

UNIT-III ARM ARCHITECTURE

Contents at a glance:

- ✓ RISC Design Philosophy
- ✓ ARM Design Philosophy
- ✓ Registers
- ✓ Current Program Status Register(CPSR)
- ✓ Instruction Pipeline
- ✓ Interrupts and Vector Table
- ✓ Architecture Revision
- ✓ ARM Processor Families.

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.
- Figure 1.1 illustrates these major differences.

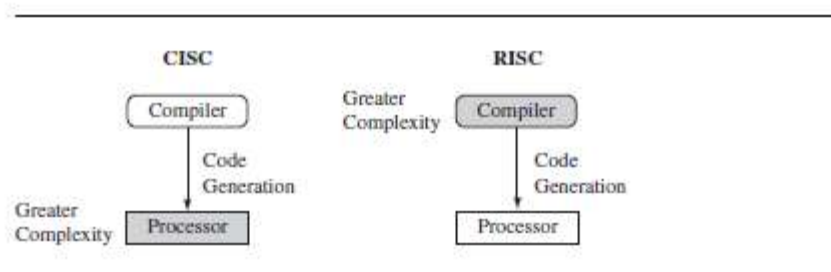


Figure 1.1 CISC vs. RISC. CISC emphasizes hardware complexity, RISC emphasizes compiler complexity.

- 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
 - 2. Pipelines**—The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines. Ideally the pipeline advances by one step on each cycle for maximum throughput. Instructions can be decoded in one pipeline stage.
 - 3. Registers**—RISC machines have a large general-purpose register set. Any register can contain either data or an address. Registers act as the fast local memory store for all data processing operations.

4. Load-store architecture—The processor operates on data held in registers. Separate load and store instructions transfer data between the register bank and external memory. Memory accesses are costly, so separating memory accesses from data processing provides an advantage because you can use data items held in the register bank multiple times without needing multiple memory accesses.

The ARM Design Philosophy:

- There are a number of physical features that have driven the ARM processor design.
- **First**, portable embedded systems require some form of battery power. The ARM processor has been specifically designed to be small to reduce power consumption and extend battery operation—essential for applications such as mobile phones and personal digital assistants (PDAs).
- High code density is another major requirement since embedded systems have limited memory due to cost and/or physical size restrictions. High code density is useful for applications that have limited on-board memory, such as mobile phones and mass storage devices.
- 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.
- 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.

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 pre processes 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.

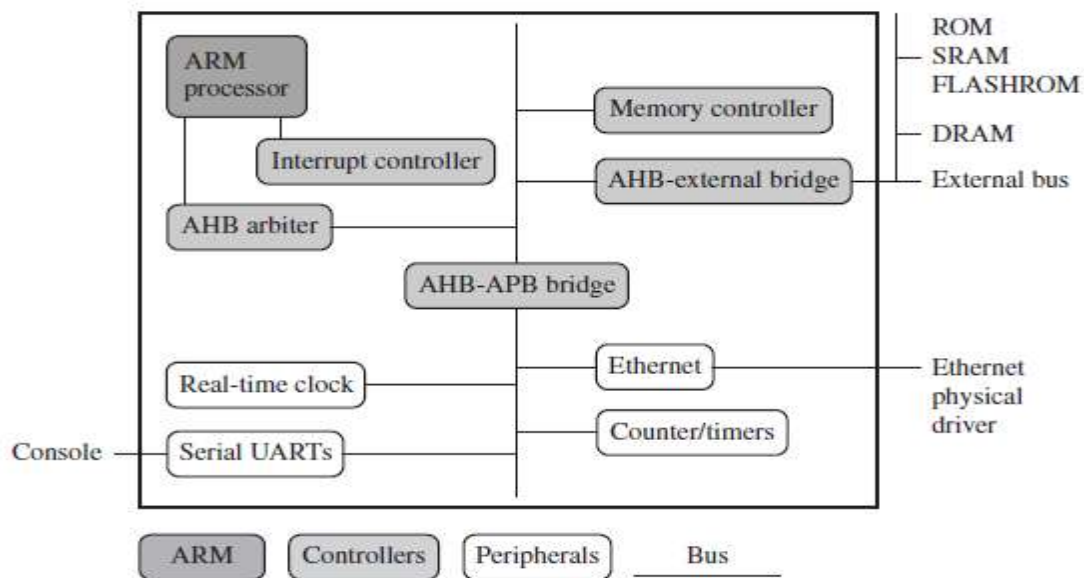
■ **Conditional execution**—An instruction is only executed when a specific condition has been satisfied. This feature improves performance and code density by reducing branch instructions.

