studocu

18CS44 MES Module 1 ppt

Data communication (PA College of Engineering and Technology)

Scan to open on Studocu

Studocu is not sponsored or endorsed by any college or university
Downloaded by Sharada Kori (smkori@git.edu)

MICROCONTROLLER AND EMBEDDED SYSTEMS (Effective from the academic year 2018 -2019) SEMESTER – IV			
Course Code	**18CS44**	**CIE Marks**	40
Number of Contact Hours/Week	3:0:0	**SEE Marks**	60
Total Number of Contact Hours	40	**Exam Hours**	03
CREDITS –3			

Course Learning Objectives: This course (18CS44) will enable students to:

- Understand the fundamentals of ARM based systems, basic hardware components, selection methods and attributes of an embedded system.
- Program ARM controller using the various instructions
- Identify the applicability of the embedded system
- Comprehend the real time operating system used for the embedded system

Module 1	**Contact Hours**
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** **RBT: L1, L2**	08
Module 2	
Introduction to the ARM Instruction Set : Data Processing Instructions , Programme Instructions, Software Interrupt Instructions, Program Status Register Instructions, Coprocessor Instructions, Loading Constants **ARM programming using Assembly language:** Writing Assembly code, Profiling and cycle counting, instruction scheduling, Register Allocation, Conditional Execution, Looping Constructs **Text book 1: Chapter 3:Sections 3.1 to 3.6 (Excluding 3.5.2), Chapter 6(Sections 6.1 to 6.6)** **RBT: L1, L2**	08
Module 3	
Embedded System Components: Embedded Vs General computing system, History of embedded systems, Classification of Embedded systems, Major applications areas of embedded systems, purpose of embedded systems Core of an Embedded System including all types of processor/controller, Memory, Sensors, Actuators, LED, 7 segment LED display, stepper motor, Keyboard, Push button switch, Communication Interface (onboard and external types), Embedded firmware, Other system components. **Text book 2:Chapter 1(Sections 1.2 to 1.6),Chapter 2(Sections 2.1 to 2.6)** **RBT: L1, L2**	08
Module 4	
Embedded System Design Concepts: Characteristics and Quality Attributes of Embedded Systems, Operational quality attributes ,non-operational quality attributes, Embedded	08

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

Systems-Application and Domain specific, Hardware Software Co-Design and Program Modelling, embedded firmware design and development **Text book 2: Chapter-3, Chapter-4, Chapter-7 (Sections 7.1, 7.2 only), Chapter-9 (Sections 9.1, 9.2, 9.3.1, 9.3.2 only)** **RBT: L1, L2**	
Module 5	
RTOS and IDE for Embedded System Design: Operating System basics, Types of operating systems, Task, process and threads (Only POSIX Threads with an example program), Thread preemption, Multiprocessing and Multitasking, Task Communication (without any program), Task synchronization issues – Racing and Deadlock, Concept of Binary and counting semaphores (Mutex example without any program), How to choose an RTOS, Integration and testing of Embedded hardware and firmware, Embedded system Development Environment – Block diagram (excluding Keil), Disassembler/decompiler, simulator, emulator and debugging techniques, target hardware debugging, boundary scan. **Text book 2: Chapter-10 (Sections 10.1, 10.2, 10.3, 10.4 , 10.7, 10.8.1.1, 10.8.1.2, 10.8.2.2, 10.10 only), Chapter 12, Chapter-13 (block diagram before 13.1, 13.3, 13.4, 13.5, 13.6 only)** **RBT: L1, L2**	08

Course Outcomes: The student will be able to :

- Describe the architectural features and instructions of ARM microcontroller
- Apply the knowledge gained for Programming ARM for different applications.
- Interface external devices and I/O with ARM microcontroller.
- Interpret the basic hardware components and their selection method based on the characteristics and attributes of an embedded system.
- Develop the hardware /software co-design and firmware design approaches.
- Demonstrate the need of real time operating system for embedded system applications

Question Paper Pattern:

- The question paper will have ten questions.
- Each full Question consisting of 20 marks
- There will be 2 full questions (with a maximum of four sub questions) from each module.
- Each full question will have sub questions covering all the topics under a module.
- The students will have to answer 5 full questions, selecting one full question from each module.

Textbooks:

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

Reference Books:

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

Downloaded by Sharada Kori (smkori@git.edu)

18CS44

MICROCONTROLLER AND EMBEDDED SYSTEMS

PREREQUISITES

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

LEVELS OF PROGRAMMING

There are three levels of programming –

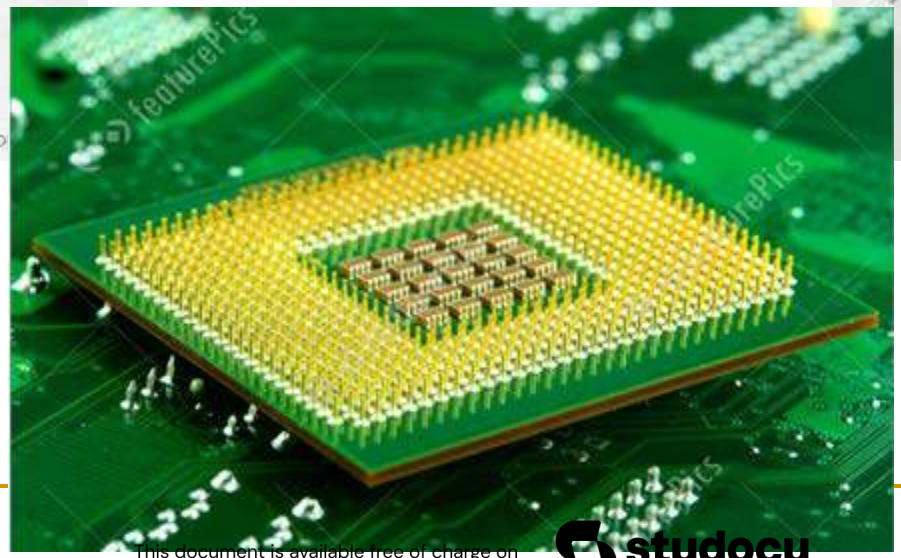
1. Machine language
2. Assembler language
3. High level language.

- A translator converts assembly or high-level language to machine language.
  - High-level language – Compiler.
  - Assembly language – Assembler.

Downloaded by Sharada Kori (smkori@git.edu)

A **microprocessor** is an electronic component that is used by a computer to do its work.

It is a central processing unit on a single integrated circuit chip containing millions of very small components including transistors, resistors, and diodes that work together.



This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

The Microprocessor

- In December 1970, Gilbert Hyatt filed a patent application entitled "*Single Chip Integrated Circuit Computer Architecture*", the first basic patent on the microprocessor. The microprocessor was invented in the year 1971 in the Intel Labs. The first processor was a 4 bit processor and was called 4004.

Downloaded by Sharada Kori (smkori@git.edu)

The Modern Microprocessor

- In 1978, Intel released the **8086** microprocessor - a 16-bit microprocessors
- Addresses 1M bytes (1M byte = 1024K bytes = 1024 * 1024 bytes = 1,048,576 bytes) of memory
- Executes 2.5 MIPs (millions of instructions per second)
- A small 6-byte instruction cache or queue that pre-fetched a few instructions before they were executed.
- Its instruction set contained over 20,000 instructions.

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

The Microprocessor (sometime called CPU)

- Heart of microprocessor based computer systems
- Controls the memory and I/O through buses
- Transfers address, data, and control information between an I/O device or memory and the microprocessor via buses
- Three buses exist for the transfer of following information –
    1. Address,
    2. Data, and
    3. Control.
- Memory and I/O are controlled through instructions
- Instructions are stored in the memory and executed by the microprocessor.

Downloaded by Sharada Kori (smkori@git.edu)

- The microprocessor performs *three main tasks* for the computer system –
  1. Data transfer between itself and the memory or I/O systems,
  2. Simple arithmetic and logic operations, and
  3. Program flow via simple decisions.
- The *power of microprocessor* is –
  - Its capability to execute hundreds of millions of instructions per second
  - Also, a microprocessor can make simple decisions based upon numerical facts.

This document is available free of charge on studocu

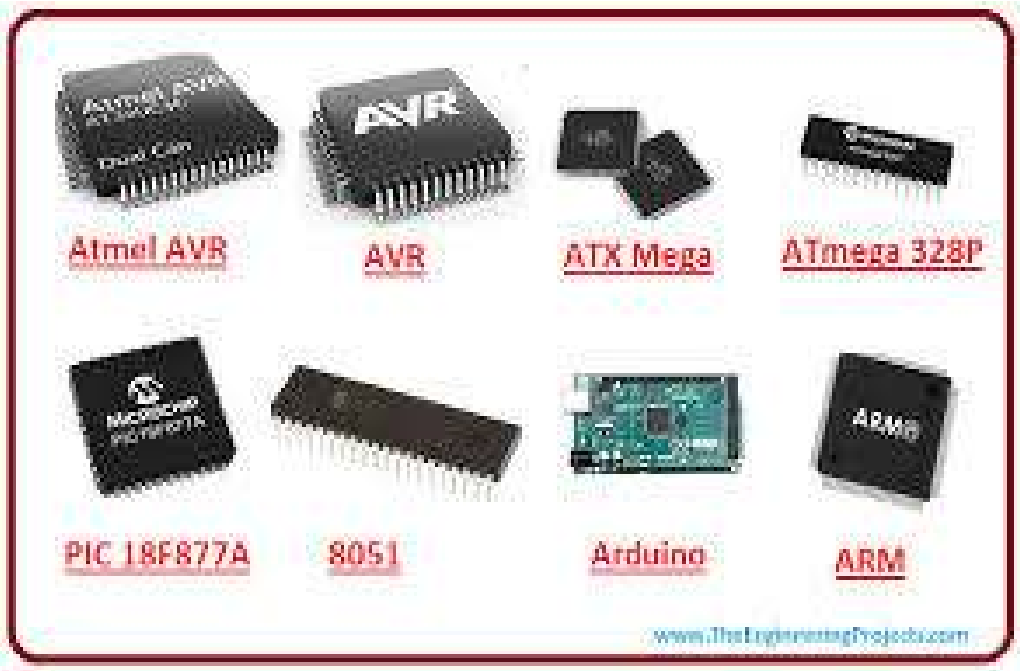
Downloaded by Sharada Kori (smkori@git.edu)

- The *applications of microprocessors* can be sub-divided into ***three*** categories.
  - The first and most important one is the computer applications.
  - The second one is the control application (micro-controllers, embedded controllers etc.).
  - The third is in communication (DSP processors, Cell phones etc.).

Downloaded by Sharada Kori (smkori@git.edu)

A **microcontroller** is a compact integrated circuit designed to govern a specific operation in an embedded system.

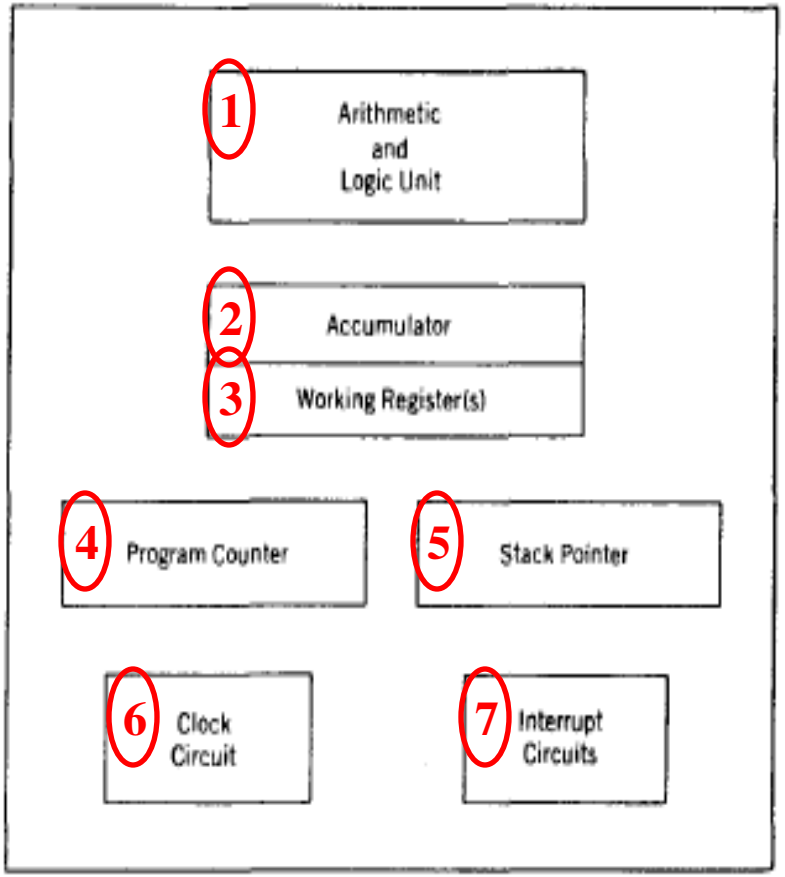
A typical microcontroller includes a processor, memory and input/output (I/O) peripherals on a single chip.



