# Embedded System Design

## (R22D6802)

# DIGITAL NOTES

# M.TECH

# (I YEAR – I SEM)

# (2023-24)

## Department of Electronics and Communication Engineering

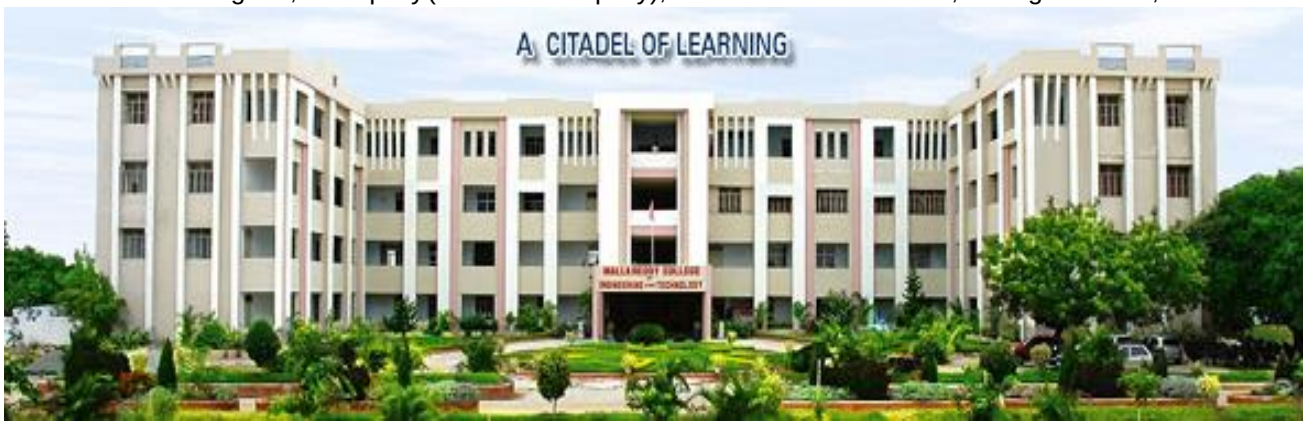

**MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY**

**(Autonomous Institution - UGC, Govt. of India)**

Recognized under 2(f) and 12 (B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Kompally), Secunderabad – 500100, Telangana State, India

## EMBEDDED SYSTEM DESIGN

**Course Objectives:**

- Discuss the basic principles of ARM system design.
- Identify the major hardware components ARM data path architecture.
- Identify the design issues ARM based embedded system with the basic knowledge of firmware, embedded OS & ARM architectures.
- Analyze the execution of instructions/program knowing the basic principles of ARM architecture and assembly language.
- Compare programs written in C & assembly to execute on ARM platform.

**UNIT –I:**

**ARM Architecture:**

ARM Design Philosophy, Registers, Program Status Register, Instruction Pipeline, Interrupts and Vector Table, Architecture Revision, ARM Processor Families.

**UNIT –II:**

**ARM Programming Model – I:**

Instruction Set: Data Processing Instructions, Addressing Modes, Branch, Load, Store Instructions, PSR Instructions, Conditional Instructions.

**UNIT –III:**

**ARM Programming Model – II:**

Thumb Instruction Set: Register Usage, Other Branch Instructions, Data Processing Instructions, Single-Register and Multi Register Load-Store Instructions, Stack, Software Interrupt Instructions

**UNIT –IV:**

**ARM Programming:**

Simple C Programs using Function Calls, Pointers, Structures, Integer and Floating Point Arithmetic, Assembly Code using Instruction Scheduling, Register Allocation, Conditional Execution and Loops.

**UNIT –V:**
**Memory Management:**
Cache Architecture, Polices, Flushing and Caches, MMU, Page Tables, Translation, Access Permissions, Context Switch.

**TEXT BOOKS:**

1. ARM Systems Developer's Guides- Designing & Optimizing System Software – Andrew N. Sloss, Dominic Symes, Chris Wright, 2008, Elsevier.
2. Professional Embedded ARM development-James A Langbridge, Wiley/Wrox

**REFERENCE BOOKS:**

1. Embedded Microcomputer Systems, Real Time Interfacing – Jonathan W. Valvano – Brookes / Cole, 1999, Thomas Learning.
2.ARM System on Chip Architecture, Steve Furber, 2nd Edition, Pearson

**Course Outcomes:**

- Become aware of the ARM Processor, Architecture, Registers, Instruction pipeline, Interrupts and Instructions, Addressing modes and conditional instructions.
- Apply and analyze the applications in various processors and domains of embedded system.
- Ability to use advanced controllers using thumb instruction for embedded system design.
- Analyze and develop embedded hardware and software development cycles and tools.
- Understanding the concept of Memory management unit, integration methods and hardwareand software design concepts associated with processor in Embedded Systems.

# UNIT-I

# ARM ARCHITECTURE

## ARM

- ARM, previously Advanced RISC Machine, originally Acorn RISC Machine, is a family of Reduced Instruction Set Computing (RISC) Architecture for Computer Processors.
- The ARM processor core is key component of many successful 32-bit embedded systems.

## The RISC design philosophy

The design philosophy aimed at delivering the following.

- simple but powerful instructions
- single cycle execution at a high clock speed
- intelligence in software rather than hardware
- Provide greater flexibility on reducing the complexity of instructions.

The ARM core uses RISC architecture.

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 (a divide operation) by combining several simple instructions. Each instruction is a fixed length to allow the pipeline to fetch future instructions before decoding the current instruction. In contrast, in CISC processors the instructions are often of variable size and take many cycles to execute.
2. **Pipelines** —The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines. Ideally the pipeline advances by one step on each cycle for maximum throughput. There is no need for an instruction to be executed by a mini program called microcode as on CISC processors.
3. **Registers**—RISC machines have a large general-purpose register set. Any register can contain either data or an address. 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. In contrast, with a CISC design the data processing operations can act on memory directly.

## The ARM Design Philosophy

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

1. Small to reduce power consumption and extend battery operation
2. High code density
3. Price sensitive and use slow and low-cost memory devices.
4. Reduce the area of the die taken up by the embedded processor.
5. Hardware debug technology
6. ARM core is not a pure RISC architecture

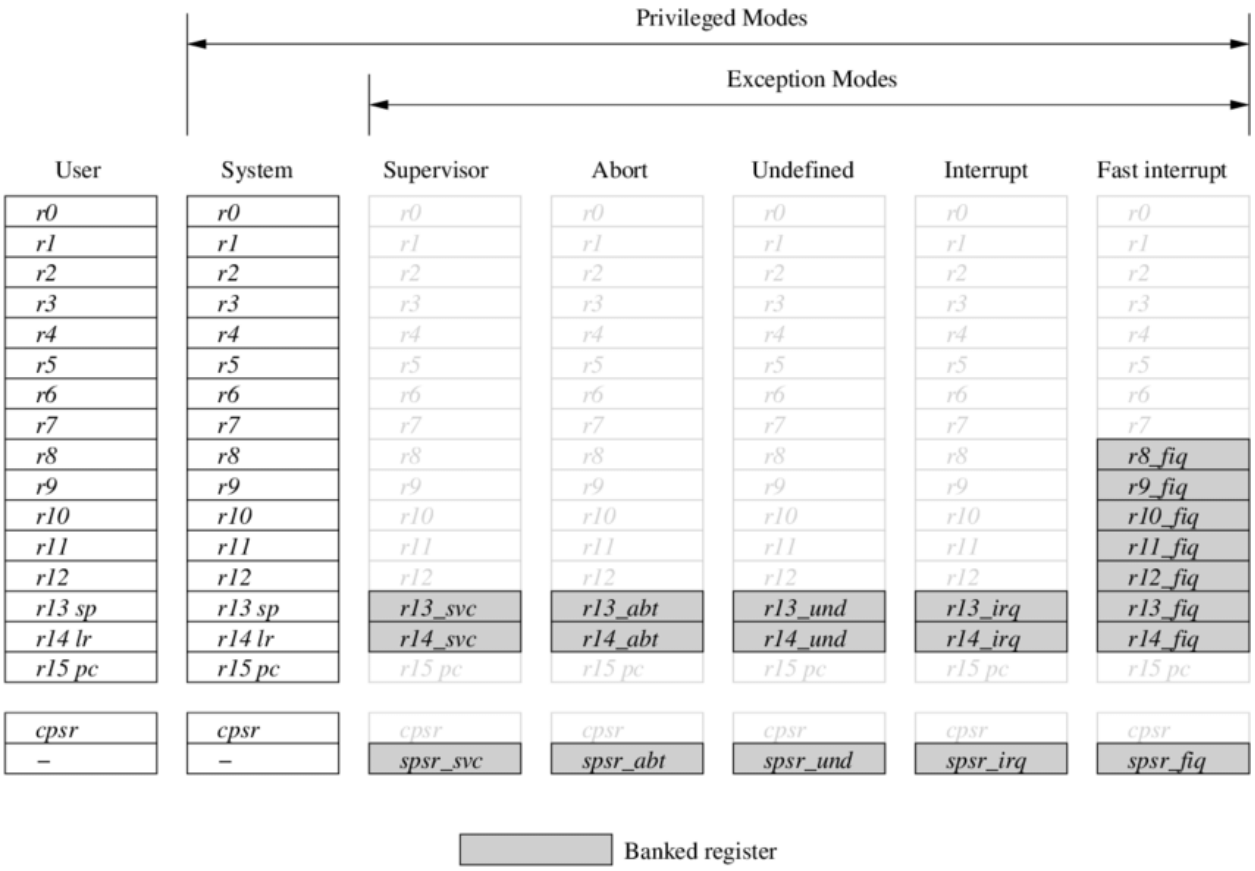
## Registers:

ARM processors provide general-purpose and special-purpose registers. Some additional registers are available in privileged execution modes.

In all ARM processors, the following registers are available and accessible in any processor mode:

- 13 general-purpose registers R0-R12.
- One *Stack Pointer* (SP).
- One *Link Register* (LR).
- One *Program Counter* (PC).
- One *Application Program Status Register* (APSR).

The amount of registers depends on the ARM version. According to the ARM Reference Manual, there are 30 general-purpose 32-bit registers, with the exception of ARMv6-M and ARMv7-M based processors. The first 16 registers are accessible in user-level mode, the additional registers are available in privileged software execution (with the exception of ARMv6-M and ARMv7-M). In this tutorial series we will work with the registers that are accessible in any privilege mode: r0-15. These 16 registers can be split into two groups: general purpose and special purpose registers.




**R0-R12**: can be used during common operations to store temporary values, pointers (locations to memory), etc. R0, for example, can be referred as accumulator during the arithmetic operations or for storing the result of a previously called function. R7 becomes useful while working with syscalls as it stores the syscall number and R11 helps us to keep track of boundaries on the stack serving as the frame pointer (will be covered later). Moreover, the function calling convention on ARM specifies that the first four arguments of a function are stored in the registers r0-r3.

**R13: SP** (Stack Pointer). The Stack Pointer points to the top of the stack. The stack is an area of memory used for function-specific storage, which is reclaimed when the function returns. The stack pointer is therefore used for allocating space on the stack, by subtracting the value (in bytes) we want to allocate from the stack pointer. In other words, if we want to allocate a 32 bit value, we subtract 4 from the stack pointer.

**R14: LR** (Link Register). When a function call is made, the Link Register gets updated with a memory address referencing the next instruction where the function was initiated from. Doing this allows the program return to the “parent” function that initiated the “child” function call after the “child” function is finished.

**R15: PC** (Program Counter). The Program Counter is automatically incremented by the size of the instruction executed. This size is always 4 bytes in ARM state and 2 bytes in THUMB mode.

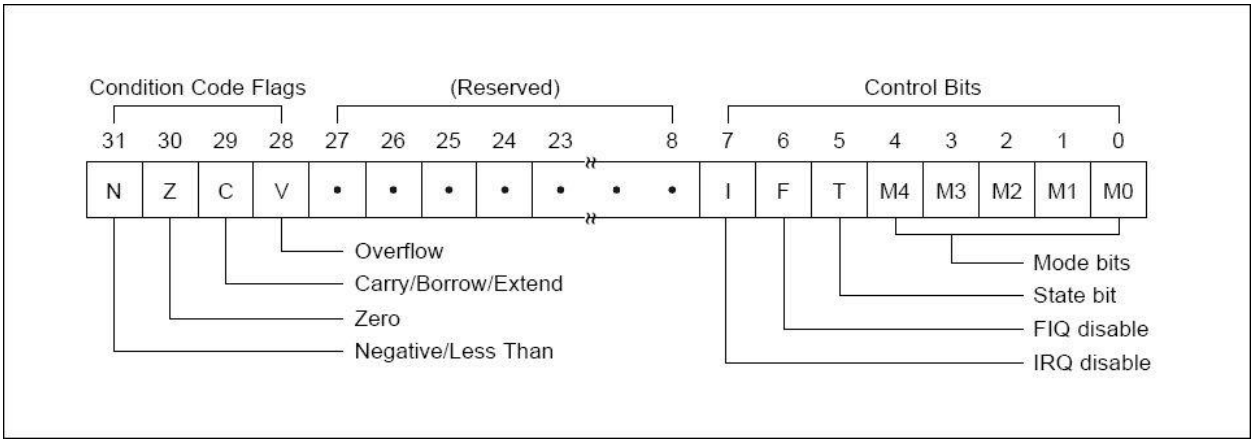
When a branch instruction is being executed, the PC holds the destination address. During execution, PC stores the address of the current instruction plus 8 (two ARM instructions) in ARM state, and the current instruction plus 4 (two Thumb instructions) in Thumb(v1) state. This is different from x86 where PC always points to the next instruction to be executed.

## Current Program Status Register

The *Current Program Status Register* (CPSR) holds the same program status flags as the APSR, and some additional information.

The CPSR holds:

- The APSR flags.
- The processor mode.
- The interrupt disable flags.
- The instruction set state (ARM, Thumb, ThumbEE, or Jazelle®).
- The endianness state (on ARMv4T and later).
- The execution state bits for the IT block (on ARMv6T2 and later).




The Current Program Status Register is a 32-bit wide register used in the ARM architecture to record various pieces of information regarding the state of the program being executed by the processor and the state of the processor. This information is recorded by setting or clearing specific bits in the register.

The top four bits (bits 31, 30, 29, and 28) are the condition code (**cc**) bits and are of most interest to us. Condition code bits are sometimes referred to as "flags". The lowest 8 bits (bit 7 through to bit 0) store information about the processor's own state. The remaining bits (i.e. bit 27 to bit 8) are currently unused in most ARM processors.

The N bit is the "negative flag" and indicates that a value is negative.

The Z bit is the "zero flag" and is set when an appropriate instruction produces a zero result.

The C bit is the "carry flag" but it can also be used to indicate "borrows" (from subtraction operations) and "extends" (from shift instructions (LINK)).

The V bit is the "overflow flag" which is set if an instruction produces a result that overflows and hence may go beyond the range of numbers that can be represented in 2's complement signed format.

For completeness, the other state bits are:

The I and F bits which determine whether interrupts (such as requests for input/output) are enabled or disabled.

The T bit which indicates whether the processor is in "Thumb" mode, where the processor can execute a subset of the assembly language as 16-bit compact instructions. As Thumb code packs more instructions into the same amount of memory, it is an effective solution to applications where physical memory is at a premium.

