



Module-4

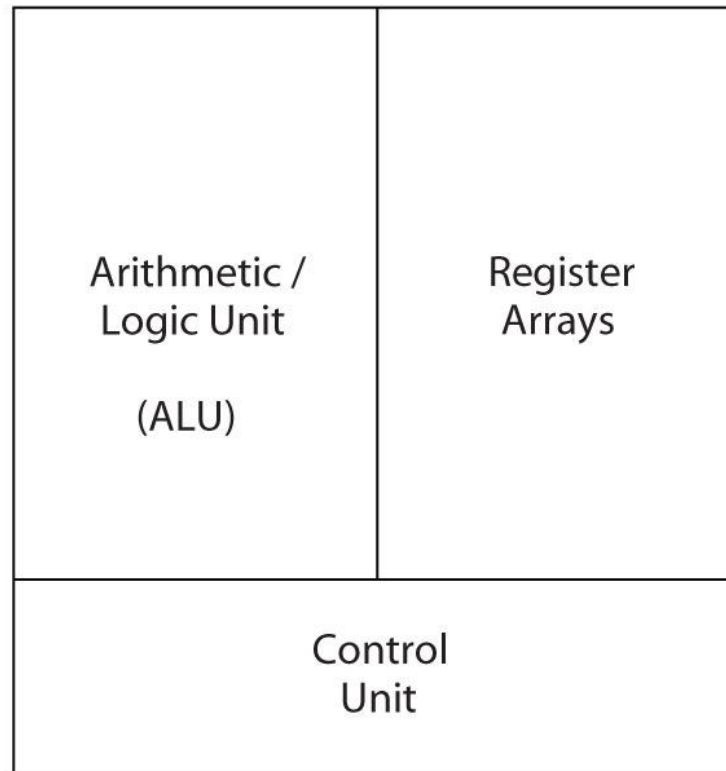
Prepared by
Prof.S.Mahalakshmi
AP/ISE
BMSIT&M

Books Referred

- ARM system developers guide, Andrew N Sloss, Dominic Symes and Chris Wright, Elsevier, Morgan Kaufman publishers, 2008.

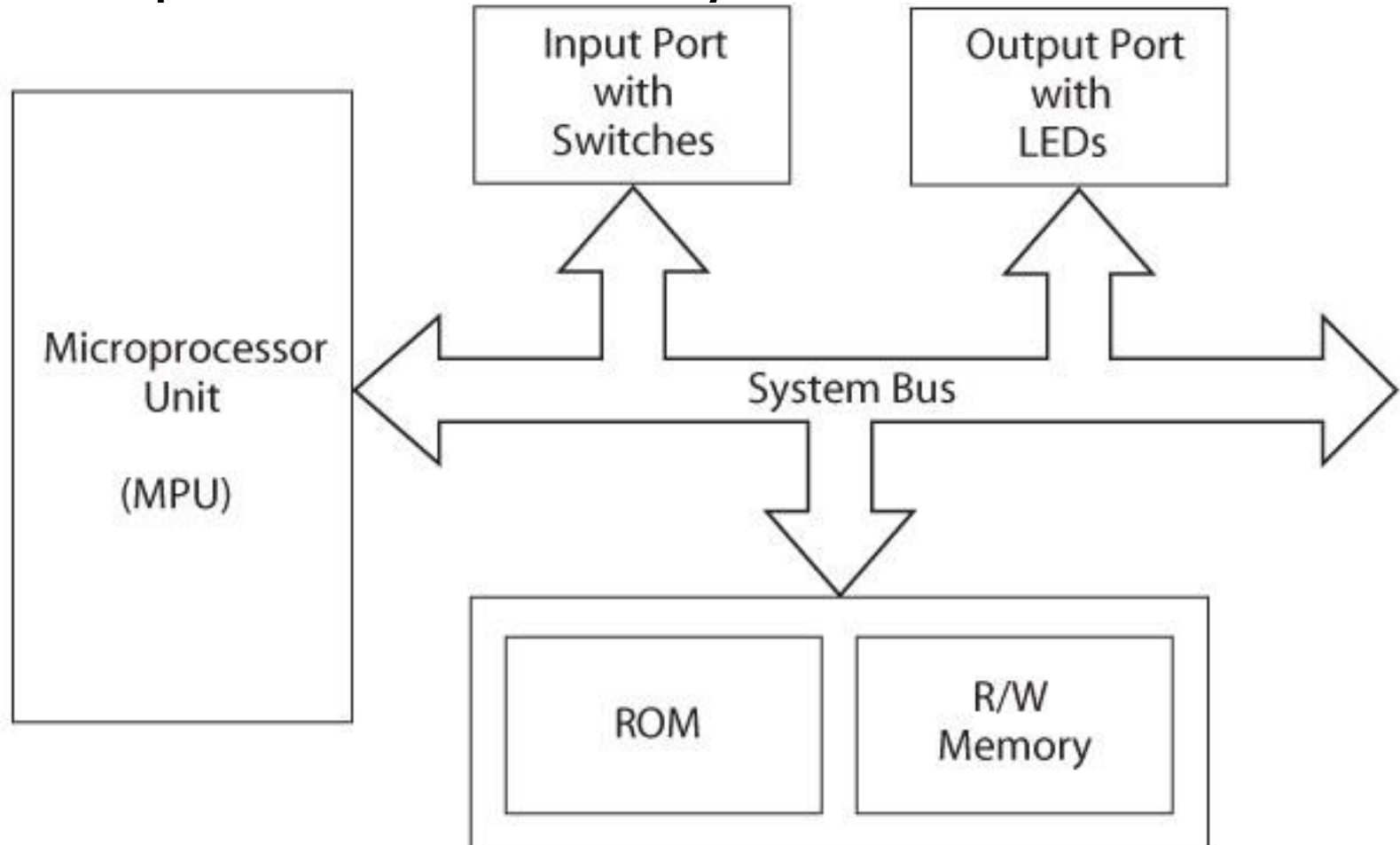
4.1 Microprocessors versus Microcontrollers

