

FUTURE VISION BIE

One Stop for All Study Materials
& Lab Programs



Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : <https://hemanthrajhemu.github.io>

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: <https://bit.ly/FVBIESHARE>

**ARM SYSTEM
DEVELOPER'S GUIDE
DESIGNING AND OPTIMIZING
SYSTEM SOFTWARE**

ANDREW N. SLOSS

DOMINIC SYMES

CHRIS WRIGHT

WITH A CONTRIBUTION BY JOHN RAYFIELD



ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier




MORGAN KAUFMANN PUBLISHERS

<https://hemanthrajhemu.github.io>

CONTENTS

	ABOUT THE AUTHORS	ii
	PREFACE	xi
CHAPTER		
1	ARM EMBEDDED SYSTEMS	3
	1.1 The RISC Design Philosophy	4
	1.2 The ARM Design Philosophy	5
	1.3 Embedded System Hardware	6
	1.4 Embedded System Software	12
	1.5 Summary	15
CHAPTER		
2	ARM PROCESSOR FUNDAMENTALS	19
	2.1 Registers	21
	2.2 Current Program Status Register	22
	2.3 Pipeline	29
	2.4 Exceptions, Interrupts, and the Vector Table	33
	2.5 Core Extensions	34
	2.6 Architecture Revisions	37
	2.7 ARM Processor Families	38
	2.8 Summary	43
CHAPTER		
3	INTRODUCTION TO THE ARM INSTRUCTION SET	47
	3.1 Data Processing Instructions	50
	3.2 Branch Instructions	58
	3.3 Load-Store Instructions	60
	3.4 Software Interrupt Instruction	73
	3.5 Program Status Register Instructions	75
	3.6 Loading Constants	78
	3.7 ARMv5E Extensions	79
	3.8 Conditional Execution	82
	3.9 Summary	84

- 
- 1.1 **THE RISC DESIGN PHILOSOPHY**
 - 1.2 **THE ARM DESIGN PHILOSOPHY**
 - 1.2.1 Instruction Set for Embedded Systems
 - 1.3 **EMBEDDED SYSTEM HARDWARE**
 - 1.3.1 ARM Bus Technology
 - 1.3.2 AMBA Bus Protocol
 - 1.3.3 Memory
 - 1.3.4 Peripherals
 - 1.4 **EMBEDDED SYSTEM SOFTWARE**
 - 1.4.1 Initialization (Boot) Code
 - 1.4.2 Operating System
 - 1.4.3 Applications
 - 1.5 **SUMMARY**

CHAPTER 1

ARM EMBEDDED SYSTEMS

The ARM processor core is a key component of many successful 32-bit embedded systems. You probably own one yourself and may not even realize it! ARM cores are widely used in mobile phones, handheld organizers, and a multitude of other everyday portable consumer devices.

ARM's designers have come a long way from the first ARM1 prototype in 1985. Over one billion ARM processors had been shipped worldwide by the end of 2001. The ARM company bases their success on a simple and powerful original design, which continues to improve today through constant technical innovation. In fact, the ARM core is not a single core, but a whole family of designs sharing similar design principles and a common instruction set.

For example, one of ARM's most successful cores is the ARM7TDMI. It provides up to 120 Dhrystone MIPS¹ and is known for its high code density and low power consumption, making it ideal for mobile embedded devices.

In this first chapter we discuss how the RISC (reduced instruction set computer) design philosophy was adapted by ARM to create a flexible embedded processor. We then introduce an example embedded device and discuss the typical hardware and software technologies that surround an ARM processor.

1. Dhrystone MIPS version 2.1 is a small benchmarking program.

1.1 THE RISC DESIGN PHILOSOPHY

The ARM core uses a RISC architecture. RISC is a design philosophy aimed at delivering simple but powerful instructions that execute within a single cycle at a high clock speed. The RISC philosophy concentrates on reducing the complexity of instructions performed by the hardware because it is easier to provide greater flexibility and intelligence in software rather than hardware. As a result, a RISC design places greater demands on the compiler. In contrast, the traditional complex instruction set computer (CISC) relies more on the hardware for instruction functionality, and consequently the CISC instructions are more complicated. Figure 1.1 illustrates these major differences.

The RISC philosophy is implemented with four major design rules:

1. *Instructions*—RISC processors have a reduced number of instruction classes. These classes provide simple operations that can each execute in a single cycle. The compiler or programmer synthesizes complicated operations (for example, a divide operation) by combining several simple instructions. Each instruction is a fixed length to allow the pipeline to fetch future instructions before decoding the current instruction. In contrast, in CISC processors the instructions are often of variable size and take many cycles to execute.
2. *Pipelines*—The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines. Ideally the pipeline advances by one step on each cycle for maximum throughput. Instructions can be decoded in one pipeline stage. There is no need for an instruction to be executed by a miniprogram called microcode as on CISC processors.
3. *Registers*—RISC machines have a large general-purpose register set. Any register can contain either data or an address. Registers act as the fast local memory store for all data

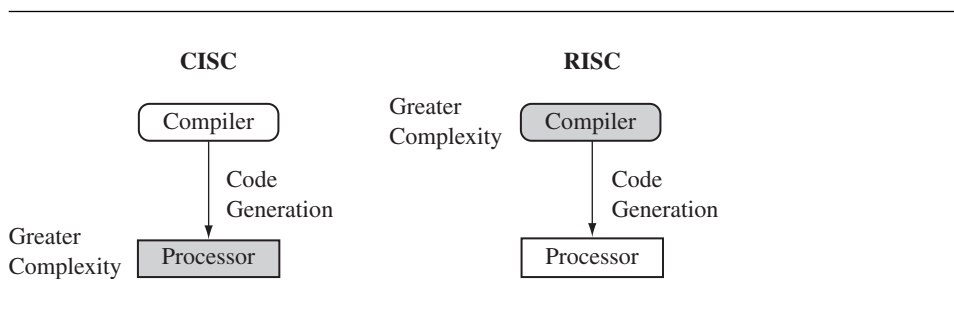


Figure 1.1 CISC vs. RISC. CISC emphasizes hardware complexity. RISC emphasizes compiler complexity.

processing operations. In contrast, CISC processors have dedicated registers for specific purposes.

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

These design rules allow a RISC processor to be simpler, and thus the core can operate at higher clock frequencies. In contrast, traditional CISC processors are more complex and operate at lower clock frequencies. Over the course of two decades, however, the distinction between RISC and CISC has blurred as CISC processors have implemented more RISC concepts.

1.2 THE ARM DESIGN PHILOSOPHY

There are a number of physical features that have driven the ARM processor design. First, portable embedded systems require some form of battery power. The ARM processor has been specifically designed to be small to reduce power consumption and extend battery operation—essential for applications such as mobile phones and personal digital assistants (PDAs).

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

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

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

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

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

1.2.1 INSTRUCTION SET FOR EMBEDDED SYSTEMS

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

- *Variable cycle execution for certain instructions*—Not every ARM instruction executes in a single cycle. For example, load-store-multiple instructions vary in the number of execution cycles depending upon the number of registers being transferred. The transfer can occur on sequential memory addresses, which increases performance since sequential memory accesses are often faster than random accesses. Code density is also improved since multiple register transfers are common operations at the start and end of functions.
- *Inline barrel shifter leading to more complex instructions*—The inline barrel shifter is a hardware component that preprocesses one of the input registers before it is used by an instruction. This expands the capability of many instructions to improve core performance and code density. We explain this feature in more detail in Chapters 2, 3, and 4.
- *Thumb 16-bit instruction set*—ARM enhanced the processor core by adding a second 16-bit instruction set called Thumb that permits the ARM core to execute either 16- or 32-bit instructions. The 16-bit instructions improve code density by about 30% over 32-bit fixed-length instructions.
- *Conditional execution*—An instruction is only executed when a specific condition has been satisfied. This feature improves performance and code density by reducing branch instructions.
- *Enhanced instructions*—The enhanced digital signal processor (DSP) instructions were added to the standard ARM instruction set to support fast 16×16 -bit multiplier operations and saturation. These instructions allow a faster-performing ARM processor in some cases to replace the traditional combinations of a processor plus a DSP.

