# Module 1

# Microprocessors versus Microcontrollers, ARM Embedded Systems and ARM Processor Fundamentals

**Contents** **Page No.**

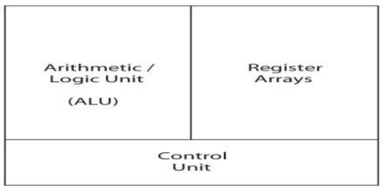
**Module 1- Syllabus**

**Microprocessors versus Microcontrollers, ARM Embedded Systems**: The RISC design philosophy, The ARM Design Philosophy, Embedded System Hardware, Embedded System Software.

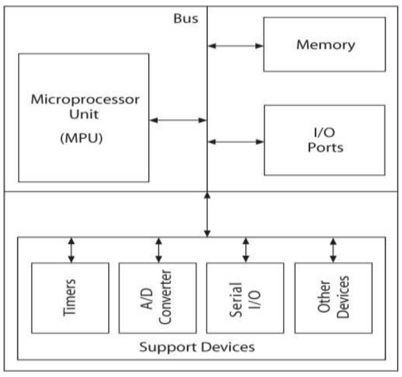
**ARM Processor Fundamentals**: Registers, Current Program Status Register, Pipeline, Exceptions, Interrupts, and the Vector Table , Core Extensions

Text book 1: Chapter 1 - 1.1 to 1.4, Chapter 2 - 2.1 to 2.5

## 1.1Microprocessors versus Microcontrollers



**Definition of Microprocessor:** An integrated circuit that contains all the functions of a central processing unit of a computer.



**Definition of Microcontroller:** A microcontroller is a computer present in a single integrated circuit which is dedicated to perform one task and execute one specific application.

It contains memory, programmable input/output peripherals as well a processor. Microcontrollers are mostly designed for embedded applications and are heavily used in automatically controlled electronic devices such as cell phones, cameras, microwave ovens, washing machines, etc.

The differences between the Microprocessor and Microcontroller with respect to the parameters are dicussed in the table 1.1.

Table 1.1: Difference between microprocessor and microcontroller.

| | **Microprocessor** | **Microcontroller** |
|---|---|---|
| **Application** | It used where intensive processing is required. It is used in personal computers, laptops, mobiles, video games, etc. | It used where the task is fixed and predefined. It is used in the washing machine, alarm, etc. |
| **Structure** | It has only the CPU in the chip. Other devices like I/O port, memory, timer is connected externally. The structure of the microprocessor is flexible. Users can decide the amount of memory, the number of I/O port and other peripheral devices. | CPU, Memory, I/O port and all other devices are connected on the single chip. The structure is fixed. Once it is designed the user cannot change the peripheral devices. |
| **Clock speed** | The clock speed of the microprocessor is high. It is in terms of the GHz. It ranges between 1 GHz to 4 GHz. | The clock speed of the microcontroller is less. It is in terms of the MHz. it ranges between 1 MHz to 300 MHz. |

| | | |
|---|---|---|
| **RAM** | The volatile memory (RAM) for the microprocessor is in the range of the 512 MB to 32 GB. | The volatile memory (RAM) for the microcontroller is in the range of 2 KB to 256 KB. |
| **ROM** | The hard disk (ROM) for the microprocessor is in the range of the 128 GB to 2 TB. | The hard drive or flash memory (ROM) is in the range of the 32 KB to 2 MB. |
| **Peripheral interface** | The common peripheral interface for the microprocessor is USB, UART, and high-speed Ethernet. | The common peripheral interface for the microcontroller is I2C, SPI, and |

## 1.2 ARM Embedded Systems

- The ARM processor core is a key component of many successful 32-bit embedded systems.
- ARM cores are widely used in mobile phones, handheld organizers, and a multitude of other everyday portable consumer devices.

**ARM history**

- 1983 developed by Acorn computers
  - To replace 6502 in BBC computers
  - 4-man VLSI design team
  - Its simplicity comes from the inexperience team
  - Match the needs for generalized SoC for reasonable power, performance and die size
  - The first commercial RISC implemenation
- 1990 ARM (Advanced RISC Machine), owned by Acorn, Apple and VLSI

**Why ARM?**

- One of the most licensed and thus widespread processor cores in the world
  - Used in PDA, cell phones, multimedia players, handheld game console, digital TV and cameras
  - ARM7: GBA, iPod
  - ARM9: NDS, PSP, Sony Ericsson, BenQ
  - ARM11: Apple iPhone, Nokia N93, N800
  - 75% of 32-bit embedded processors
- Used especially in portable devices due to its low power consumption and reasonable performance

**ARM processors**

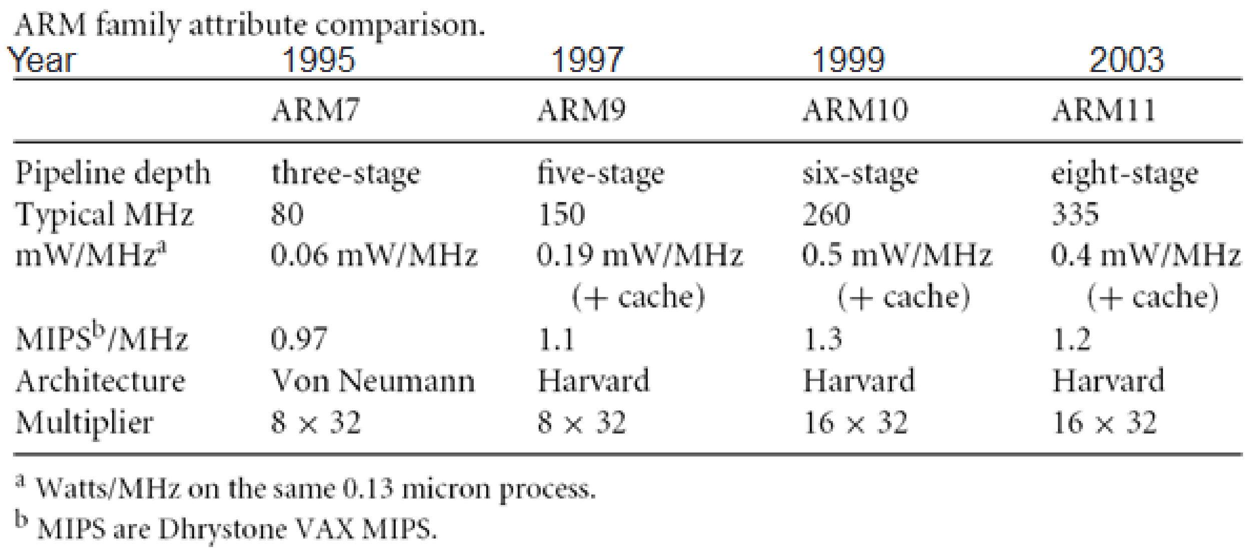
ARM processors are typically 32 bit and above. So, you use ARM processor when you need reasonably powerful computation capability that will make the heart of your embedded system. Now another thing is that ARM processors have very low power consumption and of course, reasonably good performance. Because of this low power consumption, they are very widely used in battery operated devices. There are many battery operated devices like the mobile phones.