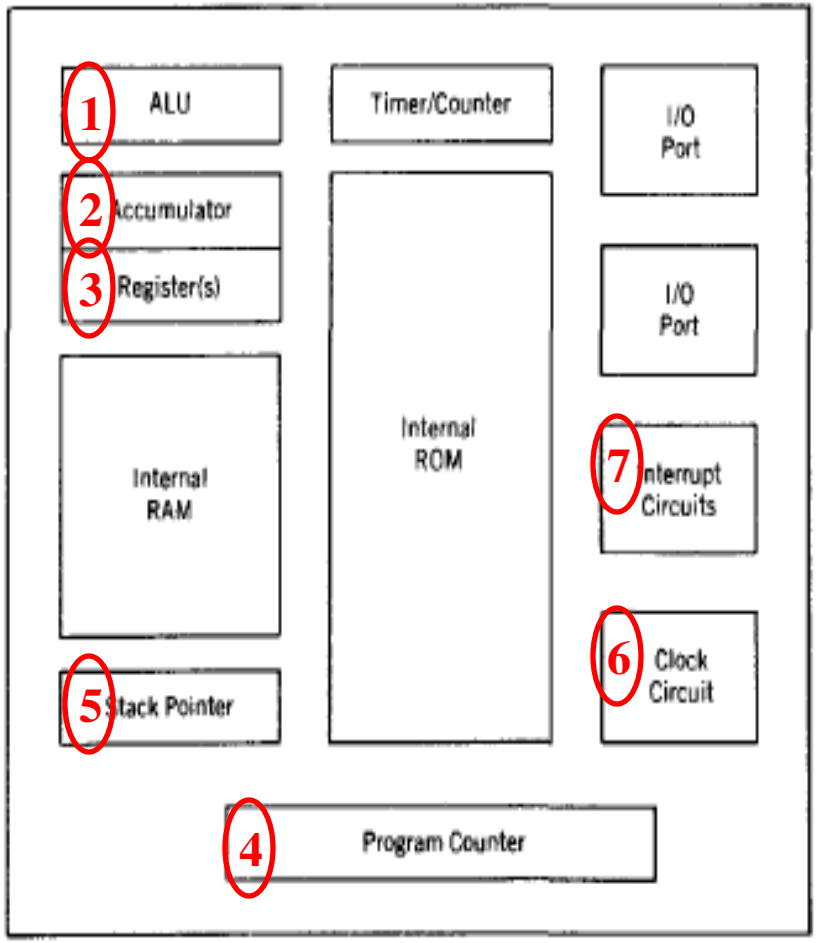
This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

A Block Diagram of a Microprocessor



A Block Diagram of a Microcontroller



Downloaded by Sharada Kori (smkori@git.edu)

18CS44

MICROCONTROLLER AND EMBEDDED SYSTEMS

MODULE 1

ARM EMBEDDED SYSTEMS

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

MICROPROCESSORS versus MICROCONTROLLERS

1) Microprocessors generally does not have RAM, ROM and I/O pins.
2) Microprocessors usually uses its pins as a bus to interface to RAM, ROM, and peripheral devices. Hence, the controlling bus is expandable at the board level.
3) Microprocessors are generally capable of being built into bigger general purpose applications.
4) Microprocessors, generally do not have power saving system.

1) Microcontroller is 'all in one' processor, with RAM, I/O ports, all on the chip.
2) Controlling bus is internal and not available to the board designer.
3) Microcontrollers are usually used for more dedicated applications.
4) Microcontrollers have power saving system, like idle mode or power saving; mode so overall it uses less power.

Cont…

Downloaded by Sharada Kori (smkori@git.edu)

5) The overall cost of systems made with Microprocessors are high, because of the high number of external components required.	5) Microcontrollers are made by using complementary metal oxide semiconductor technology; so they are far cheaper than Microprocessors.
6) Processing speed of general microprocessors is above 1 GHz ; so it works much faster than Microcontrollers.	6) Processing speed of Microcontrollers is about 8 MHz to 50 MHz.
7) Microprocessors are based on von-Neumann model; where, program and data are stored in same memory module.	7) Microcontrollers are based on Harvard architecture; where , program memory and data memory are separate.

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

- 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.
- The first ARM1 prototype was designed in 1985.
  - Success – a simple and powerful original design, which continues to improve today through constant technical innovation.
  - Example – ARM7TDMI:
    - provides up to 120 Dhrystone MIPS
    - high code density and low power consumption
    - ideal for mobile embedded devices

Dhrystone is a benchmark program written in *C* or *Pascal* (and now in *Java*) that tests a system's performance.

Downloaded by Sharada Kori (smkori@git.edu)

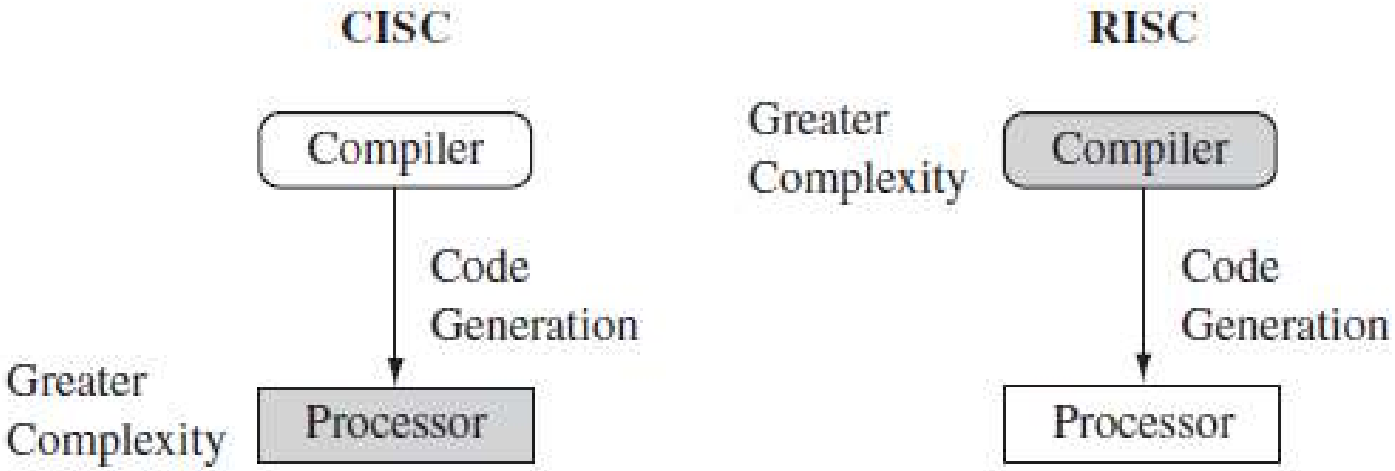
THE RISC DESIGN PHYLOSOPHY

- The ARM core uses *reduced instruction set computer (RISC)* architecture.
- RISC
  - a design philosophy
  - delivering simple but powerful instructions, that execute within a single cycle at a high clock speed
  - reducing the complexity of instructions performed by the hardware
    - it is easier to provide greater flexibility and intelligence in software rather than hardware
  - places greater demands on the compiler

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

- The *complex instruction set computer (CISC)*
  - relies more on the hardware for instruction functionality
  - instructions are more complicated



Downloaded by Sharada Kori (smkori@git.edu)

CISC	RISC
1. Complex instructions, taking multiple clock	1. Simple instructions, taking single clock
2. Emphasis on hardware, complexity is in the micro-program/processor	2. Emphasis on software, complexity is in the complier
3. Complex instructions, instructions executed by micro-program/processor	3. Reduced instructions, instructions executed by hardware
4. Variable format instructions, single register set and many instructions	4. Fixed format instructions, multiple register sets and few instructions
5. Many instructions and many addressing modes	5. Fixed instructions and few addressing modes
6. Conditional jump is usually based on status register bit	6. Conditional jump can be based on a bit anywhere in memory
7. Memory reference is embedded in many instructions	7. Memory reference is embedded in LOAD/STORE instructions

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

RISC Design Rules:

Instructions, Pipeline, Registers, Load-Store Architecture

1. Instructions –
   - » Reduced number of instruction classes
   - » Simple operations
   - » Each instruction can execute in a single clock cycle
   - » Compiler/programmer synthesizes complicated operations
   - » Each instruction is having fixed length – allows pipeline to fetch future instructions
     - o In CISC processors, the instructions are often of variable size and take many cycles to execute

Downloaded by Sharada Kori (smkori@git.edu)

2. Pipelines –

- Processing of instructions is broken down into smaller units
- Instructions are executed in parallel by pipelines
- Pipeline advances by one step on each cycle for maximum throughput
  - There is no need for an instruction to be executed by a mini-program called microcode as on CISC processors

3. Registers –

- 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
  - CISC processors have dedicated registers for specific purposes

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

4. Load-Store Architecture –

- » Processor operates on data held in registers
- » Separate load and store instructions transfer data between the register bank and external memory
- » 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 a CISC design the data processing operations can act on memory directly

- Design rules allow a RISC processor to be simpler, and the core can operate at higher clock frequencies
- In contrast, traditional CISC processors are more complex and operate at lower clock frequencies

Downloaded by Sharada Kori (smkori@git.edu)

THE ARM DESIGN PHYLOSOPHY

- Physical features that have driven the ARM processor design –
- Portable embedded systems require ***battery power***.
  - The ARM processor specially 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***
  - Embedded systems have limited memory
    - due to cost and/or physical size restrictions
      - —useful for applications that have limited on-board memory, such as mobile phones and mass storage devices

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

- Embedded systems are ***price sensitive***
  - *Use slow and low-cost memory devices*
    - to get substantial savings
      - —essential for high-volume applications like digital cameras
  - *Reduce the area of the die* taken up by the embedded processor
    - to reduced cost of the design and manufacturing for the end product
- ARM has incorporated ***hardware debug technology***
  - so that software engineers can view what is happening while the processor is executing code (visibility)
  - software engineers can resolve issues faster
- 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

Downloaded by Sharada Kori (smkori@git.edu)

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, Inline barrel shifter, Thumb 16-bit instruction set, Conditional execution, Enhanced instructions

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

- *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
  - Code density is also improved since multiple register transfers are common operations at the start and end of functions

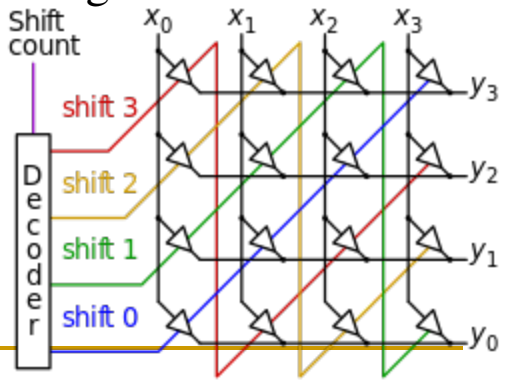
Downloaded by Sharada Kori (smkori@git.edu)

» *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

» Barrel Shifter is a digital circuit that can shift a data word by a specified number of bits without the use of any sequential logic, only pure combinatorial logic



This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

- *Thumb 16-bit instruction set*—ARM enhanced the processor core by adding a second 16-bit instruction set called Thumb
  - Thumb 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

Downloaded by Sharada Kori (smkori@git.edu)

- *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
  - These instructions allow a faster-performing ARM processor