■ **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.

- These additional features have made the ARM processor one of the most commonly used 32-bit embedded processor cores.
- Many of the top semiconductor companies around the world produce products based around the ARM processor.

Embedded System Hardware:

- Figure below shows a typical embedded device based on an ARM core.



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

- Each box represents a feature or function. The lines connecting the boxes are the buses carrying data.
- The device can be separated into four main hardware components:
 - The ARM processor controls the embedded device. An ARM processor comprises a core 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.

- 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.
- When a peripheral or device requires attention, it raises an interrupt to the processor. An interrupt controllers are programmable that allows software to determine which peripheral or device can interrupt the processor at any specific time by setting the appropriate bits in the interrupt controller registers.
- 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.
 - 80x86 based PCs use common PC bus technology, the Peripheral Component Interconnect (PCI) bus. This type of technology is external or off-chip and is built into the motherboard of a PC.
 - Embedded devices use an on-chip bus that is internal to the chip and that allows different peripheral devices to be interconnected with an ARM core.
 - 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.
 - The Advanced Microcontroller Bus Architecture (AMBA) was introduced in 1996 and has been widely adopted as the on-chip bus architecture used for ARM processors.
 - The first AMBA buses introduced were
 - ARM System Bus (ASB)
 - ARM Peripheral Bus (APB)
 - Later ARM introduced another bus design, called
 - the ARM High Performance Bus (AHB)

ARM Processor Fundamentals:

- A programmer can think of an ARM core as functional units connected by data buses, as shown in Figure 2.1, where, the arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area.
- The figure shows not only the flow of data but also the abstract components that make up an ARM core.
- Data enters the processor core through the *Data* bus. The data may be an instruction to execute or a data item.
- Figure below shows a Von Neumann implementation of the ARM— data items and instructions share the same bus. In contrast, Harvard implementations of the ARM use two different buses.

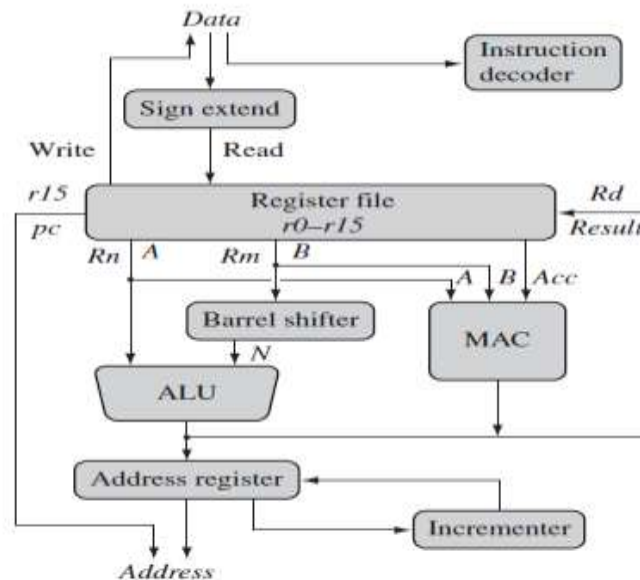


Figure 2.1 ARM core dataflow model.

- 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 *load-store architecture*.
- This means it has two instruction types for transferring data in and out of the processor:
 - Load instructions copy data from memory to registers in the core
 - 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 solely in registers.
- Data items are placed in the *register file*—a storage bank made up of 32-bit registers.
- Since the ARM core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values. The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.
- ARM instructions typically have two source registers, R_n and R_m , and a single result or destination register, R_d . Source operands are read from the register file using the internal buses A and B, respectively.
- The ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) takes the register values R_n and R_m from the A and B buses and computes a result.
- Data processing instructions write the result in R_d 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 R_m alternatively can be pre processed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses.
- After passing through the functional units, the result in R_d 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.