- Microprocessor



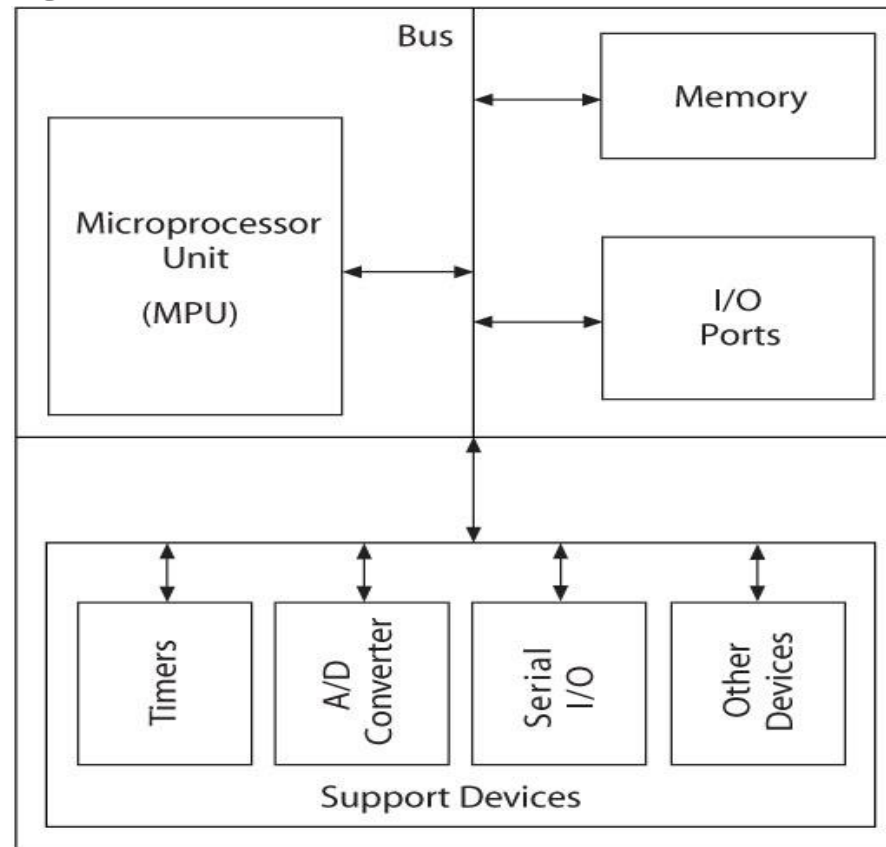
Microprocessors versus Microcontrollers

- Microprocessor based system



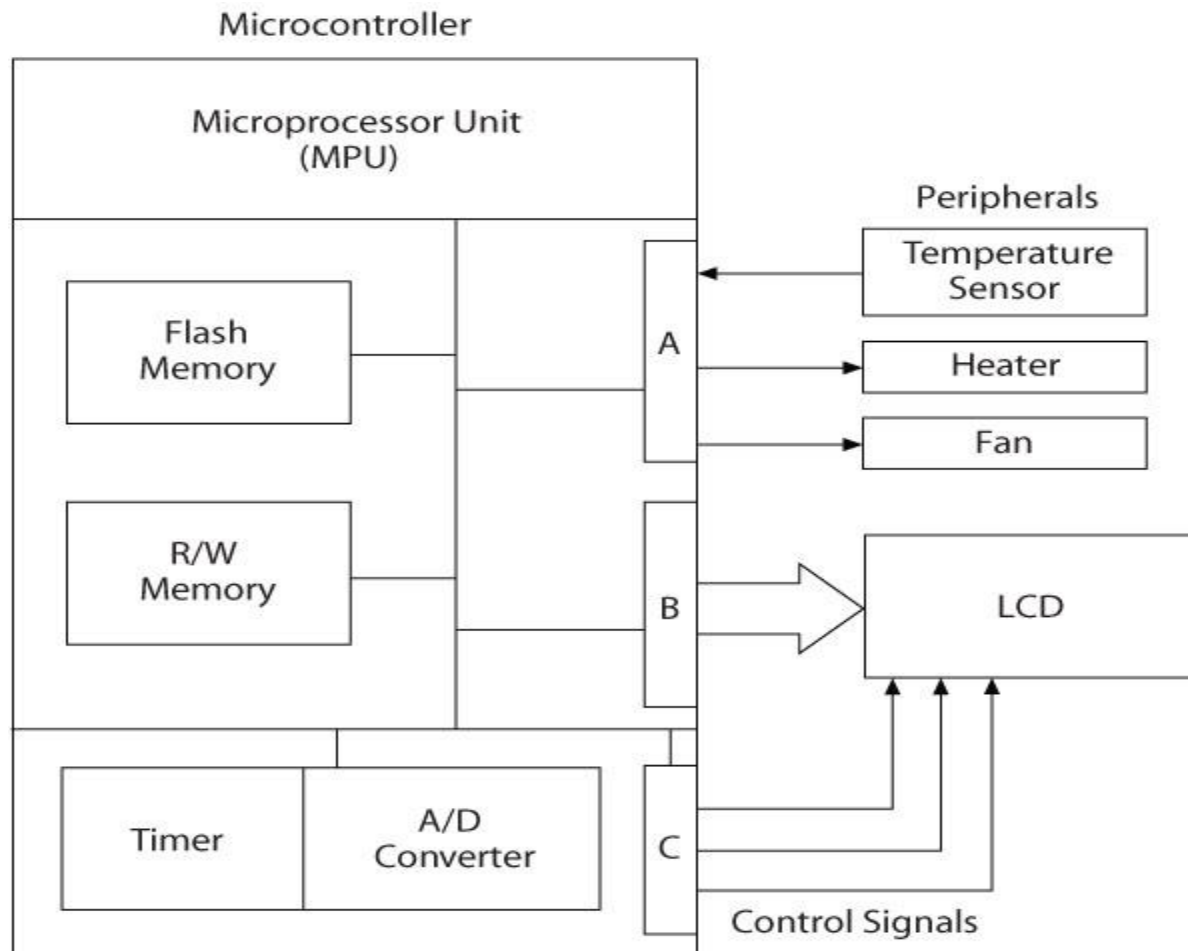
Microprocessors versus Microcontrollers

- Microcontroller



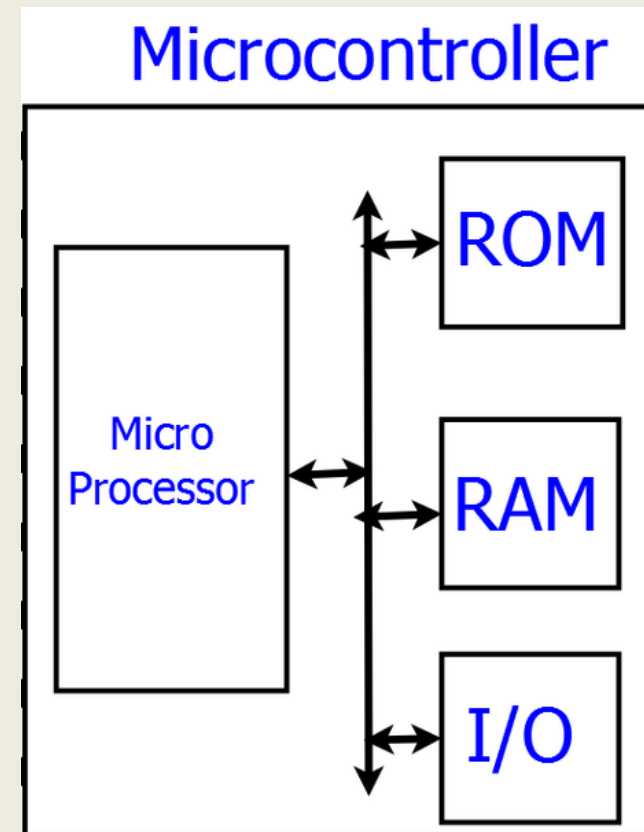
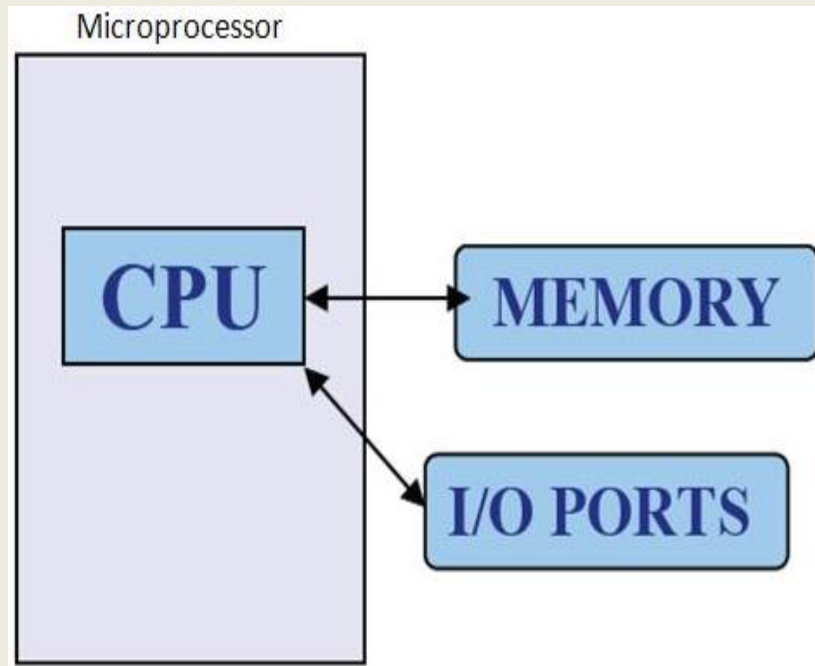
Microprocessors versus Microcontrollers