The M4 to M0 bits are the mode bits. Application programs normally run in user mode (where the mode bits are 10000). Whenever an interrupt or similar event occurs, the processor switches into one of the alternative modes allowing the software handler greater privileges with regard to memory manipulation.

| M[4:0] | Mode | Accessible registers |
|---|---|---|
| 10000 | User | PC, R14 to R0, CPSR |
| 10001 | FIQ | PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq |
| 10010 | IRQ | PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq |
| 10011 | Supervisor | PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc |
| 10111 | Abort | PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt |
| 11011 | Undefined | PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und |
| 11111 | System | PC, R14 to R0, CPSR |

## The instruction pipeline

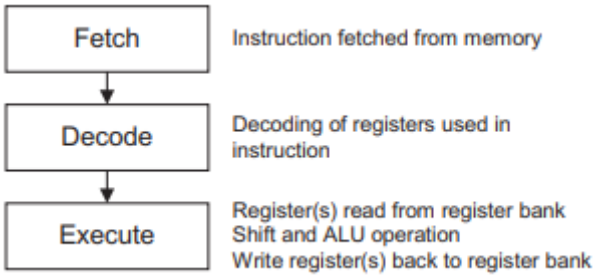
The ARM uses a pipeline to increase the speed of the flow of instructions to the processor. This allows several operations to take place simultaneously, and the processing, and memory systems to operate continuously.

A three-stage pipeline is used, so instructions are executed in three stages:

- Fetch

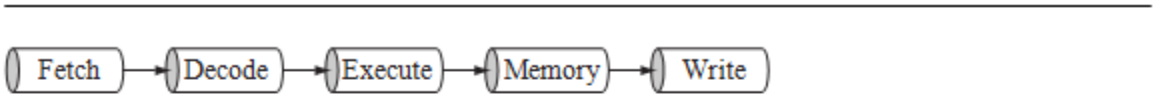
- Decode
- Execute.

The three-stage pipeline is shown in




**The instruction pipeline**

During normal operation, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory. The program counter points to the instruction being fetched rather than to the instruction being executed. This is important because it means that the Program Counter (PC) value used in an executing instruction is always two instructions ahead of the address.




ARM9 five-stage pipeline.


ARM10 six-stage pipeline.

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

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

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

## Interrupts and the Vector Table.

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

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

When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see Table 2.6). Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:

**Reset vector** is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.

**Undefined instruction vector** is used when the processor cannot decode an instruction.

**Software interrupt vector** is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.

**Prefetch** abort vector occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.

**Data** abort vector is similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions.

**Interrupt request** vector is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the cpsr.

**The vectortable.**

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

# Architecture Revision

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

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

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

# Nomenclature

ARM uses the nomenclature shown in below Figure to describe the processor implemen- tations. The letters and numbers after the word “ARM” indicate the features a processor

---

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

x—family

y—memory management/protection unit

z—cache

T—Thumb 16-bit decoder

D—JTAG debug

M—fast multiplier

I—EmbeddedICE macrocell

E—enhanced instructions (assumes TDMI)

J—Jazelle F—vector floating-point unit

S—synthesizible version

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

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

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

## Architecture Evolution

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

The various parts of the program status register and the availabil- ity of certain features on particular instruction architectures. "All" refers to the ARMv4 architecture and above.

## ARM PROCESSOR FAMILIES

ARM has designed a number of processors that are grouped into different families according to the core they use. The families are based on the ARM7, ARM9, ARM10, and ARM11 cores. The postfix numbers 7, 9, 10, and 11 indicate different core designs. The ascending number equates

to an increase in performance and sophistication. ARM8 was developed but was soon superseded.

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

Revision history.

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

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

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

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

Description of the *cpsr*.

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

ARM family attribute comparison.

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

[a] Watts/MHz on the same 0.13 micron process.
[b] MIPS are Dhrystone VAX MIPS.

## ARM7 Family

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

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

ARM processor variants.

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

[a]E extension provides enhanced multiply instructions and saturation.

One significant variation in the ARM7 family is the ARM7TDMI-S. The ARM7TDMI-S has the same operating characteristics as a standard ARM7TDMI but is also synthesizable. ARM720T is the most flexible member of the ARM7 family because it includes an MMU. The presence of the MMU means the ARM720T is capable of handling the Linux and Microsoft embedded platform operating systems. The processor also includes a unified 8K cache. The vector table can be relocated to a higher address by setting a coprocessor 15 register.

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

## ARM9 FAMILY

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

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

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

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

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

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

## ARM10 Family

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

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

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

## ARM11 Family

The ARM1136J-S, announced in 2003, was designed for high performance and power- efficient applications. ARM1136J-S was the first processor implementation to execute architecture ARMv6 instructions. It incorporates an eight-stage pipeline with separate load- store and

arithmetic pipelines. Included in the ARMv6 instructions are single instruction multiple data (SIMD) extensions for media processing, specifically designed to increase video processing performance.

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

## Specialized Processors

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

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

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

UNIT-II

# ARM Programming Model – I

ARM instructions process data held in registers and only access memory with load and store instructions. ARM instructions commonly take two or three operands. For instance the ADD instruction below adds the two values stored in 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* |

In the following sections we examine the function and syntax of the ARM instructions by instruction class—data processing instructions, branch instructions,

load-store instructions, software interrupt instruction, and program status register instructions.

## Data Processing Instructions

The data processing instructions manipulate data within registers. They are move instruc- tions, arithmetic instructions, logical instructions, comparison instructions, and multiply instructions. Most data processing 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. The carry flag is set from the result of the barrel shift as the last bit shifted out. The N flag is set to bit 31 of the result. The Z flag is set if the result is zero.

## Move Instructions

Move is the simplest ARM instruction. It copies N into a destination register Rd, where N is a register or immediate value. This instruction is useful for setting initial values and transferring data between registers.

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 = - N |

Gives a full description of the values allowed for the second operand N for all data processing instructions. Usually it is a register Rm or a constant preceded by #.

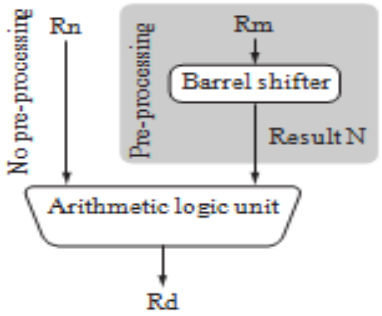
## Barrel Shifter

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 within 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. This shift increases the power and flexibility of many data processing operations.

There are data processing instructions that do not use the barrel shift, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions.

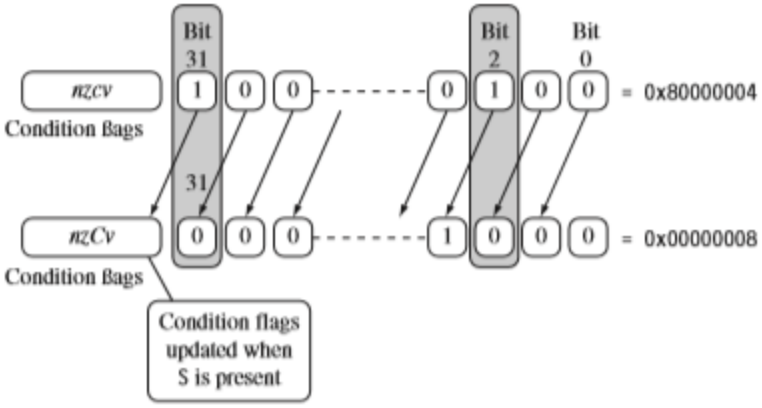
Pre-processing or shift occurs within the cycle time of the instruction. This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.



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* | (unsigned)$x \gg y$ | #1–32 or *Rs* |
| ASR | arithmetic right shift | *x*ASR *y* | (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.




Barrel shift operation syntax for data processing instructions.

| *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 This simple subtract instruction subtracts a value 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
```

## 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. illustrates the use of the inline 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 = 0x00000005

       ADD     r0, r1, r1, LSL #1

POST   r0 = 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 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 AND NOT(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*.

The logical instructions update the *cpsr* flags only if the S suffix is present. These instructions can use barrel-shifted 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. For more information on conditional execution take a look at Section 3.8. You do not need 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 $Rn \text{ ^ } N$ |
| TST | test bits of a 32-bit value | flags set as a result of $Rn \& N$ |

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 is represented 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 result discarded; similarly the TST 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*. It is important to understand that comparison instructions only modify the condition flags of the *cpsr* and do not affect the registers being compared.

## 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 onto a pair of registers representing a 64-bit value. The final result is 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 | $Rd = Rm * Rs$ |

```
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 | $[RdHi, RdLo] = Rm * Rs$ |

## 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 ARMv5E instruction set includes four different branch instructions.

Syntax: B{<cond>} label

BL{<cond>} label

BX{<cond>} Rm

BLX{<cond>} label | Rm

| | | |
|---|---|---|
| B | branch | pc = label |
| BL | branch with link | pc = label<br>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<br>pc = Rm & 0xfffffffe, T = Rm &1<br>lr = address of the next instruction after the BLX |

The address label is stored in the instruction as a signed pc-relative offset and must be within approximately 32 MB of the branch instruction. T refers to the Thumb bit in the cpsr. When instructions set T, the ARM switches to Thumb 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. The backward branch creates an infinite loop.

```
        B     forward
        ADD   r1, r2, #4
        ADD   r0, r6, #2
        ADD   r3, r7, #4
forward
        SUB   r1, r2, #4
-------------------------
backward
        ADD  r1,  r2,  #4
        SUB  r1,  r2,  #4
        ADD r4, r6, r7
        B  backward
```

Branches are used to change execution flow. Most assemblers hide the details of a branch instruction encoding by using labels. 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.

## 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 datatypes supported are signed and unsigned words (32-bit), halfwords (16-bit), and bytes. Here are the various load-store single-register transfer instructions.

Syntax: <LDR|STR>{<cond>}{B} Rd, addressing[1]
LDR{<cond>}SB|H|SH Rd, addressing[2]
STR{<cond>}H Rd, addressing[2]

| | | |
|---|---|---|
| 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])* |

## SINGLE-REGISTER LOAD-STORE ADDRESSING MODES

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods: preindex with writeback, preindex, and postindex

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.

EXAMPLE Preindex with writeback calculates an address from a base register plus address offset and then updates that address base register with the new address. In contrast, the preindex offset is the same as the preindex with writeback but does not update the address base register. Postindex only updates the address base register after the address is used. The preindex mode is useful for accessing an element in a data structure. The postindex and preindex with writeback modes are useful for traversing an array.

```
PRE       r0 = 0x00000000
          r1 = 0x00090000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202

          LDR     r0, [r1, #4]!
```

Preindexing with writeback:

```
POST(1)   r0 = 0x02020202
          r1 = 0x00009004

          LDR     r0, [r1, #4]
```

Preindexing:

```
POST(2)   r0 = 0x02020202
          r1 = 0x00009000

          LDR     r0, [r1], #4
```

Postindexing:

```
POST(3)   r0 = 0x01010101
          r1 = 0x00009004
```

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` |

## 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 into memory. 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.

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)` |

Table Single-register load-store addressing, halfword, signed halfword, signed byte, and doubleword.

| 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` |

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` |

Load-store multiple instructions can increase interrupt latency. ARM implementations do not usually interrupt instructions while they are executing. For example, on an ARM7 a 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 |

Table 3.9 shows the different addressing modes for the load-store multiple instructions. Here $N$ is the number of registers in the list of registers.

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, similiar to the single-register load-store using preindex with writeback.

Table Addressing mode for load-store multiple instructions.

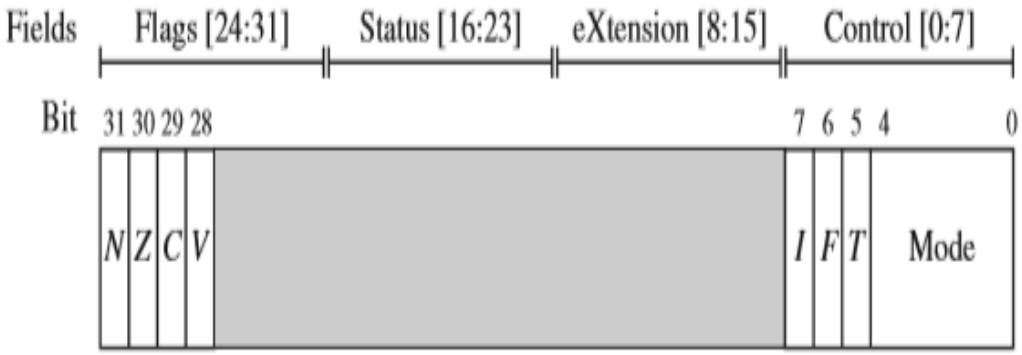
| Addressing mode | Description | Start address | End address | *Rn*! |
|---|---|---|---|---|
| IA | increment after | $Rn$ | $Rn + 4*N - 4$ | $Rn + 4*N$ |
| IB | increment before | $Rn + 4$ | $Rn + 4*N$ | $Rn + 4*N$ |
| DA | decrement after | $Rn - 4*N + 4$ | $Rn$ | $Rn - 4*N$ |
| DB | decrement before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

## 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; in the reverse direction, 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 you can see a label called *fields*. This can be any combination of control (*c*), extension (*x*), status (*s*), and flags (*f*). These fields relate to particular byte regions in a *psr*, as shown in Figure 3.9.

```
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* |

The *c* field controls the interrupt masks, Thumb state, and processor mode. Example 3.26 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 that this 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
       BIC    r1, r1, #0x80 ; 0b01000000
       MSR    cpsr_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*.

## CONDITIONAL EXECUTION

Most ARM instructions are conditionally executed—you can specify that the instruction only executes if the condition code flags pass a given condition or test. By using conditional execution instructions you can increase performance and code density.

The condition field is a two-letter mnemonic appended to the instruction mnemonic.

The default mnemonic is AL, or always execute.

Conditional execution reduces the number of branches, which also reduces the number of pipeline flushes and thus improves the performance of the executed code. Conditional execution depends upon two components: the condition field and condition flags. The condition field is located in the instruction, and the condition flags are located in the cpsr.

EXAMPLE This example shows an ADD instruction with the EQ condition appended. This instruction will only be executed when the zero flag in the *cpsr* is set to 1.