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

1.3 EMBEDDED SYSTEM HARDWARE

Embedded systems can control many different devices, from small sensors found on a production line, to the real-time control systems used on a NASA space probe. All these devices use a combination of software and hardware components. Each component is chosen for efficiency and, if applicable, is designed for future extension and expansion.

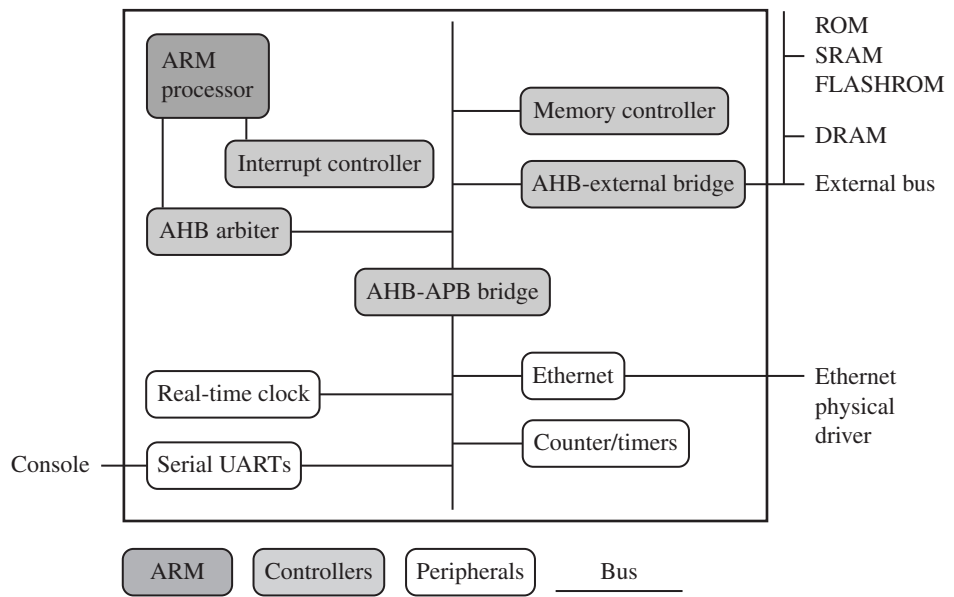


Figure 1.2 An example of an ARM-based embedded device, a microcontroller.

Figure 1.2 shows a typical embedded device based on an ARM core. Each box represents a feature or function. The lines connecting the boxes are the buses carrying data. We can separate the device 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.

1.3.1 ARM BUS TECHNOLOGY

Embedded systems use different bus technologies than those designed for x86 PCs. The most common PC bus technology, the Peripheral Component Interconnect (PCI) bus, connects such devices as video cards and hard disk controllers to the x86 processor bus. This type of technology is external or off-chip (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 contrast, embedded devices use an on-chip bus that is internal to the chip and that allows different peripheral devices to be interconnected with an ARM core.

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

A bus has two architecture levels. The first is a physical level that covers the electrical characteristics and bus width (16, 32, or 64 bits). The second level deals with *protocol*—the logical rules that govern the communication between the processor and a peripheral.

ARM is primarily a design company. It seldom implements the electrical characteristics of the bus, but it routinely specifies the bus protocol.

1.3.2 AMBA BUS PROTOCOL

The Advanced Microcontroller Bus Architecture (AMBA) was introduced in 1996 and has been widely adopted as the on-chip bus architecture used for ARM processors. The first AMBA buses introduced were the ARM System Bus (ASB) and the ARM Peripheral Bus (APB). Later ARM introduced another bus design, called the ARM High Performance Bus (AHB). Using AMBA, peripheral designers can reuse the same design on multiple projects. Because there are a large number of peripherals developed with an AMBA interface, hardware designers have a wide choice of tested and proven peripherals for use in a device. 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 and to be the first ARM bus to support 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. This bus was developed for designs that do not require the full features of the standard AHB bus.

AHB and Multi-layer AHB support the same protocol for master and slave but have different interconnects. The new interconnects in Multi-layer AHB are good for systems with multiple processors. They permit operations to occur in parallel and allow for higher throughput rates.

The example device shown in Figure 1.2 has three buses: an AHB bus for the high-performance peripherals, an APB bus for the slower peripherals, and a third bus for external peripherals, proprietary to this device. This external bus requires a specialized bridge to connect with the AHB bus.

1.3.3 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. If memory has to run twice as fast to maintain a desired bandwidth, then the memory power requirement may be higher.

1.3.3.1 Hierarchy

All computer systems have memory arranged in some form of hierarchy. Figure 1.2 shows a device that supports external off-chip memory. Internal to the processor there is an option of a cache (not shown in Figure 1.2) to improve memory performance.

Figure 1.3 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.

The cache is placed between main memory and the core. It is used to speed up data transfer between the processor and main memory. A cache provides an overall increase in performance but with a loss of predictable execution time. Although the cache increases the

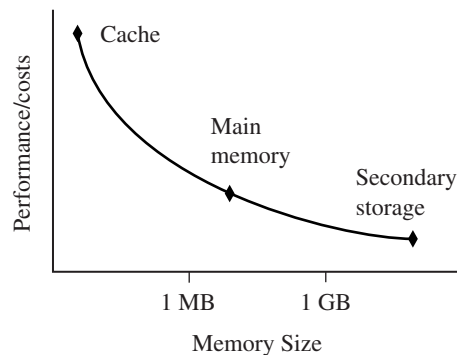


Figure 1.3 Storage trade-offs.

general performance of the system, it does not help real-time system response. Note that many small embedded systems do not require the performance benefits of a cache.

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. Hard disk drives and CD-ROM drives are examples of secondary storage. These days secondary storage may vary from 600 MB to 60 GB.

1.3.3.2 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.

If you have an uncached system using 32-bit ARM instructions and 16-bit-wide memory chips, then the processor will have to make two memory fetches per instruction. Each fetch requires two 16-bit loads. This obviously has the effect of reducing system performance, but the benefit is that 16-bit memory is less expensive.

In contrast, if the core executes 16-bit Thumb instructions, it will achieve better performance with a 16-bit memory. The higher performance is a result of the core making only a single fetch to memory to load an instruction. Hence, using Thumb instructions with 16-bit-wide memory devices provides both improved performance and reduced cost.

Table 1.1 summarizes theoretical cycle times on an ARM processor using different memory width devices.

1.3.3.3 Types

There are many different types of memory. In this section we describe some of the more popular memory devices found in ARM-based embedded systems.

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.

Table 1.1 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

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. Flash ROM has become the most popular of the read-only memory types and is currently being used as an alternative for mass or secondary storage.

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.

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. Because of its higher cost, it is used mostly for smaller high-speed tasks, such as fast memory and caches.

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. The old-style DRAM is asynchronous, so does not burst as efficiently as SDRAM.

1.3.4 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.

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.

1.3.4.1 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.

1.3.4.2 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 (VIC).

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 VIC is more powerful than the standard interrupt controller because it prioritizes interrupts and simplifies the determination of which device caused the interrupt. After associating a priority and a handler address with each interrupt, the VIC only asserts an interrupt signal to the core if the priority of a new interrupt is higher than the currently executing interrupt handler. Depending on its type, the VIC will either call the standard interrupt exception handler, which can load the address of the handler for the device from the VIC, or cause the core to jump to the handler for the device directly.

