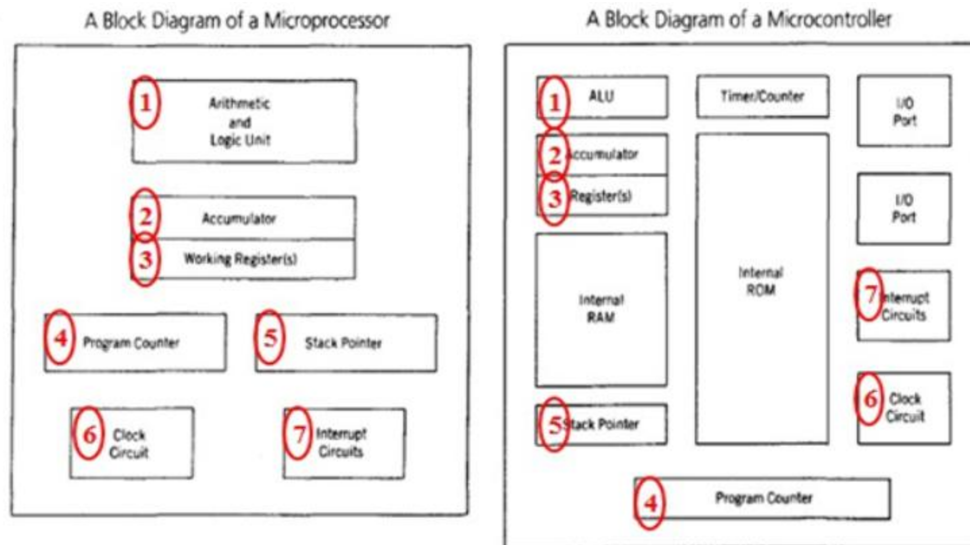
MODULE – 1 ARM EMBEDDED SYSTEMS & ARM PROCESSOR FUNDAMENTALS

MICROPROCESSORS versus MICROCONTROLLERS:

A *microprocessor* is an electronic component that is used by a computer to do its work. It is a central processing unit on a single integrated circuit chip containing millions of very small components including transistors, resistors, and diodes that work together.

A *microcontroller* is a compact integrated circuit designed to govern a specific operation in an embedded system. A typical microcontroller includes a processor, memory and input/output (I/O) peripherals on a single chip.

Microprocessors	Microcontrollers
Microprocessors generally does not have RAM, ROM and I/O pins.	Microcontroller is 'all in one' processor, with RAM, I/O ports, all on the chip.
Microprocessors usually use its pins as a bus to interface to RAM, ROM, and peripheral devices. Hence, the controlling bus is expandable at the board level.	Controlling bus is internal and not available to the board designer.
Microprocessors are generally capable of being built into bigger general purpose applications.	Microcontrollers are usually used for more dedicated applications.
Microprocessors, generally do not have power saving system.	Microcontrollers have power saving system, like idle mode or power saving; mode so overall it uses less power.
The overall cost of systems made with Microprocessors is high, because of the high number of external components required.	Microcontrollers are made by using complementary metal oxide semiconductor technology; so they are far cheaper than Microprocessors.
Processing speed of general microprocessors is above 1 GHz; so it works much faster than Microcontrollers.	Processing speed of Microcontrollers is about 8 MHz to 50 MHz.
Microprocessors are based on von-Neumann model; where, program and data are stored in same memory module.	Microcontrollers are based on Harvard architecture; where, program memory and data memory are separate.



ARM EMBEDDED SYSTEMS

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

The first ARM1 prototype was designed 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.

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

THE RISC DESIGN PHYLOSOPHY:

- ✓ The ARM core uses *reduced instruction set computer (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. The following Figure illustrates these major differences.

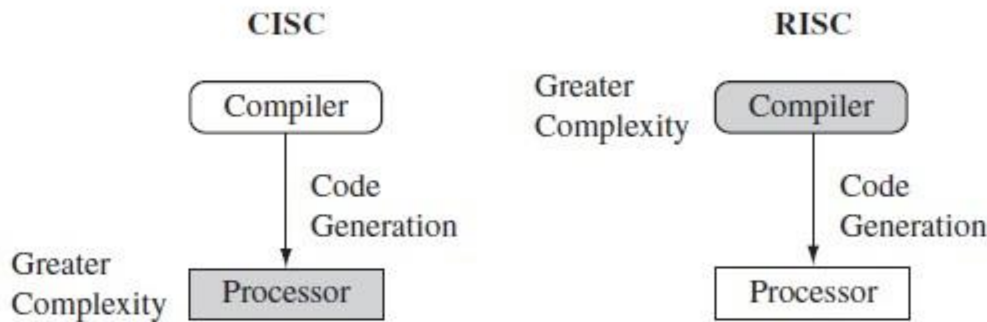


Fig: CISC vs. RISC

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

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 having 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. 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 mini-program called 3microcode as on CISC processos

3. *Registers*—RISC machines have a large general-purpose register set. Any register can contain either data or an address. Registers act as the fast local memory store for all data processing operations.
   - 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.

THE ARM DESIGN PHYLOSOPHY:

There are a number of physical features that have driven the ARM processor design.

- ✓ Portable embedded systems require ***battery power***. The ARM processor has been specially designed to be small to reduce power consumption and extend battery operation—essential for applications such as mobile phones and personal digital assistants (PDAs).
- ✓ ***High code density*** is another major requirement since embedded systems have limited memory due to cost and/or physical size restrictions—useful for applications that have limited on-board memory, such as mobile phones and mass storage devices.
- ✓ Embedded systems are ***price sensitive***
- o Hence, *use slow and low-cost memory devices* to get substantial savings—essential for high-volume applications like digital cameras.

- Also, *reduce the area of the die* taken up by the embedded processor; smaller the area used by the embedded processor, reduced cost of the design and manufacturing 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.

- ✓ 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

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

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

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.

The following Figure 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.

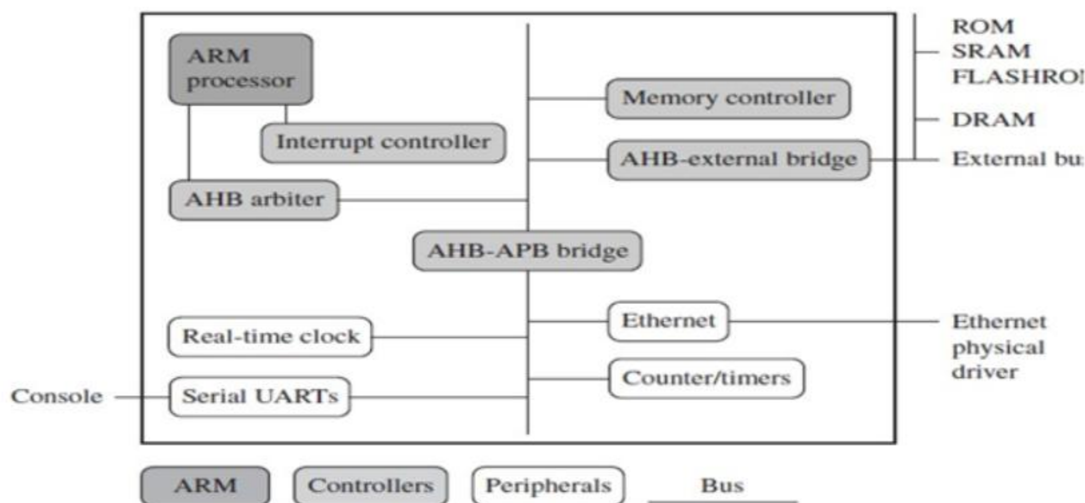


Figure: An ARM-based Embedded Device, a Microcontroller

We can separate the device into ***four main hardware components***:

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

ARM Bus Technology:

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:

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

A bus has ***two architecture*** levels:

A *physical level*—covers the electrical characteristics and bus width (16, 32, or 64 bits).

The *protocol*—the logical rules that govern the communication between the processor and a peripheral.

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. 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.
- ✓ ARM has introduced ***two variations*** on the AHB bus: *Multi-layer AHB* and *AHB-Lite*.
  - o The Multi-layer AHB bus allows multiple active bus masters.
  - o AHB-Lite is a subset of the AHB bus and it is limited to a single bus master.
- ✓ The example device shown in the above Figure has three buses:
  - o an *AHB bus* for the high- performance peripherals
  - o an *APB bus* for the slower peripherals
  - o a third *bus for external peripherals*, proprietary to this device.

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.

Hierarchy: All computer systems have memory arranged in some form of hierarchy. The following Figure shows the memory trade-offs: the fastest memory cache is physically located nearer the ARM processor core and the slowest secondary memory is set further away. Generally the closer memory is to the processor core, the more it costs and the smaller its capacity.

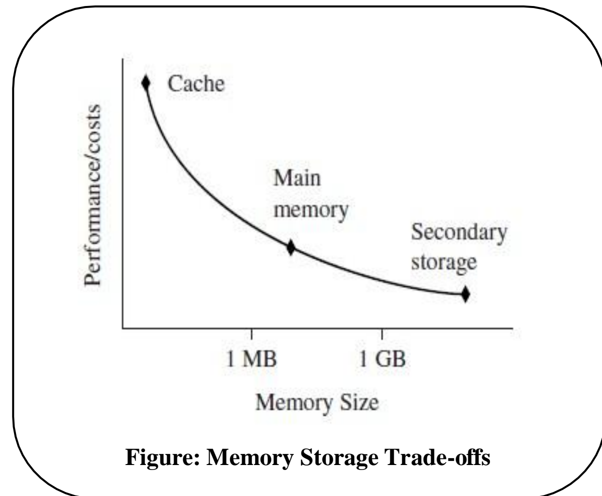


Figure: Memory Storage Trade-offs

- ✓ 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 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. Hard disk drives and CD-ROM drives are examples of secondary storage.

Width: The memory width is the number of bits the memory returns on each access—typically 8, 16, 32, or 64 bits.

- ✓ The memory width has a direct effect on the overall performance and cost ratio. Lower bit memories are less expensive, but reduce the system performance.

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

Table: Fetching Instruction from Memory

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

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

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

- ✓ *Synchronous dynamic random access memory (SDRAM)* is one of many subcategories of DRAM. It can run at much higher clock speeds than conventional memory. SDRAM synchronizes itself with the processor bus, because it is clocked. Internally the data is fetched from memory cells, pipelined, and finally brought out on the bus in a burst.

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.

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.

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.

1. 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.
2. The *vector interrupt controller (VIC)* is more powerful than the standard interrupt controller, because it prioritizes interrupts and simplifies the determination of which device caused the interrupt.
   - Depending on the type, the VIC will either call the standard interrupt exception handler, which can load the address of the handler.