- Microcontroller based system



Microprocessors versus Microcontrollers

- Microprocessors
- Microcontrollers



Sl.No	Microprocessors	Microcontrollers
1.	CPU is stand-alone, RAM, ROM, I/O, timer are separate	CPU, RAM, ROM, I/O and timer are all on a single chip
2.	designer can decide on the amount of ROM, RAM and I/O ports.	fixed amount of on-chip ROM, RAM, I/O ports
3.	Expensive, versatility	for applications in which cost, power and space are critical
4.	general-purpose	single-purpose (control-oriented)
5.	High processing power	Low processing power
6.	High power consumption	Low power consumption
7.	Instruction sets focus on processing-intensive operations	Instruction sets focus on control and bit-level operations
8.	Typically 32/64 – bit	Typically 8/16 bit
9.	Typically deep pipeline (5-20 stages)	Typically single-cycle/two-stage pipeline



ARM Embedded Systems- Introduction



- 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 implementation
- 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 them are 32-bit embedded processors
- Used especially in portable devices due to its low power consumption and reasonable performance

ARM processors

- A simple but powerful design
- A whole family of designs sharing similar design principles and a common instruction set

Naming ARM

- **ARM**^{xyz}TDMIEJFS
 - 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: Synthesizable 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 comparison

ARM family attribute comparison.

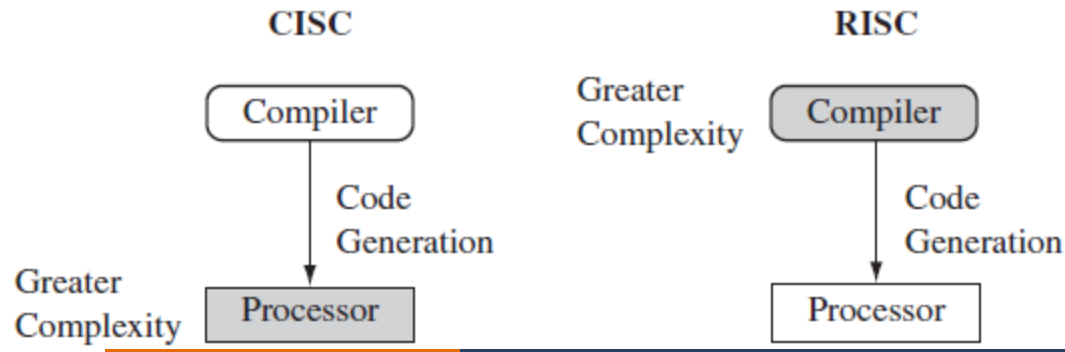
Year	1995	1997	1999	2003
	ARM7	ARM9	ARM10	ARM11
Pipeline depth	three-stage	five-stage	six-stage	eight-stage
Typical MHz	80	150	260	335
mW/MHz ^a	0.06 mW/MHz	0.19 mW/MHz (+ cache)	0.5 mW/MHz (+ cache)	0.4 mW/MHz (+ cache)
MIPS ^b /MHz	0.97	1.1	1.3	1.2
Architecture	Von Neumann	Harvard	Harvard	Harvard
Multiplier	8 × 32	8 × 32	16 × 32	16 × 32

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

^b MIPS are Dhrystone VAX MIPS.

4.2 The RISC design philosophy

- RISC: simple but powerful instructions that execute within a single cycle at high clock speed.
- RISC provides greater flexibility and intelligence in software rather than hardware (compiler)
- 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
- Results in simple design and fast clock rate



The RISC –major design roles

- **Instruction**

- RISC processor have a **Reduced number of instruction classes**.
- The classes provides 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.
- Each instruction is a **fixed length** to allow the pipeline to fetch future inst before decoding the current instruction.
- 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.
- The pipeline advances by one step on each cycle for max throughput.
- Instructions can be decoded in one pipeline stage
- There is no need for an instruction to be executed by a mini-program (microcode) as on CISC processor.

The RISC design philosophy

- **Register**

- RISC have a large General Purpose Registers (GPR) set. (either data or an address)
- Registers act as the fast local memory store for all data processing operations.
- In contrast, CISC processors have dedicated registers for specific purposes.

- **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.

Comparison of CISC and RISC

Sl.No	CISC	RISC
1.	Complex Instructions taking multiple cycles	Simple instructions taking 1 cycle
2.	Any instruction may reference memory	Only LOAD/STORE references memory
3.	NOT pipelined or less pipelined	Highly pipelined
4.	Instructions interpreted by microcode	Instructions executed by the hardware
5.	Variable length instructions	Fixed length instructions
6.	Many instructions and modes	Few instructions and modes
7.	Complexity is in the microprogram	Complexity is in the compiler
8.	Single register set	Multiple register set
9	Emphasis on hardware	Emphasis on software

4.3 The ARM Design Philosophy

- There are a number of physical features that have driven the ARM processor design:
 - **Low Power** Consumption: Smallest Core;
 - **Limited Memory**: High code density;
 - **Price sensitive**: slow and low-cost memory
 - **Die density**: Simple Hardware Executive Unit
- ARM incorporated **hardware debug technology** within processor: time to market and reduces overall development cost
- 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.

1.2.1 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

1. 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.

2. 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.

3. 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.

4. 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.

5. 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.

4.4 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.