1.4 EMBEDDED SYSTEM SOFTWARE

An embedded system needs software to drive it. Figure 1.4 shows four typical software components required to control an embedded device. Each software component in the stack uses a higher level of abstraction to separate the code from the hardware device.

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

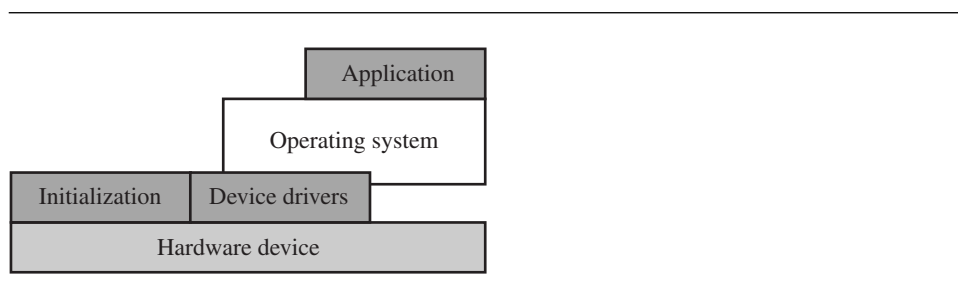


Figure 1.4 Software abstraction layers executing on hardware.

The operating system provides an infrastructure to control applications and manage hardware system resources. Many embedded systems do not require a full operating system but merely a simple task scheduler that is either event or poll driven.

The device drivers are the third component shown in Figure 1.4. They provide a consistent software interface to the peripherals on the hardware device.

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

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

1.4.1 INITIALIZATION (BOOT) CODE

Initialization code (or boot code) takes the processor from the reset state to a state where the operating system can run. It usually configures the memory controller and processor caches and initializes some devices. In a simple system the operating system might be replaced by a simple scheduler or debug monitor.

The initialization code handles a number of administrative tasks prior to handing control over to an operating system image. We can group these different tasks into three phases: initial hardware configuration, diagnostics, and booting.

Initial hardware configuration involves setting up the target platform so it can boot an image. Although the target platform itself comes up in a standard configuration, 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, as shown in Example 1.1.

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. It also tracks down standard system-related issues. This type of testing is important for manufacturing since it occurs after the software product is complete. The primary purpose of diagnostic code is fault identification and isolation.

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.

Sometimes, to reduce the image size, an image is compressed. The image is then decompressed either when it is loaded or when control is handed over to it.

EXAMPLE 1.1 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.

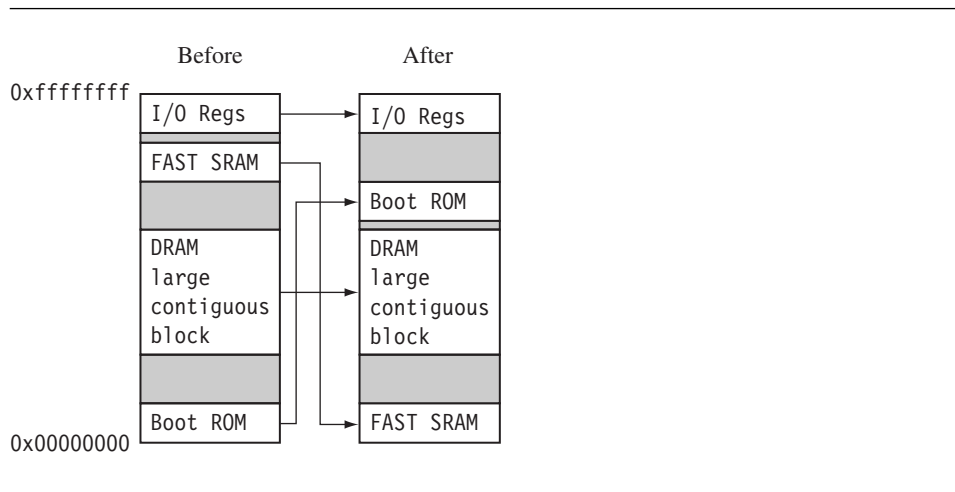


Figure 1.5 Memory remapping.

Figure 1.5 shows memory before and after reorganization. It is common for ARM-based embedded systems to provide for memory remapping because it allows the system to start the initialization code from ROM at power-up. 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. We will discuss the vector table in more detail in Section 2.4. ■

1.4.2 OPERATING SYSTEM

The initialization process prepares the hardware for an operating system to take control. An operating system organizes the system resources: the peripherals, memory, and processing time. With an operating system controlling these resources, they can be efficiently used by different applications running within the operating system environment.

ARM processors support over 50 operating systems. We can divide operating systems into two main categories: real-time operating systems (RTOSs) and platform operating systems.

RTOSs provide guaranteed response times to events. Different operating systems have different amounts of control over the system response time. A hard real-time application requires a guaranteed response to work at all. In contrast, a soft real-time application requires a good response time, but the performance degrades more gracefully if the response time overruns. Systems running an 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. The Linux operating system is a typical example of a platform 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. ARM has developed a set of processor cores that specifically target each category.

1.4.3 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. Within each segment ARM processors can be found in multiple applications.

For example, the ARM processor is found in networking applications like home gateways, DSL modems for high-speed Internet communication, and 802.11 wireless communication. The mobile device segment is the largest application area for ARM processors because of mobile phones. ARM processors are also found in mass storage devices such as hard drives and imaging products such as inkjet printers—applications that are cost sensitive and high volume.

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.

1.5 SUMMARY

Pure RISC is aimed at high performance, but ARM uses a modified RISC design philosophy that also targets good code density and low power consumption. An embedded system consists of a processor core surrounded by caches, memory, and peripherals. The system is controlled by operating system software that manages application tasks.

The key points in a RISC design philosophy are to improve performance by reducing the complexity of instructions, to speed up instruction processing by using a pipeline, to provide a large register set to store data near the core, and to use a load-store architecture.

The ARM design philosophy also incorporates some non-RISC ideas:

- It allows variable cycle execution on certain instructions to save power, area, and code size.
- It adds a barrel shifter to expand the capability of certain instructions.
- It uses the Thumb 16-bit instruction set to improve code density.

- It improves code density and performance by conditionally executing instructions.
- It includes enhanced instructions to perform digital signal processing type functions.

An embedded system includes the following hardware components: *ARM processors* are found embedded in chips. Programmers access *peripherals* through memory-mapped registers. There is a special type of peripheral called a *controller*, which embedded systems use to configure higher-level functions such as memory and interrupts. The AMBA on-chip *bus* is used to connect the processor and peripherals together.

An embedded system also includes the following software components: *Initialization code* configures the hardware to a known state. Once configured, *operating systems* can be loaded and executed. Operating systems provide a common programming environment for the use of hardware resources and infrastructure. *Device drivers* provide a standard interface to peripherals. An *application* performs the task-specific duties of an embedded system.