**Nomenclature of ARM**

- ARMxyzTDMIEJFS
    - x: series
    - y: MMU
    - z: cache
    - T: Thumb
    - D: debugger
    - M: Multiplier
    - I: EmbeddedICE (built-in debugger hardware)
    - E: Enhanced instruction
    - J: Jazelle (JVM)
    - F: Floating-point
    - S: Synthesizible version (source code version for EDA tools)

**Popular ARM architectures**

- ARM7TDMI
    - 3 pipeline stages (fetch/decode/execute)
    - High code density/low power consumption
    - One of the most used ARM-version (for low-end systems)
    - All ARM cores after ARM7TDMI include TDMI even if they do not include TDMI in their labels
- ARM9TDMI
    - Compatible with ARM7
    - 5 stages (fetch/decode/execute/memory/write)
    - Separate instruction and data cache
- ARM11


ARM family attribute comparison.

| Year | 1995 | 1997 | 1999 | 2003 |
|---|---|---|---|---|
| | 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.

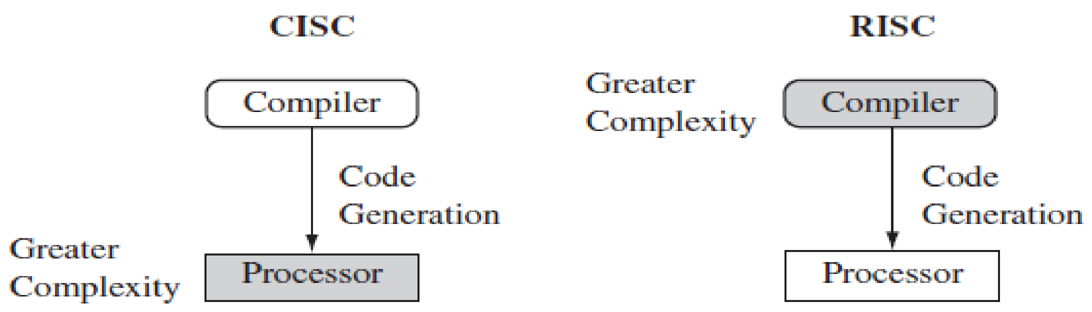
### 1.2.1 The RISC design philosophy

First thing is that, ARM is essentially a RISC (Reduced Instruction Set Computer) based architecture.

- RISC: simple but powerful instructions that execute within a single cycle at high clock speed.
- Results in simple design and fast clock rate
- The distinction blurs because CISC implements RISC concepts

The important differences between CISC and RISC are shown in the table 1.3.

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



- Four major design rules:
  - Instructions: reduced set/single cycle/fixed length
  - Pipeline: decode in one stage/no need for microcode
  - Registers: a large set of general-purpose registers
  - Load/store architecture: data processing instructions apply to registers only; load/store to transfer data from memory.

Each of them is discussed in detail as follows:

- **Instruction**
  - RISC processor has a reduced number of instruction classes.
  - Simple operations that can each execute in a single cycle.
  - The compiler or programmer synthesized complicated operations (e.g. a divide operation) by combine several simple instructions.
  - In CISC processors the instructions are often of variable size and take many cycles to execute.

- **Pipeline**
  - The processing of instructions is broken down into smaller units (stage) that can be executed in parallel by pipelines.
  - There is no need for an instruction to be executed by a mini-program (microcode) as on CISC processor.
- **Register**
  - RISC have a large General Purpose Registers (GPR) set.
  - Registers act as the fast local memory store for all data processing operations.
  - In contrast, CISC processors have dedicated registers for specific purposes.
- **Load/store architecture**
  - Separating memory access from data processing.
  - 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.

### 1.2.2 The ARM Design Philosophy

- Small processor for lower power consumption (for embedded system)
- High code density for limited memory and physical size restrictions
- The ability to use slow and low-cost memory
- Reduced die size for reducing manufacture cost and accommodating more peripherals
- There are a number of physical features that have driven the ARM processor design:
  - Low Power Consumption: Smallest Core;
  - Limited Memory: High code density;
  - Die density: Simple Hardware Executive Unit
- The ARM core is not a pure RISC architecture because of the constraints of its primary application – the embedded system.
- Simplicity favors regularity ?
  - These design rules allow a RISC processor to be simpler, and thus the core can operate at higher clock frequencies.
- Instruction Set for Embedded System
- *The ARM instruction set differs from the pure RISC definition in several ways make the ARM suitable for embedded application*
  - **Variable cycle execution for certain instruction**

- 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.

- **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.

### 1.2.3 Embedded System Hardware

- Embedded systems can control many different devices, from small sensors found on a production line, to the real-time control systems.
- All these devices use a combination of software and hardware components.
- The device separated into four main hardware components which is shown in the figure 2.1.
- 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.

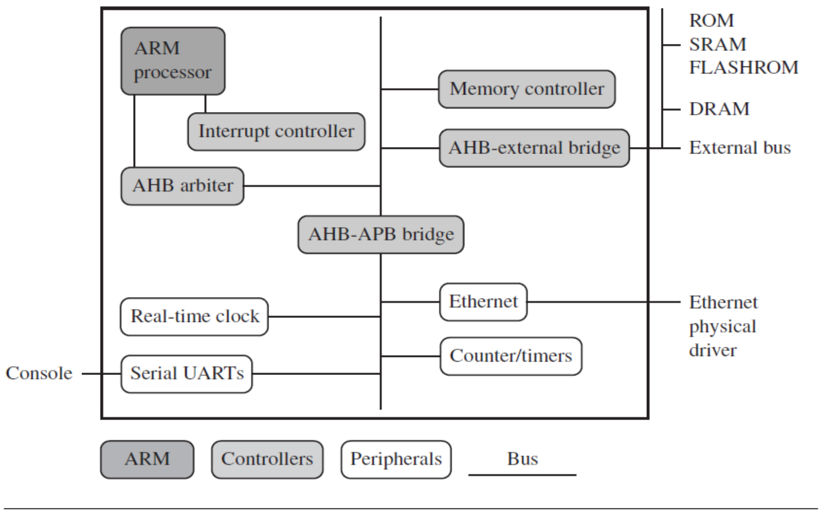


Figure 1.2 An example of an ARM-based embedded device, a microcontroller.