Registers:

- General-purpose registers hold either data or an address.
- They are identified with the letter **r** prefixed to the register number. For example, register 4 is given the label **r4**.
- Figure 2.2 shows the active registers available in user mode—a protected mode normally used when executing applications. The processor can operate in seven different modes.
- All the registers shown are 32 bits in size.

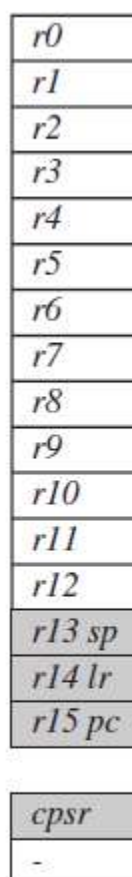


Figure 2.2 Registers available in *user* mode.

- 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 2.2, 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.

- In addition to the 16 data registers, there are two program status registers: **CPSR** and **SPSR** (the current and saved program status registers, respectively).
- The register file contains all the registers available to a programmer. Which registers are visible to the programmer depend upon the current mode of the processor.

Current Program Status Register:

- The ARM core uses the *CPSR* to monitor and control internal operations.
- The *CPSR* is a dedicated 32-bit register and resides in the register file. Figure 2.3 shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future expansion.

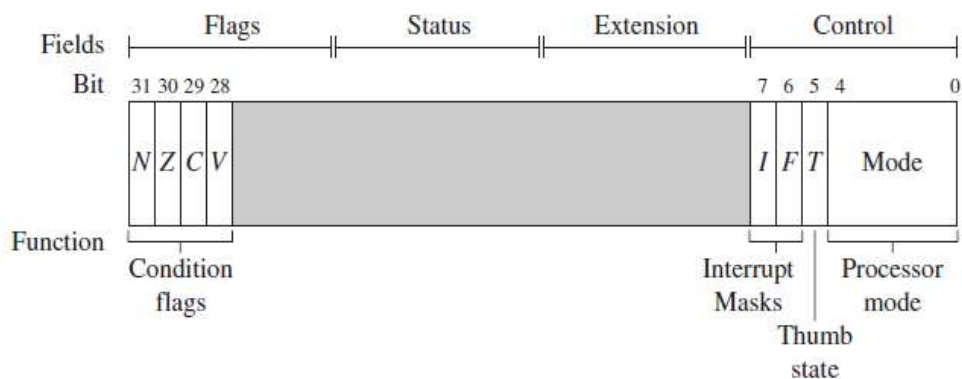


Figure 2.3 A generic program status register (*psr*).

- 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, thumb state, and interrupt mask bits.
- The flags field contains the condition flags.
 - The Negative (N) bit is set when the result is negative in two's complement arithmetic.
 - The Zero (Z) bit is set when every bit of the result is zero.
 - The carry (C) bit is set when there is a carry out of the operation.
 - The overflow (V) bit is set when an arithmetic operation results in an overflow.

Examples: Status bit computation in the ARM

- **-1 + 1 = 0:** for this computation status of flags are
 $0xffffffff + 0x1 = 0x0$; NZCV = 0110.
- **0 - 1 = -1:** for this computation status of flags are
 $0x0 - 0x1 = 0x0 + 0xffffffff = 0xffffffff$; NZCV = 1000.
- **$2^{31} - 1 + 1 = -2^{31}$:** for this computation status of flags are

$0x7ffffff + 0x1 = 0x80000000$; NZCV = 0101.

I: *I* = 1 disables IRQ interrupts.

F: *F* = 1 disables FIQ interrupts.

T: *T* = 1 indicates Thumb state. *T* = 0 indicates ARM state.

MODE: The current processor mode. See Table B.3

Table B.3 Decoding table for *mode*.