This Page Intentionally Left Blank

- 2.1 **REGISTERS**
- 2.2 **CURRENT PROGRAM STATUS REGISTER**
 - 2.2.1 Processor Modes
 - 2.2.2 Banked Registers
 - 2.2.3 State and Instruction Sets
 - 2.2.4 Interrupt Masks
 - 2.2.5 Condition Flags
 - 2.2.6 Conditional Execution
- 2.3 **PIPELINE**
 - 2.3.1 Pipeline Executing Characteristics
- 2.4 **EXCEPTIONS, INTERRUPTS, AND THE VECTOR TABLE**
- 2.5 **CORE EXTENSIONS**
 - 2.5.1 Cache and Tightly Coupled Memory
 - 2.5.2 Memory Management
 - 2.5.3 Coprocessors
- 2.6 **ARCHITECTURE REVISIONS**
 - 2.6.1 Nomenclature
 - 2.6.2 Architecture Evolution
- 2.7 **ARM PROCESSOR FAMILIES**
 - 2.7.1 ARM7 Family
 - 2.7.2 ARM9 Family
 - 2.7.3 ARM10 Family
 - 2.7.4 ARM11 Family
 - 2.7.5 Specialized Processors
- 2.8 **SUMMARY**

CHAPTER 2

ARM PROCESSOR FUNDAMENTALS

Chapter 1 covered embedded systems with an ARM processor. In this chapter we will focus on the actual processor itself. First, we will provide an overview of the processor core and describe how data moves between its different parts. We will describe the programmer's model from a software developer's view of the ARM processor, which will show you the functions of the processor core and how different parts interact. We will also take a look at the core extensions that form an ARM processor. Core extensions speed up and organize main memory as well as extend the instruction set. We will then cover the revisions to the ARM core architecture by describing the ARM core naming conventions used to identify them and the chronological changes to the ARM instruction set architecture. The final section introduces the architecture implementations by subdividing them into specific ARM processor core families.

A programmer can think of an ARM core as functional units connected by data buses, as shown in Figure 2.1, where, the arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area. The figure shows not only the flow of data but also the abstract components that make up an ARM core.

Data enters the processor core through the *Data* bus. The data may be an instruction to execute or a data item. Figure 2.1 shows a Von Neumann implementation of the ARM—data items and instructions share the same bus. In contrast, Harvard implementations of the ARM use two different buses.

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 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, and conversely the store

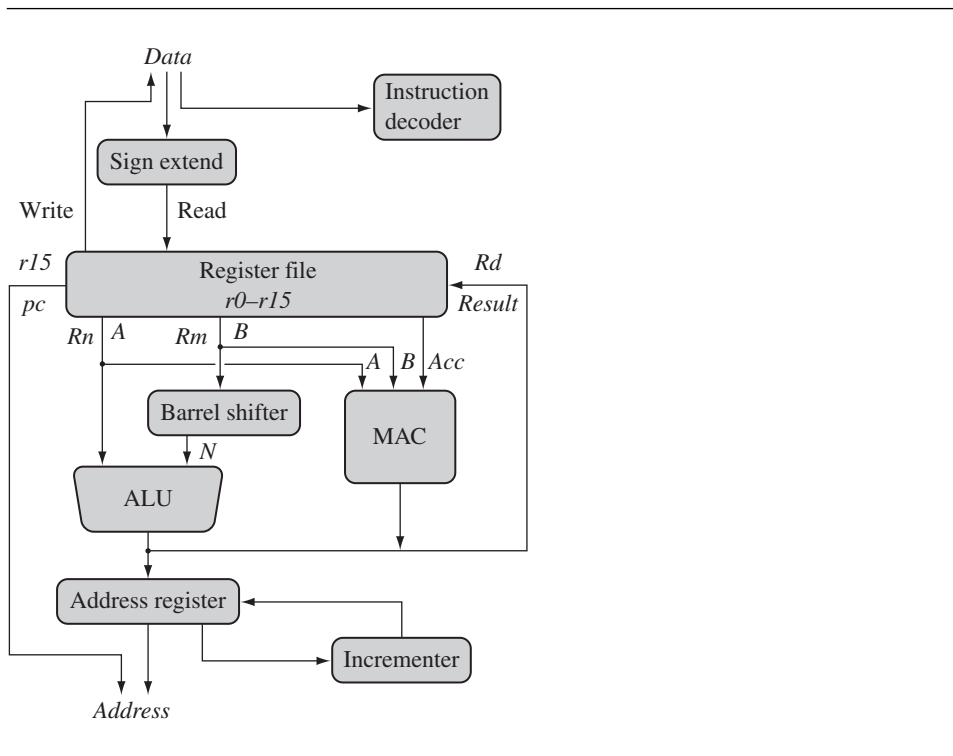


Figure 2.1 ARM core dataflow model.

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

Data items are placed in the *register file*—a storage bank made up of 32-bit registers. Since the ARM core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values. The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.

ARM instructions typically have two source registers, *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.

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.

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.

Now that you have an overview of the processor core we'll take a more detailed look at some of the key components of the processor: the registers, the current program status register (*cpsr*), and the pipeline.

2.1 REGISTERS

General-purpose registers hold either data or an address. They are identified with the letter r prefixed to the register number. For example, register 4 is given the label $r4$. Figure 2.2 shows the active registers available in *user* mode—a protected mode normally

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

Figure 2.2 Registers available in *user* mode.

used when executing applications. The processor can operate in seven different modes, which we will introduce shortly. All the registers shown are 32 bits in size.

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

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

In Figure 2.2, the shaded registers identify the assigned special-purpose registers:

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

Depending upon the context, registers *r13* and *r14* can also be used as general-purpose registers, which can be particularly useful since these registers are banked during a processor mode change. However, it is dangerous to use *r13* as a general register when the processor is running any form of operating system because operating systems 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. However, there are instructions that treat *r14* and *r15* in a special way.

In addition to the 16 data registers, there are two program status registers: *cpsr* and *spsr* (the current and saved program status registers, respectively).

The register file contains all the registers available to a programmer. Which registers are visible to the programmer depend upon the current mode of the processor.

2.2 CURRENT PROGRAM STATUS REGISTER

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

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

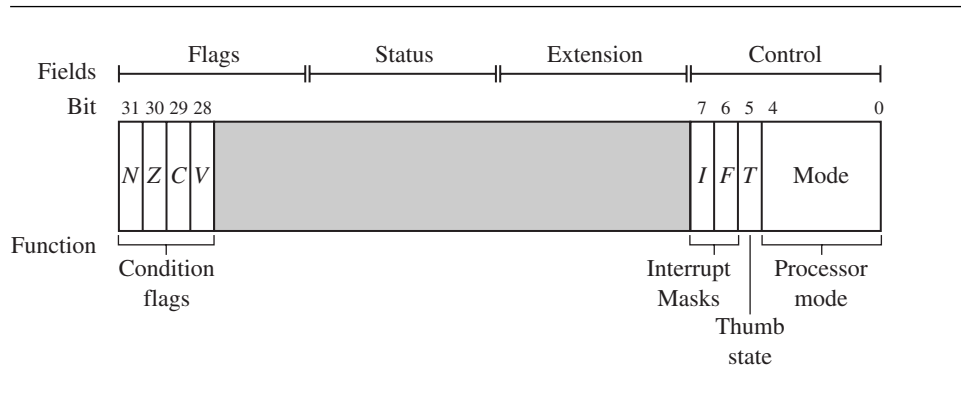


Figure 2.3 A generic program status register (*psr*).

8-bit instructions. We will discuss Jazelle more in Section 2.2.3. It is highly probable that future designs will assign extra bits for the monitoring and control of new features.

For a full description of the *cpsr*, refer to Appendix B.

2.2.1 PROCESSOR MODES

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

There are seven processor modes in total: six privileged modes (*abort*, *fast interrupt request*, *interrupt request*, *supervisor*, *system*, and *undefined*) and one nonprivileged mode (*user*).