```
; r0 = r1 + r2 if zero flag is set
ADDEQ r0, r1, r2
```

Only comparison instructions and data processing instructions with the S suffix appended to the mnemonic update the condition flags in the *cpsr*.

# Unit-III

# ARM Programming Model – II

## Thumb Instruction Set

Thumb encodes a subset of the 32-bit ARM instructions into a 16-bit instruction set space. Since Thumb has higher performance than ARM on a processor with a 16-bit data bus, but lower performance than ARM on a 32-bit data bus, use Thumb for memory-constrained systems.

Thumb has higher code density—the space taken up in memory by an executable program—than ARM. For memory-constrained embedded systems, for example, mobile phones and PDAs, code density is very important. Cost pressures also limit memory size, width, and speed.

On average, a Thumb implementation of the same code takes up around 30% less memory than the equivalent ARM implementation. As an example, the same divide code routine implemented in ARM and Thumb assembly code. Even though the Thumb implementation uses more instructions, the overall memory footprint is reduced. Code density was the main driving force for the Thumb instruction set. Because it was also designed as a compiler target, rather than for hand-written assembly code, we recommend that you write Thumb-targeted code in a high-level language like C or C++.

Each Thumb instruction is related to a 32-bit ARM instruction. A simple Thumb ADD instruction being decoded into an equivalent ARM ADD instruction. Only the branch relative instruction can be conditionally executed. The limited space available in 16 bits causes the barrel shift operations ASR, LSL, LSR, and ROR to be separate instructions in the Thumb ISA.

**ARM code**

```
ARMDivide
; IN:  r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

        MOV     r3,#0
loop
        SUBS    r0,r0,r1
        ADDGE   r3,r3,#1
        BGE     loop
        ADD     r2,r0,r1
```

$5 \times 4 = 20$ bytes

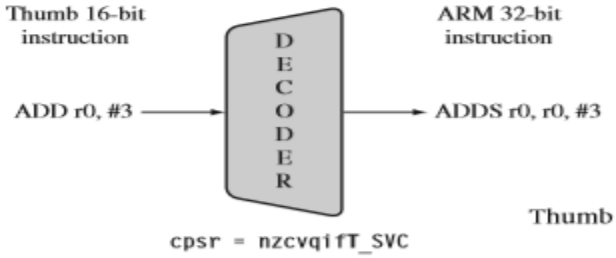
**Thumb code**

```
ThumbDivide
; IN:  r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

        MOV     r3,#0
loop
        ADD     r3,#1
        SUB     r0,r1
        BGE     loop
        SUB     r3,#1
        ADD     r2,r0,r1
```

$6 \times 2 = 12$ bytes

Code density.




Thumb instruction decoding.

| Mnemonics | THUMB 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 |
| ASR | v1 | arithmetic shift right |
| B | v1 | branch relative |
| BIC | v1 | logical bit clear (AND NOT) of two 32-bit values |
| BKPT | v2 | breakpoint instructions |
| BL | v1 | relative branch with link |
| BLX | v2 | branch with link and exchange |
| BX | v1 | branch with exchange |
| CMN | v1 | compare negative two 32-bit values |
| CMP | v1 | compare two 32-bit integers |
| EOR | v1 | logical exclusive OR of two 32-bit values |
| LDM | v1 | load multiple 32-bit words from memory to ARM registers |
| LDR | v1 | load a single value from a virtual address in memory |
| LSL | v1 | logical shift left |
| LSR | v1 | logical shift right |
| MOV | v1 | move a 32-bit value into a register |
| MUL | v1 | multiply two 32-bit values |
| MVN | v1 | move the logical NOT of 32-bit value into a register |
| NEG | v1 | negate a 32-bit value |
| ORR | v1 | logical bitwise OR of two 32-bit values |
| POP | v1 | pops multiple registers from the stack |
| PUSH | v1 | pushes multiple registers to the stack |
| ROR | v1 | rotate right a 32-bit value |
| SBC | v1 | subtract with carry a 32-bit value |
| STM | v1 | store multiple 32-bit registers to memory |
| STR | v1 | store register to a virtual address in memory |
| SUB | v1 | subtract two 32-bit values |
| SWI | v1 | software interrupt |
| TST | v1 | test bits of a 32-bit value |

**Thumb instruction set.**

## THUMB REGISTER USAGE

In Thumb state, you do not have direct access to all registers. Only the low registers r0 to r7 are fully accessible, as shown in below Table 4.2. The higher registers r8 to r12 are only accessible with MOV, ADD, or CMP instructions. CMP and all the data processing instructions that operate on low registers update the condition flags in the cpsr.

Summary of Thumb register usage.

| Registers | Access |
|---|---|
| *r0-r7* | fully accessible |
| *r8-r12* | only accessible by MOV, ADD, and CMP |
| *r13 sp* | limited accessibility |
| *r14 lr* | limited accessibility |
| *r15 pc* | limited accessibility |
| *cpsr* | only indirect access |
| *spsr* | no access |

You may have noticed from the Thumb instruction set list and from the Thumb register usage table that there is no direct access to the cpsr or spsr. In other words, there are no MSR- and MRS-equivalent Thumb instructions.

To alter the cpsr or spsr, you must switch into ARM state to use MSR and MRS. Similarly, there are no coprocessor instructions in Thumb state. You need to be in ARM state to access the coprocessor for configuring cache and memory management.

## OTHER BRANCH INSTRUCTIONS

There are two variations of the standard branch instruction, or B. The first is similar to the ARM version and is conditionally executed; the branch range is limited to a signed 8-bit immediate, or 256 to−+254 bytes. The second version removes the conditional part of the instruction and expands the effective branch range to a signed 11-bit immediate, or 2048 to +2046 bytes.

The conditional branch instruction is the only conditionally executed instruction in Thumb state.

```
Syntax: B<cond> label
        B label
        BL label
```

| | | |
|---|---|---|
| B | branch | *pc* = *label* |
| BL | branch with link | *pc* = *label*<br>*lr* = (instruction address after the BL) + 1 |

The BL instruction is not conditionally executed and has an approximate range of +/−4 MB. This range is possible because BL (and BLX) instructions are translated into a pair of 16-bit Thumb instructions. The first instruction in the pair holds the high part of the branch offset, and the second the low part. These instructions must be used as a pair.

The code here shows the various instructions used to return from a BL subroutine call:

```
        MOV     pc, lr
        BX      lr
        POP     {pc}
```

To return, we set the *pc* to the value in *lr*. The stack instruction called POP

# Data Processing Instructions

The data processing instructions manipulate data within registers. They include move instructions, arithmetic instructions, shifts, logical instructions, comparison instructions, and multiply instructions. The Thumb data processing instructions are a subset of the ARM data processing instructions.

Syntax:

```
<ADC|ADD|AND|BIC|EOR|MOV|MUL|MVN|NEG|ORR|SBC|SUB>  Rd, Rm
<ADD|ASR|LSL|LSR|ROR|SUB> Rd, Rn #immediate
<ADD|MOV|SUB> Rd,#immediate
<ADD|SUB> Rd,Rn,Rm
 ADD Rd,pc,#immediate
 ADD Rd,sp,#immediate
<ADD|SUB> sp, #immediate
<ASR|LSL|LSR|ROR> Rd,Rs
<CMN|CMP|TST> Rn,Rm
 CMP Rn,#immediate
 MOV Rd,Rn
```

| | | |
|---|---|---|
| ADC | add two 32-bit values and carry | $Rd = Rd + Rm + C$ flag |
| ADD | add two 32-bit values | $Rd = Rn + immediate$<br>$Rd = Rd + immediate$<br>$Rd = Rd + Rm$<br>$Rd = Rd + Rm$<br>$Rd = (pc \,\&\, 0xfffffffc) + (immediate \ll 2)$<br>$Rd = sp + (immediate \ll 2)$<br>$sp = sp + (immediate \ll 2)$ |

These instructions follow the same style as the equivalent ARM instructions. Most Thumb data processing instructions operate on low registers and update the *cpsr*. The exceptions are

```
MOV     Rd,Rn
ADD     Rd,Rm
CMP     Rn,Rm
ADD     sp, #immediate
SUB     sp, #immediate
ADD     Rd,sp,#immediate
ADD     Rd,pc,#immediate
```

which can operate on the higher registers *r8–r14* and the *pc*. These instructions, except for CMP, do not update the condition flags in the *cpsr* when using the higher registers. The CMP instruction, however, always updates the *cpsr*.

## Single-Register Load-Store Instructions

The Thumb instruction set supports load and storing registers, or LDR and STR. These instructions use two preindexed addressing modes: offset by register and offset by immediate.

```
Syntax: <LDR|STR>{<B|H>} Rd, [Rn,#immediate]
        LDR{<H|SB|SH>} Rd,[Rn,Rm]
        STR{<B|H>} Rd,[Rn,Rm]
        LDR Rd,[pc,#immediate]
        <LDR|STR> Rd,[sp,#immediate]
```

| | | |
|---|---|---|
| LDR | load word into a register | *Rd <- mem32[address]* |
| STR | save 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])* |

You can see the different addressing modes in Table . The offset by register uses a base register *Rn* plus the register offset *Rm*. The second uses the same base register *Rn* plus a 5-bit immediate or a value dependent on the data size. The 5-bit offset encoded in the instruction is multiplied by one for byte accesses, two for 16-bit accesses, and four for 32-bit accesses.

Table Addressing modes.

| Type | Syntax |
|---|---|
| Load/store register | `[Rn, Rm]` |
| Base register + offset | `[Rn, #immediate]` |
| Relative | `[pc\|sp, #immediate]` |

## Multiple-Register Load-Store Instructions

The Thumb versions of the load-store multiple instructions are reduced forms of the ARM load-store multiple instructions. They only support the increment after (IA) addressing mode.

```
Syntax : <LDM|STM>IA Rn!, {low Register list}
```

| | | |
|---|---|---|
| LDMIA | load multiple registers | {Rd}$^{*N}$ <- mem32[Rn + 4*N], Rn = Rn + 4*N |
| STMIA | save multiple registers | {Rd}$^{*N}$ -> mem32[Rn + 4*N], Rn = Rn + 4*N |

Here *N* is the number of registers in the list of registers. You can see that these instructions always update the base register *Rn* after execution. The base register and list of registers are limited to the low registers *r0* to *r7*.

## Stack Instructions

The Thumb stack operations are different from the equivalent ARM instructions because they use the more traditional POP and PUSH concept.

```
Syntax: POP {low_register_list{, pc}}
        PUSH {low_register_list{, lr}}
```

| | | |
|---|---|---|
| POP | pop registers from the stacks | Rd$^{*N}$ <- mem32[sp + 4*N], sp = sp + 4*N |
| PUSH | push registers on to the stack | Rd$^{*N}$ -> mem32[sp + 4*N], sp = sp − 4*N |

The interesting point to note is that there is no stack pointer in the instruction. This is because the stack pointer is fixed as register *r13* in Thumb operations and *sp* is automatically updated. The list of registers is limited to the low registers *r0* to *r7*.

The PUSH register list also can include the link register *lr*; similarly the POP register list can include the *pc*. This provides support for subroutine entry and exit,

The stack instructions only support full descending stack operations.

## Software Interrupt Instruction

Similar to the ARM equivalent, the Thumb software interrupt (SWI) instruction causes a software interrupt exception. If any interrupt or exception flag is raised in Thumb state, the processor automatically reverts back to ARM state to handle the exception.

```
Syntax: SWI immediate
```

| | | |
|---|---|---|
| SWI | software interrupt | *lr_svc* = address of instruction following the SWI<br>*spsr_svc* = *cpsr*<br>*pc* = vectors + 0x8<br>*cpsr* mode = *SVC*<br>*cpsr I* = 1 (mask IRQ interrupts)<br>*cpsr T* = 0 (ARM state) |

The Thumb SWI instruction has the same effect and nearly the same syntax as the ARM equivalent. It differs in that the SWI number is limited to the range 0 to 255 and it is not conditionally executed.

# UNIT –IV

# ARM Programming

## BASIC C DATA TYPES

There are also differences between the addressing modes available when loading and storing data of each type.

ARM processors have 32-bit registers and 32-bit data processing operations. The ARM architecture is a RISC load/store architecture. In other words you must load values from memory into registers before acting on them. There are no arithmetic or logical instructions that manipulate values in memory directly.

Early versions of the ARM architecture (ARMv1 to ARMv3) provided hardware support for loading and storing unsigned 8-bit and unsigned or signed 32-bit values.

Load and store instructions by ARM architecture.

| Architecture | Instruction | Action |
|---|---|---|
| Pre-ARMv4 | LDRB | load an unsigned 8-bit value |
| | STRB | store a signed or unsigned 8-bit value |
| | LDR | load a signed or unsigned 32-bit value |
| | STR | store a signed or unsigned 32-bit value |
| ARMv4 | LDRSB | load a signed 8-bit value |
| | LDRH | load an unsigned 16-bit value |
| | LDRSH | load a signed 16-bit value |
| | STRH | store a signed or unsigned 16-bit value |
| ARMv5 | LDRD | load a signed or unsigned 64-bit value |
| | STRD | store a signed or unsigned 64-bit value |

These architectures were used on processors prior to the ARM7TDMI. The load/store instruction classes available by ARM architecture.

In loads that act on 8- or 16-bit values extend the value to 32 bits before writing to an ARM register. Unsigned values are zero-extended, and signed values sign-extended. This means that the cast of a loaded value to an int type does not cost extra instructions. Similarly, a store of an 8- or 16-bit value selects the lowest 8 or 16 bits of the register. The cast of an int to smaller type does not cost extra instructions on a store.

The ARMv4 architecture and above support signed 8-bit and 16-bit loads and stores directly, through new instructions. Since these instructions are a later addition, they do not support as many addressing modes as the pre-ARMv4 instructions.

Finally, ARMv5 adds instruction support for 64-bit load and stores. This is available in ARM9E and later cores.

Prior to ARMv4, ARM processors were not good at handling signed 8-bit or any 16-bit values. Therefore ARM C compilers define char to be an unsigned 8-bit value, rather than a signed 8-bit value as is typical in many other compilers.

Compilers armcc and gcc use the datatype mappings in Table 5.2 for an ARM target. The exceptional case for type char is worth noting as it can cause problems when you are porting code from another processor architecture. A common example is using a char type variable i as a loop counter, with loop continuation condition i 0. As i is unsigned for the ARM compilers, the loop will never terminate. Fortunately armcc produces a warning in this situation: unsigned comparison with 0. Compilers also provide an override switch to make char signed. For example, the command line option -fsigned-char will make char signed on gcc. The command line option -zc will have the same effect with armcc.

C compiler datatype mappings.

| C Data Type | Implementation |
|---|---|
| char | unsigned 8-bit byte |
| short | signed 16-bit halfword |
| int | signed 32-bit word |
| long | signed 32-bit word |
| long long | signed 64-bit double word |

## FUNCTION ARGUMENT TYPES

local variables from types char or short to type int increases performance and reduces code size. The same holds for function arguments. Consider the following simple function, which adds two 16-bit values, halving the second, and returns a 16-bit sum:

```
short add_v1(short a, short b)
{
  return a + (b>>1);
}
```

This function is a little artificial, but it is a useful test case to illustrate the problems faced by the compiler. The input values a, b, and the return value will be passed in 32-bit ARM registers. Should the compiler assume that these 32-bit values are in the range of a short type, that is, 32,768 to 32,767? Or should the compiler force values to be in this range by sign-extending the lowest 16 bits to fill the 32-bit register? The compiler must make compatible decisions for the function caller and callee. Either the caller or callee must perform the cast to a short type.

We say that function arguments are passed wide if they are not reduced to the range of the type and narrow if they are. You can tell which decision the compiler has made by looking at the assembly output for add_v1. If the compiler passes arguments wide, then the callee must reduce function arguments to the correct range. If the compiler passes arguments narrow, then the caller must reduce the range. If the compiler returns values wide, then the caller must reduce the return value to the correct range. If the compiler returns values narrow, then the callee must reduce the range before returning the value.

For armcc in ADS, function arguments are passed narrow and values returned narrow. In other words, the caller casts argument values and the callee casts return values. The compiler uses the ANSI prototype of the function to determine the datatypes of the function arguments.

The armcc output for add_v1 shows that the compiler casts the return value to a short type, but does not cast the input values. It assumes that the caller has already ensured that the 32-bit values r0 and r1 are in the range of the short type. This shows narrow passing of arguments and return value.

```
add_v1
        ADD     r0,r0,r1,ASR #1     ; r0 = (int)a + ((int)b>>1)
        MOV     r0,r0,LSL #16
        MOV     r0,r0,ASR #16       ; r0 = (short)r0
        MOV     pc,r14              ; return r0
```

The gcc compiler we used is more cautious and makes no assumptions about the range of argument value. This version of the compiler reduces the input arguments to the range

of a short in both the caller and the callee. It also casts the return value to a short type. Here is the compiled code for add_v1:

```
add_v1_gcc
        MOV     r0, r0, LSL #16
        MOV     r1, r1, LSL #16
        MOV     r1, r1, ASR #17         ; r1 = (int)b>>1
        ADD     r1, r1, r0, ASR #16     ; r1 += (int)a
        MOV     r1, r1, LSL #16
        MOV     r0, r1, ASR #16         ; r0 = (short)r1
        MOV     pc, lr                  ; return r0
```

Whatever the merits of different narrow and wide calling protocols, you can see that char or short type function arguments and return values introduce extra casts. These increase code size and decrease performance. It is more efficient to use the int type for function arguments and return values, even if you are only passing an 8-bit value.

## C LOOPING STRUCTURES

This section looks at the most efficient ways to code for and while loops on the ARM. We start by looking at loops with a fixed number of iterations and then move on to loops with a variable number of iterations. Finally we look at loop unrolling.

### *LOOPS WITH A FIXED NUMBER OF ITERATIONS*

What is the most efficient way to write a for loop on the ARM? Let's return to our checksum example and look at the looping structure.

The first point to note about the procedure call standard is the four-register rule. Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments. For functions with four or fewer arguments, the compiler can pass all the arguments in registers. For functions with more arguments, both the caller and callee must access the stack for some arguments. Note that for C++ the first argument to an object method is the this pointer. This argument is implicit and additional to the explicit arguments.