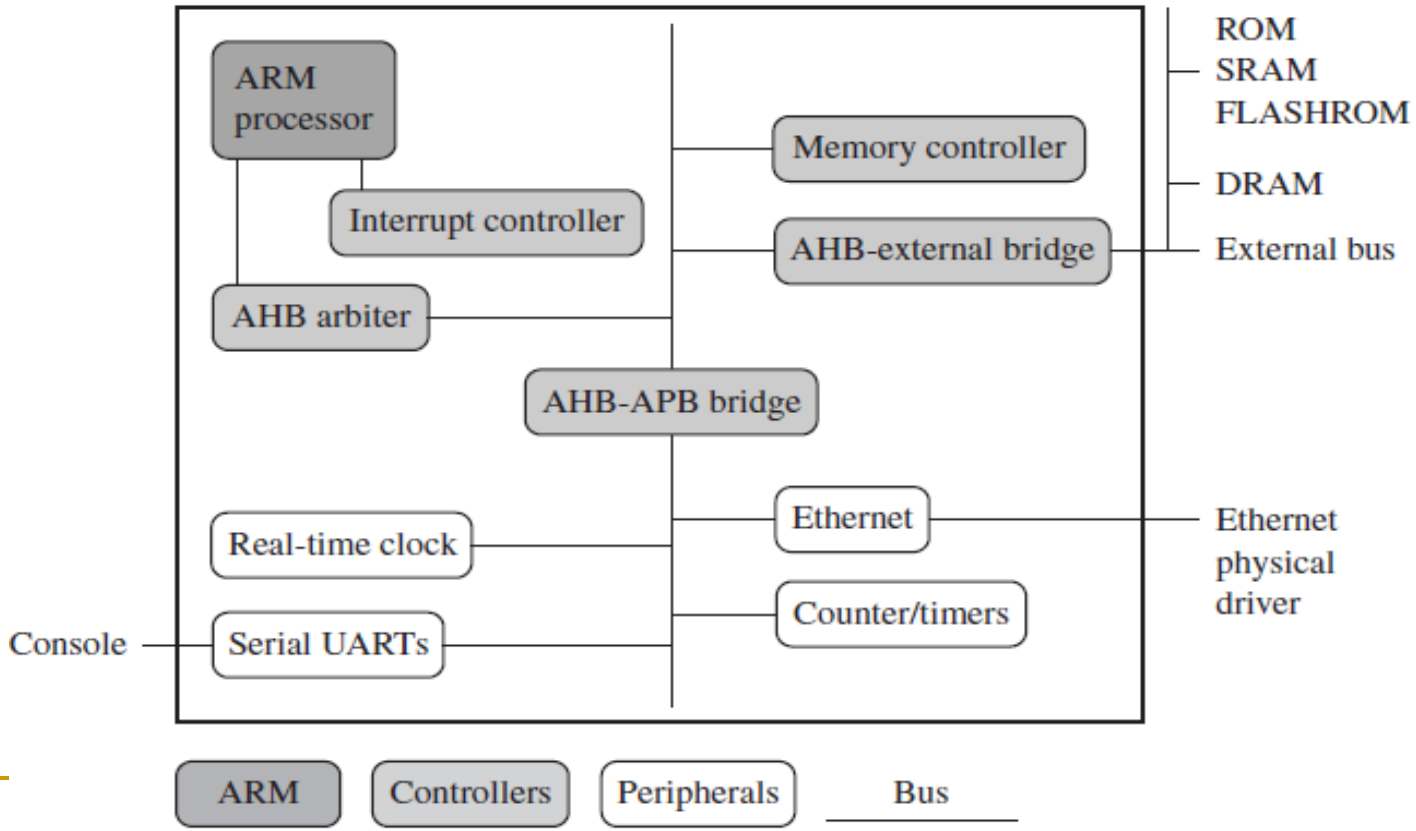
- These ***additional features*** have made the ARM processor one of the most commonly used 32-bit embedded processor cores

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

EMBEDDED SYSTEM HARDWARE

- Embedded systems can control many different devices – from small sensors to real-time control systems
  - All these devices use a combination of software and hardware components



Downloaded by Sharada Kori (smkori@git.edu)

Four Main Hardware Components –

ARM Processor, Controllers, Peripherals, Bus

1. The *ARM processor*
   - » Comprises a core
     - » the execution engine that processes instructions and manipulates data
   - » plus the surrounding components
     - » memory and cache
   - » Controls the embedded device
   - » Different versions of the ARM processor are available
     - » to suit the desired operating characteristics

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

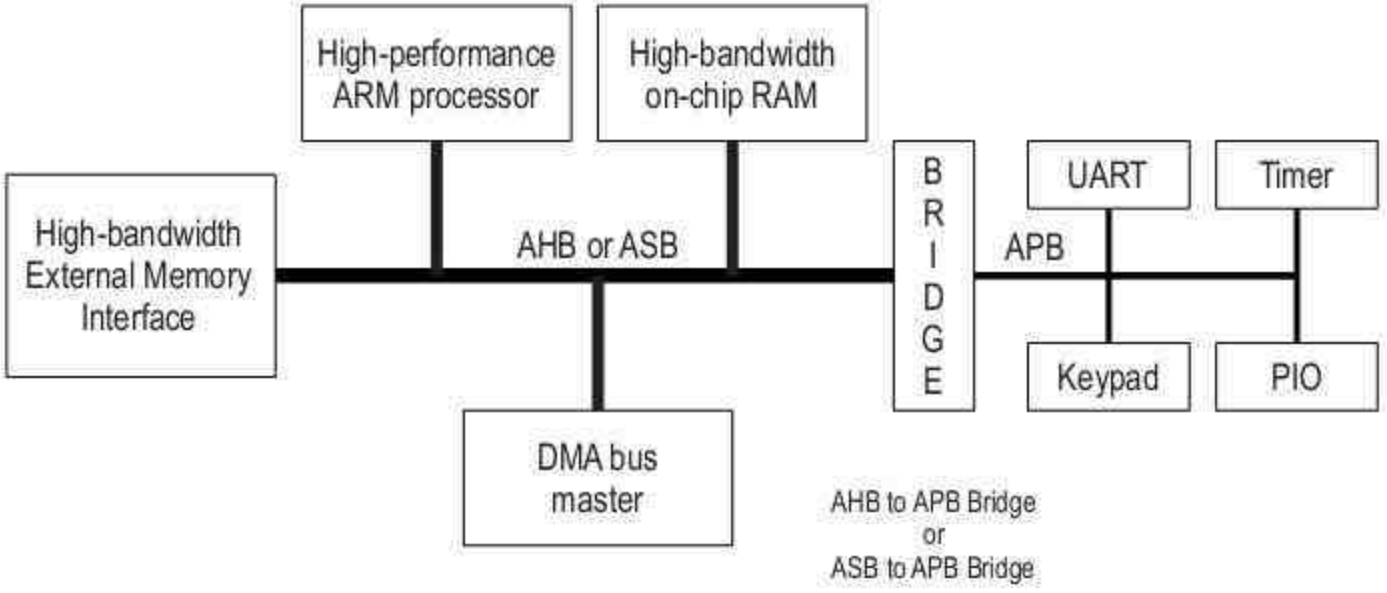
2. *Controllers*
   - » Coordinate important functional blocks of the system
   - » Two commonly found controllers are interrupt and memory controllers
3. The *peripherals*
   - » Provide all the input-output capability external to the chip
   - » Responsible for the uniqueness of the embedded device
4. A *bus* is used to
   - » Communicate between different parts of the device.

Downloaded by Sharada Kori (smkori@git.edu)

ARM Bus Technology

» Embedded devices use an on-chip bus that is internal to the chip

» allows different peripheral devices to be interconnected with an ARM core



AMBA AHB

* High performance
* Pipelined operation
* Multiple bus masters
* Burst transfers

AMBA ASB

* High performance
* Pipelined operation
* Multiple bus masters

AMBA APB

* Low power
* Latched address and control
* Simple interface
* Suitable for many peripherals

This document is available free of charge on studocu
Downloaded by Sharada Kori (smkori@git.edu)

- There are ***two different classes of devices*** attached to the bus:
  1. The *ARM processor core* is a bus master—a logical device capable of initiating a data transfer with another device across the same bus.
  2. *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:
  - A *physical level*—covers the electrical characteristics and bus width (16, 32, or 64 bits).
  - The *protocol*—the logical rules that govern the communication between the processor and a peripheral.

Downloaded by Sharada Kori (smkori@git.edu)

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 the *ARM High Performance Bus* (AHB)

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

- Using AMBA,
  - peripheral designers can reuse the same design on multiple projects.
    - A peripheral can simply be bolted onto the 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. This change allows the AHB bus to run at higher clock speeds.

Downloaded by Sharada Kori (smkori@git.edu)

- ARM has introduced ***two variations*** on the AHB bus: *Multi-layer AHB* and *AHB-Lite*
- 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
- The example device shown in the above Figure has three buses:
  - an *AHB bus* for the high- performance peripherals
  - an *APB bus* for the slower peripherals
  - a third *bus for external peripherals*

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

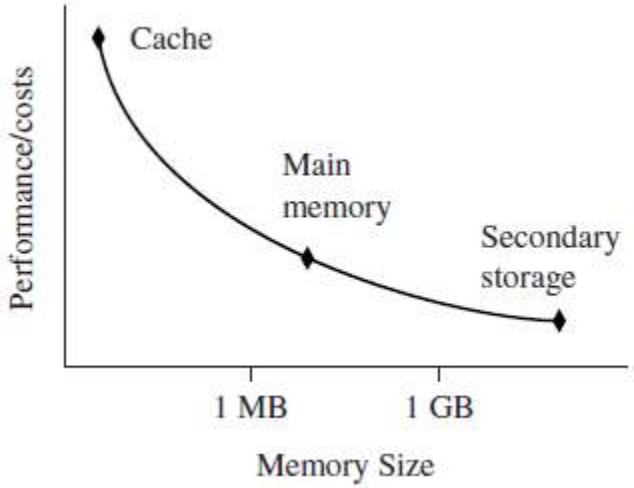
Memory

- An embedded system has to have some form of memory to store and execute code.
  - You have to compare price, performance, and power consumption
    - when deciding upon specific memory characteristics, such as hierarchy, width, and type

Downloaded by Sharada Kori (smkori@git.edu)

Memory Hierarchy

- All computer systems have memory arranged in some form of hierarchy
  - The following Figure shows the memory trade-offs: the fastest memory cache is physically located nearer the ARM processor core and the slowest secondary memory is set further away



  - Generally the closer memory is to the processor core, the more it costs and the smaller its capacity

This document is available free of charge on studocu
Downloaded by Sharada Kori (smkori@git.edu)

- The *cache* is
  - Placed between main memory and the core
  - Used to speed up data transfer between the processor and main memory
  - 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

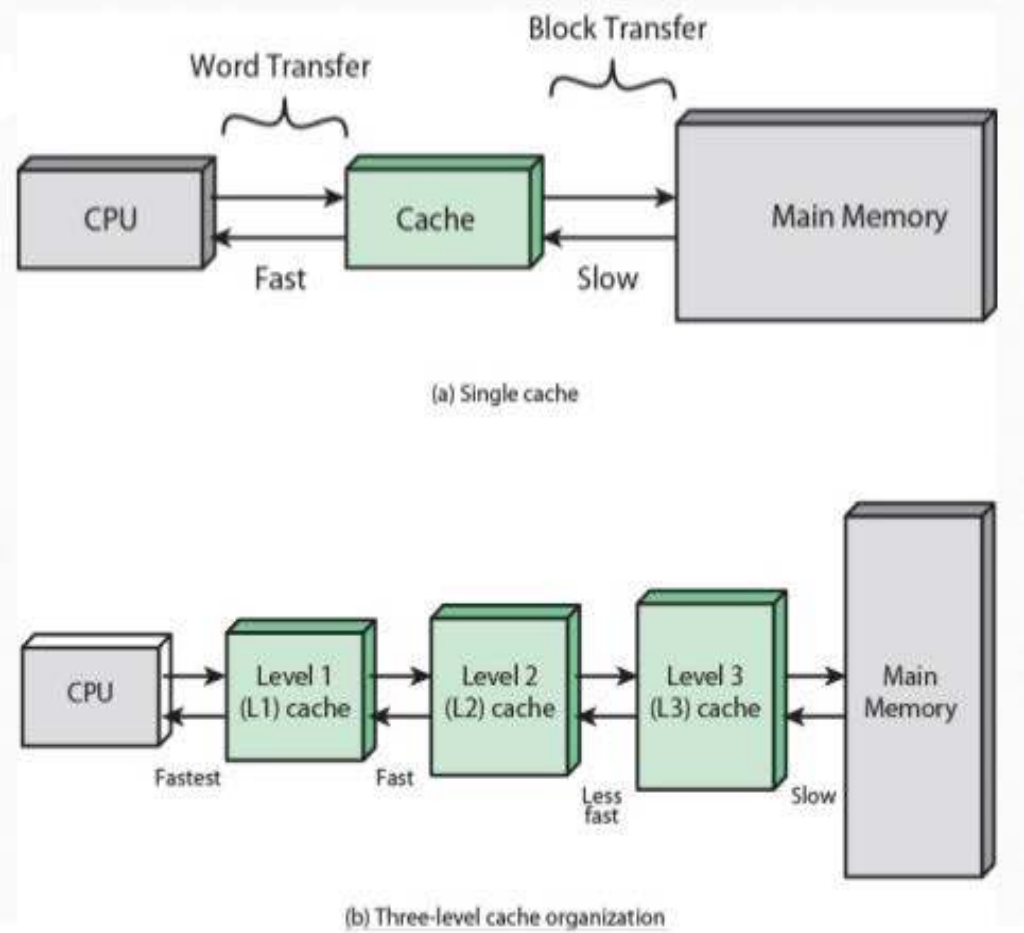
- The *main memory* is large—
  - Around 256 KB to 256 MB (or even greater), depending on the application—Generally stored in separate chips

Downloaded by Sharada Kori (smkori@git.edu)

» 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

» Examples – Hard disk drives and CD-ROM drives



This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

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
  - Lower bit memories are less expensive, but reduce the system performance
- The following Table summarizes theoretical cycle times on an ARM processor using different memory width devices

Instruction Size	8-bit Memory	16-bit Memory	32-bit Memory
ARM 32-bit	4 cycles	2 cycles	1 cycles
Thumb 16-bit	2 cycles	1 cycles	1 cycles

Downloaded by Sharada Kori (smkori@git.edu)