- ***Controllers*** coordinate important functional blocks of the system.
  - Two commonly found controllers are interrupt and memory controllers.
- The ***peripherals*** provide all the input-output capability external to the chip and are responsible for the uniqueness of the embedded device.
- A ***bus*** is used to communicate between different parts of the device.

### 1.2.3.1 ARM Bus Technology

- Embedded systems use different bus technologies than those designed for x86 PCs.
  - Like the Peripheral Component Interconnect (PCI) bus, this type of technology is *external or off-chip* (i.e., the bus is designed to connect mechanically and electrically to devices external to the chip) and is built into the motherboard of a PC.
- 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.
  - ***Bus master:*** The ARM processor core
    - A logical device capable of initiating a data transfer with another device across the same bus.
  - ***Bus slaves:*** Peripherals
    - Logical devices capable only of responding to a transfer request from a bus master device.
- A bus has two architecture levels.

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

### 1.2.3.2 AMBA Bus Protocol

- The **Advanced Microcontroller Bus Architecture (AMBA)** was introduced in 1996 and has been widely adopted as the on-chip bus architecture used for ARM processors.
    - The first AMBA buses introduced were the *ARM System Bus (ASB)* and the *ARM Peripheral Bus (APB).*
    - Later ARM introduced another bus design, called the *ARM High Performance Bus* (AHB).
- Using AMBA, peripheral designers
    - Can reuse the same design on multiple projects
    - On-chip bus without having to redesign an interface for each different processor architecture.
    - This plug-and-play interface for hardware developers improves availability and time to market.
- AHB provides higher data throughput than ASB because it is based on a centralized multiplexed bus scheme rather than the ASB bidirectional bus design.
- AHB bus to run at higher clock speeds support *bus widths of 64 and 128 bits.*
- ARM has introduced two variations on the AHB bus:
    - *Multi-layer AHB and AHB-Lite.*
- In contrast to the original AHB, which allows a single bus master to be active on the bus at any time, the Multi-layer AHB bus allows multiple active bus masters.
- AHB-Lite is a subset of the AHB bus and it is limited to a single bus master.
- Multi-layer AHB systems works with multiple processors, permit operations to occur in parallel and allow for higher throughput rates.

### 1.2.3.3 Memory- Hierarchy

- A device that supports external off-chip memory and optional internal a cache to improve memory performance.
- The fastest memory cache is physically located nearer the ARM processor core and the slowest secondary memory is set further away.

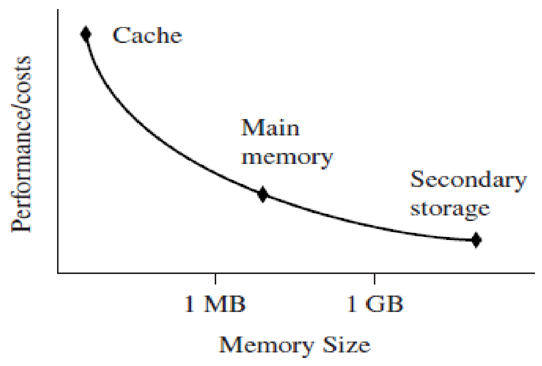


Figure 1.3 Storage trade-offs

- A cache provides
  - An overall increase in performance but with a loss of predictable execution time.
  - Although the cache increases the general performance of the system, it does not help real-time system response.
- The main memory is large—around 256 KB to 256 MB (or even greater), depending on the application—and is generally stored in separate chips.
- Load and store instructions access the main memory unless the values have been stored in the cache for fast access.
- Secondary storage is the largest and slowest form of memory.

**Memory- Width**

- The memory width is the number of bits the memory returns on each access—typically 8, 16, 32, or 64 bits.
- The memory width has a direct effect on the overall performance and cost ratio.
- For 16-bit processor need two memory cycle to access 32-bit information which decreases the performance.
- The better performance can be achieved using 16-bit Thumb instructions.

Fetching instructions from memory.

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

**Memory- Types**

- ROM- to hold boot code
- Flash ROM can be written to as well as read
- Dynamic random access memory(DRAM)is the most commonly used RAM for devices.

- Static random access memory (SRAM) is faster than the more traditional DRAM, but requires more silicon area. SRAM is *static*—the RAM does not require refreshing.
- Synchronous dynamic random access memory (SDRAM) is one of many subcategories of DRAM. It can run at much higher clock speeds than conventional memory.

**1.2.3.4 Peripherals**

- Peripherals range from a simple serial communication device to a more complex 802.11 wireless device.
- All ARM peripherals are *memory mapped*—the programming interface is a set of memory-addressed registers.
  - The address of these registers is an offset from a specific peripheral base address.
- Two important types of controllers
  - Memory controllers
  - Interrupt controllers.

**Peripherals-Memory Controllers**

- Memory controllers connect different types of memory to the processor bus.
- On power-up a memory controller is configured in hardware to allow certain memory devices to be active.
  - These memory devices allow the initialization code to be executed.
- Some memory devices must be set up by software; for example, when using DRAM, you first have to set up the memory timings and refresh rate before it can be accessed.

**Peripherals-Interrupt Controllers**

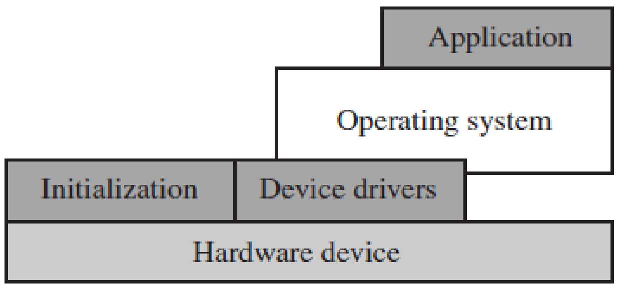
- When a peripheral or device requires attention, it raises an interrupt to the processor.
- An interrupt controller provides a programmable governing policy that allows software to determine which peripheral or device can interrupt the processor.
- Two types of interrupt controller for the ARM processor:
  - The standard interrupt controller
  - The vector interrupt controller (VIC).
- In standard interrupt controller
  - Receives 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 caused the interrupt.