```
int checksum_v5(int *data)
{
  unsigned int i;
  int sum=0;

  for (i=0; i<64; i++)
  {
    sum += *(data++);
  }
  return sum;
}
```

This compiles to

```
checksum_v5
        MOV     r2,r0           ; r2 = data
        MOV     r0,#0           ; sum = 0
        MOV     r1,#0           ; i = 0
checksum_v5_loop
        LDR     r3,[r2],#4      ; r3 = *(data++)
        ADD     r1,r1,#1        ; i++
        CMP     r1,#0x40        ; compare i, 64
        ADD     r0,r3,r0        ; sum += r3
        BCC     checksum_v5_loop ; if (i<64) goto loop
        MOV     pc,r14          ; return sum
```

It takes three instructions to implement the `for` loop structure:

- An ADD to increment i
- A compare to check if i is less than 64
- A conditional branch to continue the loop if $i < 64$

This is not efficient. On the ARM, a loop should only use two instructions:

- A subtract to decrement the loop counter, which also sets the condition code flags on the result
- A conditional branch instruction

# Function Call:

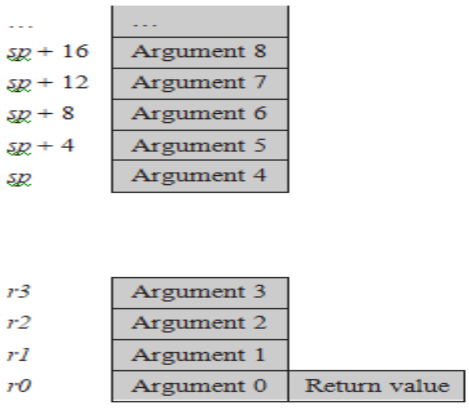
The ARM Procedure Call Standard (APCS) defines how to pass function arguments and return values in ARM registers. The more recent ARM-Thumb Procedure Call Standard (ATPCS) covers ARM and Thumb interworking as well.

The first four integer arguments are passed in the first four ARM registers: r0, r1, r2, and r3. Subsequent integer arguments are placed on the full descending stack, ascending in memory Function return integer values are passed in r0.

This description covers only integer or pointer arguments. Two-word arguments such as long long or double are passed in a pair of consecutive argument registers and returned in r0, r1. The compiler may pass structures in registers or by reference according to command line compiler options.

The first point to note about the procedure call standard is the four-register rule. Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments. For functions with four or fewer arguments, the compiler can pass all the arguments in registers. For functions with more arguments, both the caller and callee must access the stack for some arguments. Note that for C++ the first argument to an object method is the this pointer. This argument is implicit and additional to the explicit arguments.

If your C function needs more than four arguments, or your C++ method more than three explicit arguments, then it is almost always more efficient to use structures. Group related arguments into structures, and pass a structure pointer rather than mul- tiple arguments. Which arguments are related will depend on the structure of your software.




ATPCS argument passing.

The next example illustrates the benefits of using a structure pointer. First we show a typical routine to insert N bytes from array data into a queue. We implement the queue using a cyclic buffer with start address Q_start (inclusive) and end address Q_end (exclusive).

## Pointer Aliasing

Two pointers are said to alias when they point to the same address. If you write to one pointer, it will affect the value you read from the other pointer. In a function, the compiler often doesn't know which pointers can alias and which pointers can't. The compiler must be very pessimistic and assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency.

Let's start with a very simple example. The following function increments two timer values by a step amount:

```
void timers_v1(int *timer1, int *timer2, int *step)
{
  *timer1 += *step;
  *timer2 += *step;
}
```

This compiles to

```
timers_v1
        LDR     r3,[r0,#0]          ; r3 = *timer1
        LDR     r12,[r2,#0]         ; r12 = *step
        ADD     r3,r3,r12           ; r3 += r12
        STR     r3,[r0,#0]          ; *timer1 = r3
        LDR     r0,[r1,#0]          ; r0 = *timer2
        LDR     r2,[r2,#0]          ; r2 = *step
        ADD     r0,r0,r2            ; r0 += r2
        STR     r0,[r1,#0]          ; *timer2 = t0
        MOV     pc,r14              ; return
```

## STRUCTURE ARRANGEMENT

The way you lay out a frequently used structure can have a significant impact on its performance and code density. There are two issues concerning structures on the ARM: alignment of the structure entries and the overall size of the structure.

For architectures up to and including ARMv5TE, load and store instructions are only guaranteed to load and store values with address aligned to the size of the access width. Table 5.4 summarizes these restrictions.

For this reason, ARM compilers will automatically align the start address of a structure to a multiple of the largest access width used within the structure (usually four or eight bytes) and align entries within structures to their access width by inserting padding.

For example, consider the structure

```
struct {

char a;

int b;

char c;

short d;

}
```

For a little-endian memory system the compiler will lay this out adding padding to ensure that the next object is aligned to the size of that object:

For a little-endian memory system the compiler will lay this out adding padding to ensure that the next object is aligned to the size of that object:

| Address | +3 | +2 | +1 | +0 |
|---|---|---|---|---|
| +0 | pad | pad | pad | a |
| +4 | b[31,24] | b[23,16] | b[15,8] | b[7,0] |
| +8 | d[15,8] | d[7,0] | pad | c |

Load and store alignment restrictions for ARMv5TE.

| Transfer size | Instruction | Byte address |
|---|---|---|
| 1 byte | LDRB, LDRSB, STRB | any byte address alignment |
| 2 bytes | LDRH, LDRSH, STRH | multiple of 2 bytes |
| 4 bytes | LDR, STR | multiple of 4 bytes |
| 8 bytes | LDRD, STRD | multiple of 8 bytes |

## Floating Point

The majority of ARM processor implementations do not provide hardware floating-point support, which saves on power and area when using ARM in a price-sensitive, embedded application. With the exceptions of the Floating Point Accelerator (FPA) used on the ARM7500FE and the Vector Floating Point accelerator (VFP) hardware, the C compiler must provide support for floating point in software.

In practice, this means that the C compiler converts every floating-point operation into a subroutine call. The C library contains subroutines to simulate floating-point behavior using integer arithmetic. This code is written in highly optimized assembly. Even so, floating-point algorithms will execute far more slowly than corresponding integer algorithms.

If you need fast execution and fractional values, you should use fixed-point or block-floating algorithms. Fractional values are most often used when processing digital signals such as audio and video. This is a large and important area of programming, For best performance you need to code the algorithms in assembly

## Instruction Scheduling

The time taken to execute instructions depends on the implementation pipeline. For this chapter, we assume ARM9TDMI pipeline timings.

The following rules summarize the cycle timings for common instruction classes on the ARM9TDMI.

Instructions that are conditional on the value of the ARM condition codes in the cpsr take one cycle if the condition is not met. If the condition is met, then the following rules apply:

- ALU operations such as addition, subtraction, and logical operations take one cycle. This includes a shift by an immediate value. If you use a register-specified shift, then add one cycle. If the instruction writes to the pc, then add two cycles.
- Load instructions that load N 32-bit words of memory such as LDR and LDM take N cycles to issue, but the result of the last word loaded is not available on the following cycle. The updated load address is available on the next cycle. This assumes zero-wait-state memory for an uncached system, or a cache hit for a cached system. An LDM of a single value is exceptional, taking two cycles. If the instruction loads pc, then add two cycles.
- Load instructions that load 16-bit or 8-bit data such as LDRB, LDRSB, LDRH, and LDRSH take one cycle to issue. The load result is not available on the following two cycles. The updated load address is available on the next cycle. This assumes zero-wait-state memory for an uncached system, or a cache hit for a cached system.
- Branch instructions take three cycles.
- Store instructions that store N values take N cycles. This assumes zero-wait-state memory for an uncached system, or a cache hit or a write buffer with N free entries for a cached system. An STM of a single value is exceptional, taking two cycles.

- Multiply instructions take a varying number of cycles depending on the value of the second operand in the product (see Table D.6 in Section D.3).

To understand how to schedule code efficiently on the ARM, we need to understand the ARM pipeline and dependencies. The ARM9TDMI processor performs five operations in parallel:

- Fetch: Fetch from memory the instruction at address pc. The instruction is loaded into the core and then processes down the core pipeline.
- Decode: Decode the instruction that was fetched in the previous cycle. The processor also reads the input operands from the register bank if they are not available via one of the forwarding paths.
- ALU: Executes the instruction that was decoded in the previous cycle. Note this instruction was originally fetched from address pc 8 (ARM state) or pc 4 (Thumb state). Normally this involves calculating the answer for a data processing operation, or the address for a load, store, or branch operation. Some instructions may spend several cycles in this stage. For example, multiply and register-controlled shift operations take several ALU cycles.


ARM9TDMI pipeline executing in ARM state.

- *LS1:* Load or store the data specified by a load or store instruction. If the instruction is not a load or store, then this stage has no effect.
- *LS2:* Extract and zero- or sign-extend the data loaded by a byte or halfword load instruction. If the instruction is not a load of an 8-bit byte or 16-bit halfword item, then this stage has no effect.

Figure shows a simplified functional view of the five-stage ARM9TDMI pipeline. Note that multiply and register shift operations are not shown in the figure.

After an instruction has completed the five stages of the pipeline, the core writes the result to the register file. Note that *pc* points to the address of the instruction being fetched. The ALU is executing the instruction that was originally fetched from address $pc - 8$ in parallel with fetching the instruction at address *pc*.

How does the pipeline affect the timing of instructions? Consider the following examples. These examples show how the cycle timings change because an earlier instruction must complete a stage before the current instruction can progress down the pipeline. To work out how many cycles a block of code will take, use the tables in Appendix D that summarize the cycle timings and interlock cycles for a range of ARM cores.

If an instruction requires the result of a previous instruction that is not available, then the processor stalls. This is called a pipeline *hazard* or pipeline *interlock*.

## REGISTER ALLOCATION

You can use 14 of the 16 visible ARM registers to hold general-purpose data. The other two registers are the stack pointer *r13* and the program counter *r15*. For a function to be ATPCS compliant it must preserve the callee values of registers *r4* to *r11*. ATPCS also specifies that the stack should be eight-byte aligned; therefore you must preserve this alignment if calling subroutines. Use the following template for optimized assembly routines requiring many registers:

```
routine_name
        STMFD sp!,       {r4-r12, lr}      ; stack saved registers
          ; body of routine
          ; the fourteen registers r0-r12 and lr are available
        LDMFD sp!,       {r4-r12, pc}      ; restore registers and return
```

Our only purpose in stacking *r12* is to keep the stack eight-byte aligned. You need not stack *r12* if your routine doesn't call other ATPCS routines. For ARMv5 and above you can use the preceding template even when being called from Thumb code. If your routine may be called from Thumb code on an ARMv4T processor, then modify the template as follows:

```
routine_name
       STMFD sp!,      {r4-r12, lr}      ; stack saved registers
         ; body of routine
         ; registers r0-r12 and lr available
       LDMFD sp!,      {r4-r12, lr}      ; restore registers
       BX         lr                     ; return, with mode switch
```

In this section we look at how best to allocate variables to register numbers for register-intensive tasks, how to use more than 14 local variables, and how to make the best use of the 14 available registers.

## Conditional Execution

The processor core can conditionally execute most ARM instructions. This conditional execution is based on one of 15 condition codes. If you don't specify a condition, the

assembler defaults to the execute always condition (AL). The other 14 conditions split into seven pairs of complements. The conditions depend on the four condition code flags N, Z, C, V stored in the cpsr register. See Table A.2 in Appendix A for the list of possible ARM conditions.

By default, ARM instructions do not update the N, Z, C, V flags in the ARM cpsr. For most instructions, to update these flags you append an S suffix to the instruction mnemonic. Exceptions to this are comparison instructions that do not write to a destination register. Their sole purpose is to update the flags and so they don't require the S suffix.

By combining conditional execution and conditional setting of the flags, you can implement simple if statements without any need for branches. This improves efficiency since branches can take many cycles and also reduces code size.

## Looping Constructs

Most routines critical to performance will contain a loop. We saw in Section 5.3 that on the ARM loops are fastest when they count down towards zero. This section describes how to implement these loops efficiently in assembly. We also look at examples of how to unroll loops for maximum performance.

## Decremented Counted Loops

For a decrementing loop of $N$ iterations, the loop counter $i$ counts down from $N$ to 1 inclusive. The loop terminates with $i = 0$. An efficient implementation is

```
      MOV i, N
loop
      ; loop body goes here and i=N,N-1,...,1
      SUBS i, i, #1
      BGT  loop
```

The loop overhead consists of a subtraction setting the condition codes followed by a conditional branch. On ARM7 and ARM9 this overhead costs four cycles per loop. If $i$ is an array index, then you may want to count down from $N-1$ to 0 inclusive instead so that you can access array element zero. You can implement this in the same way by using a different conditional branch:

In this arrangement the *Z* flag is set on the last iteration of the loop and cleared for other iterations. If there is anything different about the last loop, then we can achieve this using the EQ and NE conditions. For example, if you preload data for the next loop (as discussed in Section 6.3.1.1), then you want to avoid the preload on the last loop. You can make all preload operations conditional on NE as in Section 6.3.1.1.

There is no reason why we must decrement by one on each loop. Suppose we require *N*/3 loops. Rather than attempting to divide *N* by three, it is far more efficient to subtract three from the loop counter on each iteration:

```
      MOV 1, N
loop
      ; loop body goes here and iterates (round up)(N/3) times
      SUBS 1, 1, #3
      BGT  loop
```

## Unrolled Counted Loops

This brings us to the subject of loop unrolling. Loop unrolling reduces the loop overhead by executing the loop body multiple times. However, there are problems to overcome. What if the loop count is not a multiple of the unroll amount? What if the loop count is smaller than the unroll amount? We looked at these questions for C code in Section 5.3. In this section we look at how you can handle these issues in assembly.

We'll take the C library function `memset` as a case study. This function sets *N* bytes of memory at address *s* to the byte value *c*. The function needs to be efficient, so we will look at how to unroll the loop without placing extra restrictions on the input operands. Our version of `memset` will have the following C prototype:

```
void my_memset(char *s, int c, unsigned int N);
```

To be efficient for large *N*, we need to write multiple bytes at a time using STR or STM instructions. Therefore our first task is to align the array pointer *s*. However, it is only worth us doing this if *N* is sufficiently large. We aren't sure yet what "sufficiently large" means, but let's assume we can choose a threshold value $T_1$ and only bother to align the array when $N \geq T_1$. Clearly $T_1 \geq 3$ as there is no point in aligning if we don't have four bytes to write!

Now suppose we have aligned the array *s*. We can use store multiples to set memory efficiently. For example, we can use a loop of four store multiples of eight words each to set 128 bytes on each loop. However, it will only be worth doing this if $N \geq T_2 \geq 128$, where $T_2$ is another threshold to be determined later on.

Finally, we are left with $N < T_2$ bytes to set. We can write bytes in blocks of four using STR until $N < 4$. Then we can finish by writing bytes singly with STRB to the end of the array.

UNIT-5

# Memory Management

## CACHE ARCHITECTURE

ARM uses two bus architectures in its cached cores, the Von Neumann and the Harvard. The Von Neumann and Harvard bus architectures differ in the separation of the instruction and data paths between the core and memory. A different cache design is used to support the two architectures.

In processor cores using the Von Neumann architecture, there is a single cache used for instruction and data. This type of cache is known as a *unified cache*. A unified cache memory contains both instruction and data values.

The Harvard architecture has separate instruction and data buses to improve overall system performance, but supporting the two buses requires two caches. In processor cores using the Harvard architecture, there are two caches: an instruction cache (I-cache) and a data cache (D-cache). This type of cache is known as a *split cache*. In a split cache, instructions are stored in the instruction cache and data values are stored in the data cache.

We introduce the basic architecture of caches by showing a unified cache in Figure
The two main elements of a cache are the cache *controller* and the cache *memory*. The cache memory is a dedicated memory array accessed in units called *cache lines*. The cache controller uses different portions of the address issued by the processor during a memory request to select parts of cache memory. We will present the architecture of the cache memory first and then proceed to the details of the cache controller.

### BASIC ARCHITECTURE OF A CACHE MEMORY

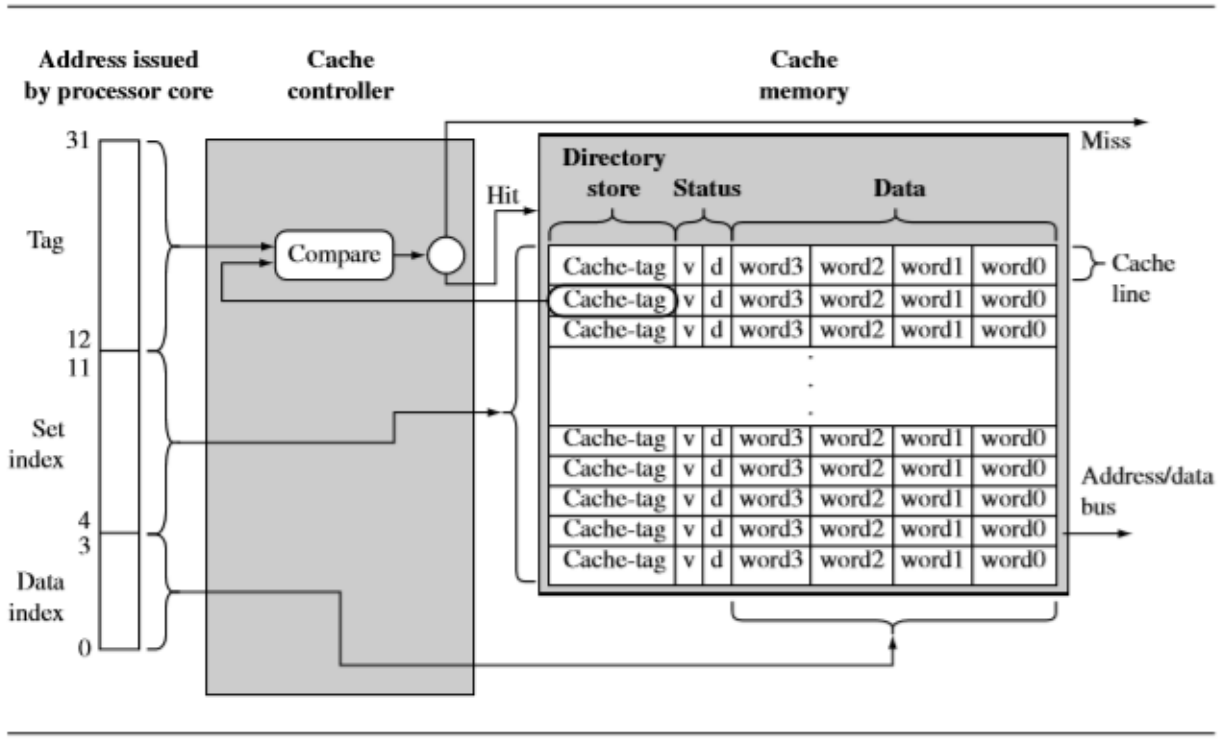
A simple cache memory is shown on the right side of Figure 12.4. It has three main parts: a directory store, a data section, and status information. All three parts of the cache memory are present for each cache line.

The cache must know where the information stored in a cache line originates from in main memory. It uses a directory store to hold the address identifying where the cache line was copied from main memory. The directory entry is known as a *cache-tag*.

A cache memory must also store the data read from main memory. This information is held in the data section (see Figure 12.4).

The size of a cache is defined as the actual code or data the cache can store from main memory. Not included in the cache size is the cache memory required to support cache-tags or status bits.

There are also status bits in cache memory to maintain state information. Two common status bits are the valid bit and dirty bit. A *valid* bit marks a cache line as active, meaning it contains live data originally taken from main memory and is currently available to the




A 4 KB cache consisting of 256 cache lines of four 32-bit words.

## BASIC OPERATION of A CACHE CONTROLLER

The cache controller is hardware that copies code or data from main memory to cache memory automatically. It performs this task automatically to conceal cache operation from the software it supports. Thus, the same application software can run unaltered on systems with and without a cache.

The cache controller intercepts read and write memory requests before passing them on to the memory controller. It processes a request by dividing the address of the request into three fields, the tag field, the set index field, and the data index field. The three bit fields are shown in Figure 12.4.

First, the controller uses the set index portion of the address to locate the cache line within the cache memory that might hold the requested code or data. This cache line contains the cache-tag and status bits, which the controller uses to determine the actual data stored there.

The controller then checks the valid bit to determine if the cache line is active, and compares the cache-tag to the tag field of the requested address. If both the status check and comparison succeed, it is a cache hit. If either the status check or comparison fails, it is a cache miss.

On a cache miss, the controller copies an entire cache line from main memory to cache memory and provides the requested code or data to the processor. The copying of a cache line from main memory to cache memory is known as a cache line fill.

On a cache hit, the controller supplies the code or data directly from cache memory to the processor. To do this it moves to the next step, which is to use the data index field of the address request to select the actual code or data in the cache line and provide it to the processor.

## CACHE POLICY

There are three policies that determine the operation of a cache: the write policy, the replacement policy, and the allocation policy. The cache write policy determines where data is stored during processor write operations. The replacement policy selects the cache line in a set that is used for the next line fill during a cache miss. The allocation policy determines when the cache controller allocates a cache line.

## WRITE POLICY—WRITEBACK OR WRITETHROUGH

When the processor core writes to memory, the cache controller has two alternatives for its write policy. The controller can write to both the cache and main memory, updating the values in both locations; this approach is known as *writethrough*. Alternatively, the cache controller can write to cache memory and not update main memory, this is known as *writeback* or *copyback*.

### 12.3.1.1 Writethrough

When the cache controller uses a writethrough policy, it writes to both cache and main memory when there is a cache hit on write, ensuring that the cache and main memory stay coherent at all times. Under this policy, the cache controller performs a write to main memory for each write to cache memory. Because of the write to main memory, a writethrough policy is slower than a writeback policy.

### 12.3.1.2 Writeback

When a cache controller uses a writeback policy, it writes to valid cache data memory and not to main memory. Consequently, valid cache lines and main memory may contain different data. The cache line holds the most recent data, and main memory contains older data, which has not been updated.

Caches configured as writeback caches must use one or more of the dirty bits in the cache line status information block. When a cache controller in writeback writes a value to cache memory, it sets the dirty bit true. If the core accesses the cache line at a later time, it

## FLUSHING AND CLEANING CACHE MEMORY

ARM uses the terms *flush* and *clean* to describe two basic operations performed on a cache.

To "flush a cache" is to clear it of any stored data. Flushing simply clears the valid bit in the affected cache line. All or just portions of a cache may need flushing to support changes in memory configuration. The term *invalidate* is sometimes used in place of the term *flush*. However, if some portion of the D-cache is configured to use a writeback policy, the data cache may also need cleaning.

To "clean a cache" is to force a write of dirty cache lines from the cache out to main memory and clear the dirty bits in the cache line. Cleaning a cache reestablishes coherence between cached memory and main memory, and only applies to D-caches using a writeback policy.

Coprocessor 15 registers that configure and control cache operation.

| Function | Primary register | Secondary registers | Opcode 2 |
|---|---|---|---|
| Clean and flush cache | *c7* | *c5, c6, c7, c10, c13, c14* | 0, 1, 2 |
| Drain write buffer | *c7* | *c10* | 4 |
| Cache lockdown | *c9* | *c0* | 0, 1 |
| Round-robin replacement | *c15* | *c0* | 0 |

Changing the memory configuration of a system may require cleaning or flushing a cache. The need to clean or flush a cache results directly from actions like changing the access permission, cache, and buffer policy, or remapping virtual addresses.

The cache may also need cleaning or flushing before the execution of self-modifying code in a split cache. Self-modifying code includes a simple copy of code from one location to another. The need to clean or flush arises from two possible conditions: First, the self- modifying code may be held in the D-cache and therefore be unavailable to load from main memory as an instruction. Second, existing instructions in the I-cache may mask new instructions written to main memory.

If a cache is using a writeback policy and self-modifying code is written to main memory, the first step is to write the instructions as a block of data to a location in main memory. At a later time, the program will branch to this memory and begin executing from that area of memory as an instruction stream. During the first write of code to main memory as data, it may be written to cache memory instead; this occurs in an ARM cache if valid cache lines exist in cache memory representing the location where the self-modifying code is written. The cache lines are copied to

the D-cache and not to main memory. If this is the case, then when the program branches to the location where the self-modifying code should be, it will execute old instructions still present because the self-modifying code is still in the D-cache. To prevent this, clean the cache, which forces the instructions stored as data into main memory, where they can be read as an instruction stream.

If the D-cache has been cleaned, new instructions are present in main memory. However, the I-cache may have valid cache lines stored for the addresses where the new data (code) was written. Consequently, a fetch of the instruction at the address of the copied code would retrieve the old code from the I-cache and not the new code from main memory. Flush the I-cache to prevent this from happening.