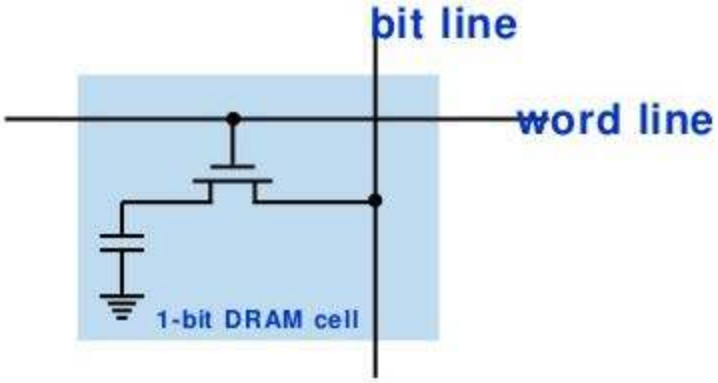
Memory Types – There are many *different types of memory*:

- *Read-only memory (ROM)* is the least flexible of all memory types because it contains an image that is permanently set at production time and cannot be reprogrammed
  - ROMs are used in high-volume devices that require no updates or corrections. Many devices also use a ROM to hold boot code
- *Flash ROM* can be written to as well as read, but it is slow to write so you shouldn't use it for holding dynamic data
- Its main use is for holding the device firmware or storing long-term data that needs to be preserved after power is off.
  - The erasing and writing of flash ROM are completely software controlled with no additional hardware circuitry required, which reduces the manufacturing costs

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

- *Dynamic random access memory (DRAM)* is the most commonly used RAM for devices
- It has the lowest cost per megabyte compared with other types of RAM
- DRAM is dynamic—it needs to have its storage cells refreshed and given a new electronic charge every few milliseconds, so you need to set up a DRAM controller before using the memory



Downloaded by Sharada Kori (smkori@git.edu)

- *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
- The access time for SRAM is considerably shorter than the equivalent DRAM because SRAM does not require a pause between data accesses
- But cost of SRAM is high
- *Synchronous dynamic random access memory (SDRAM)* is one of many subcategories of DRAM
- It can run at much higher clock speeds than conventional memory
- SDRAM synchronizes itself with the processor bus, because it is clocked
- Internally the data is fetched from memory cells, pipelined, and finally brought out on the bus in a burst

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

Dynamic RAM	Static RAM
1. Made up of Capacitors	1. Made up of Flip-flops
2. Data storage in the form of charge	2. Data storage in the form of voltage
3. Small in size and less expensive	3. Large in size and much expensive
4. High storage capacity	4. Low storage capacity
5. Slow, but consume less power	5. Fast, but consume more power
6. Data loses with time, so needs refreshing circuitry to maintain the charge in the capacitors for data	6. No data loses, does not require periodic refreshment to maintain data, data sustain with time
7. DRAM are used in Main memory	7. SRAM are used in Cache memory

Downloaded by Sharada Kori (smkori@git.edu)

Peripherals

» Embedded systems that interact with the outside world need some form of peripheral device.

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

»Each peripheral device usually performs a single function and may reside on-chip

»Peripherals range from a simple serial communication device to a more complex 802.11 wireless device

This document is available free of charge on studocu
Downloaded by Sharada Kori (smkori@git.edu)

- 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
- *Controllers* are specialized peripherals that implement higher levels of functionality within an embedded system
- Two important types of controllers are memory controllers and interrupt controllers

Downloaded by Sharada Kori (smkori@git.edu)

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

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

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 at any specific time by setting the appropriate bits in the interrupt controller registers
- There are *two types of interrupt controller* available for the ARM processor: the standard interrupt controller and the vector interrupt controller

Downloaded by Sharada Kori (smkori@git.edu)

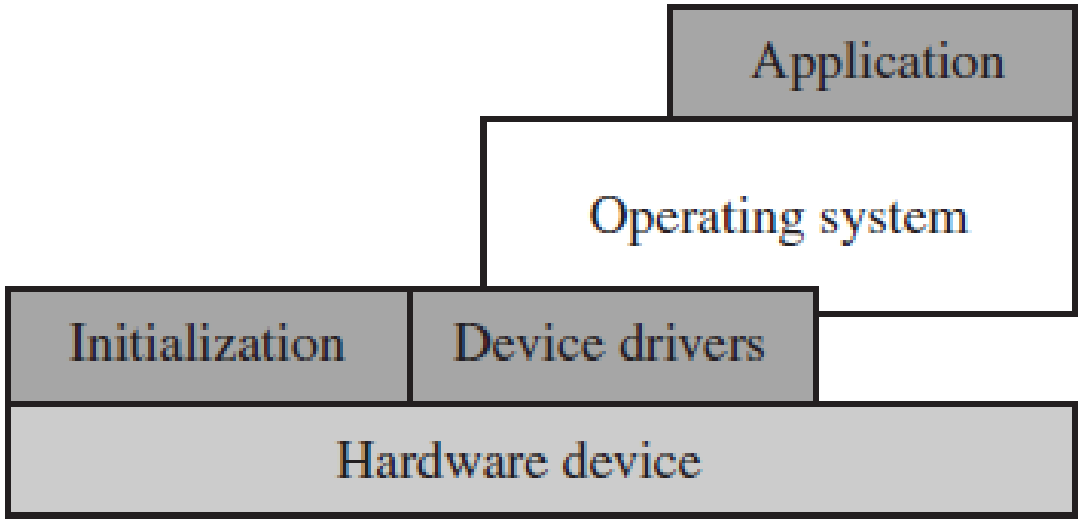
- The *standard interrupt controller* sends an interrupt signal to the processor core when an external device requests servicing
- It can be programmed to ignore or mask an individual device or set of devices.
  - The *interrupt handler* determines which device requires servicing by reading a device bitmap register in the interrupt controller
- The *vector interrupt controller (VIC)* is more powerful than the standard interrupt controller, because it prioritizes interrupts and simplifies the determination of which device caused the interrupt
  - Depending on the type, the VIC will either call the standard interrupt exception handler, which can load the address of the handler

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

EMBEDDED SYSTEM SOFTWARE

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



Downloaded by Sharada Kori (smkori@git.edu)

- The *initialization code*
  - first code executed on the board and is specific to a particular target or group of targets
  - 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
- The *device drivers*
  - provide a consistent software interface to the peripherals on the hardware device
- 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

This document is available free of charge on studocu
Downloaded by Sharada Kori (smkori@git.edu)

Initialization (Boot) Code

- Initialization code (or boot code) takes the processor from the reset state to a state where the operating system can run
- It usually configures the memory controller and processor caches and initializes some devices
- The initialization code handles a number of administrative tasks prior to handing control over to an operating system
  - We can group these different tasks into *three phases*: initial hardware configuration, diagnostics, and booting

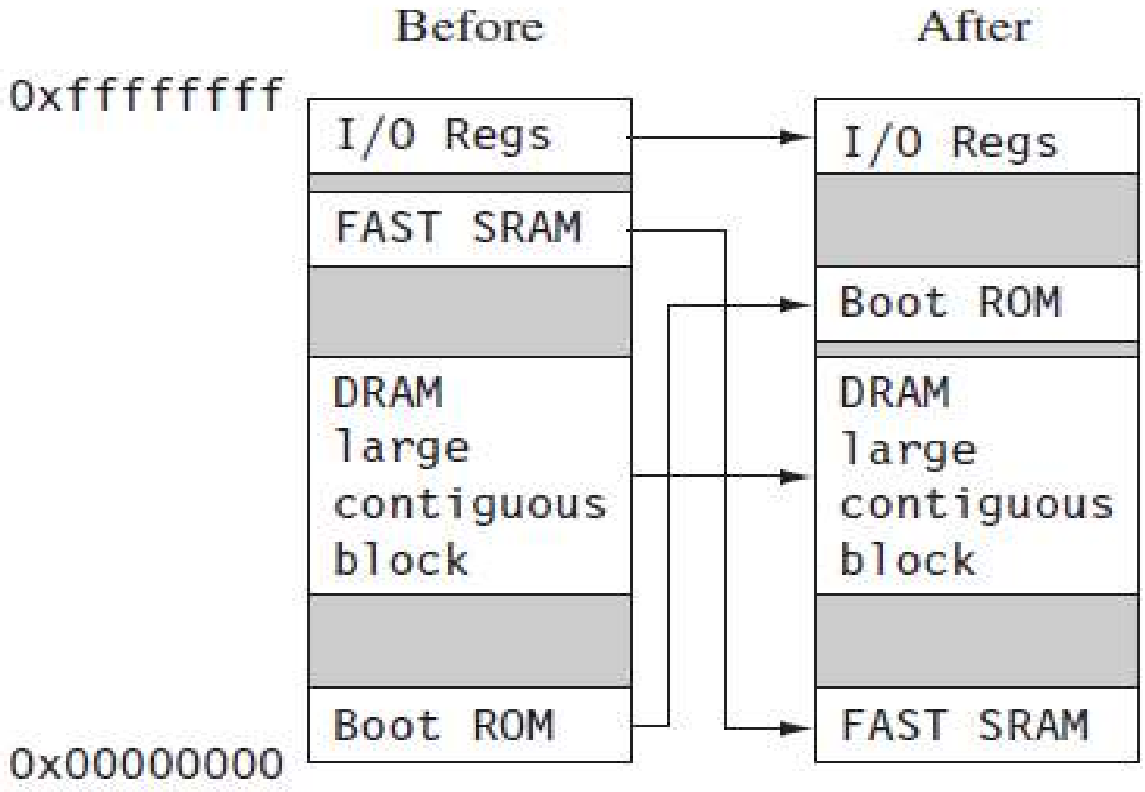
Downloaded by Sharada Kori (smkori@git.edu)

1. ***Initial hardware configuration*** involves setting up the target platform, so that it can boot an image

- The target platform comes up in a standard configuration; but, this configuration normally requires modification to satisfy the requirements of the booted image
  - For example, the memory system normally requires reorganization of the memory map
- *Initializing or organizing memory is an important part of the initialization code, because many operating systems expect a known memory layout before they can start*

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)



- *Memory remapping allows the system to start the initialization code from ROM at power-up*
- *The initialization code then redefines or remaps the memory map to place RAM at address 0x00000000—an important step because then the exception vector table can be in RAM and thus can be reprogrammed*

Downloaded by Śharadá Kori (smkori@git.edu)

2. ***Diagnostic***s 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

» It also tracks down standard system-related issues

» The primary purpose of diagnostic code is fault identification and isolation

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

3. ***Booting*** involves loading an image and handing control over to that image

- The boot process itself can be complicated if the system must boot different operating systems or different versions of the same operating system
  - Booting an image is the final phase, but first you must 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

Downloaded by Sharada Kori (smkori@git.edu)

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

- We can divide operating systems into *two main categories*:
  - real-time operating systems (RTOSs)
  - platform operating systems

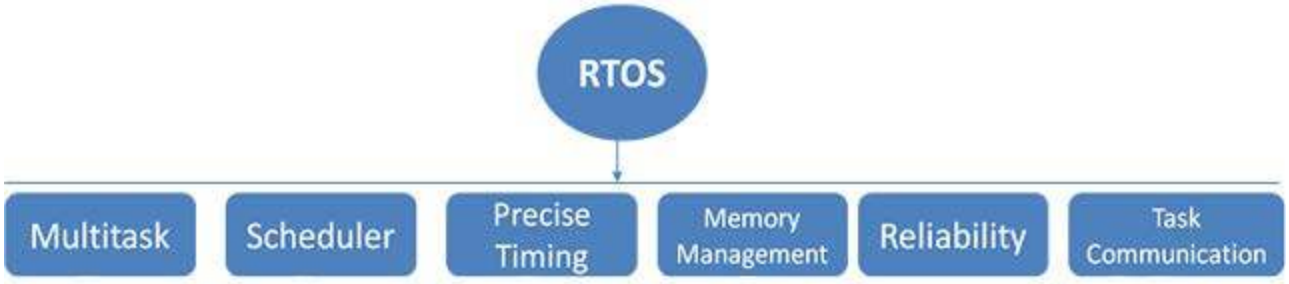
This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

1. *RTOSs* provide guaranteed response times to events

- Different operating systems have different amounts of control over the system response time

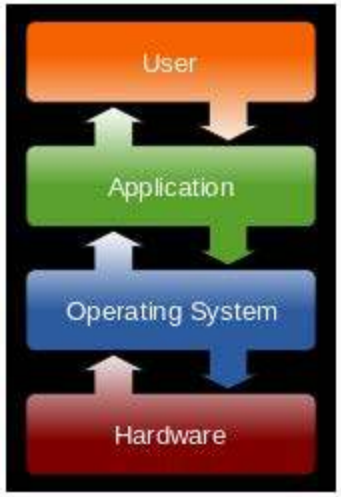
- A *hard real-time* application requires a guaranteed response to work at all
- A *soft real-time* application requires a good response time, but the performance degrades more gracefully if the response time overruns