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

2.2.2 BANKED REGISTERS

Figure 2.4 shows all 37 registers in the register file. Of those, 20 registers are hidden from a program at different times. These registers are called *banked registers* and are identified by the shading in the diagram. They are available only when the processor is in a particular

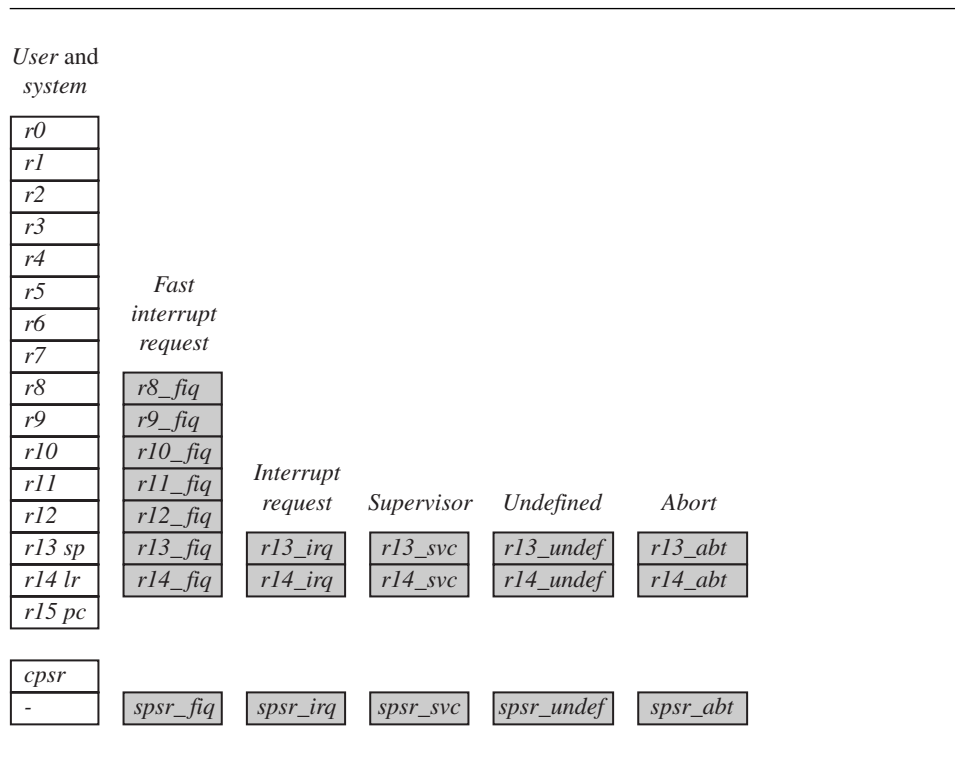


Figure 2.4 Complete ARM register set.

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.

For example, when the processor is in the *interrupt request* mode, the instructions you execute still access registers named *r13* and *r14*. However, these registers are the banked registers *r13_irq* and *r14_irq*. The *user* mode registers *r13_usr* and *r14_usr* are not affected by the instruction referencing these registers. A program still has normal access to the other registers *r0* to *r12*.

The processor mode can be changed by a program that writes directly to the *cpsr* (the processor core has to be in privileged mode) or by hardware when the core responds to

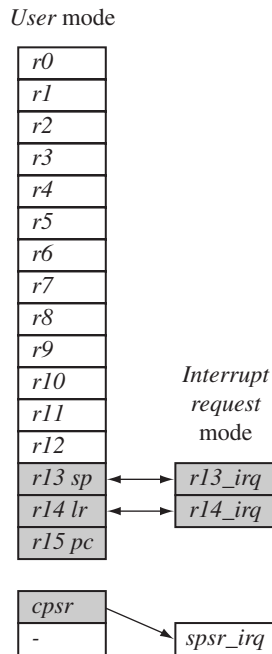


Figure 2.5 Changing mode on an exception.

an exception or interrupt. The following exceptions and interrupts cause a mode change: *reset*, *interrupt request*, *fast interrupt request*, *software interrupt*, *data abort*, *prefetch abort*, and *undefined instruction*. Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location.

Figure 2.5 illustrates what happens when an interrupt forces a mode change. 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*.

Figure 2.5 also shows a new register appearing in *interrupt request mode*: the saved program status register (*spsr*), which stores the previous mode's *cpsr*. You can see in the diagram 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*.

Table 2.1 Processor mode.

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

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.

Figure 2.3 shows that the current active processor mode occupies the five least significant bits of the *cpsr*. 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.

Table 2.1 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*.

2.2.3 STATE AND INSTRUCTION SETS

The state of the core determines which instruction set is being executed. There are three instruction sets: ARM, Thumb, and Jazelle. The ARM instruction set is only active when the processor is in ARM state. Similarly the Thumb instruction set is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16-bit instructions. You cannot intermingle sequential ARM, Thumb, and Jazelle instructions.

The Jazelle *J* and Thumb *T* bits in the *cpsr* reflect the state of the processor. When both *J* and *T* bits are 0, the processor is in ARM state and executes ARM instructions. 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. Table 2.2 compares the ARM and Thumb instruction set features.

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

To execute Java bytecodes, you require the Jazelle technology plus a specially modified version of the Java virtual machine. It is important to note that the hardware portion of Jazelle only supports a subset of the Java bytecodes; the rest are emulated in software.

Table 2.2 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 + <i>pc</i>	8 general-purpose registers + 7 high registers + <i>pc</i>

^a See Section 2.2.6.

Table 2.3 Jazelle instruction set features.

	Jazelle (<i>cpsr</i> $T = 0$, $J = 1$)
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.

The Jazelle instruction set is a closed instruction set and is not openly available. Table 2.3 gives the Jazelle instruction set features.

2.2.4 INTERRUPT MASKS

Interrupt masks are used to stop specific interrupt requests from interrupting the processor. There are two interrupt request levels available on the ARM processor core—*interrupt request* (IRQ) and *fast interrupt request* (FIQ).

The *cpsr* has two interrupt mask bits, 7 and 6 (or *I* and *F*), which control the masking of IRQ and FIQ, respectively. The *I* bit masks IRQ when set to binary 1, and similarly the *F* bit masks FIQ when set to binary 1.

2.2.5 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. This particular subtract instruction specifically updates the *cpsr*.

Table 2.4 Condition flags.

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

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

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. Table 2.4 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.

Figure 2.6 shows a typical value for the *cpsr* with both DSP extensions and Jazelle. In this book we use a notation that presents the *cpsr* data in a more human readable form. When a bit is a binary 1 we use a capital letter; when a bit is a binary 0, we use a lowercase letter. For the condition flags a capital letter shows that the flag has been set. For interrupts a capital letter shows that an interrupt is disabled.

In the *cpsr* example shown in Figure 2.6, 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* or Thumb *t* bits are set. The IRQ interrupts are enabled, and FIQ interrupts are disabled. Finally, you

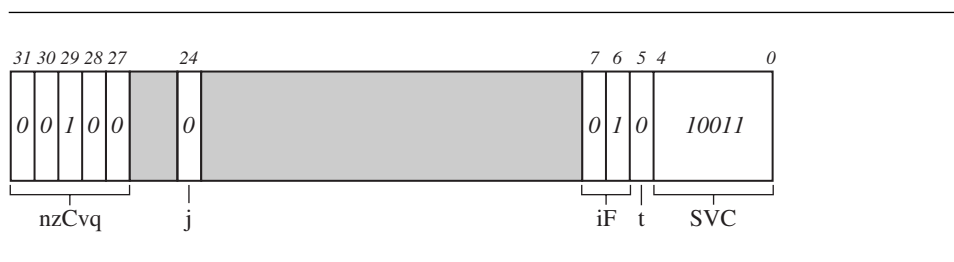
Figure 2.6 Example: *cpsr* = *nzCvqjiFt_SVC*.

Table 2.5 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

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

2.2.6 CONDITIONAL EXECUTION

Conditional execution controls whether or not the core will execute an instruction. Most instructions have a condition attribute that determines if the core will execute it based on the setting of the condition flags. 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 postfixed to the instruction mnemonic, which is encoded into the instruction. Table 2.5 lists the conditional execution code mnemonics. When a condition mnemonic is not present, the default behavior is to set it to always (*AL*) execute.