EMBEDDED SYSTEM SOFTWARE:

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

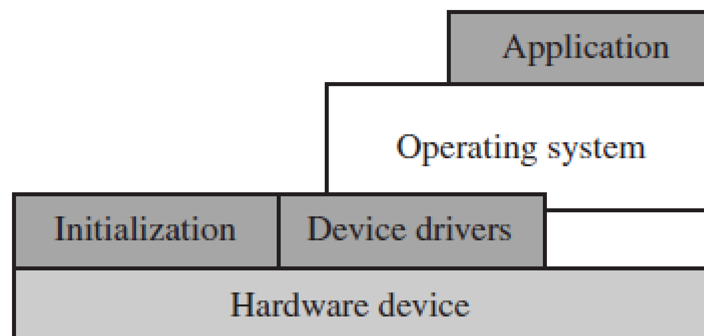


Figure: Software Abstraction Layers Executing on Hardware

- ✓ 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.
- ✓ The *operating system* provides an infrastructure to control applications and manage hardware system resources.
- ✓ The *device drivers* provide a consistent software interface to the peripherals on the hardware device.
- ✓ An *application* performs one of the tasks required for a device.
  - For example, a mobile phone might have a diary application.
- There may be multiple applications running on the same device, controlled by the operating system.

Initialization (Boot) Code:

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

1. ***Initial hardware configuration*** involves setting up the target platform, so that it can boot an image. The target platform comes up in a standard configuration; but, this configuration normally requires modification to satisfy the requirements of the booted image.
    - o For example, the memory system normally requires reorganization of the memory map, as shown in the following Example.

Example: 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.

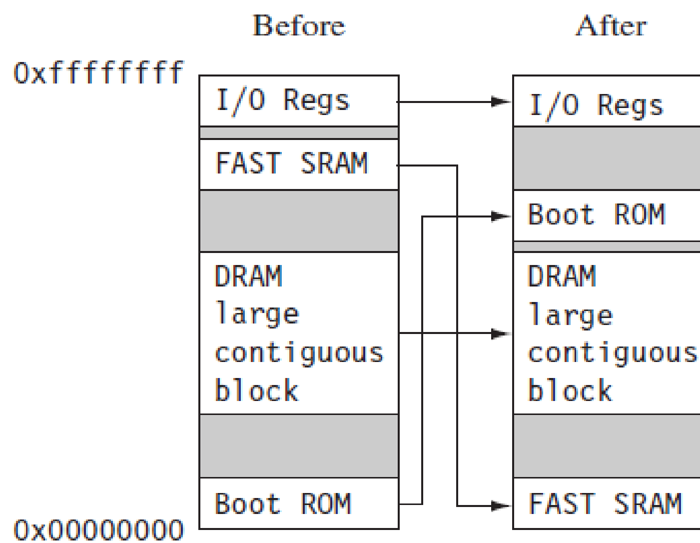


Figure: Memory Remapping

The above Figure 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.

2. ***Diagnostic***s are often embedded in the initialization code. Diagnostic code tests the system by exercising the hardware target to check if the target is in working order. It also tracks down
standard system-related issues. The primary purpose of diagnostic code is fault identification and isolation.
3. ***Booting*** involves loading an image and handing control over to that image. The boot process itself can be complicated if the system must boot different operating systems or different versions of the same operating system.
   - Booting an image is the final phase, but first you must load the image. Loading an image involves anything from copying an entire program including code and data into RAM, to just copying a data area containing volatile variables into RAM. Once booted, the system hands over control by modifying the program counter to point into the start of the image.

Operating System:

- ✓ The initialization process prepares the hardware for an operating system to take control. An operating system organizes the system resources: the peripherals, memory, and processing time.
- ✓ ARM processors support over 50 operating systems. We can divide operating systems into *two main categories*: real-time operating systems (RTOSs) and platform operating systems.

1. ***RTOSs*** provide guaranteed response times to events. Different operating systems have different amounts of control over the system response time.
   - A *hard real-time* application requires a guaranteed response to work at all.
   - In contrast, a *soft real-time* application requires a good response time, but the performance degrades more gracefully if the response time overruns.
2. ***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.

Applications:

- ✓ The operating system schedules *applications*—code dedicated to handle a particular task. An application implements a processing task; the operating system controls the environment.
    - o An embedded system can have one active application or several applications running simultaneously.
- ✓ ARM processors are found in numerous market segments, including networking, auto-motive, mobile and consumer devices, mass storage, and imaging.
- ✓ ARM processor is found in networking applications like home gateways, DSL modems for high- speed Internet communication, and 802.11 wireless communications.
- ✓ The mobile device segment is the largest application area for ARM processors, because of mobile phones.

ARM processors are also found in mass storage devices such as hard drives and imaging products such as inkjet printers—applications that are cost sensitive and high volume**.**

- In contrast, ARM processors are not found in applications that require leading-edge high performance. Because these applications tend to be low volume and high cost, ARM has decided not to focus designs on these types of applications.

ARM PROCESSOR FUNDAMENTALS

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

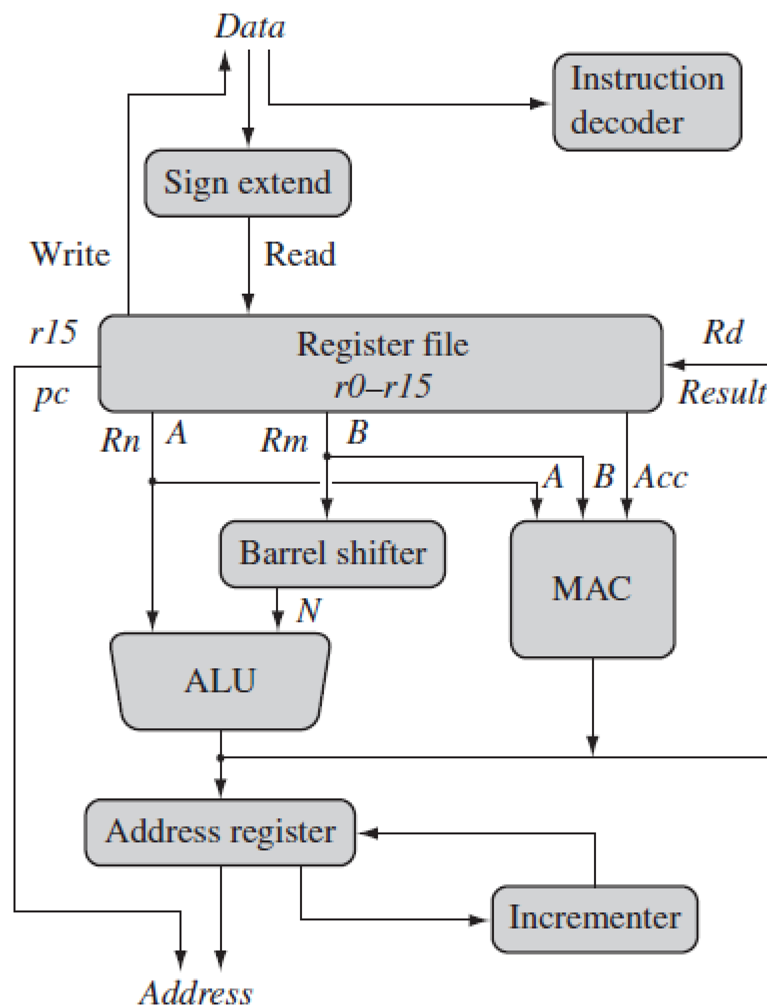


Figure: ARM Core dataflow Model

The arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area.

- ✓ Data enters the processor core through the Data bus. The data may be an instruction to execute or a data item.
- ✓ Figure 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.

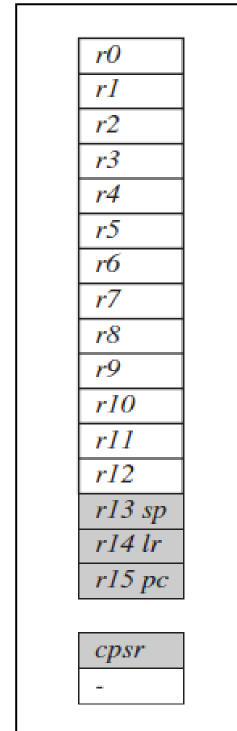
- ✓ The ARM processor, like all RISC processors, uses ***load-store architecture***—means it has two instruction types for transferring data in and out of the processor:
  - o load instructions copy data from memory to registers in the core
  - o store instructions copy data from registers to memory
- ✓ There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out in registers.
- ✓ Data items are placed in the register file—a storage bank made up of 32-bit registers.
  - o 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.
  - o 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.

<u>REGISTERS:</u>

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*.

The Figure shows the active registers available in user mode. (A protected mode is normally used when executing applications).

- ✓ The processor can operate in seven different modes.
- ✓ 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 visible to the programmer are *r0* to *r15*.
- ✓ The ARM processor has three registers assigned to a particular task or special function: *r13*, *r14*, and *r15*. They are given with different labels to differentiate them from the other registers.
  - *Register r13* is traditionally used as the ***stack pointer (sp)*** and stores the head of the stack in the current processor mode.
  - *Register r14* is called the ***link register (lr)*** and is where the core puts the return address whenever it calls a subroutine.
  - *Register r15* is the ***program counter (pc)*** and contains the address of the next instruction to be fetched by the processor.
- ✓ In 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.
- ✓ In addition to the 16 data registers, there are two program status registers: ***cpsr (current program status register)*** and ***spsr (saved program status register)***.

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. The following Figure shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future expansion.

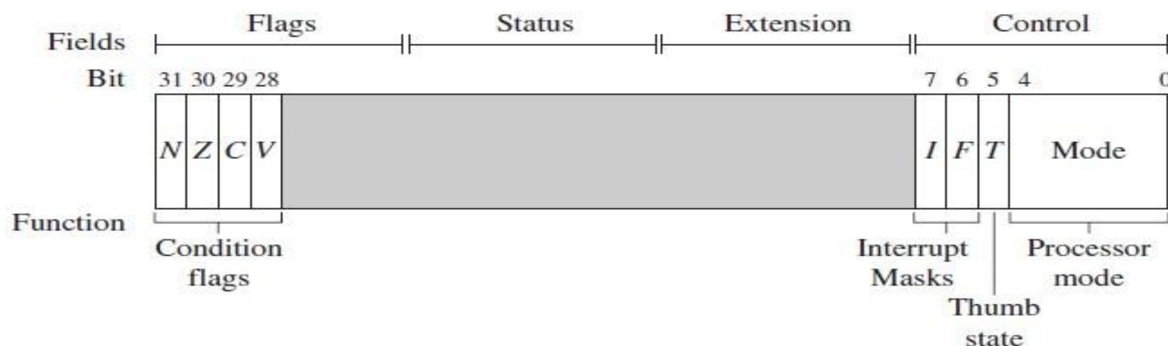


Figure: A Generic Program Status Register (psr)

The *cpsr* is divided into four fields, each 8 bits wide: *flags*, *status*, *extension*, and *control*. In current designs the extension and status fields are reserved for future use.

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

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

It is highly probable that future designs will assign extra bits for the monitoring and control of new features.

Processor Modes:

- ✓ The processor mode determines which registers are active and the access rights to the *cpsr* register itself. Each processor mode is either privileged or non-privileged:
  - A *privileged mode* allows full read-write access to the *cpsr*.
  - A *non-privileged mode* only allows read access to the control field in the *cpsr*, but still allows read-write access to the condition flags.
- ✓ There are *seven processor modes* in total:
  - *six privileged modes* (abort, fast interrupt request, interrupt request, supervisor, system, and undefined)
    - The processor enters ***abort mode*** when there is a failed attempt to access memory.
    - ***Fast interrupt request*** and ***interrupt request modes*** correspond to the two interrupt levels available on the ARM processor.
    - ***Supervisor mode*** is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in.
    - ***System mode*** is a special version of user mode that allows full read-write access to the *cpsr*.
    - ***Undefined mode*** is used when the processor encounters an instruction that is undefined or not supported by the implementation.
  - *one non-privileged mode* (user).
    - ***User mode*** is used for programs and applications.