Embedded System Hardware

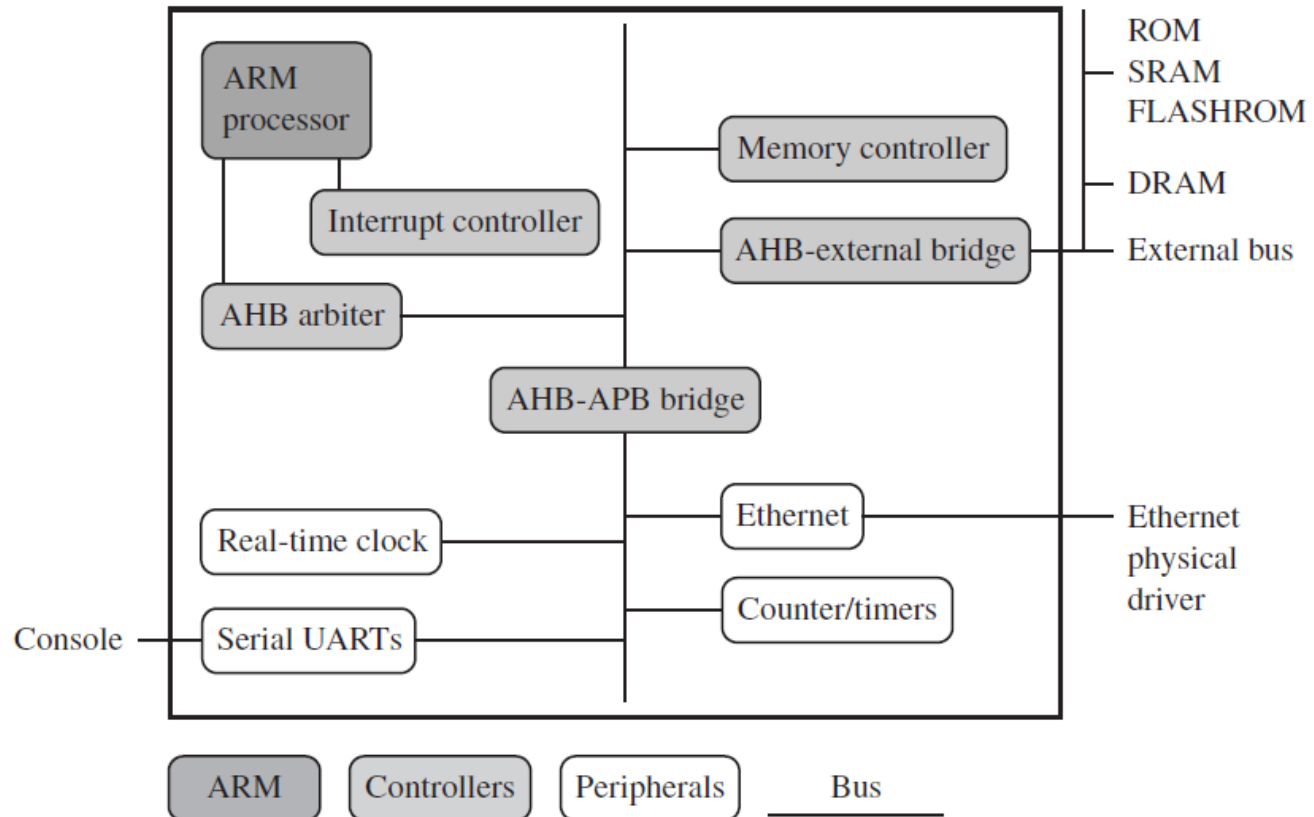


Figure 1.2 An example of an ARM-based embedded device, a microcontroller.

Embedded System Hardware

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

Embedded System Hardware

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.

Embedded System Hardware

ARM Bus Technology

- There are two different classes of devices attached to the bus.
 - The ARM processor core is a **bus master**
 - a logical device capable of initiating a data transfer with another device across the same bus.
 - Peripherals tend to be **bus slaves**
 - logical devices capable only of responding to a transfer request from a bus master device.
- A bus has two architecture levels.
 - The first is a **physical level** that covers the electrical characteristics and bus width (16, 32, or 64 bits).
 - The second level deals with **protocol**—the logical rules that govern the communication between the processor and a peripheral.

Embedded System Hardware

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.

Embedded System Hardware

AMBA Bus Protocol

- 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.



Embedded System Hardware

AMBA Bus Protocol



- Multi-layer AHB systems works with multiple processors, permit operations to occur in parallel and allow for higher throughput rates.

Embedded System Hardware

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.

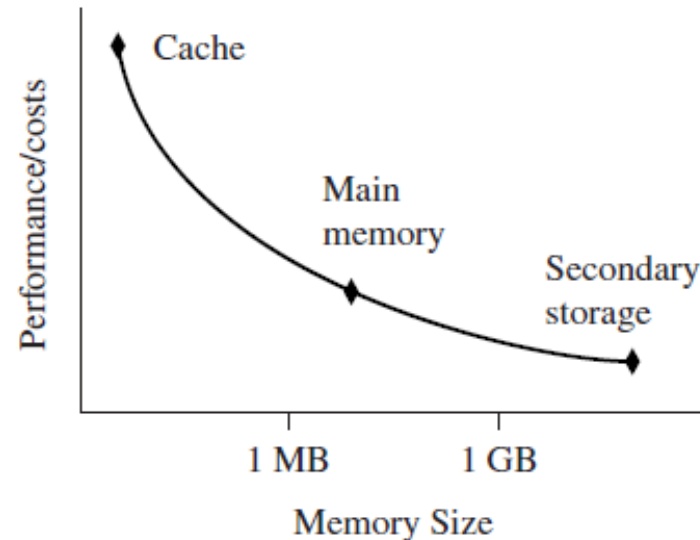


Figure 1.3 Storage trade-offs

Embedded System Hardware

Memory- Hierarchy

- 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.

Embedded System Hardware

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

Embedded System Hardware

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.

Embedded System Hardware 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.



Embedded System Hardware

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.



Embedded System Hardware

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).

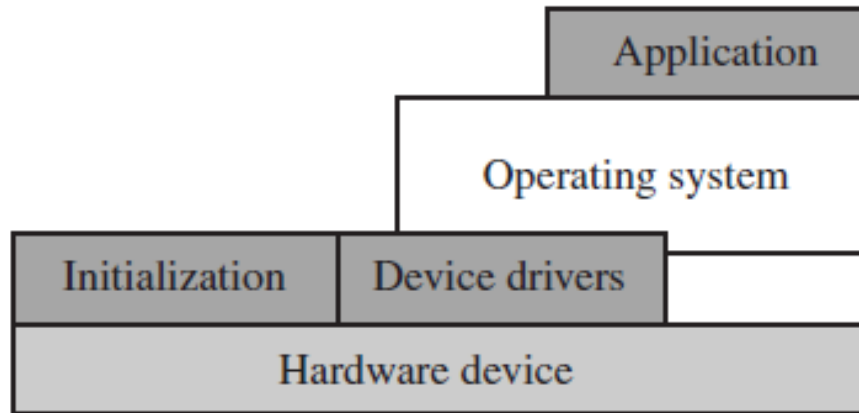
Embedded System Hardware

Peripherals-Interrupt Controllers

- 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.

4.5 Embedded System Software

- An embedded system needs software to drive it.
- Figure shows four typical software components embedded



Software abstraction layers executing on hardware.

Embedded System Software

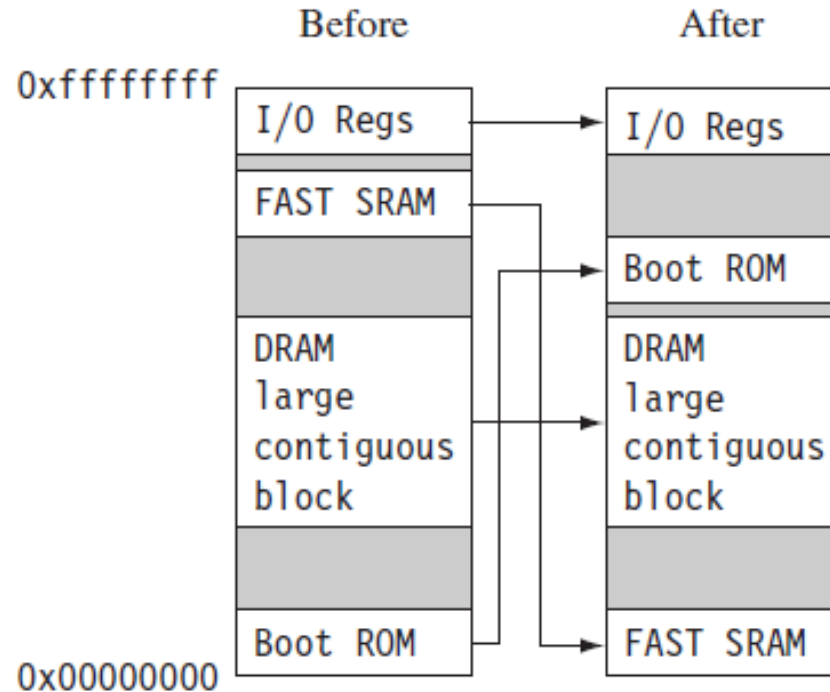
- The initialization code
 - first code executed on the board and is specific to a particular target or group of targets.
 - It sets up the minimum parts of the board before handing control over to the operating system.
- The operating system
 - provides an infrastructure to control applications and manage hardware system resources.
 - In some embedded system operating system does not require.
- Device driver
 - provide a consistent software interface to the peripherals on the hardware device.
- Application
 - 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*.