Binary	Hex	<i>mode</i>
10000	0x10	<i>user mode</i> (<i>_usr</i>)
10001	0x11	<i>FIQ mode</i> (<i>_fiq</i>)
10010	0x12	<i>IRQ mode</i> (<i>_irq</i>)
10011	0x13	<i>supervisor mode</i> (<i>_svc</i>)
10111	0x17	<i>abort mode</i> (<i>_abt</i>)
11011	0x1B	<i>undefined mode</i> (<i>_und</i>)
11111	0x1F	<i>system mode</i>

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*.
 - 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 (*abort*, *fast interrupt request*, *interrupt request*, *supervisor*, *system*, and *undefined*) and one nonprivileged mode (*user*).

Table 2.1 Processor mode.

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

- The processor enters **abort mode** when there is a failed attempt to access memory.
- **Fast interrupt request** and **interrupt request modes** correspond to the two interrupt levels available on the ARM processor.
- **Supervisor mode** is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in.
- **System mode** is a special version of *user* mode that allows full read-write access to the *CPSR*.

- **Undefined mode** is used when the processor encounters an instruction that is undefined or not supported by the implementation.
- **User mode** is used for programs and applications.

Banked Registers:

- Figure 2.4 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*.

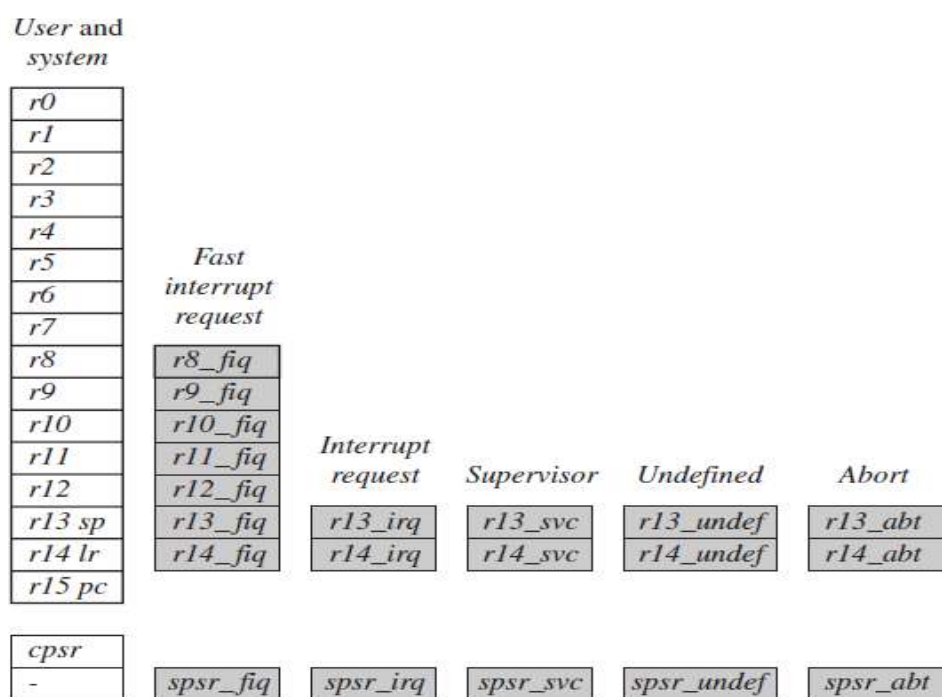


Figure 2.4 Complete ARM register set.

- Every processor mode except *user* mode can change mode by writing directly to the mode bits of the *CPSR*.
- All processor modes except system mode have a set of associated banked registers that are a subset of the main 16 registers.
- A banked register maps one-to-one onto a user mode register. If you change processor mode, a banked register from the new mode will replace an existing register..
- For example, when the processor is in the *interrupt request* mode, the instructions you execute still access registers named *r13* and *r14*.

- However, these registers are the banked registers *r13_irq* and *r14_irq*. The *user* mode registers *r13_usr* and *r14_usr* are not affected by the instruction referencing these registers. A program still has normal access to the other registers *r0* to *r12*.
- The processor mode can be changed by a program that writes directly to the *CPSR* 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.

