and *r13_irq* contains the stack pointer for *interrupt request* mode.

Figure also shows a new register appearing in *interrupt request* mode: the saved program status register *(spsr)*, which stores the previous mode's *cpsr.*

The *cpsr* is not copied into the *spsr* when a mode change is forced due to a program writing directly to the *cpsr*. The saving of the *cpsr* only occurs when an exception or interrupt is raised.

The following table lists the various modes and the associated binary patterns.

Processor mode.

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

1.5.3 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.

The Jazelle *J* and Thumb *T* bits in the *cpsr* reflect the state of the processor. When both *J* and *T* bits are 0, the processor is in ARM state and executes ARM instructions. This is the case when power is applied to the processor. When the T bit is 1, then the processor is in Thumb state.

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

ARM and Thumb instruction set features.

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

Jazelle instruction set features.

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

1.5.4 Interrupt Masks

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

1.5.5 Condition Flags

Condition flags are updated by comparisons and the result of ALU operations

Condition flags.

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

With processor cores that include the DSP extensions, the *Q* bit indicates if an overflow or saturation has occurred in an enhanced DSP instruction.

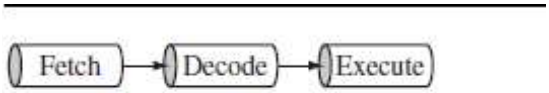
When a bit a binary 1 is use a capital letter; when a bit is a binary 0, use a lowercase letter. For the condition flags a capital letter shows that the flag has been set. For interrupts a capital letter shows that an interrupt is disabled.

In the *cpsr* example shown in the following Figure, the *C* flag is the only condition flag set. The rest *nzvq* flags are all clear. The processor is in ARM state because neither the Jazelle *j* nor Thumb *t* bits are set. The IRQ interrupts are enabled, and FIQ interrupts are disabled

Example: *cpsr = nzCvqjiFt_SVC.*

1.6 Pipeline

A pipeline is the mechanism a RISC processor uses to execute instructions. Using a pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed



ARM7 Three-stage pipeline.

Figure shows a three-stage pipeline:

■ *Fetch* loads an instruction from memory.

■ *Decode* identifies the instruction to be executed.

■ *Execute* processes the instruction and writes the result back to a register.

Figure 4.1 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.

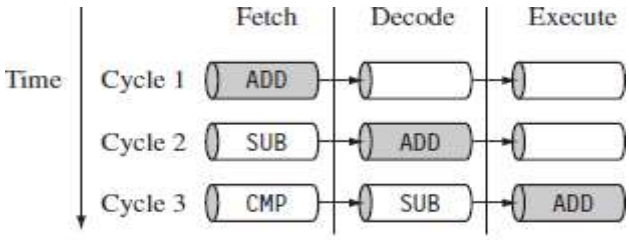


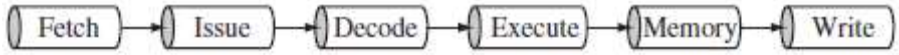
Figure 4. 1 Pipelined instruction sequence

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

As the pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency. This in turn increases the performance. The system

latency also increases because it takes more cycles to fill the pipeline before the core can execute an instruction. The increased pipeline length also means there can be data dependency between certain stages. You can write code to reduce this dependency by using *instruction scheduling*.

ARM9 five-stage pipeline.



ARM10 six-stage pipeline.

The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages, The ARM10 increases the pipeline length still further by adding a sixth stage, as shown in Figure.

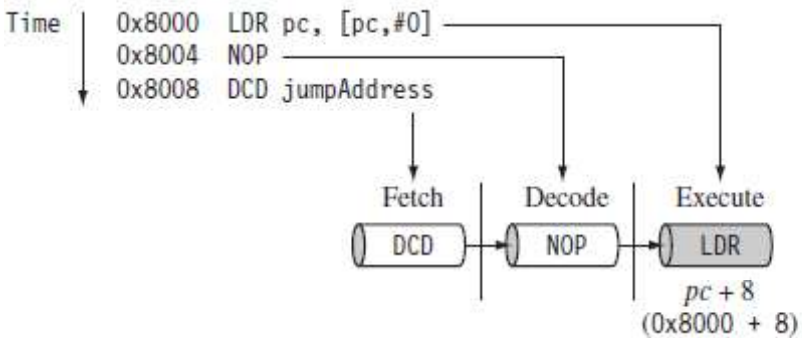


Fig: PC=address+8

There are three other characteristics of the pipeline

- First, the execution of a branch instruction or branching by the direct modification of the *pc* causes the ARM core to flush its pipeline.
- Second, ARM10 uses branch prediction, which reduces the effect of a pipeline flush by predicting possible branches and loading the new branch address prior to the execution of the instruction.
- Third, an instruction in the execute stage will complete even though an interrupt has been raised. Other instructions in the pipeline will be abandoned, and the processor will start filling the pipeline from the appropriate entry in the vector table.
-

1.7 Exceptions, Interrupts, and the Vector Table

When an exception or interrupt occurs, the processor sets the *pc* to a specific memory address. The address is within a special address range called the ***vector table***. The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt. The memory map address 0x00000000 is reserved for the vector table, a set of 32-bit words.

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

The vector table.

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

■ ***Reset vector*** **i**s the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.

■ ***Undefined instruction vector*** is used when the processor cannot decode an instruction.

■ ***Software interrupt vector*** is called when you execute a SWI instruction. The SWI instruction is Frequently used as the mechanism to invoke an operating system routine.

■ ***Prefetch abort vector*** occurs when processor Fetches an instruction from an illegal address, the instruction is flagged as invalid.The actual abort occurs in the decode stage.

■ ***Data abort vector*** is similar to a prefetch abort but is raised when an instruction attempts to access data memory at an illegal address.

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

- ***Fast interrupt request vector:*** The processor external fast interrupt request pin is asserted (LOW) and the F bit in the CPSR is clear.

Difference between Microprocessors and Microcontrollers

s.no	Microprocessors	Microcontrollers
1	Microprocessor is a heart of computer	Microprocessor is a heart of Embedded system
2	It is just a processor. Memory and I/O components have to be connected externally	Micro controller has external processor along with internal memory and i/O components
3	MP based system require more hardware	MC based system require more hardware
4	Cost of the system is more	Cost of the system is less
5	Due to external components, the entire power consumption is high. Hence it is not suitable to used with devices running on stored power like batteries	Since external components are low, total power consumption is less and can be used with devices running on stored power like batteries.
6	Mainly used in computer system	Mainly used in washing machine,MP3 players
7	INTEl 8086,Pentium series	INTEL 8051,ARM Microcontroller
8	Microprocessors are based on von Neumann model/architecture where program and data are stored in same memory module	Micro controllers are based on Harvard architecture where program memory and Data memory are separate