Embedded System Software Initialization (Boot) Code

- 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.

Embedded System Software 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.



Embedded System Software Initialization (Boot) Code

- 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.

Embedded System Software 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.

Embedded System Software Operating System

- 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.

Embedded System Software 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.

Embedded System Software 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.



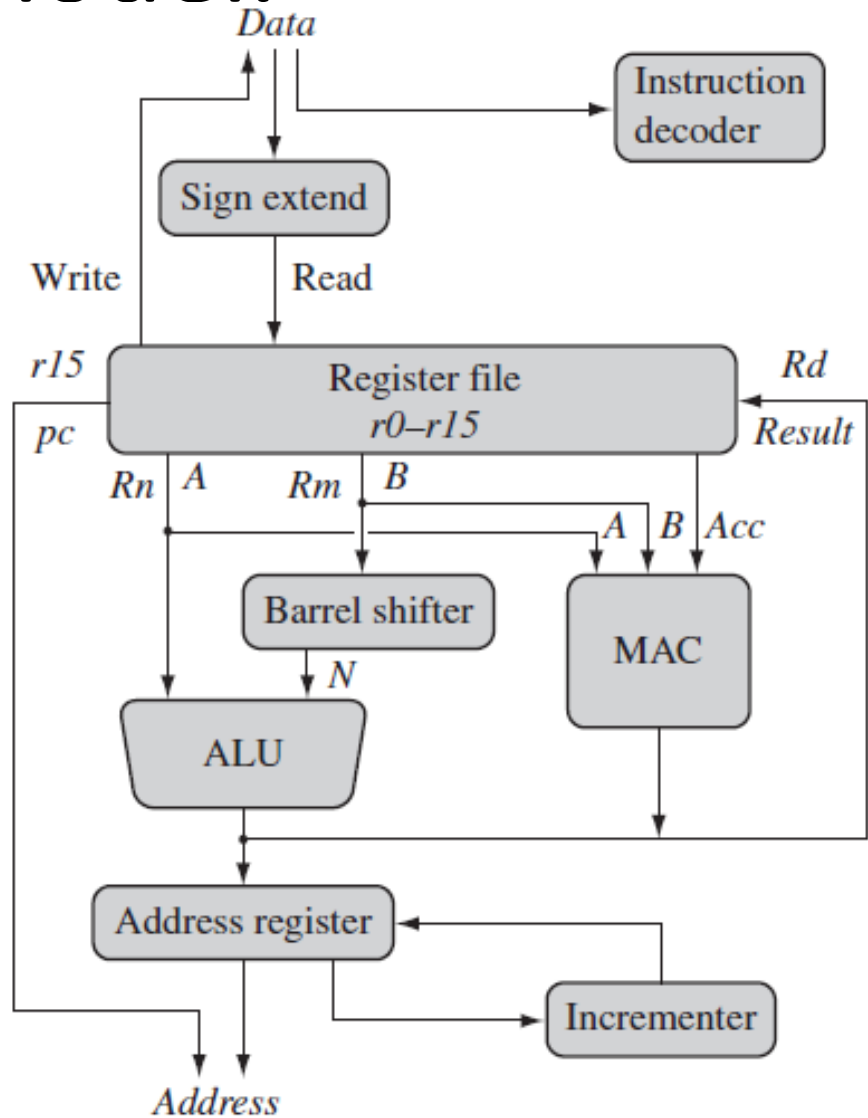
ARM Processor Fundamentals

Introduction

- Overview of the processor core
- How data moves between its different parts.
- Describe the programmer's model
- ARM core architecture

4.6 ARM core dataflow model.

- ARM core dataflow model.



ARM core data Model

- 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.

Introduction

- 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.

Introduction

- 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.

Introduction

- Key components of the processor:
 - The registers,
 - The current program status register (*CPSR*),
 - The pipeline.

4.7 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*.

<i>r0</i>
<i>r1</i>
<i>r2</i>
<i>r3</i>
<i>r4</i>
<i>r5</i>
<i>r6</i>
<i>r7</i>
<i>r8</i>
<i>r9</i>
<i>r10</i>
<i>r11</i>
<i>r12</i>
<i>r13 sp</i>
<i>r14 lr</i>
<i>r15 pc</i>

<i>cpsr</i>
-

Registers

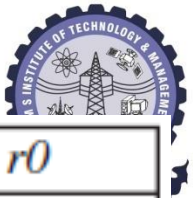
- The shaded registers identify the assigned special-purpose registers:
 - Register *r13* is traditionally used as the stack pointer (*sp*) and stores the head of the stack in the current processor mode.
 - Register *r14* is called the link register (*lr*) and is where the core puts the return address whenever it calls a subroutine.
 - Register *r15* is the program counter (*pc*) and contains the address of the next instruction to be fetched by the processor.

<i>r0</i>
<i>r1</i>
<i>r2</i>
<i>r3</i>
<i>r4</i>
<i>r5</i>
<i>r6</i>
<i>r7</i>
<i>r8</i>
<i>r9</i>
<i>r10</i>
<i>r11</i>
<i>r12</i>
<i>r13 sp</i>
<i>r14 lr</i>
<i>r15 pc</i>

<i>cpsr</i>
-

Registers

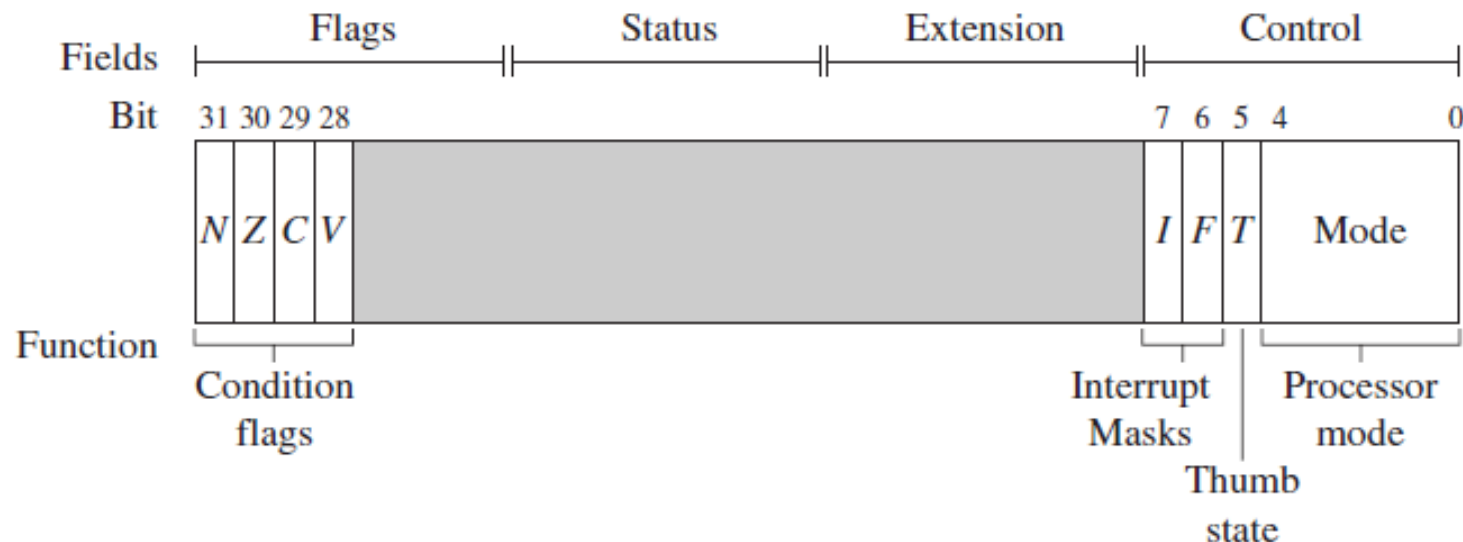
- Registers *r13* and *r14* can also be used as general-purpose registers, depending on processor mode.
 - It is dangerous to use *r13* as a general register when the processor is running any form of operating system because which often assume that *r13* always points to a valid stack frame.
- In ARM 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 depend upon the current mode of the processor.