Banked Registers:

The following Figure shows all 37 registers in the register file.

- ✓ Of these, 20 registers are hidden from a program at different times.

- ✓ These registers are called *banked registers* and are identified by the shading in the diagram.

They are available only when the processor is in a particular mode; for example, abort mode has banked registers r13_abt, r14_abt and spsr_abt.

- ✓ Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or _mode.
- ✓ 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.
  - o 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*.

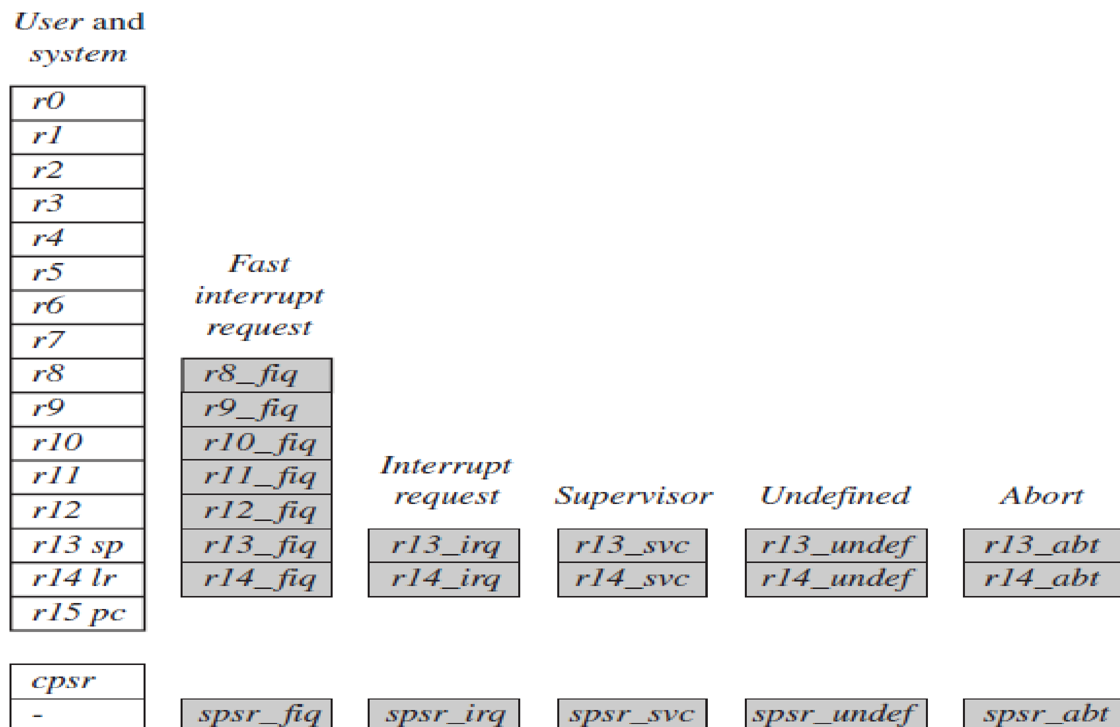


Figure: Complete ARM Register Set

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

- ✓ The following exceptions and interrupts cause a mode change: *reset*, *interrupt request*, *fast interrupt request*, *software interrupt*, *data abort*, *prefetch abort*, and *undefined instruction*.
- ✓ Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location.
- ✓ The following Figure illustrates what happens when an interrupt forces a mode change.

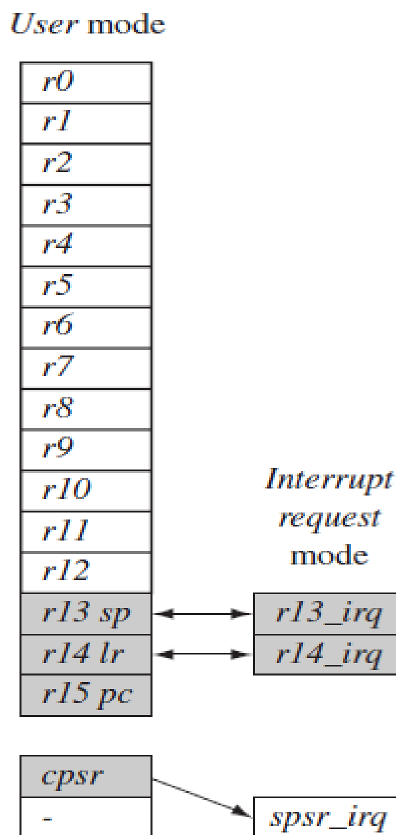


Figure: Changing Mode on an Exception

- ✓ 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.
  - o Note *r14_irq* contains the return address and *r13_irq* contains the stack pointer for interrupt request mod

The above Figure also shows a new register appearing in interrupt request mode: the saved program status register (spsr), which stores the previous mode's cpsr. The cpsr being copied to 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.
- ✓ Another important feature to note is that the *cpsr* is not copied into the *spsr* when a mode change is forced due to a program writing directly to the *cpsr*. The saving of the *cpsr* only occurs when an exception or interrupt is raised.
- ✓ When power is applied to the core, it starts in supervisor mode, which is privileged. Starting in a privileged mode is useful since initialization code can use full access to the *cpsr* to set up the stacks for each of the other modes.
- ✓ The following Table lists the various modes and the associated binary patterns. The last column of the table gives the bit patterns that represent each of the processor modes in the *cpsr*.

Table: Processor Mode

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

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.
- ✓ 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 instruction

You cannot inter-mingle 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.

The following Table compares the ARM and Thumb instruction set features.

Table: ARM and Thumb Instruction Set Features

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

- ✓ The ARM designers introduced a third instruction set called Jazelle. **Jazelle** executes 8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java byte-codes.
- ✓ To execute Java byte-codes, you require the Jazelle technology plus a specially modified version of the Java virtual machine.

The following Table gives the Jazelle instruction set features.

Table: Jazelle instruction set features

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

Interrupt Masks:

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

The *cpsr* has two interrupt mask bits, 7 and 6 (or *I* and *F*), which control the masking of IRQ and FIQ, respectively.

✓ The *I* bit masks IRQ when set to binary 1; and similarly, the *F* bit masks FIQ when set to binary 1.

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*.

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

Table: 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
N	Negative	bit 31 of the result is a binary 1

These flags are located in the most significant bits in the *cpsr*. These bits are used for conditional execution. The following Figure shows a typical value for the *cpsr* with both DSP extensions and Jazelle.

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.

Figure: Example:* cspr *= nzCvqjiFt_SVC

- ✓ In the *cpsr* example shown in above Figure, the *C* flag is the only condition flag set. The rest nzvq flags are all clear.
- ✓ The processor is in ARM state because neither the Jazelle *j* nor Thumb *t* bits are set. The IRQ interrupts are enabled, and FIQ interrupts are disabled.
- ✓ Finally, you can see from the Figure, the processor is in *supervisor (SVC) mode*, since the mode[4:0] is equal to binary 10011.

Conditional Execution:

- ✓ Conditional execution controls whether or not the core will execute an instruction.
- ✓ Prior to execution, the processor compares the condition attribute with the condition flags in the

  cpsr. If they match, then the instruction is executed; otherwise the instruction is ignored.
- ✓ The condition attribute is post-fixed to the instruction mnemonic, which is encoded into the instruction.
- ✓ The following Table lists the conditional execution code mnemonics. When a condition mnemonic is not present, the default behavior is to set it to always (AL) execute.

Table: Condition Mnemonics

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

PIPELINE:

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

Figure: ARM7 Three-stage Pipeline

The above Figure 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.

The following Figure illustrates pipeline using a simple example.

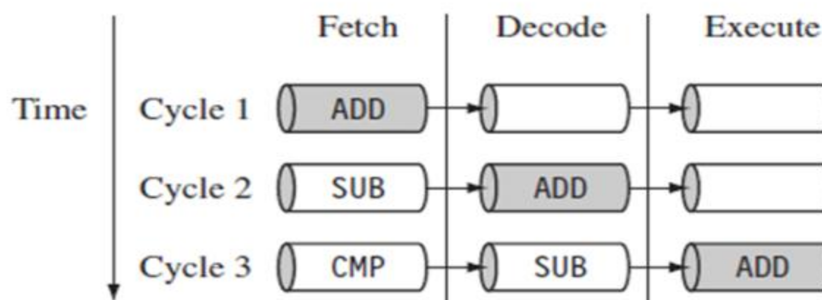


Figure: Pipelined Instruction Sequence

- ✓ The Figure shows a sequence of three instructions being fetched, decoded, and executed by the processor.
    - o The three instructions are placed into the pipeline sequentially.
    - o In the first cycle, the core fetches the ADD instruction from memory.
    - o In the second cycle, the core fetches the SUB instruction and decodes the ADD instruction.
    - o 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.
- o 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*.
- o The increased pipeline length also means increased *system latency* and there can be *data dependency* between certain stages.
- o The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages, as shown in Figure.

Figure: ARM9 Five-stage Pipeline

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

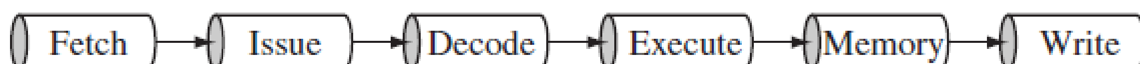


Figure: ARM10 Six-stage Pipeline

- The ARM10 –
  - can process on average 1.3 Dhrystone MIPS per MHz
  - have about 34% more throughput than an ARM7 processor core
  - but again at a higher latency cost.

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

Pipeline Executing Characteristics:

✓ The ARM pipeline will not process an instruction, until it passes completely through the execute stage.

- For example, an ARM7 pipeline (with three stages) has executed an instruction only when the fourth instruction is fetched.

The following Figure shows an instruction sequence on an ARM7 pipeline.