### 1.2.4 Embedded System Software

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

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

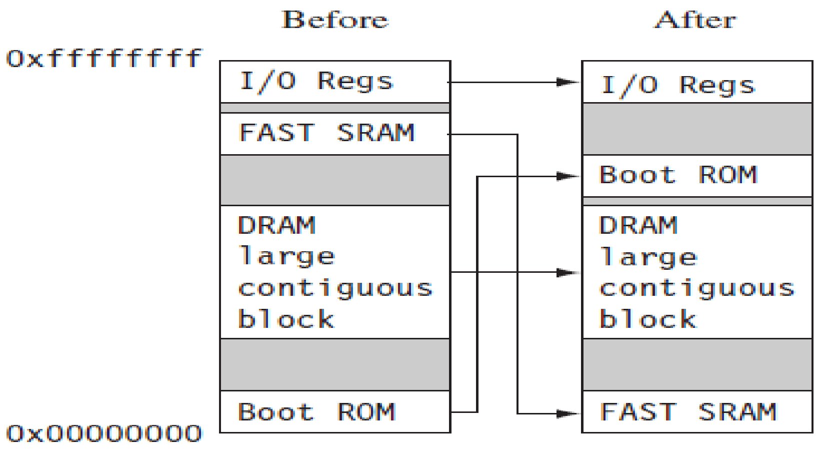


Software abstraction layers executing on hardware.

Each of them are discussed in detail as follows:

**Initialization (Boot) Code**

- Initial hardware configuration involves setting up the target platform so it can boot an image.
- For example, the memory system normally requires reorganization of the memory map, as shown in below.



Memory remapping.

- Diagnostics are often embedded in the initialization code.
  - Diagnostic code tests the system by exercising the hardware target to check if the target is in working order.
  - The primary purpose of diagnostic code is fault identification and isolation.
- Booting involves loading an image and handing control over to that image.
  - Booting different operating system
- Booting an image is the final phase,

- first you must load the image.
- Loading an image is copying an entire program including code and data into RAM, to just copying a data area containing volatile variables into RAM.

- In ARM-based embedded systems to provide for memory remapping because it allows the system to start the initialization code from ROM at power-up.

**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
    - processing time
- ARM processors support over 50 operating systems.
- It can divide operating systems into two main categories:
    - real-time operating systems (RTOSs)
    - platform operating systems.
- RTOSs provide guaranteed response times to events.
    - Different operating systems have different amounts of control over the system response time.
    - A hard real-time application
        - Requires a guaranteed response to work at all.
    - A soft real-time application
        - requires a good response time, but the performance degrades more gracefully if the response time overruns.
    - RTOS generally do not have secondary storage.
- Platform operating systems require a memory management unit to manage large, non real-time applications and tend to have secondary storage.
    - E.g. The Linux operating system.
- These two categories of operating system are not mutually exclusive:
    - There are operating systems that use an ARM core with a memory management unit and have real-time characteristics.

**Device driver**

- Provide a consistent software interface to the peripherals on the hardware device.

**Applications**

- The operating system schedules applications—code dedicated to handling a particular task.
- An application implements a processing task; the operating system controls the environment.
- An embedded system can have one active application or several applications running simultaneously.

- ARM processors are found in numerous market segments, including networking, automotive, mobile and consumer devices, mass storage, and imaging.
    - performs one of the tasks required for a device
- 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*.
- Initialization code (or boot code) takes the processor from the reset state to a state where the operating system can run.
    - Configures the memory controller and processor caches and initializes some devices.
    - Handles a number of administrative tasks prior to handing control over to an operating system image.
- Group different tasks into three phases:
    - Initial hardware configuration,
    - Diagnostics,
    - Booting

## 1.3 ARM Processor Fundamentals

### ARM core Dataflow Model

- A programmer can think of an ARM core as functional units connected by data buses as shown in the below figure.

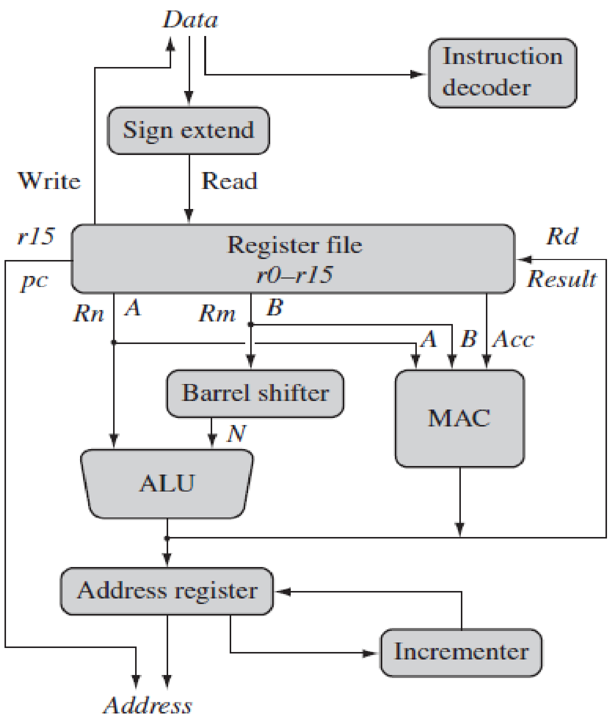


**Figure: ARM core dataflow model**

- As shown in the figure the arrows represents the flow of data, the lines represent the buses, and the boxes represents either an operation area or storage unit.

- The data enters the arm core through the **data bus**. The **instruction decoder** translates the instructions before they are executed.
- The ARM processor, like all RISC processors, uses a ***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,*
    - *Store instructions copy data from registers to memory*.

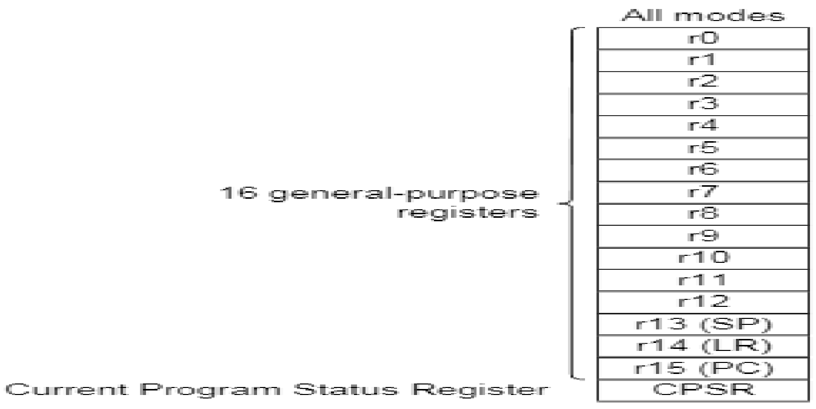
There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out solely in registers

- Data items are placed in the ***register file***—a storage bank made up of 32-bit registers.
- ARM core is a 32-bit processor, most registers as holding signed or unsigned 32-bit values.
- The **sign extend** hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.
- ARM instructions typically have
    - Two source registers, ***Rn* and *Rm***
    - A single result or destination register, ***Rd***.
- Source operands are read from the register file using the internal buses *A* and *B*, respectively.
- The **ALU (arithmetic logic unit)** or **MAC (multiply-accumulate unit)** takes the register values *Rn* and *Rm* from the *A* and *B* buses and computes a result.
- Data processing instructions write the result in *Rd* directly to the register file.
- Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the ***Address*** bus.
- Register *Rm* alternatively can be preprocessed in the **barrel shifter** before it enters the ALU.
- Together the **barrel shifter** and ALU can calculate a wide range of expressions and addresses.
- 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 reads or writes the next register value from or to the next sequential memory location.
- ***Key components of the processor:***
    - The registers,
    - The current program status register (*CPSR*)
    - The pipeline.