<i>r0</i>
<i>r1</i>
<i>r2</i>
<i>r3</i>
<i>r4</i>
<i>r5</i>
<i>r6</i>
<i>r7</i>
<i>r8</i>
<i>r9</i>
<i>r10</i>
<i>r11</i>
<i>r12</i>
<i>r13 sp</i>
<i>r14 lr</i>
<i>r15 pc</i>
<i>cpsr</i>
-

4.8 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.
 - Note: the shaded parts are reserved for future expansion.
- The *cpsr* is divided into four fields, each 8 bits wide: flags, status, extension, and control.
- The extension and status fields are reserved for future use.
- The control field contains the processor mode, state, and interrupt mask bits.
- The flags field contains the condition flags.



Current Program Status Register

- 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.

Current Program Status Register

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*

Current Program Status Register

Processor Modes

- *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.

Current Program Status Register

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).
- These are available only when the processor is in a particular mode
- E.g., *Abort* mode has banked registers *r13_abt*, *r14_abt* and *spsr_abt*.

User and system

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 sp
r14 lr
r15 pc

Fast interrupt request

r8_fiq
r9_fiq
r10_fiq
r11_fiq
r12_fiq
r13_fiq
r14_fiq

Interrupt request

r13_irq
r14_irq

Supervisor

r13_svc
r14_svc

Undefined

r13_undef
r14_undef

Abort

r13_abt
r14_abt

cpsr
-

spsr_fiq

spsr_irq

spsr_svc

spsr_undef

spsr_abt

Complete ARM register set.

Current Program Status Register

Banked Registers

- 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.

Current Program Status Register Banked Registers

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*.

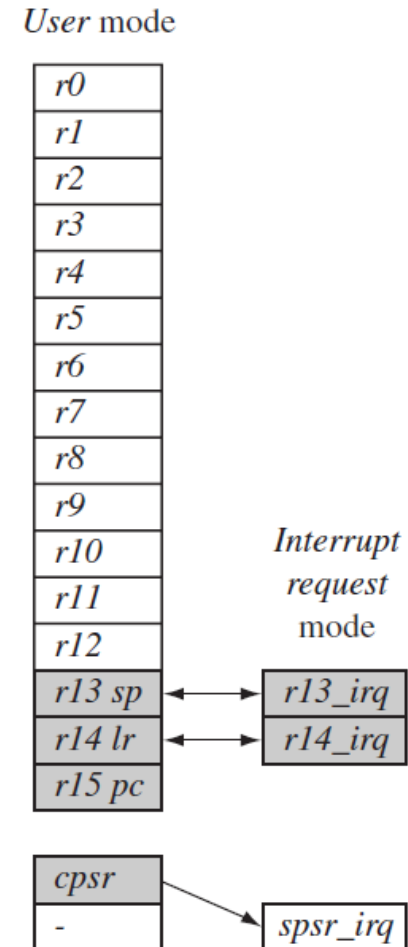
Current Program Status Register Banked Registers

- 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.

Current Program Status Register

Banked Registers

- Figure shows what happens when an interrupt forces a mode change.
- 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*.



Current Program Status Register Banked Registers

- Note that the *spsr* can only be modified and read in a privileged mode.
- 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 saving of the *cpsr* only occurs when an exception or interrupt is raised.

Current Program Status Register Banked Registers

- Lists the various modes and the associated binary patterns
Processor mode.

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

Current Program Status Register

State and Instruction Sets

- The state of the core determines which instruction set is being executed.
- There are three instruction sets:
 - ARM,
 - Thumb,
 - Jazelle.
- The ARM instruction set is only active when the processor is in ARM state.
- You cannot intermingle sequential ARM, Thumb, and Jazelle instructions.

Current Program Status Register State and Instruction Sets

- 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 (<i>cpsr</i> $T = 0$)	Thumb (<i>cpsr</i> $T = 1$)
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

Current Program Status Register State and Instruction Sets

- *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 Jazelle instruction set features.

re

	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.

Current Program Status Register 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.

Current Program Status Register

Condition Flags

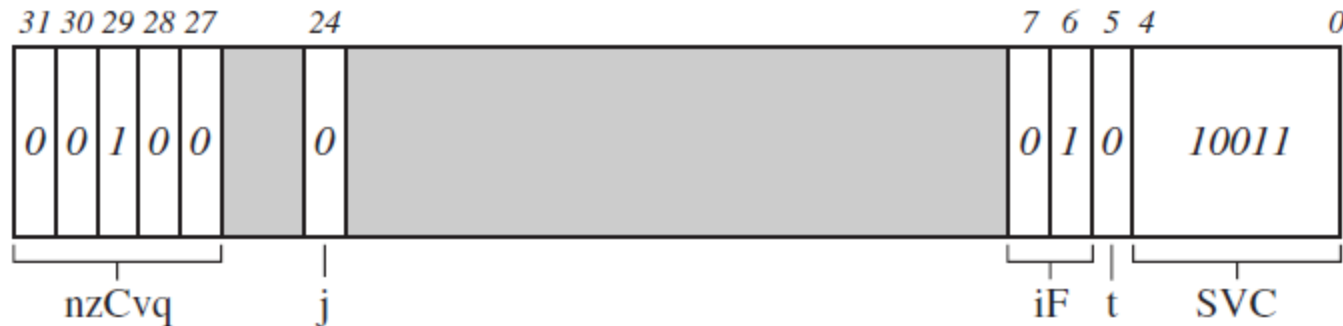
- Condition flags are updated by comparisons and the result of ALU operations that specify the S instruction suffix.
- For example,
- If a SUBS subtract instruction results is zero, then the Z flag in the *cpsr* is set.

Current Program Status Register Condition Flags

- Below shows lists the condition flags and a short description on what causes them to be set.
- These flags are located in the most significant 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

Current Program Status Register Condition Flags



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



Current Program Status Register

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.

Current Program Status Register

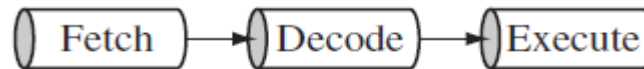
Conditional Execution

Condition mnemonics.