DETAILS of THE ARM MMU

The ARM MMU performs several tasks: It translates virtual addresses into physical addresses, it controls memory access permission, and it determines the individual behav- ior of the cache and write buffer for each page in memory. When the MMU is disabled, all virtual addresses map one-to-one to the same physical address. If the MMU is unable to translate an address, it generates an abort exception. The MMU will only abort on translation, permission, and domain faults.

The main software configuration and control components in the MMU are

- Page tables
- The Translation Lookaside Buffer (TLB)
- Domains and access permission
- Caches and write buffer
- The CP15:c1 control register
- The Fast Context Switch Extension

## Memory Management Unit (MMU)

When creating a multitasking embedded system, it makes sense to have an easy way to write, load, and run independent application tasks. Many of today's embedded systems use an operating system instead of a custom proprietary control system to simplify this process. More advanced operating systems use a hardware-based memory management unit (MMU).

One of the key services provided by an MMU is the ability to manage tasks as indepen- dent programs running in their own private memory space. A task written to run under the control of an operating system with an MMU does not need to know the memory requirements of unrelated tasks. This simplifies the design requirements of individual tasks running under the control of an operating system.

The processor cores with memory protection units. These cores have a single addressable physical memory space. The addresses generated by the processor core while running a task are used directly to access main memory, which makes it impossible for two programs to reside in main memory at the same time if they are compiled using addresses that overlap. This makes running several tasks in an embedded system difficult because each task must run in a distinct address block in main memory.

The MMU simplifies the programming of application tasks because it provides the resources needed to enable virtual memory—an additional memory space that is indepen- dent of the physical memory attached to the system. The MMU acts as a translator, which converts the addresses of programs and data that are compiled to run in virtual memory to the actual physical addresses where the programs are stored in physical main memory. This translation process allows programs to run with the same virtual addresses while being held in different locations in physical memory.

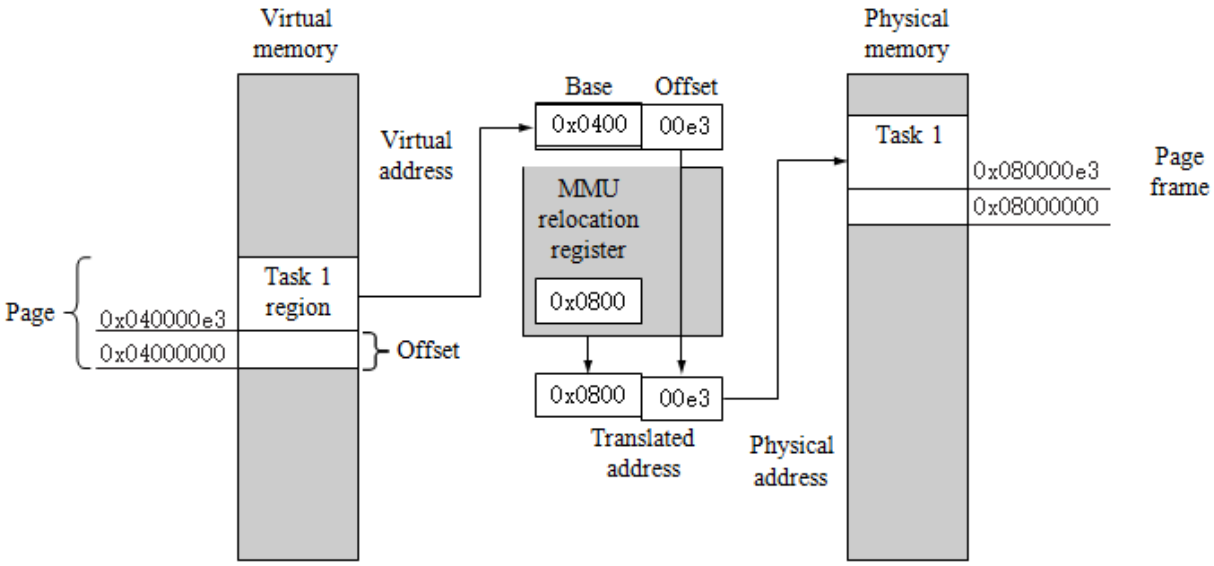
This dual view of memory results in two distinct address types: virtual addresses and physical addresses. Virtual addresses are assigned by the compiler and linker when locating a program in memory. Physical addresses are used to access the actual hardware components of main memory where the programs are physically located.

ARM provides several processor cores with integral MMU hardware that efficiently support multitasking environments using virtual memory. The goal of this chapter is to learn the basics of ARM memory management units and some basic concepts that underlie the use of virtual memory.

## Virtual Memory Works

In an MMU, tasks can run even if they are compiled and linked to run in regions with overlapping addresses in main memory. The support for virtual memory in the MMU enables the construction of an embedded system that has multiple virtual memory maps and a single physical memory map. Each task is provided its own virtual memory map for the purpose of compiling and linking the code and data, which make up the task. A kernel layer then manages the placement of the multiple tasks in physical memory so they have a distinct location in physical memory that is different from the virtual location it is designed to run in.

To permit tasks to have their own virtual memory map, the MMU hardware performs address relocation, translating the memory address output by the processor core before it reaches main memory. The easiest way to understand the translation process is to imagine a relocation register located in the MMU between the core and main memory.




When the processor core generates a virtual address, the MMU takes the upper bits of the virtual address and replaces them with the contents of the relocation register to create a physical address, shown in above Figure.

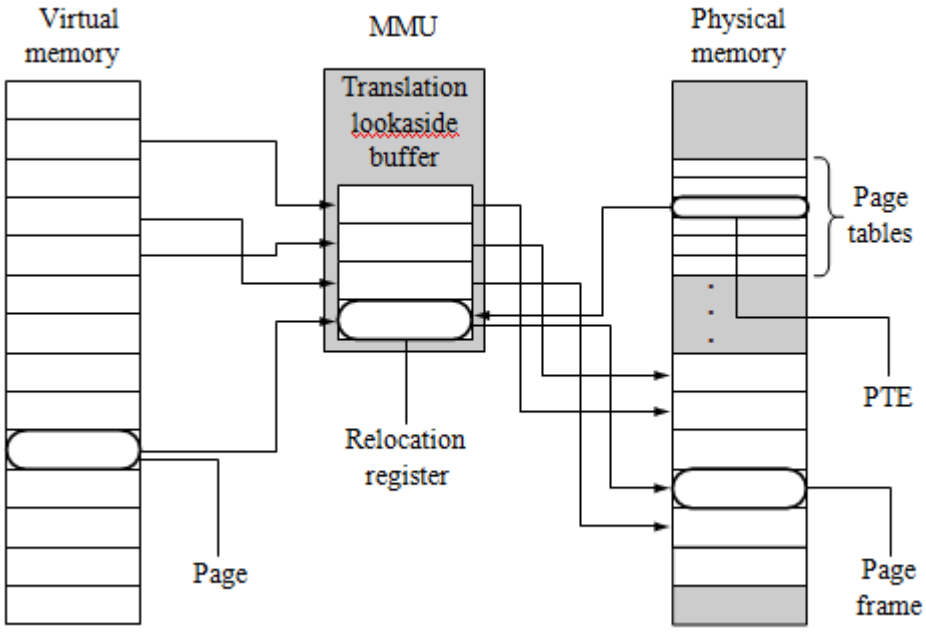
The lower portion of the virtual address is an offset that translates to a specific address in physical memory. The range of addresses that can be translated using this method is limited by the maximum size of this offset portion of the virtual address.

The above Figure shows an example of a task compiled to run at a starting address of 0x4000000 in virtual memory. The relocation register translates the virtual addresses of Task 1 to physical addresses starting at 0x8000000.

A second task compiled to run at the same virtual address, in this case 0x400000, can be placed in physical memory at any other multiple of 0x10000 (64 KB) and mapped to 0x400000 simply by changing the value in the relocation register.

A single relocation register can only translate a single area of memory, which is set by the number of bits in the offset portion of the virtual address. This area of virtual memory is known as a page. The area of physical memory pointed to by the translation process is known as a page frame.

The relationship between pages, the MMU, and page frames shows in below figure. The ARM MMU hardware has multiple relocation registers supporting the translation of virtual memory to physical memory. The MMU needs many relocation registers to effectively support virtual memory because the system must translate many pages to many page frames.

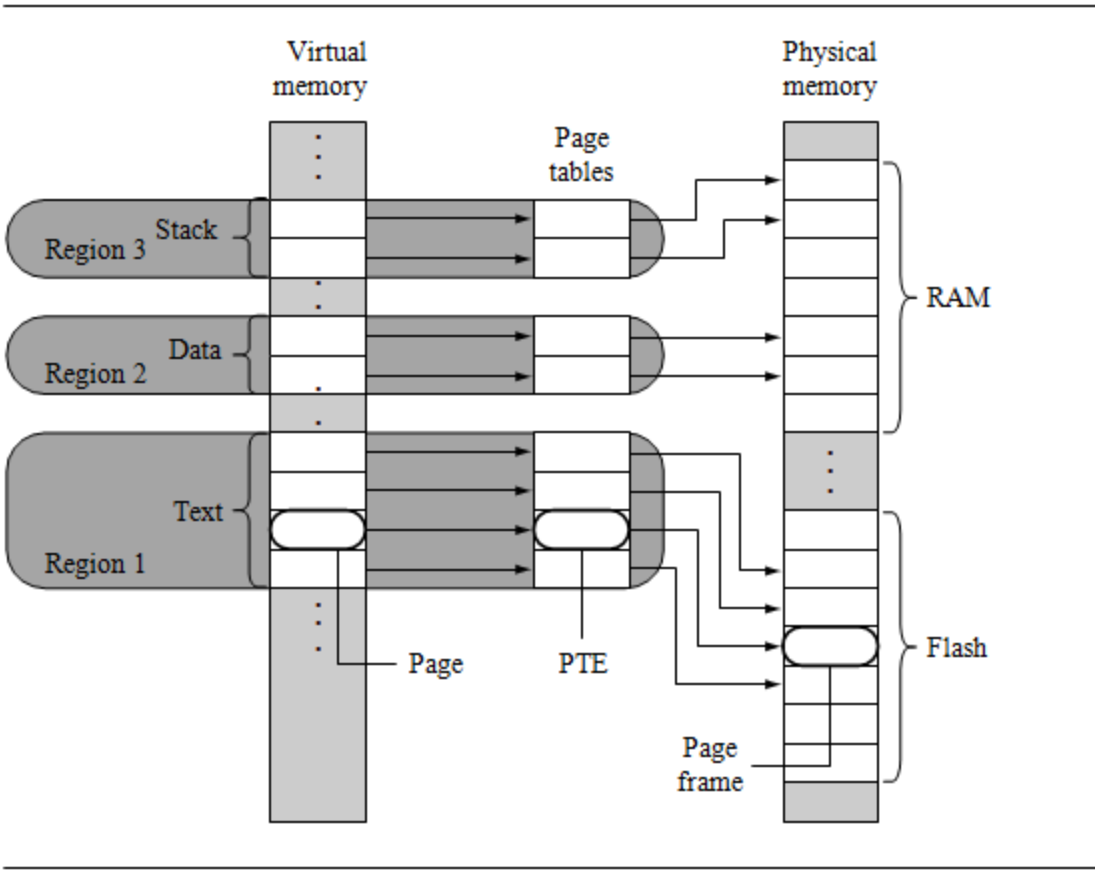



## Regions Using Pages

virtual memory has a corresponding entry in a page table, a block of virtual memory pages map to a set of sequential entries in a page table. Thus, a region can be defined as a sequential set of page table entries. The location and size of a region can be held in a software data structure while the actual translation data and attribute information is held in the page tables.

An example of a single task that has three regions: one for text, one for data, and a third to support the task stack. Each region in virtual memory is mapped to different areas in physical memory. In the figure, the executable code is located in flash memory, and the data and stack areas are located in RAM. This use of regions is typical of operating systems that support sharing code between tasks.

With the exception of the master level 1 (L1) page table, all page tables represent 1 MB areas of virtual memory. If a region's size is greater than 1 MB or crosses over the 1 MB boundary addresses that separate page tables, then the description of a region must also include a list of page tables. The page tables for a region will always be derived from sequential page table entries in the master L1 page table. However, the locations of the L2 page tables in physical memory do not need to be located sequentially.

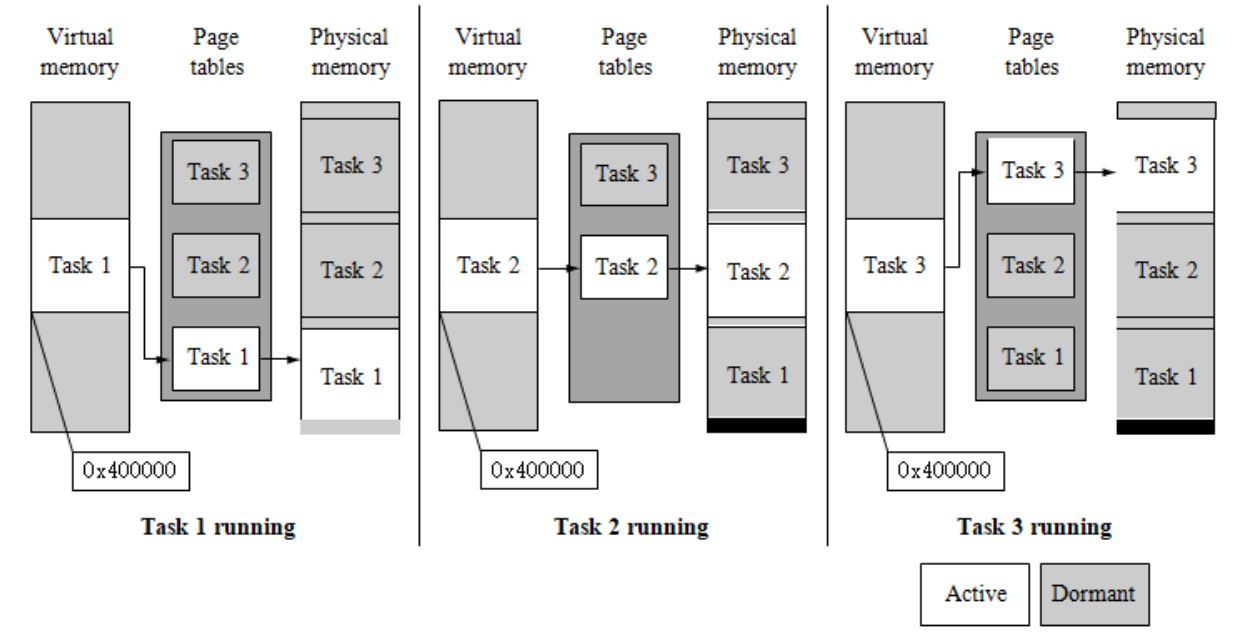



## Multitasking and the MMU

Page tables can reside in memory and not be mapped to MMU hardware. One way to build a multitasking system is to create separate sets of page tables, each mapping a unique virtual memory space for a task. To activate a task, the set of page tables for the specific task and its virtual memory space are mapped into use by the MMU. The other sets of inactive page tables represent dormant tasks. This approach allows all tasks to remain resident in physical memory and still be available immediately when a context switch occurs to activate it.

By activating different page tables during a context switch, it is possible to execute multiple tasks with overlapping virtual addresses. The MMU can relocate the execution address of a task without the need to move it in physical memory. The task's physical memory is simply mapped into virtual memory by activating and deactivating page tables. Figure 14.4 shows three views of three tasks with their own sets of page tables running at a common execution virtual address of 0x0400000.

In the first view, Task 1 is running, and Task 2 and Task 3 are dormant. In the second view, Task 2 is running, and Task 1 and Task 3 are dormant. In the third view, Task 3 is running, and Task 1 and Task 2 are dormant. The virtual memory in each of the three views represents memory as seen by the running task. The view of physical memory is the same in all views because it represents the actual state of real physical memory.




To switch between tasks requires the following steps:

- Save the active task context and place the task in a dormant state.
- Flush the caches; possibly clean the D-cache if using a writeback policy.
- Flush the TLB to remove translations for the retiring task.
- Configure the MMU to use new page tables translating the virtual memory execution area to the awakening task's location in physical memory.
- Restore the context of the awakening task.
- Resume execution of the restored task.

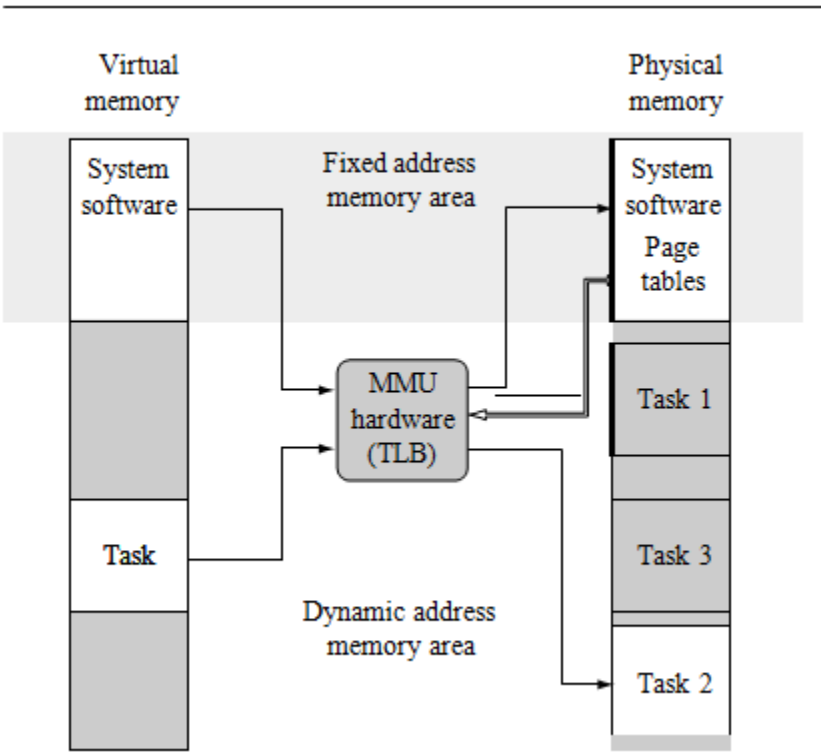
## Memory Organization in a Virtual Memory System

Typically, page tables reside in an area of main memory where the virtual-to-physical address mapping is fixed. By "fixed," we mean data in a page table doesn't change during normal operation, as shown in below Figure. This fixed area of memory also contains the operating system kernel and other processes. The MMU, which includes the TLB shown in Figure 14.5, is hardware that operates outside the virtual or physical memory space; its function is to translate addresses between the two memory spaces.

When a context switch occurs between two application tasks, the processor in reality makes many context switches. It changes from a user mode task to a kernel mode task to perform the actual movement of context data in preparation for running the next applica- tion task. It then changes from the kernel mode task to the new user mode task of the next context.

By sharing the system software in a fixed area of virtual memory that is seen across all user tasks, a system call can branch directly to the system area and not worry about needing to change page tables to map in a kernel process. Making the kernel code and data map to the same virtual address in all tasks eliminates the need to change the memory map and the need to have

an independent kernel process that consumes a time slice.