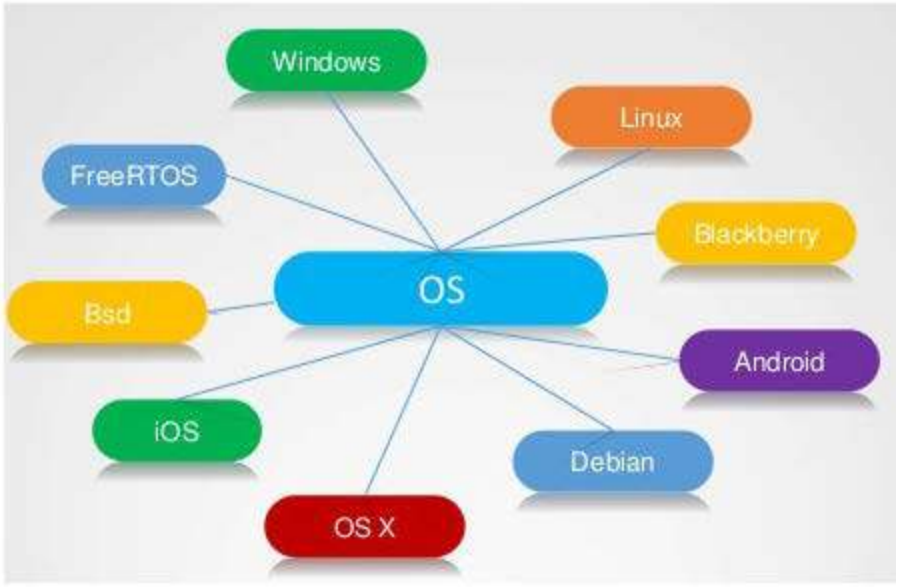


Downloaded by Sharada Kori (smkori@git.edu)

» ***Platform operating systems*** require a memory management unit to manage large, non-real-time applications and tend to have secondary storage

» The Linux operating system is a typical example of a platform operating system





This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

Applications

- The operating system schedules *applications*—code dedicated to handle 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, auto-motive, mobile and consumer devices, mass storage, and imaging

Downloaded by Sharada Kori (smkori@git.edu)

- ARM processor is found in networking applications like home gateways, DSL modems for high-speed Internet communication, and 802.11 wireless communications
- 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.
  - In contrast, ARM processors are not found in applications that require leading-edge high performance. Because these applications tend to be low volume and high cost, ARM has decided not to focus designs on these types of applications.

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

18CS44

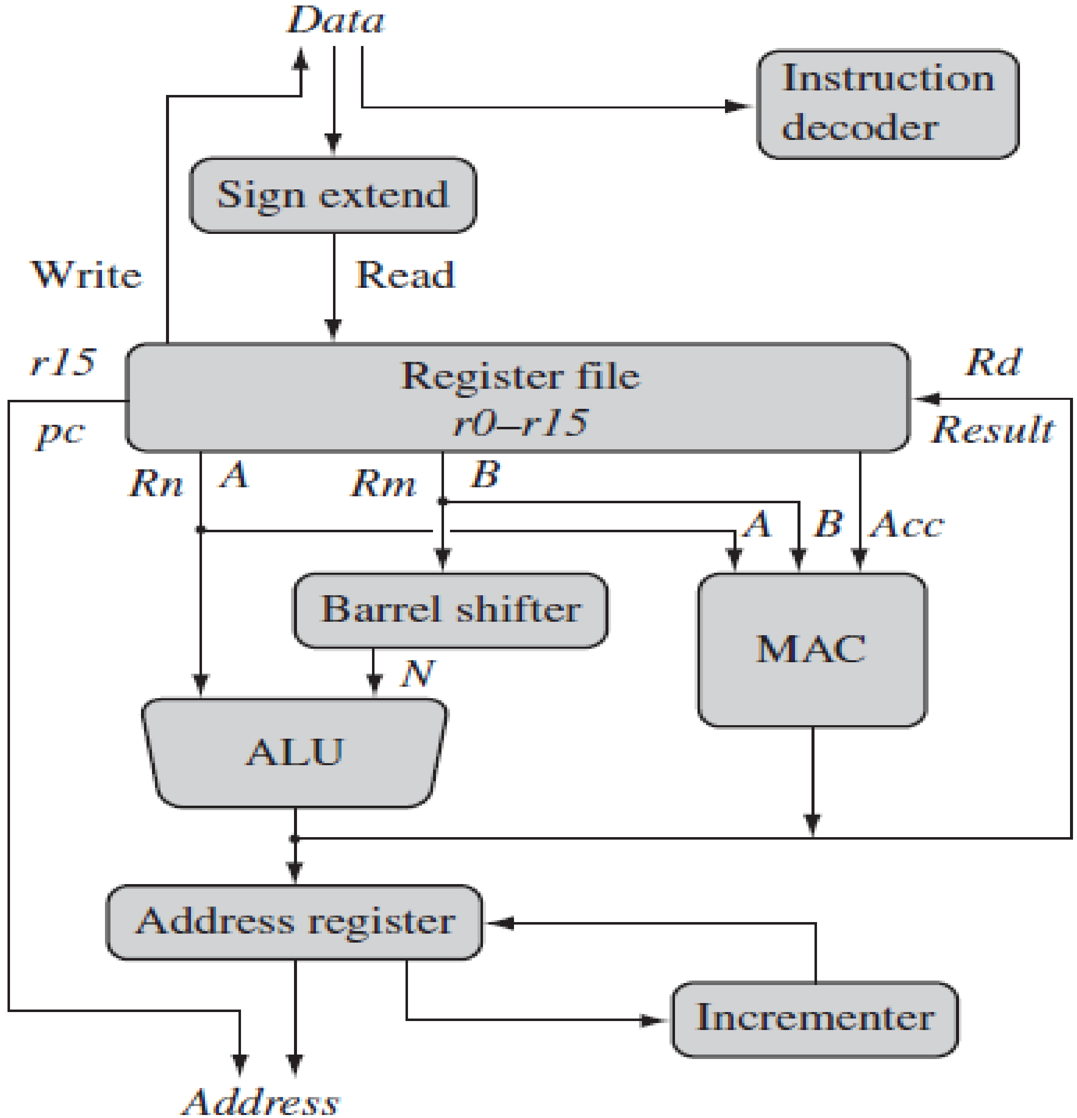
MICROCONTROLLER AND EMBEDDED SYSTEMS

MODULE 1

ARM PROCESSOR FUNDAMENTALS

Downloaded by Sharada Kori (smkori@git.edu)

» A programmer can think of an ARM core as functional units connected by data buses, as shown in the following Figure.



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)

This document is available free of charge on studocu
Downloaded by Sharada Kori (sakori@git.edu)

- 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***—means ARM 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 in registers

Downloaded by Sharada Kori (smkori@git.edu)

- 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*, *Rn* and *Rm*, and a single result or *destination register*, *Rd*
  - Source operands are read from the register file using the internal buses A and B, respectively

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

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

Downloaded by Sharada Kori (smkori@git.edu)

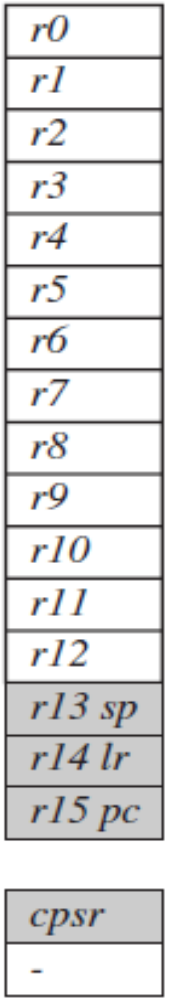
- After passing through the functional units, the result in *Rd* is written back to the register file using the *Result bus*
- For load and store instructions the *Incrementer* updates the address register before the core reads or writes the next register value from or to the next sequential memory location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

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

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.
  - Protected mode is normally used when executing applications
- The processor can operate in seven different modes
- There are up to 18 active 32-bit registers:
  - 16 data registers and 2 processor status registers.
  - The data registers visible to the programmer are *r0* to *r15*



Downloaded by Sharada Kori (smkori@git.edu)

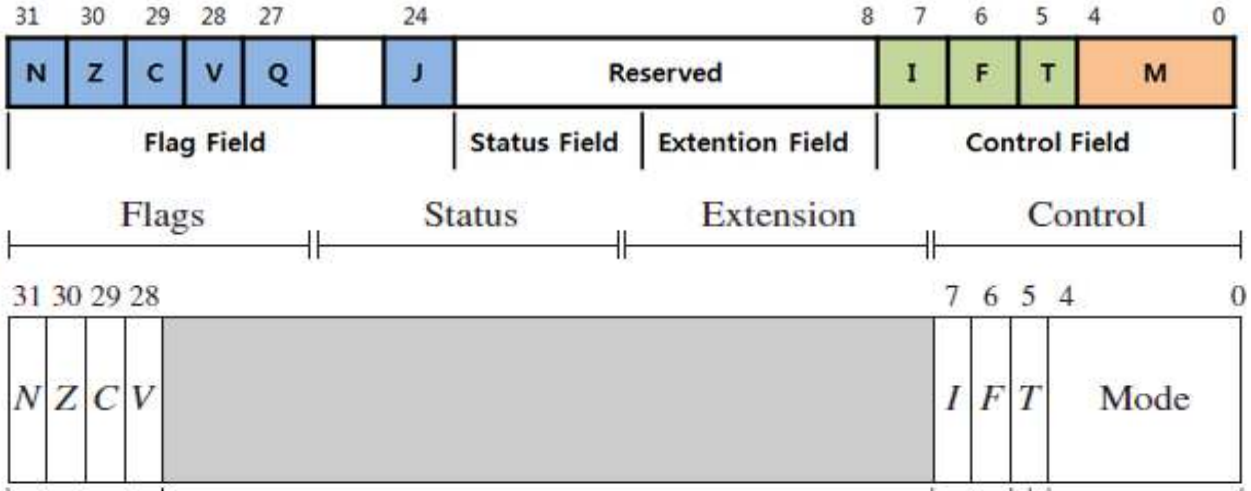
- Three registers of ARM are assigned to a particular task
  - *Register r13* is traditionally used as the ***stack pointer (sp)*** and stores the head of the stack in the current processor mode
  - *Register r14* is called the ***link register (lr)*** and is where the core puts the return address whenever 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 ARM state the registers *r0* to *r13* are orthogonal
- There are two program status registers:
  - ***cpsr (current program status register)***
  - ***spsr (saved program status register)***

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

CURRENT PROGRAM STATUS REGISTERS

- The ARM core uses the *cpsr* to monitor and control internal operations.
- The *cpsr* is a dedicated 32-bit register and resides in the register file.
- The following Figure shows the basic layout of a generic program status register



Flag Field	
N	Negative result from ALU
Z	Zero result from ALU
C	ALU operation caused Carry
V	ALU operation oVerflowed
Q	ALU operation saturated
J	Java Byte Code Execution

Control bits	
I	1: disables IRQ
F	1: disables FIQ
T	1: Thumb, 0: ARM

Mode bits

- Specify the processor mode

Mode bits M[4:0]	
0b10000	User
0b11111	System
0b10001	FIQ
0b10010	IRQ
0b10011	SVC(Supervisor)
0b10111	Abort
0b11011	Undefined

Downloaded by Sharada Kori (smkori@git.edu)

- The *cpsr* is divided into four fields, each 8 bits wide: *flags*, *status*, *extension*, and *control*
  - In current designs the extension and status fields are reserved for future use
- The ***control field*** contains the *processor mode*, *state*, and *interrupt mask* bits
- The ***flags field*** contains the *condition flags*
- Some ARM processor cores have extra bits allocated
  - For example, the *J bit*, which can be found in the flags field, is only available on *Jazelle-enabled processors*, which execute 8-bit instructions
- It is highly probable that future designs will assign extra bits for the monitoring and control of new features

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

Processor Modes:

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

Downloaded by Sharada Kori (smkori@git.edu)