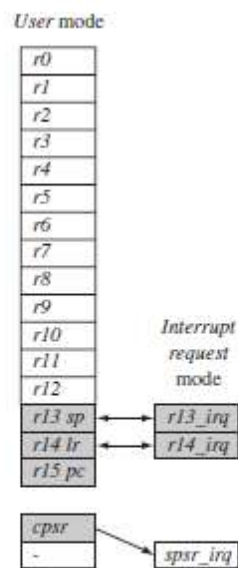
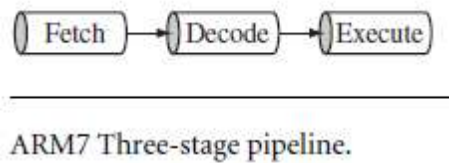


Figure 2.5 Changing mode on an exception.

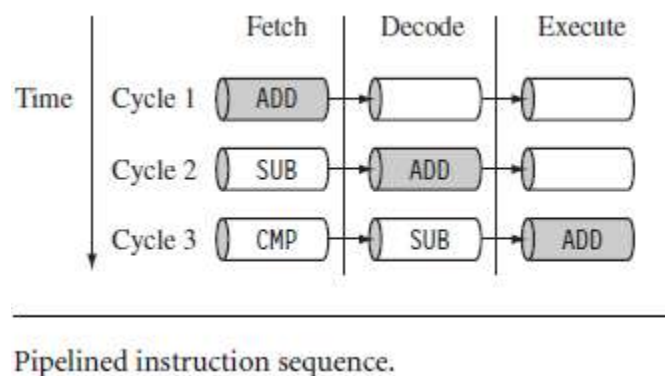
- The above figure shows the core changing from *user* mode to *interrupt request* mode, which happens when an *interrupt request* occurs due to an external device raising an interrupt to the processor core.
- This change causes *user* registers *r13* and *r14* to be banked. The *user* registers are replaced with registers *r13_irq* and *r14_irq*, respectively.
- Note *r14_irq* contains the return address and *r13_irq* contains the stack pointer for *interrupt request* mode.
- The above figure shows a new register appearing in *interrupt request* mode: the save program status register (*spsr*), which stores the previous mode's *cpsr*.
- To return back to *user* mode, a special return instruction is used that instructs the core to restore the original *cpsr* from the *spsr_irq* and bank in the *user* registers *r13* and *r14*.
- Note that the *spsr* can only be modified and read in privileged mode. There is no *spsr* available in *user* mode.
- Another important feature to note is that the *cpsr* is not copied into the *spsr* when a mode change is forced due to a program writing directly to the *cpsr*. The saving of the *cpsr* only occurs when an exception or interrupt is raised.

Instruction Pipeline:

- A pipeline is the mechanism a RISC processor uses to execute instructions.
- Pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed.
- The below figure shows a three-stage pipeline in ARM7:



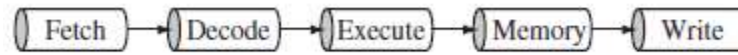
- **Fetch** loads an instruction from memory.
 - **Decode** identifies the instruction to be executed.
 - **Execute** processes the instruction and writes the result back to a register.
- The below figure illustrates the pipeline using a simple example.
 - It shows a sequence of three instructions being fetched, decoded, and executed by the processor.
 - Each instruction takes a single cycle to complete after the pipeline is filled.



- The three instructions are placed into the pipeline sequentially.
 - **First cycle:** the core fetches the ADD instruction from memory.
 - **Second cycle:** the core fetches the SUB instruction and decodes the ADD instruction.
 - **Third cycle:** The core executes the ADD instruction is executed, the SUB instruction is decoded, and the CMP instruction is fetched.
- This procedure is called *filling the pipeline*.
- 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.

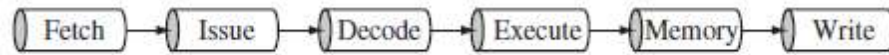
Disadvantage of pipeline length increasing:

- The system *latency* also increases because it takes more cycles to fill the pipeline before the core can execute an instruction.
- The increased pipeline length also means there can be data dependency between certain stages.
- The pipeline design for each ARM family differs.
- The ARM9 core increases the pipeline length to five stages, as shown in below figure.



ARM9 five-stage pipeline.