## Details of the ARM MMU

The ARM MMU performs several tasks: It translates virtual addresses into physical addresses, it controls memory access permission, and it determines the individual behav- ior of the cache and write buffer for each page in memory. When the MMU is disabled, all virtual addresses map one-to-one to the same physical address. If the MMU is unable to translate an address, it generates an abort exception. The MMU will only abort on translation, permission, and domain faults.

The main software configuration and control components in the MMU are

- Page tables
- The Translation Lookaside Buffer (TLB)
- Domains and access permission
- Caches and write buffer
- The CP15:c1 control register
- The Fast Context Switch Extension

## Page Tables

The ARM MMU hardware has a multilevel page table architecture. There are two levels of page table: level 1 (L1) and level 2 (L2).

There is a single level 1 page table known as the L1 master page table that can contain two

types of page table entry. It can hold pointers to the starting address of level 2 page tables, and page table entries for translating 1 MB pages. The L1 master table is also known as a section page table.

The master L1 page table divides the 4 GB address space into 1 MB sections; hence the L1 page table contains 4096 page table entries. The master table is a hybrid table that acts

Table Page tables used by the MMU.

| Name | Type | Memory consumed by page table (KB) | Page sizes supported (KB) | Number of page table entries |
|---|---|---|---|---|
| Master/section | level 1 | 16 | 1024 | 4096 |
| Fine | level 2 | 4 | 1, 4, or 64 | 1024 |
| Coarse | level 2 | 1 | 4 or 64 | 256 |

as both a page directory of L2 page tables and a page table translating 1 MB virtual pages called sections. If the L1 table is acting as a directory, then the PTE contains a pointer to either an L2 coarse or L2 fine page table that represents 1 MB of virtual memory. If the L1 master table is translatinga1 MB section, then the PTE contains the base address of the 1 MB page frame in physical memory. The directory entries and 1 MB section entries can coexist in the master page table.

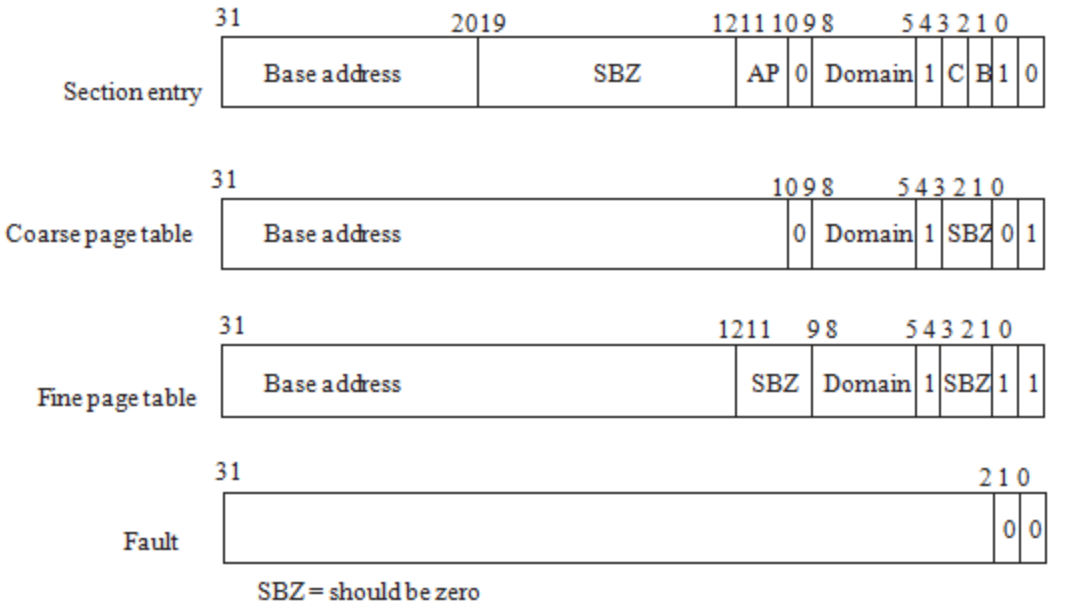
A coarse L2 page table has 256 entries consuming 1 KB of main memory. Each PTE in a coarse page table translatesa4 KB block of virtual memory toa4 KB block in physical memory. A coarse page table supports either 4 or 64 KB pages. The PTE in a coarse page contains the base address to eithera4 or 64 KB page frame; if the entry translates a 64 KB page, an identical PTE must be repeated in the page table 16 times for each 64 KB page.

A fine page table has 1024 entries consuming 4 KB of main memory. Each PTE in a fine page translatesa1 KB block of memory. A fine page table supports 1, 4, or 64 KB pages in virtual memory. These entries contain the base address of a 1, 4, or 64 KB page frame in physical memory. If the fine table translatesa4 KB page, then the same PTE must be repeated 4 consecutive times in the page table. If the table translates a 64 KB page, then the same PTE must be repeated 64 consecutive times in the page table.

## Level 1 Page Table Entries

The level 1 page table accepts four types of entry:

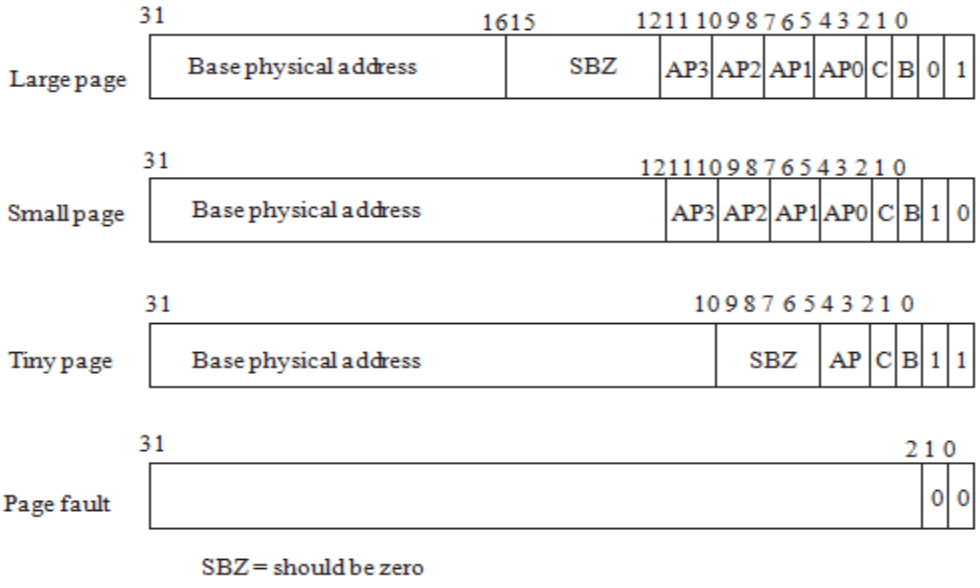
- A 1 MB section translation entry
- A directory entry that points to a fine L2 page table
- A directory entry that points to a coarse L2 page table
- A fault entry that generates an abort exception




## Level 2 Page Table Entries

- There are four possible entries used in L2 page tables:

- A large page entry defines the attributes for a 64 KB page frame.
- A small page entry definesa4 KB page frame.
- A tiny page entry definesa1 KB page frame.
- A fault page entry generates a page fault abort exception when accessed.



# The Translation Lookaside Buffer

The TLB is a special cache of recently used page translations. The TLB maps a virtual page to an active page frame and stores control data restricting access to the page. The TLB is a cache and therefore has a victim pointer and a TLB line replacement policy. In ARM processor cores the TLB uses a round-robin algorithm to select which relocation register to replace on a TLB miss.

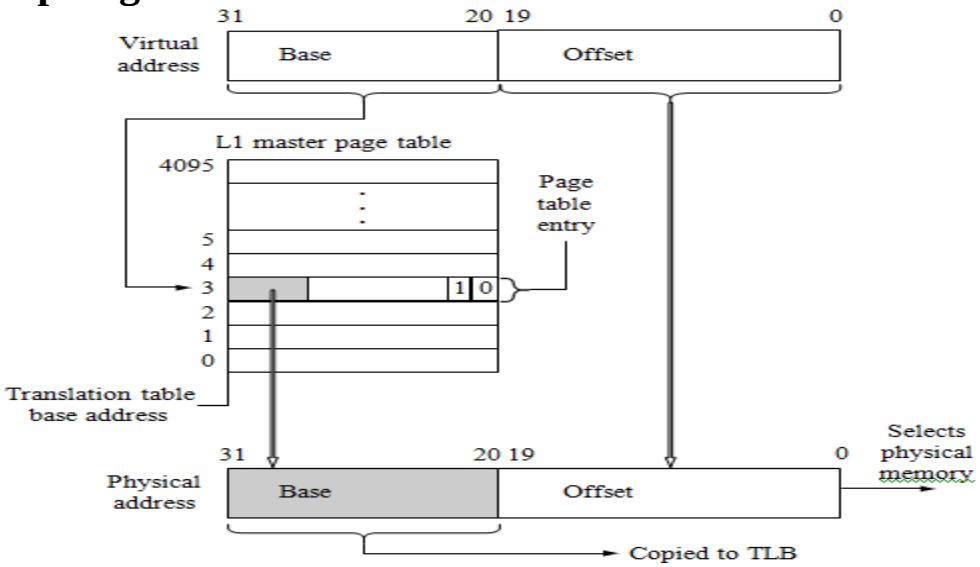
The TLB in ARM processor cores does not have many software commands available to control its operation. The TLB supports two types of commands: you can flush the TLB, and you can lock translations in the TLB.

During a memory access, the MMU compares a portion of the virtual address to all the values cached in the TLB. If the requested translation is available, it is a TLB hit, and the TLB provides the translation of the physical address.

If the TLB does not contain a valid translation, it is a TLB miss. The MMU automatically handles TLB misses in hardware by searching the page tables in main memory for valid translations and loading them into one of the 64 lines in the TLB. The search for valid translations in the page tables is known as a page table walk. If there is a valid PTE, the hardware copies the translation address from the PTE to the TLB and generates the physical address to access main memory. If, at the end of the search, there is a fault entry in the page table, then the MMU hardware generates an abort exception.

During a TLB miss, the MMU may search up to two page tables before loading data to the TLB and generating the needed address translation. The cost of a miss is generally one or two main memory access cycles as the MMU translation table hardware searches the page tables. The number of cycles depends on which page table the translation data is found in. A single-stage page table walk occurs if the search ends with the L1 master page table; there is a two-stage page table walk if the search ends with an L2 page table.

# Single-Step Page Table Walk

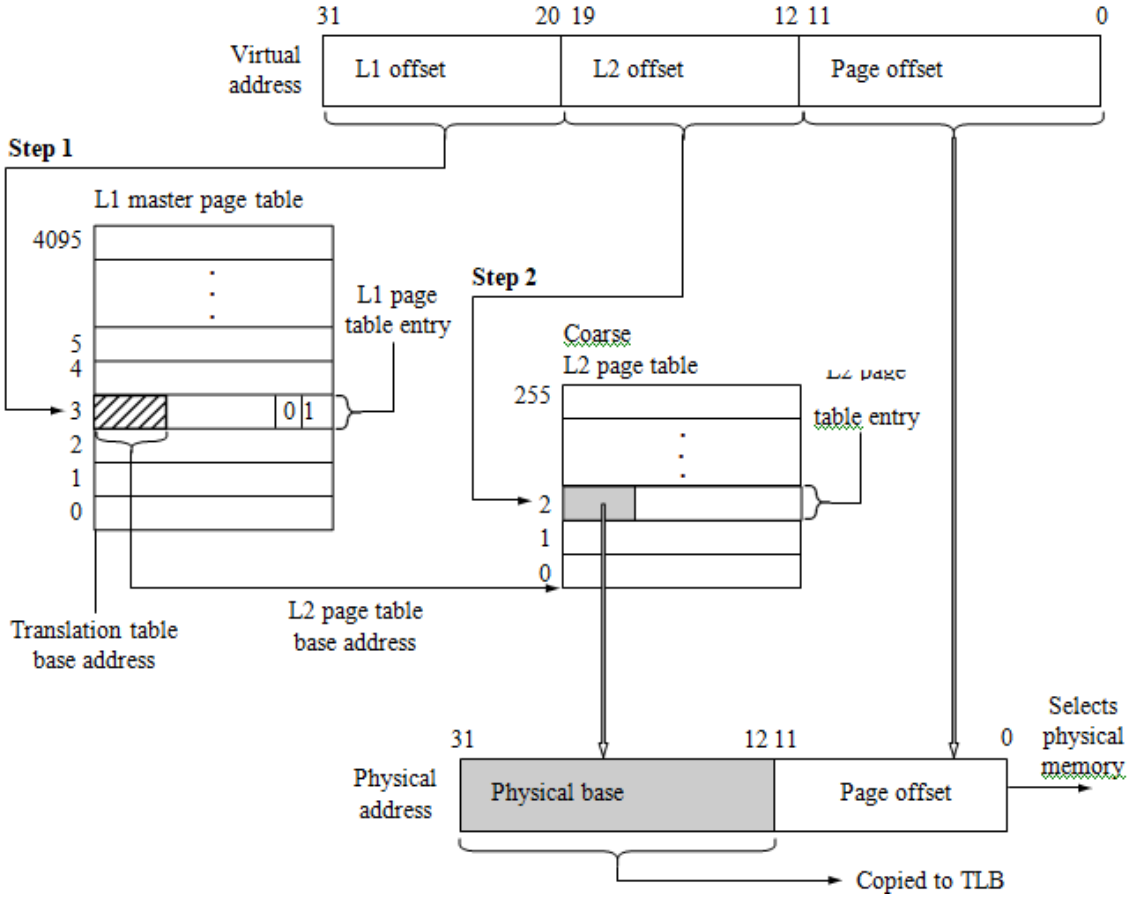


# Two-Step Page Table Walk

If the MMU ends its search for a page that is 1, 4, 16, or 64 KB in size, then the page table walk will have taken two steps to find the address translation. the two-stage process for a translation held in a coarse L2 page table. Note that the virtual address is divided into three parts.

In the first step, the L1 offset portion is used to index into the master L1 page table and find the L1 PTE for the virtual address. If the lower two bits of the PTE contain the binary value 01, then the entry contains the L2 page table base address to a coarse page

In the second step, the L2 offset is combined with the L2 page table base address found in the first stage; the resulting address selects the PTE that contains the translation for the page. The MMU transfers the data in the L2 PTE to the TLB, and the base address is combined with the offset portion of the virtual address to generate the requested address in physical memory.



## Domains and Memory Access Permission

There are two different controls to manage a task's access permission to memory: The primary control is the domain, and a secondary control is the access permission set in the page tables.

Domains control basic access to virtual memory by isolating one area of memory from another when sharing a common virtual memory map. There are 16 different domains that

Commands to access the TLB lockdown registers.

| Command | MCR instruction | Value in *Rd* | Core support |
|---|---|---|---|
| Read D TLB lockdown | MRC p15, 0, Rd, c10, c0, 0 | TLB lockdown | ARM920T, ARM922T, ARM926EJ-S, ARM1022E, ARM1026EJ-S, StrongARM, XScale |
| Write D TLB lockdown | MCR p15, 0, Rd, c10, c0, 0 | TLB lockdown | ARM920T, ARM922T, ARM926EJ-S, ARM1022E, ARM1026EJ-S, StrongARM, XScale |
| Read I TLB lockdown | MRC p15, 0, Rd, c10, c0, 1 | TLB lockdown | ARM920T, ARM922T, ARM926EJ-S, ARM1022E, ARM1026EJ-S, StrongARM, XScale |
| Write I TLB lockdown | MCR p15, 0, Rd, c10, c0, 1 | TLB lockdown | ARM920T, ARM922T,ARM926EJ-S, ARM1022E,ARM1026EJ-S, StrongARM, XScale |

ARM920T, ARM922T, ARM926EJ-S, ARM1022E


ARM1026EJ-S


SBZ = should be zero

can be assigned to 1 MB sections of virtual memory and are assigned to a section by setting the domain bit field in the master L1 PTE (see Figure 14.6).

When a domain is assigned to a section, it must obey the domain access rights assigned to the domain. Domain access rights are assigned in the CP15:c3 register and control the processor core's ability to access sections of virtual memory.

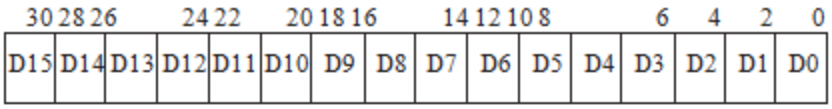
The CP15:c3 register uses two bits for each domain to define the access permitted for each of the 16 available domains. Table 14.5 shows the value and meaning of a domain access bit field. Figure 14.12 gives the format of the CP15:c3:c0 register, which holds the domain access control information. The 16 available domains are labeled from D0 to D15 in the figure.

Even if you don't use the virtual memory capabilities provided by the MMU, you can still use these cores as simple memory protection units: first, by mapping virtual memory directly to

physical memory, assigning a different domain to each task, then using domains to protect dormant tasks by assigning their domain access to "no access."

Domain access bit assignments.

| Access | Bit field value | Comments |
|---|---|---|
| Manager | 11 | access is uncontrolled, no permission aborts generated |
| Reserved | 10 | unpredictable |
| Client | 01 | access controlled by permission values set in PTE |
| No access | 00 | generates a domain fault |




Format of the domain access control register CP15:c3.

Access permission and control bits.

| Privileged mode | User mode | AP bit field | System bit | Rom bit |
|---|---|---|---|---|
| Read and write | read and write | 11 | ignored | ignored |
| Read and write | read only | 10 | ignored | ignored |
| Read and write | no access | 01 | ignored | ignored |
| No access | no access | 00 | 0 | 0 |
| Read only | read only | 00 | 0 | 1 |
| Read only | no access | 00 | 1 | 0 |
| Unpredictable | unpredictable | 00 | 1 | 1 |

## The Fast Context Switch Extension

The Fast Context Switch Extension (FCSE) is additional hardware in the MMU that is considered an enhancement feature, which can improve system performance in an ARM embedded system. The FCSE enables multiple independent tasks to run in a fixed overlap- ping area of memory without the need to clean or flush the cache, or flush the TLB during a context switch. The key feature of the FCSE is the elimination of the need to flush the cache and TLB.

Without the FCSE, switching from one task to the next requires a change in virtual memory maps. If the change involves two tasks with overlapping address ranges, the information stored in the caches and TLB become invalid, and the system must flush the caches and TLB. The process of flushing these components adds considerable time to the task switch because the core must not only clear the caches and TLB of invalid data, but it must also reload data to the caches and TLB from main memory.