### 1.3.1 Registers

- General-purpose registers hold either data or an address.
- ***Identified with the letter r prefixed to the register number. E.g. r4.***
- Figure shows the active registers available in *user* mode
    - a protected mode normally used when executing applications.

- The processor can operate in seven different modes.
- **There are up to 18 active registers:**
    - 16 data registers
    - 2 processor status registers.
- The data registers are visible to the programmer as *r0* to *r15*.

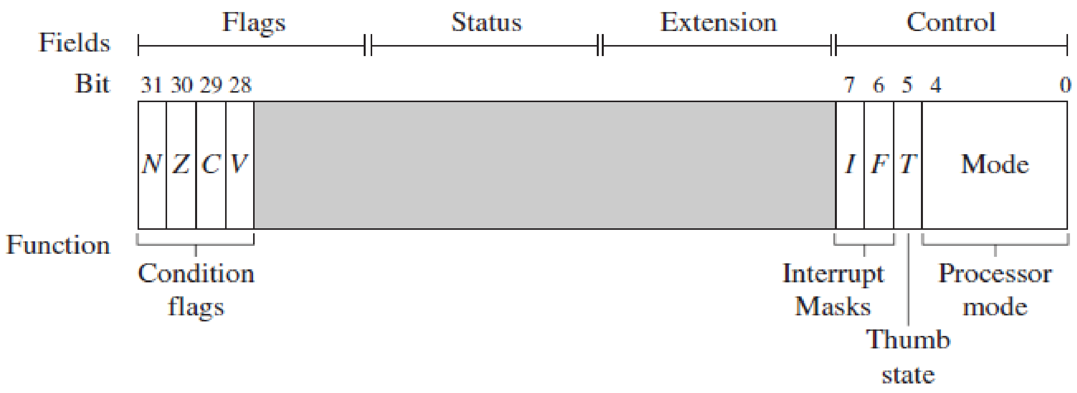


- **Register *r13*** is traditionally used as the **stack pointer (*SP*)** and stores the head of the stack in the current processor mode.
- **Register *r14*** is called the **link register (*LR*)** and is where the core puts the return address whenever it calls a subroutine.
- **Register *r15*** is the **program counter (*PC*)** and contains the address of the next instruction to be fetched by the processor.
- Registers *r13* and *r14* can also be used as general-purpose registers, depending on processor mode.
- It is dangerous to use *r13* as a general register when the processor is running any form of operating system because which often assume that *r13* always points to a valid stack frame.
- In ARM state the registers ***r0* to *r13* are *orthogonal***
    - any instruction that you can apply to *r0* you can equally well apply to any of the other registers
- There are two program status registers: ***cpsr* and *spsr*** (the current and saved program status registers, respectively).
- Visibility of registers to the programmer depends upon the current mode of the processor.

## 1.3.2 Current Program Status Register

- ***CPSR*** to monitor and control internal operations.
- The *cpsr* is a dedicated 32-bit register and resides in the register file.
- 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 shaded parts are reserved for future expansion.

- The **control field** contains the processor mode, state, and interrupt mask bits.
- The **flags** field contains the condition flags.



- Some ARM processor cores have extra bits allocated.
- For example: the ***J* bit**, which can be found in the flags field, is only available on Jazelle-enabled processors, which execute 8-bit instructions.

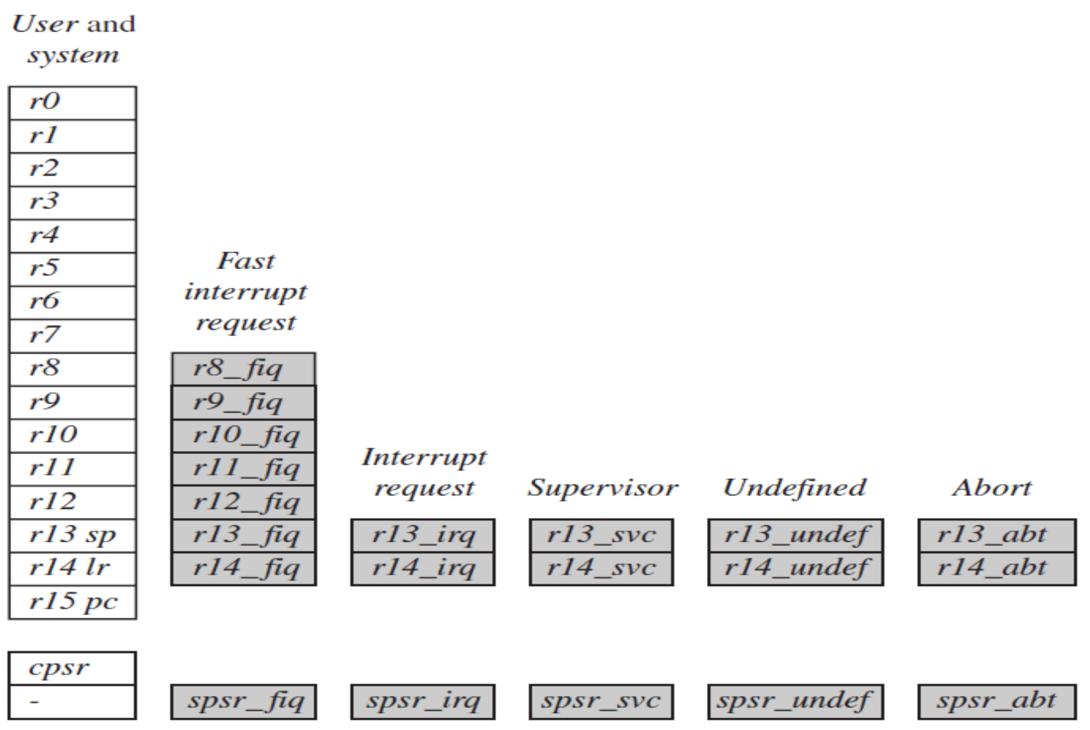
**1.3.2.1 CPSR-Processor Modes**

- 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:
  - In privileged mode allows full read-write access to the *cpsr*.
  - In 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,*
    - *Undefined*
  - **One nonprivileged mode**
    - *User*
- ***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.