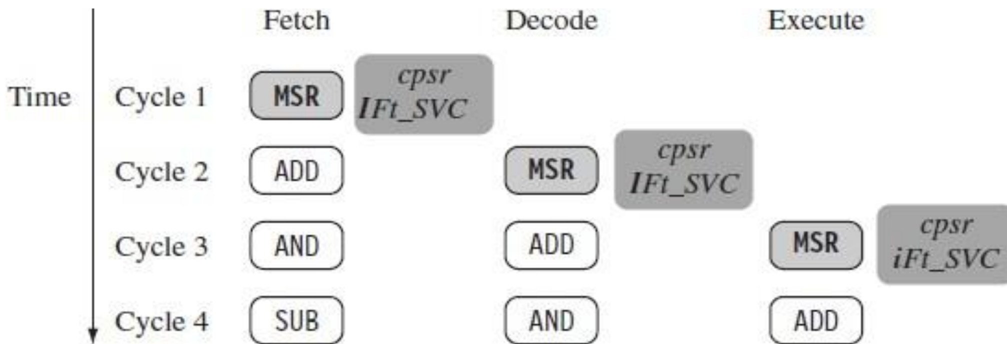


Figure: ARM Instruction Sequence

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

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

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

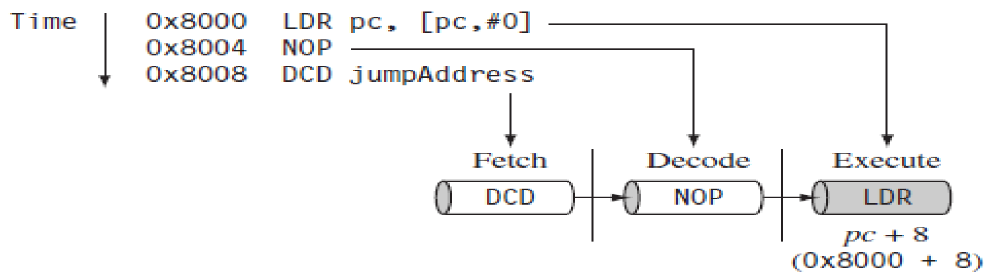


Figure: Example: *pc* = address + 8

- ✓ In the execute stage, the *pc* always points to the address of the instruction plus 8 bytes. In other words, the *pc* always points to the address of the instruction being executed plus two instructions ahead.
- ✓ Note when the processor is in Thumb state the *pc* is the instruction address plus 4.
- ✓ There are three other characteristics of the pipeline.
  - First, the execution of a branch instruction or branching by the direct modification of the *pc* causes the ARM core to flush its pipeline.
  - Second, ARM10 uses branch prediction, which reduces the effect of a pipeline flush by predicting possible branches and loading the new branch address prior to the execution of the instruction.
  - Third, an instruction in the execute stage will complete even though an interrupt has been raised. Other instructions in the pipeline will be abandoned, and the processor will start filling the pipeline.

EXCEPTIONS, INTERRUPTS AND THE VECTOR TABLE:

- ✓ When an exception or interrupt occurs, the processor sets the *pc* to a specific memory address. The address is within a special address range called the *vector table*.
  - The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.
  - The memory map address 0x00000000 (or in some processors starting at the offset 0xffff0000) is reserved for the vector table, a set of 32-bit words.

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

TabTable: The Vector Table

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

- ✓ Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:
  - o ***Reset*** vector is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
  - o ***Undefined*** instruction vector is used when the processor cannot decode an instruction.
  - o ***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.
  - o ***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.
  - o ***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.
  - o ***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*.
  - o ***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*.

CORE EXTENSIONS:

- ✓ *Core extensions* are the standard hardware 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: cache and tightly coupled memory, memory management, and the coprocessor interface.

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.
- ✓ 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 the following Figure.

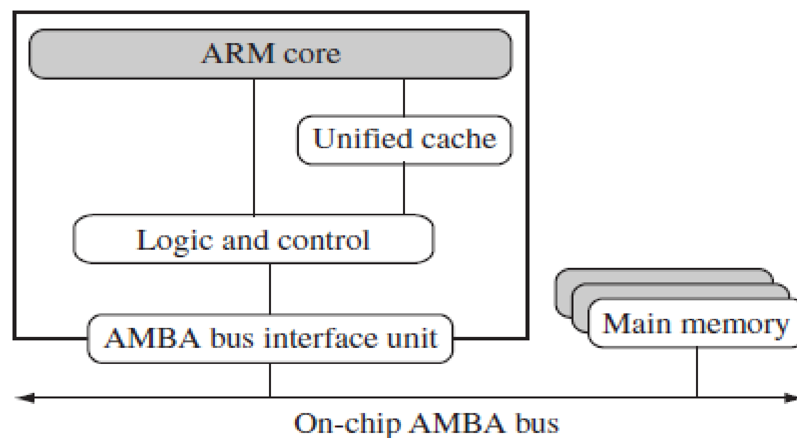


Figure: Von Neumann Architecture with Cache

- ✓ The second form, attached to the Harvard-style cores, has separate caches for data and instruction, as shown in the following Figure.

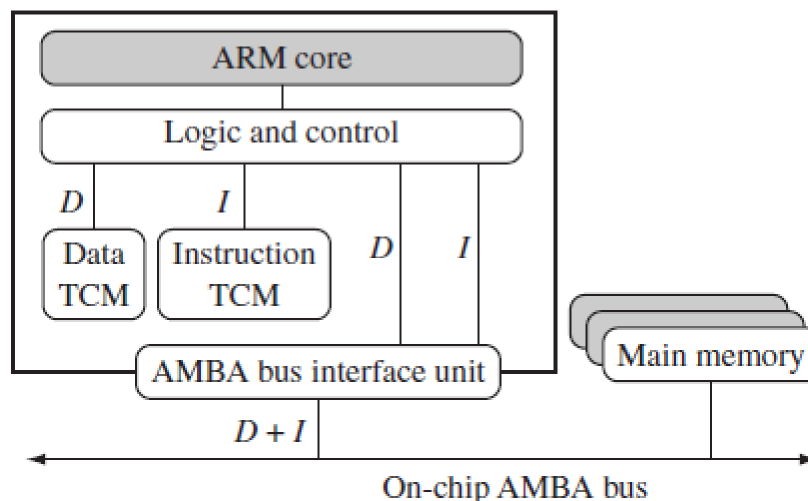


Figure: Harvard Architecture with TCMs

A cache provides an overall increase in performance, but at the expense of predictable execution. But the real-time systems require the code execution to be deterministic— the time taken for loading and storing instructions or data must be predictable.

- ✓ This is achieved using a form of memory called *tightly coupled memory (TCM)*. TCM is fast SRAM located close to the core and guarantees the clock cycles required to fetch instructions or data.
- ✓ TCMs appear as memory in the address map and can be accessed as fast memory.

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

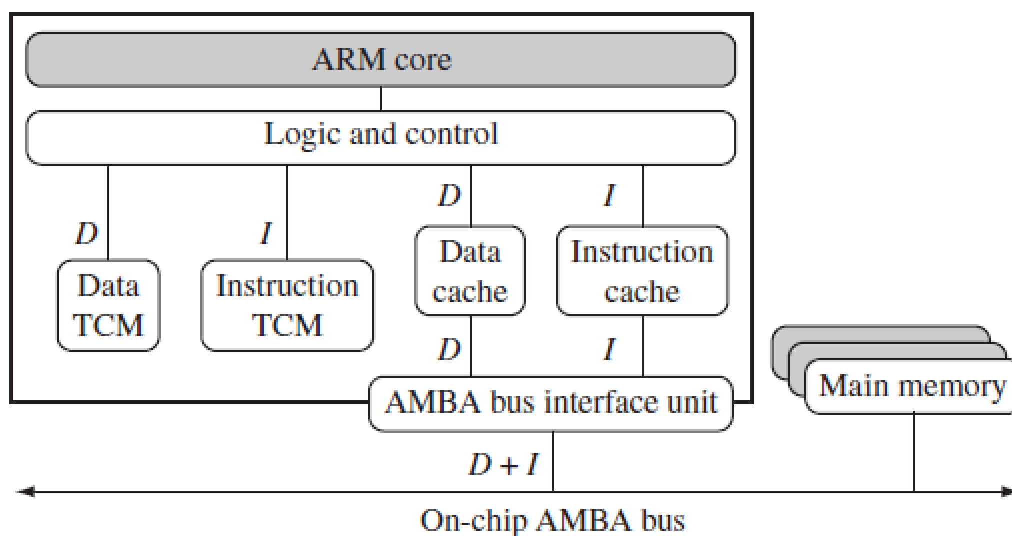


Figure: Harvard Architecture with Caches and TCMs

Memory Management:

- ✓ Embedded systems often use multiple memory devices. It is usually necessary to have a method to organize these devices and protect the system from applications trying to make inappropriate accesses to hardware. This is achieved with the assistance of memory management hardware.
- ✓ ARM cores have *three different types of memory management hardware*—
  - o no extensions providing no protection
  - o a memory protection unit (MPU) providing limited protection
  - o a memory management unit (MMU) providing full protection
- ✓ ***Non protected memory*** is fixed and provides very little flexibility. It is normally used for small, simple embedded systems that require no protection from rogue applications.

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

Coprocessors:

- ✓ Coprocessors can be attached to the ARM processor. A coprocessor extends the processing features of a core by extending the instruction set or by providing configuration registers. More than one coprocessor can be added to the ARM core via the coprocessor interface.
- ✓ The coprocessor can be accessed through a group of dedicated ARM instructions that provide a load-store type interface.
  - For example, coprocessor 15: The ARM processor uses coprocessor 15 registers to control the cache, TCMs, and memory management.
- ✓ The coprocessor can also extend the instruction set by providing a specialized group of new instructions.
  - For example, there are a set of specialized instructions that can be added to the standard ARM instruction set to process vector floating-point (VFP) operations.
- ✓ These new instructions are processed in the decode stage of the ARM pipeline.
  - If the decode stage sees a coprocessor instruction, then it offers it to the relevant coprocessor.
  - If the coprocessor is not present or doesn't recognize the instruction, then the ARM takes an undefined instruction exception, which allows you to emulate the behavior of the coprocessor in software.

****

****

MODULE–2

INTRODUCTION TO THE ARM INSTRUCTION SET

Different ARM architecture revisions support different instructions. However, new revisions usually add instructionsand remain backwardlycompatible. Codeyou writeforarchitectureARMv4T should execute on an *ARMv5TE* processor.

The following Table provides a complete list of ARM instructions available in the *ARMv5E instruction set architecture* (*ISA*). This ISA includes all the core ARM instructions as well as some of the newer features in the ARM instruction set.

Table:ARM Instruction Set

Mnemonics	ARM ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
B	v1	branch relative +/− 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation
CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory

Mnemonics	ARM ISA	Description
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register
MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register

Mnemonics	ARM ISA	Description
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values
SMLA*xy*	v5E	signed multiply accumulate instructions ($(16 \times 16) + 32 = 32$-bit)
SMLAL	v3M	signed multiply accumulate long ($(32 \times 32) + 64 = 64$-bit)
SMLAL*xy*	v5E	signed multiply accumulate long ($(16 \times 16) + 64 = 64$-bit)
SMLAW*y*	v5E	signed multiply accumulate instruction ($((32 \times 16) \gg 16) + 32 = 32$-bit)
SMULL	v3M	signed multiply long ($32 \times 32 = 64$-bit)

Mnemonics	ARM ISA	Description
SMUL*xy*	v5E	signed multiply instructions ($16 \times 16 = 32$-bit)
SMULW*y*	v5E	signed multiply instruction ($(32 \times 16) \gg 16 = 32$-bit)
STC STC2	v2 v5	store to memory single or multiple 32-bit values from coprocessor
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interruption
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long ($(32 \times 32) + 64 = 64$-bit)
UMULL	v3M	unsigned multiply long ($32 \times 32 = 64$-bit)

In thefollowingsections,thehexadecimalnumbersarerepresentedwiththeprefix*0x*andbinarynumbers with the prefix *0b*. The examples follow this format:

PRE *<pre-conditions>*

<instruction/s>

POST*<post-conditions>*

Inthepre-andpost-conditions,memoryisdenotedas

mem<data_size>[address]

This refers to *data_size* bits of memory starting at the given byte address. For example, *mem32[1024]* is the 32-bit value starting at address 1 KB.