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

4.9 Pipeline

- 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:

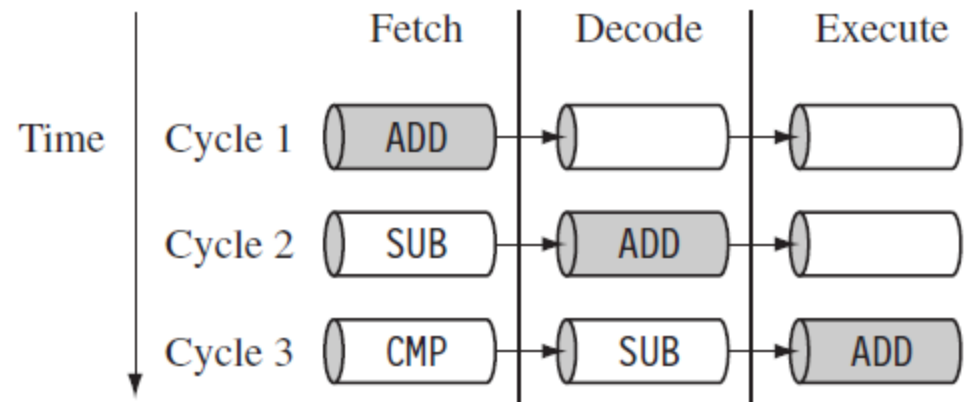


ARM7 Three-stage pipeline.

- *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.

Pipeline

- Figure illustrates the pipeline using a simple example.
- It shows a sequence of three instructions being fetched, decoded, and executed by the processor.
- Each instruction takes a single cycle to complete after the pipeline is filled.



Pipelined instruction sequence.

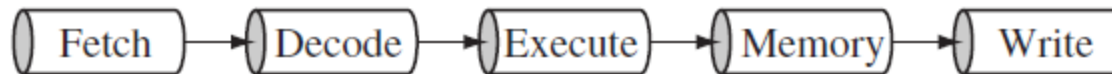
Pipeline

- 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.

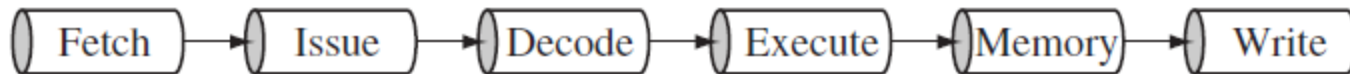
Pipeline

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

Pipeline



ARM9 five-stage pipeline.



ARM10 six-stage pipeline.

Pipeline

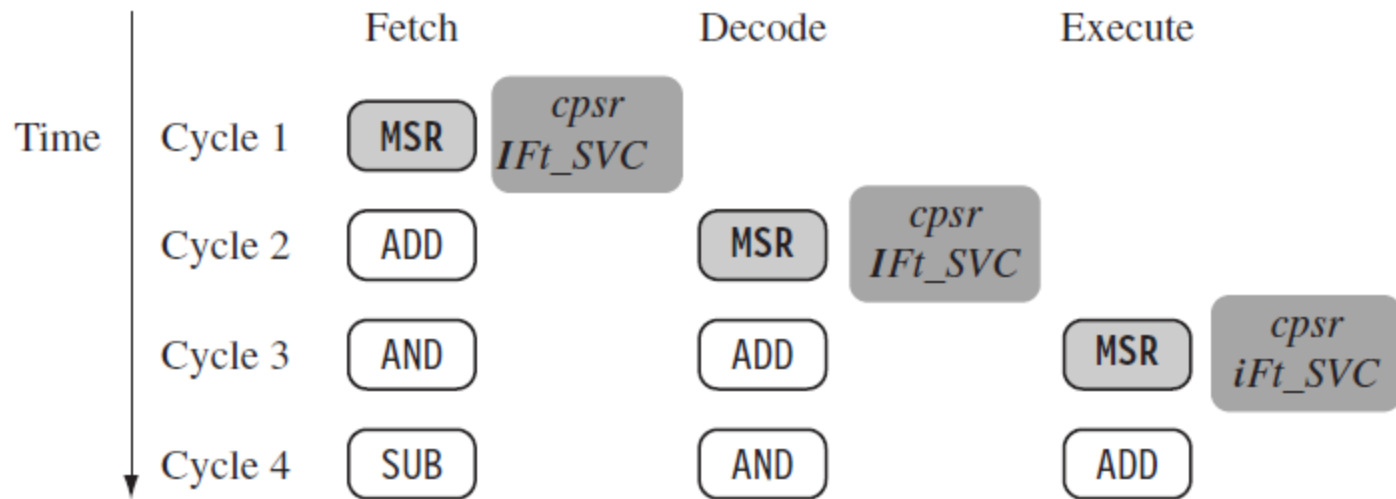
Pipeline Executing Characteristics

- 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.

Pipeline

Pipeline Executing Characteristics

- Figure shows an instruction sequence on an ARM7 pipeline.



ARM instruction sequence.

Pipeline

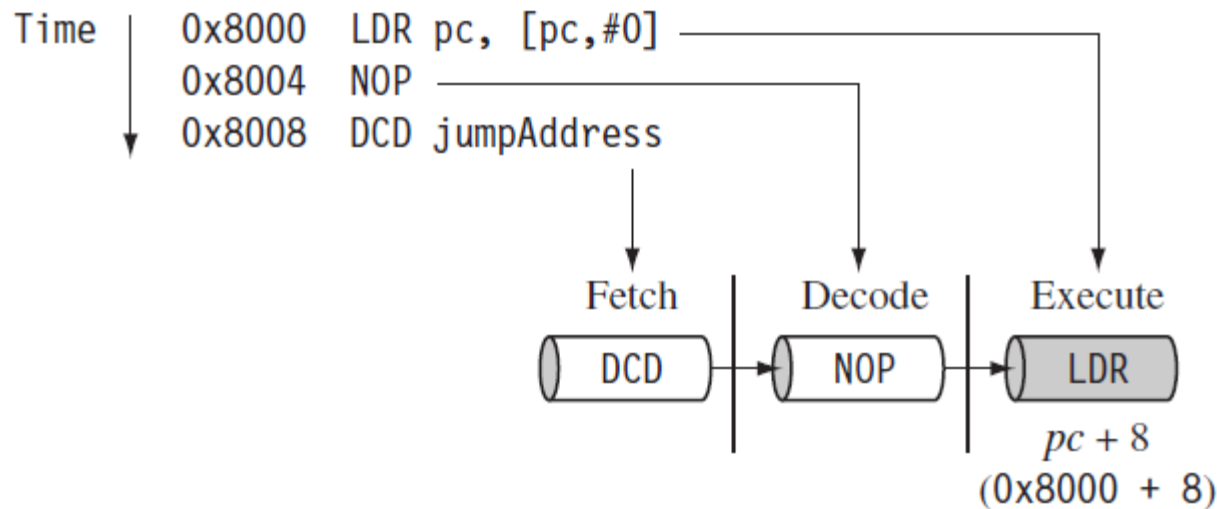
Pipeline Executing Characteristics

- 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.

Pipeline

Pipeline Executing Characteristics

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



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

Pipeline

Pipeline Executing Characteristics

- 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 and is an architectural characteristic across all the pipelines.
- Note when the processor is in Thumb state the *pc* is the instruction address plus 4.

Pipeline

Pipeline Executing Characteristics

- 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.

4.10 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.

Exceptions, Interrupts and the Vector Table

- 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

Exceptions, Interrupts and the Vector Table

- 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*.

4.11 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.