- The ARM9 adds a memory and write back stage, which allows the ARM9 to process on average 1.1 Dhrystone MIPS per MHz—an increase in instruction throughput by around 13% compared with an ARM7.
- The ARM10 increases the pipeline length still further by adding a sixth stage, as shown in below figure.

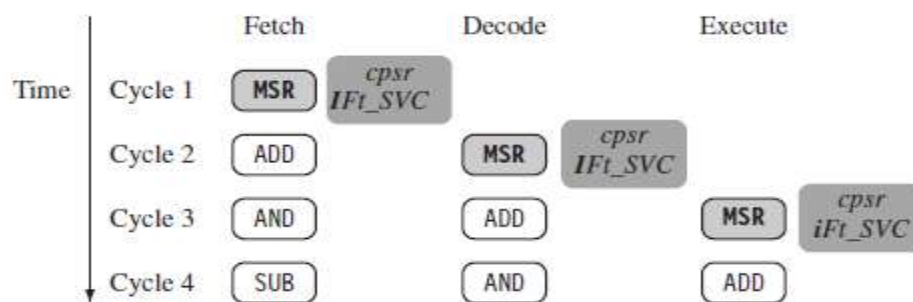


ARM10 six-stage pipeline.

- Even though the ARM9 and ARM10 pipelines are different, they still use the same *pipeline executing characteristics* as an ARM7.
- Code written for the ARM7 will execute on an ARM9 or ARM10.

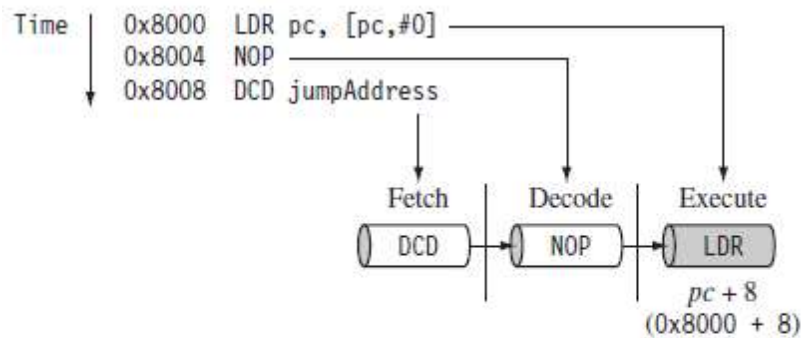
Pipeline Executing Characteristics:

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



ARM instruction sequence.

- The MSR instruction is used to enable IRQ interrupts, which only occurs once the MSR instruction completes the execute stage of the pipeline.
- It clears an *I* bit in the *cpsr* to enable the IRQ interrupts.
- Once the ADD instruction enters the execute stage of the pipeline, IRQ interrupts are enabled.
- The below figure illustrates the use of the pipeline and the program counter *pc*.
- In the execute stage, the *pc* always points to the address of the instruction plus 8 bytes.
- In other words, the *pc* always points to the address of the instruction being executed plus two instructions ahead.



Example: $pc = \text{address} + 8$.

- There are three other characteristics of the pipeline worth mentioning.
 - 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.

Exceptions, Interrupts, and the Vector Table:

- When an exception or interrupt occurs, the processor sets the pc to a specific memory address.
- The address is within a special address range called the *vector table*.
- The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.
- The memory map address 0x00000000 is reserved for the vector table, a set of 32-bit words.
- On some processors the vector table can be optionally located at a higher address in memory (starting at the offset 0xffff0000).
- When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see Table 2.6)

Table 2.6 The vector table.

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

- **Reset vector** is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
- **Undefined instruction vector** is used when the processor cannot decode an instruction.
- **Software interrupt vector** is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.
- **Prefetch abort vector** occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.
- **Data abort vector** is similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions.
- **Interrupt request vector** is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the *cpsr*.
- **Fast interrupt request vector** is similar to the interrupt request but is reserved for hardware requiring faster response times. It can only be raised if FIQs are not masked in the *cpsr*.

Architecture Revisions:

- Before we go on to explain the evolution of the architecture, we must introduce the ARM processor nomenclature.
- The nomenclature identifies individual processors and provides basic information about the feature set.