2.3 PIPELINE

A pipeline is the mechanism a RISC processor uses to execute instructions. Using a pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed. One way to view the pipeline is to think of it as an automobile assembly line, with each stage carrying out a particular task to manufacture the vehicle.



Figure 2.7 ARM7 Three-stage pipeline.

Figure 2.7 shows a three-stage pipeline:

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

Figure 2.8 illustrates the pipeline using a simple example. It shows a sequence of three instructions being fetched, decoded, and executed by the processor. Each instruction takes a single cycle to complete after the pipeline is filled.

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

As the pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency. This in turn increases the performance. The system *latency* also increases because it takes more cycles to fill the pipeline before the core can execute an instruction. The increased pipeline length also means there can be data dependency between certain stages. You can write code to reduce this dependency by using *instruction scheduling* (for more information on instruction scheduling take a look at Chapter 6).

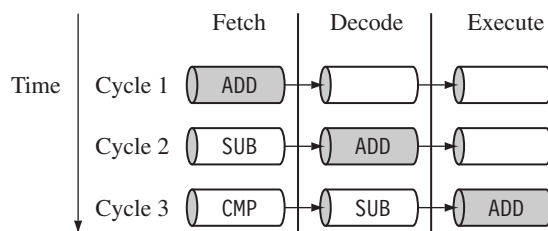


Figure 2.8 Pipelined instruction sequence.

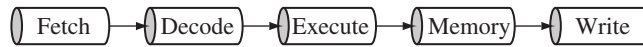


Figure 2.9 ARM9 five-stage pipeline.

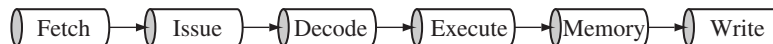


Figure 2.10 ARM10 six-stage pipeline.

The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages, as shown in Figure 2.9. The ARM9 adds a memory and writeback stage, which allows the ARM9 to process on average 1.1 Dhrystone MIPS per MHz—an increase in instruction throughput by around 13% compared with an ARM7. The maximum core frequency attainable using an ARM9 is also higher.

The ARM10 increases the pipeline length still further by adding a sixth stage, as shown in Figure 2.10. The ARM10 can process on average 1.3 Dhrystone MIPS per MHz, about 34% more throughput than an ARM7 processor core, but again at a higher latency cost.

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

2.3.1 PIPELINE EXECUTING CHARACTERISTICS

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

Figure 2.11 shows an instruction sequence on an ARM7 pipeline. 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.

Figure 2.12 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. This is important when the *pc* is used for calculating a relative offset and is an

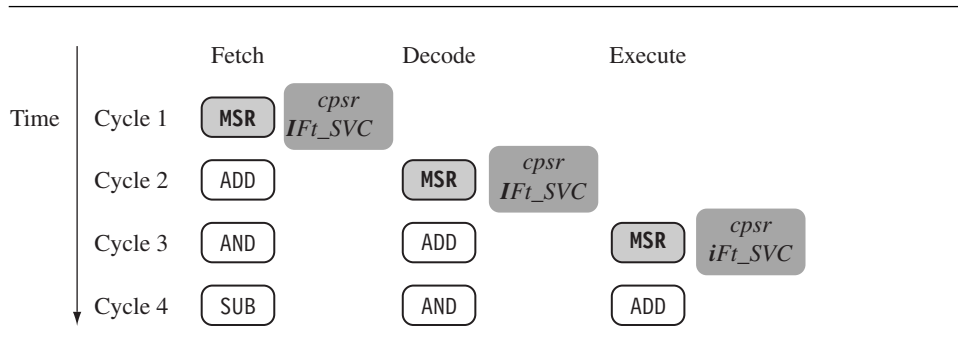
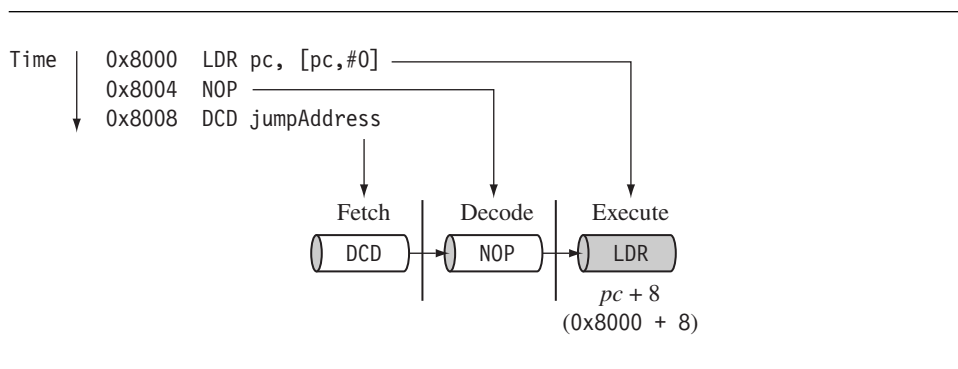


Figure 2.11 ARM instruction sequence.

Figure 2.12 Example: $pc = \text{address} + 8$.

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

There are three other characteristics of the pipeline worth mentioning. First, the execution of a branch instruction or branching by the direct modification of the pc causes the ARM core to flush its pipeline.

Second, ARM10 uses branch prediction, which reduces the effect of a pipeline flush by predicting possible branches and loading the new branch address prior to the execution of the instruction.

Third, an instruction in the execute stage will complete even though an interrupt has been raised. Other instructions in the pipeline will be abandoned, and the processor will start filling the pipeline from the appropriate entry in the vector table.

2.4 EXCEPTIONS, INTERRUPTS, AND THE VECTOR TABLE

When an exception or interrupt occurs, the processor sets the *pc* to a specific memory address. The address is within a special address range called the *vector table*. The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.

The memory map address 0x00000000 is reserved for the vector table, a set of 32-bit words. On some processors the vector table can be optionally located at a higher address in memory (starting at the offset 0xffff0000). Operating systems such as Linux and Microsoft's embedded products can take advantage of this feature.

When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see Table 2.6). 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*.

Table 2.6 The vector table.

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

- *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*.

2.5 CORE EXTENSIONS

The hardware extensions covered in this section are standard components placed next to the ARM core. They improve performance, manage resources, and provide extra functionality and are designed to provide flexibility in handling particular applications. Each ARM family has different extensions available.

There are three hardware extensions ARM wraps around the core: cache and tightly coupled memory, memory management, and the coprocessor interface.

2.5.1 CACHE AND 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. Of course, many small embedded systems do not require the performance gains that a cache brings.

ARM has two forms of cache. The first is found attached to the Von Neumann-style cores. It combines both data and instruction into a single unified cache, as shown in Figure 2.13. For simplicity, we have called the glue logic that connects the memory system to the AMBA bus *logic and control*.

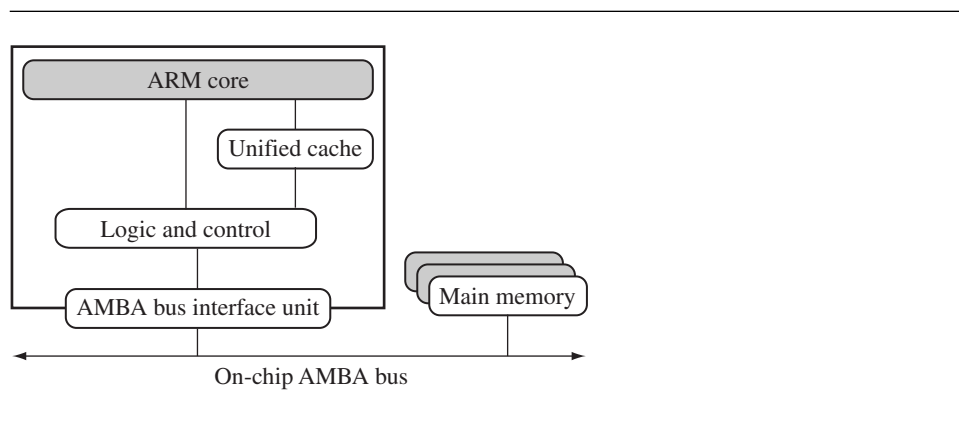


Figure 2.13 A simplified Von Neumann architecture with cache.