With the FCSE there is an additional address translation when managing virtual memory. The FCSE modifies virtual addresses before it reaches the cache and TLB using a special

relocation register that contains a value known as the process ID. ARM refers to the addresses in virtual memory before the first translation as a virtual address (VA), and those addresses after the first translation as a modified virtual address(MVA), shown in Figure 14.4. When using the FCSE, all modified virtual addresses are active. Tasks are protected by using the domain access facilities to block access to dormant tasks. We discuss this in more detail in the next section.

Switching between tasks does not involve changing page tables; it simply requires writing the new task's process ID into the FCSE process ID register located in CP15. Because a task switch does not require changing the page tables, the caches and TLB remain valid after the switch and do not need flushing.

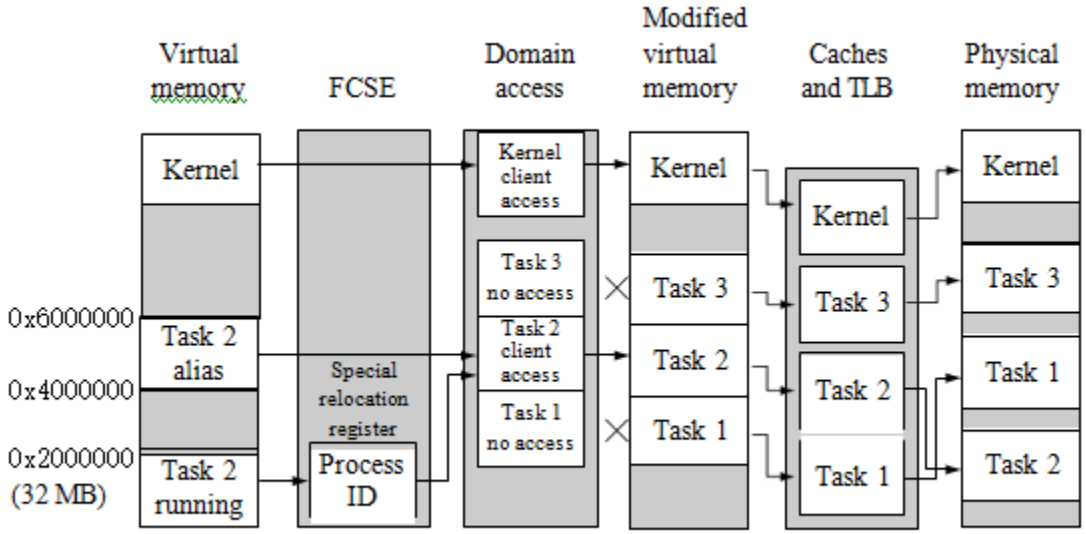
When using the FCSE, each task must execute in the fixed virtual address range from 0x00000000 to 0x1FFFFFFF and must be located in a different 32 MB area of modified virtual memory. The system shares all memory addresses above 0x2000000, and uses domains to protect tasks from each other. The running task is identified by its current process ID.

To utilize the FCSE, compile and link all tasks to run in the first 32 MB block of virtual memory (VA) and assign a unique process ID. Then place each task in a different 32 MB partition of modified virtual memory using the following relocation formula:

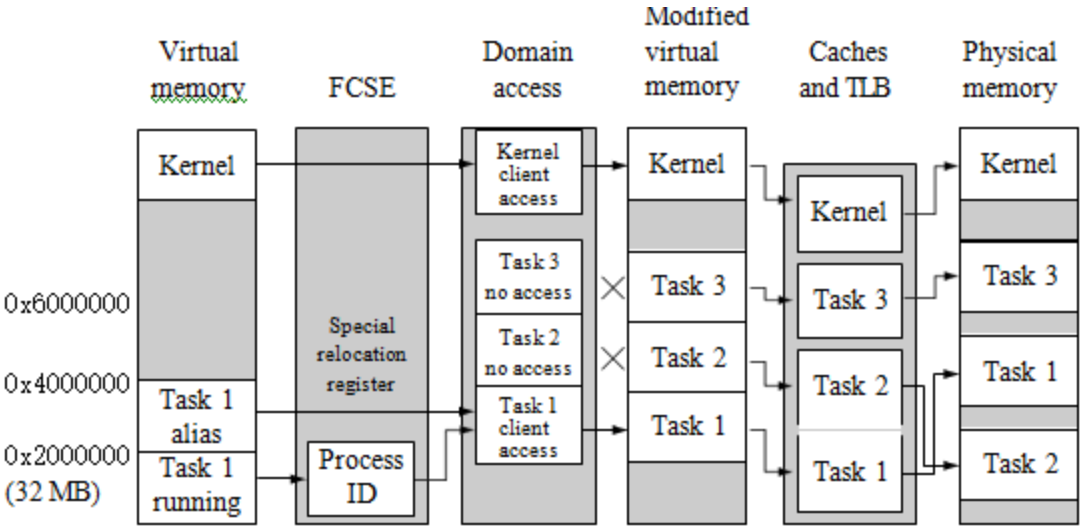
$$MVA = VA + (0x2000000 * \text{process ID})$$

To calculate the starting address of a task partition in modified virtual memory, take a value of zero for the VA and the task's process ID, and use these values in Equation.

The value held in the CP15:c13:c0 register contains the current process ID. The process ID bit field in the register is seven bits wide and supports 128 process IDs. The format of the register.




**Task 2 running**




**Task 1 running**

**MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY (AUTONOMOUS)**

**M. Tech – (VLSI & Embedded Systems)**

**(R18D6803) EMBEDDED SYSTEM DESIGN**

**Internal Marks Assessment**

| S.No. | ROLL NO. | Name of the Student | MID-I | MID-II |
|---|---|---|---|---|
| 1 | 19N31D6801 | ADEPU HARI PRIYA | 26 | 12 |
| 2 | 19N31D6802 | ALGAPALLY SANTOSH SAGAR | 27 | 27 |
| 3 | 19N31D6803 | BADDAM SINDHU REDDY | 12 | 26 |
| 4 | 19N31D6804 | BANGARIGALLA SHEKARBABU | 12 | 24 |
| 5 | 19N31D6805 | CHIPPA PRITHVI RAJ | 21 | 23 |
| 6 | 19N31D6806 | DANAVATH NAGAMMA | 26 | 23 |
| 7 | 19N31D6807 | DEETI ADIKYA | 26 | 27 |
| 8 | 19N31D6808 | DOKKU VARA LAKSHMI | 26 | 24 |
| 9 | 19N31D6809 | GADHARI KEERTHANA | 26 | 28 |
| 10 | 19N31D6810 | GEDDADA SANDEEP VARMA | 26 | 26 |
| 11 | 19N31D6811 | KATTA VIDUSHEE KUMARI VISHWA KARMA | 25 | 26 |
| 12 | 19N31D6812 | KODE SASIKALA | 29 | 24 |
| 13 | 19N31D6813 | KUNDETI MAMATHA | 27 | 22 |
| 14 | 19N31D6814 | NAGANABOINA SUMAN | 24 | 26 |
| 15 | 19N31D6815 | POLAPALLI MANASA | 22 | 24 |
| 16 | 19N31D6816 | POLAPALLI SRIKANTH | 26 | 27 |
| 17 | 19N31D6817 | SAIKUMAR KURAKULA | 24 | 23 |

| | | | | |
|---|---|---|---|---|
| 18 | 19N31D6818 | SANDEEP BALLEM | 24 | 24 |
| 19 | 19N31D6819 | SIDDA MANOJ KUMAR REDDY | 20 | 20 |
| 20 | 19N31D6820 | SUNKARI JASNAVI | 28 | 28 |
| 21 | 19N31D6821 | TALLURI SAI PRIYANKA | 19 | 24 |
| 22 | 19N31D6822 | TALLURI VENKATA RESHMA | 12 | 26 |
| 23 | 19N31D6823 | VENKATREDDY GARI PRATHAP REDDY | 21 | 22 |
| 24 | 19N31D6824 | VUTUKURI ANUSHA | 22 | 23 |

**R18**

**Code No: R18D6803**

## MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

**(Autonomous Institution – UGC, Govt. of India)**

**M.Tech I Year - I Semester Regular/Supplementary Examinations, January-2020**
**Embedded System Design(VLSI&ES)**

| Roll No | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|

**Time: 3 hours** **Max. Marks: 70**

**Note:** This question paper Consists of 5 Sections. Answer **FIVE** Questions, Choosing ONE Question from each SECTION and each Question carries 14 marks.

****

**SECTION-I**

| | | |
|---|---|---|
| **1** | a) Discuss about various types of ARM Registers. | **[7M]** |
| | b) Describe about the instruction pipeline. | **[7M]** |

OR

| | | |
|---|---|---|
| **2** | a) Explain about the interrupts and vector table of ARM. | **[7M]** |
| | b) Explain about the architecture revision. | **[7M]** |

**SECTION-II**

| | | |
|---|---|---|
| **3** | a) Explain about the addressing modes of ARM. | **[7M]** |
| | b) With a suitable example, explain about the PSR instructions. | **[7M]** |

OR

| | | |
|---|---|---|
| **4** | Why do we use controllers in embedded systems? Explain the | **[14M]** |

instruction set of ARM programming model-1.

**SECTION-III**

| | | | |
|---|---|---|---|
| **5** | a) | What is the difference between instruction set and thumb instruction set? | **[7M]** |
| | b) | Explain about the Branch instructions and register usage instructions. | **[7M]** |

OR

| | | | |
|---|---|---|---|
| **6** | a) | Discuss about Software Interrupt Instructions | **[6M]** |
| | b) | Explain about Single-Register and Multi Register Load-Store Instructions | **[8M]** |

**SECTION-IV**

| | | | |
|---|---|---|---|
| **7** | a) | Explain about the conditional execution and loops in ARM programming with a suitable example. | **[7M]** |
| | b) | With a suitable example, explain about the assembly code using instruction scheduling in ARM programming. | **[7M]** |

OR

| | | | |
|---|---|---|---|
| **8** | a) | Explain about ARM programming with one example. | **[7M]** |
| | b) | Describe about the integer and floating point with a suitable example. | **[7M]** |

**SECTION-V**

| | | | |
|---|---|---|---|
| **9** | a) | Explain about the Memory management unit and page tables. | **[7M]** |
| | b) | Explain about the cache architecture in memory management. | **[7M]** |

OR

**10** Write a short notes on

(i) Context switch and Register allocation **[7 M]**

(ii) Flushing and cashes **[7 M]**

**********

**R17**

**Code No: R17D9303**

# MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

**(Autonomous Institution – UGC, Govt. of India)**

## M.Tech I-Year - I Semester Regular/Supplementary Examinations, Dec-18/Jan 19

**Embedded System Design**
**(VLSI&ES)**

| Roll No | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|

**Time: 3 hours** **Max. Marks: 70**

**Note:** This question paper Consists of 5 Sections. Answer **FIVE** Questions, Choosing ONE Question from each SECTION and each Question carries 14marks.

*****

## SECTION-I

| | | | |
|---|---|---|---|
| **1** | a. | Describe the complete ARM register set ? | **[7M]** |
| | b. | Describe the conditional flags of ARM processor? | **[7M]** |

OR

| | | | |
|---|---|---|---|
| **2** | a. | Describe the ARM nomenclature and architecture evaluation ? | **[7M]** |
| | b. | Describe the pipelining execution process in ARM ? | **[7M]** |

## SECTION-II

| | | | |
|---|---|---|---|
| **3** | | Describe various addressing modes in ARM ? | **[14M]** |

OR

| | | | |
|---|---|---|---|
| **4** | | Describe load-store instruction in detail ? | **[14M]** |

## SECTION-III

| | | | |
|---|---|---|---|
| **5** | | Explain various thumb data processing instruction ? | **[14M]** |
| | | OR | |
| **6** | | Explain with example single-register and multiple-register load-store instruction? | **[14M]** |

## SECTION-IV

| | | | |
|---|---|---|---|
| **7** | a. | Explain pointer aliasing with an example? | **[7M]** |
| | **b.** | Explain with example conditional execution ? | **[7M]** |
| | | OR | |
| **8** | a. | ARM9TDMI processor performs various operations in parallel explain them in detail? | **[10M]** |
| | **b.** | What is pipeline interlock explain with example **?** | **[4M]** |

## SECTION-V

| | | | |
|---|---|---|---|
| **9** | a. | How is memory organised in MMU? | **[7M]** |
| | b. | Explain access permission in memory management | **[7M]** |
| | | OR | |
| **10** | a. | Explain flush and clean operation in cache? | **[7M]** |
| | b. | What are the main software configuration and control components in MMU? Explain in detail any two? | **[7M]** |

**********

**R15**

**Code No: R15D9303**

# MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

**(Autonomous Institution – UGC, Govt. of India)**

## M.Tech I-Year - I Semester Supplementary Examinations, Dec-18/Jan-19

## Embedded System Design
## (VLSI&ES & SSP)

| Roll No | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|

**Time: 3 hours** **Max. Marks: 75**

**Note:** This question paper Consists of 5 Sections. Answer **FIVE** Questions, Choosing ONE Question from each SECTION and each Question carries 15 marks.

*****

## SECTION-I

**1** a) With a neat sketch discuss ARM programming model. **[15M]**
b) What do you mean by pipelining? Briefly discuss about five stage pipeline in ARM.

OR

**2** Explain how to measure the processor performance of an embedded hardware in detail and explain the major application areas of embedded system. **[15M]**

## SECTION-II

**3** a)Explain Load, store instructions with examples. **[15M]**
b) What is the primary difference between a load/store architecture and a register/memory architecture

OR

**4** a) What are the unique features of the ARM instruction set? Explain **[7M]**
b) Briefly explain the ARM data processing instructions in detail with suitable example. **[8M]**

## SECTION-III

**5** Explain processor modes of ARM7 , also specify different branch instruction used to exchange branch from ARM mode to THUMB mode. **[15M]**

OR

**6** Draw the format of ARM data processing instructions **[15M]**

Explain the various data operations in ARM.

**SECTION-IV**

**7** a) Explain the different features of FPA10. **[15M]**
b) Discuss the coprocessor Register transfer instructions? Why the instruction cannot used for Register transfer of CP15 coprocessor.

OR

**8** Briefly explain the functions, pointers and structures using in ARM C programming **[15M]**

**SECTION-V**

**9** a) With a neat diagram discuss set associate cache and fully associative cache. **[15M]**
b) Elaborate advantages of having embedded memory on chip? How it is useful in increasing the efficiency of the system.

OR

**10** What are the different types of memories used in embedded system design? Explain each with examples. **[15M]**

**********