ARM instructions process data held in registers and memory is accessed only with load and store instructions.

ARM instructions commonly take two or three operands. For instance,theADD instruction

below adds the two values store din registers *r1*and *r2*(the source registers).It writes the result to register *r3* (the destination register).

Instruction Syntax	Destination register (*Rd*)	Source register 1 (*Rn*)	Source register 2 (*Rm*)
ADD r3, r1, r2	*r3*	*r1*	*r2*

ARMinstructionsclassifiedas—dataprocessinginstructions,branchinstructions,load-store instructions, software interrupt instruction, and program status register instructions.

DATA PROCESS INGINSTRUCTIONS:

Thedataprocessinginstructionsmanipulatedatawithinregisters.They are—

- ✓ move instructions, arithmetic instructions, logical instructions, comparison instructions, and multiply instructions.

Most data pocessing instructions can process one of their operands using the barrel shifter. If you use the S suffix on a data processing instruction,then it updates the flags in the *cpsr*. Move and logical operations update the carry flag *C*, negative flag *N*, and zero flag *Z*.

- o The *C* flag is set from the result of the barrel shift as the last bit shifted out.
- o The *N* flag is set to bit 31of the result.
- o The *Z* flag is set if the result is zero.

MOVE Instructions:

Move instruction copies *N* into a destination register *Rd*, where *N* is a register or immediate

```
Syntax: <instruction>{<cond>}{S} Rd, N
```

MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

value. This instruction is useful for setting initial values and transferring data between registers.

Example: This example shows a simple move instruction. The MOV instruction takes the contents of register *r5* and copies them into register *r7*, in this case, taking the value *5*, and overwriting the value *8* in register *r7*.

PRE ***r5=5***

r7 =8

MOV ***r7, r5*** ***;let r7=r5***

POST ***r5=5***

r7 =5

Barrel Shifter:

In above Example, we showed a MOV instruction where *N* is a simple register. But *N* can be more than just a register or immediate value; it can also be a register *Rm* that has been preprocessed by the barrel shifter prior to being used by a data processing instruction.

- ✓ Data processing instructions are processed with in the arithmetic logic unit (ALU).
- ✓ A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.
- ✓ Pre-processing or shift occurs with in the cycle time of the instruction.
    - o This shift increases the power and flexibility of many data processing operations.
    - o This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.
- ✓ There are data processinginstructionsthatdonotusethebarrelshift,forexample, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add)instructions.

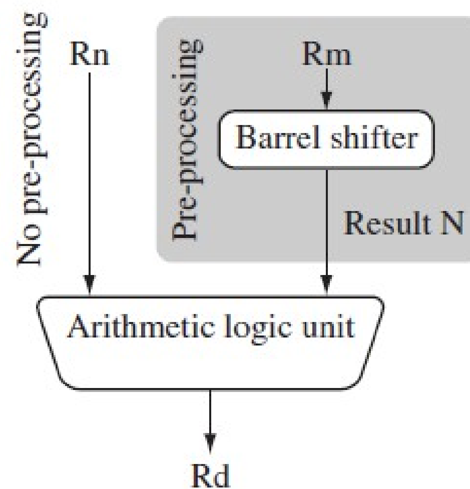


Figure:BarrelShifterand ALU

- ✓ Figure shows the data flow between the ALU and the barrel shifter.
- ✓ Register *Rn* enters the ALU without any pre-processing of registers.
- ✓ We apply a logical shift left (LSL) to register *Rm* before moving it to the destination register.This is the same as applying the standard *C* language shift operator « to the register.
- ✓ The MOV instruction copies the shift operator result *N* into register *Rd*. *N* represents the result of the LSL operation described in the following Table.

Table:Barrel Shifter Operations

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	*x*LSL *y*	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	*x*LSR *y*	$(\text{unsigned})x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x*ASR *y*	$(\text{signed})x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x*ROR *y*	$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$	#1–31 or *Rs*
RRX	rotate right extended	*x*RRX	$(c\ \text{flag} \ll 31) \mid ((\text{unsigned})x \gg 1)$	none

Note: *x* represents the register being shifted and *y* represents the shift amount.

- ✓ The five different shift operations that you can use within the barrel shifter are summarized in the above Table.

PRE r5=5

r7 =8

MOV r7,r5, LSL#2 ;let r7=r5*4 =(r5 <<2)

POSTr5=5

r7 =20

- ✓ The above example multiplies register *r5* by four and then places the result in to register *r7*.
- ✓ The following Figure illustrates a logical shift left by one.

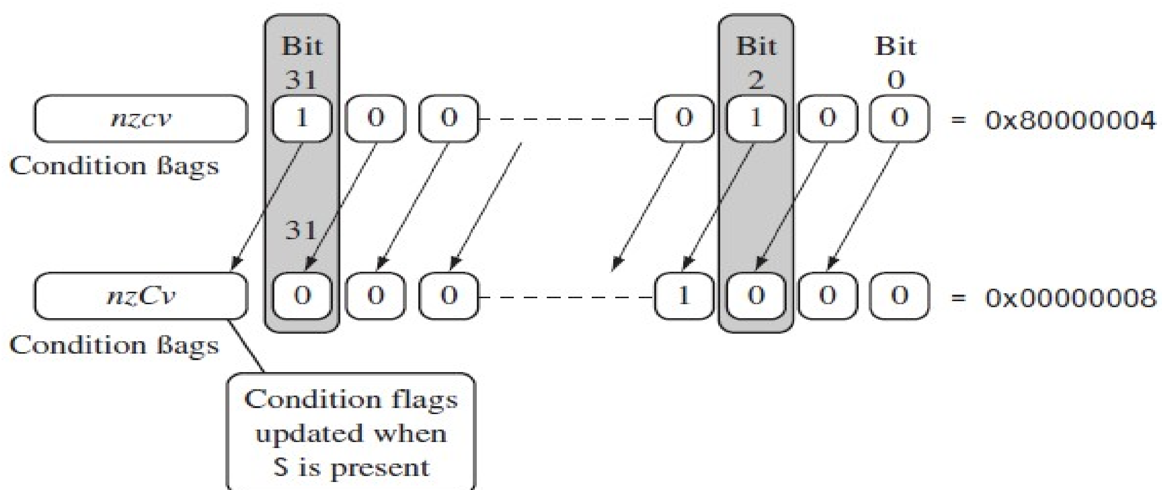


Figure:Logical Shift Left by One

- ✓ For example, the contents of bit *0* are shifted to bit *1*. Bit *0* is cleared. The *C* flag is updated with the last bit shifted out of the register. This is bit (*32 - y*) of the original value, where *y* is the shift amount. When *y* is greater than one, then a shift by *y* positions is the same as a shift by one position executed *y* times.

Example:This example of a MOVS instruction shifts register r1left by one bit.This multiplies register r1 by a value 2^1.As you can see, the C flag is updated in the cpsr because the S suffix is present in the instruction mnemonic.

```
PRE   cpsr=nzcvqiFt_USER
      r0 = 0x00000000
      r1=0x80000004
MOVS r0,r1,LSL#1
POST  cpsr=nzCvqiFt_USER
      r0 = 0x00000008
      r1 =0x80000004
```

The following Table lists the syntax for the different barrel shift operations available on data processing instructions. The second operand N can be an immediate constant preceded by #, a register value Rm, or the value of Rm processed by a shift.

Table:BarrelShifterOperationSyntaxfordataProcessingInstructions

N shift operations	Syntax
Immediate	#immediate
Register	Rm
Logical shift left by immediate	Rm, LSL #shift_imm
Logical shift left by register	Rm, LSL Rs
Logical shift right by immediate	Rm, LSR #shift_imm
Logical shift right with register	Rm, LSR Rs
Arithmetic shift right by immediate	Rm, ASR #shift_imm
Arithmetic shift right by register	Rm, ASR Rs
Rotate right by immediate	Rm, ROR #shift_imm
Rotate right by register	Rm, ROR Rs
Rotate right with extend	Rm, RRX

Arithmetic Instructions:

The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.

Syntax: <instruction>{<cond>}{S} Rd, Rn, N

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

N is the result of the shifter operation.

Example:The following simple subtract instruction subtracts avalue stored in register *r2*from a value stored in register *r1*. The result is stored in register *r0*.

PRE *r0=0x00000000*

r1=0x00000002

r2 =0x00000001

SUB r0,r1,r2

POST *r0 =0x00000001*

Example:The following reverse subtract instruction (RSB) subtracts *r1* from the constant value#0, writing the result to *r0*. You can use this instruction to negate numbers.

PRE *r0 =0x00000000*

r1 =0x00000077

RSB r0, r1,#0 ;Rd=0x0-r1

POST *r0=-r1=0xffffff89*

Example:The SUBS instruction is useful for decrementing loop counters.In this example, we subtract the immediate value one from the value one stored in register *r1*. The result value zero is written to register *r1*. The *cpsr* is updated with the ZC flags being set.

PRE *cpsr=nzcvqiFt_USER*

r1 = 0x00000001

SUBS r1,r1, #1

POST *cpsr=nZCvqiFt_USER*

r1 = 0x00000000

Using the Barrel Shifter with Arithmetic Instructions:

The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set.The following Example illustrates the use of the in line barrel shifter with an arithmetic instruction. The instruction multiplies the value stored in register r1 by three.

Example: Register r1 is first shifted one location to the left to give the value of twice r1. The ADD instruction then adds the result of the barrel shift operation to register r1. The final result transferred into register r0 is equal to three times the value stored in register r1.

PRE r0=0x00000000

r1=0x0000005

ADDr0,r1,r1,LSL#1

POSTr0 = 0x0000000f

r1 =0x00000005

Logical Instructions:

Logical instructions perform bitwise logical operations on the two source registers.

```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

Example: This example shows a logical OR operation between registers *r1* and *r2*. Register *r0* holds the result.

PRE r0 =0x00000000

r1=0x02040608

r2 =0x10305070

ORR r0,r1,r2

POSTr0=0x12345678

Example:This example shows a more complicated logical instruction called BIC,which carries out a logical bit clear.

PRE r1 =0b1111

r2=0b0101

BIC r0, r1, r2

POST r0=0b1010

This is equivalent to– Rd=Rn**ANDNOT**(N)

In this example, register *r2* contains a binary pattern where every binary*1* in *r2*clears a corresponding bit location in register *r1*.

This instruction is particularly useful when clearing status bits and is frequently used to change interrupt masks in the *cpsr*.

NOTE:The logical instructions update the *cpsr* flags only if theS suffix is present. These instructions can use barrel-shif ted second operands in the same way as the arithmetic instructions.

Comparison Instructions:

- ✓ The comparison instructions are used to compare or test a register with a 32-bit value.
- ✓ They update the *cpsr* flag bits according to the result,but do not affect other registers.
- ✓ After the bits have been set, the information can then be used to change program flow by using conditional execution.
- ✓ It is not required to apply the S suffix for comparison instructions to update the flags.