### 1.3.2.2 CPSR-Banked Registers

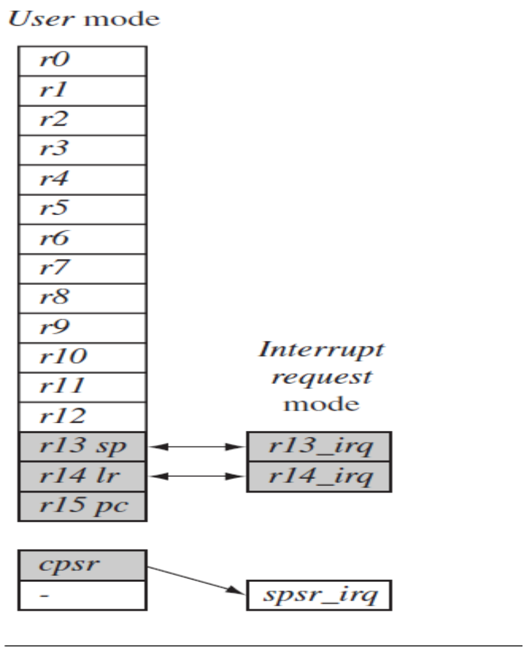
- Figure shows all 37 registers in the register file, where 20 registers are hidden from a program at different times. These registers are called ***banked registers*** (shading in the diagram).



Complete ARM register set.

- These are available only when the processor is in a particular mode
  - E.g., A*bort* mode has banked registers *r13_abt*, *r14_abt* and *spsr_abt*.
- 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 change in 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 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 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 shows what happens when an interrupt forces a mode change. This shows the change from *user* mode to *interrupt request* mode.



Changing mode on an exception.

- This change the *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 saved 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 a privileged mode and there is no *spsr* available in *user* mode.
- Another feature 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 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 |

### 1.3.2.3 CPSR-State and Instruction Sets

- The state of the core determines which instruction set is being executed.
- **There are three instruction sets:**
  - ARM: supports 32 bit instructions
  - Thumb: supports 16 bit instructions
  - Jazelle: supports 8 bit instructions
- The ARM instruction set is only active when the processor is in ARM state.
- You cannot intermingle sequential ARM, Thumb, and Jazelle instructions.
- The ***Jazelle J*** and ***Thumb T*** bits in the *cpsr* reflect the state of the processor.
- When both *J* and *T* bits are 0, the processor is in ARM state and executes ARM instructions. When the T bit is 1, then the processor is in Thumb state.

ARM and Thumb instruction set features.

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

- *Jazelle* executes 8-bit instructions and is a hybrid mix of software (Java virtual machine) and hardware designed to speed up the execution of Java bytecodes.
- Hardware portion of Jazelle only supports a subset of the Java bytecodes; the rest are emulated in software (Java virtual machine).

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. |

### 1.3.2.4 CPSR- 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)
    - *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*.

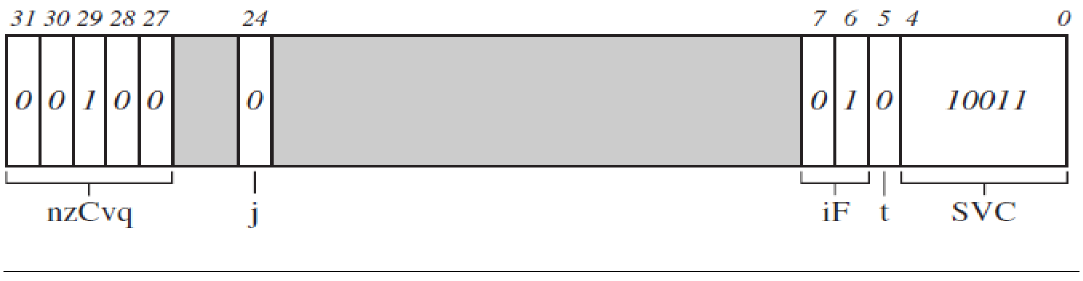
### 1.3.2.5 CPSR-Condition Flags

- 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 is zero, then the Z flag in the cpsr is set.*
- The following table shows the list of condition flags and a short description on what causes them to be set.

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 |

- These flags are located in the most significant bits in the *cpsr*.
- These bits are used for conditional execution.
- The following figure indicates the typical value of cpsr.



Example: *cpsr* = *nzCvqjiFt_SVC*.

- 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.

### 1.3.2.6 CPSR-Conditional Execution

- Conditional execution controls whether or not the core will execute an instruction. When a condition mnemonic is not present, the default behavior is to set it to always (***AL***) execute.
- Conditional execution controls whether or not the core will execute an instruction.
- When a condition mnemonic is not present, the default behavior is to set it to always (***AL***) execute.

Condition mnemonics.

| Mnemonic | Name | Condition flags |
|---|---|---|
| EQ | equal | *Z* |
| NE | not equal | *z* |
| CS HS | carry set/unsigned higher or same | *C* |
| CC LO | carry clear/unsigned lower | *c* |
| MI | minus/negative | *N* |
| PL | plus/positive or zero | *n* |
| VS | overflow | *V* |
| VC | no overflow | *v* |
| HI | unsigned higher | *zC* |
| LS | unsigned lower or same | *Z* or *c* |
| GE | signed greater than or equal | *NV* or *nv* |
| LT | signed less than | *Nv* or *nV* |
| GT | signed greater than | *NzV* or *nzv* |
| LE | signed less than or equal | *Z* or *Nv* or *nV* |
| AL | always (unconditional) | ignored |

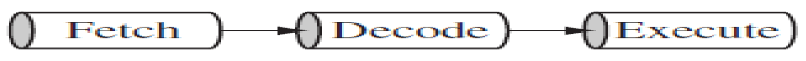
- Example:

  CMP R1,R2 ; compares the instructions and the sets the flags accordingly

  ADDEQ R0,R1,R2 ; checks for Z flag in cpsr if it is set to 1then performs addition

### 1.3.3 Pipeline

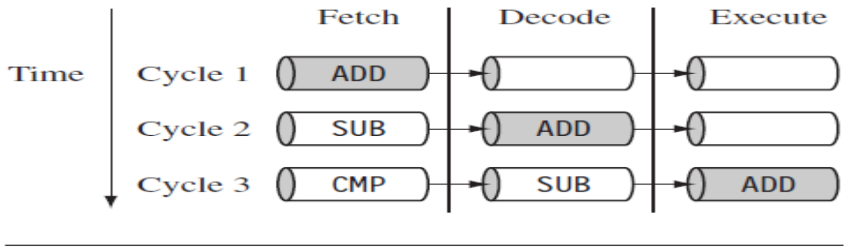
- A pipeline is the mechanism a RISC processor uses to execute instructions. Using a pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed.
- Figure shows a three-stage pipeline:



ARM7 Three-stage pipeline.