Nomenclature:

- ARM uses the nomenclature shown in Figure 2.16 to describe the processor implementations.

ARM{x}{y}{z}{T}{D}{M}{I}{E}{J}{F}{-S}

x—family

y—memory management/protection unit

z—cache

T—Thumb 16-bit decoder

D—JTAG debug

M—fast multiplier

I—EmbeddedICE macrocell

E—enhanced instructions (assumes TDMI)

J—Jazelle

F—vector floating-point unit

S—synthesizable version

ARM nomenclature.

- The letters and numbers after the word “ARM” indicate the features a processor may have. In the future the number and letter combinations may change as more features are added. Note the nomenclature does not include the architecture revision information.
- There are a few additional points to make about the ARM nomenclature:

- All ARM cores after the ARM7TDMI include the *TDMI* features even though they may not include those letters after the “ARM” label.
 - The processor *family* is a group of processor implementations that share the same hardware characteristics. For example, the ARM7TDMI, ARM740T, and ARM720T all share the same family characteristics and belong to the ARM7 family.
 - *JTAG* is described by IEEE 1149.1 Standard Test Access Port and boundary scan architecture. It is a serial protocol used by ARM to send and receive debug information between the processor core and test equipment.
 - *EmbeddedICE macrocell* is the debug hardware built into the processor that allows breakpoints and watch points to be set.
 - *Synthesizable* means that the processor core is supplied as source code that can be compiled into a form easily used by EDA tools.
- Table 2.7 shows the significant architecture enhancements from the original architecture version 1 to the current version 6 architecture. One of the most significant changes to the ISA was the introduction of the Thumb instruction set in ARMv4T (the ARM7TDMI processor).

Table 2.7 Revision history.

Revision	Example core implementation	ISA enhancement
ARMv1	ARM1	First ARM processor
ARMv2	ARM2	26-bit addressing
ARMv2a	ARM3	32-bit multiplier
		32-bit coprocessor support
ARMv3	ARM6 and ARM7DI	On-chip cache
		Atomic swap instruction
		Coprocessor 15 for cache management
ARMv3M	ARM7M	32-bit addressing
ARMv4	StrongARM	Separate <i>cpsr</i> and <i>spsr</i>
		New modes— <i>undefined instruction</i> and <i>abort</i>
		MMU support—virtual memory
		Signed and unsigned long multiply instructions
		Load-store instructions for signed and unsigned halfwords/bytes
		New mode— <i>system</i>
		Reserve SWI space for architecturally defined operations
ARMv4T	ARM7TDMI and ARM9T	26-bit addressing mode no longer supported
ARMv5TE	ARM9E and ARM10E	Thumb
		Superset of the ARMv4T
		Extra instructions added for changing state between ARM and Thumb
		Enhanced multiply instructions
		Extra DSP-type instructions
		Faster multiply accumulate
ARMv5TEJ	ARM7EJ and ARM926EJ	Java acceleration
ARMv6	ARM11	Improved multiprocessor instructions
		Unaligned and mixed endian data handling
		New multimedia instructions

ARM Processor Families:

- ARM has designed a number of processors that are grouped into different families according to the core they use.
- The families are based on the ARM7, ARM9, ARM10, and ARM11 cores.
- The postfix numbers 7, 9, 10, and 11 indicate different core designs. The ascending number equates to an increase in performance and sophistication.
- Table 2.9 shows a rough comparison of attributes between the ARM7, ARM9, ARM10, and ARM11 cores.

ARM family attribute comparison.

	ARM7	ARM9	ARM10	ARM11
Pipeline depth	three-stage	five-stage	six-stage	eight-stage
Typical MHz	80	150	260	335
mW/MHz ^a	0.06 mW/MHz	0.19 mW/MHz (+ cache)	0.5 mW/MHz (+ cache)	0.4 mW/MHz (+ cache)
MIPS ^b /MHz	0.97	1.1	1.3	1.2
Architecture	Von Neumann	Harvard	Harvard	Harvard
Multiplier	8 × 32	8 × 32	16 × 32	16 × 32

^a Watts/MHz on the same 0.13 micron process.

^b MIPS are Dhrystone VAX MIPS.