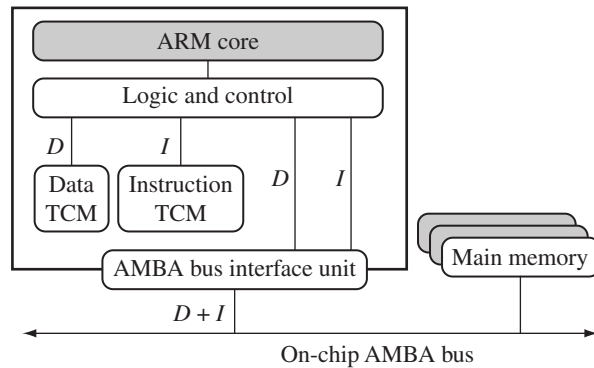


Figure 2.14 A simplified Harvard architecture with TCMs.

By contrast, the second form, attached to the Harvard-style cores, has separate caches for data and instruction.

A cache provides an overall increase in performance but at the expense of predictable execution. But for real-time systems it is paramount that 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 shown in Figure 2.14.

By combining both technologies, ARM processors can have both improved performance and predictable real-time response. Figure 2.15 shows an example core with a combination of caches and TCMs.

2.5.2 MEMORY MANAGEMENT

Embedded systems often use multiple memory devices. It is usually necessary to have 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.

ARM cores have three different types of memory management hardware—no extensions providing no protection, a memory protection unit (MPU) providing limited protection, and a memory management unit (MMU) providing full 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.

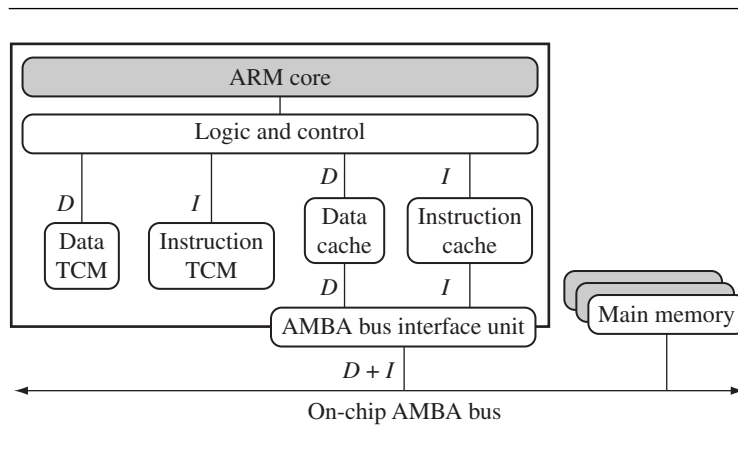


Figure 2.15 A simplified Harvard architecture with caches and TCMs.

- *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. The MPU is explained in Chapter 13.
- *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. The MMU is explained in Chapter 14.

2.5.3 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. Consider, for example, coprocessor 15: The ARM processor uses coprocessor 15 registers to control the cache, TCMs, and memory management.

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. But 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.

2.6 ARCHITECTURE REVISIONS

Every ARM processor implementation executes a specific *instruction set architecture* (ISA), although an ISA revision may have more than one processor implementation.

The ISA has evolved to keep up with the demands of the embedded market. This evolution has been carefully managed by ARM, so that code written to execute on an earlier architecture revision will also execute on a later revision of the architecture.

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

2.6.1 NOMENCLATURE

ARM uses the nomenclature shown in Figure 2.16 to describe the processor implementations. The letters and numbers after the word “ARM” indicate the features a processor

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

x—family
y—memory management/protection unit
z—cache
T—Thumb 16-bit decoder
D—JTAG debug
M—fast multiplier
I—EmbeddedICE macrocell
E—enhanced instructions (assumes TDMI)
J—Jazelle
F—vector floating-point unit
S—synthesizable version

Figure 2.16 ARM nomenclature.

may have. In the future the number and letter combinations may change as more features are added. Note the nomenclature does not include the architecture revision information.

There are a few additional points to make about the ARM nomenclature:

- All ARM cores after the ARM7TDMI include the *TDMI* features even though they may not include those letters after the “ARM” label.
- The processor *family* is a group of processor implementations that share the same hardware characteristics. For example, the ARM7TDMI, ARM740T, and ARM720T all share the same family characteristics and belong to the ARM7 family.
- *JTAG* is described by IEEE 1149.1 Standard Test Access Port and boundary scan architecture. It is a serial protocol used by ARM to send and receive debug information between the processor core and test equipment.
- *EmbeddedICE macrocell* is the debug hardware built into the processor that allows breakpoints and watchpoints to be set.
- *Synthesizable* means that the processor core is supplied as source code that can be compiled into a form easily used by EDA tools.

2.6.2 ARCHITECTURE EVOLUTION

The architecture has continued to evolve since the first ARM processor implementation was introduced in 1985. Table 2.7 shows the significant architecture enhancements from the original architecture version 1 to the current version 6 architecture. One of the most significant changes to the ISA was the introduction of the Thumb instruction set in ARMv4T (the ARM7TDMI processor).

Table 2.8 summarizes the various parts of the program status register and the availability of certain features on particular instruction architectures. “All” refers to the ARMv4 architecture and above.

2.7 ARM PROCESSOR FAMILIES

ARM has designed a number of processors that are grouped into different families according to the core they use. The families are based on the ARM7, ARM9, ARM10, and ARM11 cores. The postfix numbers 7, 9, 10, and 11 indicate different core designs. The ascending number equates to an increase in performance and sophistication. ARM8 was developed but was soon superseded.

Table 2.9 shows a rough comparison of attributes between the ARM7, ARM9, ARM10, and ARM11 cores. The numbers quoted can vary greatly and are directly dependent upon the type and geometry of the manufacturing process, which has a direct effect on the frequency (MHz) and power consumption (watts).

Table 2.7 Revision history.

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

Within each ARM family, there are a number of variations of memory management, cache, and TCM processor extensions. ARM continues to expand both the number of families available and the different variations within each family.

You can find other processors that execute the ARM ISA such as StrongARM and XScale. These processors are unique to a particular semiconductor company, in this case Intel.

Table 2.10 summarizes the different features of the various processors. The next subsections describe the ARM families in more detail, starting with the ARM7 family.

Table 2.8 Description of the *cpsr*.

Parts	Bits	Architectures	Description
Mode	4:0	all	processor mode
<i>T</i>	5	ARMv4T	Thumb state
<i>I & F</i>	7:6	all	interrupt masks
<i>J</i>	24	ARMv5TEJ	Jazelle state
<i>Q</i>	27	ARMv5TE	condition flag
<i>V</i>	28	all	condition flag
<i>C</i>	29	all	condition flag
<i>Z</i>	30	all	condition flag
<i>N</i>	31	all	condition flag

Table 2.9 ARM family attribute comparison.

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

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

^b MIPS are Dhrystone VAX MIPS.

2.7.1 ARM7 FAMILY

The ARM7 core has a Von Neumann-style architecture, where both data and instructions use the same bus. The core has a three-stage pipeline and executes the architecture ARMv4T instruction set.