```
Syntax: <instruction>{<cond>} Rn, N
```

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

N is the result of the shifter operation.

Example: This example shows a CMP comparison instruction. You can see that both registers, *r0* and *r9*, are equal before executing the instruction. The value of the *Z* flag prior to execution is *0* and isrepresented by a lowercase *z*. After execution the *Z* flag changes to *1* or an uppercase *Z*. This change indicates equality.

PRE cpsr=nzcvqiFt_USER

r0 = 4

r9 =4

CMP r0, r9

POST cpsr=nZcvqiFt_USER

- ✓ The CMP is effectively a subtract instruction with the resultd is carded;similarly the TS T instruction is a logical AND operation, and TEQ is a logical exclusive OR operation.
- ✓ For each,the results are discarded but the condition bits are updated in the *cpsr*.
- ✓ Itisimportanttounderstandthatcomparisoninstructionsonlymodifytheconditionflagsofthe *cpsr*anddonotaffecttheregistersbeingcompared.

Multiply Instructions:

The multiply instructions multiply the contents of a pair of registers and, depending upon the instruction, accumulate the results in with another register.

The long multiplies accumulate on to a pair of registers representing a 64-bit value.The final resultis placed in a destination register or a pair of registers.

```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

MLA	multiply and accumulate	$Rd = (Rm^{*} Rs) + Rn$
MUL	multiply	$Rd = Rm^{*} Rs$

The number of cycles taken to execute a multiply instruction depends on the processor implementation. For some implementations the cycle timing also depends on the value in *Rs*.

Example: This example shows a simple multiply instruction that multiplies registers *r1* and *r2* together and places the result in to register *r0*.In this example, register *r1* is equal to the value *2*, and *r2* is equal to *2*.The result,*4*,is then placed into register *r0*.

PRE *r0 =0x00000000*

r1=0x00000002

r2 =0x00000002

*MULr0,r1, r2 ;r0 =r1*r2*

POST*r0=0x00000004*

r1=0x00000002

r2 =0x00000002

```
Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs
```

SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm * Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm * Rs$

The long multiply instructions (SMLAL, SMULL, UMLAL, and UMULL) produce a 64-bit result. The result is too large to fit a single 32-bit register so the result is placed in two registers labeled *RdLo* and *RdHi*. *RdLo* holds the lower 32 bits of the 64-bit result, and *RdHi* holds the higher 32 bits of the 64-bit result. The following shows an example of a long unsigned multiply instruction.

Example: The instruction multiplies registers *r2* and *r3* and places the result into register *r0* and *r1*. Register *r0* contains the lower 32 bits, and register *r1* contains the higher 32 bits of the 64-bit result.

PRE r0= 0x00000000
r1=0x00000000
r2 = 0xf0000002
r3 = 0x00000002

UMULLr0,r1,r2,r3 ;[r1,r0]=r2*r3

POSTr0 = 0xe0000004 ;=RdLo

r1 = 0x00000001 ;= RdHi

BRANCH INSTRUCTIONS:

A branch instruction changes the flow of execution or is used to call a routine. This type of instruction allows programs to have subroutines, if-then-else structures, and loops.

The change of execution flow forces the program counter *pc* to point to a new address. The

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

ARMv5E instruction set includes four different branch instructions.

- ✓ The address *label* is stored in the instruction a sasigned *pc*-relative offset and must be within approximately 32 MB of the branch instruction.
- ✓ *T* refers to theThumb bit in the *cpsr*.When instructions set *T*,the ARM switches toThumb state.

Example: This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. Thebackward branch creates an infinite loop.

B forward

ADD r1, r2, #4

ADD r0, r6, #2

ADD r3, r7, #4

forward

SUBr1,r2,#4

```
backward
        ADD r1, r2, #4
        SUB r1, r2,#4
        ADD  r4,r6, r7
        B       backward
```

In this example, *forward* and *backward* are the labels. The branch labels are placed at the beginning of the line and are used to mark an address that can be used later by the assembler to calculate the branch offset.

- ✓ The branch with link, or BL, instruction is similar to the B instruction but overwrites the link register *lr* with a return address. It performs a subroutine call.

Example: This example shows a simple fragment of code that, branches to a subroutine using the BL instruction. To return from a subroutine, you copy the link register to the *pc*.

```
        BL      subroutine      ; branch to subroutine
        CMP r1, #5              ; compare r1 with 5
        MOVEQ r1, #0            ;if(r1==5)thenr1=0

Subroutine:
        <subroutinecode>
        MOVpc,lr                ;returnbymovingpc=lr
```

- ✓ The branch exchange (BX) and branch exchange with link (BLX) are the third type of branch instruction.
- ✓ The BX instruction uses an absolute address stored in register *Rm*. It is primarily used to branch to and from Thumb code. The *T* bit in the *cpsr* is updated by the least significant bit of the branch register.
- ✓ Similarly the BLX instruction updates the *T* bit of the *cpsr* with the least significant bit and additionally sets the link register with the return address.

LOAD-STORE INSTRUCTIONS:

Load-store instructions transfer data between memory and processor registers. There are three types of load-store instructions: single-register transfer, multiple-register transfer, and swap.

Single-Register Transfer:

- ✓ These instructions are used for moving a single data item in and out of a register.
- ✓ The data types supported are signed and unsigned words (32-bit),half-words(16-bit),and bytes.

Here are the various load-store single-register transfer instructions.

```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1
        LDR{<cond>}SB|H|SH Rd, addressing2
        STR{<cond>}H Rd, addressing2
```

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*

LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

- ✓ LDR and STR instructions can load and store data on a boundary alignment that is the same as the data type size being loaded or stored.
  - For example, LDR can only load 32-bit words on a memory address that is a multiple of four bytes—0, 4, 8, and so on.

Example: This example shows a load from a memory address contained in register *r1*, followed by a store back to the same address in memory.

```
;
;load register r0 with the contents of
;the memory address pointed to by register
;r1.
;
        LDRr0, [r1]          ;=LDRr0,[r1,#0]
;store the contents of register r0 to
;the memory address pointed to by
;registerr1.
;
        STRr0,[r1]           ;=STRr0,[r1,#0]
```

The first instruction loads a word from the address stored in register *r1* and places it into register *r0*. The second instruction goes the other way by storing the contents of register *r0* to the address

contained in register *r1*. The offset from register *r1* is zero. Register *r1* is called the *base address register*.

Single-Register Load-Store Addressing Modes:

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods: pre index with write back, pre index, and post index.

Table: Index Methods

Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

Note: ! indicates that the instruction writes the calculated address back to the base address register.

- ✓ *Pre index with write back* calculates an address from a base register plus address offset and then updates that address base register with the new address.
- ✓ *Pre index* offset is the same as the pre index with write back but does not update the address base register.
  - o The pre index mode is useful for accessing an element in a data structure.
- ✓ *Post index* only updates the address base register after the address is used.
  - o Thepostindexandpreindexwithwritebackmodesareusefulfortraversingan array.

Example: Pre indexing with write back:

PRE r0=0x00000000

r1 =0x00090000

mem32[0x00009000]=0x01010101

mem32[0x00009004]=0x02020202

LDR r0, [r1, #4]!

POST (1) r0=0x02020202

r1=0x00009004

LDRr0,[r1, #4]

Pre indexing:

POST (2) r0=0x02020202

r1=0x00009000

LDRr0, [r1], #4

Post indexing:

POST (3) r0=0x01010101

r1 = 0x00009004

- ✓ The above Example used a pre index method. This example shows how each indexing method affects the address held in register *r1*, as well as the data loaded into register *r0*.

The addressing modes available with a particular load or store instruction depend on the instruction class. The following Table shows the addressing modes available for load and store of a 32-bit word or an unsigned byte.

Table: Single-Register Load-Store Addressing, Word or Unsigned Byte

Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	`[Rn, #+/-offset_12]`
Preindex with register offset	`[Rn, +/-Rm]`
Preindex with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]`
Preindex writeback with immediate offset	`[Rn, #+/-offset_12]!`
Preindex writeback with register offset	`[Rn, +/-Rm]!`
Preindex writeback with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]!`
Immediate postindexed	`[Rn], #+/-offset_12`
Register postindex	`[Rn], +/-Rm`
Scaled register postindex	`[Rn], +/-Rm, shift #shift_imm`

- ✓ A signed offset or register is denoted by "+/-", identifying that it is either a positive or negative offset from the base address register *Rn*. The base address register is a pointer to a byte in memory, and the offset specifies a number of bytes.
- ✓ Immediate means the address is calculated using the base address register and a 12-bit offset encoded in the instruction.
- ✓ Register means the address is calculated using the base address register and a specific register's contents.
- ✓ Scaledmeanstheaddressiscalculatedusingthebaseaddressregisterandabarrelshift operation.

The following Table provides an example of the different variations of the LDR instruction.

Table: Examples of LDR Instructions using Different Addressing Modes

	Instruction	*r0* =	*r1* +=
Preindex with writeback	`LDR r0,[r1,#0x4]!`	mem32[r1 + 0x4]	0x4
	`LDR r0,[r1,r2]!`	mem32[r1+r2]	r2
	`LDR r0,[r1,r2,LSR#0x4]!`	mem32[r1 + (r2 LSR 0x4)]	(r2 LSR 0x4)
Preindex	`LDR r0,[r1,#0x4]`	mem32[r1 + 0x4]	*not updated*
	`LDR r0,[r1,r2]`	mem32[r1 + r2]	*not updated*
	`LDR r0,[r1,-r2,LSR #0x4]`	mem32[r1-(r2 LSR 0x4)]	*not updated*
Postindex	`LDR r0,[r1],#0x4`	mem32[r1]	0x4
	`LDR r0,[r1],r2`	mem32[r1]	r2
	`LDR r0,[r1],r2,LSR #0x4`	mem32[r1]	(r2 LSR 0x4)

The following Table shows the addressing modes available on load and store instructions using 16-bit half word or signed byte data.

Table: Single-Register Load-Store Addressing, Half word, Signed Half word, Signed Byte and Double word

Addressing[2] mode and index method	Addressing[2] syntax
Preindex immediate offset	[Rn, #+/-offset_8]
Preindex register offset	[Rn, +/-Rm]
Preindex writeback immediate offset	[Rn, #+/-offset_8]!
Preindex writeback register offset	[Rn, +/-Rm]!
Immediate postindexed	[Rn], #+/-offset_8
Register postindexed	[Rn], +/-Rm

These operations cannot use the barrel shifter. There are no STRSB or STRSH instructions since STRH stores both a signed and unsigned halfword; similarly STRB stores signed and unsigned bytes.

The following Table shows the variations for STRH instructions.

Table: Variations of STRH Instructions

	Instruction	Result	*r1 +=*
Preindex with writeback	STRH r0,[r1,#0x4]!	mem16[r1+0x4]=r0	0x4
	STRH r0,[r1,r2]!	mem16[r1+r2]=r0	r2
Preindex	STRH r0,[r1,#0x4]	mem16[r1+0x4]=r0	*not updated*
	STRH r0,[r1,r2]	mem16[r1+r2]=r0	*not updated*
Postindex	STRH r0,[r1],#0x4	mem16[r1]=r0	0x4
	STRH r0,[r1],r2	mem16[r1]=r0	r2

Multiple-Register Transfer:

- ✓ Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction.
- ✓ The transfer occurs from a base address register *Rn* pointing in to memory.
  - o Multiple-register transfer instructions are more efficient from single-register transfers for
    - ▪ Moving blocks of data around memory and
    - ▪ Saving and restoring context and stacks.
- ✓ Load-store multiple instructions can increase interrupts latency.
- ✓ ARM implementations do not usually interrupt instructions while they are executing.
  - o For example, on an ARM7a load multiple instruction takes $2 + Nt$ cycles, where N is the number of registers to load and t is the number of cycles required for each sequential access to memory.
- ✓ If an interrupt has been raised, then it has no effect until the load-store multiple instruction is complete.