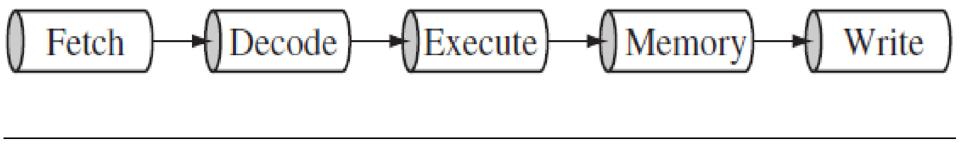
- *Fetch* loads an instruction from memory.
- *Decode* identifies the instruction to be executed.
- *Execute* processes the instruction and writes the result back to a register.

- The following figure illustrates the pipeline using a simple example.

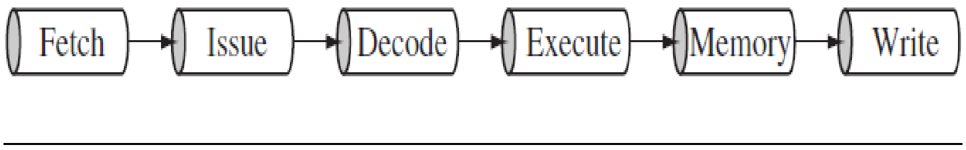


Pipelined instruction sequence.

- 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. 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.
- So write code to reduce this dependency by using *instruction scheduling*.



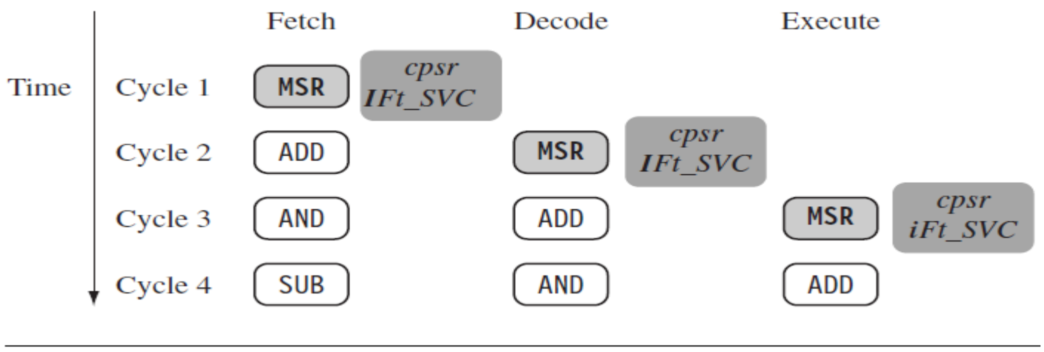
ARM9 five-stage pipeline.



ARM10 six-stage pipeline.

**Pipeline Executing Characteristics**

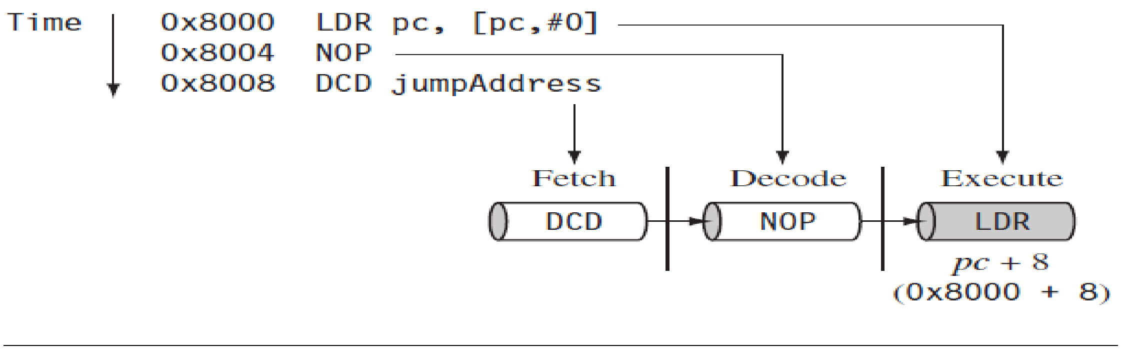
- The ARM pipeline has not processed an instruction until it passes completely through the execute stage.
- For example, an ARM7 pipeline (with three stages) has executed an instruction only when the fourth instruction is fetched.
- Figure shows an instruction sequence on an ARM7 pipeline.



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. t clears the *I* bit in the *cpsr* to enable the IRQ interrupts.
- Once the ADD instruction enters the execute stage of the pipeline, IRQ interrupts are enabled.

Figure illustrates the use of the pipeline and the program counter *pc*.



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

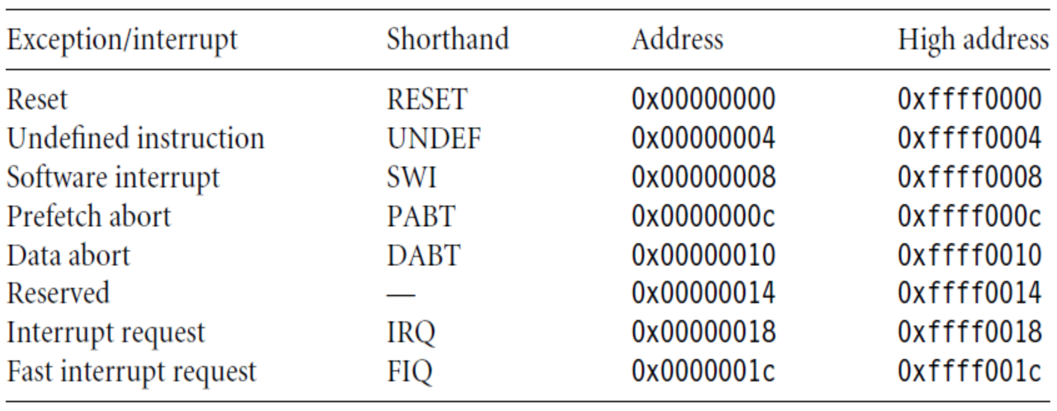
- . Architectural characteristics of all pipeline are: In the execute stage, the *pc* always points to the address of the instruction plus 8 bytes. This is important when the *pc* is used for calculating a relative offset. Note when the processor is in Thumb state the *pc* is the instruction address plus 4.
- **Characteristics of the pipeline**
    - The execution of a ***branch instruction or branching*** by the direct modification of the *pc* causes the ARM core to flush its pipeline.