The ARM7TDMI was the first of a new range of processors introduced in 1995 by ARM. It is currently a very popular core and is used in many 32-bit embedded processors. It provides a very good performance-to-power ratio. The ARM7TDMI processor core has been licensed by many of the top semiconductor companies around the world and is the first core to include the Thumb instruction set, a fast multiply instruction, and the EmbeddedICE debug technology.

Table 2.10 ARM processor variants.

CPU core	MMU/MPU	Cache	Jazelle	Thumb	ISA	E ^a
ARM7TDMI	none	none	no	yes	v4T	no
ARM7EJ-S	none	none	yes	yes	v5TEJ	yes
ARM720T	MMU	unified—8K cache	no	yes	v4T	no
ARM920T	MMU	separate—16K /16K <i>D + I</i> cache	no	yes	v4T	no
ARM922T	MMU	separate—8K/8K <i>D + I</i> cache	no	yes	v4T	no
ARM926EJ-S	MMU	separate—cache and TCMs configurable	yes	yes	v5TEJ	yes
ARM940T	MPU	separate—4K/4K <i>D + I</i> cache	no	yes	v4T	no
ARM946E-S	MPU	separate—cache and TCMs configurable	no	yes	v5TE	yes
ARM966E-S	none	separate—TCMs configurable	no	yes	v5TE	yes
ARM1020E	MMU	separate—32K/32K <i>D + I</i> cache	no	yes	v5TE	yes
ARM1022E	MMU	separate—16K/16K <i>D + I</i> cache	no	yes	v5TE	yes
ARM1026EJ-S	MMU and MPU	separate—cache and TCMs configurable	yes	yes	v5TE	yes
ARM1136J-S	MMU	separate—cache and TCMs configurable	yes	yes	v6	yes
ARM1136JF-S	MMU	separate—cache and TCMs configurable	yes	yes	v6	yes

^a E extension provides enhanced multiply instructions and saturation.

One significant variation in the ARM7 family is the ARM7TDMI-S. The ARM7TDMI-S has the same operating characteristics as a standard ARM7TDMI but is also synthesizable.

ARM720T is the most flexible member of the ARM7 family because it includes an MMU. The presence of the MMU means the ARM720T is capable of handling the Linux and Microsoft embedded platform operating systems. The processor also includes a unified 8K cache. The vector table can be relocated to a higher address by setting a coprocessor 15 register.

Another variation is the ARM7EJ-S processor, also synthesizable. ARM7EJ-S is quite different since it includes a five-stage pipeline and executes ARMv5TEJ instructions. This version of the ARM7 is the only one that provides both Java acceleration and the enhanced instructions but without any memory protection.

2.7.2 ARM9 FAMILY

The ARM9 family was announced in 1997. Because of its five-stage pipeline, the ARM9 processor can run at higher clock frequencies than the ARM7 family. The extra stages improve the overall performance of the processor. The memory system has been redesigned to follow the Harvard architecture, which separates the data D and instruction I buses.

The first processor in the ARM9 family was the ARM920T, which includes a separate $D + I$ cache and an MMU. This processor can be used by operating systems requiring virtual memory support. ARM922T is a variation on the ARM920T but with half the $D + I$ cache size.

The ARM940T includes a smaller $D + I$ cache and an MPU. The ARM940T is designed for applications that do not require a platform operating system. Both ARM920T and ARM940T execute the architecture v4T instructions.

The next processors in the ARM9 family were based on the ARM9E-S core. This core is a synthesizable version of the ARM9 core with the E extensions. There are two variations: the ARM946E-S and the ARM966E-S. Both execute architecture v5TE instructions. They also support the optional *embedded trace macrocell* (ETM), which allows a developer to trace instruction and data execution in real time on the processor. This is important when debugging applications with time-critical segments.

The ARM946E-S includes TCM, cache, and an MPU. The sizes of the TCM and caches are configurable. This processor is designed for use in embedded control applications that require deterministic real-time response. In contrast, the ARM966E does not have the MPU and cache extensions but does have configurable TCMs.

The latest core in the ARM9 product line is the ARM926EJ-S synthesizable processor core, announced in 2000. It is designed for use in small portable Java-enabled devices such as 3G phones and personal digital assistants (PDAs). The ARM926EJ-S is the first ARM processor core to include the Jazelle technology, which accelerates Java bytecode execution. It features an MMU, configurable TCMs, and $D + I$ caches with zero or nonzero wait state memories.

2.7.3 ARM10 FAMILY

The ARM10, announced in 1999, was designed for performance. It extends the ARM9 pipeline to six stages. It also supports an optional vector floating-point (VFP) unit, which adds a seventh stage to the ARM10 pipeline. The VFP significantly increases floating-point performance and is compliant with the IEEE 754.1985 floating-point standard.

The ARM1020E is the first processor to use an ARM10E core. Like the ARM9E, it includes the enhanced E instructions. It has separate 32K $D + I$ caches, optional vector floating-point unit, and an MMU. The ARM1020E also has a dual 64-bit bus interface for increased performance.

ARM1026EJ-S is very similar to the ARM926EJ-S but with both MPU and MMU. This processor has the performance of the ARM10 with the flexibility of an ARM926EJ-S.

2.7.4 ARM11 FAMILY

The ARM1136J-S, announced in 2003, was designed for high performance and power-efficient applications. ARM1136J-S was the first processor implementation to execute architecture ARMv6 instructions. It incorporates an eight-stage pipeline with separate load-store and arithmetic pipelines. Included in the ARMv6 instructions are single instruction multiple data (SIMD) extensions for media processing, specifically designed to increase video processing performance.

The ARM1136JF-S is an ARM1136J-S with the addition of the vector floating-point unit for fast floating-point operations.

2.7.5 SPECIALIZED PROCESSORS

StrongARM was originally co-developed by Digital Semiconductor and is now exclusively licensed by Intel Corporation. It has been popular for PDAs and applications that require performance with low power consumption. It is a Harvard architecture with separate $D + I$ caches. StrongARM was the first high-performance ARM processor to include a five-stage pipeline, but it does not support the Thumb instruction set.

Intel's XScale is a follow-on product to the StrongARM and offers dramatic increases in performance. At the time of writing, XScale was quoted as being able to run up to 1 GHz. XScale executes architecture v5TE instructions. It is a Harvard architecture and is similar to the StrongARM, as it also includes an MMU.

SC100 is at the other end of the performance spectrum. It is designed specifically for low-power security applications. The SC100 is the first SecurCore and is based on an ARM7TDMI core with an MPU. This core is small and has low voltage and current requirements, which makes it attractive for smart card applications.

2.8 SUMMARY

In this chapter we focused on the hardware fundamentals of the actual ARM processor. The ARM processor can be abstracted into eight components—ALU, barrel shifter, MAC, register file, instruction decoder, address register, incrementer, and sign extend.

ARM has three instruction sets—ARM, Thumb, and Jazelle. The register file contains 37 registers, but only 17 or 18 registers are accessible at any point in time; the rest are banked according to processor mode. The current processor mode is stored in the *cpsr*. It holds the current status of the processor core as well interrupt masks, condition flags, and state. The state determines which instruction set is being executed.

An ARM processor comprises a core plus the surrounding components that interface it with a bus. The core extensions include the following:

- *Caches* are used to improve the overall system performance.
- *TCMs* are used to improve deterministic real-time response.

- *Memory management* is used to organize memory and protect system resources.
- *Coprocessors* are used to extend the instruction set and functionality. Coprocessor 15 controls the cache, TCMs, and memory management.

An ARM processor is an implementation of a specific instruction set architecture (ISA). The ISA has been continuously improved from the first ARM processor design. Processors are grouped into implementation families (ARM7, ARM9, ARM10, and ARM11) with similar characteristics.

This Page Intentionally Left Blank