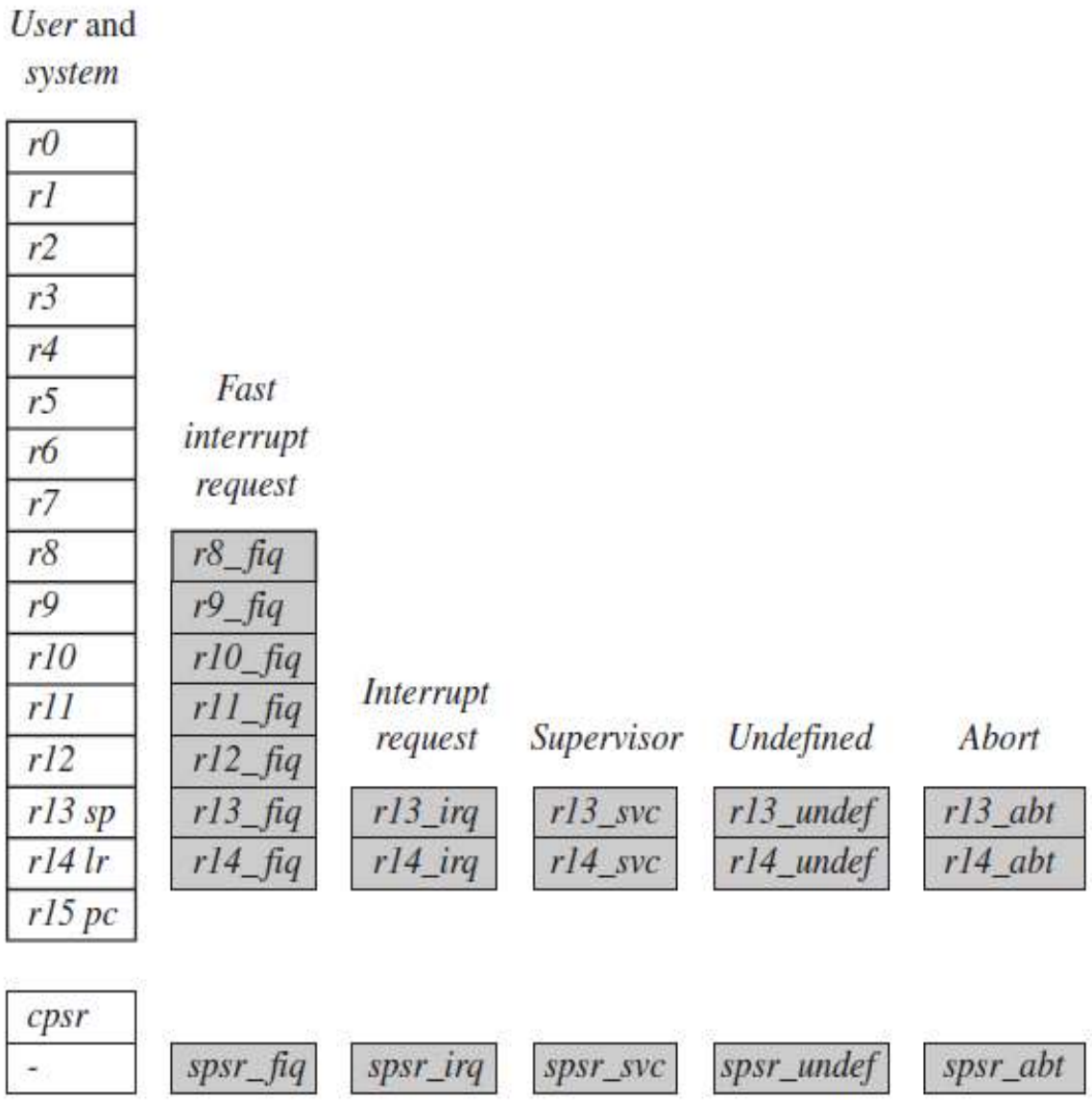
» There are ***seven*** *processor modes* in total:

- ***six*** *privileged modes*
  - 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
- ***one*** *non-privileged mode*
  - ***User mode*** is used for programs and applications

This document is available free of charge at studocu.com
Downloaded by Sharada Kori (smkori@git.edu)

Banked Registers:

- The following Figure shows all 37 registers in the register file
- Of these, 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



Downloaded by Sharada Kori (smkori@git.edu)

- *Banked registers* 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
- 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

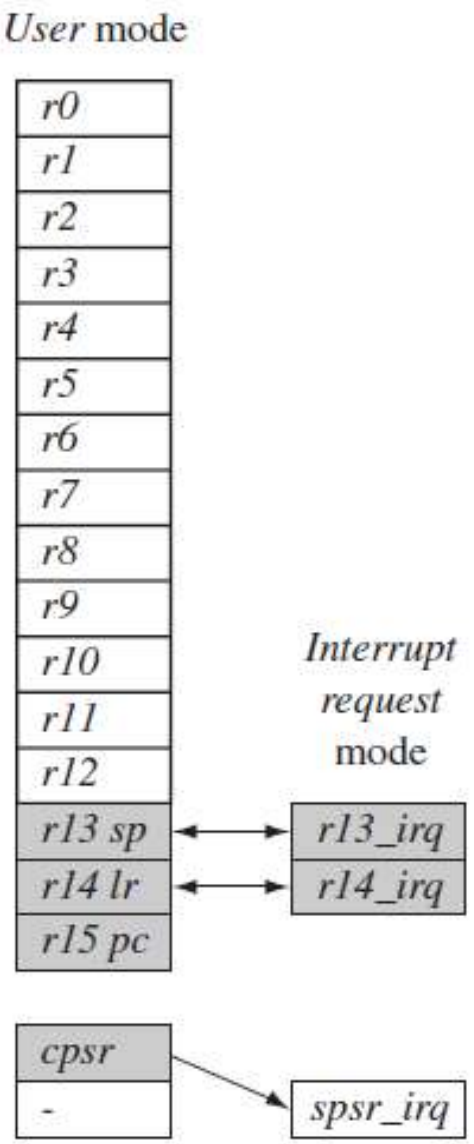
This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

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

Downloaded by Sharada Kori (smkori@git.edu)

- The 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



This document is available free of charge on studocu
Downloaded by Sharada Kori (smkori@git.edu)

- The above 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* being copied into *spsr_irq*
- 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. There is no *spsr* available in user mode

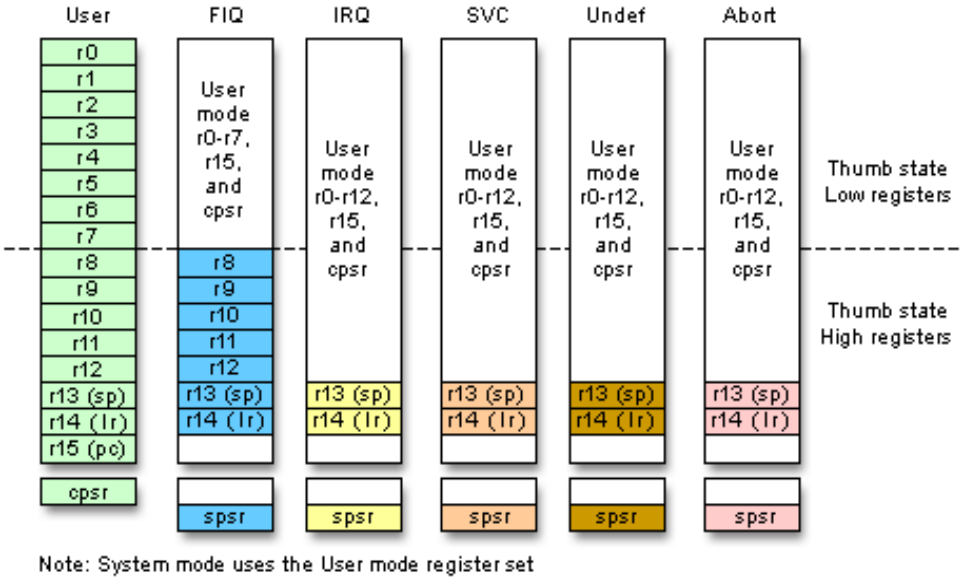
Downloaded by Sharada Kori (smkori@git.edu)

- 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
- When power is applied to the core, it starts in supervisor mode, which is privileged.
  - Starting in a privileged mode is useful since initialization code can use full access to the *cpsr* to set up the stacks for each of the other modes
- The following Table lists the various modes and the associated binary patterns. The last column of the table gives the bit patterns that represent each of the processor modes in the *cpsr*

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

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



Downloaded by Sharada Kori (smkori@git.edu)

State and Instruction Sets:

- The state of the core determines which instruction set is being executed. There are three instruction sets:
  - 32-bit ARM instruction set – 32-bit Load / Store architecture with every instruction being conditional
  - 16-bit Thumb instruction set – 16-bit with only branch instructions being conditional and only half of the registers used
  - 8-bit Jazelle instruction set – Allows Java byte code to be directly executed in ARM architecture
- The ***ARM instruction set*** is only active when the processor is in ARM state
- The ***Thumb instruction set*** is only active when the processor is in Thumb state
  - Once in Thumb state the processor is executing purely Thumb 16-bit instructions
- You cannot inter-mingle sequential ARM, Thumb, and Jazelle instructions

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

- 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
- To change states the core executes a specialized branch instruction
- The following Table compares the ARM and Thumb instruction set features

-	**ARM (*cspr T* = 0)**	**Thumb (*cspr T* = 1)**
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution	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

Downloaded by Sharada Kori (smkori@git.edu)

- 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 byte-codes
- To execute Java byte-codes, you require the Jazelle technology plus a specially modified version of the Java virtual machine
- The following Table gives the Jazelle instruction set features

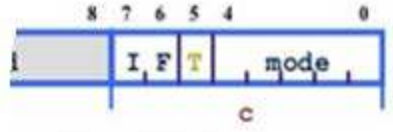
-	**Jezelle (*cspr T* = 0, *J* = 1)**
Instruction size	8-bit
Core Instructions	Over 60% of the Java byte-codes are implemented in hardware; the rest of the codes are implemented in software

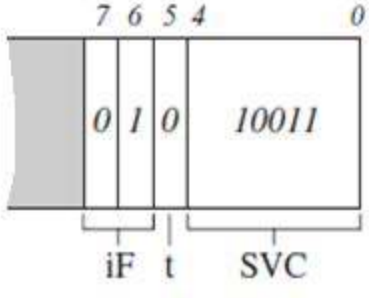
This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

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
  - The *F* bit masks FIQ when set to binary 1





Downloaded by Sharada Kori (smkori@git.edu)

Condition Flags:

- Condition flags are updated by comparisons and the result of ALU operations that specify the **S** instruction suffix
  - For example, if a SUBS subtract instruction results in a register value of zero, then the *Z* Flag in the *cpsr* is set
- With processor cores that include the DSP extensions, the *Q* bit indicates if an overflow or saturation has occurred in an enhanced DSP instruction
- The flag is "sticky" in the sense that the hardware only sets this flag. To clear the flag you need to write to the *cpsr* directly

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

- In Jazelle-enabled processors, the *J* bit reflects the state of the core;
  - if it is set, the core is in Jazelle state
- The *J* bit is not generally usable and is only available on some processor cores
- To take advantage of Jazelle, extra software has to be licensed from both ARM Limited and Sun Microsystems
- Most ARM instructions can be executed conditionally on the value of the condition flags

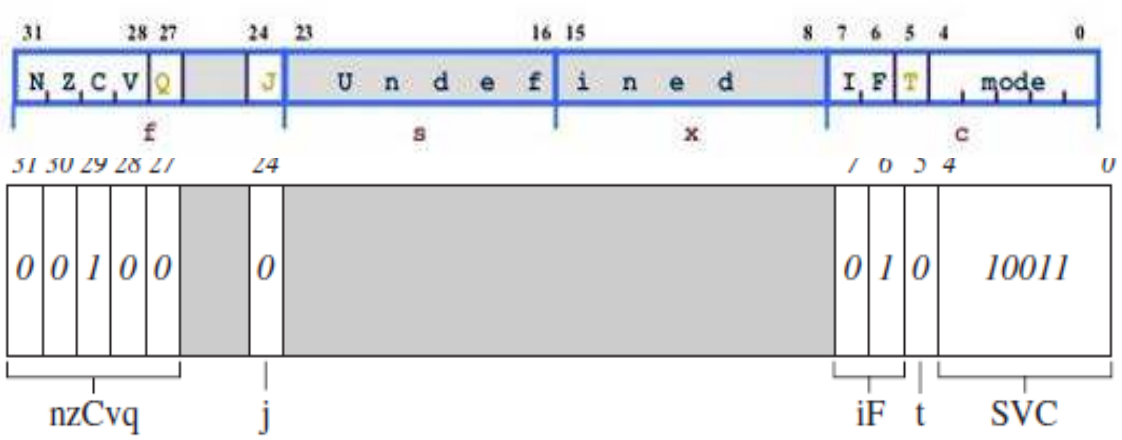
Downloaded by Sharada Kori (smkori@git.edu)

» The following Table lists the condition flags and a short description on what causes them to be set

» These flags are located in the most significant bits in the *cpsr*. These bits are used for conditional execution

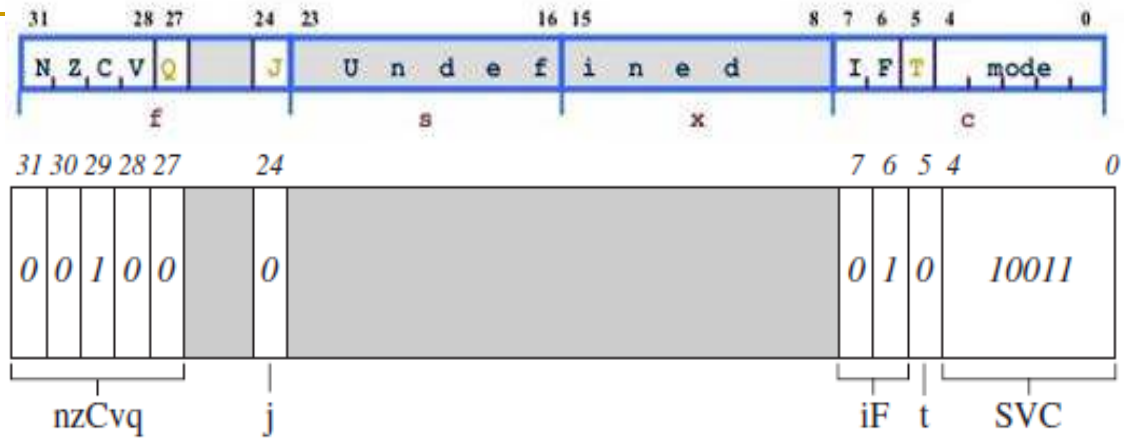
Flag	Flag Name	Set When
Q	Saturation	the result causes an overflow and/or saturation
V	oVerflow	the result causes a signed overflow
C	Carry	the result causes an unsigned carry
Z	Zero	the result is zero
N	Negative	bit 31 of the result is a binary 1

» The following Figure shows a typical value for the *cpsr* with both DSP extensions and Jazelle



This document is available free of charge on studocu
Downloaded by Sharada Kori (smkori@git.edu)

» cspr = 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

  » In the *cpsr* example shown in above 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

» Finally, you can see from the Figure, the processor is in *supervisor (SVC) mode*, since the mode[4:0] is equal to binary 10011.