- 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.
- 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.

### 1.3.4 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). Operating systems such as Linux and Microsoft's embedded products can take advantage of this feature.
- When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table.

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 |

- ***Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:***
    - ***Reset vector*** is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.

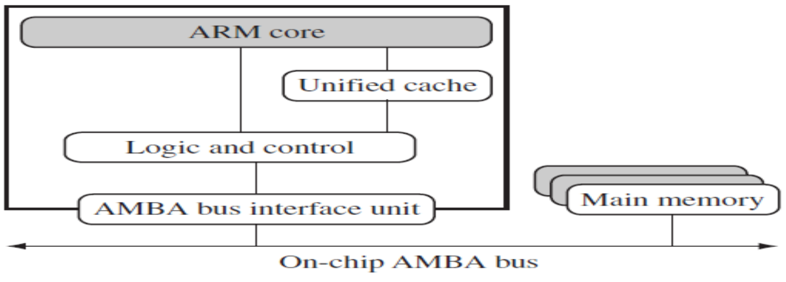
- ***Undefined instruction vector*** is used when the processor cannot decode an instruction.
- ***Software interrupt vector*** is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.
- ***Prefetch abort vector*** occurs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

### 1.3.5 Core Extensions

- The hardware extensions are standard components placed next to the ARM core.
- **There are three hardware extensions ARM wraps around the core:**
  - cache and tightly coupled memory,
  - memory management,
  - the coprocessor interface.

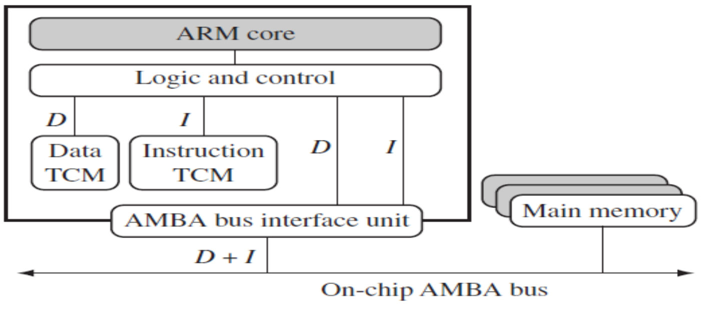
#### 1.3.5.1 Cache and Tightly Coupled Memory

- With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory.
- Most ARM-based embedded systems use a single-level cache internal to the processor.
- ***ARM has two forms of cache.***
  - **The Von Neumann–style cores**: it combines both data and instruction into a single unified cache. This logic is simply called as **glue logic.**



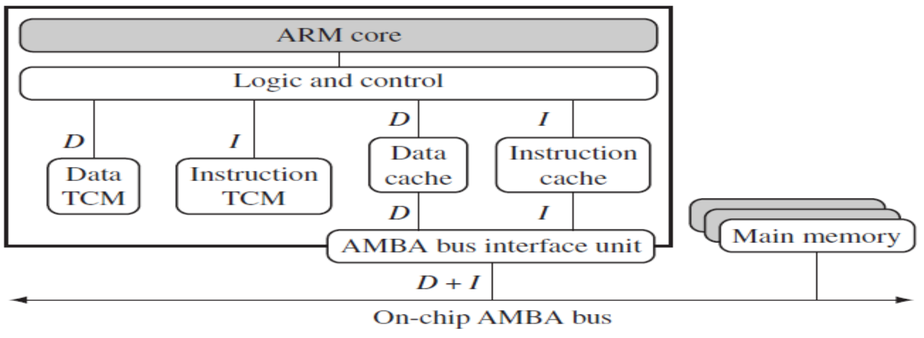
A simplified Von Neumann architecture with cache.

- **The Harvard-style cores:** has separate caches for data and instruction as shown in the figure below.



A simplified Harvard architecture with TCMs.

- A cache provides an overall increase in performance but at the expense of predictable execution.
- But for real-time systems the code execution is *deterministic*— the time taken for loading and storing instructions or data must be predictable. This is achieved using a form of memory **called *tightly coupled memory* (TCM).**
- TCM is fast SRAM located close to the core and guarantees the clock cycles required to fetch instructions or data—critical for real-time algorithms requiring deterministic behaviour.
- TCMs appear as memory in the address map and can be accessed as fast memory.
- By combining both technologies, ARM processors can have both improved performance and predictable real-time response.
- Figure shows an example core with a combination of caches and TCMs.



A simplified Harvard architecture with caches and TCMs.

### 1.3.5.2 Memory Management

- ARM core have three different types of memory management hardware. Each of them are discussed as follows.
- **No Extensions providing No Protection**

- *Nonprotected memory* is fixed and provides very little flexibility. It is normally used for small, simple embedded systems that require no protection from rogue applications.

- **A Memory Protection Unit (MPU) providing limited protection,**
  - *MPU*s employ a simple system that uses a limited number of memory regions. These regions are controlled with a set of special coprocessor registers, and each region is defined with specific access permissions. This type of memory management is used for systems that require memory protection but don't have a complex memory map.
- **A Memory Management Unit (MMU) providing full protection**
  - *MMU*s are the most comprehensive memory management hardware available on the ARM. The MMU uses a set of translation tables to provide fine-grained control over memory. These tables are stored in main memory and provide a virtual-to-physical address map as well as access permissions. MMUs are designed for more sophisticated platform operating systems that support multitasking.

### 1.3.5.3 Coprocessors

- A coprocessor extends the processing features of a core by extending the instruction set or by providing configuration registers.
- More than one coprocessor can be added to ARM core.
- The coprocessor accessed through a group of dedicated ARM instructions that provide a load-store type interface.
  - **Consider, for example, coprocessor 15:** The ARM processor uses coprocessor 15 registers to control the cache, TCMs, and memory management.
- Specialized instructions that can be added to the ARM instruction set to process vector floating-point (VFP) operations.
- If the decode stage sees a coprocessor instruction, then it offers it to the relevant coprocessor.
  - But if the coprocessor is not present or doesn't recognize the instruction, then the ARM takes an undefined instruction exception.

**********************************END of MODULE 1******************************