Core Extensions

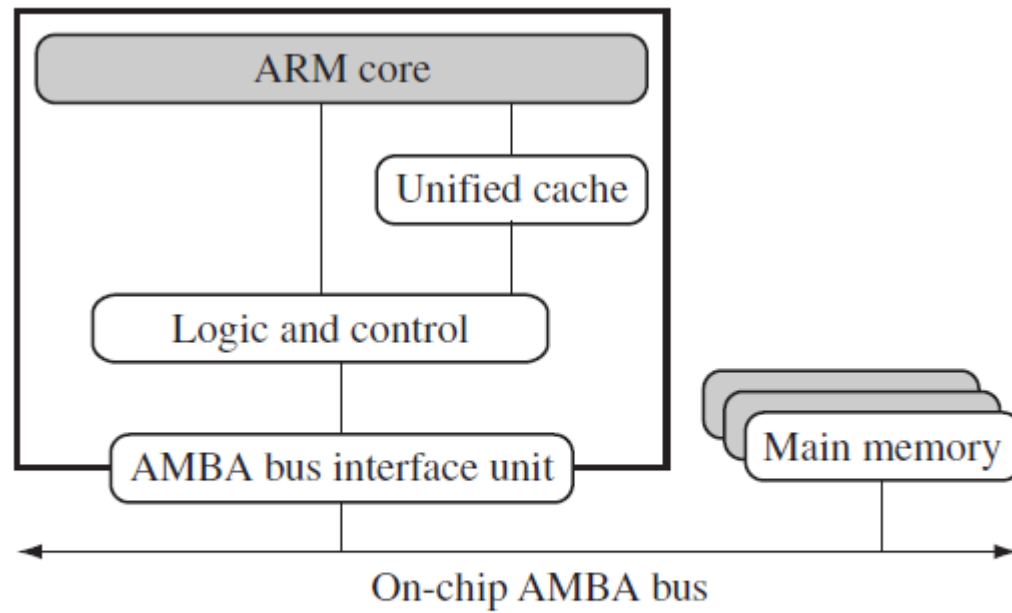
Cache and Tightly Coupled Memory

- 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.
 - The Harvard-style cores

Core Extensions

Cache and Tightly Coupled Memory

- Von Neumann–style cores.
- It combines both data and instruction into a single unified cache.
- The glue logic that connects the memory system to the AMBA bus *logic and control*.

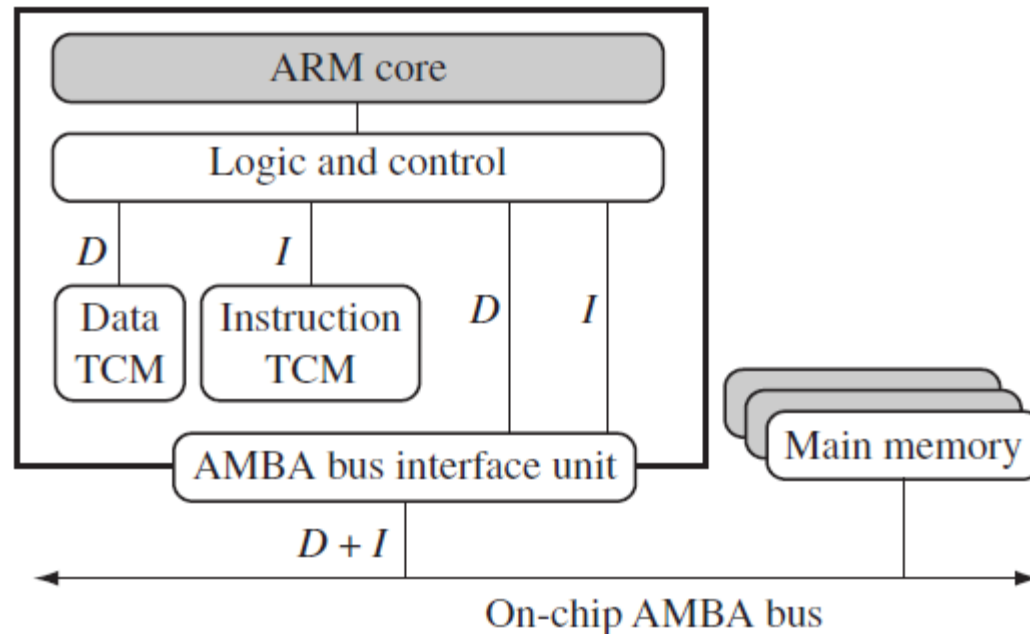


A simplified Von Neumann architecture with cache.

Core Extensions

Cache and Tightly Coupled Memory

- The Harvard-style cores, has separate caches for data and instruction



A simplified Harvard architecture with TCMs.

Core Extensions

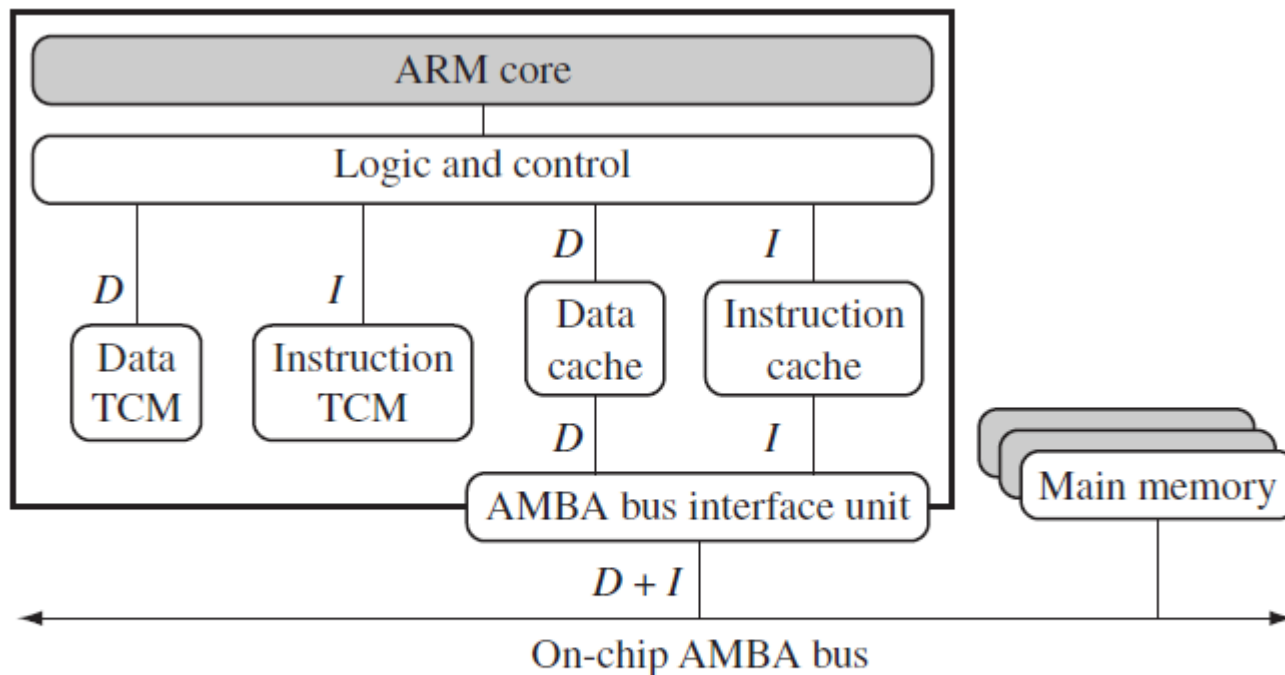
Cache and Tightly Coupled Memory

- 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 behavior.
- TCMs appear as memory in the address map and can be accessed as fast memory.
- An example of a processor with TCMs is Harvard-style cores.

Core Extensions

Cache and Tightly Coupled 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.

Core Extensions

Memory Management

- Embedded systems often use multiple memory devices.
- So a method to help 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.
- ARMcores 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

Core Extensions

Memory Management

- **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,**
 - *MPUs* 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**
 - *MMUs* 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.

Core Extensions

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