» **cspr = nZcvqjIft_SYS ???**

Downloaded by Sharada Kori (smkori@git.edu)

Conditional Execution:

- Conditional execution controls whether or not the core will execute an instruction
- Prior to execution, the processor compares the condition attribute with the condition flags in the *cpsr*.
  - If they match, then the instruction is executed; otherwise the instruction is ignored
- The condition attribute is post-fixed to the instruction mnemonic, which is encoded into the instruction
- The following Table lists the conditional execution code mnemonics
  - When a condition mnemonic is not present, the default behavior is to set it to always (AL) execute

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

» The following Table lists the conditional execution code mnemonics

» When a condition mnemonic is not present, the default behavior is to set it to always (AL) execute

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

```
    CMP   r3, #0
    BEQ   next
    ADD   r0, r0, r1
    SUB   r0, r0, r2
next
...
```

```
CMP   r0, #0    ; if (x <= 0)
MOVLE r0, #0    ;    x = 0;
MOVGT r0, #1    ; else x = 1;
```

```
CMP   r0, #'A' ; if (c == 'A'
CMPNE r0, #'B' ;  || c == 'B')
MOVEQ r1, #1   ;    y = 1;
```

```
    MOV   r1, #10
loop
    ...
    SUBS  r1, r1, #1
    BNE   loop
```

Downloaded by Sharada Kori (smkori@git.edu)

PIPELINE

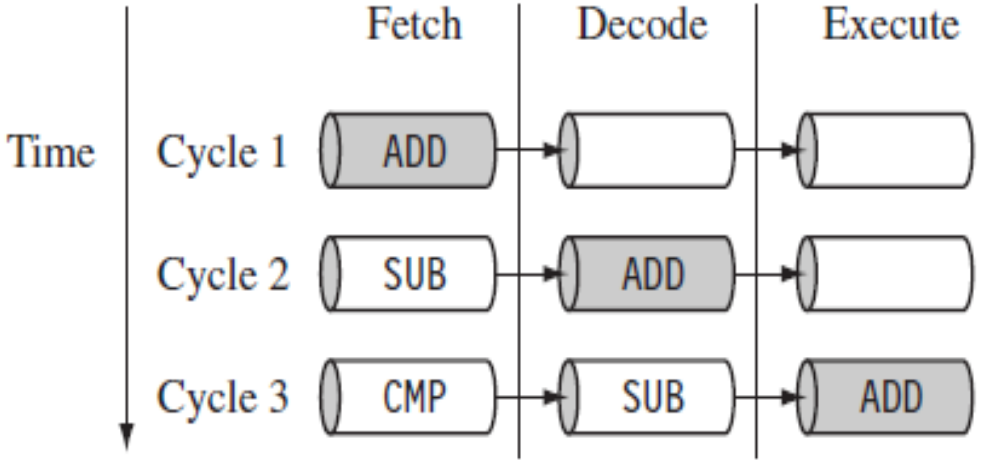
- A *pipeline* is the mechanism in a RISC processor, which is used to execute instructions
- Pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed

- The above Figure shows a Three-Stage Pipeline in ARM7 Processor
  - *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

This document is available free of charge on studocu
Downloaded by Sharada Kori (smkori@git.edu)

» The following Figure illustrates pipeline using a simple example

» The Figure shows a sequence of three instructions being fetched, decoded, and executed by the processor



- » 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***

Downloaded by Sharada Kori (smkori@git.edu)

- 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 increased pipeline length also means
  - increased *system latency* and
  - there can be *data dependency* between certain stages

Latency is a networking term. When a data packet is transmitted and returned back to its source, the total time for the round trip is known as latency.

Latency refers to time interval or delay when a system component is waiting for another system component to do something. This duration of time is called latency.

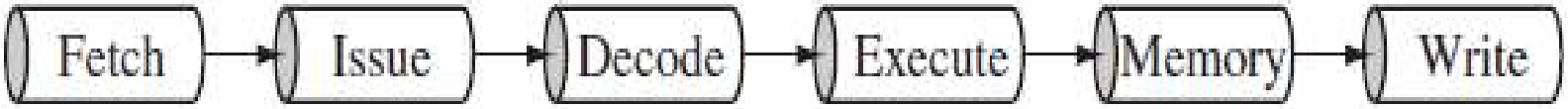
This document is available free of charge on studocu
Downloaded by Sharada Kori (smkori@git.edu)

- The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages, as shown in Figure

- The ARM9 adds a memory and writeback stage, which allows the ARM9 to –
  - process on average 1.1 Dhrystone MIPS per MHz
  - increase the instruction throughput in ARM9 by around 13% compared with an ARM7

Downloaded by Sharada Kori (smkori@git.edu)

» The ARM10 increases the pipeline length still further by adding a sixth stage, as shown in the following Figure



» The ARM10 –

» can process on average 1.3 Dhrystone MIPS per MHz

» have about 34% more throughput than an ARM7 processor core

» but again at a higher latency cost

» ***NOTE:*** Even though the ARM9 and ARM10 pipelines are different, they still use the same *pipeline executing characteristics* as an ARM7

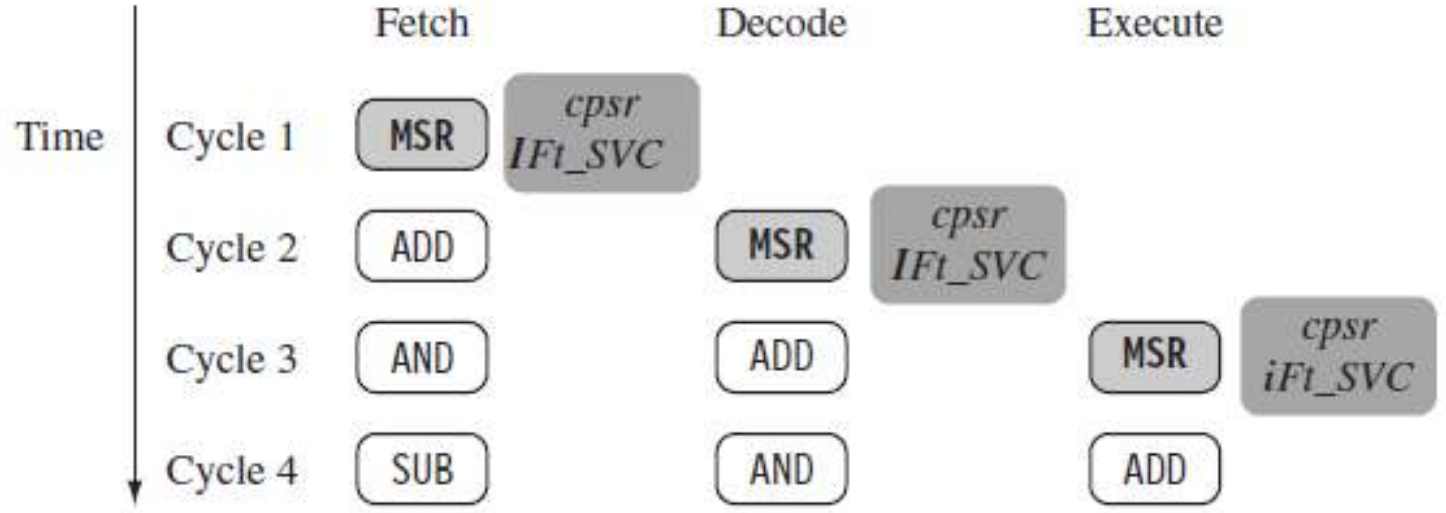
» Hence, the code written for the ARM7 will execute on an ARM9 or ARM10

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

Pipeline Executing Characteristics:

- The ARM pipeline will not process 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 following Figure shows an instruction sequence on an ARM7 pipeline



Downloaded by Sharada Kori (smkori@git.edu)

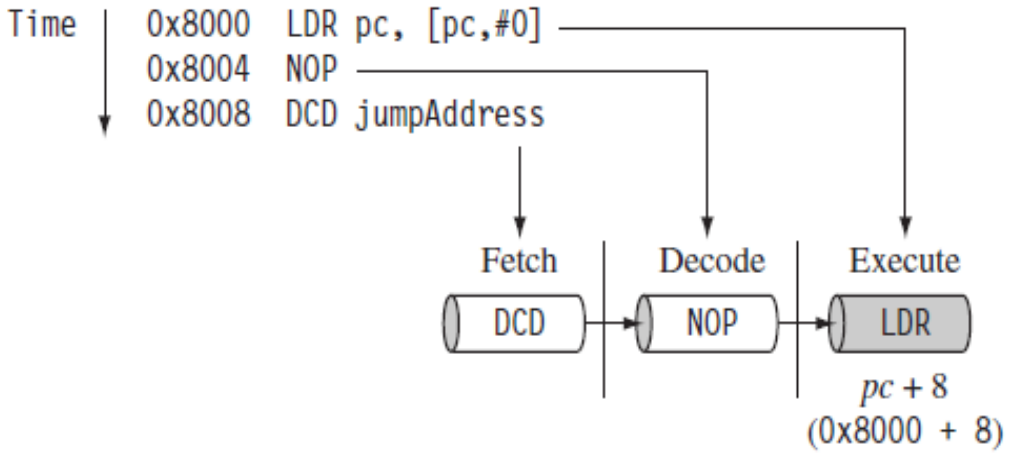
» The MSR instruction is used to enable IRQ interrupts, which only occurs once the MSR instruction completes the execute stage of the pipeline. It clears the *I* bit in the *cpsr* to enable the IRQ interrupts

» Once the ADD instruction enters the execute stage of the pipeline, IRQ interrupts are enabled

» The following 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



» Note when the processor is in Thumb state the *pc* is the instruction address plus 4

This document is available free of charge on studocu
Downloaded by Sharada Kori (smkori@git.edu)

- 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

Downloaded by Sharada Kori (smkori@git.edu)

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 (or in some processors starting at the offset 0xffff0000) 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 the following Table)

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

Exception/Interrupt	Shorthand	Address	High Address
Reset	RESET	0x00000000	0x00000000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	SABT	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

Downloaded by Sharada Kori (smkori@git.edu)

- ***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*

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

CORE EXTENSIONS

- *Core extensions* are the standard hardware components placed next to the ARM core
  - They improve performance, manage resources, and provide extra functionality
  - They are designed to provide flexibility in handling particular applications
- Each ARM family has different extensions available –
- There are *three hardware extensions*: cache and tightly coupled memory, memory management, and the coprocessor interface

Downloaded by Sharada Kori (smkori@git.edu)

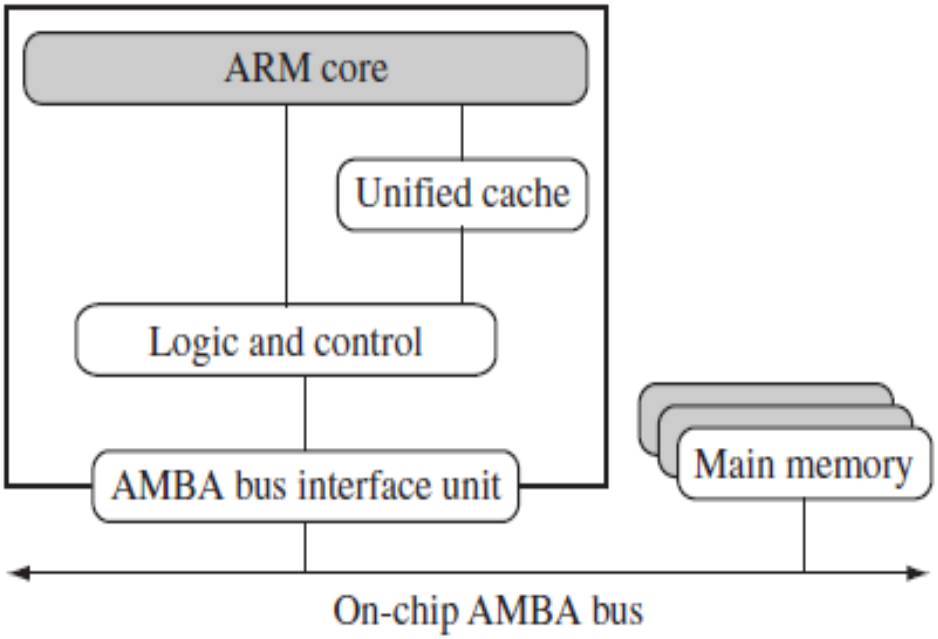
Cache Tightly Coupled Memory:

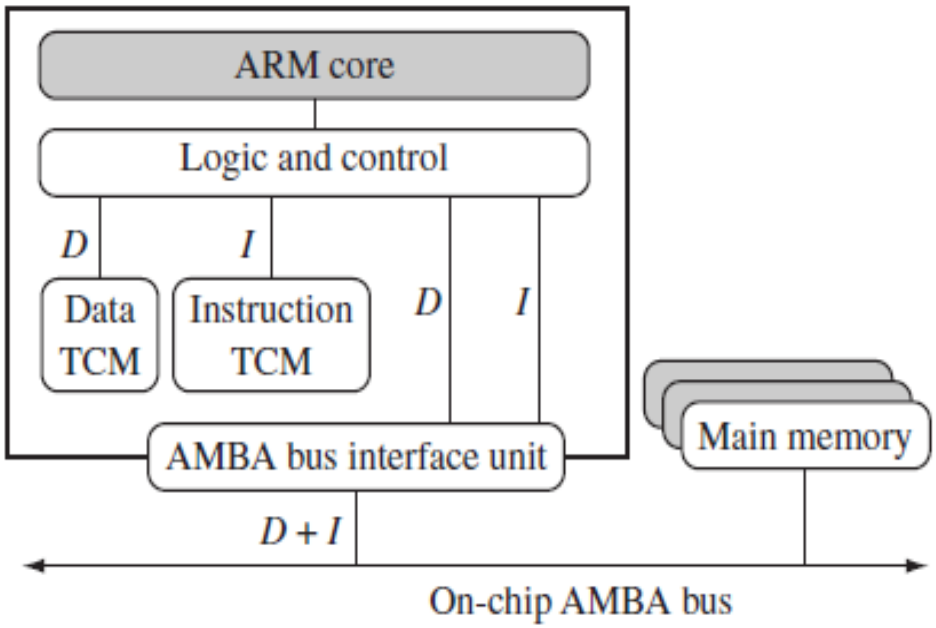
- The cache is a block of fast memory placed between main memory and the core
  - It allows for more efficient fetches from some memory types
  - With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory
- Most ARM-based embedded systems use a single-level cache internal to the processor
- ARM has *two forms of cache* –
  - The first is found attached to the Von Neumann–style cores
  - The second form, attached to the Harvard-style cores

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

» Von Neumann–style cores – combines both data and instruction into a single unified cache





» Harvard-style cores – has separate caches for data and instruction

Downloaded by Sharada Kori (smkori@git.edu)

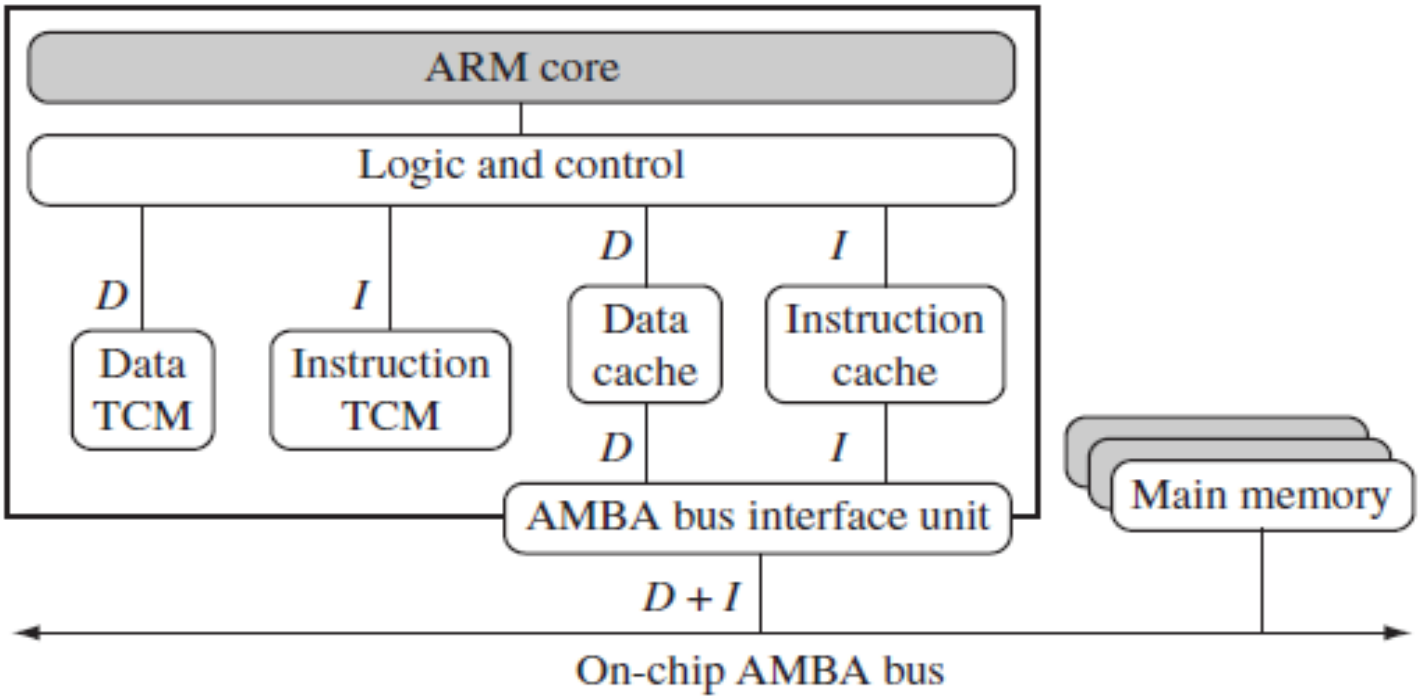
- A cache provides an overall increase in performance, but at the expense of predictable execution
- But the real-time systems require the code execution to be deterministic— the time taken for loading and storing instructions or data must be predictable
- This is achieved using a form of memory called *tightly coupled memory (TCM)*. TCM is fast SRAM located close to the core and guarantees the clock cycles required to fetch instructions or data
- TCMs appear as memory in the address map and can be accessed as fast memory

This document is available free of charge on studocu

Downloaded by Sharada Kori (smkori@git.edu)

» By combining both technologies, ARM processors can have both improved performance and predictable real-time response

» The following Figure shows an example core with a combination of caches and TCMs



Downloaded by Sharada Kori (smkori@git.edu)

Memory Management:

- Embedded systems often use multiple memory devices
- It is necessary to have a method to organize these devices and protect the system from applications trying to make inappropriate accesses to hardware
- This is achieved with the assistance of memory management hardware
- ARM cores have *three different types of memory management hardware*—
  - no extensions providing no protection
  - a memory protection unit (MPU) providing limited protection
  - a memory management unit (MMU) providing full protection

This document is available free of charge on studocu

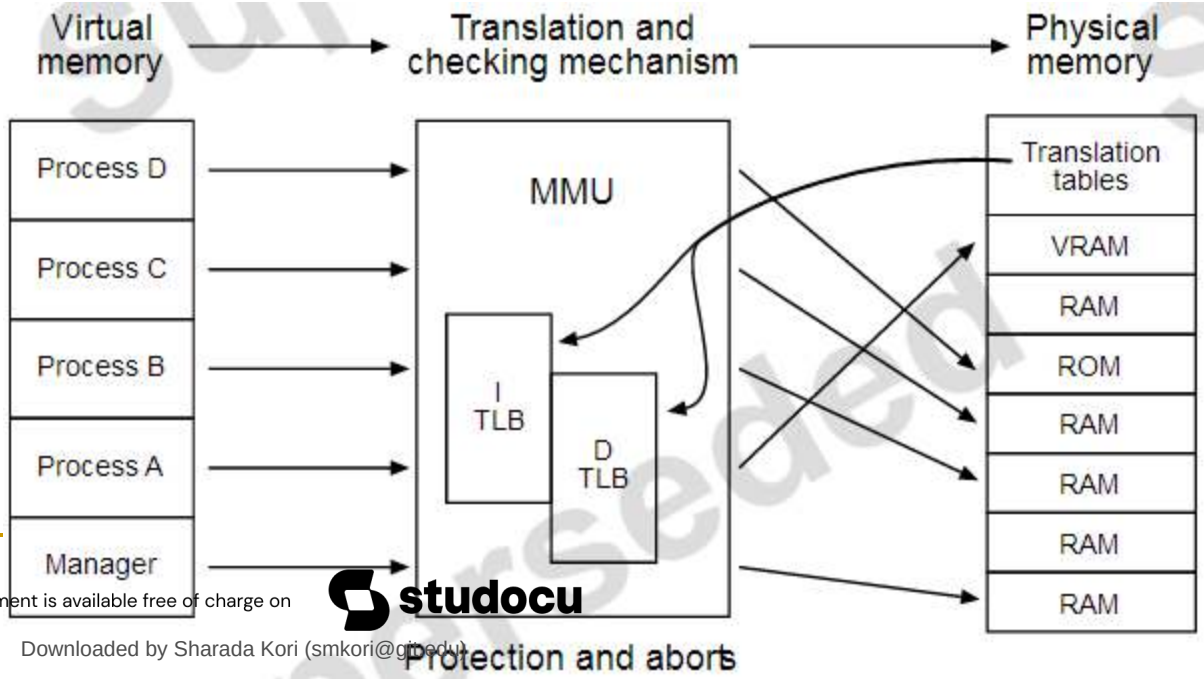
Downloaded by Sharada Kori (smkori@git.edu)

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

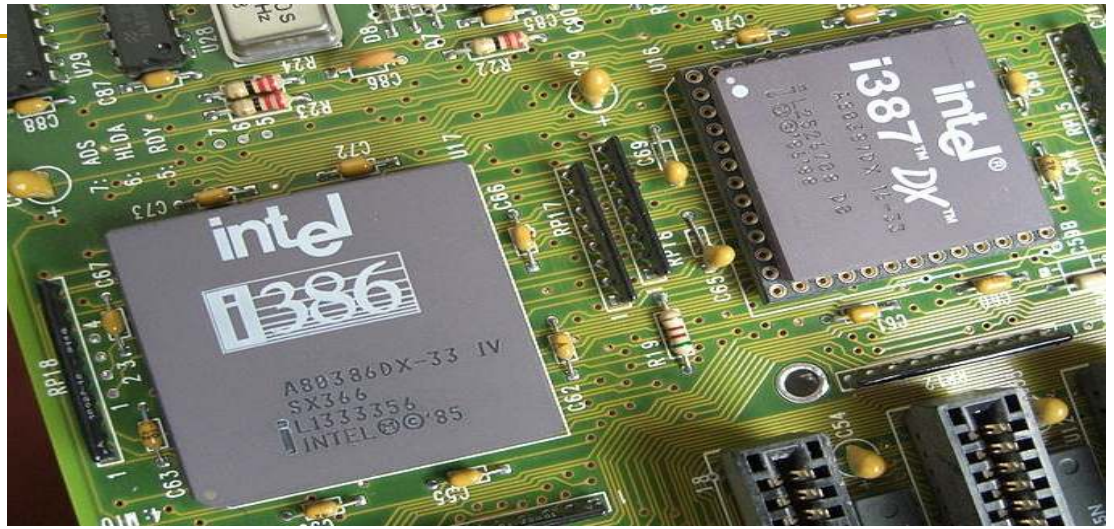
- ***MPU***s employ a simple system that uses a limited number of memory regions
- These regions are controlled with a set of special coprocessor registers, and each region is defined with specific access permissions
- This type of memory management is used for systems that require memory protection but don't have a complex memory map

Downloaded by Sharada Kori (smkori@git.edu)

- ***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



This document is available free of charge on studocu
Downloaded by Sharada Kori (smkori@gitedu)



Coprocessors:

- Coprocessors can be attached to the ARM processor
  - 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 the ARM core via the coprocessor interface
- The coprocessor can be accessed through a group of dedicated ARM instructions that provide a load-store type interface
  - For example, coprocessor 15: The ARM processor uses coprocessor 15 registers to control the cache, TCMs, and memory management

Downloaded by Sharada Kori (smkori@git.edu)

- The coprocessor can also extend the instruction set by providing a specialized group of new instructions
  - For example, there are a set of specialized instructions that can be added to the standard ARM instruction set to process vector floating-point (VFP) operations
- These new instructions are processed in the decode stage of the ARM pipeline.
  - If the decode stage sees a coprocessor instruction, then it offers it to the relevant coprocessor
  - If the coprocessor is not present or doesn't recognize the instruction, then the ARM takes an undefined instruction exception, which allows you to emulate the behavior of the coprocessor in software

This document is available free of charge on studocu
Downloaded by Sharada Kori (smkori@git.edu)