- ✓ Compilers, such as *armcc*, provide a switch to control the maximum number of registers being transferred on a load-store, which limits the maximum interrupt latency.

```
Syntax: <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}
```

LDM	load multiple registers	{Rd}*N <- mem32[start address + 4*N] optional Rn updated
STM	save multiple registers	{Rd}*N -> mem32[start address + 4*N] optional Rn updated

The following Table shows the different addressing modes for the load-store multiple instructions. Here *N* is the number of registers in the list of registers.

Table: Addressing Mode for Load-Store Multiple Instructions

Addressing mode	Description	Start address	End address	*Rn*!
IA	increment after	Rn	$Rn + 4*N - 4$	$Rn + 4*N$
IB	increment before	$Rn + 4$	$Rn + 4*N$	$Rn + 4*N$
DA	decrement after	$Rn - 4*N + 4$	Rn	$Rn - 4*N$
DB	decrement before	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

- ✓ Any subset of the current bank of registers can be transferred to memory or fetched from memory.
- ✓ The base register *Rn* determines the source or destination address for a load-store multiple instruction. This register can be optionally updated following the transfer. This occurs when register *Rn* is followed by the! character, similar to the single-register load-store using pre index with write back.

Example: In this example, register *r0* is the base register *Rn* and is followed by !, indicating that the register is updated after the instruction is executed. You will notice within the load multiple instruction that the registers are not individually listed. Instead the "-" character is used to identify a range of registers. In this case the range is from register *r1* to *r3* inclusive.

Each register can also be listed, using a comma to separate each register within"{"and"}"brackets.

PRE mem32[0x80018]=0x03
mem32[0x80014]=0x02
mem32[0x80010]=0x01
r0 = 0x00080010
r1=0x00000000
r2=0x00000000
r3 =0x00000000
LDMIAr0!, {r1–r3}

POST r0=0x0008001c

r1=0x00000001

r2=0x00000002
r3=0x00000003

The following Figure shows a graphical representation.

Address pointer	Memory address	Data	
	0x80020	0x00000005	
	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000000
	0x80014	0x00000002	*r2* = 0x00000000
***r0* = 0x80010** →	0x80010	0x00000001	*r1* = 0x00000000
	0x8000c	0x00000000	

Figure: Pre-condition for LDMIA Instruction

- ✓ The base register *r0* points to memory address 0x80010 in the PRE condition.
- ✓ Memory addresses 0x80010, 0x80014, and 0x80018 contain the values1, 2, and 3 respectively.
- ✓ After the load multiple instruction executes, registers *r1*, *r2*, and *r3* contain these values as shown in the following Figure.

Address pointer	Memory address	Data	
	0x80020	0x00000005	
***r0* = 0x8001c** →	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000003
	0x80014	0x00000002	*r2* = 0x00000002
	0x80010	0x00000001	*r1* = 0x00000001
	0x8000c	0x00000000	

Figure: Post Condition for LDMIA Instruction

- ✓ The base register *r0* now points to memory address 0x8001c after the last loaded word.
- ✓ Now replace the LDMIA instruction with a load multiple and increment before LDMIB instruction and use the same PRE conditions.
- ✓ The first word pointed to by register *r0* is ignored and register *r1* is loaded from the next memory location as shown in the following Figure.

Address pointer	Memory address	Data	
	0x80020	0x00000005	
r0 = 0x8001c →	0x8001c	0x00000004	r3 = 0x00000004
	0x80018	0x00000003	r2 = 0x00000003
	0x80014	0x00000002	r1 = 0x00000002
	0x80010	0x00000001	
	0x8000c	0x00000000	

Figure: Post Condition for LDMIB Instruction

- ✓ After execution, register *r0* now points to the last loaded memory location. This is in contrast with the LDMIA example, which pointed to the next memory location.
- The decrement versions DA and DB of the load-store multiple instructions decrement the start address and then store to ascending memory locations.
- This is equivalent to descending memory but accessing the register list in reverse order.
- With the increment and decrement load multiples; you can access arrays forwards or backwards.
- They also allow for stack push and pull operations.

The following Table shows a list of load-store multiple instruction pairs.

Table: Load-Store Multiple Pairs when Base Update used

Store Multiple	Load Multiple
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

- If you use a store with base update, then the paired load instruction of the same number of registers will reload the data and restore the base address pointer.
- This is useful when you need to temporarily save a group of registers and restore them later.

Example: This example shows an STM *increment before* instruction followed by an LDM *decrement after* instruction.

PRE r0=0x00009000

r1=0x0000009

```
            r2=0x0000008
            r3=0x00000007
         STMIBr0!,{r1–r3}
         MOV r1, #1
         MOVr2,#2
         MOVr3,#3
PRE(2) r0=0x0000900c
         r1=0x00000001
         r2=0x00000002
         r3 =0x00000003
       LDMDAr0!,{r1–r3}
POST    r0=0x00009000
         r1=0x00000009
         r2=0x00000008
         r3 =0x00000007
```

The STMIB instruction stores the values 7, 8, 9 to memory. We then corrupt register *r1* to *r3*. The LDMDA reloads the original values and restores the base pointer *r0*.

Example: We illustrate the use of the load-store multiple instructions with a block memory copy example. This example is a simple routine that copies blocks of 32 bytes from a source address location to a destination address location.

The example has two load-stores multiple instructions, which use the same increment after addressing mode.

```
;r9 points to start of source data
;r10 points to start of destination data
;r11 points to end of the source
        loop
;load 32 bytes from source and update r9 pointer
        LDMIAr9!, {r0–r7}
;store 32 bytes to destination and updater 10 pointer
        STMIAr10!,{r0–r7}    ;and store them
;have were ached the end
        CMP    r9, r11
        BNE loop
```

- ✓ This routine relies on registers *r9*, *r10*, and *r11*being set up before the code is executed.
- ✓ Registers *r9* and *r11* determine the data to be copied, and register *r10* points to the destination in memory for the data.
- ✓ LDMIA loads the data pointed to by register *r9* into registers *r0* to *r7*. It also updates *r9*

to point to the next block of data to be copied.

- ✓ STMIA copies the contents of registers *r0* to *r7* to the destination memory address pointed to by register *r10*. It also updates *r10* to point to the next destination location.

CMP and BNE compare pointers r9 and r11 to check whether the end of the block copy has been reached.

- ✓ If the block copy is complete, then the routine finishes; otherwise the loop repeats with the updated values of register *r9* and *r10*.

- The BNE is the branch instruction B with a condition mnemonic NE (not equal). If the previous compare instruction sets the condition flags to not equal, the branch instruction is executed.

The following Figure shows the memory map of the block memory copy and how the routine moves through memory.

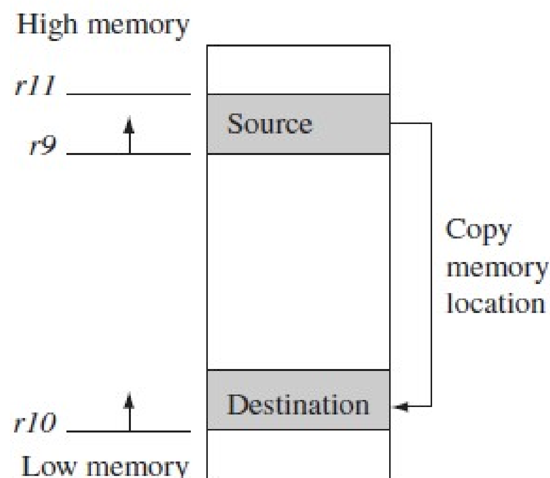


Figure: Block Memory Copy in the Memory map

Theoretically this loop can transfer 32 bytes (8 words) in two instructions, for a maximum possible throughput of 46 MB/second being transferred at 33 MHz. These numbers assume a perfect memory system with fast memory.

Stack Operation: The ARM architecture uses the load-store multiple instructions to carry outs tack operations.

- The *pop operation* (removing data from a stack) uses a load multiple instruction.
- The *push operation* (placing data on to the stack) uses as to remultiple instruction.

- ✓ Whenusingastackyouhavetodecidewhetherthestackwillgrowupordowninmemory.
    - o A stack is either–
        - ▪ *ascending(A)*–stacks grow towards higher memory addresses or

  - *descending(D)*–stacks grow towards lower memory addresses.

✓ When you use a *full stack (F)*, the stack pointer *sp* points to an address that is the last used or full location (i.e., *sp* points to the last item on the stack).

✓ If you use an empty stack (E) the sp points to an address that is the first unused or empty location (i.e., it points after the last item on the stack).

- There are number of load-store multiple addressing mode aliases available to support stack operations (see the following Table).

Table: Addressing Methods for Stack Operations

Addressing mode	Description	Pop	= LDM	Push	= STM
FA	full ascending	LDMFA	LDMDA	STMFA	STMIB
FD	full descending	LDMFD	LDMIA	STMFD	STMDB
EA	empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	empty descending	LDMED	LDMIB	STMED	STMDA

- Next to the *pop* column is the actual load multiple instruction equivalent.
  - For example, a full ascending stack would have the notation FA appended to the load multiple instruction—LDMFA. This would be translated into an LDMDA instruction.
- ARM has specified an *ARM-Thumb Procedure Call Standard (ATPCS)* that defines how routines are called and how registers are allocated. In the ATPCS, stacks are defined as being full descending stacks. Thus, the LDMFD and STMFD instructions provide the *pop* and *push* functions, respectively.

Example: The STMFD instruction pushes registers onto the stack, updating the *sp*. The following

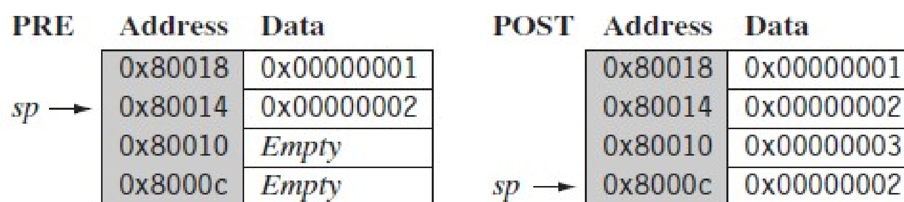


Figure shows a *push* onto a full descending stack.

Figure: STMFD Instruction–Full Stack *push* Operation

You can see that when the stack grows the stack pointer points to the last full entry in the stack.

PRE r1 =0x00000002

r4=0x00000003

sp =0x00080014

STMFD sp!,{r1,r4}

POST r1=0x00000002

r4 =0x00000003

sp =0x0008000c

Example: The following Figure shows a *push* operation on an empty stack using the STMED instruction.

PRE

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
sp →	0x80010	*Empty*
	0x8000c	*Empty*
	0x80008	*Empty*

POST

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
	0x8000c	0x00000002
sp →	0x80008	*Empty*

Figure: STMED Instruction–Empty Stack *push* Operation

The STMED instruction pushes the registers on to the stack but updates register *sp* to point to the next empty location.

PRE *r1 =0x00000002*

r4=0x00000003

sp =0x00080010

STMED sp!,{r1,r4}

POST*r1=0x00000002*

r4=0x00000003

sp =0x00080008

- ✓ When handling a checked stack there are three attributes that need to be preserved: the stack base, the stack pointer, and the stack limit.
- ✓ The stack base is the starting address of the stack in memory.
- ✓ The stack pointer initially points to the stack base; as data is pushed onto the stack, the stack pointer descends memory and continuously points to the top of stack. If the stack pointer passes the stack limit, then a stack overflow error has occurred.
- ✓ Here is a small piece of code that checks for stack overflow errors for a descending stack:

;checkforstackoverflow

SUBsp,sp,#size

CMP sp, r10

BLLO_stack_overflow *;condition*

- ATPCS defines register *r10* as the stack limit or *sl*. This is optional since it is only used when stack checking is enabled.
- The BLLO instruction is a branch with link instructionplustheconditionmnemonicLO.
  - If *sp* is less than register *r10* after the new items are pushed on to the stack,then *stack overflow* error has occurred.
  - If the stack pointer goes back past the stack base,then a *stack underflow* error has occurred.

Swap Instruction:

The swap instruction is a special case of a load-store instruction. It swaps the contents of memory withthe contents of a register.

This instruction is an *atomic operation*—it reads and writes alocation in the same bus operation,

```
Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]
```

SWP	swap a word between memory and a register	*tmp = mem32[Rn]* *mem32[Rn] = Rm* *Rd = tmp*
SWPB	swap a byte between memory and a register	*tmp = mem8[Rn]* *mem8[Rn] = Rm* *Rd = tmp*

preventing any other instruction from reading or writing to that location until it completes.

Swap cannot be interrupted by any otherinstruction orany other bus access. We say the system "holdsthe bus" until the transaction is complete. Also, swap instruction allows for both a word and a byte swap.

Example:Theswap instruction loadsaword from memory into register*r0*and overwritesthememory with register *r1*.

PRE mem32[0x9000]=0x12345678

r0=0x00000000

r1=0x11112222

r2 =0x00009000

SWPr0,r1, [r2]

POST mem32[0x9000]=0x11112222

r0 = 0x12345678

r1=0x11112222

r2 =0x00009000

Example: This example shows a simple data guard that can be used to protect data from being written by another task. The SWP instruction "holds the bus" until the transaction is complete.

```
spin
    MOV r1,=semaphore
    MOV r2, #1
    SWP r3,r2, [r1]     ;holdthebusuntil complete
    CMP r3, #1
    BEQ  spin
```

The address pointed to by the semaphore either contains the value *0* or *1*. When the semaphore equals *1*, then the service in question is being used by another process. The routine will continue to loop around until the service is released by the other process—in other words, when the semaphore address location contains the value *0*.

SOFTWARE INTERRUPT INSTRUCTION:

A *software interrupt instruction (SWI)* causes a software interrupt exception, which provides a mechanism for applications to call operating system routines.

Syntax: SWI{<cond>} SWI_number

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts)

When the processor executes an SWI instruction, it sets the program counter *pc* to the offset *0x8* in the vector table. The instruction also forces the processor mode to SVC, which allows an operating system routine to be called in a privileged mode.

Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.

Example: Here we have a simple example of an SWI call with SWI number 0x123456, used by ARM toolkits as a debugging SWI. Typically the SWI instruction is executed in user mode.

PRE cpsr=nzcVqift_USER
pc=0x00008000
lr = 0x003fffff ;lr=r14
r0 = 0x12

0x00008000 SWI 0x123456

POST cpsr=nzcVqIft_SVC
spsr=nzcVqift_USER
pc = 0x00000008
lr=0x00008004
r0 = 0x12

Since SWI instructions are used to call operating system routines, you need some form of parameter passing. This is achieved using registers. In this example, register *r0* is used to pass the parameter *0x12*. The return values are also passed back via registers.

Code called the ***SWIhandler*** is required to process the SWI call. The handler obtains the SWI number using the address of the executed instruction, which is calculated from the link register *lr*.

The SWI number is determined by

*SWI_Number=<SWIinstruction>**ANDNOT**(0xff000000)*

Here the *SWI instruction* is the actual 32-bit SWI instruction executed by the processor.

Example: This example shows the start of an SWI handler implementation. The code fragment determines what SWI number is being called and places that number into register *r10*.

You can see from this example that the load instruction first copies the complete SWI instruction into register *r10*. The BIC instruction masks off the top bits of the instruction, leaving the SWI number. We assume the SWI has been called from ARM state.

```
SWI_handler
;Store registers r0-r12 and the link register
        STMFD sp!,{r0–r12,lr}
;Read the SWI  instruction
        LDR r10,[lr,#–4]
;Mask off top 8 bits
        BIC r10,r10,#0xff000000
```

;r10- contains the SWI number

BL service_routine

;return from SWI handler

LDMFD sp!,{r0–r12, pc}^

The number in register *r10* is then used by the SWIhandler to call the appropriate SWI service routine.

PROGRAM STATUS REGISTER INSTRUCTIONS:

The ARM instruction set provides two instructions to directly control a *program status register(psr)*.

- ✓ The *MRS instruction* transfers the contents of either the *cpsr* or *spsr* into a register.
- ✓ The *MSR instruction* transfers the contents of a register into the *cpsr* or *spsr*. Together these instructions are used to read and write the *cpsr* and *spsr*.

In the syntax we can see a *label* called fields. This can be any combination of *control (c)*, *extension (x)*, *status (s)*, and *flags (f)*.

```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```

MRS	copy program status register to a general-purpose register	*Rd = psr*
MSR	move a general-purpose register to a program status register	*psr[field] = Rm*
MSR	move an immediate value to a program status register	*psr[field] = immediate*

These fields relate to particular byte regions in a *psr*, as shown in the following Figure.

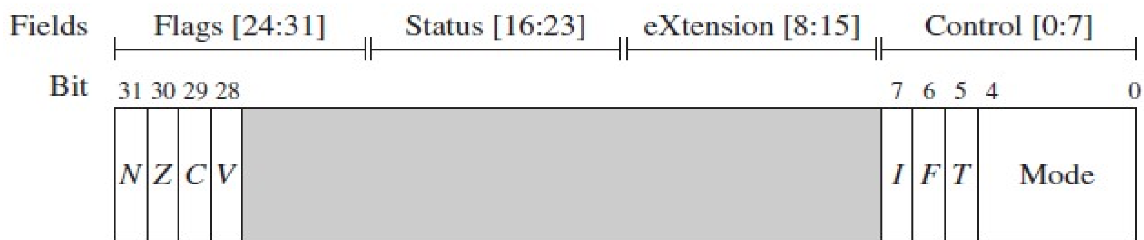


Figure: *psr* Byte Fields

The *c* field controls the interrupt masks, Thumb state, and processor mode.

The following Example shows how to enable IRQ interrupts by clearing the *I* mask. This operation involves using both the MRS and MSR instructions to read from and then write to the *cpsr*.

Example: The MSR first copies the *cpsr* into register *r1*. The BIC instruction clears bit *7* of *r1*.

Register*r1* is then copied back into the *cpsr*, which enables IRQ interrupts. You can see from this example thatthis code preserves all the other settings in the *cpsr* and only modifies the *I* bit in the control field.

PRE *cpsr=nzcvqIFt_SVC*

```
MRS r1,cpsr
BICr1, r1,#0x80     ;0b01000000
MSRcpsr_c,r1
```

POST *cpsr=nzcvqiFt_SVC*

This example is in SVC mode. In user mode you can read all *cpsr* bits, but you can only update the condition flag field *f*.

Coprocessor Instructions:

Coprocessor instructions are used to extend the instruction set.

- ✓ A coprocessor can either provide additional computation capability or be used to control the memory subsystem including caches and memory management.
- ✓ The coprocessor instructions include data processing, register transfer, and memory transfer instructions.

```
Syntax: CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2}
        <MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}
        <LDC|STC>{<cond>} cp, Cd, addressing
```

- ✓ Note that these instructions are only used by cores with a coprocessor.

CDP	coprocessor data processing—perform an operation in a coprocessor
MRC MCR	coprocessor register transfer—move data to/from coprocessor registers
LDC STC	coprocessor memory transfer—load and store blocks of memory to/from a coprocessor

- ✓ In the syntax of the coprocessor instructions,
  - o The *cp* field represents the coprocessor number between *p0* and *p15*
  - o The *op code* fields describe the operation to take place on the coprocessor.
  - o The *Cn*, *Cm*, and *Cd* fields describe registers within the coprocessor.
- ✓ The coprocessor operations and registers depend on the specific coprocessor you are using.
- ✓ *Coprocessor 15 (CP15)* is reserved for system control purposes, such as memory management, write buffer control, cache control, and identification registers.

Example: This example shows a *CP15* register being copied into a general-purpose register.

; transferring the contents of CP15 register c0 to register r10

MRCp15,0, r10, c0, c0, 0

Here *CP15 register-0* contains the processor identification number. This register is copied into the general-purpose register *r10*.

LOADING CONSTANTS:

You might have noticed that there is no ARM instruction to move a 32-bit constant into a register. Since ARM instructions are 32 bits in size, they obviously cannot specify a general 32-bit constant. To aid programming there are two pseudo-instructions to move a 32-bit value into a register.

```
Syntax: LDR Rd, =constant
        ADR Rd, label
```

LDR	load constant pseudoinstruction	*Rd* = 32-bit constant
ADR	load address pseudoinstruction	*Rd* = 32-bit relative address

- The first pseudo-instruction writes a 32-bit constant to a register using whatever instructions are available. It defaults to a memory read if the constant cannot be encoded using other instructions.
- The second pseudo- instruction writes a relative address into a register, which will be encoded using a pc-relative expression.

Example: This example shows an LDR instruction loading a 32-bit constant *0xff00ffff* into register *r0*.

LDR r0,[pc,#constant_number-8-{PC}]

:

constant_number

DCD 0xff00ffff

This example involves a memory access to load the constant, which can be expensive for time-critical routines.

The following Example shows an alternative method to load the same constant into register *r0* by using an MVN instruction.

Example: Loading the constant *0xff00ffff* using an MVN

PRE *none...*

MVN r0,#0x00ff0000

POST *r0=0xff00ffff*

As you can see, there are alternatives to accessing memory, but they depend upon the constant you are trying to load.

The LDR pseudo-instruction either inserts an MOV or MVN instruction to generate a value (if possible) or generates an LDR instruction with a *pc*-relative address to read the constant from a literal pool—a data area embedded within the code.

The following Table shows two pseudo-code conversions.

Table: LDR pseudo-instruction Conversion

Pseudoinstruction	Actual instruction
LDR r0, =0xff	MOV r0, #0xff
LDR r0, =0x55555555	LDR r0, [pc, #offset_12]

The first conversion produces a simple MOV instruction; the second conversion produces a *pc*-relative load. Another useful pseudo-instruction is the ADR instruction, or address relative. This instruction places the address of the given label into register *Rd*, using a *pc*-relative add or subtract.